מבני נתונים

מבנים לאחסון נתונים:

1. מחסנית

פעולה	מימוש ע"י מערך	מימוש ע"י רשימה מקושרת
create-stack() מחזיר מחסנית ריקה	סיבוכיות זמן: O(1) סיבוכיות מקום: O(n)	סיבוכיות זמן: O(1) סיבוכיות מקום: O(n)
push(x, S) מכניס איבר x למחסנית S	סיבוכיות זמן: O(1)	סיבוכיות זמן: O(1)
top(S) מחזיר את האיבר שבראש מחסנית S	סיבוכיות זמן: O(1)	סיבוכיות זמן: O(1)
pop(S) מוציא את האיבר שבראש מחסנית S	סיבוכיות זמן: O(1)	סיבוכיות זמן: O(1)
is-empty(S) בודק האם המחסנית S ריקה	סיבוכיות זמן: O(1)	סיבוכיות זמן: O(1)
size(S) מחזיר את מספר האיברים במחסנית S	סיבוכיות זמן: O(n)	סיבוכיות זמן: O(n)

עובד בשיטת LIFO – Last In First Out.

2. תור

פעולה	מימוש ע"י מערך	מימוש ע"י רשימה מקושרת
create-queue(Q) מחזיר תור ריק	סיבוכיות זמן: O(1) סיבוכיות מקום: O(n)	סיבוכיות זמן: O(1) סיבוכיות מקום: O(n)
enqueue(x, Q) מכניס איבר x לתור Q	סיבוכיות זמן: O(1)	סיבוכיות זמן: O(1)
front(Q) מחזיר את האיבר שבראש תור Q	סיבוכיות זמן: O(1)	סיבוכיות זמן: O(1)
dequeue(Q) מוציא את האיבר שבראש תור Q	סיבוכיות זמן: O(1)	סיבוכיות זמן: O(1)
is-empty(Q) בודק האם התור Q ריק	סיבוכיות זמן: O(1)	סיבוכיות זמן: O(1)
size(Q) מחזיר את מספר האיברים בתור Q	סיבוכיות זמן: O(n)	סיבוכיות זמן: O(n)

עובד בשיטת FIFO – First In First Out.

3. רשימה מקושרת

פעולה	סיבוכיות
create-list() מחזיר רשימה מקושרת ריקה	סיבוכיות זמן : O(1)
is-empty(L) בודק האם הרשימה L ריקה	סיבוכיות זמן : O(1)
find(k, L) מחזיר את האיבר שמפתחו k ברשימה L	סיבוכיות זמן (במקרה הגרוע) : $\theta(n)$
insert-first(x, L) מכניס איבר x לתחילת הרשימה L	סיבוכיות זמן : O(1)
insert(x, y, L) מכניס איבר x לאחר האיבר y ברשימה L	סיבוכיות זמן : O(1)
delete-first(L) מסיר את האיבר בתחילת רשימה L	סיבוכיות זמן : O(1)
delete(x, L) מסיר את האיבר x מרשימה L	סיבוכיות זמן : O(n)
delete-after(y, L) מסיר את האיבר לאחר האיבר y ברשימה L	סיבוכיות זמן : O(1)

עצים:

עץ הוא מבנה היררכי.

מושגים בסיסיים:

1. עץ מושרש – עץ שבו אחד הקודקודים נבחר. הקודקוד הנבחר נקרא שורש.
   הערה: צומת בודד הוא עץ מושרש.
2. צומת – כל איבר בעץ נקרא צומת (או קודקוד).
3. צלע – כל 2 צמתים בעץ מחוברים ע״י צלע.
4. דרגה – מספר הילדים של הצומת.
5. עלה – צומת ללא ילדים.
6. צומת פנימית – צומת שאינה עלה (כלומר, יש לה ילדים).
7. מסלול – סדרת צמתים שכל אחד הורה של הקודם.
   אורך מסלול – מספר הצלעות (או: מספר הצמתים פחות אחד).
8. גובה העץ – אורך המסלול הארוך ביותר מהשורש עד לעלה כלשהו (מסומן: height).
9. עומק צומת – אורך המסלול מהצומת לשורש (מסומן: depth).
10. תת-עץ המושרש ב-x – העץ ששורשו הוא x ומכיל את כל צאצאיו.
11. רמה בעץ – קבוצת צמתים הנמצאת באותו עומק.

עץ בינארי:

עץ בינארי הוא עץ ריק או עץ שלכל צומת יש תת-קבוצה של {ילד ימני, ילד שמאלי}. כלומר, עץ בינארי הוא עץ ריק או עץ בעל 2 ילדים לכל היותר.

עץ בינארי מלא - עץ שלכל צומת פנימי יש 2 ילדים.

עץ בינארי שלם - עץ בינארי מלא שבו העלים באותו עומק.

<u>משפטים:</u>

1. מספר העלים בעץ מלא הוא 1 יותר ממספר הצמתים הפנימיים בעץ.
2. מספר הצמתים בעץ בינארי שלם בגובה h הוא $2^{h+1}-1$.
3. מספר הצמתים בעץ בינארי כלשהו בגובה h הוא $\log(n+1)-1$.
4. בעץ בינארי שלם מתקיים: $h = \theta(log(n))$
5. בעץ בינארי כלשהו מתקיים: $h = O(n)$ & $h = \Omega(log(n))$.

<u>סריקות עץ בינארי:</u>

1. **סריקה תוכית (inorder):**
   א) סרוק את תת-העץ השמאלי
   ב) בקר בשורש
   ג) סרוק את תת-העץ הימני

   <u>הערה:</u> ניתן לממש אלגוריתם לא רקורסיבי של סריקה תוכית באמצעות מחסנית.

2. **סריקה תחילית (preorder):**
   א) בקר בשורש
   ב) סרוק את תת-העץ השמאלי
   ג) סרוק את תת-העץ הימני
3. **סריקה סופית (postorder):**
   א) סרוק את תת-העץ השמאלי
   ב) סרוק את תת-העץ הימני
   ג) בקר בשורש
4. **סריקה רוחבית:**
   צמתי העץ נסרקים רמה אחר רמה משמאל לימין. באלגוריתם זה יש שימוש בתור, שבו נכניס את הצומת שעליו עומד x. לאחר ההכנסה, נוציא את האיבר הראשון בתור ונדפיס אותו. אחרי זה, תתבצע בדיקה אם קיים בן שמאלי, ואם יש תכניס אותו לתור ובאופן זהה גם לבן הימני.

<u>חישוב גובה העץ:</u>

נגדיר: $h = \max(h_1, h_2) + 1$. האלגוריתם יהיה רקורסיבי שיחל מהשורש של העץ ועד לעלים. על כל צומת שאין לה ילדים יוחזר 1-.

<u>הערה:</u> מנוסחה זו ניתן לומר שגובה של עץ עם צומת אחת הוא 0 (גובה של בן שמאלי יחזיר 1- וכך גם גובה של בן ימני, ומכאן: $h = \max(-1, -1) + 1 = -1 + 1 = 0$).

