האוניברסיטה הפתוחה

מבני נתונים ומבוא לאלגוריתמים
מפגש הנחיה מס' 7

מדעי המחשב, קורס מס' 20407

סמסטר 2016ב

מנחה: ג'ון מרברג

מה ראינו במפגש הקודם?

- חציונים וערכי מיקום
- בעיית הבחירה
- מציאת מינימום ומקסימום
- מציאת האיבר ה- i בגודלו – פתרון אקראי
- מציאת האיבר ה- i בגודלו – פתרון דטרמיניסטי

מפגש שביעי

- נושאי השיעור
  - פרק 8 בספר – מיונים בזמן לינארי
    - מיון-מנייה
    - מיון-בסיס
    - מיון-דלי
  - פרק 10 בספר – מבני נתונים בסיסיים
    - מחסניות ותורים
    - רשימות מקושרות

מבוסס על מצגת של ברוך חייקין ואיציק בייז

מיון מבוסס השוואות

- אין מידע על תכונות כלשהן של ערכי הקלט
  - ההנחה היחידה: כל הערכים ניתנים להשוואה זה עם זה (סדר מלא)
  - כל המידע לסידור הקלט מתקבל מפעולות השוואה בין המפתחות
    - יש 5 פעולות השוואה שונות: $a < b$, $a \leq b$, $a = b$, $a \geq b$, $a > b$
    - לכל פעולת השוואה 2 תוצאות בלבד: נכון או לא נכון
  - דוגמאות: מיון-הכנסה, מיון-בועות, מיון-מיזוג, מיון-ערמה...
- אלגוריתם מיון מבוסס השוואות – עורך השוואות בין זוגות מפתחות עד לקבלת הסידור הנכון
  - <u>הקלט</u>: סדרת מפתחות $a = \langle a_1, a_2, \ldots, a_n \rangle$
  - <u>הסידור</u>: תמורה π של האינדקסים $\langle 1, 2, \ldots, n \rangle$

    $\pi = \langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$
  - <u>הפלט</u>: הסדרה $a_\pi = \langle a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)} \rangle$,

    כאשר $a_{\pi(i)} \leq a_{\pi(j)}$ לכל $1 \leq i \leq j \leq n$

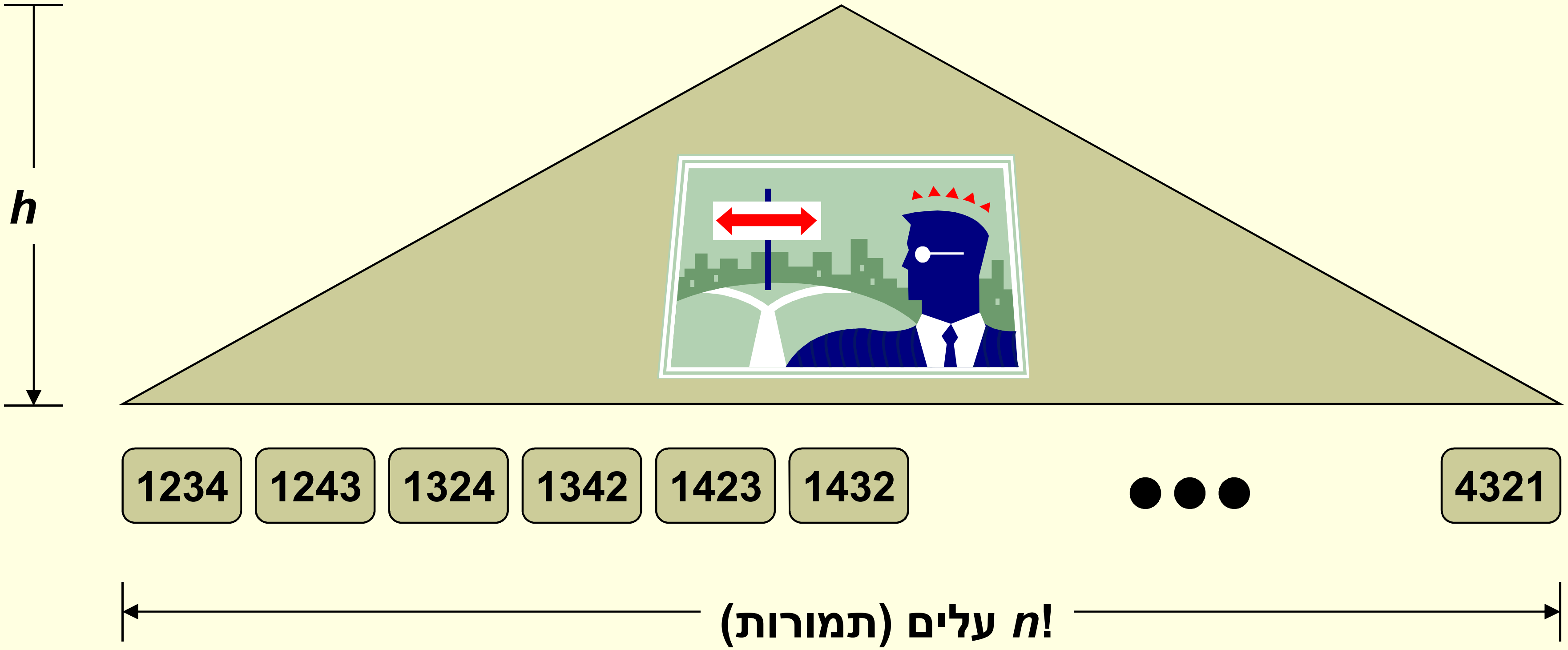
חסם תחתון על מיון מבוסס השוואות

- נניח בלי הגבלת כלליות שכל *n* המפתחות שונים זה מזה
- נתאר את סדרת ההשוואות שמבצע אלגוריתם A על הקלט כעץ החלטה בינארי
  - <u>שורש העץ</u>: ההשוואה הראשונה שמבצע A
  - <u>צומת פנימית</u>: פעולת השוואה בין שני מפתחות כלשהם $a_i \le a_j$ עם שתי תוצאות אפשריות (המוליכות לשני בנים)
  - <u>עלה</u>: תמורה של הסדרה <1, 2, …, *n*> בהתאם למסלול ההשוואות מהשורש עד לעלה הנתון
- תכונות עץ ההחלטה:
  - מספר העלים = מספר הסידורים האפשריים של הקלט = $n!$
  - גובה העץ = המסלול הארוך ביותר משורש לעלה

= מספר ההשוואות שמבצע A כדי להגיע לפלט במסלול זה

= חסם תחתון על זמן הריצה של A

עץ החלטה



- גובה העץ הוא המסלול הארוך ביותר מהשורש לעלה
- מס' העלים בעץ ההחלטה הוא !n ולכן גובה העץ הוא לפחות $\log(n!)$

$$\log(n!) = \Theta(n\log n)$$

ולכן: $h = \Omega(n\log n)$

מיון מבוסס השוואות – מסקנות

- *לכל* אלגוריתם מיון מבוסס השוואות A, ולכל n, יש *לפחות קלט אחד* בגודל n עליו A יבצע $\Omega(n\log n)$ השוואות
- מסקנה: $\Omega(n\log n)$ הוא חסם תחתון על זמן הריצה של מיון מבוסס השוואות
  - כאשר החסם העליון על זמן הריצה של האלגוריתם הוא $O(n\log n)$ האלגוריתם הוא אופטימלי (אסימפטוטית)
  - דוגמאות: מיון-ערמה, מיון-מיזוג

האוניברסיטה הפתוחה

האם ניתן לשבור את החסם התחתון על מיון?

