מבני נתונים ומבוא לאלגוריתמים
מפגש הנחיה מס' 8

מדעי המחשב, קורס מס' 20407

סמסטר 2016ב

מנחה: ג'ון מרברג

מה ראינו במפגש הקודם?

- מיונים בזמן לינארי
  - מיון-מנייה
  - מיון-בסיס
  - מיון-דלי
- מבני נתונים בסיסיים
  - מחסניות ותורים
  - רשימות מקושרות

מפגש שמיני

- נושאי השיעור
  - פרק 11 בספר – טבלאות גיבוב (Hash Tables)
    - פעולות מילוניות
    - מיעון ישיר
    - טבלאות גיבוב
    - פתרון התנגשויות
      - שרשור
      - מיעון פתוח
    - תרגול: דוגמאות עם שילוב מספר מבני נתונים
  - פרק 12 – עצי חיפוש בינארים (מבוא)
    - סריקה של עץ בינארי לרוחב ולעומק
    - הגדרת עץ חיפוש בינארי

מבוסס על מצגת של ברוך חייקין ואיציק בייז

פעולות מילוניות

- דרושה טבלה דינמית התומכת ב"פעולות מילוניות": הכנסה, מחיקה, וחיפוש של איברים לפי מפתח
  - כל פעולה מתייחסת למפתח בודד
  - אין סדר בין המפתחות בטבלה
  - נחוצה סיבוכיות "טובה" במקרה הממוצע
- שימושים לדוגמא
  - ספירת מופעי מלים בטקסט נתון
  - טבלת סמלים שמחזיק מהדר
- סימונים
  - מרחב המפתחות: U
    - *גודל המרחב* $|U|$ אינו מוגבל
  - גודל הטבלה: m
    - *זהו מספר המקומות (תאים, slots) בטבלה*
  - מספר המפתחות המאוחסנים בטבלה: n
  - מקדם העומס: $\alpha = n/m$

שיטות מיעון לטבלה

- ייצוג המפתחות
  - בהינתן מרחב מפתחות, U, ניתן (בד"כ) לייצג כל מפתח כמספר טבעי, באמצעות תרגום חד-חד ערכי פשוט
  - למשל: מפתח שהוא מחרוזת תווים ניתן לייצוג כמספר בבסיס 256
    - כל ספרה היא קוד ASCII של תו
    - לדוגמה: $Ab2 = 65*256^2 + 98*256 + 50 = 4284978$
- מיעון ישיר (Direct Access)
  - האיבר שמפתחו (הטבעי) k, ורק הוא, יוכנס לתא ה-k בטבלה
  - זמן ריצה: כל הפעולות מתבצעות בזמן $O(1)$ *במקרה <u>הגרוע</u>*
  - גודל הטבלה: $m \geq |U|$
- מיעון באמצעות גיבוב (Hashing)
  - נתונה פונקציה $h()$ ה"מגבבת" מפתחות לתאי הטבלה

    $h : U \rightarrow \mathbf{N}_m = \{0, \ldots, m-1\}$
  - האיבר שמפתחו k יוכנס לתא ה-$h(k)$ בטבלה
    - יש צורך לטפל ב"התנגשויות" – מקרים שבהם בתא נמצא כבר מפתח אחר
  - זמן ריצה: בהנחה שפונקצית הגיבוב $h()$ נותנת פיזור "אחיד" של המפתחות בטבלה, הפעולות מתבצעות בזמן $O(\alpha)$ *<u>בממוצע</u>*
  - כאשר n ידוע מראש, נוכל להקצות טבלה בגודל $m=\Theta(n)$
    - במילים אחרות, נשאף למקדם עומס $\alpha = \Theta(1)$

האוניברסיטה הפתוחה

מיעון ישיר (Direct Access)

- נניח בה"כ שהמפתחות שייכים למרחב $U = \mathbf{N}_m = \{0, \ldots, m-1\}$
- אם m קטן מספיק, ניתן להשתמש במיעון ישיר
- נגדיר טבלה $T[0..m-1]$
  - התא $T[k]$ מכיל מצביע לאיבר שמפתחו k, או את המצביע nil
  - יש לאתחל את כל הטבלה ל-nil
  - לעתים ניתן לאחסן את האיבר עצמו בטבלה

DirectAccessSearch(T, k)

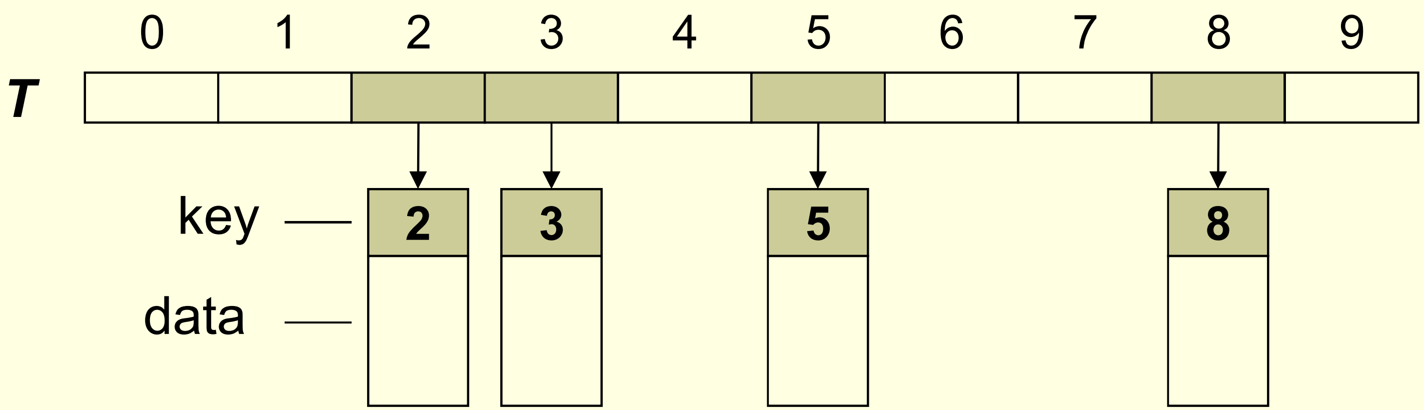
1. **return** $T[k]$

DirectAccessInsert(T, x)

1. $T[x.key] \leftarrow x$

DirectAccessDelete(T, k)

1. $T[k] \leftarrow$ **nil**



תרגיל 11.1-4:
מיעון ישיר על טבלה עצומה בגודלה

- רוצים לממש מילון עם מיעון ישיר על מערך (טבלה) בעל מספר עצום של איברים.
  בהתחלה תאי המערך עשויים להכיל "זבל", אך אתחול המערך כולו
  על-ידי כתיבת ערכים בכל התאים אינו מעשי, בשל גודל המערך.

