מפגש מס' 1: מבוא

1. אלגוריתם : תהליך חישובי לפתרון בעיה. התהליך המקבל קלט, ומפיק פלט מתאים כפתרון.
2. מבנה נתונים : צורת שמירת נתונים במחשב. לכל מבנה מוגדרות פעולות אותן ניתן להפעיל על המבנה, ודרישה ליעילות מסוימת של פעולות אלו. ניתן לצור משתנה מטיפוס מבנה הנתונים.

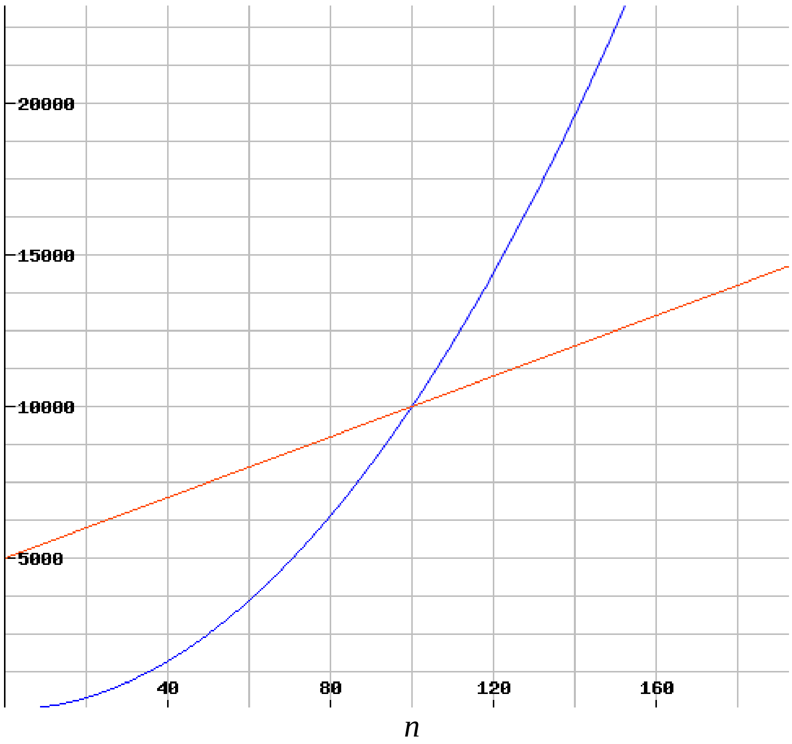
בד"כ, בכל מבני הנתונים, תהיה פעולה המאפשרת לאתחל משתנה מטיפוס מבנה הנתונים, וגם פעולה שתאפשר את שחרורו. כעת ובנוסף, בכל מבנה נתונים, יוצגו פעולות שונות שאופייניות לאותו מבנה נתונים.

3. **הערה כללית לקורס כולו ועבור הבוחן / המבחן** : אין להשתמש במשתנים גלובליים במימוש פונקציות אלא אם נאמר אחרת. כי מטרת הפונקציה היא להיות כללית ולא תלויה במשתנים שמחוץ לה. גם שימוש במשתנה סטטי יעשה רק אם אין אפשרות להשתמש במשתנה מקומי.

המשך מפגש מס׳ 1. נושא המפגש: סיבוכיות.

1. סיבוכיות (Complexity): רוצים לדעת למה שואף מספר הצעדים של האלגוריתם כשגודל הקלט שואף לאינסוף. כלומר, רוצים הערכה של הקשר בין זמן הריצה לבין גודל הקלט עבור קלטים גדולים. נשתמש במשתנה n כדי לסמן את גודל הקלט. (כמות האברים של הקלט).
2. תרגיל כיתה: נתונים שני אלגוריתמים. איזה מהם יותר יעיל:

$T_2(n) = 50n + 5000$, $T_1(n) = n^2$



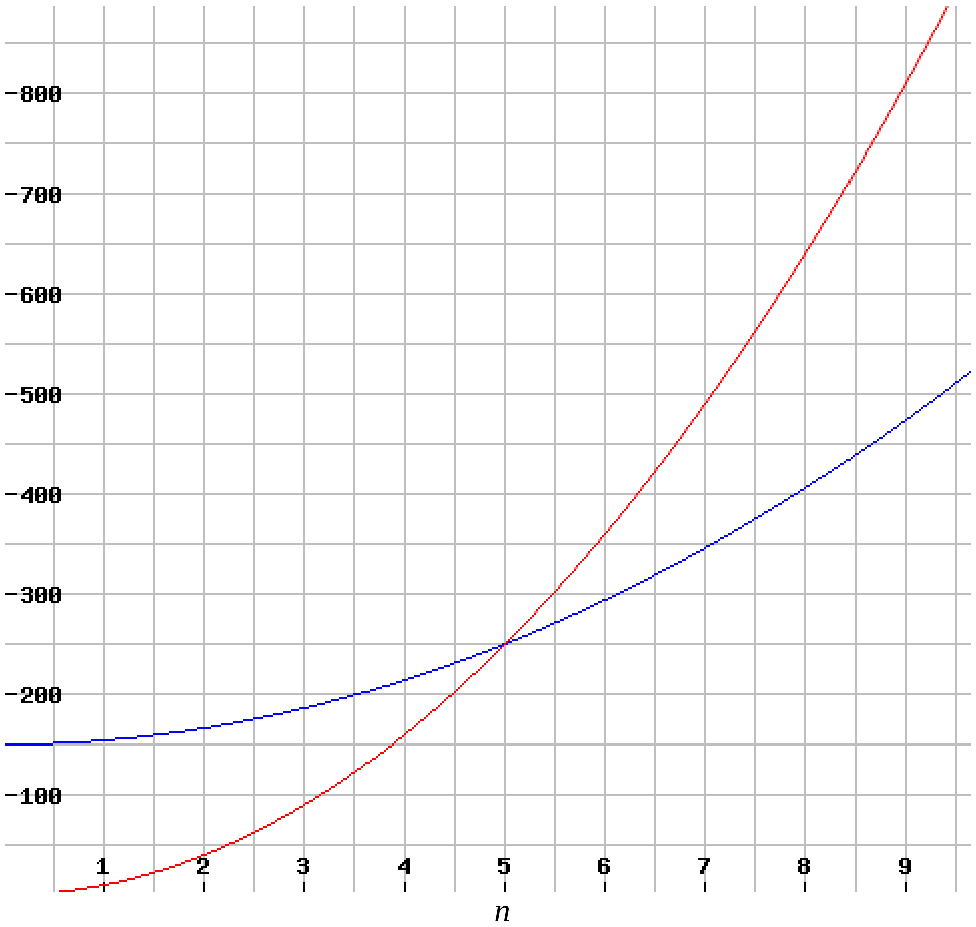
3. בד״כ לא מעוניינים בחישוב מדויק של זמן הריצה /מספר צעדי האלגוריתם, אלא רק בהערכה טובה של הקשר בין זמן הריצה לבין גודל הקלט, כאשר גודל הקלט גדול מספיק. לכן משתמשים בחסמי הסיבוכיות. (יותר קלים לחישוב.) חסם סיבוכיות עליון מסומן באות O. ניתן לראות את הסיבוכיות, גם כהערכת סדר הגודל, של כמה פעמים מבוצע בתוכנית, המשפט שיתבצע בה הכי הרבה פעמים, ביחס לכמות האברים שבקלט.

לדוגמא: הסיבוכיות של קטע הקוד הבא היא $O(n)$, כי יש לבצע את הלולאה n פעמים, בלי קשר לכמה צעדים או זמן מדויק דורשת כל איטרציה של הלולאה:

```
int i;
for(i = 0; i < n ; i++)
   printf("%d, ", i);
```

4. לא מתעניינים בכמה זמן לוקחת כל פעולה, או כמה פעולות מבוצעות ללא קשר לקלט, כי רוצים לדעת עד כמה הקוד עצמו של האלגוריתם יעיל בטיפול בקלט, ולא גורמים אחרים כגון מהירות מחשב.
5. זמן הריצה יכול להיות תלוי לא רק בגודל הקלט, אלא גם בגורמים אחרים, למשל איך האברים בקלט מסודרים. דוגמא: מחפשים איבר כלשהו במערך.
6. בד״כ נמצא את הסיבוכיות עבור המקרה הגרוע, כלומר זה שגורם לאלגוריתם לרוץ הכי הרבה זמן, כי המקרה הגרוע הוא חסם עליון, והוא בד״כ המצב הנפוץ ביותר. לפעמים נמצא את הסיבוכיות עבור המקרה הממוצע.

7. בד"כ נאמר שאלגוריתם מסוים יעיל יותר מאחר, אם הסיבוכיות שלו במקרה הגרוע נמוכה יותר.
8. הסימון O, חסם אסימפטוטי עליון: כאשר אנו אומרים למשל שזמן הריצה של אלגוריתם מסוים הוא $O(n^2)$, הכוונה היא שבמקרה הגרוע ביותר הוא חסום ע"י n^2 כפול קבוע.
(כשמדברים על חסמים, מדובר תמיד על תחום שאינו שלילי.)
באופן הגדרתי: אם $f(n)=O(g(n))$, אז קיימים קבועים חיוביים C ו- n_0, כך ש- $0 \le f(n) \le C*g(n)$ לכל $n \ge n_0$.
הכוונה היא ש- $f(n)$ שייכת לקבוצה $O(g(n))$. למשל: $4*n^2+150 = O(n^2)$.
כי עבור $\mathbf{f(n)=4*n^2+150}$, ניתן לקחת למשל את $C=10$, ואז: $0 \le \mathbf{4*n^2+150} \le \mathbf{10*n^2}$ לכל $n \ge 5$:



9. ניתן לראות, שהכפל בקבוע אינו חשוב לחישוב הסיבוכיות, לכן בד"כ מתעלמים מהקבועים. (אולם בזהירות.)
10. ניתן לרשום את הסימון כחלק ממשוואה: $2n^2+5n+10 = 2n^2+O(n)$. הכוונה היא ש- $O(n)$ מייצגת פונקציה כלשהי שהיא שייכת לקבוצה $O(n)$, ואין לנו עניין לנקוב בשמה.
11. אם הפונקציה $f(n)$ היא פולינום, שהחזקה הגבוהה ביותר שמופיעה בו היא k, אזי: $f(n) = O(n^k)$
למשל: $n*\log(n)+3n^2 = O(n^2)$ (תמיד, באם לא מצוין אחרת בסיס הלוגריתם הוא 2.)
(כלומר: הכפלה של n בקבוע לא תופיע בסיבוכיות של הפונקציה.)
12. $O(1)$ משמעותו: אין קשר בין זמן הריצה/כמות הצעדים, לגודל הקלט. (כלומר, פולינום מדרגה 0.)
13. הסימון Θ: חסם הדוק אסימפטוטית. כשאנו אומרים למשל שזמן הריצה של אלגוריתם מסוים הוא $\Theta(n^2)$, הכוונה היא שבמקרה הגרוע ביותר הוא חסום מלמעלה ע"י n^2 כפול קבוע, וגם מלמטה ע"י n^2 כפול קבוע אחר.
14. סיבוכיות נמוכה+סיבוכיות גבוהה $\leftarrow$ = הסיבוכיות הגבוהה. למשל: $O(n)+O(1)=O(n)$.
15. אם k קבוע כלשהו אזי: $O(n*k)=O(n)$.

16. עקרון החיבור: אם יש לולאות שאינן מקוננות, ואין קשר ביניהן, הסיבוכיות הכוללת היא סכום הסיבוכיות של שתיהן, דהיינו הסיבוכיות היותר גדולה מביניהן. (או של אחת מהן אם לשתיהן אותה הסיבוכיות.)
17. <u>תרגיל כיתה ו/או בתרגול 1</u>: מה הסיבוכיות של קטע הקוד הבא:

```
for(i = 0 ; i < n ; i++)
    printf("%d, ", i);
for(i = 0 ; i < 2*n ; i++)
    printf("%d, ", i);
```

18. עקרון הכפל: אם יש לולאות מקוננות, ושאין קשר ביניהן, הסיבוכיות הכוללת היא מכפלה של הסיבוכיות של כ״א מהן.
19. <u>תרגיל כיתה ו/או בתרגול 2</u>: מה הסיבוכיות של קטע הקוד הבא:

```
for(i=0; i<n; i++)
    for(i=0; i<2*n; i++)
        printf("%d, ", i);
```

20. שיטת האיטרציות: לפעמים נוח לצור טבלה המציגה את מצב המשתנים בלולאה בתום כל איטרציה, ובעזרתה למצוא את הקשר בין כמות האיטרציות לגודל הקלט, וכך למצוא את סיבוכיות הלולאה.
21. <u>תרגיל כיתה ו/או בתרגול 3</u>: מה הסיבוכיות של קטע הקוד הבא:

```
int i=1;
while(i<n)
{
    printf("%d, ", i);
    i*=2;
}
```

22. <u>תרגול</u> - נציג כעת בכיתה תרגילי כיתה בנושא של סיבוכיות.

מפגש מס' 2. נושא המפגש : רקורסיה.

1. תזכורת עבור נושא זה : כמו בכל הקורס, אין לממש פונקציות רקורסיביות עם משתנה גלובלי, אלא אם נאמר במפורש אחרת. (כי הפונקציה צריכה להיות כללית, ולא תלויה בגורמים חיצוניים.)
2. פונקציה רקורסיבית היא פונקציה שקוראת לעצמה.
3. יש בעיות בתכנות, שיותר טבעי או נוח לפתור אותם בגישה הרקורסיבית.
4. מכיוון שהפונקציה קוראת לעצמה בכל פעם, יש להוסיף בפונקציה תנאי עצירה שבו הקריאה הרקורסיבית של הפונקציה לעצמה תיפסק.
5. לדוגמא : פונקציה רקורסיבית שמוצאת עצרת של מספר :

```
double factorial(unsigned n)
{
   if(n==0)return 1;          //תנאי העצירה
   return n*factorial(n-1);   //קריאה רקורסיבית
}
```

6. בד"כ תנאי העצירה יהיה המצב הפשוט והבסיסי, שבו ברורה התשובה של הרקורסיה.
7. יש לשים לב, שבכל קריאה רקורסיבית נוצר עותק חדש של הפונקציה factorial, עם משתנים חדשים וערכים שונים בהתאם.
8. המהדר עוקב אחר סדר הקריאות והביצוע של הפונקציות במבנה הנתונים מחסנית (Stack).
9. אם ברקורסיה אין תנאי עצירה, ניכנס למעשה ללולאה אין סופית. בפועל, התוכנית תעוף/תיעצר מכיוון שכמות העותקים שניתן לצור מהפונקציה וגודל המחסנית מוגבלים.
10. שלב הקריאה הרקורסיבית : זהו השלב שבו הפונקציה קוראת לעצמה. הקריאות מצטברות במחסנית, עד שמגיעים לתנאי העצירה.
11. שלב החזרה הרקורסיבית : לאחר שלב הקריאה הרקורסיבית, כל עותק של הפונקציה מבצע את הפקודות שעדיין נשארו לו לבצע, ומחזיר את התוצאה/חוזרים לעותק הקודם לו, כלומר לעותק שזימן אותו
12. רקורסיית זנב : הקריאה הרקורסיבית היא הפקודה האחרונה שיש בפונקציה. המהדר ממיר את רקורסיית הזנב ללולאה, כי לולאה תרוץ יותר מהר. (כי הוא לא צריך להקצות משתנים לכל עותק, קריאות לעותקים וכו'.)
13. סיבוכיות של פתרון עם רקורסיה לעומת פתרון עם לולאות : כפי שהוסבר, בבעיות רבות הפתרון הרקורסיבי הוא יותר פשוט ומתבקש. אולם מבחינת יעילות, לפעמים הפתרון האיטרטיבי יותר יעיל, לפעמים הרקורסיבי, ולפעמים אין הבדל מבחינת היעילות בין שני הפתרונות. כאשר אין הבדל יעילות מבחינת סיבוכיות, עדיף להשתמש בפתרון איטרטיבי, מכיוון שאינו דורש את העלויות שדורש הפתרון הרקורסיבי, כגון ניהול מחסנית הקריאות, הקצאת משתנים לכל עותק, וכו'.
14. מפגש זה אינו מציג את השיטות למציאת סיבוכיות של אלגוריתם רקורסיבי.