- רק אם יש מידע מוקדם על תכונות מתמטיות כלשהן של ערכי הקלט

- ניתן להשתמש בתכונות הקלט כדי לבצע על ערכי הקלט פעולות שאינן השוואות
  - פעולות אלה שוברות את החסם התחתון על הזמן $\Omega(n\log n)$
  - בד"כ נשלם על השיפור בזמן עם עלות גדולה יותר של מקום

- <u>לא ניתן </u>להשתמש במיון מסוג זה על קלט כללי כלשהו
  - לכל אלגוריתם יש לפרט את ההנחות על ערכי הקלט עליהם הוא מתבסס
  - האלגוריתם לא יעבוד נכון על קלט שאינו מקיים את ההנחות
  - זוהי טעות נפוצה!

מיון שאינו מבוסס השוואות

- נכיר שלשה אלגוריתמים למיון בזמן לינארי
- כל אלגוריתם מתבסס על תכונות אחרות של הקלט

- מיון-מניה CountingSort
  - תכונות הקלט: ערכים שלמים בתחום $1..k$

- מיון-בסיס RadixSort
  - תכונות הקלט: ערכים שלמים בני d ספרות בבסיס k

- מיון-דלי BucketSort
  - תכונות הקלט: ערכים בהתפלגות אחידה בתחום $[0..1)$

מיון-מנייה

- הנחות על הקלט: כל אחד מ- n מפתחות הקלט הוא מספר שלם בתחום $1..k$
- הרעיון:
  - עבור כל מפתח x נספור כמה מפתחות קטנים או שווים לו *(כולל x עצמו)*
  - *אם m מפתחות קטנים או שווים ל- x*, נציב את x במקום m בפלט
- האלגוריתם משתמש במערך עזר C שגודלו k, *ומבצע* 4 שלבים:
  1. *איפוס*: $C[i]$ מקבל את הערך ההתחלתי 0 $(1 \le i \le k)$
  2. *מנייה*: $C[i]$ מקבל את מספר המופעים של המפתח i בקלט
  3. *צבירה*: $C[i]$ מקבל את סה"כ מספר המופעים של המפתח i *ושל כל המפתחות הקטנים ממנו*
  4. *הצבה*: נשתמש במערך העזר C כדי להציב כל איבר במקום הנכון בפלט
  - נזכור שכל מפתח בקלט הוא יותר מסתם מספר – הוא גם אינדקס לתוך מערך העזר

מיון-מנייה – דוגמה

מערך הקלט (n = 11)

	1	2	3	4	5	6	7	8	9	10	11
A	**7**	**1**	**3**	**1**	**2**	**4**	**5**	**7**	**2**	**4**	**3**

מערך העזר (k = 8)

	1	2	3	4	5	6	7	8
C								

	1	2	3	4	5	6	7	8	
C	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**שלב 1 – איפוס**
C	**2**	**2**	**2**	**2**	**1**	**0**	**2**	**0**	**שלב 2 – מנייה**
C	**2**	**4**	**6**	**8**	**9**	**9**	**11**	**11**	**שלב 3 – צבירה**

מיון-מנייה – דוגמה (המשך)

האוניברסיטה הפתוחה

שלב 4: הצבה

A

1	2	3	4	5	6	7	8	9	10	**11**
7	1	3	1	2	4	5	7	2	4	**3**

C

1	2	**3**	4	5	6	7	8
2	4	**6**	8	9	9	11	11

B

1	2	3	4	5	**6**	7	8	9	10	11
					3					

A[j] = A[11] = 3
C[A[j]] = C[3] = 6
B[C[A[j]]] = B[6] ← A[j] = 3
C[A[j]] = C[3] ← C[A[j]] – 1 = 5

A

1	2	3	4	5	6	7	8	9	**10**	11
7	1	3	1	2	4	5	7	2	**4**	3

C

1	2	3	**4**	5	6	7	8
2	4	5	**8**	9	9	11	11

B

1	2	3	4	5	6	7	**8**	9	10	11
					3		**4**			

A[j] = A[10] = 4
C[A[j]] = C[4] = 8
B[C[A[j]]] = B[8] ← A[j] = 4
C[A[j]] = C[4] ← C[A[j]] – 1 = 7

מיון-מנייה – דוגמה (סוף)

האוניברסיטה הפתוחה

	1	2	3	4	5	6	7	8	**9**	10	11
A	7	1	3	1	2	4	5	7	**2**	4	3

	1	**2**	3	4	5	6	7	8
C	2	**4**	5	7	9	9	11	11

	1	2	3	**4**	5	6	7	8	9	10	11
B				**2**		3		4			

A[j] = A[9] = 2
C[A[j]] = C[2] = 4
B[C[A[j]]] = B[4] ← A[j] = 2
C[A[j]] = C[2] ← C[A[j]] – 1 = 3

●●●

	1	2	3	4	5	6	7	8	9	10	11
A	7	1	3	1	2	4	5	7	2	4	3

	1	2	3	4	5	6	7	8
C	0	2	4	6	8	9	9	11

	1	2	3	4	5	6	7	8	9	10	11
B	**1**	**1**	**2**	**2**	**3**	**3**	**4**	**4**	**5**	**7**	**7**

האוניברסיטה הפתוחה

מיון-מנייה – האלגוריתם

CountingSort (A, B, k)

Input: An input array A, an output array B and an integer k, where $length[A] = length[B] = n$ and $1 \le A[j] \le k$ for all $1 \le j \le length[A]$.

Output: The output array B contains all elements of array A, sorted.

Notes: The algorithm uses an auxiliary array $C[1..k]$.

for $i \leftarrow 1$ **to** k ►stage 1: initialization
    do $C[i] \leftarrow 0$
for $j \leftarrow 1$ **to** $length[A]$ ►stage 2: counting
    do $C[A[j]] \leftarrow C[A[j]] + 1$
►$C[i]$ now contains the number of elements equal to i.
for $i \leftarrow 2$ **to** k ►stage 3: accumulation
    do $C[i] \leftarrow C[i] + C[i-1]$
►$C[i]$ now contains the number of elements less than or equal to i.
for $j \leftarrow length[A]$ **downto** 1 ►stage 4: placement
    do $B[C[A[j]]] \leftarrow A[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$

מיון-מנייה – תכונות

- זמן ריצה: $O(n + k)$
  - אם $k = O(n)$ אז זמן הריצה הוא $O(n)$
  - "נשבר" החסם התחתון $\Omega(n\log n)$ (איך?)
- מקום: $O(n + k)$
- המיון יציב, כלומר איברים בעלי מפתחות שווים יופיעו בפלט באותו סדר יחסי כמו בקלט
  - נובע מכך שהלולאה בשורה 9 רצה "מהסוף להתחלה"
  - תכונת היציבות שימושית אם רוצים למיין רשומות לפי כמה מפתחות (למשל סטודנטים לפי ציון ואחר כך לפי גיל)
  - נראה שימוש ביציבות של מיון-מניה ככלי עזר במיון-בסיס