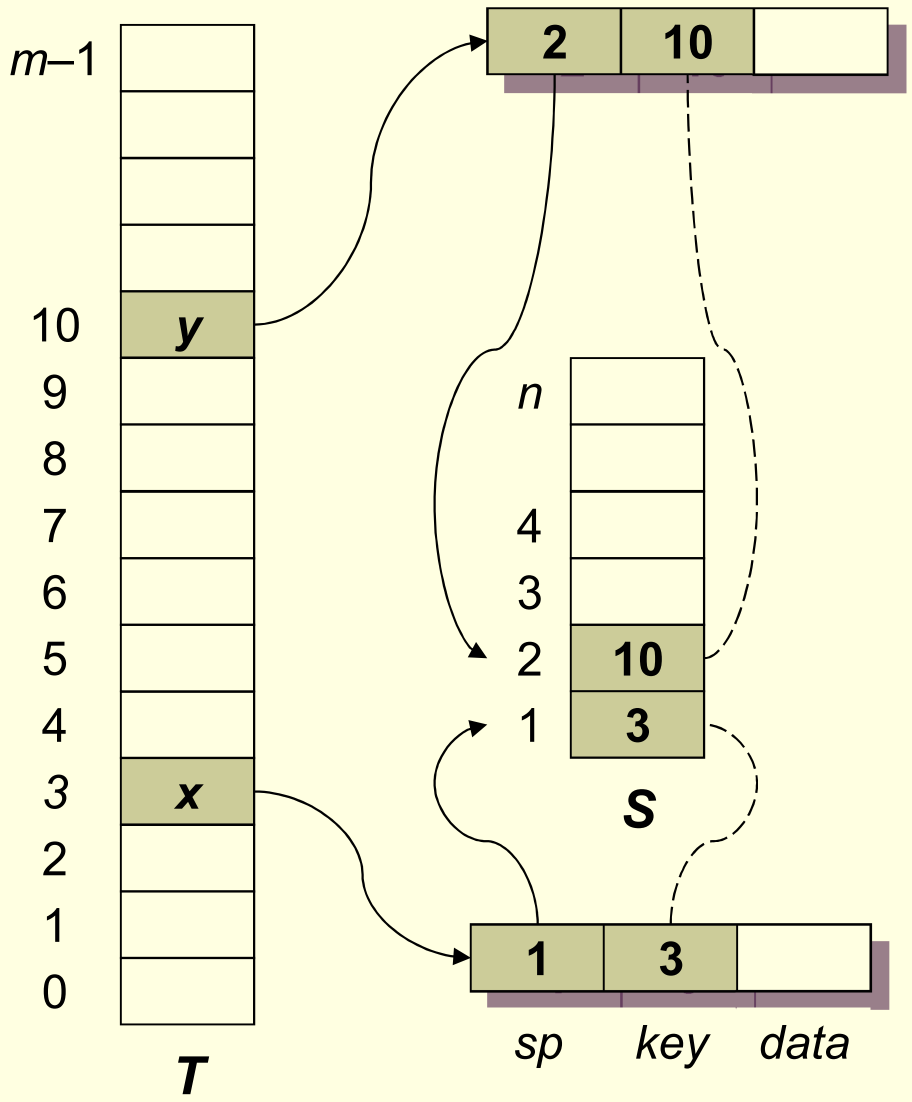
- עליכם לכתוב אלגוריתמים עבור הפעולות המילוניות במיעון ישיר במערך העצום.
  הפעולות הנדרשות הן: אתחול המערך, חיפוש, הכנסה ומחיקה.

  על כל הפעולות לרוץ בזמן $O(1)$.

  הניחו כי כל איבר במערך תופס מקום בגודל $O(1)$.

<u>רמז</u>: כדי לקבוע אם הערך בתא נתון במערך הוא ערך תקף, השתמשו בנוסף
למערך גם במחסנית. מספר האיברים שנמצאים במחסנית שווה למספר המילים
המאוחסנות בפועל במערך. <u>הניחו</u> שאפשר לגשת ישירות לכל איבר במחסנית.

פיתרון תרגיל 11.1-4
מיעון ישיר למערך עצום בגודלו



- נשתמש במחסנית עזר S לאחסון המפתחות
  - $S[1..top[S]]$ הם המפתחות של האיברים המאוחסנים כרגע במערך T
  - המחסנית מאפשרת גישה ישירה לאיבריה
- לכל איבר x בטבלה T שני שדות:
  - *x.key הוא המפתח של האיבר*
  - $x.sp$ הוא האינדקס של התא במחסנית בו מאוחסן המפתח $x.key$
- מתקיימות השמורות הבאות:
  - לכל איבר שמפתחו k המוחזק במערך T מתקיים: $S[T[k].sp] = k$
  - לכל $0 < i \leq top[S]$ מתקיים $T[S[i]].sp = i$

פיתרון תרגיל 11.1-4 (המשך)
מיעון ישיר למערך עצום בגודלו

HugeInsert(T, x)
1. **if** HugeSearch$(T, x.key)$ = **nil**
2. **then** Push$(S, x.key)$
3. $x.sp \leftarrow top[S]$
4. $T[x.key] \leftarrow x$

HugeDelete(T, k)
1. $x \leftarrow$ HugeSearch(T, k)
2. If $x \neq$ **nil**
3. **then** $skey \leftarrow$ Pop(S)
4. **if** $skey \neq x.key$
5. **then** $T[skey].sp \leftarrow x.sp$
6. $S[x.sp] \leftarrow skey$

HugeInit(m)
1. $top[S] \leftarrow 0$

HugeSearch(T, k)
1. $p \leftarrow T[k].sp$
2. **if** $p \leq 0$ **or** $p > top[S]$ **or** $S[p] \neq k$
3. **then return nil**
4. **return** $T[k]$

האוניברסיטה הפתוחה

טבלאות גיבוב (Hash Tables)

