הוכן ע"י אמיר רובינשטיין

מבני נתונים ומבוא לאלגוריתמים

נושא 2

מבני נתונים בסיסיים
Basic data structures

בתוכנית

פרק 10 בספר הלימוד

- נכיר ונדגים את המושגים:

- טיפוס נתונים מופשט (ADT=abstract data type)

- מבנה נתונים (data structure)

- נחזור ונעמיק בנושא של מבני נתונים בסיסיים:

מחסנית, תור, מערך, רשימה מקושרת, עץ.

Data Structures - מבני נתונים

<u>טיפוס נתונים אבסטרקטי (Abstract Data Type = ADT):</u>

אוסף של פעולות על נתונים כלשהם.

- המשתמש ב-ADT מכיר את הפעולות והשפעתן על הנתונים.

- אינו נדרש להכיר את פרטי המימוש.

לדוגמה:

LIFO
last in first out

<u>מחסנית (Stack)</u>

מוגדרת ע"י הפעולות הבאות:

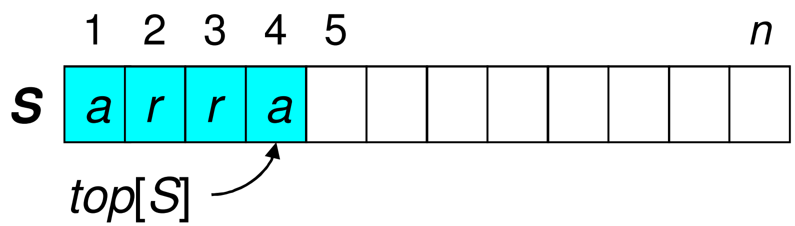
- Push(S, x) – הכנסת איבר לראש המחסנית (כולל בדיקת overflow)
- Pop(S) – הוצאת האיבר שבראש המחסנית (כולל בדיקת underflow)
- StackEmpty(S) ו- StackFull(S) – בדיקה אם המחסנית ריקה/מלאה

<u>מבנה נתונים (data structure):</u> שיטת אחסון וארגון של נתונים שמאפשרת ביצוע פעולות מסוימות. מבנה נתונים הוא מימוש ל- ADT.

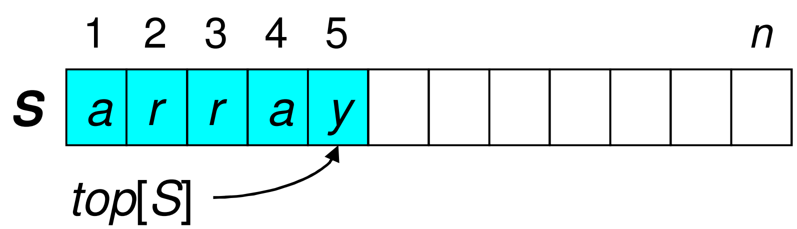
מבני נתונים אפשריים למימוש מחסנית:
1. מערך
2. רשימה מקושרת

מימוש מחסנית באמצעות מערך

```
Push(S, x)
if StackFull(S)
        error "overflow"
top[S] ← top[S] + 1
S[top[S]] ← x
```

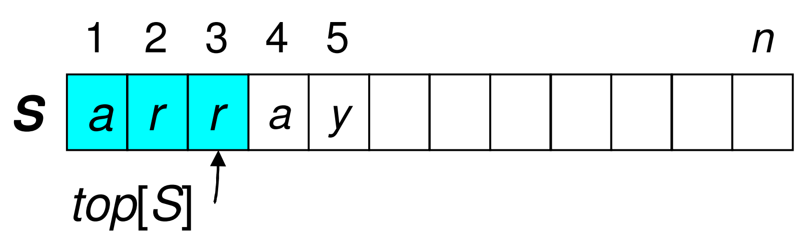



```
Pop(S)
if StackEmpty(S)
        error "underflow"
x ← S[top[S]]
top[S] ← top[S] – 1
return x
```



Push(*S*, 'y')

```
StackFull(S)
return (top[S] = length[S])
```



2 x **Pop**(*S*)

```
StackEmpty(S)
return (top[S] = 0)
```

Data Structures - מבני נתונים

FIFO
first in first out

תור (Queue)

מוגדר ע"י הפעולות הבאות:

- Enqueue(Q, x) – הכנסת איבר לסוף התור (כולל בדיקת overflow)
- Dequeue(Q) – הוצאת האיבר שבראש התור (כולל בדיקת underflow)
- QueueEmpty(Q) ו- QueueFull(Q) – בדיקה אם התור ריק/מלא

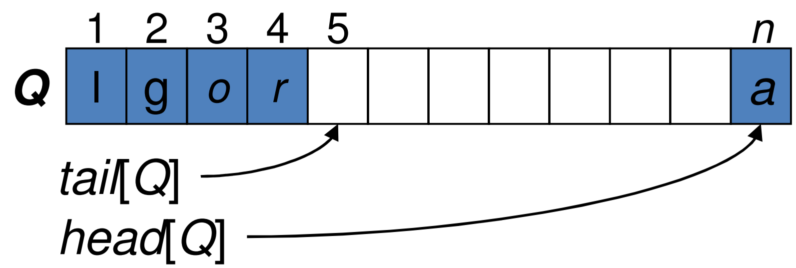
מבני נתונים אפשריים למימוש תור:

1. מערך
2. רשימה מקושרת

מימוש תור באמצעות מערך

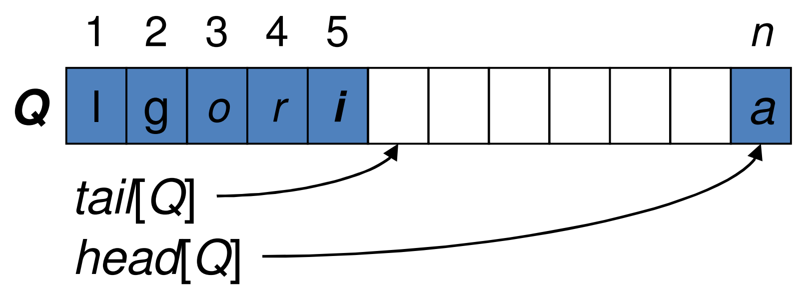
Enqueue(Q, x)

1. **if** QueueFull(Q)
2.   **error** “overflow”
3. $Q[tail[Q]] \leftarrow x$
4. $tail[Q] \leftarrow (tail[Q] \bmod n) + 1$



Dequeue(Q)