האוניברסיטה הפתוחה

מיון-בסיס

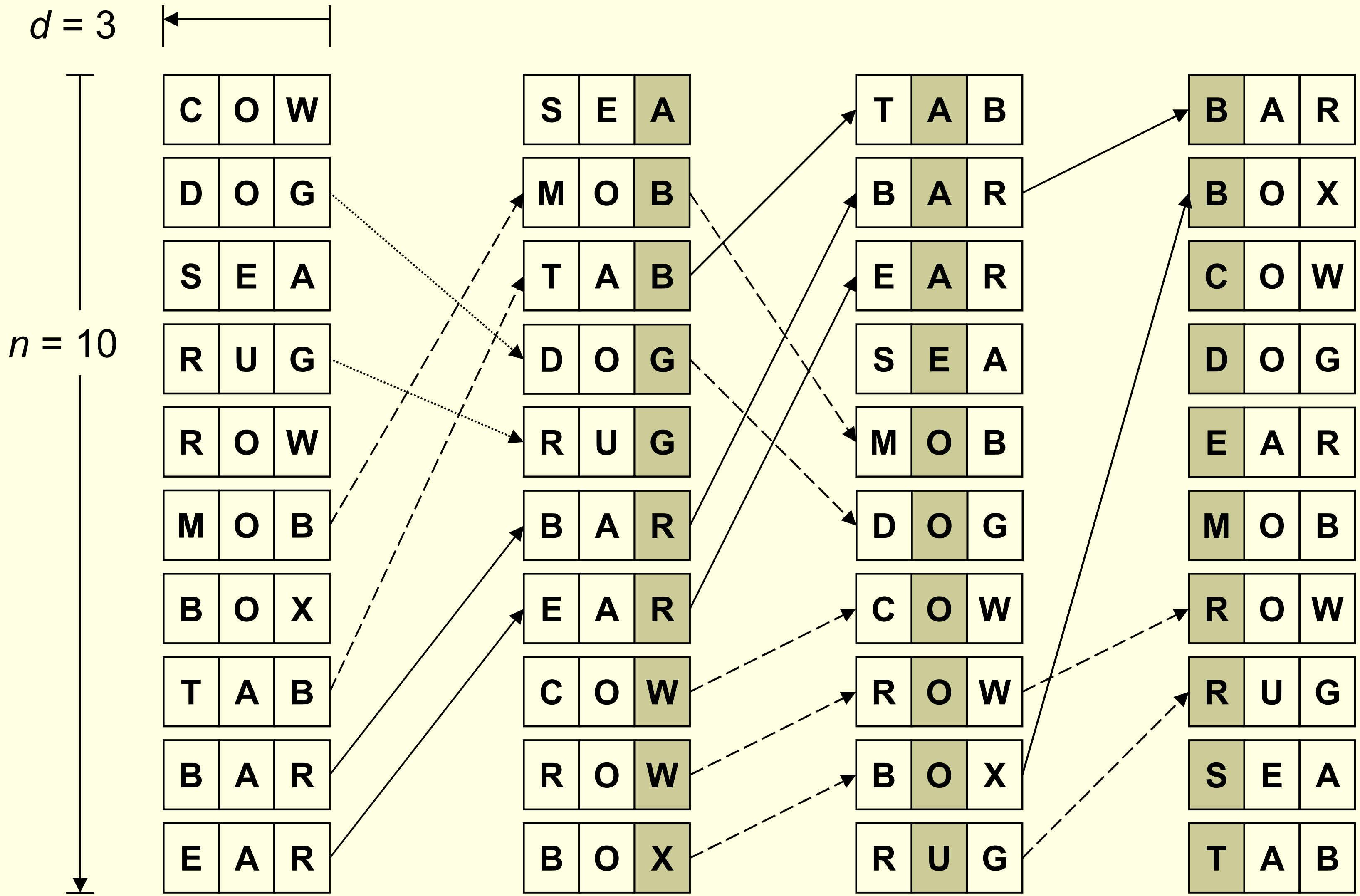
- הנחות על הקלט: כל אחד מ- n מפתחות הקלט הוא מספר שלם בן d ספרות בבסיס k כלשהו
  - הבסיס יכול להיות בינרי (0-1), עשרוני (0-9), הקסאדצימלי (0-9,*A-F*), וכו'
  - אפשר לחשוב גם על אותיות (א-ת, *A-Z*, וכו'), או קוד אסקי
- הרעיון:
  - נבצע מיון יציב לפי ספרה אחר ספרה, החל מהמקום הכי פחות משמעותי
  - בעת המיון על הספרה ה-*j*, הסדר היחסי בין המפתחות שנוצר על-ידי המיונים הקודמים לפי הספרות 1..*j*–1 לא ישתנה

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

$d = 3$
k = decimal

האוניברסיטה הפתוחה

מיון-בסיס – דוגמה (תרגיל 8.3-1)



מיון-בסיס – האלגוריתם

RadixSort(A, d)

Input: An input array A and an integer d, where $A[j]$ is an integer with exactly d digits, for $1 \le j \le length[A]$.

Output: The array A sorted.