- טבלת גיבוב
  - כדי לחסוך במקום, הטבלה T תהיה "קטנה" (יחסית למרחב המפתחות) $m << |U|$
  - נגדיר פונקצית גיבוב h

    $$h : U \rightarrow \mathbf{N}_m = \{0, \ldots, m-1\}$$
  - המפתח k מגובב (hashed) לתא $T[h(k)]$
- התנגשויות (Collisions)
  - אם $k_1 \neq k_2$ אבל $h(k_1) = h(k_2)$, נאמר שהמפתחות <u>מתנגשים</u>
    - יש לתת פתרון למצבי התנגשות
  - כדי להפחית את מספר ההתנגשויות, נרצה שהפונקציה h תפזר את המפתחות באופן אחיד בין כל התאים בטבלה
  - בכל מקרה קיים סיכוי מסויים להתנגשות בין מפתחות
    - "יריב מרושע" המכיר את h יכול ליצור סדרת מפתחות שכולם מתנגשים!
- פתרון התנגשויות
  - באמצעות שרשור (chaining)
  - באמצעות מיעון פתוח (open addressing)

האוניברסיטה הפתוחה

שרשור (Chaining)

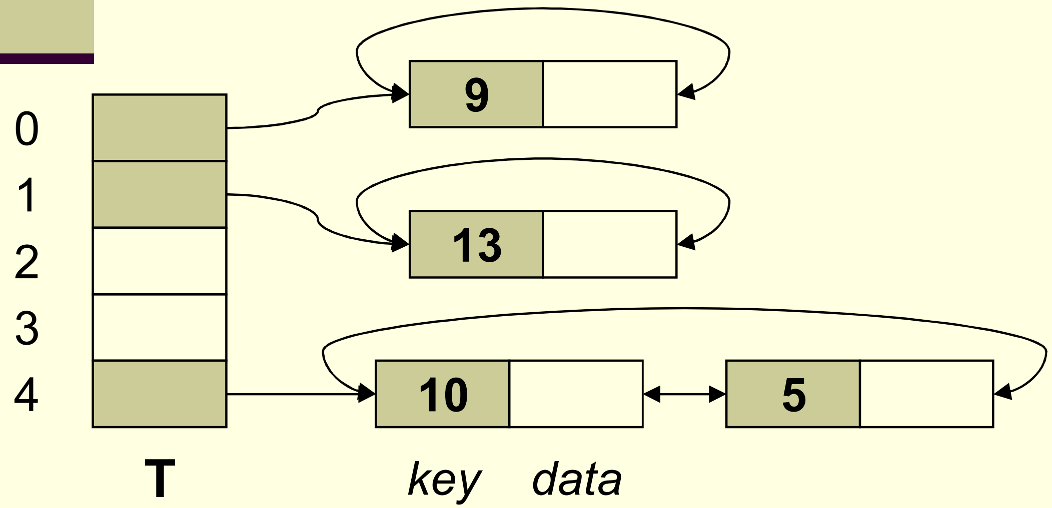
- כל תא בטבלה מכיל מצביע לרשימה מקושרת של האיברים שגובבו אליו
  - אופציונלית, הרשימה דו-מקושרת ו/או מעגלית
- *הפונקציה h* מבצעת גיבוב **אחיד ופשוט**:
  - ההסתברות שהמפתח *k* יגובב לתא מסוים היא זהה לכל התאים: $1/m$
  - הגיבוב של המפתח *k* אינו תלוי בגיבוב של מפתחות אחרים
  - החישוב של $h(k)$ מתבצע בזמן $O(1)$
- הכנסת איבר לרשימה
  - הכנסה לראש הרשימה בזמן $O(1)$ במקרה <u>הגרוע</u>
- חיפוש איבר בהינתן מפתח
  - זמן <u>ממוצע</u>: $\Theta(1+\alpha)$
    (לפי משפטים 11.1, 11.2 ב-CLRS)
  - במקרה <u>הגרוע</u> (כאשר מרבית האיברים משורשרים בתא אחד): זמן $O(n)$
- הוצאת איבר
  - בהינתן מצביע לאיבר: זמן $O(1)$ במקרה <u>הגרוע</u>
  - בהינתן מפתח: זמן $\Theta(1+\alpha)$ <u>בממוצע</u>

HashSearch(T, k)
► returns a pointer, nil if not found
1. **return** ListSearch($T[h(k)]$, k)

HashInsert(T, x)
1. ListInsert($T[h(x.key)]$, x)

HashDelete(T, x)
1. ListDelete($T[h(x.key)]$, x)

HashDeleteKey(T, k)
1. $x \leftarrow$ HashSearch(T, k)
2. HashDelete(T, x)



פונקציות גיבוב (Hash Functions)

- פונקצית גיבוב
  - ממפה את מרחב המפתחות לאינדקסים בטבלה $h : U \rightarrow \mathbf{N}_m$
    - נניח בה"כ שהמפתחות הם מספרים טבעיים (כלומר, $U = \mathbf{N}$)
- פונקצית גיבוב "טובה"
  - מקיימת (בקירוב) את הנחת הגיבוב האחיד והפשוט
    - קשה לבדוק, כי בד"כ התפלגות המפתחות לא ידועה מראש
    - פתרון חלקי: מפתחות "קרובים" ימופו לאינדקסים "מרוחקים"
- שיטות לבנית פונקצית גיבוב
  - שיטת החילוק
  - שיטת הכפל
  - גיבוב אוניברסלי

שיטת החילוק (Division Method)

- שיטת החילוק
  - פונקצית הגיבוב: $h(k) = k \bmod m$
  - מומלץ לבחור את m (גודל הטבלה) כמספר ראשוני שאינו קרוב לחזקה של 2
    - כי כאשר $m=2^p$, אז $h(k)$ הוא p הביטים הנמוכים של k,
      כלומר פונקצית הגיבוב אינה משתמשת בכל הסיביות של המפתח
- דוגמה:
  - מעוניינים להקצות טבלת גיבוב עבור $n = 2000$ מפתחות, עם פתרון התנגשויות ע"י שרשור, כך שבממוצע יהיו 3 איברים בכל רשימה מקושרת
  - האורך הממוצע של רשימה (בהנחה שהגיבוב אחיד ופשוט) הוא $\alpha = n/m$
  - נבחר $m = 701$
    - m ראשוני, וגדול מ-2000/3=667
    - רחוק מחזקה של 2 (512 או 1024)

שיטת הכפל (Multiplication Method)