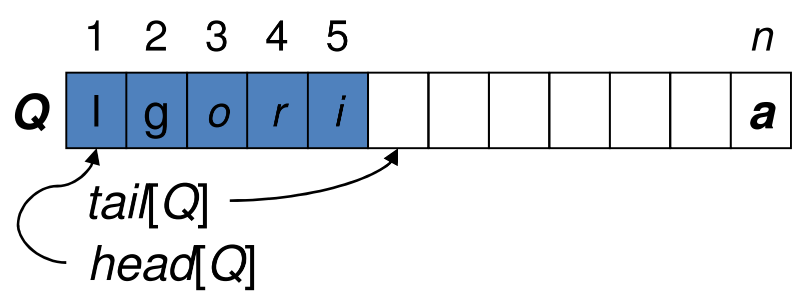
1. **if** QueueEmpty(Q)
2.   **error** “underflow”
3. $x \leftarrow Q[head[Q]]$
4. $head[Q] \leftarrow (head[Q] \bmod n) + 1$
5. **return** x



Enqueue(Q,’i’)

QueueEmpty(Q)

1. **return** ($head[Q] = tail[Q]$)



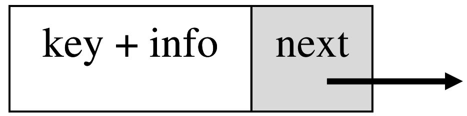
Dequeue(Q)

QueueFull(Q)

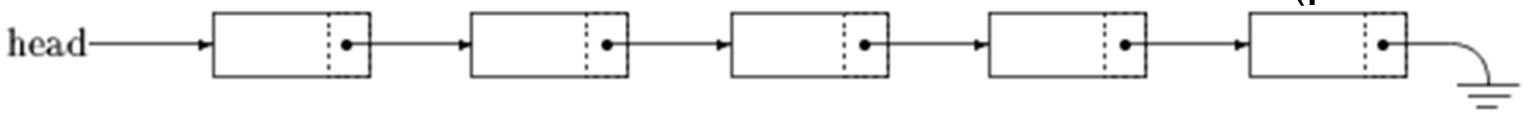
1. **return** ($(tail[Q] \bmod n) + 1 = head[Q]$)

רשימות מקושרות – Linked lists

<u>רשימה מקושרת</u> היא מבנה נתונים המורכב מסדרת רשומות, שבכל אחת יש מצביע לרשומה הבאה.



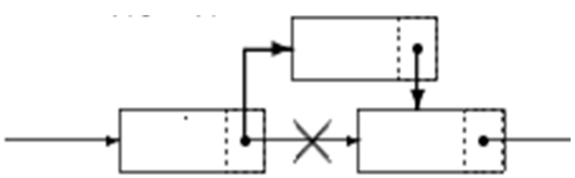
כמו במערך, האיברים מסודרים בסדר ליניארי, אבל בעזרת מצביעים (ואינם בהכרח רצופים בזיכרון).



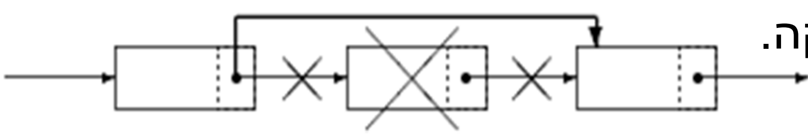
הרשומה האחרונה מצביעה ל- Nil, וראש הרשימה הוא מצביע לאיבר הראשון.

<u>פעולות על רשימות מקושרות</u>

- חיפוש: ע"י מעבר ליניארי על הרשימה, עד שנמצא איבר בעל המפתח המבוקש, או עד שמגיעים לסוף הרשימה.
  זמן: $\Theta(n)$.



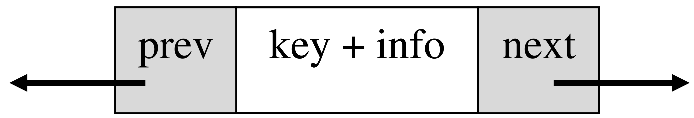
- הכנסה: בהינתן מצביע לאיבר <u>שלפני</u> מקום ההכנסה.
  זמן: $\Theta(1)$

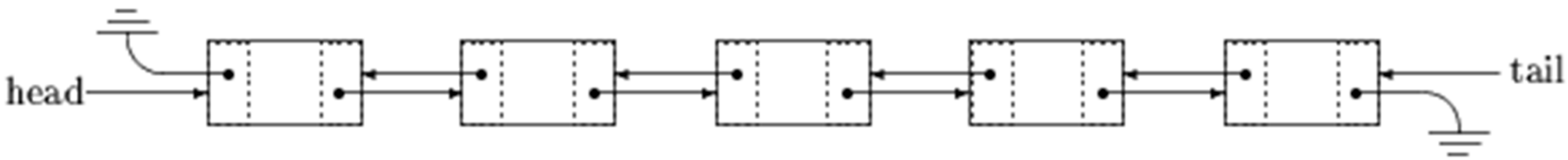


- הוצאה: בהינתן מצביע לאיבר <u>שלפני</u> האיבר למחיקה.
  זמן: $\Theta(1)$

<u>גרסאות של רשימות מקושרות</u>

• <u>רשימה מקושרת דו-כיוונית</u>:





<u>יתרון</u>: אפשר להוציא איבר בהינתן מצביע <u>אליו</u> (ולא לאיבר שלפניו),

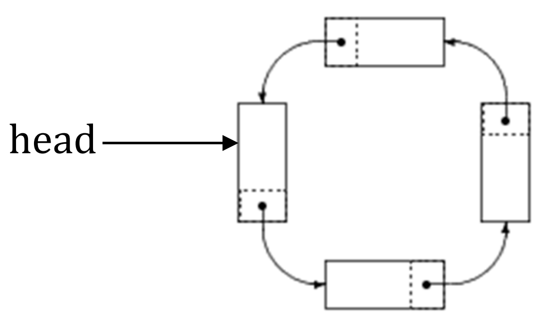
וכן להכניס איבר לפני או אחרי <u>איבר נתון</u>.

מימוש חיפוש, הכנסה ומחיקה מרשימה מקושרת דו-כיוונית בספר הלימוד, עמ' 172-173.

שמות הפעולות: List-Search(L, k), List-Insert(L, x), List-Delete(L, x)

• <u>רשימה מקושרת ממוינת</u>, בה סדר הרשומות מתאים לסדר המפתחות.