15. בניית עץ מעקב: כדי לעקוב אחר מהלך הרקורסיה ולהבין את דרך ההתנהלות שלה, ניתן להציג עץ מעקב. את העץ מציגים עם דוגמא פשוטה, המאפשרת מעקב אחר סדר הקריאות וערכי המשתנים. לדוגמא: נציג עץ מעקב אחר הפונקציה factorial, כאשר מעבירים לה את הערך 3. שלב הקריאה הרקורסיבי מוצג עם החיצים היורדים. שלב החזרה הרקורסיבי מוצג בעזרת החיצים העולים.

factorial(3) 6
3*factorial(2) 2
2* factorial(1) 1
1* factorial(0) מתקיים תנאי העצירה (=1)

16. <u>תרגיל כיתה 1/או בתרגול</u>: כתוב פונקציה רקורסיבית שתקבל מספר חיובי ושלם. על הפונקציה להחזיר את סכום כל המספרים השלמים מ- 0 עד למספר וכולל.

17. פתרון תרגיל כיתה 1:

```
unsigned long sum(unsigned n)
{
    if(n==0)return 0;          //תנאי העצירה
    return n+sum(n-1);         //קריאה רקורסיבית
}
```

18. <u>תרגיל כיתה 2/או בתרגול</u>: כתוב פונקציה רקורסיבית שתדפיס את תווי המחרוזת שהיא תקבל, תו אחרי תו. (אין להשתמש בלולאות.)

19. פתרון תרגיל כיתה 2:

```
void printStr(const char* str)
{
    if(!*str)return;           //תנאי עצירה - מחרוזת ריקה//
    printf("%c",*str);
    printStr(str+1);
}
```

20. בדוגמא הנ"ל לא רשמנו ++str אלא str+1. כאשר מעבירים משתנה לעותק הבא, מומלץ להעביר אותו בלי לקדם או לשנות אותו בעותק שבו הוא נמצא. התוצאות בד"כ לא צפויות. למשל במקרה הנ"ל, אם היינו מבצעים ++str, הקידום היה מתבצע לאחר שעוברים לעותק הבא, כלומר העותק הבא לא היה מקבל את הקידום לתו הבא, ולכן הרקורסיה לא הייתה מגיעה אל תנאי העצירה.

אם היינו רושמים כאן ++str, זה היה בסדר בדוגמא הזו, אולם בדוגמאות אחרות השינוי של המשתנה בעותק שבו הוא נמצא, יכול להיות בעייתי בשלב החזרה הרקורסיבית, כי יהיה לו את הערך לאחר השינוי שלא התכוונו אליו.

21. <u>תרגול</u> - נציג כעת בכיתה תרגילי כיתה נוספים.

מפגש מס' 3. נושא המפגש: שיטות למיון, מיזוג וחיפוש.

1. במיון (*sorting*) הכוונה היא לסדר סדרה של n מספרי קלט, כך שהקטן ביותר יהיה בהתחלה, כלומר לסדר את אברי הסדרה בסדר עולה.
2. סדרת הקלט מיוצגת בד"כ ע"י מערך, אולם ניתן לייצג גם אחרת, למשל עם רשימה מקושרת.
3. במערך A ממוין, האיבר שנמצא בדיוק באמצע המערך, $A[(n\text{-}1)/2]$, נקרא החציון.
4. אלגוריתם המיזוג (*Merge*): מטרתו צירוף של שני מערכים <u>ממוינים</u> למערך יעד <u>ממוין</u>, ביעילות המרבית. (המערכים המקוריים נשארים ללא שינוי, ואבריהם רק מועתקים למערך היעד. יש להניח שבמערך היעד יש מספיק מקום לנדרש.)

   <u>תרגיל כיתה 1</u>: תאר אפשרויות שונות לביצוע המיזוג. מה הסיבוכיות של כל שיטה? מה השיטה היעילה ביותר?

   <u>תרגיל כיתה 2</u>: כתוב פונקציה *Merge* שתבצע את המיזוג באופן הכי יעיל. הוסף *main* לבדיקה.
5. שיטות המיון מתחלקות ל- 2 קבוצות: מיוני השוואה (*comparison sorts*) ומיונים בזמן ליניארי.
6. מיוני ההשוואה עובדים רק ע"י השוואה בין אברי הקלט. מיוני ההשוואה החשובים: מיון בועות, מיון בחירה, מיון מיזוג, מיון מהיר, מיון ערימה. החסם התחתון על זמן הריצה של מיונים אילו הוא $\Omega(n \log n)$.
7. מיונים בזמן ליניארי מניחים משהו על הקלט ובאופן כזה הם יותר מהירים. למשל: הקלט הוא מספרים שלמים בטווח מסוים. מיונים אלו מבצעים פעולות בנוסף להשוואות, כגון ספירה. זמן הריצה שלהם הוא בד"כ $O(n)$.

   מיונים בזמן ליניארי: מיון מנייה (*counting sort*), מיון בסיס (*radix sort*), מיון דלי (*bucket sort*).
8. הגדרה: אלגוריתם מיון ממיין במקום (*in place*), אם במהלך ריצתו הוא מאחסן לכל היותר מספר קבוע של אברים ממערך הקלט מחוץ למערך. כלומר, מיון שאינו משתמש במערך עזר לאחסון האברים במהלך המיון, ממיין במקום.
9. מיון בחירה (*Selection sort*):

Selection-Sort(A , n)

1. כל עוד $n > 1$, בצע:

   1.1. סרוק את n אברי המערך ומצא את האיבר המקסימאלי.

   1.2. החלף בינו לבין האיבר האחרון במערך. (האיבר האחרון נמצא כעת במקומו הסופי.)

   1.3. הקטן את כמות האיברים הרלבנטיים במערך להמשך טיפול, ע"י ביצוע $n \leftarrow n - 1$.

4	2	-3	1

1	2	-3	4

1	-3	2	4

-3	1	2	4

סיבוכיות מיון הבחירה: $O(n^2)$.

שים לב: ניתן לבצע את האלגוריתם גם ע"י מציאת האיבר הקטן ביותר והחלפתו עם האיבר הראשון.

נעבור כעת על קובץ המיונים הנמצא באתר הקורס, ונראה את מימוש המיון.

10. מיון בועות (*Bubble sort*) :

Bubble-Sort(A , n)

1. כל עוד $n > 1$, בצע :

1.1. עבור i המקבל ערכים, החל מאינדקס האיבר השני במערך ועד אינדקס האיבר האחרון, בצע :

1.1.1. אם $A[i-1] > A[i]$ אזי החלף ביניהם.

1.2. הקטן את כמות האיברים הרלבנטיים במערך להמשך טיפול, ע״י ביצוע $n \leftarrow n - 1$.

4	2	-3	1
2	4	-3	1
2	-3	4	1
2	-3	1	4
-3	2	1	4
-3	1	2	4
-3	1	2	4

סיבוכיות מיון הבועות : $O(n^2)$.

שים לב : ניתן לבצע את האלגוריתם גם ע״י בעבוע האיבר הקטן ביותר שמאלה, עד תחילת המערך הרלבנטי בכל פעם.

נעבור כעת על קובץ המיונים הנמצא באתר הקורס, ונראה את מימוש המיון.

11. מיון הכנסה (*Insertion sort*):

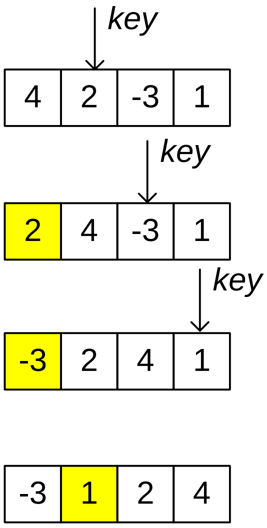
Insertion-Sort(A , n)

1. עבור *i* המקבל ערכים, החל מאינדקס האיבר השני במערך ועד אינדקס האיבר האחרון, בצע:

   1.1. *key* ← A [*i*].

   1.2. החל מתחילת המערך ועד לאיבר שבאינדקס *i* - 1, הזז מקום אחד ימינה רק את כל האיברים במערך שגדולים מ- *key*. (כלומר ייווצר מקום פנוי ל- *key* בדיוק במקום המתאים לו.)

   1.3. הכנס את *key* למקום הפנוי.



סיבוכיות מיון ההכנסה: $O(n^2)$.

נעבור כעת על קובץ המיונים הנמצא באתר הקורס, ונראה את מימוש המיון.

12. שיטת הפרד ומשול (*divide and conquer*):

הפרד: חלק את הבעיה לכמה תתי בעיות.

משול: פתור את תת הבעיה באופן רקורסיבי.

צרף: צרף את הפתרונות של תת הבעיות לפתרון מלא לבעיה המקורית.

13. מיון מהיר (*quick sort*): במיון מהיר מפעילים את הפונקציה *partition*. בוחרים את אחד מאברי המערך כציר (*pivot*). הפונקציה מחלקת את המערך לשני חלקים: חלק ימני ובו מספרים גדולים מהציר או שווים לו, וחלק שמאלי ובו מספרים קטנים מהציר או שווים לו. הציר הוא בד"כ האיבר הראשון, אבל זה לא חייב להיות כך.

מיון מהיר פועל בשיטת הפרד ומשול:

הפרד: המערך מאורגן מחדש לשני תתי מערכים ע"פ הפונקציה *partition*.

משול: שני תתי המערכים ממוינים באמצעות קריאות רקורסיביות למיון מהיר.

צרף: מכיוון ששני תתי המערכים ממוינים במקום (ללא מערכי עזר כלשהם), אין צורך בעבודה נוספת כדי לצרף אותם, והמערך המקורי יהיה כעת ממוין.

שלושת הצעדים הללו מתבצעים בכל רמה של הרקורסיה.

Quick-Sort(A , n)

1. אם במערך יש יותר מאיבר אחד, בצע:
   1.1. האיבר האחרון במערך ← pivot (ציר).
   1.2. אינדקס האיבר הראשון במערך ← i, אינדקס האיבר האחרון במערך ← j.
   1.3. בצע כל עוד i קטן מ- j:
      1.3.1. כל עוד i קטן מ- j , וגם A [i] קטן או שווה לציר, קדם את i לאינדקס הבא.
      1.3.2. כל עוד i קטן מ- j , וגם A [j] גדול או שווה לציר, הקטן את j לאינדקס הקודם.
      1.3.3. אם $i < j$ החלף בין A [i] לבין A [j].

      (בסוף התהליך: המערך מחולק למספרים קטנים או שווים לציר משמאל, כלומר עד האינדקס j,,

   ומספרים גדולים או שווים לציר בצד ימין, כלומר החל מאינדקס j.) (הציר לא בהכרח במקומו הסופי עדיין.)
   1.4. החלף בין האיבר האחרון (הציר) לבין A [j]. (הציר מגיע למקומו הסופי.)
   1.5. מיין את חלק המערך מתחילת המערך עד (ולא כולל) j במיון מהיר. (כלומר מיין את החלק השמאלי שנוצר.)
   1.6. מיין את חלק המערך מ- j+1 ועד סוף המערך במיון מהיר. (כלומר מיין את החלק הימני שנוצר.)

14. <u>תרגיל כיתה 3</u>: תאר בתרשים, את מהלך המיון המהיר עבור המערך שבתרשימים הנ"ל.

15. סיבוכיות של מיון מהיר: במקרה הטוב, כאשר כל החלוקות של השלבים השונים מחלקות את המערך לחלקים שווים, נקבל $O(n \log n)$. (למשל, המערך הוא משמאל לימין: 3 ,4 ,5 ,1 ,2.)

אם בכל קריאה רקורסיבית נוכל לבחור את החציון כציר, אז המערך מתחלק כל פעם לשני חלקים שווים, ומקבלים סיבוכיות זו.

במקרה הגרוע כאשר מחלקים לאיבר אחד וכל השאר בכל שלב, נקבל $O(n^2)$. זה קורה למשל כשהמערך ממוין בסדר עולה/יורד ואבריו שונים.

הסיבוכיות במקרה הממוצע: כאשר יש חלוקות מכל הסוגים במהלך השלבים, באופן אקראי בהתאם לסדר של הקלט, אזי תוחלת זמן הריצה במקרה זה היא $\Theta(n \log n)$. לכן, אחת השיטות לייעל את המיון, היא לבחור את הציר באופן אקראי, ולאו דווקא את האיבר האחרון בכל פעם.

מיון זה מקובל מבחינה מעשית, מכיוון שהוא גם ממין במקום. (לא צריך מערך עזר).

16. נעבור כעת על קובץ המיונים הנמצא באתר הקורס, ונראה את מימוש המיון. שים לב: במיון שבאתר, אכן בוחרים את הציר באופן אקראי, כדי לשפר את האלגוריתם. (לאחר בחירת אינדקס אקראי מתוך המערך, מחליפים את האיבר שבאינדקס זה, עם האיבר שבסוף המערך, כדי שישמש כציר.)

קיימים גרסאות שונות למיון המהיר, למשל בחירה של הציר בתור האיבר הראשון ולא האחרון, בחירה באקראי של הציר בצורות שונות, ועוד.

17. מיון מיזוג (*merge sort*): פועל גם הוא בשיטת הפרד ומשול:

הפרד: מחלק את הבעיה לשתי תתי בעיות בכל פעם.

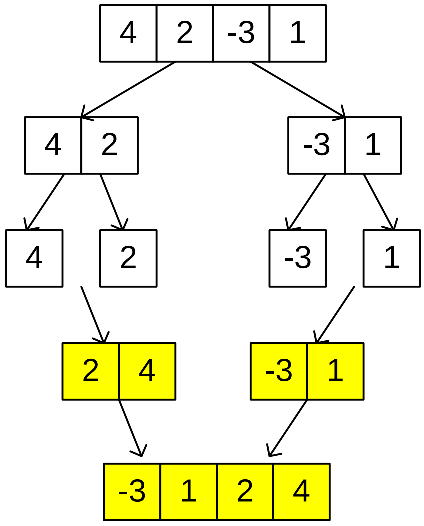
משול: ממיין כל אחד מהחלקים רקורסיבית באמצעות מיון מיזוג.