- שיטת הכפל
  - פונקציית הגיבוב: $h(k) = \lfloor m((kA) \bmod 1) \rfloor$, כאשר $0 < A < 1$ קבוע
    - כופלים את k ב-A ולוקחים מהתוצאה את החלק שאחרי הנקודה
    - כופלים ערך זה ב-m, ומעגלים את התוצאה כלפי מטה
    - בד"כ בוחרים m שהוא חזקה של 2 כדי לפשט את המימוש במחשב בינרי
  - השיטה עובדת עם כל A, אך הבחירה המיטבית תלויה במאפייני המפתחות

- דוגמה:

$$m = 10000$$

$$A = (\sqrt{5}-1)/2 = 0.6180339887...$$

$$h(123456) = \lfloor 10000 \cdot ((123456 \cdot 0.6180339887...) \bmod 1) \rfloor$$
$$= \lfloor 41.151... \rfloor = 41$$

גיבוב אוניברסלי (Universal Hashing)

- יריב מרושע המכיר את h יכול לבחור מפתחות כך שכולם יגובבו לאותו תא!
- כדי למנוע זאת, נבנה "אוסף אוניברסלי" של פונקציות גיבוב ונבחר מתוכו <u>בזמן ריצה</u> פונקצית גיבוב *אקראית*
- הגדרה: אוסף סופי H של פונקציות גיבוב (עבור U ו-m נתונים) הוא **אוניברסלי** אם עבור כל זוג מפתחות $x \neq y$, יש באוסף לכל היותר $|H|/m$ פונקציות המקיימות $h(x) = h(y)$
- משפט 11.3 ב-CRLS:

תהי h פונקציית גיבוב שנבחרה מתוך אוסף אוניברסלי של פונקציות גיבוב, ומשמשת לגיבוב n מפתחות בטבלה בגודל m, שפתרון ההתנגשויות בה נעשה ע"י שרשור.

אזי תוחלת אורך הרשימה שאליה מגובב מפתח כלשהו k היא:

- α אם k לא נמצא בטבלה
- $\alpha + 1$ אם k כבר נמצא בטבלה

מיעון פתוח (Open Addressing)

- כל מפתח מאוחסן בתא נפרד בטבלה
  - לכן, $\alpha = n/m \le 1$
- פתרון התנגשויות: אם התא תפוס, ננסה תא אחר...
  - פונקציית הגיבוב מקבלת כפרמטר גם את מספר הנסיונות הקודמים, ומחזירה את התא הבא לבדיקה: $h : U \times \mathbf{N}_m \rightarrow \mathbf{N}_m$
  - סדרת הבדיקות $\{h(k, 0), h(k, 1), ..., h(k, m–1)\}$ חייבת להיות תמורה של הסדרה $\{0, 1, ..., m–1\}$
- נרצה ש-h תבצע **גיבוב אחיד**:
  - לכל אחת מ- $m!$ התמורות של $\{0, 1, ..., m–1\}$ יש סיכוי שווה להיות סדרת הבדיקה של כל מפתח נתון
  - גיבוב אחיד קשה למימוש, לכן משתמשים בקירובים:
    - בדיקה לינארית (linear probing) – מאפשרת m תמורות
    - בדיקה ריבועית (quadratic probing) – מאפשרת m תמורות
    - גיבוב כפול (double hashing) – מאפשרת m^2 תמורות

בדיקה לינארית (Linear Probing)

$h'(k) = k \bmod 17$

$h(k, i) = (h'(k) + i) \bmod 17$

Input: 3, 5, 8, 13, 21, 55, 76, 131, 207, 338

	k	h(k)
0		
1		
2		
3	**3**	3
4	**21**	4
5	**5**	5
6	**55**	4
7	**207**	3
8	**8**	8
9	**76**	8
10		
11		
12	**131**	12
13	**13**	13
14		
15	**338**	15
16		

207	3
207	4
207	5
207	3

76	8

- פונקצית הגיבוב:
  $h(k, i) = (h'(k) + i) \bmod m$
  - h' היא פונקצית גיבוב בסיסית "רגילה"
- מתקבלת סדרת מיקומים רציפה (מעגלית במודולו m)

$$\{h'(k),\ h'(k)+1,\ \ldots,\ h'(k)+m-1\}$$

- יש רק m סדרות כאלה
- השיטה קלה למימוש
- בעיית ההצטברות הראשונית
  - אם לתא פנוי קודמים i תאים תפוסים, הסיכוי ש- i יהיה התא הבא שיתמלא הוא $(i+1)/m$ ולא $1/m$
  - נוצרים רצפים ארוכים של תאים תפוסים

מימוש הפעולות במיעון פתוח

```
OpenHashInsert(T, x)
i ← 0; k ← x.key
repeat j ← h(k, i)
        if T[j].key = nil
            then T[j] ← x
                return j
            else i ← i + 1
until i = m
return –1 ► hash table overflow
```

- אם במחיקה נציב $T[i].key \leftarrow$ **nil**, זה עלול לקטוע את סדרת הבדיקות בחיפוש ערך כלשהו שנמצא בטבלה.
- במקרה זה HashSearch תחזיר תשובה שגוייה, שהערך לא קיים.

```
OpenHashSearch(T, k)
i ← 0
repeat j ← h(k, i)
        if T[j].key = k
            then return j
            else i ← i + 1
until T[j].key = nil or i = m
return –1
```

```
OpenHashDelete(T, k)
i ← HashSearch(T, k)
if (i ≥ 0)
    then T[i].key ← ► Problem!...
```

מימוש הפעולות במיעון פתוח

OpenHashInsert(*T*, *x*)

```
i ← 0; k ← x.key
repeat j ← h(k, i)
        if T[j].key = (nil or DELETED)
            then T[j] ← x
                 return j
            else i ← i + 1
until i = m
return –1 ► hash table overflow
```

OpenHashSearch(*T*, *k*)

```
i ← 0
repeat  j ← h(k, i)
        if T[j].key = k
            then return j
            else i ← i + 1
until T[j].key = nil or i = m
return –1
```

OpenHashDelete(*T*, *k*)

```
i ←HashSearch(T, k)
if (i ≥0)
   then T[i].key ← DELETED
```

- עם השימוש ב- *DELETED*, זמן החיפוש בטבלה כבר אינו פונקציה רק של מקדם העומס α, אלא תלוי גם בהיסטוריה
- לפיכך, כאשר יש צורך במחיקות, עדיף להשתמש בשירשור