<u>מבנה נתונים מילון (Dictionary):</u>

פעולות	הסבר
create-dictionary()	**אתחול** - יוצר מילון ריק.
insert(k, D)	**הכנסת איבר** - מוסיף ל-D איבר שמפתחו k.
delete(k, D)	**הוצאת איבר** - מסיר מ-D איבר שמפתחו k.
find(k, D)	**חיפוש** - מחזיר מצביע לאיבר ב-D שמפתחו k או null אם לא נמצא כזה.
successor(k, D)	**עוקב** - מחזיר מצביע לאיבר ב-D שמפתחו עוקב ל-k או null אם לא קיים כזה.
predecessor(k, D)	**קודם** - מחזיר מצביע לאיבר ב-D שמפתחו קודם ל-k או null אם לא קיים כזה.
min(D)	**מינימום** - מחזיר את המפתח המינימלי ב-D.
max(D)	**מקסימום** - מחזיר את המפתח המקסימלי ב-D.
catenate(D_1, D_2)	**שרשור** (ההנחה שכל המפתחות ב-D_1 קטנים מהמפתחות ב-D_2.
split(k, D)	**פיצול** - מפצל את D ל-D_1 שבו כל האיברים בעלי מפתח קטן יותר מ-k, וב-D_2 כל האיברים בעלי מפתח גדול מ-k.

מימוש בעזרת רשימה מקושרת - לא יעיל !

לכן, ניתן לממש בעזרת **עץ חיפוש בינארי**.

עץ חיפוש בינארי:

עץ חיפוש בינארי הוא עץ בינארי המקיים - לכל צומת x :

- כל המפתחות בתת-העץ השמאלי קטנים מהמפתח של x.
- כל המפתחות בתת-העץ הימני גדולים מהמפתח של x.

העלות של פעולות החיפוש, ההכנסה וההוצאה במילון היא O(h), כאשר h הוא גובה העץ. צורת עץ חיפוש בינארי תלויה בסדר הכנסת האיברים לעץ, ולכן נחלק את גובה העץ ל-3 מקרים :

1. <u>במקרה הטוב :</u> העץ הוא עץ שלם, ולכן גובה העץ הוא $\theta(\log n)$.
2. <u>במקרה הגרוע :</u> העץ הוא 'מסלולי', ולכן גובה העץ הוא $n-1$.
3. <u>במקרה הממוצע :</u> $\frac{1}{n!}\cdot\sum_{i=1}^{n!} h(i)$ כאשר h(i) הוא גובה העץ T_i.

המטרה היא למצוא משפחה של עצים, שבמקרה הגרוע, גובה העץ = $O(\log n)$.

עצי AVL:

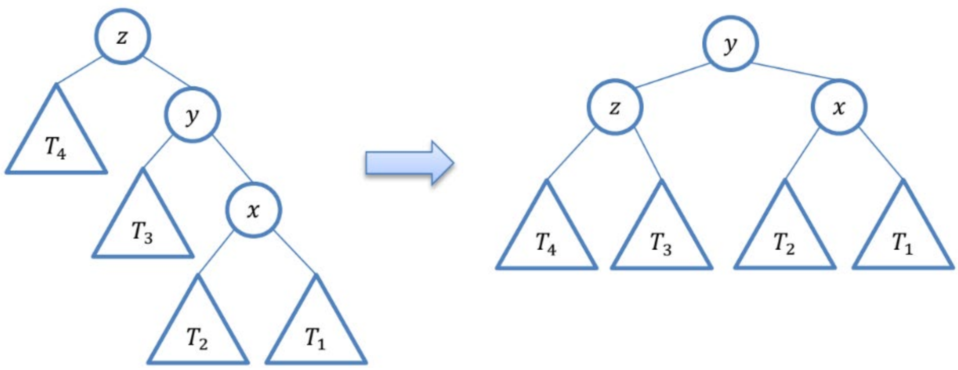
<u>עצים מאוזנים</u>: משפחה של עצים נקראת <u>מאוזנת</u> אם לכל עץ במשפחה מתקיים: גובה העץ הוא O(log n) כאשר n הוא מספר הצמתים בעץ.

עץ AVL הוא עץ חיפוש בינארי המקיים: לכל צומת, **ההפרש** בין הגובה של תת-העץ השמאלי ותת-העץ הימני של הצומת הוא: 1, 0 או 1-. ערך זה נקרא <u>גורם האיזון</u>.

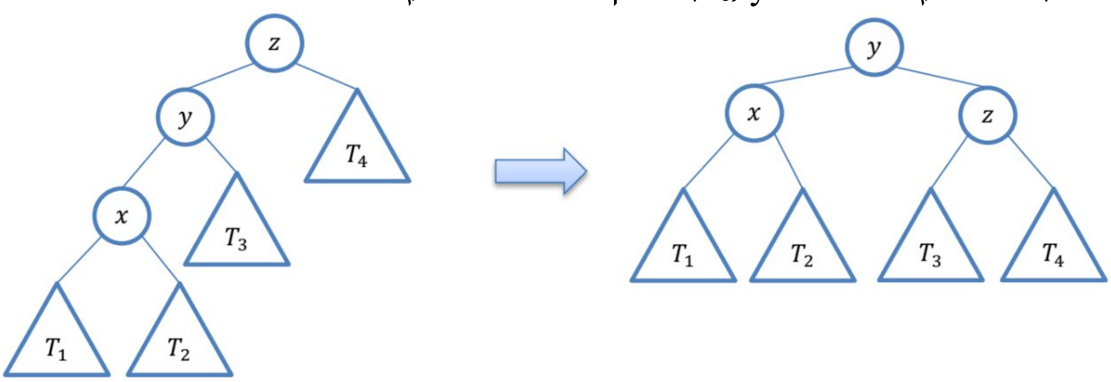
נייצג עץ AVL באופן כזה שלכל צומת בעץ יהיה שדה נוסף (height) המייצג את גובה תת-העץ המושרש בצומת. בנוסף, יש פעולות הכנסה והוצאה (בסיבוכיות: O(log n)).

בכל פעולת הכנסה או מחיקה של צמתים, ייתכן שהעץ לא יישאר מאוזן כפי שהיה. לכן, נאזן את העץ ע"י סריקתו מהצומת שהכנסנו (או הוצאנו) כלפי מעלה, עד שנמצא את הצומת הראשון שאינו מאוזן. קיימים 4 מקרים של הפרות איזונים:

1. <u>איזון עץ RR = Right Right</u>: במקרה זה, תת-העץ הימני של השורש גדול מתת-העץ השמאלי (גורם האיזון הוא 2-). כמו כן, בתת-העץ הימני (של השורש), תת-העץ הימני שלו גדול מתת-העץ השמאלי (גורם האיזון הוא 1-).
נבצע את האיזון בעזרת **Left-Rotate**:
   א) הבן השמאלי y של השורש z יהיה השורש החדש.
   ב) הצומת z יהיה הבן הימני של השורש החדש.
   ג) תת העץ הימני של y (T_3) יהפוך להיות תת-העץ השמאלי של z.



