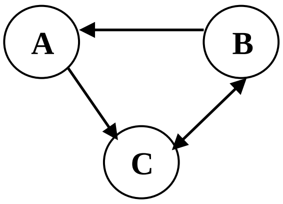
מפגש מס׳ 12. נושא המפגש: גרפים.

1. גרף הוא אוסף של קדקודים/צמתים וצלעות המחברות ביניהם. בגרף מכוון לכל צלע יש כיוון.



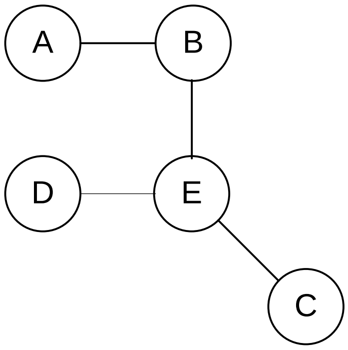
2. גרף הינו זוג סדור $G = (V, E)$, כך ש- V הינה קבוצה סופית של קדקודים, ו- E היא קבוצת הקשתות. (כל קשת מקשרת בין שני קדקודים.) למשל עבור הגרף הנ״ל:

$V = \{A, B, C\}$

$E = \{(A, C), (B, C), (C, B), (B, A)\}$

כלומר, אני יכול לשחזר את הגרף מנתונים אלו.

3. בגרף מכוון (directed graph, digraph), לכל קשת יש כיוון, כלומר היא יוצאת מקדקוד אחד ונכנסת לקדקוד אחר. (כיוון זה מסמן לי את כיוון התנועה האפשרית.)
4. בגרף לא מכוון (undirected graph), לכל קשת יש בעצם את שני הכיוונים, כלומר ניתן לנוע בשני הכיוונים, ולכן לא נסמן חיצים על הקשתות. ניתן לראות גרף לא מכוון כמקרה פרטי של גרף מכוון.
5. דרגה של קדקוד בגרף = מספר הקשתות שמחברות אותו, כלומר יוצאות ונכנסות ממנו.
6. מסלול בגרף: סדרת קשתות $\{e1, e2, \ldots..e_k\}$ המייצגות את סדר המסלול.
7. מעגל פשוט: מסלול בו הקדקוד הראשון הינו גם הקדקוד האחרון, ושאר הקדקודים שונים זה מזה.
8. עץ בגרף **לא מכוון**: זהו גרף לא מכוון שיש בו מסלול בין כל זוג קדקודים (נקרא גרף קשיר), וגם אין בו מעגליות כלשהי. (שים לב, זאת עדיין לא ההגדרה של עץ בינארי שלמדנו, אלא הגדרה יותר כללית.) למשל:



נשים לב למשל, שאם למשל הקדקוד A היה בודד ובנפרד משאר הגרף, (כלומר ללא קשת אליו או ממנו), זה לא היה עץ כי הגרף לא היה קשיר.

9. גרף תשתית של גרף מכוון, הוא הגרף הלא מכוון, שמקבלים אם מתעלמים מהכיוונים של הקשתות שהיו באותו גרף מקורי מכוון. (כלומר, כל קשת הופכת לדו-כיוונית כמו בגרף לא מכוון.)
10. עץ בגרף **מכוון**: גרף התשתית שלו הוא עץ, ואחד הצמתים יכול להיות שורש, כך שיש ממנו מסלול לכל צומת בעץ. (= עץ מושרש, כלומר שאחד הקדקודים מוגדר כשורש.)

11.עץ בינארי הוא עץ בגרף מכוון כך שלכל קדקוד יש לכל היותר שני בנים.

12.שימושים של גרף: ניתן לייצג דברים רבים, וגם לתת משקל לכל קשת בגרף: למשל מערכת של כבישים ויישובים, כאשר אורך כל כביש מיוצג כמשקל הקשת, ייצוג של רשתות תקשורת, רשתות תחבורה, ועוד.

13.מימוש של גרף ע"י מטריצת סמיכויות (שכנות) (**Adjacency Matrix**): נבנה מטריצה בגודל |V| X |V|, כאשר כל תא במטריצה $mat[i][j]$ יקבל 1 אם יש קשת בין קדקוד i ל- j, או 0 אם לא. שים לב: בגרף לא מכוון, המטריצה תהיה סימטרית. (כלומר סימטריות ביחס לאלכסון הראשי).

הערה: אם יש משקלות לקשתות, אזי במקום 1 נרשום את משקל הקשת.

14.תרגיל כיתה 1: בנה את מטריצת הסמיכויות עבור הגרף שבסעיף הראשון.

פתרון:

	A	**B**	**C**
A	0	0	1
B	1	0	1
C	0	1	0

15.תרגיל כיתה 2: בנה את מטריצת הסמיכויות עבור אותו הגרף הנ"ל, אולם הפעם עבור אותו הגרף אבל כגרף לא מכוון, כלומר עבור גרף התשתית שלו.