בדיקה ריבועית (Quadratic Probing)

$h'(k) = k \bmod 17$

$h(k, i) = (h'(k) + i + i^2) \bmod 17$

Input: 3, 5, 8, 13, 21, 55, 76, 131, 207, 338

0				
1				
2				
3	**3**	3	**207**	3
4	**21**	4	**55**	4
5	**5**	5	**207**	3
6	**55**	4		
7				
8	**8**	8	**76**	8
9	**207**	3		
10	**76**	8		
11				
12	**131**	12		
13	**13**	13		
14				
15	**338**	15		
16				

- פונקצית הגיבוב:
  $h(k, i) = (h'(k) + ai + bi^2) \bmod m$
  - קבועים שלמים $a, b \neq 0$
- מתקבלת סדרת מיקומים לא רציפה
  - יש רק m סדרות כאלה
- בעיית ההצטברות המשנית
  - אם עבור שני מפתחות $k_1 \neq k_2$ מתקיים $h'(k_1) = h'(k_2)$, אז לשני המפתחות יש בדיוק אותה סדרת בדיקה.

גיבוב כפול (Double Hashing)

$h_1(k) = k \bmod 17$
$h_2(k) = k \bmod 16 + 1$
$h(k, i) = (h_1(k) + ih_2(k)) \bmod 17$
Input: 3, 5, 8, 13, 21, 55, 76, 131, 207, 338

0	**76**	8,13		
1				
2	**207**	3,16		
3	**3**	3,4	**207**	3,16
4	**21**	4,6	**76**	8,13
5	**5**	5,6		
6				
7				
8	**8**	8,9	**76**	8,13
9				
10				
11				
12	**55**	4,8	**131**	12,4
13	**13**	13,14		
14				
15	**338**	15,3		
16	**131**	12,4		

- פונקצית הגיבוב מורכבת משתי פונקציות גיבוב: h_1, h_2

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

  - הערך $h_2(k)$ צריך להיות זר ל-m
  - למשל: נבחר m ראשוני, ונבנה את h_2 כך שלכל מפתח k יתקיים $0 < h_2(k) < m$
- מתקבלת סדרת מיקומים לא רציפה
  - יש $\Theta(m^2)$ סדרות כאלה

סיבוכיות של מיעון פתוח

- כל מפתח מאוחסן בתא נפרד בטבלה
  - לפיכך, $\alpha = n/m \leq 1$
- הנחת הגיבוב האחיד
  - לכל אחת מ- $m!$ התמורות של $\{0, 1, ..., m–1\}$ יש סיכוי שווה להיות סדרת הבדיקה של כל מפתח נתון
- הסיבוכיות במיעון פתוח

משפטים 11.6, 11.8 ב-CLRS:

  - תוחלת מספר הבדיקות בחיפוש <u>כושל</u> היא לכל היותר $1/(1–\alpha)$
  - תוחלת מספר הבדיקות בחיפוש <u>מוצלח</u> היא לכל היותר $(1/\alpha)\ln(1/(1–\alpha))$

תרגיל: מציאת זוג מספרים שסכומם נתון

נתון מערך A בגודל n *המכיל* מספרים כלשהם,
וכן נתון מספר z.

כתבו אלגוריתם המחזיר זוג מספרים ב-A שסכומם z.
אם לא קיים זוג מספרים כזה, האלגוריתם יחזיר nil.

על האלגוריתם לרוץ בתוחלת זמן $O(n)$.

האוניברסיטה הפתוחה

פתרון התרגיל:
מציאת זוג מספרים שסכומם נתון

```
FindPair (A, z)
► Input: array A of n numbers
         and a number z
► Output: pair of numbers (x, y)
          such that z = x + y, or nil
n ← length[A]
T ← new hash table with n slots
for i ← 1 to n
    do HashInsert(T, A[i])
for i ← 1 to n
    do x ← A[i]
       y ← z – x
       if HashSearch(T, y) ≠ nil
          then return (x, y)
return nil
```

- ראשית, נכניס את כל המספרים מ-A לטבלת גיבוב בגודל n עם שרשור
  - זמן הכנסת איבר $O(1)$
  - זמן הריצה $O(n)$
- נעבור שנית על כל איברי A.
  לכל מספר x ב-A נבדוק אם קיים בטבלת הגיבוב המספר $y = z - x$
  - תוחלת זמן החיפוש לכל איבר $O(1)$
  - תוחלת זמן הריצה $O(n)$

תרגיל: פיצול מערך עם זוגות

- נתון מערך A בגודל $2n$ *המכיל* מספרים שכולם שונים זה מזה, ופונקציה $f(x)$ שניתנת לחישוב בזמן $O(1)$.

כתבו אלגוריתם המפצל את A לשני מערכים $A1$ ו- $A2$ *בגודל* n,
כך שכל המספרים ב- $A1$ קטנים מכל המספרים ב- $A2$,
ולכל איבר x ב- $A1$ קיים איבר y ב- $A2$ כך ש- $y = f(x)$.

האלגוריתם מחזיר את זוג המערכים $A1$, $A2$.
אם לא ניתן למצוא פיצול כמתואר, האלגוריתם יחזיר nil.

על האלגוריתם לרוץ בתוחלת זמן $O(n)$.

פתרון התרגיל:
פיצול מערך עם זוגות

- נשתמש באלגוריתם Select למציאת החציון של המערך A, ונפצל סביבו את A לשני חצאי מערכים $A[1..n]$, $A[n+1..2n]$, כך שכל איבר ב- $A[1..n]$ קטן מכל איבר ב- $A[n+1..2n]$
  - זמן הריצה $O(n)$
- נכניס את כל איברי $A[n+1..2n]$ לטבלת גיבוב T בגודל n עם שרשור
  - תוחלת זמן הכנסת איבר $O(1)$
  - תוחלת זמן הריצה $O(n)$
- לכל איבר x ב- $A[1..n]$ נחשב $y = f(x)$ ונחפש את y בטבלה T
  - תוחלת זמן החיפוש של איבר $O(1)$
  - תוחלת זמן הריצה $O(n)$

PairPartition(A, f)

► *Input*: array A of $2n$ numbers and function f

► *Output*: pair of arrays of n numbers or nil

```
n ← length[A] / 2
Select(A, 1, 2n, n)
T ← new hash table with n slots
for j ← n + 1 to 2n
    do HashInsert(T, A[j])
for i ← 1 to n
    do if HashSearch(T, f(A[i])) = nil
          then return nil
return (A[1..n], A[n+1..2n])
```

תרגיל: מימוש מטמון (Cache)

האוניברסיטה הפתוחה

- דרוש מבנה נתונים עבור מטמון C בגודל n *איברים*, התומך בפעולות הכנסה וחיפוש בלבד.