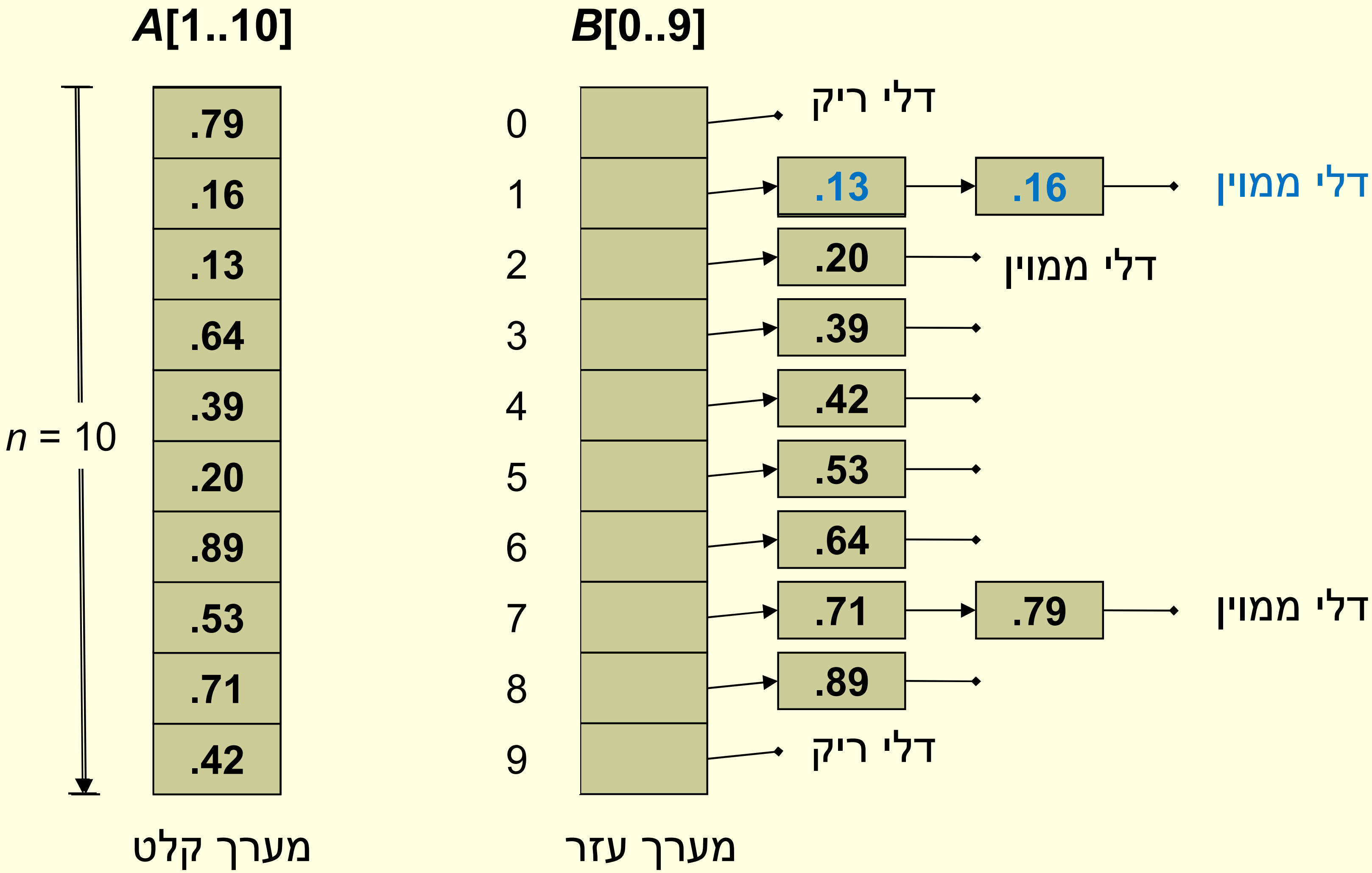
1. **for** $i \leftarrow 1$ **to** d ► go from least significant to most significant digit
2. **do** sort array A on digit i using a stable sort

- הערות לאלגוריתם:
  - המיון לפי ספרה (שורה 2) חייב להיות יציב כדי לשמור את הסדר המצטבר
  - הערכים נתונים בבסיס k כלשהו, כלומר כל ספרה היא בתחום $0..k–1$
    - לכן מתאים להשתמש בשורה 2 במיון-מניה
    - ננרמל את התחום $0..k$-1 לתחום $1..k$
- כאשר הערכים בבסיס k ומשתמשים במיון-מניה לכל ספרה, זמן הריצה של מיון-בסיס הוא $O(d{\cdot}(n+k))$

מיון-דלי

- הנחות על הקלט: *n* מפתחות הקלט הם מספרים ממשיים המתפלגים בצורה אחידה בקטע $[0,1)$
  - *התפלגות אחידה*: ההסתברות שמפתח *x* כלשהו יימצא בתוך תת-קטע נתון נמצאת ביחס ישר לאורך התת-קטע
    - (לדוגמא, ההסתברות שהמפתח יימצא בתת-קטע $[½,¾)$ היא ¼)
  - אם המפתחות מתפלגים בצורה אחידה בקטע $[a,b)$, ניתן "לנרמל" כל מפתח x באמצעות הנוסחה $y = (x–a)/(b–a)$
- הרעיון:
  - נפזר את המפתחות ל-*n* "דליים" (ממוספרים מ-0 עד *n–1*) שכל אחד מהם "מכסה" תת-קטע שגודלו $1/n$. המפתח שערכו *x* נכנס לדלי שמספרו $\lfloor nx \rfloor$
  - נמיין כל אחד מהדליים. המיון הכולל מתקבל ע"י שרשור הדליים הממוינים
  - מכיוון שהתפלגות המפתחות אחידה, אנו מצפים למצוא מספר קטן של מפתחות בכל דלי, ולכן מיון המפתחות בכל דלי יהיה מהיר

מיון-דלי – דוגמה (תרגיל 8.4-1)



האוניברסיטה הפתוחה

מיון-דלי – האלגוריתם

BucketSort(A)

Input: An array A with n keys, distributed uniformly in the range [0,1)

Output: The array A, sorted.

Notes: The algorithm uses an auxiliary array $B[0..n-1]$ of n buckets (lists).

1. $n \leftarrow length[A]$
2. **for** $i \leftarrow 1$ **to** n
3.   **do** insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i \leftarrow 0$ **to** $n-1$
5.   **do** sort list $B[i]$ using (e.g.) InsertionSort
6. concatenate the lists $B[0]$, $B[1]$, ..., $B[n-1]$ together in order

- <u>תוחלת</u> זמן הריצה של מיון דלי היא $O(n)$
  - רק שורות 4-5 רלוונטיות לחישוב התוחלת (n הקריאות למיון הכנסה)
  - ההוכחה מופיעה בספר בעמ' 146
- משמעות התוחלת: אמנם ייתכן קלט "גרוע" שבו רוב המפתחות ייכנסו למספר קטן של דליים אבל מכיוון שההתפלגות אחידה, הסיכוי שזה יקרה הוא קטן מאוד
  - זמן הריצה עבור המקרה הגרוע הוא $O(n^2)$ (לפי מיון הכנסה)

מיון-דלי – הוכחת נכונות

- אם שני מפתחות $A[i]$, $A[j]$ נכנסים לאותו דלי, הם עוברים מיון בדלי ולכן יוצאים לפלט בסדר הנכון
- אחרת, המפתחות נכנסים לשני דליים שונים:
  - $A[i]$ נכנס לדלי $B[x]$, *ואילו* $A[j]$ *נכנס לדלי* $B[y]$
  - נניח בלי הגבלת הכלליות כי $x < y$
    - לפיכך $A[i]$ נמצא בפלט לפני $A[j]$
  - נותר להוכיח שמתקיים $A[i] \le A[j]$
    - נניח בשלילה כי $A[i] > A[j]$
    - מהנחה זו ושורה 3 של האלגוריתם נקבל

$y = \lfloor nA[j] \rfloor \geq \lfloor nA[i] \rfloor = x$, בסתירה לכך ש- $x < y$

מיון-דלי – תרגול

- (תרגיל 8.4-2)

מהו המקרה הגרוע של מיון-דלי? מהו זמן הריצה במקרה הגרוע?
איזה שינוי פשוט ניתן להכניס באלגוריתם כך שזמן הריצה במקרה הגרוע יהיה $O(n\lg n)$ ותישמר התוחלת הלינארית של זמן הריצה?

האוניברסיטה הפתוחה

אלגוריתמי מיון בזמן לינארי - סיכום

- מיון-מניה CountingSort
  - תכונות הקלט: ערכים שלמים בתחום $1..k$
  - זמן ריצה: $O(n+k)$
- מיון-בסיס RadixSort
  - תכונות הקלט: ערכים שלמים בני d ספרות בבסיס k
  - זמן ריצה: $O(d\cdot(n+k))$
- מיון-דלי BucketSort
  - תכונות הקלט: ערכים בהתפלגות אחידה בתחום $[0..1)$
  - זמן ריצה: $O(n)$ במקרה הממוצע

האוניברסיטה הפתוחה

מיון-דלי – תרגול

- (תרגיל 8.4-4)

נתונות n נקודות $p_i = (x_i, y_i)$ בעיגול היחידה. דהיינו, עבור $1 \le i \le n$ מתקיים $0 < x_i^2 + y_i^2 \le 1$.