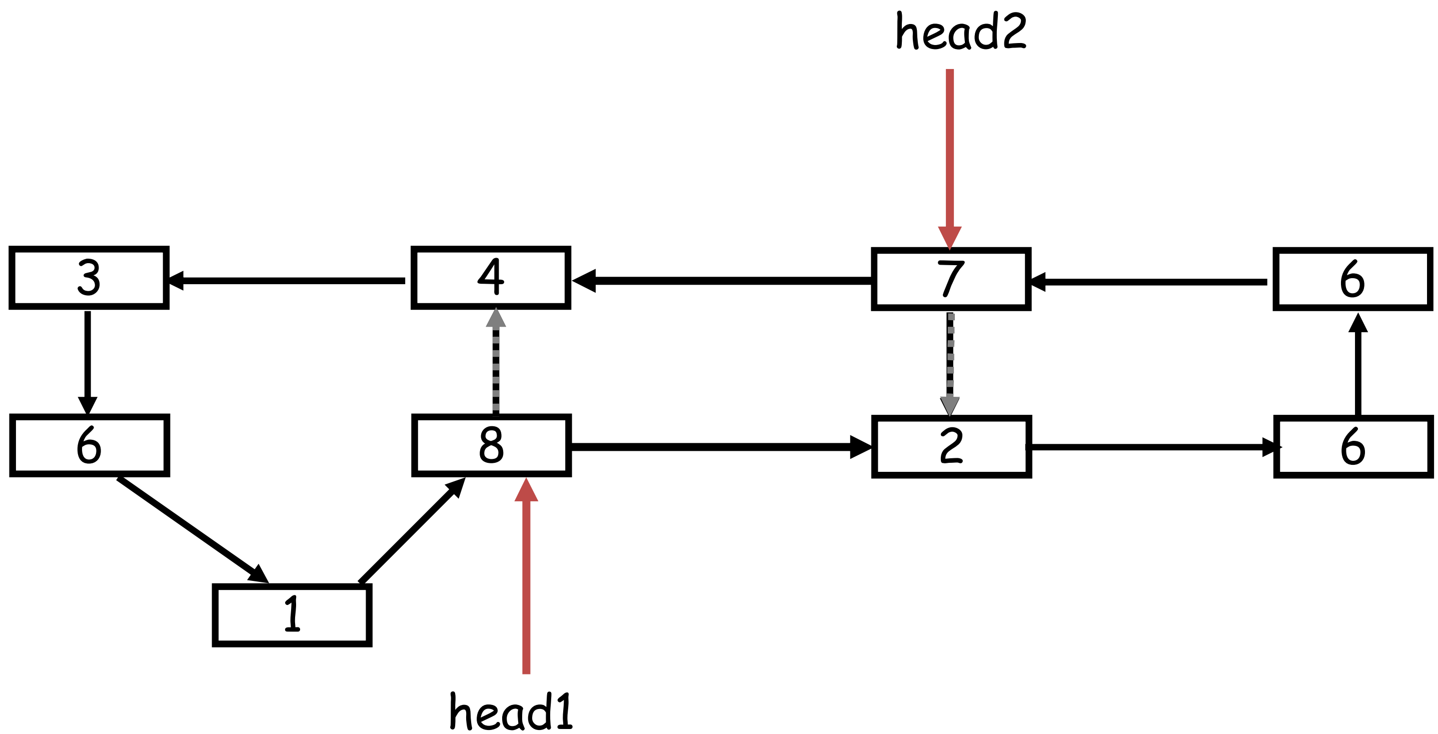
• <u>רשימה מקושרת מעגלית</u>:



גרסאות של רשימות מקושרות

בחירת סוג הרשימה בה נשתמש תלוי באילוצים ובמטרות.

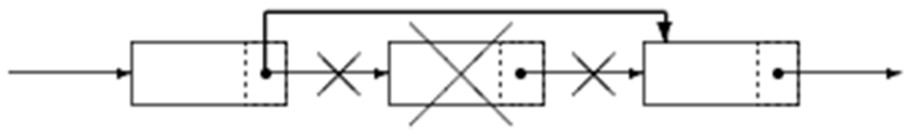
למשל, איחוד רשימות קל לביצוע בזמן קבוע ברשימות מעגליות.



<u>גרסאות של רשימות מקושרות</u>

<u>תרגיל</u>

כאמור, לצורך מחיקת איבר מרשימה מקושרת חד כיוונית צריך להיות בידינו מצביע לאיבר <u>שלפניו</u>:



הציעו שיטה למחוק איבר מרשימה חד כיוונית בהינתן מצביע <u>אליו</u>.

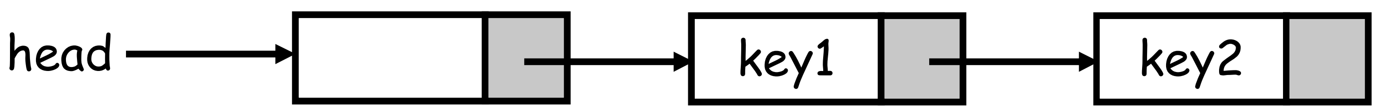
מה החסרונות של השיטה?

זקיף ברשימה מקושרת

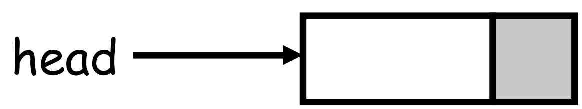
כאשר מוחקים איבר מראש רשימה, או מכניסים איבר לראש רשימה, יש לעדכן את המצביע לראשה.

הדבר יוצר קוד מסורבל יותר, עקב הטיפול במקרי הקצה האלו.

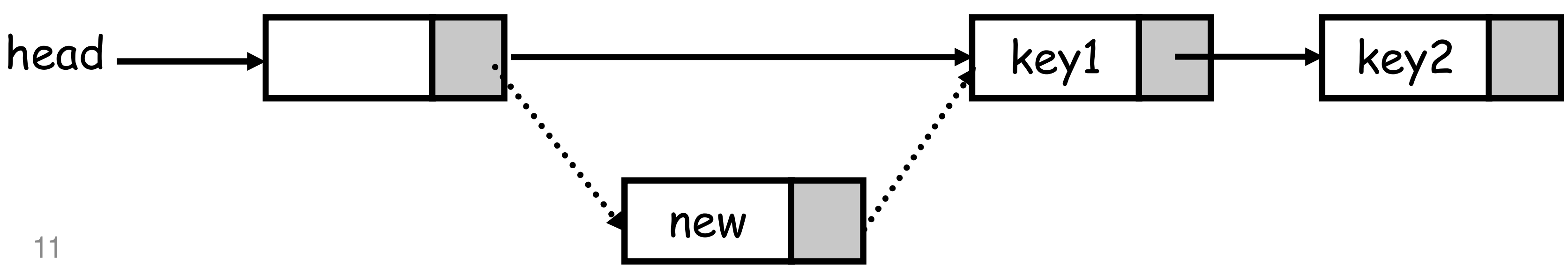
ניתן להוסיף רשומה ריקה בראש הרשימה המקושרת, ואז מקרי קצה אלו מטופלים במסגרת המקרים הרגילים. רשומה כזו נקראת **זקיף (sentinel)**.



רשימה ריקה:



כעת אין הבדל בין הכנסה בראש הרשימה להכנסה רגילה (head לא משתנה):



רשימות מקושרות לעומת מערכים

כאמור, רשימה מקושרת היא מבנה נתונים אפשרי עבור ה- ADT’s מחסנית ותור.