צרף: ממזג את שתי הסדרות הממוינות לסדרה ממוינת אחת.

שלושת השלבים הללו מתבצעים בכל רמה של הרקורסיה.

Merge-Sort(A , n)

1. אם במערך יש יותר מאיבר אחד, בצע:
   1.1. חלק את המערך לשני חצאים.
   1.2. מיין את החצי הראשון של המערך במיון מיזוג.
   1.3. מיין את החצי השני של המערך במיון מיזוג.
   1.4. בצע מיזוג לחצאים הממוינים.



סיבוכיות מיון המיזוג: בכל קריאה לפונקציה המערך מתחלק לשניים, ולכן יש $\log(n)$ חלוקות. בכל חלוקה קוראים לפונקציה *Merge*, שמתבצעת ב- $O(n)$, ולכן הסיבוכיות כולה היא: $O(n \log n)$.

סיבוכיות זו מתקיימת בכל המקרים האפשריים.

חסרון של מיון מיזוג: אינו ממיין במקום, כי הפונקציה *Merge* דורשת מערך עזר.

נעבור כעת על קובץ המיונים הנמצא באתר הקורס, ונראה את מימוש המיון.

18. מיון ערימה: מבוסס על מבנה הנתונים ערימה. הסיבוכיות היא $O(n \log n)$. (בכל המקרים). המיון נחשב לממיין במקום, כי ניתן לבנות גרסה שלו ללא מערך עזר. המיון יוסבר בסיכום על ערימה.

19. מיונים בזמן ליניארי: מיון מנייה (*counting sort*): מניח שכל אחד מאברי הקלט הוא מספר שלם בתחום $1 \ldots k$. (בשפת סי התחום יכול להתחיל מ- 0). מכיוון שהמיון אינו ממיין במקום, באופן מעשי נשתמש בו כאשר k אינו גדול.

הרעיון של המיון: עבור כל איבר בקלט, נבדוק כמה אברים יש שהם יותר קטנים ממנו בקלט. אם למשל מצאתי שעבור המספר 15 שבקלט, יש שלושה אברים שקטנים ממנו, אז הוא צריך להיות במקום ה- 4 במערך הפלט.

עובדים עם מערך עזר, שכל אינדקס בו מייצג מספר בקלט.

Counting-Sort(A , B , k) (A - מערך הקלט, B - מערך הפלט הממוין, k - המספר המקסימאלי שיכול להיות בקלט.)

1. הקצה מערך עזר C באורך k ואפס את איבריו. (או כך שהאינדקס של האיבר האחרון יהא k.)
2. עבור כל איבר x במערך A, בצע $C [x] \leftarrow C [x] + 1$
3. עבור i מהאינדקס של האיבר השני במערך ועד אינדקס האיבר ה- k-י, בצע:
   3.1. $C [i] \leftarrow C [i] + C [i - 1]$ (כל תא יכיל כמה מספרים קטנים או שווים לאינדקס התא יש בקלט.)
4. עבור i החל מהאינדקס של האיבר האחרון במערך A, ועד לאינדקס של האיבר הראשון במערך A, בצע:
   4.1. $B [C [A [i]]] \leftarrow A [i]$ (ממקם כל איבר במקום המתאים לו במערך הפלט.)
   4.2. $C [A [i]] \leftarrow C [A [i]] - 1$ (אם יש מספרים זהים צריך לעדכן שהכמות ירדה.)

20. תרגיל כיתה 4: תאר בתרשים, את מהלך מיון המנייה עבור המערך הבא, משמאל לימין: 2 ,4 ,1 ,3 ,2 ,4.
21. הסיבוכיות של מיון המנייה: כאשר $k=O(n)$, כלומר הטווח הוא פונקציה ליניארית של כמות האברים בקלט, אז הסיבוכיות של המיון היא $O(n)$.
22. מיון ליניארי נוסף: מיון בסיס (*radix sort*): ממיין n מספרים שלמים שכל אחד מהם הוא בן d ספרות, וכל ספרה שייכת לקבוצה $1 \ldots k$. זמן הריצה של מיון זה הוא גם ליניארי.
23. שיטות לחיפוש: רוצים לחפש איבר כלשהו במערך ממוין. אפשרות אחת היא לסרוק אותו כרגיל ולחפש את האיבר המבוקש. שיטה זו אינה יעילה כי אינה מנצלת את העובדה שהמערך כבר ממוין. בשיטה זו הסיבוכיות תהיה $O(n)$, גם אם המערך ממוין או לא. שיטת חיפוש זו נקראת **חיפוש סדרתי**.
24. שיטה אחרת לחפש במערך ממוין, היא שיטת החיפוש הבינארי (binary search):

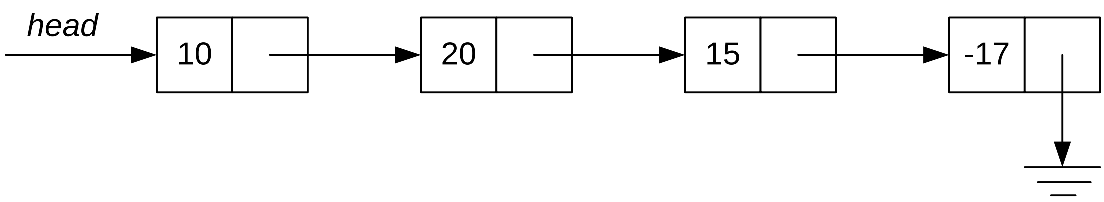
Binary-Search(A , n , x)

1. כל עוד במערך יש לפחות איבר אחד, בצע:
   1.1. $mid \leftarrow$ אינדקס אמצע המערך
   1.2. אם $x = A [mid]$ החזר את mid. (מחזיר את אינדקס האיבר המבוקש.)
   1.3. אם $x < A [mid]$ המשך לחפש בחלק השמאלי של המערך.
   1.4. אחרת, המשך לחפש בחלק הימני של המערך.
2. החזר 1-, כערך המסמן את אי מציאת האיבר המבוקש.

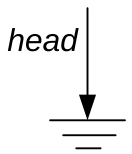
25. תרגיל כיתה 5: ממש פונקציה לביצוע של חיפוש בינארי באופן איטרטיבי. הפונקציה תקבל את המערך ואת גודלו.
26. תרגיל כיתה 6: ממש פונקציה לביצוע של חיפוש בינארי באופן רקורסיבי, ללא לולאות. במימוש זה, אתה תעביר לפונקציה את האינדקסים של תחילת המערך וסופו, במקום את כמות האיברים.
27. תרגיל כיתה 7: מה הסיבוכיות של חיפוש בינארי?
28. תרגול - נציג כעת בכיתה תרגילי כיתה נוספים.

מפגש מס' 4. נושא המפגש: רשימות מקושרות - חלק א': רשימות מקושרות חד- כיווניות.

1. חזרה בכיתה על: עבודה עם מודול תוכנה.
2. ברשימה מקושרת חד-כיוונית (singly linked list) מקצים צמתים בצורה דינמית בזמן הריצה. בכל צומת שומרים אינפורמציה וכן מצביע לצומת הבאה. המצביע האחרון ברשימה יצביע ל- *NULL*. המצביע לצומת הראשונה נקרא *head*.
3. תרשים לדוגמא של רשימה מקושרת חד- כיוונית:



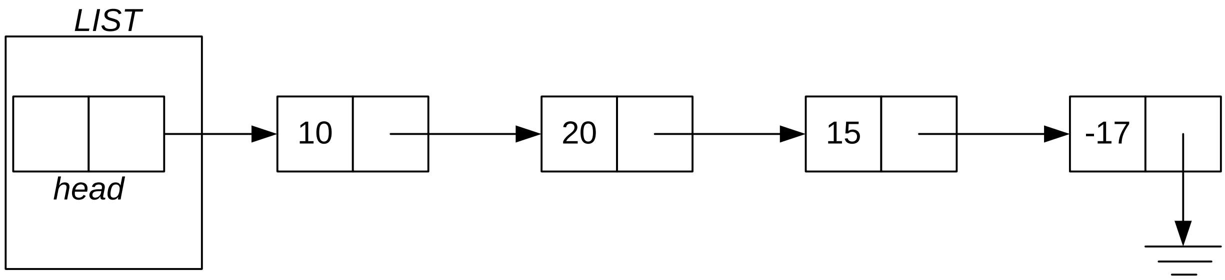
4. רשימה ריקה: אינה מכילה אף צומת ואז המצביע לצומת הראשונה, *head*, מצביע ישירות ל- *NULL*:



5. יתרון הרשימה המקושרת ע"פ מערך רגיל הוא שניתן להקצות דינאמית את הכמות הדרושה בזמן הריצה, בלי לדעת אותה מראש. אז נבדוק מה היתרון של הרשימה ע"פ מערך ***<u>דינאמי</u>***, וגם ע"פ מערך שאינו דינאמי:
   - ניתן להוסיף אברים באמצע הרשימה ללא תלות בגודל הרשימה, ובלי להזיז אברים אחרים. לא מעתיקים את כל הרשימה מחדש כמו ב- *realloc*, אלא רק מעדכנים מצביעים לצומת חדשה.
     אותו דבר אם רוצים להסיר איבר שהוא באמצע הרשימה.
6. חסרונות הרשימה המקושרת ביחס למערך:
   - הגישה לאיבר אינה ישירה ובמערך היא כן ישירה. בנוסף, בהקצאה דינמית לא ניתן להניח שהאברים מוקצים ברצף אחד אחרי השני. לכן גם לא ניתן לבצע על הרשימה, אלגוריתמים באותה הצורה שבה הם פועלים על מערכים. למשל, מיון בועות. (כי אלגוריתמים כאלו מתבססים על כך שאברי המערך נמצאים ברצף. בנוסף, ברשימה יש צורך לשמור את ההצבעה של המצביע בכל איבר לצומת הבאה.)
   - חסרון נוסף: יש צורך במקום נוסף בזיכרון כדי לשמור את המצביעים.
7. נדגיש שוב: לא ניתן לגשת לאיבר של רשימה מקושרת בצורה שניגשים לאיבר במערך, למשל A[n], והאיבר הבא ברשימה אינו A[n+1]. כי רשימה מקושרת זה לא מערך, והאיברים שלה אינה בהכרח נמצאים ברצף בזיכרון כמו במערך.
8. סוגים שונים של רשימות מקושרות: קיימים סוגים שונים של רשימות מקושרות. למשל רשימה מקושרת עם צומת כותר, רשימה מעגלית, דו-כיוונית ועוד.
9. חסרון הרשימה שהוצגה הנ"ל: יש צורך להקצות פעולות מיוחדות לטיפול בהכנסה או בהוצאה של האיבר הראשון. פעולות אלו יהיו שונות מפעולות הכנסה או הוצאה של שאר האברים. ***<u>תרגיל כיתה</u>***: למה?

10. ברשימה עם כותר, הרשימה מכילה קודם כל צומת אחת, צומת כותר, שכל מטרתה היא לשמש כצומת ראשונה, כך שכל הפעולות יבוצעו למעשה על שאר הצמתים. כלומר, צומת הכותר משמשת רק כצומת עזר.

בתרשים הבא, *head* הוא שמה של צומת הכותר כולה, כאשר צומת זו נמצאת בתוך מבנה בשם LIST.



11. בקורס הזה, נשתמש ברשימה עם כותר, מדוגמת תרשים זה. (קיימות אפשרויות שונות לממש רשימות מאותו הסוג.)

12. הגדרת המבנה צומת של רשימה מקושרת בשפת C :

```
typedef int DATA;
typedef struct node
{
    DATA   key;          //מידע מטיפוס כלשהו
    struct    node *next;    //מצביע לצומת הבאה
}NODE;                   //צומת
```

13. הגדרת המבנה של רשימה מקושרת :

```
typedef struct
{
    NODE head;           //צומת כותר שהיא מצביעה לצומת הראשונה האמיתית של הרשימה
}LIST;                   //רשימה מקושרת
```

14. יצירת משתנים מטיפוס הרשימה :

```
LIST mylist;
```

15. שים לב : עבור :

```
LIST mylist;             //יצרתי משתנה מטיפוס הרשימה
```

אז המצביע לצומת הכותר הינו :

```
&mylist.head
```

והמצביע לאיבר הראשון האמיתי ברשימה (לא הכותר) הינו :

```
mylist.head.next;
```

(אם mylist הינו **<u>מצביע</u>** לרשימה, לאחר mylist יהיה -> ולא נקודה.)

16. פעולות האתחול והשחרור של הרשימה : (BOOL - מוגדר כטיפוס בוליאני, אמת/שקר.)

```
BOOL L_init(LIST* pList);          //אתחול רשימה. חובה לאתחל כל משתנה מטיפוס הרשימה
BOOL L_free(LIST* pList);          //שחרור רשימה. חייב לשחרר הקצאות
```

17. פעולות אחרות המוגדרות על הרשימה :

```
NODE* L_insert(NODE* pNode, DATA Value);      //pNode הוספת צומת חדשה לאחר
BOOL L_delete(NODE* pNode);                   //pNode הסרת הצומת שלאחר
```

```
NODE* L_find(NODE* pNode, DATA Value);          //Value מציאת הצומת הראשונה בעלת הערך
int L_print(LIST* pList);                       //הדפסת תוכן הרשימה
```

18. שים לב: פעולות **ההכנסה והמחיקה**, מתבצעות על **<u>האיבר הבא</u>** ברשימה. לכן, יש גם לשים לב, שכאשר **<u>מוחקים</u>** איבר, יש לוודא קודם שבכלל קיים האיבר שאמור להימחק, דהיינו האיבר הבא.
19. כעת נעבור על מודול הרשימה המקושרת המוצג באתר הקורס, נראה את מימוש הפעולות המוגדרות על הרשימה, כמו גם דוגמא לשימוש ברשימה.
20. מה היעילות של כל אחת מהפעולות המוגדרות על הרשימה?
21. סריקה ועדכון של רשימה מקושרת: בד"כ נשתמש באחת משתי התבניות הבאות:

דוגמא לעבודה עם לולאת ***for***:

```
void scanList(LIST * list)
{
   NODE  *p;
   //מתחיל כמצביע לאיבר הראשון, כלומר הראשון שאחרי הכותר, ומתקדם על אברי הרשימה p
   for(p=list->head.next;p!=NULL;p=p->next)
   {
      //p->key: גישה לשדה המידע שבצומת
      //…עבודה על כל איבר ואיבר
   }
}
```

או ע"י עבודה עם לולאת ***while***. לדוגמא:

```
void scanList(LIST * list)
{
   NODE *p = list->head.next;        //מצביע לאיבר הראשון, כלומר הראשון שאחרי הכותר
   while(p!=NULL)
   {
      //p->key: גישה לשדה המידע שבצומת
      //…עבודה על כל איבר ואיבר
      p=p->next;                     //מתקדם להצביע על האיבר הבא שברשימה
   }
}
```

22. <u>תרגיל כיתה</u>: מה פלט התוכנית הבאה:

```
void main()
{
   LIST list;
   NODE   *p;
   L_init(&list);               //אתחול
   L_insert(&list.head, 10);
   p = L_insert(&list.head, 20);
   L_insert(&list.head , 30);
   L_insert(p , 40);
   L_delete(p->next);
   L_delete(list.head.next);
   L_print(&list);              //הדפסה
   L_free(&list);               //שחרור
}
```

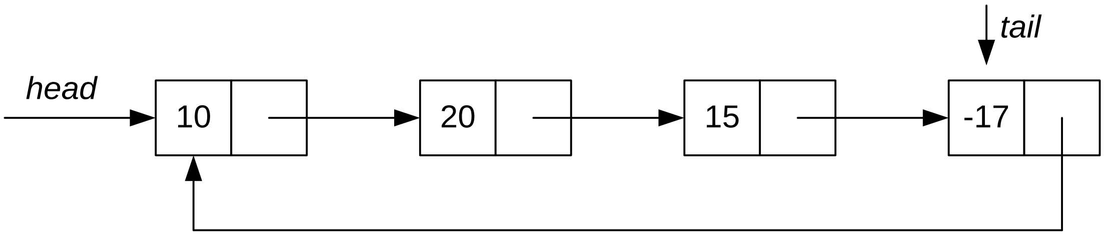
23. <u>תרגול</u> - נציג כעת בכיתה תרגילי כיתה נוספים.