  המטמון מאותחל ריק, ומתמלא אחרי n הכנסות.

  כדי להכניס איבר חדש כאשר המטמון מלא, על פעולת ההכנסה למחוק מהמטמון את אחד האיברים הקיימים, וכך לפנות מקום לאיבר החדש.

  פעולת ההכנסה בוחרת את האיבר שיימחק לפי אחת השיטות הבאות:
  - האיבר הכי ותיק (מדיניות FIFO).
  - האיבר שה"שימוש" האחרון בו היה לפני הכי הרבה זמן (מדיניות LRU), ("שימוש" הוא פעולת הכנסה או חיפוש של האיבר במטמון)

א. כתבו אלגוריתמים לפעולות ההכנסה והחיפוש במדיניות FIFO

ב. כתבו אלגוריתמים לפעולות ההכנסה והחיפוש במדיניות LRU

על כל האלגוריתמים לרוץ בתוחלת זמן $O(1)$.

<u>רמז</u>: מותר להשתמש בקריאות לפעולות ידועות על מבני נתונים מוכרים (כגון תור, רשימה (דו)-מקושרת, וכד').

פתרון התרגיל:
מטמון עם מדיניות FIFO

- הרעיון: המטמון C מורכב משני מבנים: טבלת גיבוב $T[C]$ עם שרשור, ותור $Q[C]$
  - הטבלה והתור שניהם בגודל n
  - טבלת הגיבוב מכילה את האיברים ומאפשרת חיפוש מהיר
  - התור מכיל רק מפתחות
  - כשמכניסים איבר לטבלה, מכניסים את המפתח שלו לסוף התור
  - לפיכך, המפתח של האיבר הוותיק ביותר נמצא בראש התור
  - התור מלא כשיש בו n מפתחות
  - כדי להכניס איבר חדש למטמון כאשר התור מלא, מוציאים מהתור את המפתח שבראש התור, ובאמצעותו מחפשים ומוחקים את האיבר בטבלה

```
FIFOCacheSearch(C, k)
    return HashSearch(T[C], k)

FIFOCacheInsert(C, x)
    if QueueFull(Q[C])
       then old ← Dequeue(Q[C])
            HashDeleteKey(T[C], old.key)
    young ← new queue item
    young.key ← x.key
    Enqueue(Q[C], young)
    HashInsert(T[C], x)
```

פתרון התרגיל:
מטמון עם מדיניות LRU

- הרעיון: המטמון מורכב משני מבנים: טבלת גיבוב $T[C]$ ורשימה דו-מקושרת $L[C]$ המתפקדת גם כתור
  - הטבלה בגודל n, והרשימה המקושרת תוגבל למקסימום n איברים
  - טבלת הגיבוב מכילה את האיברים ומאפשרת חיפוש מהיר
  - הרשימה המקושרת מכילה רק מפתחות
  - מפתח האיבר ה- least recently used נמצא בראש הרשימה
  - כשמכניסים איבר לטבלה, מכניסים את המפתח שלו לסוף הרשימה
  - כל איבר בטבלה מחזיק מצביע *lip* לאיבר המקביל ברשימה
  - כשמחפשים (ומוצאים) איבר בטבלה, מעבירים את המפתח שלו לסוף הרשימה
  - הרשימה "מלאה" כשיש בה n מפתחות
  - כדי להכניס איבר חדש למטמון כאשר הרשימה מלאה, מוציאים מהרשימה את המפתח שבראש הרשימה, ובאמצעותו מחפשים ומוחקים את האיבר בטבלה

LRUCacheSearch(C, k)

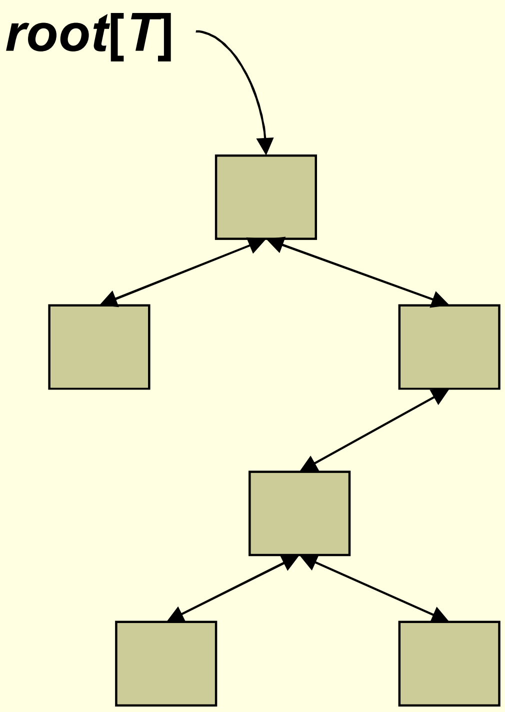
```
x ← HashSearch(T[C], k)
if x ≠ nil
   then ListRemove(L[C], x.lip)
        Enqueue(L[C], x.lip)
return x
```

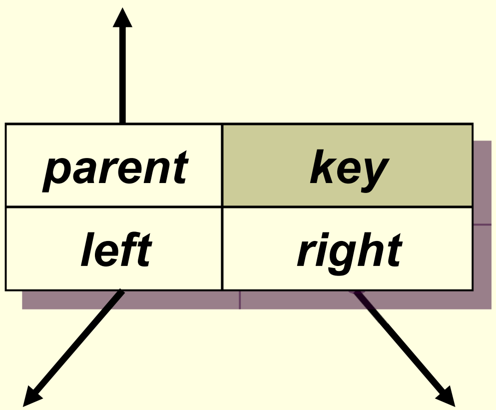
LRUCacheInsert(C, x)

```
if QueueFull(L[C])
   then lru ← Dequeue(L[C])
        HashDeleteKey(T[C], lru.key)
mru ← new list item
mru,key ← x.key
x.lip ← mru
Enqueue(L[C], mru)
HashInsert(T[C], x)
```