2. <u>איזון עץ LL = Left Left</u>: במקרה זה, תת-העץ השמאלי של השורש גדול מתת-העץ הימני (גורם האיזון הוא 2). כמו כן, בתת-העץ השמאלי (של השורש), תת-העץ השמאלי שלו גדול מתת-העץ הימני (גורם האיזון הוא 1).
נבצע את האיזון בעזרת **Right-Rotate**:
   א) הבן השמאלי y של השורש z יהיה השורש החדש.
   ב) הצומת z יהיה הבן הימני של השורש החדש.
   ג) תת העץ הימני של y (T_3) יהפוך להיות תת-העץ השמאלי של z.

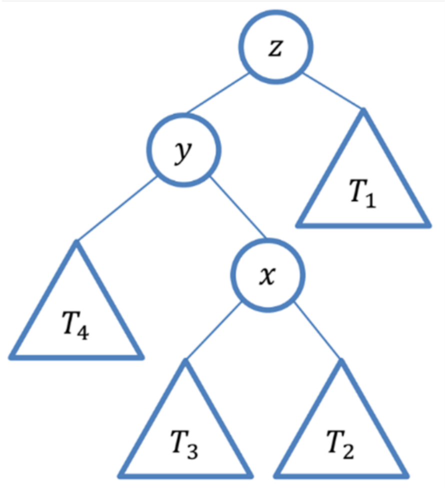


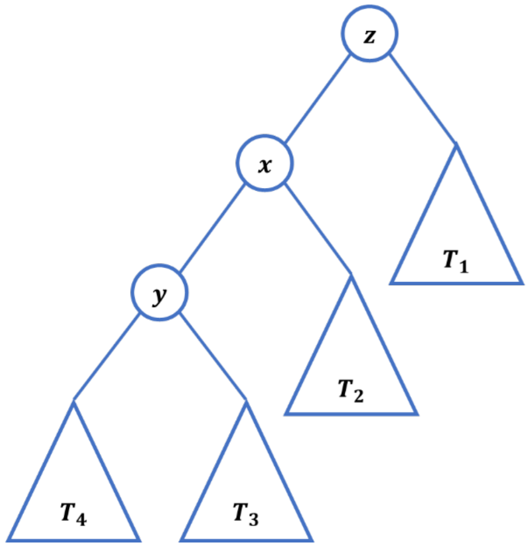
3. <u>איזון עץ LR = Left Right</u>: במקרה זה, תת-העץ השמאלי של השורש גדול מתת-העץ הימני (גורם האיזון הוא 2). בתת-העץ השמאלי (של השורש), תת-העץ הימני שלו גדול מתת-העץ השמאלי (גורם האיזון הוא 1-).

נבצע את האיזון בעזרת **Left-Rotate** על תת-העץ השמאלי, ואז **Right-Rotate** על העץ כולו:

שלב ראשון - **Left-Rotate** על תת-העץ השמאלי:

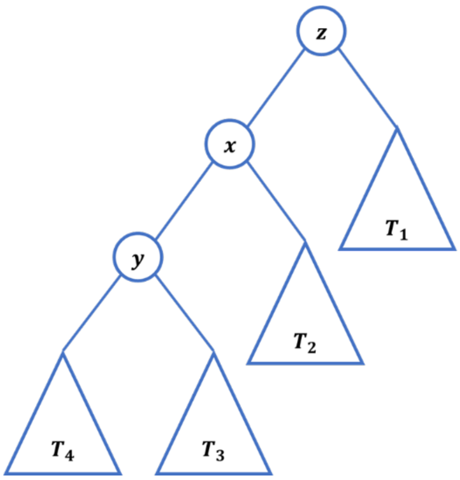
א) x יהפוך להיות הבן השמאלי של z.
ב) y יהפוך להיות הבן השמאלי של x.
ג) תת-העץ השמאלי של x - T_3 יהפוך להיות תת-העץ הימני של y.

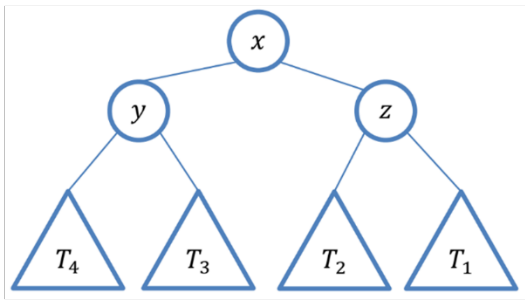




שלב שני - **Right-Rotate** על העץ כולו:

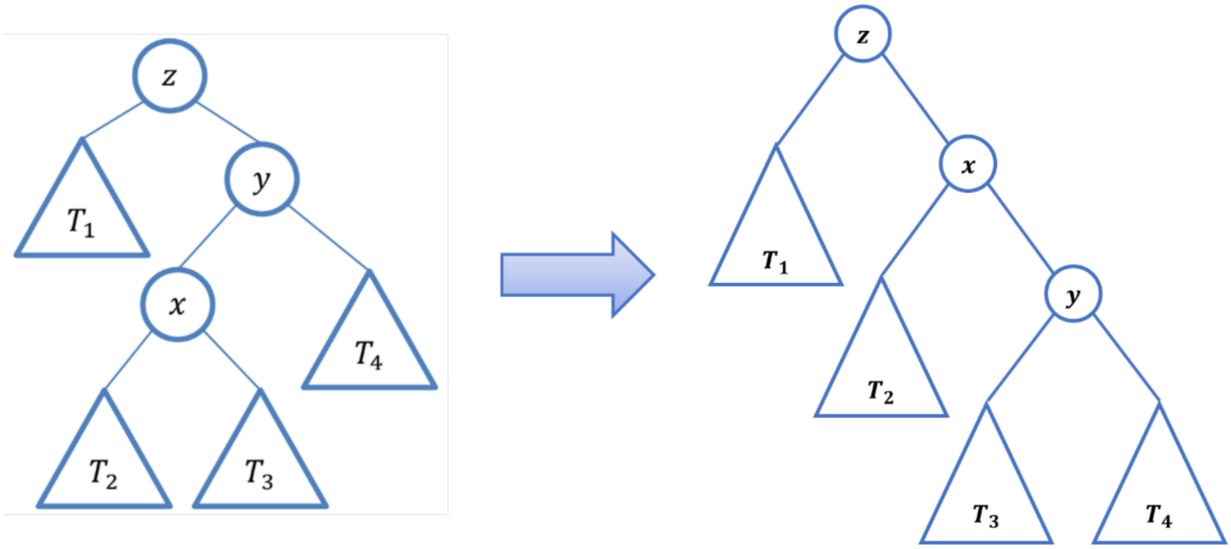
א) הבן השמאלי x של השורש z יהיה השורש החדש.
ב) הצומת z יהיה הבן הימני של השורש החדש.
ג) תת העץ הימני של x (T_2) יהפוך להיות תת-העץ השמאלי של z.