מפגש מס' 5. נושא המפגש: רשימות מקושרות - חלק ב': רשימות מעגליות, רשימות דו-כיווניות ודו-מקושרות.

1. רשימה מעגלית: (circularly linked list) היא רשימה רגילה אלא שהמצביע של האיבר האחרון שלה מצביע לאיבר הראשון ולא ל- *NULL*. גם כאן קיים המצביע *head*, ובנוסף, בד"כ גם נוסיף עוד מצביע בשם *tail* שיצביע תמיד על הצומת האחרונה.

   בכל פעולה שנבצע על הרשימה, נעדכן את המצביע *tail* כך שיצביע תמיד על הצומת האחרונה.

   דוגמא לרשימה מעגלית (ללא כותר):



2. כאשר הרשימה ריקה המצביעים *tail* ו- *head* מצביעים ל- *NULL*. (ואין מצביע בין האיבר האחרון לראשון, כי אין צמתים.)

   אם יש בה רק איבר אחד, המצביע שלו מצביע לעצמו, וגם המצביעים *tail* ו- *head* מצביעים לאיבר יחיד זה.

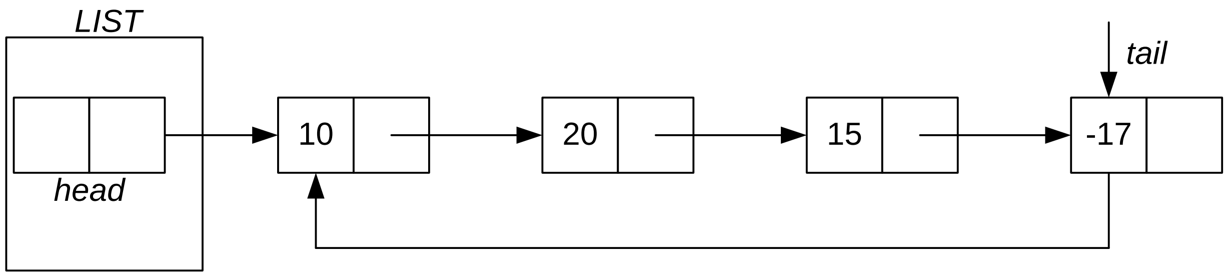
3. קיימים מימושים של רשימה מעגלית רק עם המצביע *tail* וללא המצביע *head* כלל.

   תרגיל כיתה: האם גם ברשימה חד כיוונית ניתן לעבוד רק עם המצביע *tail*?

4. יתרונותיה של רשימה מעגלית:
   - אפשר להגיע מכל איבר לכל איבר.
   - אם קיים המצביע *tail*, ניתן לשרשר רשימות מעגליות בזמן קבוע. (כלומר ללא תלות בגודל הרשימה.)

   הערה: ניתן גם ברשימה חד-כיוונית להוסיף את המצביע *tail*, ואז לשרשר רשימות חד-כיווניות מסוג זה בזמן קבוע.

5. רשימה מעגלית עם צומת כותר: גם כאן ניתן להוסיף צומת כותר. לדוגמא:



6. קיימים גם מימושים בהם הצומת האחרונה מצביעה לצומת הכותר במקום לאיבר הראשון.

7. מימוש רשימה מעגלית עם צומת כותר: הגדרת הרשימה:

```
typedef struct
{
   NODE head;          //צומת כותר
   NODE *tail;         //המצביע לאיבר האחרון
}CLIST;                //רשימה מעגלית
```

8. שים לב, שכעת בחלק מהפונקציות, יש צורך להוסיף מצביע גם לרשימה עצמה, וזאת כדי שיהיה ניתן לעדכן את המצביע *tail*. למשל: פונקצית ההכנסה של איבר תיראה כעת כך:

```
NODE* CL_insert(CLIST* pList, NODE* pNode, DATA Value);
```

9. <u>תרגיל כיתה 1</u>: נתונה רשימה מקושרת בעלת מצביע *tail*, המכילה לפחות איבר אחד.
יתכן והרשימה היא מעגלית באופן שהאיבר האחרון אולי מצביע לאחד האברים הקודמים, או שיתכן והרשימה אינה מעגלית.
א. הצע דרך לבדוק זאת.
ב. הצע דרך לבדוק זאת, באם נתון שברשימה לא קיים המצביע *tail*.
<u>פתרון</u>:
א.אם tail->next==NULL הרשימה אינה מעגלית.
ב. אם לא קיים המצביע *tail*: נרוץ על הרשימה עם 2 מצביעים: איטי - מתקדם רשומה אחת כל פעם, מהיר - מתקדם שתי רשומות בכל פעם.
אם המהיר הגיע ל- NULL: אין מעגליות.
אם המהיר אינו משיג יותר את האיטי - יש מעגליות.

10. <u>תרגיל כיתה 2</u>: נתונים אנשים המסודרים במעגל.
החל מהאדם הראשון, סופרים *n* אנשים בכל פעם, כאשר האדם ה- *n*-י יוצא מהמשחק והולך הביתה. מנצח המשחק הוא זה שנשאר האחרון. הצע מבנה נתונים ואלגוריתם מתאים כדי למצוא את המנצח.
<u>פתרון</u>:
נשתמש במבנה הנתונים של רשימה מעגלית.
1. נכניס את מספרי האנשים לרשימה לפי הסדר.
2. התחל מראש הרשימה.
3. כל עוד יש ברשימה יותר מאיבר אחד, בצע:
3.1 הוצא מהרשימה את האיבר ה- *n*-י החל מהמקום הבא ברשימה.

11. <u>תרגיל כיתה 3/או בתרגול</u>: ממש את פונקצית האתחול של מודול הרשימה המעגלית בעלת צומת כותר. (ובעלת מצביע *tail*.) קרא לפונקציה בשם CL_init. (בצע כך, שהמצביעים יצביעו לצומת הכותר ולא ל- *NULL*.)

12. <u>תרגיל כיתה 4/או בתרגול</u>: ממש את פונקצית ההכנסה של הרשימה המעגלית הנ"ל. קרא לפונקציה בשם CL_insert. (כנ"ל.) עבוד לפי האלגוריתם הבא:
- בצע קודם את עדכון המצביעים כרגיל.
- כעת בדוק: אם *pNode* מצביע על הצומת האחרונה, אז יש לעדכן את *tail* כך שיצביע לצומת החדשה.
- אם הכנסנו איבר חדש ראשון או אחרון, ההצבעה המעגלית משתבשת. לכן נבצע הצבה, של האיבר האחרון כך שיצביע על הראשון. (אין צורך לבדוק אם אכן חל השיבוש.)

13. תרגול - נציג כעת בכיתה תרגילי כיתה נוספים בנושא של רשימה מעגלית, כולל מימוש המודול של הרשימה המעגלית. (עם צומת כותר.)

14. תרגיל כיתה 5/או בתרגול: לאחר השלמת מודול הרשימה המעגלית שבנית, ממש כעת את הפונקציה עבור המשחק מתרגיל כיתה 1. הדפס את תוכן הרשימה בסיום המשחק.

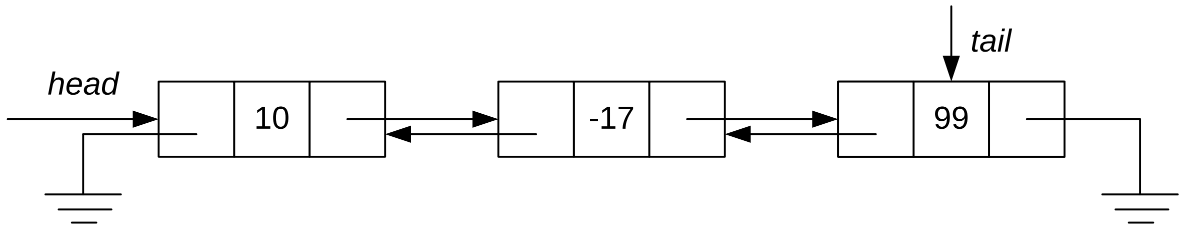
15. רשימות דו- כיוונית: (doubly linked list): ברשימה כזו לכל צומת יש מצביע גם לאיבר שלפניו:

```
typedef struct node
{
    DATA   key;           //מידע מטיפוס כלשהו
    struct    node *next;   //מצביע לצומת הבאה
    struct    node *prev;   //מצביע לצומת הקודמת
}NODE;                    //צומת
```

גם כאן שומרים מצביע *tail* לאיבר האחרון של הרשימה.

כאשר אין צומת כותר, באיבר הראשון המצביע *prev* מצביע ל- *NULL*.

דוגמא לרשימה דו-כיוונית ללא כותר:



16. היתרונות של רשימה דו-כיוונית:

- תנועה בכל הכיוונים.
- אפשר להוציא את האיבר ש- *pNode* מצביע אליו ולא רק את זה שאחריו.
- אפשר להכניס איבר חדש גם מימין וגם משמאל ביחס לאיבר ש- *pNode* מצביע אליו.

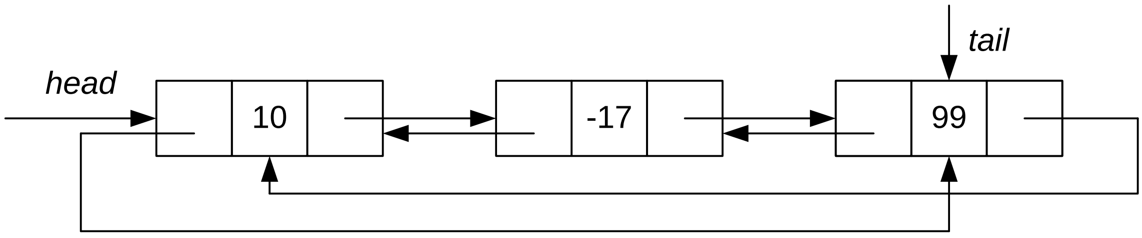
חסרונות:

- עבודה עם מצביע נוסף בכל פעם, ותפיסה של מקום נוסף בכל מבנה עבורו.

17. רשימה דו-מקושרת: (doubly circular linked list): היא רשימה דו-כיוונית מעגלית. כלומר המצביע *next* של הצומת האחרונה מצביע לצומת הראשונה, והמצביע *prev* של הצומת הראשונה מצביע לצומת האחרונה. אם יש איבר אחד אז כל המצביעים שלו מצביעים לעצמו. בנוסף, כמו ברשימה מעגלית, גם כאן יהיו לנו את המצביעים *head* ו- *tail* כשדות במבנה הרשימה. רשימה דו-מקושרת כזו היא בעצם שתי רשימות מעגליות: אחת עם כיוון השעון והשנייה בכיוון המנוגד לכיוון השעון. גם כאן ישנם מימושים שונים, כמו למשל מימוש עם צומת כותר, כאשר המצביע *next* של הצומת האחרונה, והמצביע *prev* של הצומת הראשונה, מצביעים לכותר, וכו'.

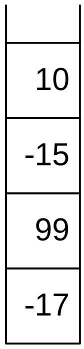
18. יתרונותיה של הרשימה הדו-מקושרת: שילוב היתרונות של הרשימה המעגלית והרשימה הדו-כיוונית. כלומר, גם הגעה מהירה יותר מאיברים הנמצאים בסוף הרשימה לאלו שבתחילתה למשל, ושאר היתרונות האחרים אותן ציינו קודם לכן עבור שני סוגי הרשימות.

דוגמא לרשימה דו מקושרת (ללא כותר):

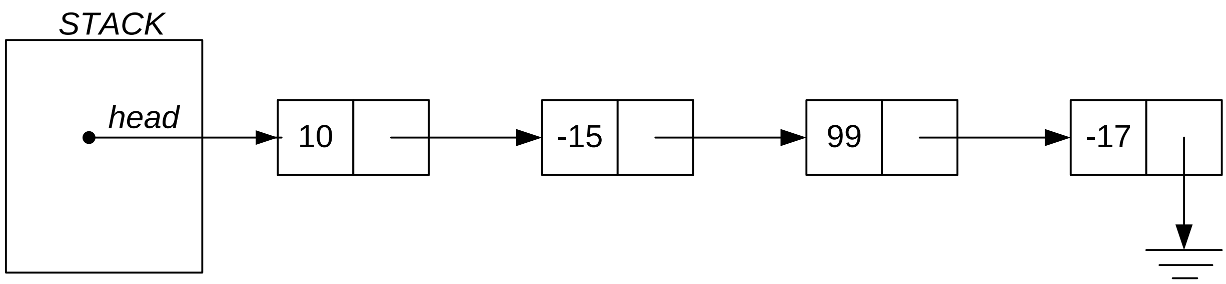


מפגש מס' 6. נושא המפגש: מחסנית.

1. המחסנית (stack) היא מבנה נתונים המדמה מחסנית אמיתית. למחסנית יש רק פתח **אחד** לצורך הכנסה או הוצאה של נתונים. הכנסת הנתונים למחסנית מתבצעת כך שהנתון שנכנס ראשון נמצא בתחתית המחסנית, והנתון שנכנס אחרון נמצא בראש המחסנית,ליד הפתח, והוא זה שיצא ראשון. שיטה זו מקיימת את העיקרון הנקרא ***LIFO***, כלומר מה שנכנס אחרון יוצא ראשון. דוגמא למצב המחסנית לאחר הכנסת הנתונים הבאים:
-17, 99, -15, 10 (ההכנסה מבוצעת משמאל לימין.)