נניח שהתפלגותן של הנקודות אחידה; כלומר, ההסתברות למצוא נקודה באזור כלשהו של העיגול נמצאת ביחס ישר לשטחו.

תכננו אלגוריתם שתוחלת זמן הריצה שלו היא $\Theta(n)$ למיון n הנקודות על פי מרחקיהן $d_i = \sqrt{(x_i^2 + y_i^2)}$ ממרכז העיגול.

(**רמז**: תכננו את גודלי הדליים כך שישקפו את ההתפלגות האחידה של הנקודות בעיגול היחידה.)

מבנה נתונים

- משמש לשיכון קבוצת נתונים בזיכרון באופן שמאפשר גישה יעילה לנתונים לפי הפעולות הנדרשות על הקבוצה
- יש להפריד בין **מימוש** המבנה לבין **שימוש** בו!
  - מימוש:
    - אופן שיכון הנתונים: בחירת מבני אחסון ומשתנים בהתאם לפעולות הנדרשות
    - אתחול המבנה
    - מימוש אלגוריתם לכל פעולה נדרשת בזמן הנדרש
  - שימוש:
    - אך ורק דרך הפעולות המוגדרות על המבנה
    - אף פעם לא בגישה ישירה לנתונים שבתוך המבנה

מחסנית

- מחסנית (stack) – מדיניות "נכנס אחרון יוצא ראשון" (LIFO)
- פעולות
  - Push(S, x) – הכנסת איבר לראש המחסנית, המחסנית גְדֵלה
    - שגיאת גלישה (overflow) אם המחסנית מלאה
  - Pop(S) – הוצאת האיבר שבראש המחסנית, המחסנית קְטֵנה
    - שגיאת חמיקה (underflow) אם המחסנית ריקה
  - StackEmpty(S) ו- StackFull(S) – בדיקה אם המחסנית ריקה/מלאה
- מימושים
  - באמצעות מערך S ואינדקס top[S] שמצביע לאיבר שהוא ראש המחסנית
    - המחסנית מוגבלת לגודל המערך
  - באמצעות רשימה מקושרת
  - בשני המימושים סיבוכיות הזמן של כל פעולה היא $\Theta(1)$

מחסנית – מימוש ע"י מערך

Push(S, x)

if StackFull(S)
    then error "overflow"
$top[S] \leftarrow top[S] + 1$
$S[top[S]] \leftarrow x$

StackFull(S)

return $top[S] = length[S]$

Pop(S)

if StackEmpty(S)
    then error "underflow"
$x \leftarrow S[top[S]]$
$top[S] \leftarrow top[S] - 1$
return x

StackEmpty(S)

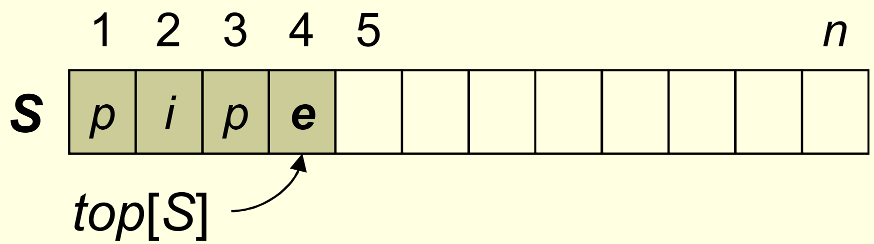
return $top[S] = 0$

StackInit(S)

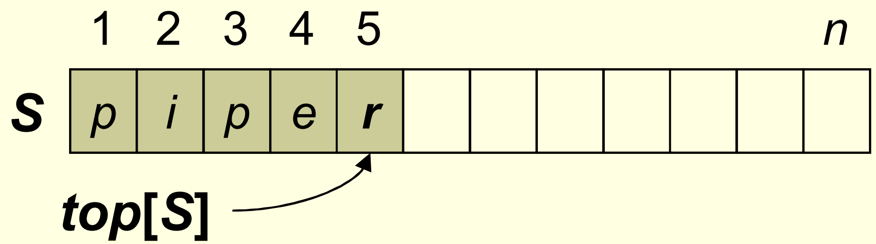
$top[S] \leftarrow 0$

מחסנית – דוגמה

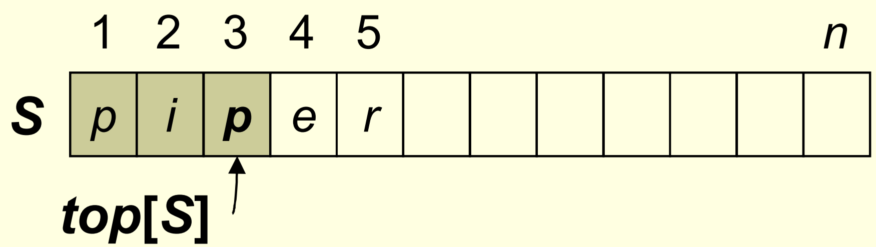
מצב התחלתי



המצב לאחר קריאה ל-Push(*S*, *r*)



המצב לאחר שתי קריאות ל-Pop(*S*)



תור

- תור (queue) – מדיניות "נכנס ראשון יוצא ראשון" (FIFO)
- פעולות
  - Enqueue(*Q*, *x*) – הכנסת איבר לסוף התור, התור גְדֵל
    - שגיאת גלישה (overflow) אם התור מלא
  - Dequeue(*Q*) – הוצאת האיבר שבראש התור, התור קָטֵן
    - שגיאת חמיקה (underflow) אם התור ריק
  - QueueEmpty(*Q*) ו- QueueFull(*Q*) – בדיקה אם התור ריק/מלא
- מימושים
  - באמצעות מערך "מעגלי" Q ושני אינדקסים, *head[Q]* ו- *tail[Q]*
  - באמצעות רשימה מקושרת עם מצביע נוסף לסוף הרשימה
  - בשני המימושים סיבוכיות הזמן של כל פעולה היא $\Theta(1)$

האוניברסיטה הפתוחה

תור – מימוש ע"י מערך "מעגלי"

Enqueue(Q, x)

if QueueFull(Q)
    then error “overflow”
$Q[tail[Q]] \leftarrow x$
$tail[Q] \leftarrow$ Next($tail[Q]$, $length[Q]$)

Dequeue(Q)

if QueueEmpty(Q)
    then error “underflow”
$x \leftarrow Q[head[Q]]$
$head[Q] \leftarrow$ Next($head[Q]$, $length[Q]$)
return x

QueueEmpty(Q)

return ($head[Q] = tail[Q]$)