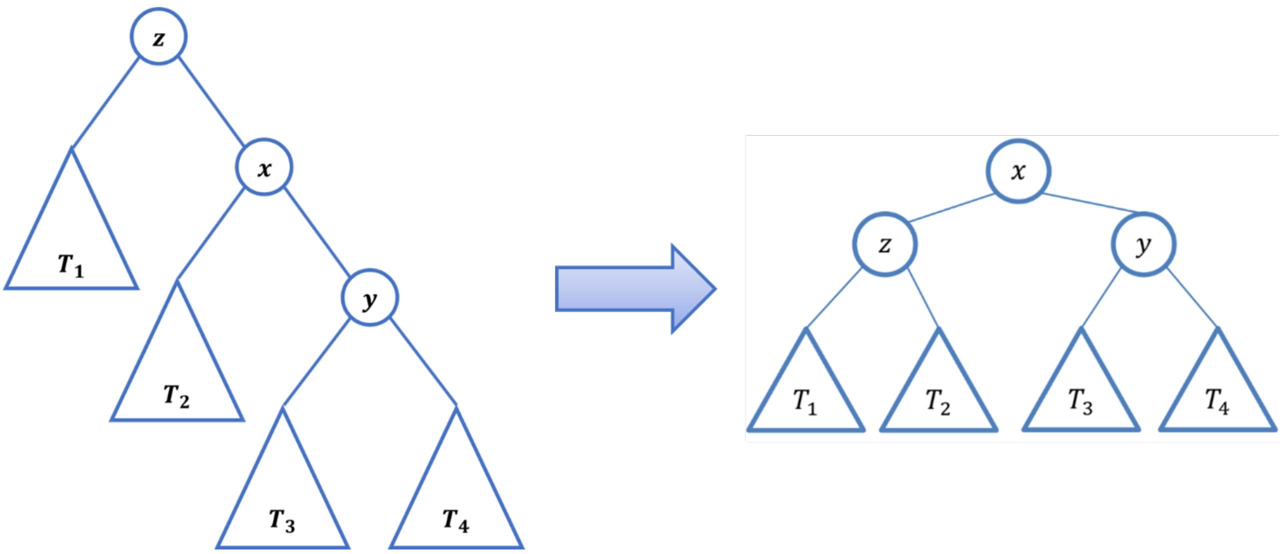


4. <u>איזון עץ RL = Right Left</u>: במקרה זה, תת-העץ הימני של השורש גדול מתת-העץ השמאלי (גורם האיזון הוא -2). בתת-העץ הימני (של השורש), תת-העץ השמאלי שלו גדול מתת-העץ הימני (גורם האיזון הוא 1).
נבצע את האיזון בעזרת **Right-Rotate** על תת-העץ הימני, ואז **Left-Rotate** על העץ כולו:

שלב ראשון - **Right-Rotate** על תת-העץ הימני:
א) x יהפוך להיות הבן הימני של z.
ב) y יהפוך להיות הבן הימני של x.
ג) תת-העץ הימני של x - T_3 יהפוך להיות תת-העץ השמאלי של y.



שלב שני - **Left-Rotate** על העץ כולו:
א) הבן הימני x של השורש z יהיה השורש החדש.
ב) הצומת z יהיה הבן השמאלי של השורש החדש.
ג) תת העץ השמאלי של x (T_2) יהפוך להיות תת-העץ הימני של z.



זמן ריצה של כל סוגי הגלגולים: $O(1)$

ערימות:

ייצוג עץ בינארי במערך:

שורש העץ נשמר בתא הראשון במערך.

- הבן השמאלי של הצומת בתא ה-i נמצא בתא ה-2i+1.
- הבן הימני של הצומת בתא ה-i נמצא בתא ה-2i+2.

עבור עץ עם כמות מידע של n, המקום שנדרש הוא: 2^n-1.

עץ בינארי כמעט שלם:

עץ שבו כל הרמות מלאות, פרט אולי לרמה התחתונה בה קיים רצף משמאל לימין.

שם	הסבר	נוסחה
Left(i)	בן שמאלי של צומת i	2i+1
Right(i)	בן ימני של צומת i	2i+2

Parent(i) - הורה של צומת i. הנוסחה שלו: $\left\lfloor \frac{i-1}{2} \right\rfloor$.

הגובה של עץ בינארי כמעט שלם עם n צמתים: $h = \lfloor \log n \rfloor$.

ערימת מינימום:

מבנה נתונים מופשט המוגדר ע"י הפעולות:

פעולות	הסבר
create-heap()	**אתחול** - יוצר ערימה ריקה.
insert(x, Q)	**הכנסת איבר** - מוסיף לערימה Q איבר שמפתחו x.
delete-min(Q)	**הוצאת איבר** - מסיר מ-Q את האיבר עם המפתח הקטן ביותר.
min(Q)	**מינימום** – מציאת האיבר המפתח המינימלי ב-Q.

מימוש ערימה נעשה בעזרת עץ חיפוש מאוזן, כאשר כל אחת מהפעולות בזמן $O(\log n)$:

א. נבנה עץ בינארי כמעט שלם.
ב. לכל צומת v, המפתחות של שני הבנים גדולים מהמפתח של v.

ערימה היא מבנה נתונים מופשט של **תור עדיפויות**: מבנה נתונים התומך בקבוצה Q של איברים, כך שלכל איבר מצורף ערך הנקרא מפתח.

<u>פעולות נוספות:</u>

פעולות	הסבר
HeapifyDown(Q, i)	פעולה זו מורידה מטה את האיבר במקום ה-i בערימה Q בסיבוכיות O(log n).
HeapifyUp(Q, i)	פעולה זו מעלה למעלה את האיבר במקום ה-i בערימה Q בסיבוכיות O(log n).

מיון ערימה:

- הכנסת האיברים לתוך מערך A.
- יצירת ערימה מאיברי המערך A ע"י הצבת ערכי המערך בתור עץ, ואז באמצעות לולאה להשתמש בפעולות HeapifyDown או HeapifyUp (בהתאם לסוג הערימה).
- יהי B מערך ריק. ביצוע n פעמים: הוצאת המינימום מהערימה והכנסתו למקום הפנוי האחרון במערך B.

<u>זמן ריצה:</u> O(nlog n). היתרון במיון זה, שאין בנייה של מערכי עזר (בניגוד למיון-מיזוג). כלומר, המיון הוא in-Place.

מיונים:

<u>משפט:</u> כל אלגוריתם מיון מבוסס השוואות עורך $\Omega(n \cdot \log n)$ השוואות במקרה הגרוע.

<u>הערה:</u> סיבוכיות זמן ריצה של **מיון הכנסה** הוא $O(n^2)$.