- מחסנית תמומש למשל כרשימה מקושרת חד-כיוונית (איברים יוכנסו ויוצאו מתחילת הרשימה)
- תור ימומש למשל כרשימה מקושרת חד-כיוונית עם מצביע נוסף לסוף הרשימה.

<u>מה עדיף – מערך או רשימה מקושרת?</u>

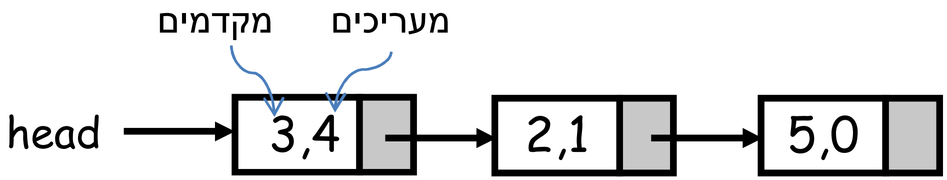
<u>יתרונות לרשימה</u>

1. מאפשר הקצאת זיכרון דינמית בעת הצורך.
2. אין צורך להקצות מקום זיכרון רצוף.
3. הוצאת איבר מאמצע רשימה לא מותירה "חור". לכן סריקת האיברים בזמן ליניארי במספרם.

<u>יתרונות למערך</u>

1. גישה לאיבר לפי אינדקס בזמן $\Theta(1)$.

דוגמה: נניח שברצוננו למצוא שיטה יעילה לייצוג פולינומים ממעלה 6 לכל היותר.
איך ייוצג הפולינום $3x^4+2x+5$? מה יותר חסכוני?



מקדמים	5	2	0	0	3	0	0
מעריכים	0	1	2	3	4	5	6

מערך ורשימה מקושרת כ- ADT

למעשה, אפשר להתייחס למערך ולרשימה מקושרת לא רק כאל מבני נתונים למימוש ADT’s, אלא גם כאל ADT’s בעצמם. למשל, עבור מערך:

<u>מערך (array)</u>

מוגדר ע"י הפעולות הבאות:

- $\text{Get}(A, i)$ – החזרת האיבר בעל אינדקס i
- $\text{Put}(A, i, x)$ – אחסן באינדקס i את האיבר x
- $\text{ArrayEmpty}(A)$ ו- $\text{ArrayFull}(A)$ – בדיקה אם המערך ריק/מלא

מבני נתונים למערך:

1. איזור רציף בזיכרון, כאשר אינדקס i מחושב ע"י $base + (i\text{-}1)\cdot \text{sizeof}(Type)$
2. איזור לא רציף בזיכרון, ואז חישוב האינדקס מסובך יותר
3. רשימה מקושרת (בד"כ לא סביר)...

אם כן המושגים ADT ומבנה נתונים תלויים בהקשר.

מערכים דו-/רב-מימדיים כ- ADT

מערך דו-מימדי	מערך k-מימדי
• $\text{Get}(A, i, j)$	• $\text{Get}(A, i_1, i_2, \ldots, i_k)$
• $\text{Put}(A, i, j, x)$	• $\text{Put}(A, i_1, i_2, \ldots, i_k, x)$
• $\text{ArrayEmpty}(A)$ ו- $\text{ArrayFull}(A)$	• $\text{ArrayEmpty}(A)$ ו- $\text{ArrayFull}(A)$

מבנה נתונים סטנדרטי עבור מערך דו-מימדי מסדר $m\times n$ הוא איזור רציף בזיכרון, המורכב מ- m מערכים חד-מימדיים באורך n כל אחד.

- המיקום ה- (i, j) מחושב ע"י $base + ((i\text{-}1)\cdot n+(j\text{-}1))\cdot\ \text{sizeof}(Type)$

<u>שאלה</u>

במימוש הסטנדרטי של מטריצה $n\times n$, סיבוכיות הזמן של פעולת השחלוף (transpose) היא $\Theta(n^2)$.
הציעו מימוש אחר למטריצה $n\times n$, שבו סיבוכיות השחלוף היא $\Theta(1)$, מבלי לפגוע בסיבוכיות של הפעולות Get ו- Put.

מערכים דו-/רב-מימדיים

נגדיר את ה- ADT הבא:

מטריצת אלכסונים (מטריצת Toeplitz)

מטריצה ריבועית $n\times n$ שאיברי כל אלכסון בה זהים זה לזה.

מוגדרת ע"י הפעולות הבאות:

- $\mathrm{Get}(i, j)$ - החזר את האיבר במיקום (i, j).
- $\mathrm{Put}(i, j, x)$ – אחסן את האיבר x במיקום (i, j). יש לעדכן את כל האלכסון.

$$
\begin{array}{cc}
 & n \\
n & \begin{bmatrix} 4 & 2 & -1 \\ 0 & 4 & 2 \\ 7 & 0 & 4 \end{bmatrix}
\end{array}
$$

מימוש סטנדרטי

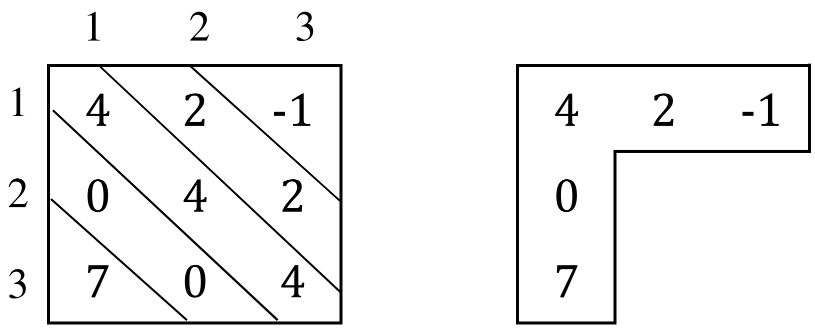
(n מערכים חד-מימדיים):

$\mathrm{Get}(i, j)$	$\mathrm{Put}(i, j, x)$	**סיבוכיות מקום**

מערכים דו-/רב-מימדיים

מבנה נתונים יעיל יותר?

נשמור רק "נציג" אחד מכל אלכסון.



כלומר נשמור מערך באורך $2n$-1.

$2n$-1

A	7	0	4	2	-1
j-i =	-2	-1	0	1	2

נשים לב ש- j-i קבוע בכל אלכסון:

Get(i, j)

1. **return** $A[j\text{-}i+n]$

Put(i, j, x)

1. $A[j\text{-}i+n] \leftarrow x$

Get(i, j)	Put(i, j, x)	**סיבוכיות מקום**
$\Theta(1)$	$\Theta(1)$	$\Theta(n)$

<u>איתחול מערך בזמן קבוע</u>

איתחול מערך A באורך n באפסים (או כל ערך התחלתי אחר) דורש $\Theta(n)$ זמן.