QueueFull(Q)

return Next($tail[Q]$, $length[Q]$) $= head[Q]$

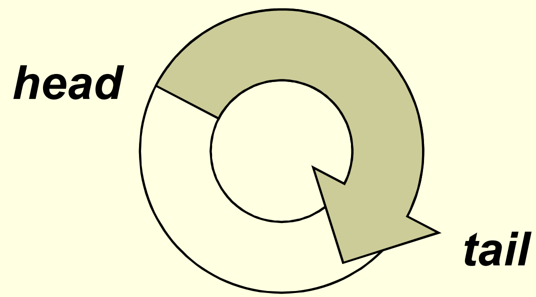
Next(i, n)

return (i mod n) + 1

<u>לתשומת לב:</u> Next היא שגרה לשימוש פנימי בלבד ואי אפשר לקרוא לה מבחוץ

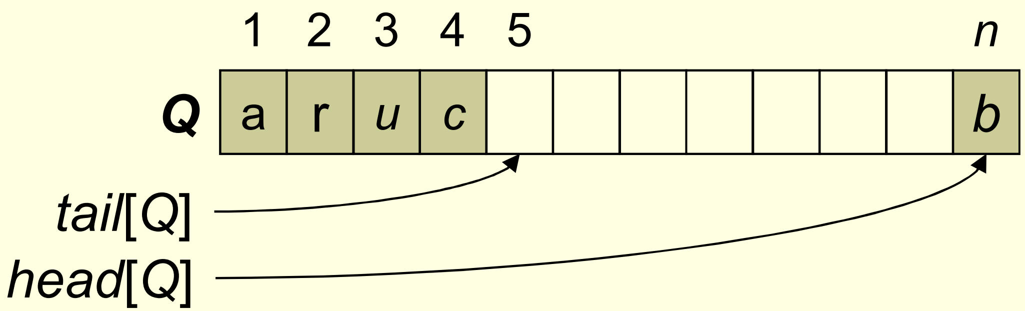
QueueInit(Q)

$tail[Q] \leftarrow 1$
$head[Q] \leftarrow 1$

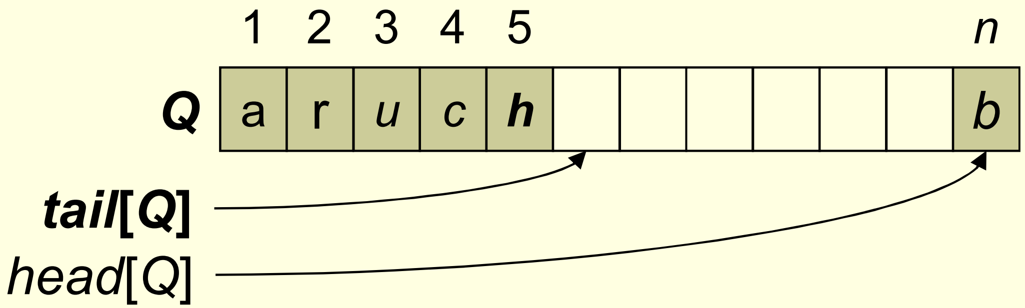


תור – דוגמה

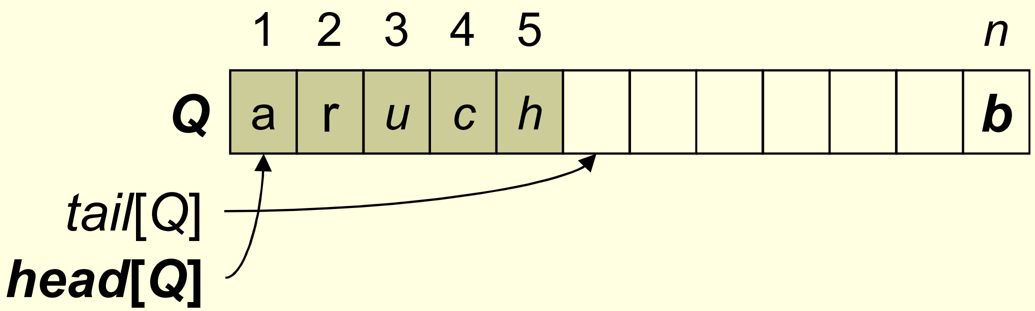
מצב התחלתי



המצב לאחר קריאה ל-Enqueue(*Q*, *h*)



המצב לאחר קריאה ל-Dequeue(*Q*)

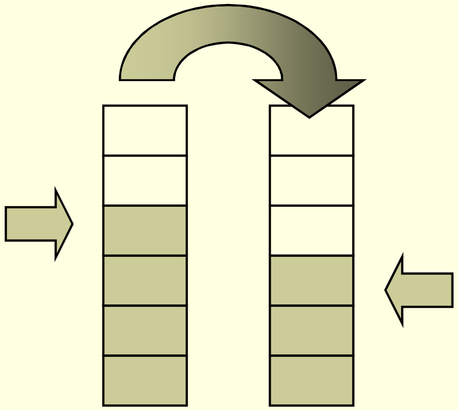


מחסניות ותורים – תרגול

- (תרגיל 10.1-6)

הראו כיצד ניתן לממש תור באמצעות שתי מחסניות.

נתחו את זמן הריצה של הפעולות על התור.



פתרון תרגיל 10.1-6
מימוש תור באמצעות שתי מחסניות

- נשתמש בשתי מחסניות:
  - *tail*[Q] מחזיקה את כל האיברים
  - *head*[Q] מחסנית עזר להוצאה

```
QueueFull(Q)
   return StackFull(tail[Q])

QueueEmpty(Q)
   return StackEmpty(tail[Q])

MoveStack(src, dst)
   while not StackEmpty(src)
       do Push(dst, Pop(src))

QueueInit(Q)
   StackInit(tail[Q])
   StackInit(head[Q])

Enqueue(Q, x)
   if QueueFull(Q)
       then error "queue overflow"
   Push(tail[Q], x)

Dequeue(Q)
   if QueueEmpty(Q)
       then error "queue underflow"
   MoveStack(tail[Q], head[Q])
   x ← Pop(head[Q])
   MoveStack(head[Q], tail[Q])
   return x
```

פתרון תרגיל 10.1-6 (המשך)
מימוש משופר

- נשתמש בשתי מחסניות:
  - *tail*[Q] מחסנית להכנסה לתור
  - *head*[Q] מחסנית להוצאה מהתור

QueueFull(*Q*)
1. **return** StackFull(*tail*[*Q*]) **and**
   not StackEmpty(*head*[Q])

QueueEmpty(*Q*)
1. **return** StackEmpty(*tail*[*Q*]) **and**
   StackEmpty(*head*[*Q*])

MoveStack(*src*, *dst*)
1. **while not** StackEmpty(*src*)
2. **do** Push(*dst*, Pop(*src*))

QueueInit(*Q*)
1. StackInit(*tail*[*Q*])
2. StackInit(*head*[*Q*])

Enqueue(*Q*, *x*)
1. **if** QueueFull(*Q*)
2. **then error** "queue overflow"
3. **if** StackFull(*tail*[*Q*]) **and**
   StackEmpty(*head*[*Q*])
4. **then** MoveStack(*tail*[*Q*], *head*[*Q*])
5. Push(*tail*[*Q*], *x*)

Dequeue(*Q*)
1. **if** QueueEmpty(*Q*)
2. **then error** "queue underflow"
3. **if** StackEmpty(*head*[*Q*])
4. **then** MoveStack(*tail*[*Q*], *head*[*Q*])
5. **return** Pop(*head*[*Q*])