2. המחסנית יכולה להכיל נתונים רבים בזמן מסוים, כלומר את כל הנתונים שהכנסנו אליה. כלומר המחסנית היא בעצם גם מאגר שמירה של קבוצת נתונים.
3. למחסנית יש תכונה של הפיכת סדר הנתונים בעת הוצאתם, מהסדר שבו הם הוכנסו. לכן נעזרים במבנה נתונים זה במצבים בהם נדרשת הפיכה של מידע, בדיקה של סימטריות וכו'.
4. נממש את המחסנית בדומה לאופן שבו מימשנו את הרשימה המקושרת החד- כיוונית. שים לב, שבמימוש המוצג כאן, ההגדרה של המבנה מחסנית מעט שונה ממקודם. (אין צומת כותרת אלא מצביע לאיבר הראשון.) (פעולות ההכנסה וההוצאה על המחסנית מתבצעות רק על האיבר הראשון, כך שלא צריך פונקציות אחרות לאיבר שאינו ראשון.)



5. הגדרת המבנה צומת של מחסנית: נשתמש באותה ההגדרה שראינו עבור הרשימה המקושרת:

```
typedef int DATA;
typedef struct node
{
   DATA   key;          //מידע מטיפוס כלשהו
   struct    node *next;   //מצביע לצומת הבאה
}NODE;                   //צומת
```

6. הגדרת המבנה מחסנית: כאמור קודם, נגדיר כאן באופן מעט שונה להגדרה שעשינו ברשימה:

```
typedef struct{
    NODE* head;            //מצביע לצומת הראשונה
}STACK;
```

7. דוגמה ליצירת משתנים מטיפוס המחסנית:

```
STACK mystack1, mystack2;
```

8. שים לב: אם pStk הינו <u>מצביע</u> למחסנית, אזי pStk->head הינו המצביע לצומת הראשונה של המחסנית.

9. הפעולות שנגדיר עבור המחסנית:

```
BOOL S_init(STACK* pStk);           //אתחל מחסנית
BOOL S_isEmpty(STACK* pStk);        //האם המחסנית ריקה
BOOL S_push(STACK* pStk, DATA Value);          //דחוף איבר חדש בראש המחסנית
DATA S_pop(STACK* pStk);            //הוצא/שלוף את האיבר שבראש המחסנית והחזר את ערכו
DATA S_top(STACK* pStk);            //החזר את הערך שבראש המחסנית
void S_free(STACK* pStk);           //שחרור המחסנית
BOOL S_print(STACK* pStk);          //הדפס תוכן המחסנית החל מראש המחסנית
```

10. שים לב: בכל הפעולות הארגומנט היחיד יהיה המצביע למחסנית, מלבד בפעולה דחוף. (S_push)

11. <u>**תרגיל כיתה**</u>: תאר בקצרה כיצד תממש את כל אחת מהפעולות של המחסנית.

12. כעת נעבור על מודול המחסנית המוצג באתר הקורס, נראה את מימוש הפעולות המוגדרות על המחסנית, כמו גם דוגמה לשימוש במחסנית.

13. מצא מה הסיבוכיות של כ"א מהפעולות על המחסנית.

14. מימוש מבנה הנתונים מחסנית בעזרת <u>מערך</u>: ניתן לממש את המחסנית גם ללא שימוש בצמתים ומצביעים, אלא ע"י שימוש במערך. לדוגמא: נגדיר את המבנה מחסנית:

```
typedef struct
{
    DATA* key;             //מצביע למערך דינאמי ובו איברי המחסנית
    int top;               //סמן שמחזיק את האינדקס במערך, של האיבר שבראש המחסנית
    int N;                 //כמות האיברים במערך הדינאמי שבו איברי המחסנית
}STACK_A;
```

15. נציג כעת מימוש לפונקצית האתחול של המחסנית, ומימוש לפונקציה לדחיפת איבר למחסנית:

```
BOOL SA_init(STACK_A* pStk, int N)
{
   if(!pStk || !(pStk->key = (DATA*)malloc(N * sizeof(DATA))))
      return False;
   pStk->top = -1;
   pStk->N = N;
   return True;
}

BOOL SA_push(STACK_A * pStk, DATA x)
{
   if(! pStk || pStk->top == pStk->N -1)
      return False;
   pStk->top++;
   pStk->key[pStk->top] = x;
   return True;
}
```

16. שים לב: הפעולות על המחסנית מתבצעות לוגית ולא פיזית על המערך.

למשל: הוצאת איבר היא לא מחיקתו פיזית אלא הזזת הסמן top.

17. יתרונות המימוש עם מערך לעומת מימוש כרשימה: המימוש במערך אינו דורש שמירת מקום עבור המצביעים, ובנוסף מימוש פשוט וללא הקצאות דינאמיות. חסרונות: צריך לדעת מראש את הגודל המרבי הנדרש, ובזבוז מקום כאשר לא משתמשים בכולו.

18. בקורס זה נשתמש במימוש כרשימה מקושרת אם לא נאמר אחרת.

19. <u>תרגיל כיתה 1/או בתרגול</u>: השלם את המימוש של מודול זה. (מימוש מחסנית בעזרת מערך.)

20. <u>תרגיל כיתה 2/או בתרגול</u>: נתון ביטוי המכיל סוגריים פותחות וסוגרות. בדוק בעזרת מחסנית האם לכל סוגריים פותחות יש סוגריים סוגרות באופן תואם. כתוב אלגוריתם מתאים.

<u>פתרון</u>:

1. צור ואתחל מחסנית S.
2. כל עוד ישנם תווים בביטוי: החל מהתו הראשון, לכל תו c בביטוי, בצע:
   1.1 אם c הוא סוגר פתיחה, דחוף אותו למחסנית S.
   1.2 אחרת, אם c הוא סוגר סגירה:
      1.2.1 אם המחסנית S ריקה, שחרר את המחסנית S והחזר שקר.
      1.2.2 אחרת, שלוף את ראש המחסנית S.
3. אם המחסנית S ריקה, אזי שחרר את המחסנית S והחזר אמת.
4. אחרת שחרר את המחסנית S והחזר שקר.

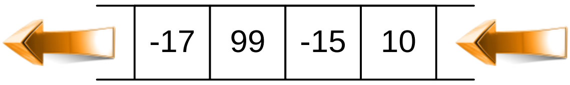
21. <u>תרגיל כיתה 3/או בתרגול</u>: כתוב תוכנית שתקלוט ממשתמש מחרוזת, ותוך מימוש האלגוריתם הנ"ל, תודיע האם סדר הסוגריים תקין.

22. <u>תרגול</u> - נציג כעת בכיתה תרגילי כיתה נוספים.

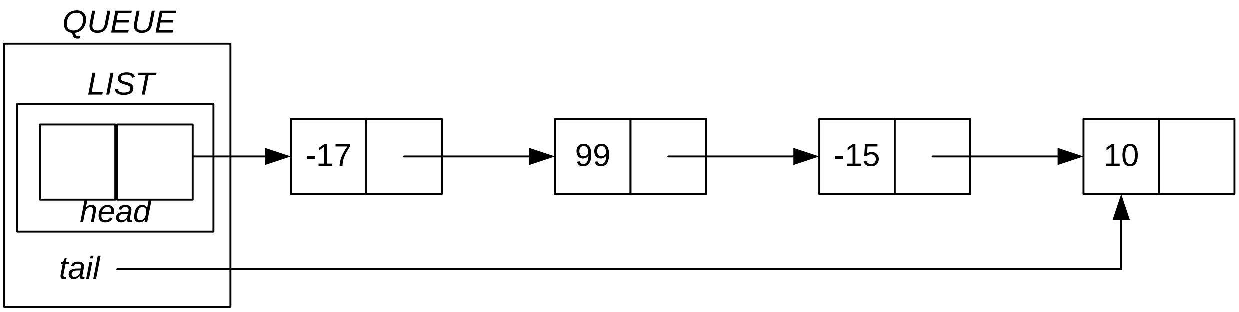
מפגש מס' 7. נושא המפגש: תור.

1. תור (queue) הוא מבנה נתונים המטפל בנתונים לפי סדר הגעתם. למבנה תור יש **<u>שני פתחים</u>**: כניסה ויציאה. נתונים נכנסים בפתח הכניסה ויוצאים מפתח היציאה. התור לפיכך, מקיים את העיקרון הנקרא ***FIFO***, כלומר נכנס ראשון יוצא ראשון. כמו למשל תור של אנשים. דוגמא למצב התור לאחר הכנסת הנתונים הבאים:

-17, 99, -15, 10 (יש לקרוא את המספרים משמאל לימין.)



2. התור יכול להכיל נתונים רבים בזמן מסוים, כלומר את כל הנתונים שהכנסנו אליו. כלומר התור הוא בעצם גם מאגר שמירה של קבוצת נתונים.
3. שימושים של תור: טיפול בנתונים לפי סדר הגעתם. למשל: שמירת תור גישה של תהליכים למדפסת, וכו'.
4. מימוש של תור כרשימה מקושרת: נממש הפעם את התור, ע"י שימוש במודול הרשימה המקושרת החד-כיוונית. שים לב, שמבנה התור, כולל מבנה של הרשימה, וגם מצביע *tail* שיצביע על האיבר האחרון של הרשימה:



5. כאשר התור ריק, המצביע *tail* יצביע לצומת הכותר של מבנה הרשימה.
6. הגדרת המבנה צומת של תור: נשתמש באותה ההגדרה שראינו עבור הרשימה המקושרת:

```
typedef int DATA;
typedef struct node
{
    DATA   key;            //מידע מטיפוס כלשהו
    struct   node *next;   //מצביע לצומת הבאה
}NODE;                     //צומת
```

7. הגדרת המבנה תור: כאמור קודם, המבנה כולל שני שדות: מבנה של רשימה, ומצביע *tail* לאיבר האחרון ברשימה:

```
typedef struct{
    LIST Q;                //רשימה מקושרת עם צומת כותר
    NODE *tail;            //מצביע לאיבר האחרון ברשימה
}QUEUE;
```

8. דוגמא ליצירת משתנים מהטיפוס תור:

```
QUEUE myq1, myq2;
```

9. שים לב: אם pQue הינו <u>מצביע</u> לתור, אזי pQue->Q.head.next הינו המצביע לצומת הראשונה של התור. ו- pQue->tail הינו המצביע לצומת האחרונה של התור.

10. הפעולות שנגדיר עבור התור:

```
BOOL Q_init(QUEUE* pQue);          //אתחול תור
BOOL Q_isEmpty(QUEUE* pQue);  //האם התור ריק
BOOL Q_enque(QUEUE* pQue, DATA Value);        //הכנס לסוף התור
DATA Q_deque(QUEUE* pQue);      //הוצא את האיבר שבראש התור והחזר את ערכו
DATA Q_head(QUEUE* pQue);        //החזר את ערך האיבר שבראש התור
void Q_free(QUEUE* pQue);           //שחרור התור
BOOL Q_print(QUEUE* pQue);        //הדפסת תוכן התור החל מראש התור
```

11. שים לב: בכל הפעולות הארגומנט היחיד יהיה המצביע לתור, מלבד בפעולה הכנס. (Q_enque)

12. **<u>תרגיל כיתה</u>**: תאר בקצרה כיצד תממש את כל אחת מהפעולות של התור.

13. כעת נעבור על מודול התור המוצג באתר הקורס, נראה את מימוש הפעולות המוגדרות על התור, כמו גם דוגמא לשימוש בתור.

14. מצא מה הסיבוכיות של כ"א מהפעולות על התור.

15. מימוש מבנה הנתונים תור בעזרת <u>מערך</u>: ניתן לממש את התור גם ללא שימוש בצמתים ומצביעים, אלא ע"י שימוש במערך. לדוגמא: נגדיר את המבנה תור:

```
typedef struct
{
   DATA *key;               //מצביע למערך דינאמי ובו איברי התור
   int head;       //סמן שמחזיק את האינדקס במערך, של האיבר שבראש התור (זה שייצא ראשון)
   int tail;                    //סמן שמחזיק את האינדקס במערך, של התא הבא הפנוי בסוף התור
   int counter;             //כמה איברים יש כרגע בתור
   int N;                       //כמות האיברים במערך הדינאמי שבו איברי התור
}QUEUE_A;
```

16. שים לב: התור מתחיל מהסמן *head* ועד לסמן *tail* (לא כולל), בצורה מעגלית בכיוון השעון. הפעולות על התור מתבצעות באופן לוגי ולא פיזי. למשל: הוצאת איבר היא לא מחיקתו פיזית אלא הזזת הסמן המתאים. לדוגמא: מצב התור הנ"ל לאחר שביצענו פעמיים פעולה של הוצאה מראש התור: (17- ו- 99 הפכו לזבל.)

-17	99	-15	10		

head *tail*

אם כעת נכניס לתור את הערכים 1, 2, ולבסוף את 3, מצב התור יהיה:

3	99	-15	10	1	2

tail *head*

17. נציג כעת מימוש לפונקצית האתחול של התור, ומימוש לפונקציה להכנסת איבר בסוף התור:

```
BOOL QA_init(QUEUE_A* pQue , int N)
{
   if(!pQue || !(pQue->key = (DATA*)malloc(N * sizeof(DATA))))
      return False;
   pQue->head = 0;
   pQue->tail = 0;
   pQue->counter = 0;
   pQue->N = N;
   return True;
}
BOOL QA_enque(QUEUE_A* pQue, DATA Value)
{
   if ( !pQue || pQue->counter ==pQue->N) return False;
   pQue->key[pQue->tail] = Value;
   pQue->tail++;
   if(pQue->tail == pQue->N)
      pQue->tail = 0;
   pQue->counter++;
   return True;
}
```

18. יתרונות המימוש עם מערך לעומת מימוש כרשימה: המימוש במערך אינו דורש שמירת מקום עבור המצביעים, ובנוסף מימוש פשוט וללא הקצאות דינאמיות. חסרונות: צריך לדעת מראש את הגודל המרבי הנדרש, ובזבוז מקום כאשר לא משתמשים בכולו.

19. בקורס זה נשתמש במימוש כרשימה מקושרת אם לא נאמר אחרת.

20. <u>תרגיל כיתה 1/או בתרגול</u>: השלם את המימוש של מודול זה. (מימוש תור בעזרת מערך.)

21. <u>תרגיל כיתה 2/או בתרגול</u>: מימוש עם מערך: מה פלט ה- *main* הנ"ל? הצג את מצב התור ושדות המבנה של התור בכל שלב.

```
void main()
{
   int y;
   QUEUE_A q;
   QA_init (&q , 8);
   QA_enque (&q,10);
   QA_enque (&q,20);
   QA_enque (&q,30);
   QA_deque (&q);
   QA_deque (&q);
   QA_enque (&q,100);
   QA_deque (&q);
   y= Q_head (&q);
   printf("%d\n",y);
   QA_free (&q);
}
```

22. <u>תרגיל כיתה 3/או בתרגול</u>: באתר של חברה מסוימת, במערכת רישום ההזמנות של לקוחות, מקבלים בקשות של לקוחות לרכישת מוצר. כל לקוח יכול לבקש כמות מסוימת מהמוצר, אחד או יותר.
לאחר קבלת כל הבקשות, מעדכנים סופית את כמות המוצרים שבמלאי למכירה. אספקת המוצר ללקוחות הינה לפי סדר הגעת הבקשות למערכת הרישום. הצע אלגוריתם מתאים שידפיס מי הם הלקוחות שיקבלו את המוצר ומה הכמות שיקבל כ"א מהם.