הדבר בעייתי במיוחד כאשר מדובר על מערכים גדולים, שרק חלק קטן מאיבריהם יהיו בשימוש. כלומר פעולות קריאה וכתיבה יהיו "זולות", אך האיתחול יהיה "יקר".

נראה כעת שיטה לאיתחול מערך בזמן $\Theta(1)$, ובתמורה נשתמש בעוד זיכרון.

<u>פתרון שגוי</u>

במקום לאפס את A, נחזיק מערך נוסף $INIT$ ונרשום בו 1 רק במקומות בהם A מכיל ערך "אמיתי". בעת גישה לאיבר של A שאינו מכיל ערך אמיתי נחזיר 0.

A	?	8	?	?	3	?	?	?
$INIT$		1			1			

הבעיה:

שאר המקומות של $INIT$ עלולים גם להכיל 1, ולכן יש לאתחל אותו!

איתחול מערך בזמן קבוע

<u>פתרון</u>

נחזיק שני מערכים נוספים ומשתנה נוסף.

מצב התחלתי – כל התאים מזוהים כ"זבל".

```
Is-Garbage(i)
if  1≤from[i]<top  and  to[from[i]]=i
        return "not garbage"
else    return "garbage"
```

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

```
Put(A, i, x)
A[i] ← x
if  Is-Garbage(i)
        from[i] ← top
        to[top] ← i
        top ← top +1
```

	1	2	3	4	5	6	7	8
A	X	X	X	X	X	X	X	X
from	X	X	X	X	X	X	X	X
to	X	X	X	X	X	X	X	X

$top=1$

- לדוגמה $A[5] \leftarrow 3$

Get תחזיר 3 עבור $i=5$

לכל i אחר תחזיר 0

	1	2	3	4	5	6	7	8
A	X	X	X	X	**3**	X	X	X
from	X	X	X	X	**1**	X	X	X
to	**5**	X	X	X	X	X	X	X

$top=2$

איתחול מערך בזמן קבוע

- עוד דוגמה: $A[8] \leftarrow 12$

  Get תחזיר 12 עבור $i=8$

  3 עבור $i=5$

  לכל i אחר תחזיר 0

	1	2	3	4	5	6	7	8
A	X	X	X	X	**3**	X	X	**12**
from	X	X	X	X	**1**	X	X	**2**
to	**5**	**8**	X	X	X	X	X	X

$top=3$

```
Is-Garbage(i)
  if  1≤ from[i]<top  and  to[from[i]]=i
          return "not garbage"
  else    return "garbage"
```

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

```
Put(A, i, x)
  A[i] ← x
  if  Is-Garbage(i)
          from[i] ← top
          to[top] ← i
          top ← top +1
```

עצים בינאריים (binary trees)

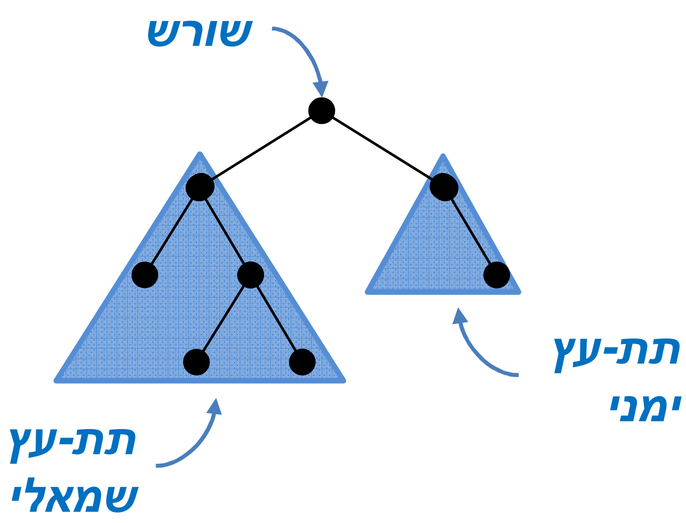
עץ בינארי הוא מבנה נתונים שמורכב מרשומות המסודרות בסדר היררכי.

(זאת בניגוד למשל לרשימה מקושרת, בה הרשומות מסודרות ליניארית).

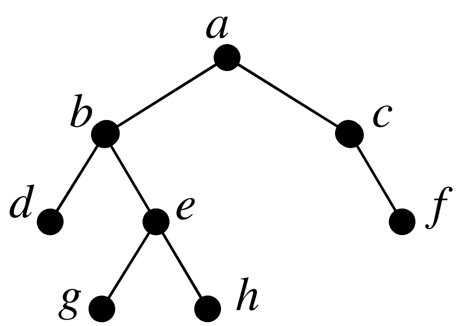
ניתן להגדיר עץ בינארי רקורסיבית:

עץ בינארי:

- או שהוא אינו מכיל כלל רשומות (עץ בינארי ריק)
- או שהוא מכיל 3 קבוצות זרות של רשומות:
  1. רשומה אחת הנקראת שורש (root)
  2. עץ בינארי הנקרא תת-העץ השמאלי
  3. עץ בינארי הנקרא תת-העץ הימני



עצים בינאריים (binary trees)



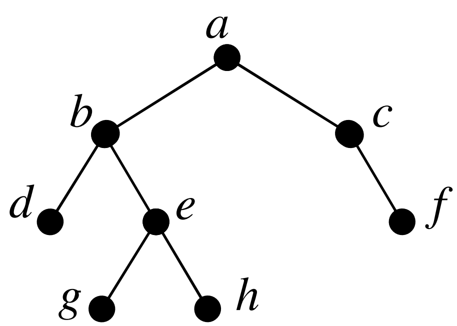
מונחים

- רשומה בעץ בינארי נקראת צומת (node)
- "חיבור" בין שתי רשומות נקרא קשת (edge)

- עבור צומת v:
  - הבן השמאלי (left child) של v הוא שורש תת-העץ השמאלי שלו
  - הבן הימני (right child) של v הוא שורש תת-העץ הימני שלו
  - v נקרא האב (parent) של בניו, והם נקראים אחים.
  - אב קדמון (ancestor) של v הוא כל צומת שנמצא בינו לבין השורש
  - צאצא (descendant) של v הוא כל צומת שנמצא בתת העץ ש- v שורשו

- צומת ללא בנים ייקרא עלה (leaf), אחרת ייקרא צומת פנימי (internal node).

עצים בינאריים (binary trees)



מונחים

- מסלול (path) בעץ הוא סדרת צמתים שבין כל שניים יש קשת.
  אורך מסלול הוא מספר הקשתות שבו.