האוניברסיטה הפתוחה

רשימות מקושרות

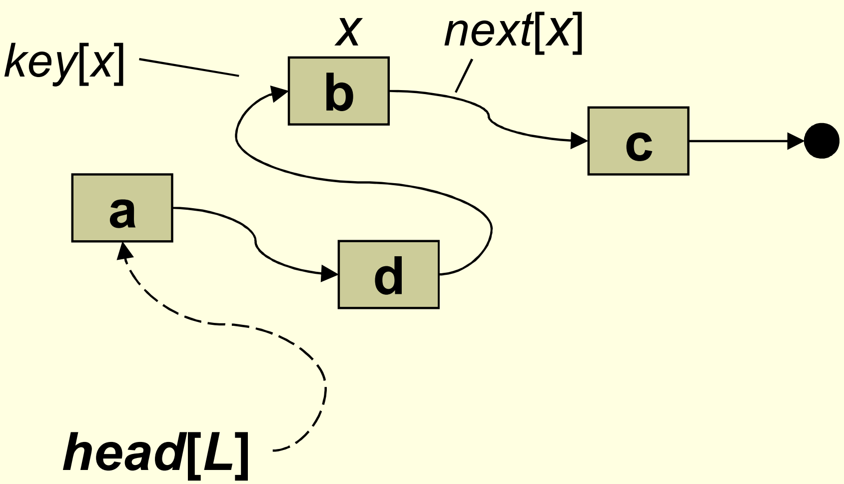
- רשימה מקושרת (linked list) – האיברים משורשרים זה לזה

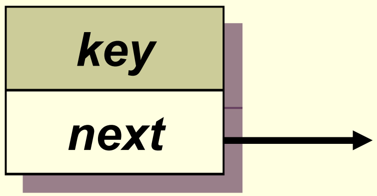
- דוגמאות לפעולות על רשימה:
  - חיפוש איבר בעל מפתח נתון
  - הכנסת איבר חדש לתוך הרשימה
  - מחיקת איבר מן הרשימה
  - החזרת אורך הרשימה

- סוגים שונים של רשימות:
  - כיוון המצביעים: חד-מקושרת או דו-מקושרת
  - סדר המפתחות: ממוינת או לא ממוינת
  - צורת הרשימה: מעגלית או קווית
  - שימוש בזקיפים (sentinels) – מאפשר לפשט את הקוד

רשימה חד-מקושרת

- רשימה חד-מקושרת (singly-linked list)
  - לכל איבר *x* יש מצביע *next*[*x*] לאיבר שאחריו
    - *next*[*x*] = nil אם *x* אחרון ברשימה
  - לרשימה *L* יש מצביע *head*[*L*] לאיבר הראשון
    - *head*[*L*] = nil אם הרשימה ריקה

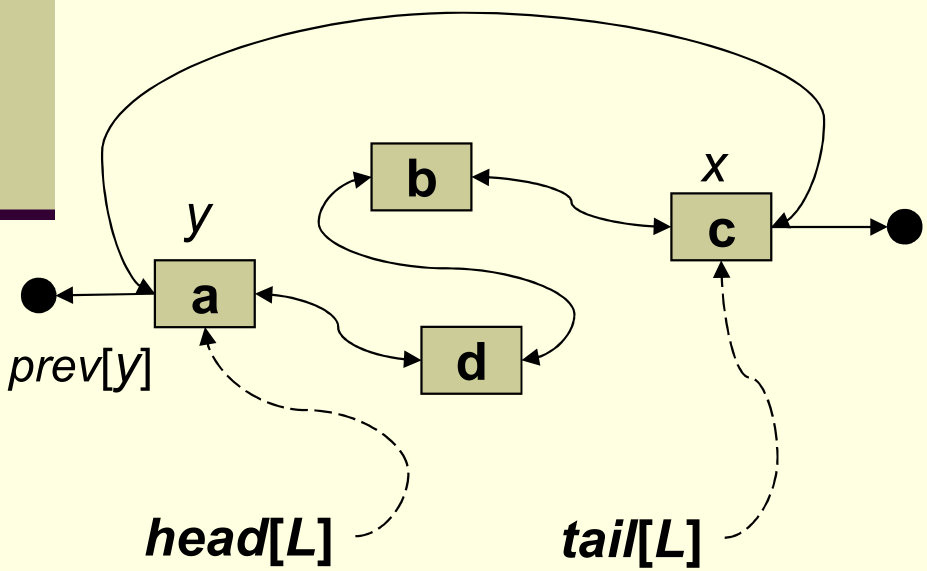


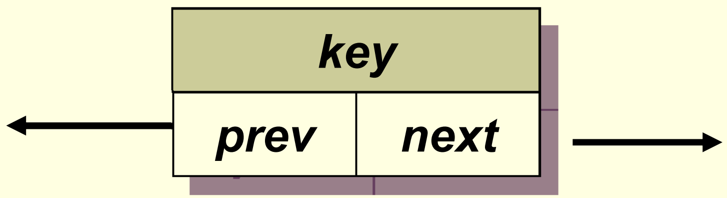


האוניברסיטה הפתוחה

רשימה דו-מקושרת

- רשימה דו-מקושרת (doubly-linked list)
  - כמו רשימה חד-מקושרת, ובנוסף:
  - לכל איבר *x* יש מצביע *prev*[*x*] לאיבר שלפניו
    - *prev*[*x*] = nil אם *x* ראשון ברשימה
  - לרשימה *L* יש מצביע *tail*[*L*] לאיבר האחרון
    - *tail*[*L*] = nil אם הרשימה ריקה
- רשימה מעגלית
  - רשימה דו-מקושרת, אבל המצביעים של איברי הקצה "סוגרים" מעגל
    - *next*[*tail*[*L*]] = *head*[*L*]
    - *prev*[*head*[*L*]] = *tail*[*L*]





חיפוש איבר ברשימה ממוינת

- חיפוש האיבר הראשון שמפתחו *k* ברשימה (דו-)מקושרת *L* ממוינת

SortedListSearch(*L*, *k*)

$x \leftarrow head[L]$

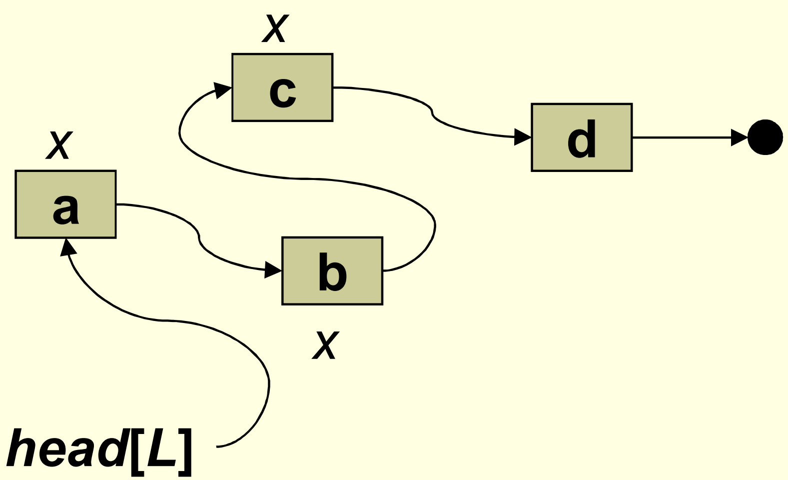
while $x \neq$ **nil and** $key[x] < k$

  do $x \leftarrow next[x]$

if $x =$ **nil or** $key[x] > k$

  then return nil

  else return x



דוגמה: SortedListSearch(*L*, “c”)