<u>פתרון</u>:

1. צור תור Q, ובצע init(Q)
2. לכל לקוח נחזיק רשומה Customer, ובה שדות המתארים את פרטיו ובנוסף גם שדה amount המתאר את הכמות שהוא מעוניין לרכוש מהמוצר.
3. sum ← 0
4. כל עוד מגיעות עוד בקשות, בצע enque(Q ,Customer) לכל בקשה.
5. כמות המוצרים הסופית שבמלאי ← N
6. כל עוד התור אינו ריק וגם sum < N, בצע:
   6.1 sum = sum + amount(head(Q))
   6.2 C ← deque(Q) והדפס את פרטי הלקוח C. (אם sum > N הדפס את הכמות החלקית שיקבל לקוח זה.)
7. בצע free(Q)

23. <u>תרגיל כיתה 4/או בתרגול</u>: כתוב אלגוריתם המקבל תור שאבריו שונים זה מזה. על האלגוריתם להחזיר את כמות האברים שיש בתור. בגמר הפעולה התור צריך להיות ללא שינוי. (הערה: תמיד יתכן גם שמבנה הנתונים ריק, כלומר שהתור ריק במקרה זה.)

<u>אילוצים</u>:

- מותר להשתמש רק בפונקציות שהוגדרו במיוחד עבור התור, וללא ידיעה כיצד מומש התור.
  (שים לב: הערה זו תקפה תמיד בכל הקורס.)
- אסור לך להשתמש במבנה נתונים אחר כעזר, למשל אסור להשתמש במערך עזר, וכו'.
  תמיד אסור (בכל הקורס) להשתמש במערך עזר או במשתנים גלובליים אלא אם נאמר אחרת.

<u>פתרון</u>:

countQ (q)

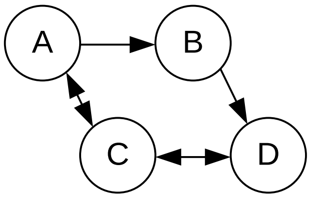
```
counter ← 0
if  isEmpty (q)
  then return  0
temp ← head (q)
repeat
  counter ← counter +1
  enque (q , deque (q))
until head (q) ≠ temp
return counter
```

24. <u>תרגיל כיתה 5/או בתרגול</u>: כתוב תוכנית ובה תממש את האלגוריתם הנ"ל.

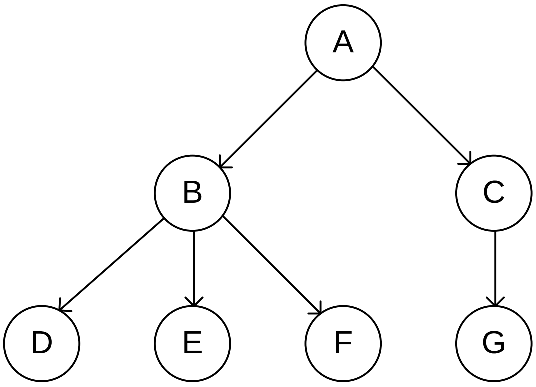
25. <u>תרגול</u> - נציג כעת בכיתה תרגילי כיתה נוספים.

מפגש מס' 8. נושא המפגש: עצים בינאריים.

1. גרף (graph) הוא אוסף של קדקודים/צמתים וצלעות/קשתות (edges) המחברות ביניהם.
2. בגרף מכוון (digraph / directed graph)לכל צלע יש כיוון:

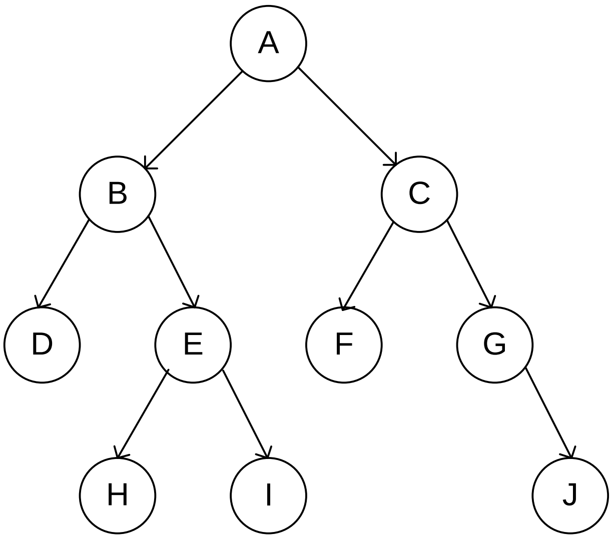


3. עץ מכוון הוא גרף מכוון ללא מעגלים, ואשר לו מקור אחד הנקרא שורש.
(מקור - צומת שאף צומת לא מצביעה אליו.)



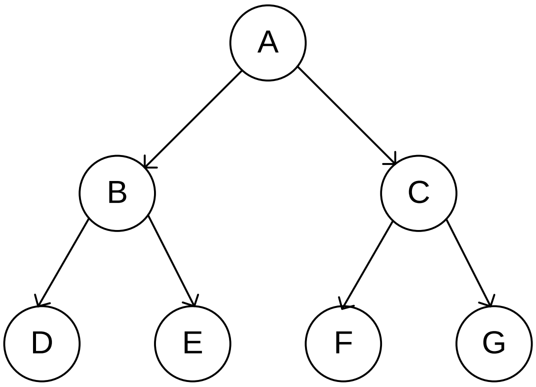
4. לכל צומת פרט לשורש יש אב אחד בדיוק.
5. קודקוד (vertex) V הוא צאצא של A אם קיים מסלול מכוון מצומת A ל- V.
6. תת העץ של גרף, ששורשו הוא V: הוא העץ המכוון ששורשו הוא V עצמו.
7. דרגה של צומת V: מספר הבנים של V.
8. עלה: צומת שאין לה בנים. למשל: D, E, F, G בתרשים הנ"ל.
9. צומת פנימית: צומת שהיא לא עלה.
10. עומק של צומת V: כמה קשתות יש מהשורש עד לצומת V. למשל, העומק של הצומת G בתרשים הנ"ל הוא 2.
11. רמה x בעץ, היא כל הצמתים בעלי עומק x.
12. גובה של צומת V: כמה קשתות יש מהצומת V עד לצאצא/עלה הרחוק ממנה ביותר. למשל: הגובה של הצומת A בתרשים הנ"ל הוא 2.
13. גובה העץ: גובה השורש.

14. עץ בינארי (binary tree): עץ שבו לכל צומת שאינו עלה יש בן שמאלי ו/או בן ימני בלבד, כלומר הדרגה של כל צומת היא 2 לכל היותר.



15. עץ בינארי מלא (full): לכל צומת פנימית יש שני בנים, כלומר הדרגה של כל צומת פנימית היא 2 בדיוק.

16. עץ בינארי שלם (complete): עץ בינארי מלא ובו כל העלים באותו העומק.



17. בעץ בינארי **שלם** שיש לו n צמתים, L עלים, וגובה h, מתקיים:

מספר הצמתים בעומק i: $n_i = 2^i$

מספר העלים בעץ: $L = 2^h$

מספר הצמתים בעץ: $n = 2^{h+1} - 1$

הגובה של העץ: $h = \log_2(n+1) - 1$

18. סיור בעצים בינאריים: מבוצע באופן רקורסיבי.

Inorder: הסדר הוא שמאל ← שורש ← ימין.

(סייר בתת העץ השמאלי, בקר בשורש, סייר בתת העץ הימני)

Preorder: הסדר הוא שורש ← שמאל ← ימין.

(בקר בשורש, סייר בתת העץ השמאלי, סייר בתת העץ הימני)

Postorder: הסדר הוא שמאל ← ימין ← שורש.

(סייר בתת העץ השמאלי, סייר בתת העץ הימני, בקר בשורש)

19. הסיבוכיות של כל אחת משיטות הסיור היא $O(n)$. (עוברים על כל צומת פעם אחת בלבד).

20. <u>תרגיל כיתה 1</u>: מה סדר הסריקה בכל אחת מהשיטות, בדוגמת העץ שבתרשים הנ"ל?

21. מימוש עץ בינארי בעזרת מערך: ניתן לשמור עץ בינארי גם במערך

בשיטה זו, האב יהיה באינדקס i, הבן השמאלי יהיה באינדקס $2*i+1$, והבן הימני יהיה באינדקס $2*i+2$.

אם נתון אינדקס של בן, ניתן להגיע לאינדקס של האב לפי: $\left\lfloor \frac{i-1}{2} \right\rfloor$.

22. תרגיל כיתה 2: צייר את המערך המתאים לעץ שבתרשים הנ"ל.

23. יתרונות המימוש עם מערך לעומת מימוש כרשימה: המימוש במערך אינו דורש שמירת מקום עבור המצביעים, ובנוסף מימוש פשוט וללא הקצאות דינאמיות. חסרונות: צריך לדעת מראש את הגודל המרבי הנדרש, ובזבוז מקום כאשר העץ אינו שלם ו/או כמות הצמתים בעץ קטנה מהגודל המירבי.

24. מימוש עץ בינארי עם מצביעים - נקצה דינמית כל צומת קיימת בעץ, ולא נבזבז מקום לצמתים שאינן קיימים, כאשר העץ אינו שלם.

25. הגדרת המבנה צומת/קודקוד של עץ בינארי: (ההגדרה כאן שונה במעט מההגדרה של צומת רשימה מקושרת.)

```
typedef int DATA;
typedef struct node
{
   DATA   key;                      //מידע מטיפוס כלשהו
   struct    node2 *left , *right;  //מצביעים לבנים
}NODE2;                             //צומת/קודקוד של עץ בינארי
```

26. הגדרת המבנה עץ: המבנה כולל שדה אחד בלבד, והוא צומת כותר, בדומה לצומת הכותר שהיה לנו ברשימה המקושרת. השדה *next* של צומת הכותר כאן, יצביע על שורש העץ.

```
typedef struct{
   NODE2 head;          //צומת כותר. הצומת הוא מהטיפוס צומת של עץ בינארי
}TREE;
```

27. דוגמא ליצירת משתנים מהטיפוס עץ:

```
TREE t1, t2;
```

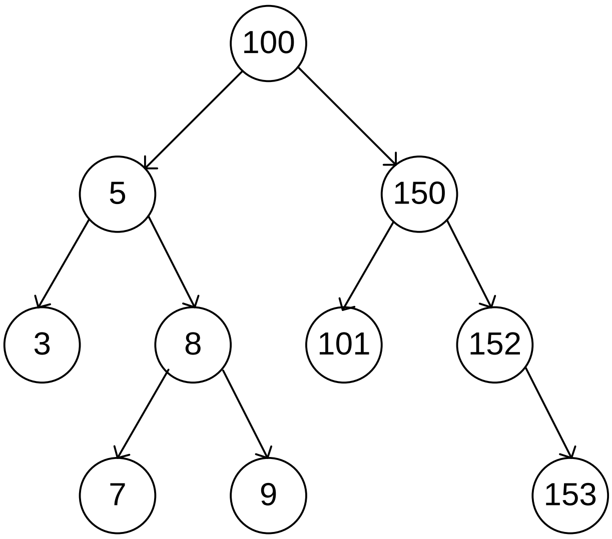
28. שים לב: אם T הינו <u>מצביע</u> לעץ, אזי T->head.left הינו המצביע לצומת של העץ.

(לא עושים שימוש ב- T->head.right, והוא ישאר עם ערך זבל.)

29. במפגש הבא (עצי חיפוש), נעבור על הפעולות שנגדיר עבור העץ, ונעבור על מודול העץ המוצג באתר הקורס. נראה את מימוש הפעולות המוגדרות על העץ, כמו גם דוגמא לשימוש בעץ.

מפגש מס' 9. נושא המפגש: עצי חיפוש.

1. עץ חיפוש בינארי (binary search tree) הוא עץ בינארי שבכל צומת יש מפתח ייחודי, ובנוסף הצמתים בו ממוינים באופן הבא:
   הערך בכל צומת/קודקוד, גדול מערכי כל אחד מהצמתים שנמצאים בתת העץ השמאלי של צומת זו.
   הערך בכל צומת/קודקוד, קטן מערכי כל אחד מהצמתים שנמצאים בתת העץ הימני של צומת זו.
2. כלומר, בכל צומת גם מתקיים שאם יש לה בנים, אז הערך בבן השמאלי שלה יותר קטן מהערך שבה, והערך שיש בבן הימני שלה יותר גדול מהערך שיש בה.



3. מבנה נתונים זה מאפשר למצוא נתון במהירות גבוהה יותר מאשר במבנה ליניארי כגון רשימה מקושרת.
4. <u>תרגיל כיתה 1</u>: צייר את העץ המתקבל עבור הכנסת הנתונים הבאים לעץ ריק: 5, 1, 8, 6, 10, 3 (משמאל לימין).
5. <u>תרגיל כיתה 2</u>: היכן נמצא הערך הכי קטן בעץ חיפוש?
6. <u>תרגיל כיתה 3</u>: היכן נמצא הערך הכי גדול בעץ חיפוש?
7. <u>תרגיל כיתה 4</u>: כיצד ניתן לדעת האם עץ בינארי הוא עץ חיפוש?
8. מימוש עץ חיפוש בינארי עם מצביעים: נחזור שוב על ההגדרות הכלליות של עץ בינארי, מהמפגש הקודם:
9. הגדרת המבנה צומת/קודקוד של עץ בינארי: (ההגדרה כאן שונה במעט מההגדרה של צומת רשימה מקושרת.)

```
typedef int DATA;
typedef struct node
{
   DATA   key;                        //מידע מטיפוס כלשהו
   struct    node2 *left , *right;    //מצביעים לבנים
}NODE2;                               //צומת/קודקוד של עץ בינארי
```

10. הגדרת המבנה עץ: המבנה כולל שדה אחד בלבד, והוא צומת כותר, בדומה לצומת הכותר שהיה לנו ברשימה המקושרת. השדה *next* של צומת הכותר כאן, יצביע על שורש העץ.

```
typedef struct{
   NODE2 head;                        //צומת כותר. הצומת הוא מהטיפוס צומת של עץ בינארי
}TREE;
```

11. דוגמא ליצירת משתנים מהטיפוס עץ:

```
TREE t1, t2;
```

12. שים לב: אם T הינו <u>מצביע</u> לעץ, אזי T->head.left הינו המצביע לצומת של העץ.

(לא עושים שימוש ב- T->head.right, והוא ישאר עם ערך זבל.)