עצים בינריים

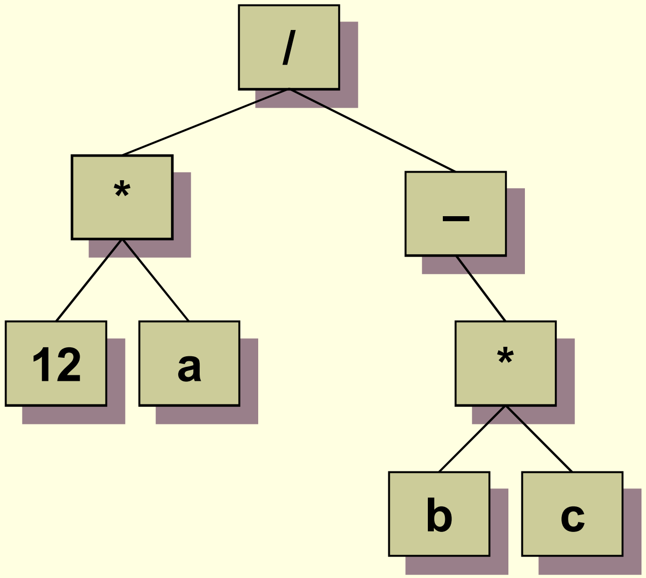
- דרגת כל צומת (מספר הבנים) לכל היותר 2
- לכל צומת *x* יש מצביעים *left*[*x*], *right*[*x*] לשני הבנים, ו-*parent*[*x*] לאב
  - parent[*x*] = nil אם *x* שורש העץ
  - right[*x*] = left[*x*] = nil אם *x* עלה
- לעץ *T* יש מצביע *root*[*T*] לשורש
  - root[*T*] = nil אם העץ ריק





סריקת עץ בינרי

$$\frac{12*a}{-(b*c)}$$



- סריקה לרוחב
  - הצמתים ברמה *i* לפני הצמתים ברמה *i* + 1
    - דוגמת פלט: / * – 12 a * b c
- סריקה לעומק Pre-order
  - שורש, תת-עץ שמאלי, תת-עץ ימני
    - דוגמת פלט: / * 12 a – * b c
- סריקה לעומק In-order
  - תת-עץ שמאלי, שורש, תת-עץ ימני
    - דוגמת פלט: 12 * a / – b * c
- סריקה לעומק Post-order
  - תת-עץ שמאלי, תת-עץ ימני, שורש
    - דוגמת פלט: 12 a * b c * – /

האוניברסיטה הפתוחה

אלגוריתם לסריקת עץ בינרי לרוחב

- מימוש איטרטיבי עם תור
  - מוציאים את האיבר שנמצא בראש מהתור ומדפיסים אותו
  - מכניסים לתור את שני המצביעים לבנים

```
BFS(T)
Enqueue(Q, root[T])
while not QueueEmpty(Q)
      do x ← Dequeue(Q)
         if x ≠ nil
            then output key[x]
                 Enqueue(Q, left[x])
                 Enqueue(Q, right[x])
```

- סיבוכיות:
  - זמן: $O(n)$, כאשר n מספר הצמתים בעץ
  - זיכרון: $O(m)$, כאשר m המספר המקסימלי של צמתים ברמה כלשהי בעץ

אלגוריתם רקורסיבי לסריקת עץ בינרי לעומק

- מימוש רקורסיבי
  - כל קריאה מקבלת צומת בעץ
  - הרקורסיה היא על תת העץ השמאלי והימני (שתי קריאות רקורסיביות)
  - הפלט נעשה במקום המתאים לפי הסדר הנדרש: לפני, בין, או אחרי שתי הקריאות הרקורסיביות
- סיבוכיות:
  - זמן: $O(n)$, כאשר n מספר הצמתים בעץ
  - עומק הרקורסיה המקסימאלי: $O(h)$, כאשר h גובה העץ

```
PreOrderDFS(T)
    DFS(root[T], "PreOrder")

InOrderDFS(T)
    DFS(root[T], "InOrder")

PostOrderDFS(T)
    DFS(root[T], "PostOrder")

DFS(x, order)
    if x ≠ nil
        then  if order = "PreOrder"
                  then output key[x]
              DFS(left[x], order)
              if order = "InOrder"
                  then output key[x]
              DFS(right[x], order)
              if order = "PostOrder"
                  then output key[x]
```

האוניברסיטה הפתוחה

אלגוריתם איטרטיבי לסריקת עץ בינרי לעומק (תרגיל 10.4-5)

```
DFS(x, order)
    if x = NIL then return
    root ← x
    visit ← 1  ▶ which visit of node x: 1st, 2nd or 3rd
    while true do
        if visit = 1
            then if order = "PreOrder"
                    then output key[x]
                if left[x] = NIL
                    then visit ← 2  ▶ short-wire left subtree
                    else x ← left[x]
        if visit = 2
            then if order = "InOrder"
                    then output key[x]
                if right[x] = NIL
                    then visit ← 3  ▶ short-wire right subtree
                    else x ← right[x]
                        visit ← 1
        if visit = 3
            then if order = "PostOrder"
                    then output key[x]
                if x = root then return
                if x = left[parent[x]]
                    then visit ← 2
                x ← parent[x]
```

- מימוש איטרטיבי של השגרה *DFS*(*x*, *order*)
  - לכל צומת נכנסים שלש פעמים: מהאב, מהבן השמאלי, ומהבן הימני
  - הפלט נעשה באחת מהכניסות, לפי הסדר הנדרש
- סיבוכיות:
  - זמן: $O(n)$, כאשר n מספר הצמתים בעץ
  - זיכרון: $O(1)$

לתשומת לב:
הפתרון במדריך הלמידה שגוי
(תרגיל ח-15, עמ 147)

אלגוריתם איטרטיבי לסריקה Pre-Order עם מחסנית

PreOrderDFS(T)

```
Push(S, root[T])
while not StackEmpty(S)
      do x ← Pop(S)
         if x ≠ nil
            then output key[x]
                 Push(S, right[x])
                 Push(S, left[x])
```

- הרעיון: המחסנית משמשת להדמייה של הרקורסיה
  - מכניסים את השורש למחסנית
  - בכל איטרציה:
    - מוציאים את האיבר שנמצא בראש המחסנית ומדפיסים אותו
    - מכניסים למחסנית את הבן הימני, <u>ואחריו</u> את הבן השמאלי