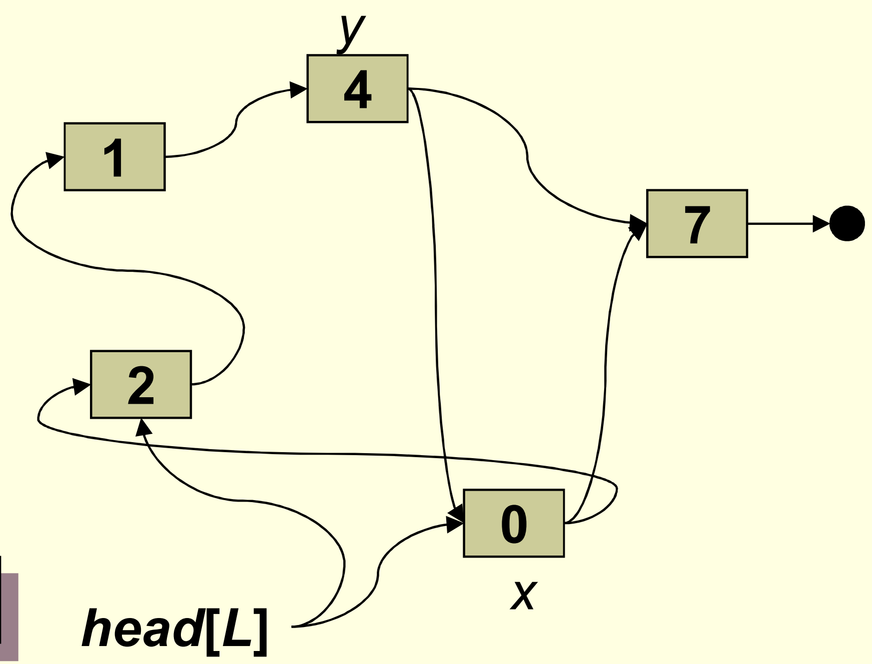
האוניברסיטה הפתוחה

הכנסה לרשימה במקום נתון

- הכנסת איבר *x* לרשימה חד-מקושרת אחרי איבר *y*

SinglyListInsertAfter(*L*, *x*, *y*)

```
if y = nil ► x becomes the head
    then next[x] ← head[L]
         head[L] ← x
    else next[x] ← next[y]
         next[y] ← x
```



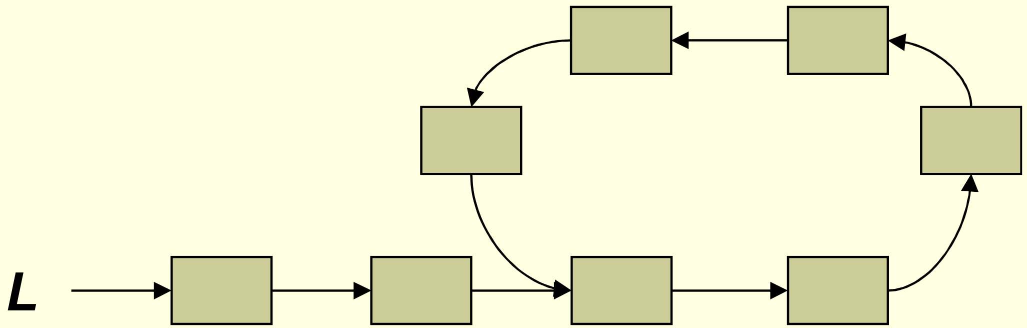
דוגמה: SinglyListInsertAfter(*L*, x, y)

רשימה חד-מקושרת – תרגול

- רשימת "שבלול" (lollipop)

רשימת שבלול היא רשימה חד-מקושרת שהאיבר האחרון שלה מצביע אל איבר כלשהו בתוך הרשימה.

הציעו אלגוריתם הבודק בזמן לינארי ובזיכרון קבוע אם רשימה חד-מקושרת נתונה היא רשימת שבלול.



האוניברסיטה הפתוחה

פתרון

בדיקה אם רשימה חד-מקושרת היא שבלול

- נשתמש בשני מצביעים הרצים במהירויות שונות לאורך הרשימה
  - מצביע "איטי" p המתקדם צעד אחד בכל איטרציה
  - מצביע "מהיר" q המתקדם שני צעדים בכל איטרציה
  - שני המצביעים "יוצאים למרוץ" יחד
- מדוע האלגוריתם נכון?
  - אם זו רשימת שבלול, אז p ו-q ייפגשו מתישהו בתוך המעגל, ובשורה 7 יוחזר הערך true
  - אחרת, המצביע q יגיע לסוף הרשימה, תתבצע יציאה מהלולאה ובשורה 8 יוחזר הערך false

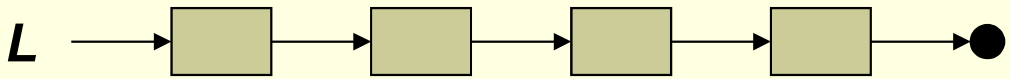
```
IsLollipop(L)
1.  p ← head[L]
2.  q ← head[L]
3.  while next[q] ≠ nil
        and next[next[q]] ≠ nil
4.          do p ← next[p]
5.             q ← next[next[q]]
6.             if q = p
7.                then return true
8.  return false
```

רשימה חד-מקושרת – תרגול

- (תרגיל 10.2-3)

ממשו תור באמצעות רשימה חד-מקושרת *L*.

הפעולות Enqueue ו- Dequeue צריכות להתבצע גם עתה בזמן $O(1)$.



פתרון תרגיל 10.2-3
מימוש תור באמצעות רשימה חד-מקושרת

- נשתמש במצביע לאיבר האחרון ברשימה
  - *head*[*L*] מצביע לאיבר הראשון
  - *tail*[*L*] מצביע לאיבר האחרון

QueueInit(*L*)
```
head[L] ← nil
tail[L] ← nil
```

QueueEmpty(*L*)
```
return head[L] = nil
```

Enqueue(*L*, *x*)
```
next[x] ← nil
if QueueEmpty(L)
   then head[L] ← x
   else next[tail[L]] ← x
tail[L] ← x
```

Dequeue(*L*)
```
if QueueEmpty(L)
   then error "underflow"
x ← head[L]
head[L] ← next[x]
if head[L] = nil
   then tail[L] ← nil
next[x] ← nil
return x
```

רשימה חד-מקושרת – תרגול

- תרגיל 10.2-7

כתבו שגרה לא רקורסיבית שהופכת את סדר האיברים ברשימה חד-מקושרת בת n איברים. זמן הריצה O(n).

השגרה יכולה להשתמש לכל היותר בכמות קבועה של זיכרון, בנוסף למקום הנדרש לאחסון הרשימה עצמה.