13. כעת, הפעולות שנגדיר עבור עץ החיפוש:

```
BOOL T_init(TREE* T);                    //אתחול העץ
NODE2* T_find(TREE* T, DATA x);          //מציאת צומת עם ערך מבוקש
NODE2* T_insert(TREE* T, DATA x);        //הוספת צומת חדשה לעץ
BOOL T_delete(TREE* T, DATA x);          //מחיקת צומת עם ערך מבוקש
BOOL T_free(TREE* T);                    //שחרור העץ
void preorderPC(NODE2* v);               //סיור preOrder בעץ והדפסה
void inorderPC(NODE2* v);                //סיור inOrder בעץ והדפסה
void postorderPC(NODE2* v);              //סיור postOrder בעץ והדפסה
```

14. חיפוש ערך בעץ - מבוצע רקורסיבית.

אם הערך שאנו מחפשים קטן מהערך שבצומת הנוכחית, נמשיך את החיפוש בתת העץ השמאלי, ולהפך.

סיבוכיות החיפוש במקרה הטוב ביותר, כאשר העץ הוא שלם (מאוזן ומלא), היא $O(\log n)$.

במקרה הגרוע ביותר, אם העץ אינו מלא ויש לו רק בנים ימניים למשל, נקבל למעשה רשימה מקושרת, ואז הסיבוכיות היא $O(n)$.

15. הכנסת צומת חדשה לעץ: מבוצע כמו החיפוש, ע"י הגעה למקום הנכון להוספת הצומת החדשה.

גם כאן נקבל אותה סיבוכיות כמו בחיפוש. (וגם $O(n)$ במקרה הגרוע שבו העץ הוא למעשה רשימה מקושרת.)

16. מחיקת צומת מהעץ: אסור להרוס את המיון של העץ. נבחין ב- 4 מקרים:

- הצומת אינה בנמצא - לא נעשה כלום.
- הצומת היא עלה - צריך רק לסלק אותה.
- לצומת יש בן יחיד - תן לסבא של אותו הבן להיות האבא שלו מעכשיו.
- אם לצומת יש שני בנים - נחפש את הצומת שמכילה את הערך שגדול ממנו והקרוב לו ביותר, ע"י הליכה לבן הימני ומשם כל הדרך שמאלה.

  או שנחפש את הערך שקטן ממנו והקרוב לו ביותר, ע"י הליכה לבן השמאלי ומשם כל הדרך ימינה.

  נעביר את הערך מהצומת שמצאנו, כך שהוא יכנס במקום הערך של הצומת שאותה מוציאים.

  כעת נסלק את הצומת שמצאנו לפי הסעיפים הקודמים.

גם במחיקת צומת מהעץ, סיבוכיות הפעולות היא כמו בהכנסת צומת לעץ, או בחיפוש של צומת.

17. <u>תרגיל כיתה 5</u>: הצג את מצב העץ הנ"ל לאחר מחיקת הצומת 100, ואח"כ מחיקת הצומת 152.

18. בעץ חיפוש בינארי יש הוכחה, **<u>שבממוצע</u>** כל הפעולות בעץ מתבצעות ב- $O(\log n)$.

19. צורת העץ (טובה - עץ שלם, גרועה - כרשימה מקושרת) נקבעת לפי סדר הכנסת האברים. למשל:

הכנסת 1, 2, 3 משמאל לימין תצור רשימה, והכנסת 2, 1, 3 תצור עץ שלם.

20. כעת נעבור על מודול העץ המוצג באתר הקורס, נראה את מימוש הפעולות המוגדרות על העץ, כמו גם דוגמא לשימוש בעץ.

21. תרגיל כיתה 6: מה הסיבוכיות של שאר הפעולות על העץ?

22. תרגיל כיתה 7/או בתרגול: כתוב פונקציה שתקבל את המצביע לעץ, ותחזיר את גובה העץ.

```
int heightOfTree(TREE* T);
```

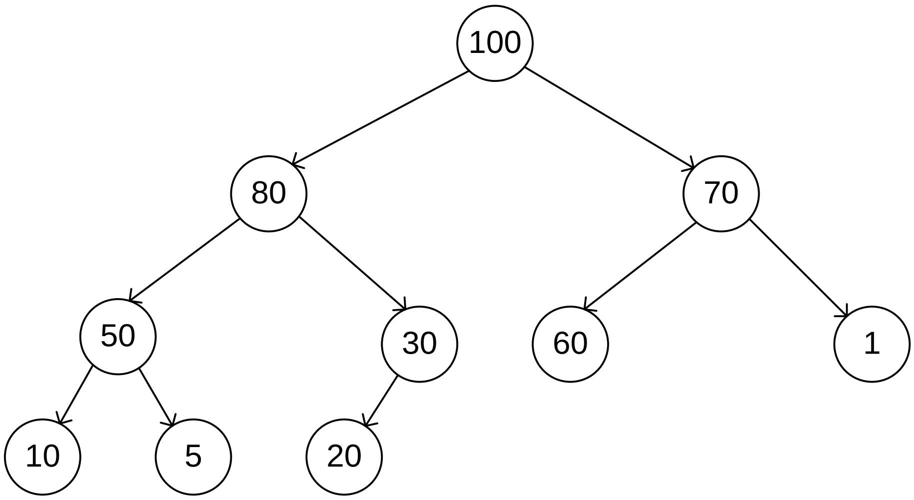
23. תרגיל כיתה 8/או בתרגול: כתוב פונקציה שתקבל את המצביע לעץ בינארי, ותחזיר את התשובה: האם העץ הוא עץ חיפוש, או לא. מותר לך להיעזר במבנה נתונים אחד כעזר.

אילוץ: אסור להשתמש במערך.

24. תרגול - נציג כעת בכיתה תרגילי כיתה נוספים.

מפגש מס' 10. נושא המפגש: ערימה.

1. עץ כמעט שלם, הוא עץ מלא ושלם בכל רמותיו, פרט אולי לאחרונה, שמלאה משמאל באופן חלקי.
2. ערימה (heap) היא עץ בינארי כמעט שלם (או שלם), שבו כל צומת גדולה משני בניו. (או לפחות שווה להם.) ערימה כזו נקראת ערימת מקסימום.

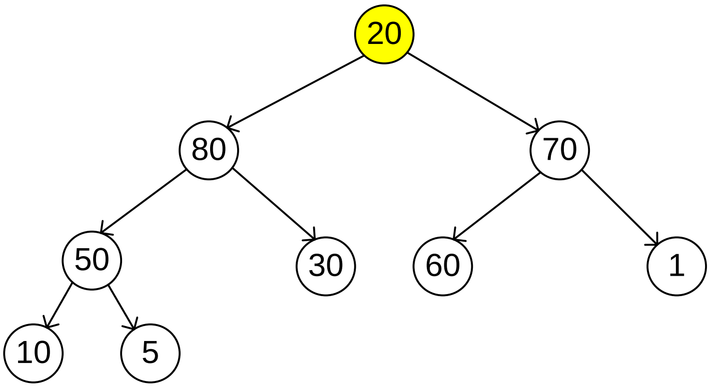


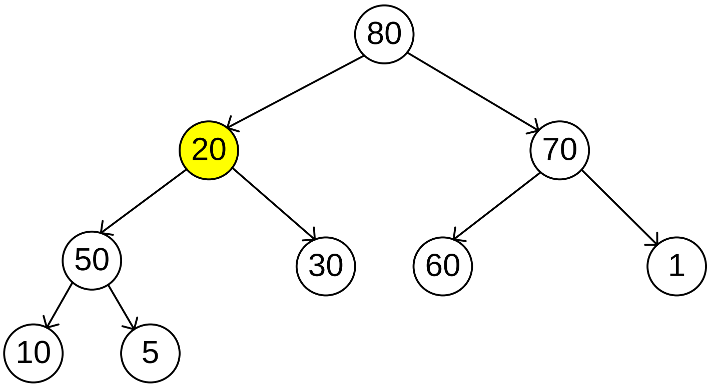
3. ערימה שבה כל צומת קטנה מהבנים שלה (או לפחות שווה להם) נקראת ערימת מינימום. בסיכום זה יוצג המקרה של ערימת מקסימום, והסעיפים מתייחסים לערימה כזו.
4. בראש הערימה יש תמיד את האיבר המכסימלי.
5. שימושים של ערימה: ערימה נקראת גם תור קדימויות (priority queue) שבו יש את הכלל: הגדול תמיד יוצא ראשון. כי תמיד ידוע שבראש הערימה יש את הגדול ביותר. לכן ניתן להשתמש בערימה לצורך מתן סדר עדיפות למטלות הדפסה, ניהול הקצאות זמן CPU לתהליכים, ועוד.
6. מימוש ערימה: במפגש שנושאו עצים בינאריים, הצגנו מימוש של עץ בינארי בעזרת מערך. במימוש שהוצג, שורש העץ נשמר באינדקס 0 של המערך. ערימה מקובל לממש באמצעות מערך, מכיוון שהיא עץ שלם או כמעט שלם.
במימוש שנציג עבור הערימה, השורש יאוחסן באינדקס 1 ולא באינדקס 0, אולם ניתן לממש כמובן בכל אחת משתי השיטות. ולכן, אם i זה האינדקס של האב, אז הבן השמאלי יהיה באינדקס $2*i$ (תמיד זוגי), והבן הימני יהיה באינדקס $2*i+1$ (תמיד אי זוגי). אם נתון בן אז האבא נמצא באינדקס $\left\lfloor \frac{i}{2} \right\rfloor$.
שים לב: יוצרים מספור של הצמתים מלמעלה למטה, ומשמאל לימין בכל רמה.
7. <u>תרגיל כיתה 1</u>: צייר את המערך המתקבל עבור הערימה הנ"ל. (השורש כאמור מאוחסן באינדקס 1 של המערך.)
8. <u>תרגיל כיתה 2</u>: במערך שציירת, בדוק מי האבא של 20, ומי האבא של 5.
9. הוצאת שורש הערימה: נשים במקומו את האיבר האחרון במערך, כי בערימה צריך להיות רצף, ולכן הכי נוח לקחת את האחרון ולהקטין את גודל הערימה ב- 1. אולם כעת, יתכן והשורש החדש מפר את תכונות הערימה. לכן נריץ את הפונקציה *sift_down* על השורש החדש. *sift_down* היא פונקציה שעובדת על ערימה, שבה יתכן ואיבר <u>אחד</u>, שמצוי באינדקס *i*, מפר את תכונות הערימה. *sift_down* בודקת אם הערך קטן מידי יחסית לבניו, ואם כן, היא מבעבעת אותו כלפי מטה, ע"י זה שהיא מחליפה אב עם הגדול מבין בניו כל עוד יש בנים והבעיה עדיין נמשכת. סיבוכיות

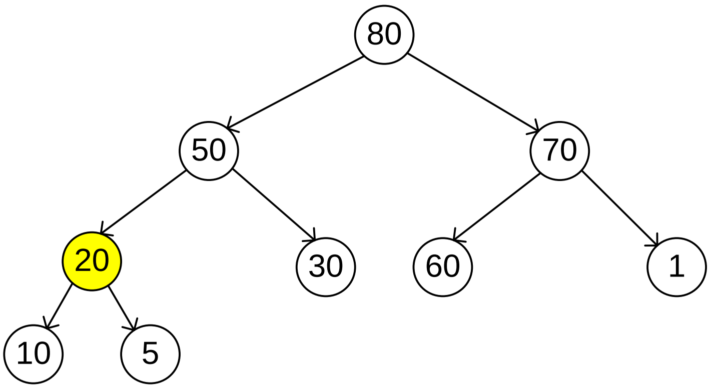
(השימוש מותר רק לתלמידי הקורס שלומדים איתי. אין לשכפל/להעתיק/להשתמש בחומר זה ללא אישור בכתב)

הריצה שלה היא $O(\log(n))$. לעיתים מקובל לקרוא לפונקציה זו גם בשם *Heapify*.

נסיר מהערימה הנ״ל את השורש, נחליף אותו באיבר האחרון, ולאחר מכן ניעזר ב- *sift_down* כדי לשמור את תכונות הערימה:







10. הכנסת איבר חדש לערימה: נכניס אותו כאיבר האחרון במערך. כעת יש לבדוק, האם איבר זה קטן מאביו. לצורך כך, נריץ את הפונקציה *sift_up* על האיבר החדש. *sift_up* היא פונקציה שבודקת אם הערך החדש יותר קטן מאביו, ואם כן מחליפה ביניהם. *sift_up* מבעבעת את האיבר החדש במעלה הערימה, עד שאינו גדול מאביו.
11. <u>תרגיל כיתה 3</u>: צייר את הערימה המתקבלת, אם מוסיפים לערימה הנ״ל את הערך 200.
12. יצירת ערימה ממערך נתון: כדי לצור ערימה ממערך נתון, מספיק לבצע *sift_down* על כל איברי הערימה שאינם עלים. שים לב: העלים נמצאים במחצית האחרונה של איברי המערך. כלומר, אם N היא כמות האיברים בערימה, יש לבצע *sift_down* רק על האיברים באינדקסים N/2 ומטה.

<u>תרגיל כיתה 4</u>: נתון המערך הבא משמאל לימין: 10, 8, 2, 4, 100, 33, 5, 1. צייר את הערימה שתיווצר ממערך זה.

13. הגדרת המבנה ערימה:

```
typedef struct
{
    DATA *keys;          //מצביע למערך דינאמי ובו איברי הערימה
    int  N, count;       //כמות מקסימאלית וכמות בפועל של איברים בערימה
}HEAP;
```

14. דוגמא ליצירת משתנים מטיפוס ערימה:

```
Heap h1, h2;
```

15. שים לב: אם H הוא מצביע לערימה, אזי H->keys[1] הוא שורש הערימה.

16. הפעולות שנגדיר עבור הערימה:

```
BOOL H_init(HEAP *H, int N);                              //אתחול ערימה
int H_insert(HEAP *H, DATA x);                            //הכנסת איבר חדש עם הערך x לערימה
DATA H_delMax(HEAP *H);                                   //הוצאת שורש הערימה
DATA H_findMax(HEAP *H);                                  //החזרת ערך השורש
BOOL H_makeHeap(HEAP *H, int N, DATA* values); //יצירת ערימה ממערך נתון
void H_free(HEAP *H);                                     //שחרור הערימה
void H_print(HEAP *H);                                    //הדפסת הערימה
```

17. כעת נעבור על מודול הערימה המוצג באתר הקורס, נראה את מימוש הפעולות המוגדרות על הערימה, כמו גם דוגמא לשימוש בערימה.

18. הסיבוכיות של בניית ערימה ממערך נתון (הפונקציה H_makeHeap), היא $O(n)$. וזאת בגלל שמבצעים *sift_down* רק על מחצית מהצמתים, ובנוסף, זמן הריצה של *sift_down* משתנה עם גובה הצומת, ורוב הצמתים אינם גבוהים.

19. <u>תרגיל כיתה 5</u>: מה הסיבוכיות של שאר הפעולות המוגדרות על הערימה?