עץ החלטה:

ניתן לתאר מיוני השוואה באופן מופשט באמצעות עצי החלטה. עץ החלטה מייצג את ההשוואות שמבצע אלגוריתם מיון כשהוא פועל על קלט בגודל נתון. ההחלטה היא שאלת ההשוואה באלגוריתם.

- צומת פנימית בעץ - שאלת ההשוואה הנשאלת במהלך האלגוריתם.
- בן ימני - ההחלטה שעונה "כן" על השאלה בצומת זו.
- בן שמאלי - ההחלטה שעונה "לא" על השאלה בצומת זו.
- עלה בעץ - הסדר של הנתונים שנקלטו לקבלת הסדר הממוין.

עץ החלטה הוא **עץ מלא**. כמות העלים היא n! כאשר n מייצג את מספר האיברים שנקלטו. מכאן ניתן לומר שגובה העץ מייצג את מספר ההשוואות המתבצעות במקרה הגרוע ביותר, כלומר גובהו לפחות log n!.

מיון מניה:

הרעיון של מיון זה הוא לחשב לכל איבר כמה איברים קטנים או שווים לו. אם עבור איבר x מסוים יש 18 איברים שקטנים או שווים לו, אז x ימוקם במקום ה-18.

<u>הערה:</u> ההנחה היא שטווח המספרים למיון חסום בטווח 1, ..., k.

<u>האלגוריתם:</u>

1. ניצור מערך עזר C בגודל k כאשר כל התאים שלו מאופסים.
2. נסרוק את מערך A (המערך המקורי), ונמנה את מספר המופעים של כל איבר בו.
3. נסכום את מספר המופעים - לכל C[i] נוסיף את קודמו C[i-1]. באופן זה, בתא C[i] שמור מספר האיברים שקטנים או שווים ל-i במערך A.

4. נסרוק את מערך A מהסוף להתחלה, ועבור כל איבר A[i] נמקם אותו במערך המטרה B במקום ה-C[A[i]], ולאחר מכן נקטין את ערכו של C[A[i]] ב-1.

מיון מנייה הוא מיון **יציב** - הוא שומר על סדר הערכים המקורי כאשר האיברים שווים.

סיבוכיות מיון זה: O(n+k) עבור איברים בטווח k ,…,1.

מיון בסיס:

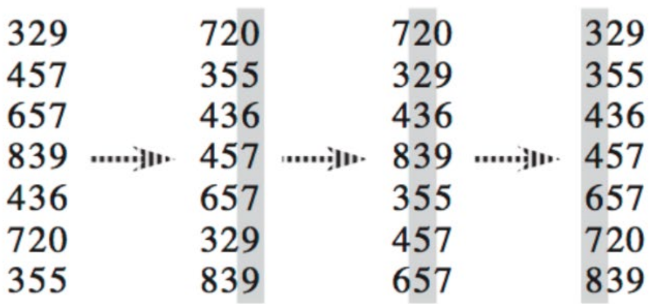
הרעיון של מיון זה הוא למיין את המספרים ב-k מיונים (k מהווה את כמות הספרות), בכל פעם נמיין ספרה אחרת במספר - החל מהספרה הכי פחות משמעותית ועד הספרה המשמעותית ביותר. ניעזר במיון יציב (כמו: מיון מניה).

<u>הערה:</u> מספר הספרות בייצוג כל המספרים למיון חסום ע״י קבוע k.

<u>תיאור האלגוריתם:</u>

עבור מערך A המכיל n מספרים בעלי d ספרות:

```
RADIXSORT(A,d)
    for i = 1 to d
        Use a stable sort (counting sort for example) to sort array A on digit i
```



סיבוכיות מיון זה: O(n * k) עבור מספרים בעלי k ספרות.

<u>הפרד ומשול:</u>

- <u>הפרד:</u> פצל את הבעיה לכמה תת-בעיות זרות.
- <u>משול:</u> פתור את תת-הבעיות באופן רקורסיבי.
- <u>צרף:</u> צרף את הפתרונות של תת-הבעיות לפתרון הבעיה המקורית.

חלק זה יעסוק בשיטות השונות לפתירת נוסחאות נסיגה.

קיימים מספר אפשרויות לפתרון נוסחאות נסיגה:

1. **עץ רקורסיה:**
- מפתחים את נוסחת הנסיגה בצורת עץ, כאשר הבנים של צומת הם הקריאה הבאה ברקורסיה, עד למקרה של תנאי ההתחלה.
- העומק המינימלי של עלה הוא המקרה הטוב, העומק המקסימלי של עלה (הגובה של העץ) הוא המקרה הרע.
- אם המקרים זהים, אז קיים חסם הדוק. אחרת, אלו הם החסמים.
- **כדאי להיעזר בעץ רקורסיה עבור ניחוש הפתרון, ואז להוכיח את נכונותו בעזרת שיטת ההצבה.**

2. **שיטת ההצבה:**
- ננחש את צורת הפתרון הכללית (לדוגמה: בעזרת עץ רקורסיה).
- נציב את התשובה שניחשנו במקום הפונקציה.
- נוכיח ת נכונות הפתרון באמצעות **אינדוקציה**.

<u>הערה:</u> יש להוכיח חסם תחתון ועליון <u>בנפרד</u>. אין אפשרות להוכיח ישירות עבור חסם הדוק.

3. **משפט האב (המאסטר):**

המשפט תקף לנוסחאות נסיגה מהצורה הבאה:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \qquad b > 1,\ a \geq 1$$

נבצע השוואה בין $n^{log\ b\ a}$ ל- $f(n)$.

א) אם $f(n) = O(n^{\log_b a - \varepsilon})$ עבור קבוע $\varepsilon > 0$, אזי $T(n) = \Theta(n^{\log_b a})$.

ב) אם $f(n) = \Theta(n^{\log_b a})$, אזי $T(n) = \Theta(n^{\log_b a} \cdot \log n)$.

מורחב: אם $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$, אז מתקיים: $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$.

ג) אם $f(n) = \Omega(n^{\log_b a - \varepsilon})$ עבור קבוע $\varepsilon > 0$, וגם קיים קבוע c < 1 כך ש: $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$, אזי $T(n) = \Theta(f(n))$.

4. **שיטת האיטרציה:**
- מפתחים את נוסחת הנסיגה עד שמתקבל סכום של איברים התלוי רק ב-n ובתנאי ההתחלה.
- חוסמים את הפתרון באמצעות שיטות למציאת ערכי סכומים.

5. **החלפת משתנים:**
- נסמן את המשתנה n בביטוי שונה השקול לו שתלוי במשתנה m שאיתו יהיה קל יותר לפתור את הנוסחה.
- נשתמש בכלים המוכרים לנו לפתרון הנוסחה שתלויה ב-m.
- נציב בחזרה במקום m את n ונקבל פתרון.

לדוגמה: עבור נוסחה עם $\sqrt{n}$ נציב $m = \log n$, ונמצא את הביטוי $S(m)$.