לדוגמה: אורך המסלול (a, b, e, h) הוא 3

- עומק (depth) של צומת הוא אורך המסלול משורש העץ עד אליו

לדוגמה: $\text{depth}(e) =$

$\text{depth}(a) =$

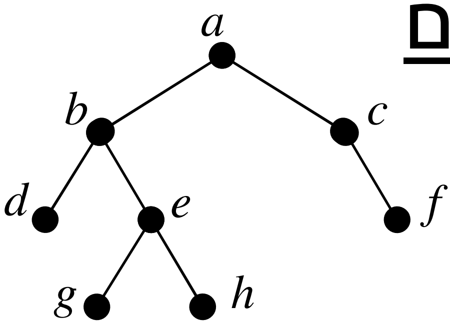
- גובה (height) של צומת הוא אורך המסלול לעלה הכי עמוק שהוא צאצא שלו

לדוגמה: $\text{height}(b) =$

$\text{height}(a) =$

גובה של עץ הוא גובה השורש שלו.

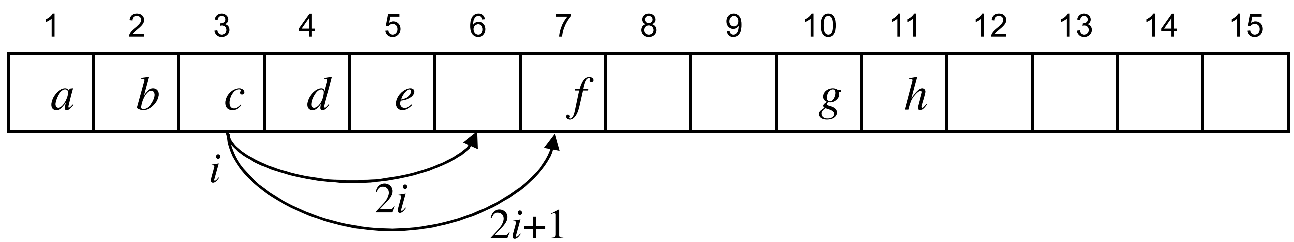
אכסון בזיכרון של עצים בינאריים



ניתן לאכסן עץ בינארי במערך, רמה אחרי רמה משמאל לימין.

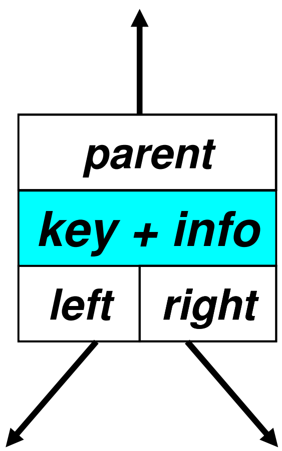
כדי לשמור על המבנה ההיררכי, בנים "חסרים" ייוצגו כתאים ריקים

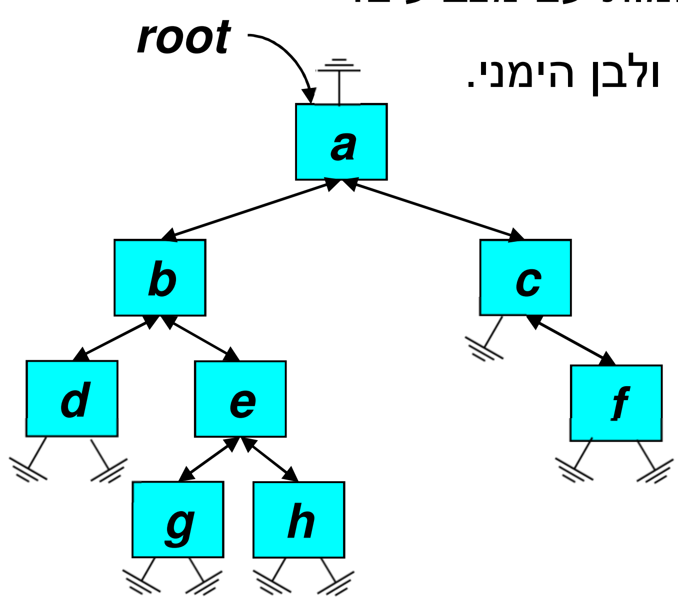
(מה שעלול לגרום בזבוז זיכרון):



ברוב המקרים יעיל יותר להשתמש ברשומות עם מצביעים:

- בכל רשומה יש מצביעים לבן השמאלי ולבן הימני.
- לעיתים גם מצביע לאב.

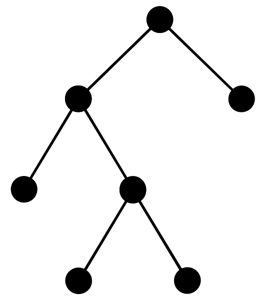




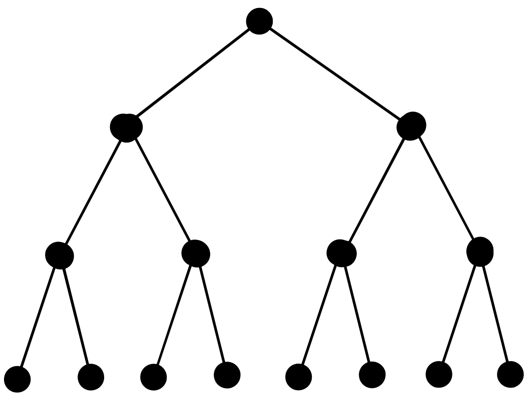
בשורש העץ parent=NIL.

בעלים left=right=NIL.

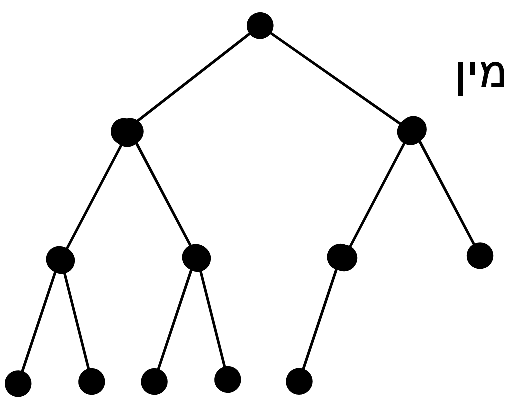
סיווג עצים בינאריים



- בעץ מלא (full) לכל צומת יש 2 או 0 בנים.



- עץ שלם (complete) הוא עץ מלא שבו על העלים באותו עומק.



- עץ כמעט שלם הוא עץ שלם שבו חסרים עלים ברצף מצד ימין

אכסון במערך מתאים יותר עבור עצים שלמים או כמעט שלמים, כי אז אין "חורים".