20. מיון ערימה (heap sort): ניתן לבצע מיון מערך המבוסס על ערימה, לפי האלגוריתם הבא:

1. צור ערימה H ממערך נתון A, ע"י ביצוע MakeHeap.
2. עבור n מאינדקס האיבר האחרון במערך, ועד לאינדקס האיבר השני במערך, בצע:

   2.1 החלף בין האיבר הראשון (שהוא כעת הגדול ביותר), לבין האיבר באינדקס n. (האיבר הגדול ביותר מצטרף לשמורה.)

   2.2 הקטן את גודל הערימה ב- 1. (כלומר מתוך כל המערך, הערימה הרלבנטית עבור סעיף 2.3 תהיה כעת יותר קטנה.)

   2.3 בצע *sift_down* לאיבר הראשון במערך. (שים לב שהערימה הרלבנטית הוקטנה בסעיף 2.2)

21. <u>תרגיל כיתה 6</u>: מה הסיבוכיות של מיון ערימה?

22. <u>תרגול</u> - נציג כעת בכיתה תרגילי כיתה נוספים.

מפגש מס' 11. נושא המפגש: טבלאות ערבול.

1. מילון - אוסף רשומות שלכל רשומה יש מפתח ייחודי. במילון כל המפתחות שונים זה מזה, והם מספרים שלמים שאינם שליליים.
2. רוצים לממש את הפעולות הבאות על המילון: הכנסה, הוצאה, חיפוש, בסיבוכיות של $O(1)$.
3. טבלת ערבול (hash table): מבנה נתונים המבוסס על מערך, למימוש הפעולות על המילון ביעילות המרבית.
4. הפעולות שנגדיר עבור טבלת הערבול:

BOOL HASH_init(HASH *H, int m);	אתחול טבלת ערבול בגודל m//
BOOL HASH_insert(HASH *H, DATA k);	הכנסת מפתח חדש עם הערך k לטבלת הערבול//
BOOL HASH_ delete(HASH *H , DATA k);	מחיקת מפתח עם הערך k מטבלת הערבול//
int HASH _search(HASH *H , DATA k);	חיפוש מפתח עם הערך k בטבלת הערבול//
BOOL HASH_free(HEAP *H);	שחרור טבלת הערבול//
BOOL HASH_print(HEAP *H);	הדפסה של טבלת הערבול//

5. שיטת המיעון הישיר: אם אני רוצה לאחסן למשל רק מפתחות ללא שדות, אפשר לקחת מערך שכולו אפסים והטווח שלו הוא כטווח המפתחות. בהכנסת מפתח חדש נעלה את האינדקס שהוא כמו המפתח במערך לערך 1. ע"י גישה ישירה לאינדקס המתאים נוכל לדעת אם הוא 1 או 0 וכו'. למשל במערך הבא נכניס את המפתחות 2, 4, 5:

0	0	1	0	1	1	0
0	1	2	3	4	5	6

6. אם רוצים לשמור לכל מפתח גם שדות נוספים, ניתן לבצע למשל עם מערך מצביעים למבנים. למשל אם יש מבנה שהמפתח שלו הוא 2, המצביע מהתא עם האינדקס 2 יצביע על מבנה זה. אפשר גם לעבוד עם מערך של מבנים במקום מערך מצביעים.
7. הבעיה: אם גודל הטווח שלי הוא יותר גדול מהמערך שניתן להקצות, כלומר כמות המפתחות גדולה מידי יחסית למערך האפשרי, היכן נאחסן את המבנים שאינם בטווח של המערך? עדיין רוצים לממש את הפעולות ב- $O(1)$. מצב אחר שיתכן, הוא שמקצים את המערך, אולם יהיה בזבוז של זיכרון אם הטווח גדול ביחס לכמות המפתחות. (למשל שמירה של מספרי זהות עבור מספר קטן יחסית של אנשים.)
8. נשתמש בפונקציית *Hash*, כלומר בפונקציית ערבול. מטרתה ניתוב המפתחות הקיימים **<u>רק לאינדקסים שקיימים במערך</u>**. הדרישה היא שהפונקציה תפזר היטב ותהיה קלה לחישוב. קיימות שיטות שונות ליצירת פונקציית ערבול. אחת השיטות היא שיטת החילוק, ובה $h(k) = k \bmod m$. דוגמא לפונקציית *Hash* עבור מימוש מילון עם מערך בגודל 10 תאים: $h(k) = k \% 10$.
9. בעיית ההתנגשויות (Collision): יתכנו שני מפתחות או יותר שמנותבים לאותו המקום.

10. <u>פתרון עם רשימות מקושרות / שיטת השרשור (chain hashing) :</u> בכל תא במערך תהיה רשימה מקושרת. כלומר המערך יהיה מערך של רשימות מקושרות, או מערך מצביעים לרשימות מקושרות. נבדוק את סיבוכיות הפעולות: הכנסה: $O(1)$ - כי נכניס לראש הרשימה הנכונה. הוצאה וחיפוש: $O(n)$ - כי במקרה הגרוע כל האברים נכנסו לאותה הרשימה.
11. לכן משתמשים בערבול בגלל הזמן **<u>הממוצע</u>** לפעולות ההוצאה והחיפוש ולא בגלל הזמן המקסימאלי.
12. הנחת הפיזור האחיד הפשוט (Simple uniform hash): פונקצית ה- *Hash* מפזרת את המפתחות באופן אחיד. כלומר, ההסתברות שמפתח כלשהו ינותב לתא מסוים, שווה בין כל התאים.
13. נסמן את גודל הקלט, כלומר כמות המפתחות ב- n, ונסמן את גודל הטבלה ב- m.

<u>הגדרה</u>: **פקטור העומס α:** $\alpha = n / m$. כלומר, כמה מפתחות יש בממוצע לכל אינדקס בטבלה.

14. תחת הנחת הפיזור האחיד, אורך כל רשימה מקושרת יהיה α. ולכן זמן החיפוש בממוצע בפתרון עם רשימות מקושרות יהיה $O(1 + \alpha)$. ה- 1, הוא הזמן עבור החישוב של פונקצית ה- *Hash*. אנו מניחים ש- α נשאר קבוע כאשר m ו- n שואפים לאינסוף, ולכן הסיבוכיות היא במונחי α. מכיוון ש- α הוא מספר קבוע, המשמעות היא שניתן לממש את כל הפעולות על המילון בזמן ממוצע של $O(1)$!
15. <u>תרגיל כיתה 1</u>: הדגם את הכנסתם של המפתחות 1 ,28 ,29 ,15 ,11 ,33 ,25 ,20 ,10 (משמאל לימין) לטבלת *hash* שבה ההתנגשויות נפתרות ע"י רשימות מקושרות. הטבלה מכילה 9 תאים ופונקצית הערבול היא $h(k) = k \% 9$. (רשום גם את ערכו של המקדם α.)
16. הגדרת המבנה של טבלת ערבול עם רשימות מקושרות:

```
typedef struct
{
   LIST *keys;            //מצביע למערך דינאמי ובו הרשימות המקושרות
   int  m;                //גודלה של טבלת הערבול
}HASH;
```

17. דוגמא ליצירת משתנים מהטיפוס של טבלת הערבול:

```
HASH h1, h2;
```

18. <u>פתרון של מיעון פתוח (open addressing)</u>: בשיטה זו מקדם העומס α לא יעלה על 1 ($\alpha \le 1$), כלומר מאחסנים את כל האיברים בטבלה. בפועל מקובל לבחור $\alpha \approx 0.5$.
19. האלגוריתם להכנסת איבר לטבלה בשיטת המיעון הפתוח:

נחזיק קבוצה של m פונקציות ערבול: $h(k, 0), h(k, 1), \ldots, h(k, m-1)$. באופן כזה שכל הפונקציות יחד מכסות את כל התאים, כלומר אין מיקום שלא ניתן להגיע אליו בטבלה.

Hash-Insert(HT , k)

1. בצע $i \leftarrow 0$
2. כל עוד $i < m$, בצע:

2.1 $j \leftarrow h(k, i)$

2.2 אם התא $HT(j)$ אינו תפוס, הכנס את k לתא וסיים, אחרת בצע $i \leftarrow i + 1$.

20. האלגוריתם לחיפוש איבר בטבלה בשיטת המיעון הפתוח:

Hash-Search(HT , k)

1. בצע $i \leftarrow 0$

2. כל עוד $i < m$, בצע:

2.1 $j \leftarrow h(\mathrm{k}, i)$

2.2 אם בתא HT(j) יש את הערך k, החזר את j.

2.3 אם התא HT(j) פנוי, החזר שקר.

2.4 אחרת, בצע $i \leftarrow i + 1$.

21. בסעיפים הבאים נציג שיטות לפתור את בעיית ההתנגשויות בהכנסת אברים לטבלה בשיטה זו.

22. <u>שיטה ראשונה</u>: סריקה/בדיקה ליניארית (Linear probing): בשיטה זו: $h(\mathrm{k}, i) = (h(\mathrm{k}) + i) \bmod m$.
בהתחלה כל התאים הריקים במערך מסומנים כ- NoValue. אם המקום המיועד להכנסה לפי פונקצית ה- *Hash* תפוס, אז שים את הנתון אותו אתה רוצה להכניס לטבלה בתא הבא. אם גם הוא תפוס חפש הלאה עד שתמצא תא פנוי ראשון. אם הגעת לסוף הטבלה תמשיך את החיפוש מהתחלת הטבלה. את התא שבו הכנסת את הנתון סמן כעת כ- Value. (מימוש בשפת C: כל תא בטבלת ה- *Hash* יהיה מבנה ששדה אחד שלו מתאים לסימונים הנ״ל, ובשדה השני יהיה את הנתון המבוקש. כלומר טבלת ה- *Hash* תהיה מערך של מבנים.)

23. <u>תרגיל כיתה 2</u>: נתונה טבלה עם 10 תאים ($m = 10$), ופונקצית ה- *Hash* היא: $h(\mathrm{k}) = \mathrm{k} \% 10$.
מכניסים את המספרים הבאים לטבלה (משמאל לימין): 57, 12, 37, 19, 17, 62, 53. צייר את תוכן המערך המתקבל. (עובדים עם שיטת הבדיקה הליניארית.)

24. הוצאת אברים בשיטת הסריקה/בדיקה ליניארית: ההוצאה היא הוצאה לוגית מהמערך, ורוצים להוציא את האיבר המבוקש בכמה שפחות צעדים.
לכן, נסרוק את התאים החל מהמקום המיועד, וכל עוד לא הגענו לתא שמוגדר כ- NoValue אז נמשיך. אם מצאנו את האיבר המבוקש, נסמן את התא שלו עם הערך Deleted (ולא עם NoValue), כדי שאם נחפש בעתיד איברים אחרים שאוחסנו אחריו, כי לא היה מקום לאחסן אותם במקום המיועד להם בהתחלה, אז לא ניעצר בסימן NoValue. כי נראה שיש שם Deleted ואז נמשיך את החיפוש בהמשך. כאמור, החיפוש נעצר כשיש סימן NoValue בתא כלשהו, ואז אני יודע שאין מה לחפש אחרי זה כי אין שם איברים שאחסנו אותם בהמשך רק כי לא היה מקום. (כמובן שלא רוצים לסרוק את כל המערך כולו בחיפוש כל פעם, ולכן סורקים רק עד לתא הראשון שיש בו NoValue.) לדוגמא: אם במערך המתקבל בתרגיל הכיתה 2 היינו מוציאים את 37, ומסמנים את התא שלו ב- NoValue במקום ב- Deleted, אז אם עכשיו היינו מחפשים את 17, לא היינו מגיעים אליו. אם התא מסומן נכון כעת כ- Deleted, אז לא ניעצר שם.

25. <u>תרגיל כיתה 3</u>: בהמשך לתרגיל הכיתה 2, בטבלה שהתקבלה, מכניסים עוד שני ערכים נוספים (משמאל לימין): 42, 92. כעת מסירים מהטבלה את 53, 62, 12. צייר את תוכן המערך המתקבל, ותאר כיצד יתבצע החיפוש של 92.

26. יתרון שיטת הסריקה הליניארית: פשטות. חסרונות: אינה מהווה קירוב טוב להנחת הפיזור האחיד הפשוט. נוצרים גושים של תאים תפוסים. תופעה זו נקראת הצטברות ראשונית (Primary clustering). כמו כן אורך החיפוש תלוי גם במה שהוצאתי ולא רק במה שקיים. בהוצאת איברים עדיפה שיטת הרשימות המקושרות.

27. <u>שיטה שנייה</u>: ערבול נשנה (Rehashing): נעבוד עם m פונקציות הערבול הנ״ל. נפעיל אותן אחת אחרי השנייה עד שנצליח. (גם כאן נסמן את התאים עם Value וכו׳. השוני הוא בשיטת החיפוש אחר המקום.)

28. <u>שיטה שלישית</u>: ערבול כפול (Double Hashing): זו אחת השיטות הטובות ביותר כי הפיזור שהיא יוצרת קרוב להנחת הפיזור האחיד. השיטה כאן היא להשתמש בשתי פונקציות *Hash*.

למשל: $h1(k) = k \% m$, $h2 = 1 + k \% (m - 1)$.

אם התא שיצא לפי $h1(k)$ תפוס, נספור ממנו והלאה (בצורה מעגלית על המערך) $h2(k)$ מקומות כל פעם, עד שנמצא תא ריק. כלומר, פונקצית הערבול *h1* היא עבור הגעה לתא הראשון האפשרי, ופונקצית הערבול *h2* היא פונקצית צעידה החל מתא זה. כלומר, בשיטה זו: $h (k , i) = (h (k) + i * h2 (k)) \bmod m$.

29. יתכנו מצבים בהם לא מגיעים לכל התאים. למשל, אם הטבלה היא בגודל 10 תאים, ומטיילים מתא כלשהו בקפיצות של 2, הרי לא עוברים על מחצית התאים. כדי להימנע מכך, יש לבחור את *m*, כלומר כמות התאים של טבלת הערבול, כך שתהיה מספר ראשוני, ו- *h2* כך שתפיק תמיד מספר שלם חיובי קטן מ- *m*. אפשרות נוספת היא לבחור את *m*, כמות התאים בטבלה, כחזקה של 2, ו- *h2* שתפיק תמיד מספר אי-זוגי. (גם כאן נסמן את התאים עם Value וכו'. השוני הוא בשיטת החיפוש אחר המקום.)

30. <u>תרגיל כיתה 4</u>: הצג הכנסת איבר לטבלה בשיטת הערבול הכפול, עבור שני פתרונות אלו.

31. סיבוכיות הפעולות של ערבול עם מיעון פתוח: בהנחת הגיבוב האחיד, מספר הבדיקות הממוצע במהלך חיפוש כושל הוא $1 / (1 - \alpha)$. אם α הוא קבוע, אז חיפוש כושל מתבצע בזמן $O(1)$. חיפוש מוצלח מתבצע בממוצע בכמות בדיקות יותר קטנה, שהיא $1 / \alpha * \ln (1 / (1 - \alpha))$.