/* אלגוריתם SELECT */

טבלאות גיבוב:

מבנה נתונים יעיל למימוש מילון - מבנה המכיל איברים לפי מפתחות. במקרה הגרוע, זמן הריצה של פעולת החיפוש בטבלת גיבוב היא $O(n)$, אך תחת הנחות סבירות, תוחלת זמן החיפוש הינה $O(1)$.

טבלת גיבוב היא הכללה של מערך רגיל, כאשר מספר המפתחות המאוחסנים בפועל קטן יחסית למספר הכולל של המפתחות האפשריים.

במקום להשתמש במפתח עצמו כאינדקס למערך, האינדקס למערך מחושב מתוך המפתח (לפי פונקציית גיבוב). כך נמפה את המפתחות מקבוצת האיברים U לתחום מצומצם $\{0, 1, \dots, m-1\}$ כאשר m הוא גודל המערך.

נשים לב שניתן למפות מחרוזת למספר שלם באמצעות ערך ה-ASCII של כל תו.

פתרון התנגשויות:

ייתכן מצב שבו פונקציית הגיבוב החזירה לנו ערך זהה עבור שני מפתחות שונים.

במצב כזה, כל איבר במערך יצביע לראש רשימה מקושרת שתכיל את האיברים שמופו לאותו תא.

בהכנסה, נכניס את האיבר לראש הרשימה המקושרת.

פונקציית הגיבוב:

- מוגדרת כך: $h: U \to \{0, 1, \dots, m-1\}$. (לא יכולה להיות חח"ע)
- חייבת להיות מחושבת בזמן קבוע O(1).
- יש לבחור פונקציית גיבוב שתמזער ככל שניתן את ההתנגשויות, ותפזר באופן אחיד את האיברים.

שיטת החילוק:

הפונקציה: $h(k) = k \bmod m$.

נמפה כל מפתח k אל אחד מ-m התאים ע"י לקיחת השארית של חלוקת k ב- m . רצוי ש- m לא יהיה חזקה של 2 (כלומר: $m = 2^p$) או קרוב לחזקה כלשהי של 2 (כי אז $h(k)$ הוא הביט הפחות משמעותי של k).

בחירה מוצלחת של m היא מספרים ראשוניים שאינם קרובים לחזקות מדויקות של 2.

<u>שיטת הכפל:</u>

הפונקציה: $h(k) = \lfloor m \cdot (kA \bmod 1) \rfloor, \quad 0 < A < 1$

1. נכפול את המפתח k בקבוע $0 < A < 1$.
2. נכפול את החלק השברי של kA ב-m וניקח את ערך הרצפה של התוצאה.

<u>בשתי השיטות האחרונות מתקיים:</u>

Insert (T, x)	הכנס את x מהרשימה. במקרה הגרוע: O(1).
Search(T, k)	חפש את האיבר עם המפתח k ברשימה. במקרה הגרוע: θ(אורך הרשימה)
Delete(T, x)	מחק את x מהרשימה. במקרה הגרוע: θ(אורך הרשימה)

<u>מיעון פתוח:</u>

בשיטה זו, נטפל בהתנגשויות שיש לנו ע"י חיפוש מקום בטבלה עצמה. אם מנסים להכניס איבר והתא תפוס אז בודקים את התא הבא באופן שיטתי, ואם הוא תפוס נבדוק תא נוסף וכן הלאה.

- **<u>דגימה לינארית:</u>**

במקרה של התנגשות, מאחסנים את האיבר במקום הפנוי הבא בטבלה, לפי סדר, באופן ציקלי - אחרי התא האחרון, עוברים לתא הראשון.

פונקציית הגיבוב: $h(k) = k \bmod m$.

1. <u>הכנסה:</u> (הכנסת איבר עם המפתח k)
   (א) הפעל את פונקציית הגיבוב h(k).
   (ב) עבור על התאים עם האינדקס h(k) (באופן ציקלי) עד שמגיעים לתא ריק (או תא המכיל ערך "פנוי").
   (ג) הכנס את האיבר.
2. <u>חיפוש:</u> (חיפוש איבר בעל מפתח k)
   (א) הפעל את פונקציית הגיבוב h(k).
   (ב) עבור על התאים מהתא עם אינדקס h(k) עד לאחד מהמקרים הבאים:
   - נמצא איבר עם מפתח k - החזר אותו.
   - הגעת לתא ריק (שלא סומן עם delete) - האיבר לא נמצא.
   - עברת על m תאים - האיבר לא נמצא.
3. <u>הוצאה:</u> (הוצאת איבר בעל מפתח k)
   (א) חפש איבר עם מפתח k.
   (ב) אם נמצא - הצב בתא זה ערך "פנוי", והחזר את האיבר.
   אחרת - החזר "לא קיים".

יתרונות	**חסרונות**
פשוט לביצוע	ייתכנו רצפים ארוכים של תאים תפוסים (הצטברות ראשונית), המאריכים את זמן החיפוש הממוצע.
	כאשר השימוש דורש הוצאות, אורך החיפוש תלוי גם באיברים שכבר הוצאו ולא רק באיברים שכרגע במבנה.

- **<u>בדיקה ריבועית:</u>**
אם ישנם 2 מרכזים קרובים אליהם יגובבו מספר רב של איברים, הם לא יקרינו האחד על השני והטבלה תהיה מאוזנת יותר.
אם יגובבו המון איברים לתאים אחדים, נקבל מצב של הצטברות משנית.
פונקציית הגיבוב מוגדרת כך: $h(k,i) = (h(k) + c_1 i + c_2 i^2) \bmod m$
(כאשר: h פונקציית הגיבוב, k המפתח המבוקש, i מספר הניסיון להכנסה שהמספור מתחיל מ-0).

- **<u>גיבוב כפול:</u>**
מחפשים מקום פנוי בטבלה בקפיצות משתנות ע"י בחירת פונקציית גיבוב נוספת d(k) ומחפשים מקום פנוי בטבלה בקפיצות של d(k). זאת אומרת, הקפיצות יהיו jd(k) עבור j = 0, 1, …, m-1 באופן ציקלי.
פונקציית הגיבוב מוגדרת כך: $g(k,j) = (h(k) + j \cdot d(k)) \bmod m$
כדי שהחיפוש אחר תא פנוי יסרוק את טבלת הגיבוב כולה, הערך d(k) חייב להיות זר לגודל m של טבלת הגיבוב. לכן, נבחר m ראשוני, ונבנה d כך שתפיק תמיד מספר שלם חיובי וקטן מ-m.