- סיבוכיות:
  - זמן: $O(n)$, כאשר n מספר הצמתים בעץ
  - זיכרון: $O(h)$, כאשר h גובה העץ

האוניברסיטה הפתוחה

אלגוריתם איטרטיבי לסריקה In-Order עם מחסנית (תרגיל 12.1-3)

- הרעיון: נחזיק במחסנית את <u>הענף</u> השמאלי של תת העץ שטרם נסרק
  - בהתחלה העץ כולו טרם נסרק, לכן מכניסים למחסנית (שורה 2) את הענף השמאלי ביותר של העץ
  - כשמוציאים צומת *x* מהמחסנית, כל תת-העץ השמאלי שלו כבר נסרק, לכן מדפיסים (שורה 5) את המפתח של *x*
  - כדי לסרוק גם את התת-עץ הימני של *x*, מכניסים למחסנית (שורה 6) את הענף השמאלי ביותר של תת-העץ הימני של *x*

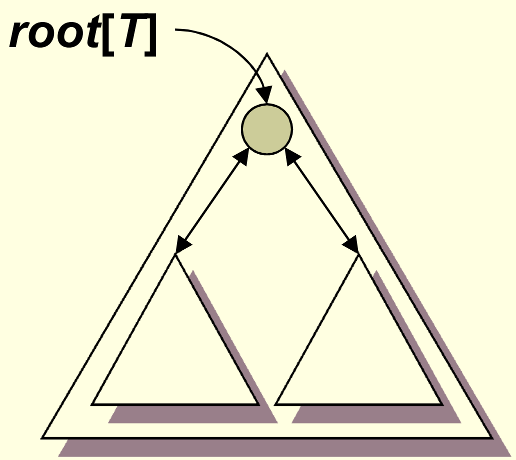
```
InOrderDFS(T)
1.  S ← new stack of tree nodes
2.  PushLeftBranch(S, root[T])
3.  while not StackEmpty(S)
4.      do x ← Pop(S)
5.          output key[x]
6.          PushLeftBranch(S, right[x])
```

```
PushLeftBranch(S, x)
1.  while x ≠ nil
2.      do Push(S, x)
3.          x ← left[x]
```

- סיבוכיות:
  - זמן: $O(n)$, כאשר n מספר הצמתים בעץ
  - זיכרון: $O(h)$, כאשר h גובה העץ

עצי חיפוש בינריים (Binary Search Trees)

- עץ חיפוש בינרי – מבנה נתונים התומך בפעולות על אוסף דינמי של מפתחות
  - שאילתות:
    - חיפוש מפתח נתון
    - מציאת המפתח המינימלי/מקסימלי
    - מציאת המפתח העוקב/קודם של מפתח נתון
  - עדכונים:
    - הכנסה/הוצאה של מפתח
- שימושים לדוגמא
  - מילון
  - תור קדימויות
- סימונים
  - מספר הצמתים בעץ: *n*
  - גובה העץ: *h*



עץ חיפוש בינרי (עח"ב) - הגדרה

- עץ חיפוש בינרי (Binary Search Tree) הוא עץ בינרי בו בכל צומת *x* מתקיימת התכונה הבאה (שנקראת תכונת העח"ב):
  - עבור כל צומת *y* בתת-העץ השמאלי של *x*: $key[y] \leq key[x]$
  - עבור כל צומת *z* בתת-העץ הימני של *x*: $key[z] \geq key[x]$
- בניגוד לתכונת הערימה, תכונת העח"ב אינה התנהגות מקומית של צומת ושני בניו, אלא התנהגות יותר גלובלית
  - יש לוודא ש- $key[x]$ גדול שווה מכל המפתחות בתת העץ השמאלי, וקטן שווה מכל המפתחות בתת העץ הימני.
  - תכונת העח"ב מכתיבה סדר גלובלי בין כל המפתחות בעץ
- הגדרה אלטרנטיבית לעח"ב:

עץ בינרי שסריקה In-Order שלו מניבה סדרת מפתחות ממוינת

האוניברסיטה הפתוחה

בדיקת חוקיות של עח"ב

הרעיון:

- כל הפעלה רקורסיבית של האלגוריתם על עץ נתון תחזיר שלישית ערכים:
  - התשובה true/false,
  - המפתח המקסימאלי בעץ
  - המפתח המינימאלי בעץ
- נבדוק רקורסיבית את תת העץ השמאלי והימני.
- נבדוק שמפתח השורש גדול שווה למפתח המקסימאלי בתת העץ השמאלי, וקטן שווה למפתח המינימאלי בתת העץ הימני

```
isBST(x)    ► returns triplet {ans, max, min}
if x = nil
    then return {false, nil, nil}
if left[x] = nil
    then {ansL, maxL, minL} ← {true, key[x], key[x]}
    else  {ansL, maxL, minL} ← isBST(left[x])
if right[x] = nil
    then {ansR, maxR, minR} ← {true, key[x], key[x]}
    else  {ansR, maxR, minR} ← isBST(right[x])
ans ← ansR and ansL and  (maxL ≤ key[x] ≤ minR)
max ← max(maxR, maxL, key[x])
min ← min(minR, minL, key[x])
return {ans, max, min}
```

- סיבוכיות:
  - זמן: $O(n)$, כאשר n מספר הצמתים בעץ
  - עומק הרקורסיה: $O(h)$, כאשר h גובה העץ

בדיקת חוקיות עח"ב לפי ההגדרה האלטרנטיבית

- הרעיון:
  - נבצע סריקה תוכית איטרטיבית
  - נזכור תמיד את המפתח האחרון שהודפס ונשווה אליו את המפתח הבא שיודפס.
  - אם המפתח הבא קטן יותר אזי העץ אינו עח"ב והאלגוריתם יחזיר מיידית כישלון.
  - אם האלגוריתם סיים את כל הסריקה, תוחזר הצלחה

```
isBST(T)
prevKey ← −∞
S ← new stack of tree nodes
PushLeftBranch(S, root[T])
while not StackEmpty(S)
    do x ← Pop(S)
        if key[x] < prevKey
            then return false
        prevKey ← key[x]
        PushLeftBranch(S, right[x])
return true
```

- סיבוכיות:
  - זמן: $O(n)$, כאשר n מספר הצמתים בעץ
  - זיכרון: $O(h)$, כאשר h גובה העץ

```
PushLeftBranch(S, x)
while x ≠ nil
    do Push(S, x)
        x ← left[x]
```