שאלות חזרה

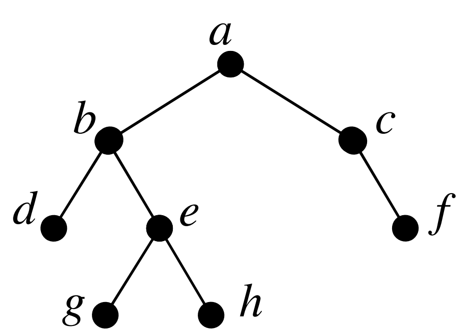
1. בשפת C ניתן ליצור מערך דו-מימדי בגודל 10x10 באמצעות השורה:

```
int A[10][10];
```

הסבירו את ההבדל בין ה- ADT מערך דו-מימדי לבין מבנה הנתונים שנוצר בעת ביצוע השורה הנ"ל.

2. הסבירו מדוע לא ניתן לבצע חיפוש בינארי על רשימה מקושרת ממוינת, בזמן $\Theta(\log n)$.
איזו תכונה של מערך לא מתקיימת ברשימה מקושרת?

3. ראינו שתי שיטות לאכסון עצים בינאריים: מערך, ורשומות עם מצביעים.
נניח שכל רשומה (עם המידע הנוסף) תופסת 10 בתים, וכל מצביע תופס 4 בתים.
כמה בתים יתפוס כל אחד מהמימושים עבור העץ הבא:



תשובות לשאלות חזרה

1. ה- ADT מערך דו-מימדי מאפשר קריאה ואיחסון של ערכים לפי שני אינדקסים. בעת ביצוע השורה הנ"ל, נוצר מימוש ל- ADT כזה, באמצעות איזור רציף בזיכרון הכולל 10 מערכים חד-מימדיים באורך 10 כל אחד.

2. בחיפוש בינארי במערך ממוין יש הנחה שניתן לגשת לכל איבר במערך בזמן $\Theta(1)$ באמצעות האינדקס שלו. ברשימה מקושרת הנחה זו לא מתקיימת. למשל, כדי להגיע לאיבר האמצעי צריך לעבור על מחצית מהאיברים בערך.

3. במערך, נזדקק ל- 11 תאים שכל אחד יתפוס 10 בתים. סה"כ 110 בתים.
ברשומות עם מצביעים: יש 8 רשומות, כל אחת תופסת 10 בתים עבור המפתח והמידע הנוסף ועוד 4 בתים עבור כל אחד מ- 3 המצביעים. סה"כ: 176 בתים.
כלומר עבור העץ הזה עדיפה הדרך הראשונה.
אבל, אילו למשל הוספנו בן לצומת h, הרי שהמימוש במערך היה הופך לבזבזני ביותר.

תרגילים

תרגילים נוספים

1. שאלה 10.1-6 מספר הלימוד.
הראו כיצד ניתן לממש תור באמצעות שתי מחסניות. נתחו את זמן הריצה של הפעולות על התור.

2. נגדיר "מחסנית מינימום" כ- ADT התומך בפעולות הבאות:

- Create - אתחול מבנה הנתונים כאשר הוא ריק.
- Insert(x) - הכנסת המספר x למבנה.
- RemoveLast - הוצאת המספר שהוכנס אחרון והחזרתו כפלט.
- Min - החזרת המספר הקטן ביותר במבנה (ללא הוצאתו).
- ChangeMin(k) - שינוי ערך המספר הקטן ביותר במבנה ל- k.

מותר להניח כי בכל זמן נתון כל המספרים במבנה שונים זה מזה.

א. הציעו מימוש ל"מחסנית מינימום", כאשר סיבוכיות הזמן הנדרשת לארבע הפעולות הראשונות היא $O(1)$, ולפעולה ChangeMin ‏$O(t)$, כש- t הינו מספר האיברים במבנה שהוכנסו אחרי האיבר המינימלי.

ב. הוחלט להוסיף את הפעולה הבאה:

- Add(d) - הוספת d לכל המספרים במבנה. סיבוכיות זמן נדרשת: $O(1)$.

הסבירו כיצד לממש את הפעולה החדשה, ומהם השינויים הדרושים במימוש הפעולות מסעיף א' כך שלא יהיה שינוי בסיבוכיותן.

3. נתונות m קבוצות $F_1, F_2,\ldots, F_m$. כל קבוצה יכולה להכיל איברים מתוך $\{1, 2,\ldots, n\}$.

עליכם לתכנן מבנה נתונים, שיתמוך בפעולות הבאות (תארו את מבנה הנתונים והסבירו איך תתבצע כל אחת מהפעולות):

- Relate (i, a) - פעולה שבודקת אם האיבר a נמצא בקבוצה F_i.

  סיבוכיות הזמן הנדרשת: $O(1)$.

- Insert (i, a) - פעולה שמוסיפה את האיבר a לקבוצה F_i.

  סיבוכיות הזמן הנדרשת: $O(1)$.

- Intersect (i, j, k) - פעולה שמבצעת חיתוך בין הקבוצות F_i ו- F_j ומכניסה את התוצאה ל- F_k

  (כלומר: $F_k = F_i \cap F_j$).

  לפני ביצוע הפעולה, F_k היא קבוצה ריקה.

  סיבוכיות הזמן הנדרשת: $O(\min(|F_i|, |F_j|))$.

- Union (i, j, k) - פעולה שמבצעת איחוד בין הקבוצות F_i ו- F_j ומכניסה את התוצאה ל- F_k

  (כלומר: $F_k = F_i \cup F_j$).

  לפני ביצוע הפעולה, F_k היא קבוצה ריקה.

  סיבוכיות הזמן הנדרשת: $O(\max(|F_i|, |F_j|))$.

פתרון 1

נממש את התור באמצעות שתי מחסניות S_1 ו- S_2 באופן הבא:

המחסנית S_1 תייצג את סוף התור והמחסנית S_2 תייצג את ראש התור. כלומר, איברים חדשים יוכנסו ל-S_1 והאיבר הוותיק ביותר יישלף מתוך S_2.

בעת הוצאת איבר מהתור, אם S_2 ריקה, נעביר את כל האיברים בזה אחר זה מ-S_1 ל- S_2 ואז נשלוף את האיבר שבראש S_2. שימו לב, שלאחר שנעשה זאת אין צורך להחזיר את כל האיברים ל-S_1 !

להלן הפסידו-קוד של הפעולות על התור, ללא בדיקת overflow ו- underflow:

```
Enqueue(Q, x)
    Push(S1, x)
```

זמן הריצה: $\Theta(1)$

```
Dequeue(Q)
    if StackEmpty(S2)
        while not StackEmpty(S1)
            Push(S2, Pop(S1))
    return Pop(S2)
```

זמן הריצה: $\Theta(n)$ במקרה הגרוע.

כיצד ימומשו QueueEmpty ו- QueueFull ?

כיצד יש לשנות את Enqueue ו- Dequeue כדי לבדוק overflow ו- underflow?