- **<u>גיבוב קוקייה:</u>**
רעיון של גיבוב קוקייה הוא שיש 2 פונקציות גיבוב במקום אחת. ניצור 2 טבלאות גיבוב בגודל זהה, וכל פונקציית גיבוב תיתן לנו אינדקס של כל אחד מהטבלאות.
מפתח x יישמר ב- $T_1[h_1(x)]$ או ב- $T_2[h_2(x)]$
  1. <u>חיפוש:</u> מכיוון שיש שימוש ב-2 טבלאות גיבוב, אז יש לחפש את הערך בשתי הטבלאות. הפונקציה תחזיר את איבר x מופיע בטבלה 1 או בטבלה 2. במקרה הגרוע, יהיה O(1).
  2. <u>הכנסה:</u> תחילה, נבדוק האם האינדקס בתא $h_1(x)$ בטבלה T_1 תפוס. אם לא, אז האיבר יוכנס לתא זה. אם האיבר תפוס, נכניס את האיבר החדש לתא $T_1[h_1(x)]$, ונוציא את האיבר שהיה שם. עבור האיבר היוצא, נעשה את אותה הבדיקה עם הטבלה T_2. כך נבצע שוב ושוב עד שיוכנסו כלל האיברים לשתי הטבלאות (או כאשר הטבלאות מלאות).

<u>קבוצות זרות - בעיות Union / Find:</u>

נרצה לממש מבנה נתונים המייצג אוסף של קבוצות זרות, כך שכל קבוצה מאופיינת ע"י נציג-איבר מהקבוצה.

פעולות	הסבר
Makeset(i)	**אתחול** - יוצר קבוצה חדשה בעלת איבר בודד i ומחזיר אותה.
Find(i)	**מציאה** -מחזיר את הקבוצה לה שייך האיבר i
Union(p, q)	**איחוד** - מחזיר קבוצה חדשה שמהווה איחוד של הקבוצות p, q (הקבוצות p, q המקוריות חדלות מלהתקיים ובמקומן קיימת הקבוצה החדשה).

כעת נדבר על המימושים הקיימים עבור מבנה נתונים מסוג זה:

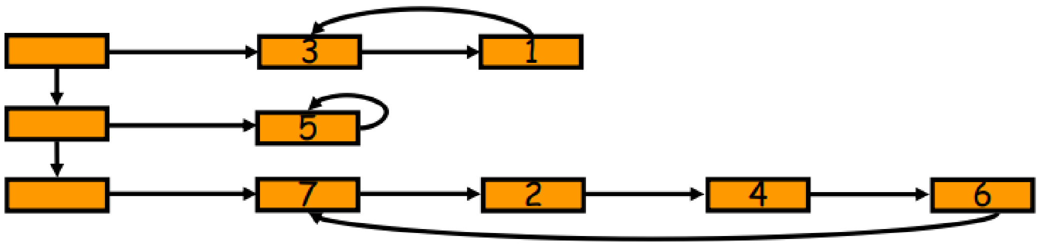
1. **<u>מימוש נאיבי בעזרת מערך:</u>**

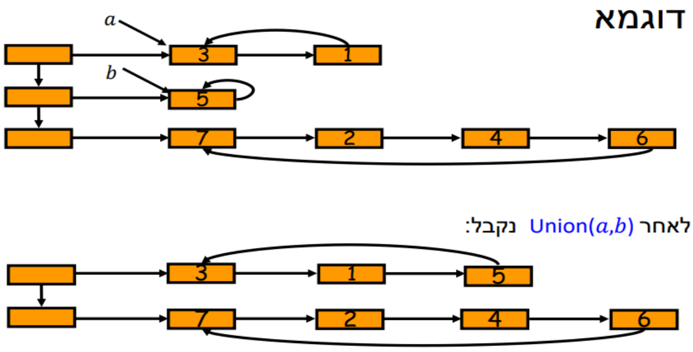
נשתמש במערך A. בתא A[i] נשמור את שם הקבוצה אליה שייך האיבר i.
החיסרון במימוש זה הוא שדרוש חסם על מספר האיברים (כדי להקצות מערך בגודל n).
לצורך מתן שמות לקבוצות, נשתמש במשתנה בשם counter שיאותחל ל-0. בכל פעולת Union ערך האינדקסים, שעבורם בוצעה הפעולה, יעודכנו לערך של counter (במידה ויבוצע פעולת Union על אינדקסים שבהם יש אותו ערך, אז שני האינדקסים יעודכנו, וכך הלאה).

פעולות	סיבוכיות
Makeset(i)	סיבוכיות הזמן : O(1).
Find(i)	סיבוכיות הזמן : O(1).
Union(p, q)	סיבוכיות הזמן : O(n).

<u>הערה :</u> במימוש זה, הנחנו ש : U = {1, 2, 3, …, n} וש-n ידוע מראש. לכן, יכולנו להניח את קיומו של מערך בגודל n לצורך המימוש שלנו. אם זה לא המצב, אז ניאלץ לטפל בזה.

2. **<u>מימוש נאיבי בעזרת רשימות מקושרות מעגליות:</u>**
נייצג כל קבוצה כרשימה מעגלית. נחזיק רשימה מקושרת של מצביעים אל הרשימות המעגליות :

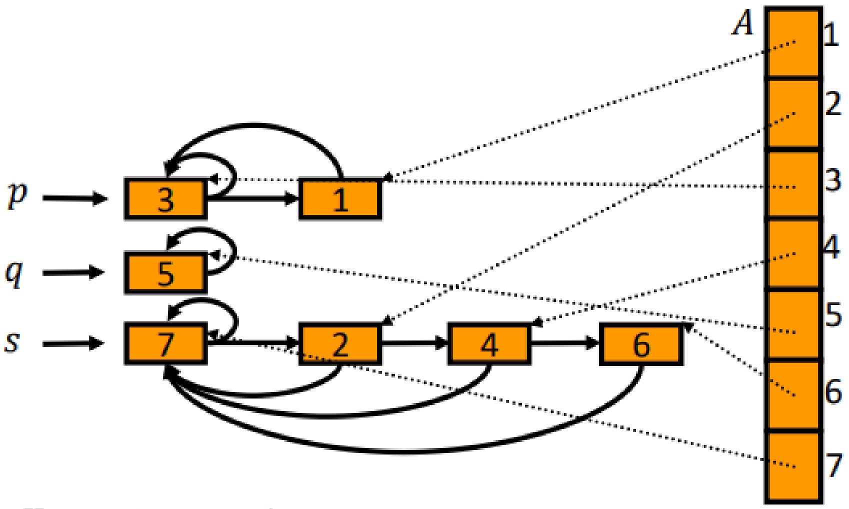


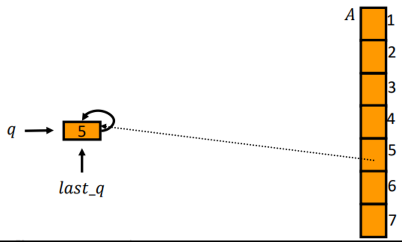
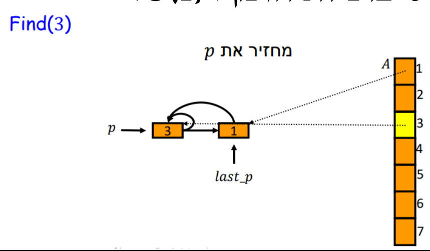
פעולות	סיבוכיות
Makeset(i)	ממומשת ע"י הוספת רשימה עם איבר בודד. סיבוכיות הזמן : O(1).
Find(i)	ממומשת ע"י מעבר על כל הרשימות עד שמוצאים את i. סיבוכיות הזמן : O(n).
Union(p, q)	ממומשת ע"י איחוד הרשימות המוצבעות ע"י p, q. סיבוכיות הזמן : O(n). **דוגמא** 

3. **<u>מימוש ע״י רשימות מקושרות ומערך מיפוי:</u>**

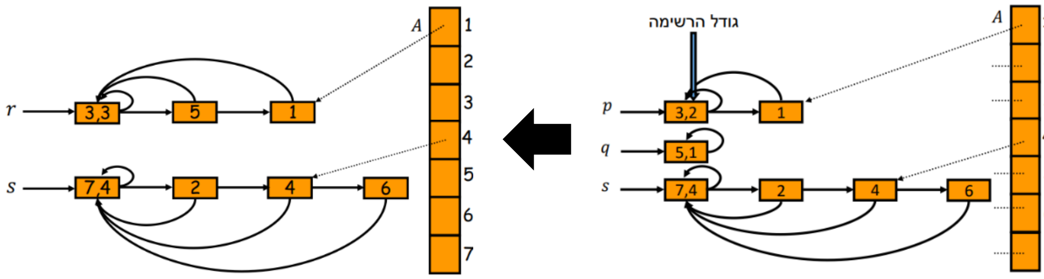
<u>תזכורת - זמן משוערך (Amortized time) :</u> אם m פעולות מתבצעות בתוך זמן M, אז הזמן המשוערך לכל פעולה מוגדר להיות : m / M.
כאשר ישנה סדרה ארוכה של פעולות שרובן קלות לביצוע וחלקן קשות לביצוע, אז ההשפעה של הפעולות הקשות מתחלקת על סדרת הפעולות כולה.

נייצג כל קבוצה כרשימה. כל איבר ברשימה מצביע גם לראש הרשימה וגם לאיבר הבא ברשימה. בנוסף, נחזיק מערך מיפוי איברים A[i] ובו מצביע לאיבר i. קבוצה מיוצגת ע״י המצביע לראש הרשימה המתאימה.



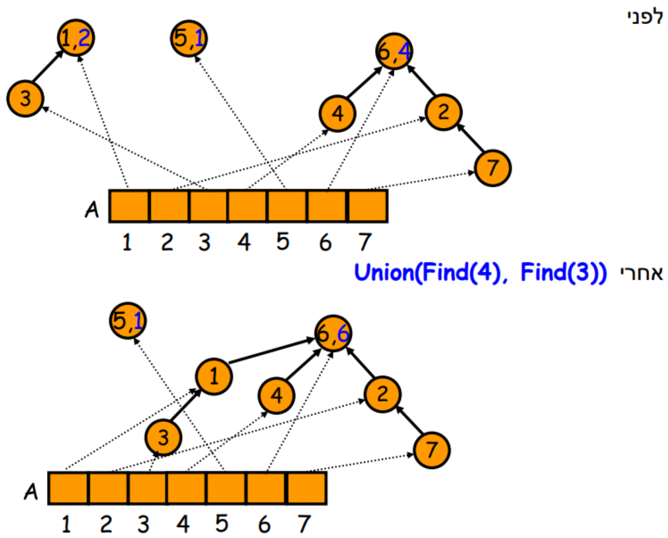
פעולות	סיבוכיות
Makeset(i)	ניצור צומת חדשה שיוצבע מ-A[i]. הצומת החדש יצביע על עצמו ונחזיר מצביע לראש הרשימה, ומצביע לסוף הרשימה. סיבוכיות הזמן : O(1). 
Find(i)	מ-A[i] נעבור אל הצומת של i ונחזיר את ראש הרשימה שלו. סיבוכיות הזמן : O(1). 
Union(p, q)	נאחד את הרשימות המוצבעות ע״י p, q לרשימה אחת ונעדכן את כל המצביעים לראש הרשימה החדשה. מחזירים את ראש הרשימה החדשה. סיבוכיות הזמן : O(n).

שיפור המימוש:
בראש הרשימה נשמור את גודלה של הרשימה. הדבר לא ישפיע על פעולת Find, אך תהיה לו השפעה על פעולת Union: הפעולה תתבצע ע"י הוספת הקבוצה הקטנה לתוך הקבוצה הגדולה. כלומר, ראש הרשימה החדשה יהיה ראש הרשימה הגדולה יותר, וכל המצביעים יצביעו עליו. יש לעדכן רק את המצביעים של הקבוצה הקטנה, ולעדכן את גודל הרשימה לאחר הוספת איברי הקבוצה הקטנה.



טענה: אם בכל איחוד מוסיפים את הקבוצה הקטנה לגדולה, אז כל איבר x משנה את ראש הקבוצה אליה הוא שייך לכל היותר $\log_2 n$ פעמים על כל אוסף של פעולות איחוד.
משפט: אם בכל איחוד מוסיפים את הקבוצה הקטנה לגדולה, אזי הזמן הכולל לביצוע סדרה כלשהי של m פעולות Union הוא לכל היותר O(m * log n).

4. **מימוש ע"י עצים הפוכים (Up Trees):**
לכל קבוצה ניצור עץ הפוך (בנים מצביעים להורה) ובו צומת לכל איבר בקבוצה. שורש העץ יכיל גם את מספר איברי הקבוצה. בנוסף, נחזיק מערך גישה לאיברים. בביצוע פעולת Union "תולים" את שורש העץ הקטן יותר מתחת לשורש העץ הגדול יותר ומעדכנים את גודל הקבוצה המאוחדת.



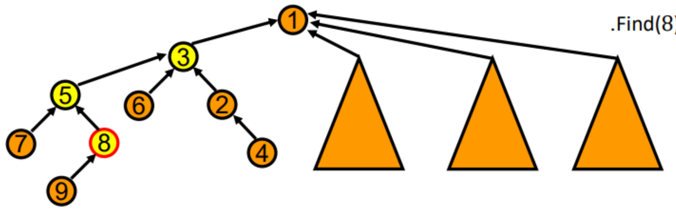
טענה: עבור יער של עצים הפוכים בן n צמתים בעל גובה h שנבנה מאיחודים של קבוצות קטנות לתוך גדולות, מתקיים: $h \le \log_2 n$.
המסקנה מכך היא שזמן פעולת Find היא O(log n) במקרה הגרוע.

פעולות Makeset(i), Union(p, q) הן בסיבוכיות זמן O(1).

שיפור:

כיווץ מסלולים - בזמן ביצוע Find(i), נעדכן את שדה ה-parent של כל הצמתים במסלול מ-i ועד השורש, כך שיצביעו ישירות לשורש.

דוגמא: Find(8).



אחרי Find(8):