פתרון 2

תיאור מבנה הנתונים:

נשתמש ברשימה מקושרת דו-כיוונית.

בכל רשומה ברשימה, בנוסף לערך האיבר, נשמור שדה נוסף ובו הערך המינימלי מבין ערכי כל האיברים הותיקים יותר (כולל האיבר עצמו). נקרא לשדה זה *min*.

מימוש הפעולות:

Create - אתחול רשימה מקושרת ריקה.

Insert(x) - הוספת המספר x לראש הרשימה. השדה *min* של האיבר החדש יהיה המינימום מבין x לבין השדה *min* של ראש הרשימה הקודם.

RemoveLast - מחיקת ראש הרשימה, והדפסת ערכו.

Min - החזרת ערך השדה *min* של ראש הרשימה.

ChangeMin(k) - מתחילים מראש הרשימה, ומתקדמים עד שמגיעים לאיבר המינימלי (איך יודעים שהגענו אליו?). משנים את ערך האיבר המינימלי ל-k, ומעדכנים את שדה ה-*min* שלו כמו בהכנסה. חוזרים אחורה עד לראש הרשימה, ובדרך מעדכנים את שדה ה-*min* של כל איבר כמו בהכנסה.

כל הפעולות מתבצעות בזמן קבוע, למעט האחרונה שרצה ב- $O(t)$.

פתרון 2 ב'

נחזיק משתנה $offset$ שיאותחל ל- 0. בכל קריאה ל- Add(d) נוסיף למשתנה זה d.

שינוי בפעולות מסעיף א':

Insert(x) - במקום להכניס את הערך x נכניס את (x - $offset$).

RemoveLast - נוסיף לערך המודפס את $offset$.

Min - נוסיף לערך המוחזר את $offset$.

ChangeMin(k) - במקום לשנות את ערך האיבר המינימלי ל-k, נשנה אותו ל- (k - $offset$).

כלומר בעת קריאת ערך יש להוסיף לו $offset$, ואילו בעת כתיבה של ערך (הוספה / שינוי) יש להפחית $offset$.

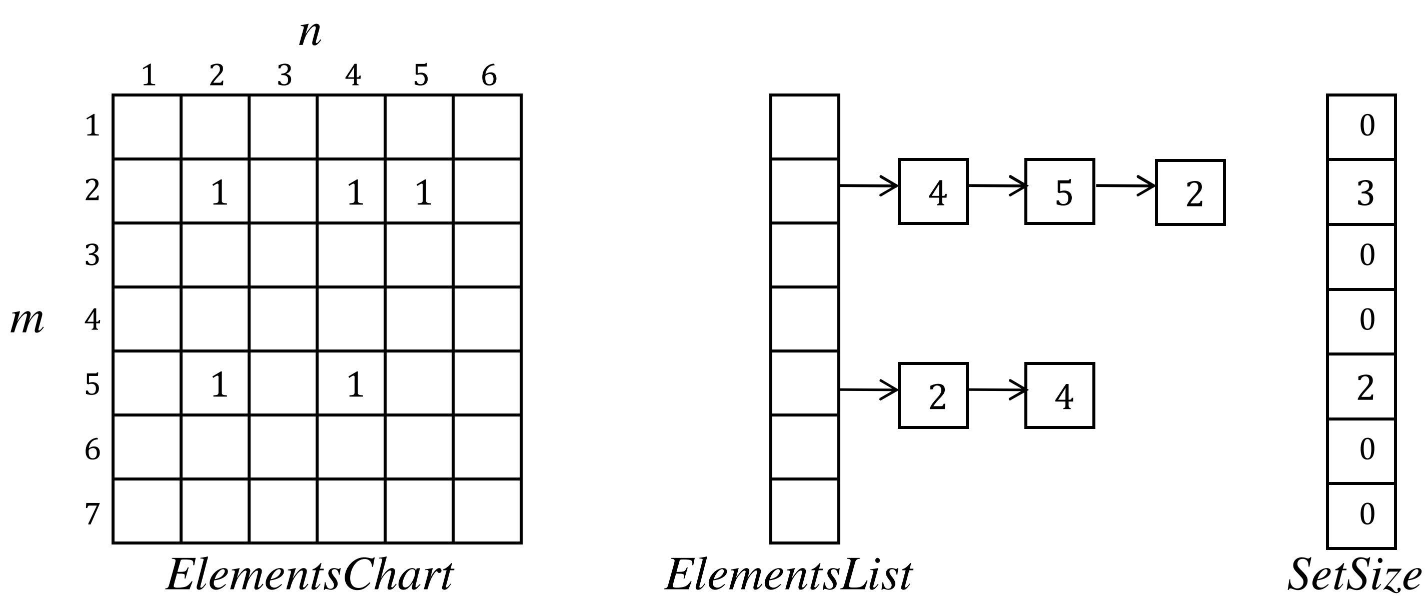
פתרון 3

תיאור מבנה הנתונים:

- מערך בוליאני בגודל $m \times n$ בשם $ElementsChart$, כאשר:
  $ElementsChart[i, j] = True$ אמ"מ $j \in F_i$.

- מערך בגודל m של רשימות מקושרות בשם $ElementsList$, כאשר:
  j מופיע ברשימה $ElementsList[i]$ אמ"מ $j \in F_i$.

- מערך $SetSize$ בגודל m שבו $SetSize[i] = |F_i|$.



המשך פתרון 3

מימוש הפעולות:

□(1)

```
RELATE(i, a)
        return ElementsChart[i, a]
```

□(1)

```
INSERT(i, a)
        if not ElementsChart[i, a]
            then    ElementsChart[i, a] ← true
                    LIST-INSERT(ElementsList[i], a)
                    SetSize[i] ← SetSize[i] + 1
```

Intersect(i, j, k)

ראשית, אם $SetSize[j] < SetSize[i]$ נקרא ל- Intersect(j,i,k), כך שבכל מקרה F_i תהיה הקטנה.

כעת נעבור על $ElementsList[i]$ ולכל איבר x ($x \in F_i$) נבדוק האם Relate(j,x)=$True$, ואם כן נקרא ל- Insert(k, x).

מבצעים $\min\{|F_i|, |F_j|\}$ פעולות, כל אחת בזמן □(1).
סה"כ: □$(\min\{|F_i|, |F_j|\})$

Union(i, j, k)

נעבור על איברי שתי הרשימות F_i ו- F_j.
לכל איבר x נקרא ל- Insert(k, x).

מבצעים $|F_i|+|F_j|$ פעולות, כל אחת בזמן □(1).

נשים לב ש- $|F_i|+|F_j| \le 2\max\{|F_i|, |F_j|\}$

ולכן סה"כ: □$(\max\{|F_i|, |F_j|\})$