16.אפשרות ייצוג אחרת היא ע"י רשימות סמיכויות (שכנויות) (**Adjacency lists**): נבנה מערך דינמי (או רשימה) של הקדקודים שבגרף. נניח שנצייר את הרשימה הזו כרשימה 'אנכית'.

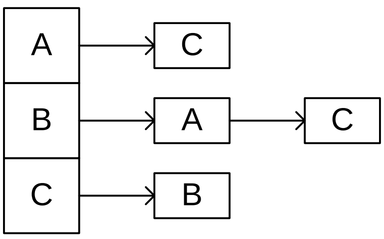
בכל מבנה של קדקוד יהיה לנו גם מצביע לרשימה מקושרות אחרת שבה יש לנו את הקדקודים שמחוברים ישירות לאותו קדקוד. (נצייר אותה 'אופקית'.)

כלומר, כל מצביע כזה יצביע על רשימה 'אופקית' שיוצאת ממנו, ומכילה את הצמתים המקושרים לקדקוד שהוא מייצג. (בפועל, כל רשימה 'אופקית' כזו יכולה להיות רשימה של הקשתות שיוצאות מאותו קדקוד, כאשר כל צלע מחזיקה מידע על הקדקוד השני שאליו היא מחוברת, וגם הצבעה לצלע הבאה בהמשך אותה רשימה 'אופקית'.)

(אין משמעות לסדר ברשימות השונות. בנוסף, אם יש משקלות לקשתות, נוסיף את המשקל כשדה במבנה המתאים לאותה קשת, ראה למשל מבנה הנתונים שבהמשך בו יש לנו מבנה של קשת.)

17.תרגיל כיתה 3: בנה ייצוג עם רשימות סמיכויות עבור הגרף שבסעיף הראשון.

פתרון:



18.הבדלים עיקריים בין ייצוג עם רשימות וייצוג עם מטריצה: במטריצה, בדיקת קיום קשת עם $O(1)$ לעומת $O(n)$ בייצוג עם רשימה. לעומת זאת, אם הגרף הוא דליל, כלומר שאין בו מספר רב של קשתות, אזי השימוש במטריצה מהווה בזבוז זיכרון. בנוסף, יותר מסובך לבנות אלגוריתמים (כגון חיפוש מעגל ועוד) במימוש עם מטריצה.

19.מימוש גרף עם רשימות:

הגדרות המבנים של קדקוד, קשת וגרף:

```
typedef int DATA;
typedef struct vertex
{
  DATA info;                //מידע
  struct vertex* next;      //מצביע לקדקוד הבא
  struct edge* headEdge;    //מצביע לצומת הראשונה של רשימת הקשתות
}vertex;
typedef struct edge         //קשת
{
  vertex* succesor;         //הצומת השנייה שאליה מחוברת הקשת
  struct edge* nextEdge;    //מצביע לקשת הבאה ברשימה
}edge;
typedef struct              //גרף
{
  vertex* head;             //מצביע לקדקוד הראשון
}Graph;
```

20.דוגמא ליצירת משתנים מטיפוס הגרף:

```
Graph g1, g2;
```

21. פעולות לדוגמא שניתן להגדיר עבור הגרף:

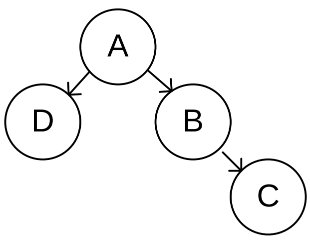
```
void init(vertex** head);                        //אתחול
vertex *addVertex(vertex** head, DATA x);        //הוספת קדקוד חדש
void addEdge(vertex** first, vertex* last);      //הוספת קשת חדשה בין שני קדקודים
const vertex *find(DATA x, const vertex* head); //מציאת קדקוד
void clearGraph(vertex** head);                  //שחרור כל ההקצאות הדינאמיות בגרף
```

22. סיבוכיות: הפעולות הן פעולות על רשימות מקושרות:

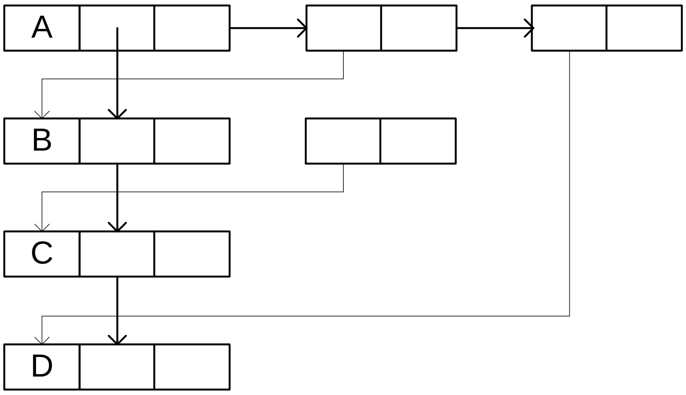
ב- *find*, יש ריצה רק על רשימת הקדקודים. את ההוספה של קשת חדשה, ניתן גם לבצע כך שנוסיף אותה בתחילת רשימת הקשתות כל פעם, כלומר $O(1)$.

23. שים לב: מודול הגרף אינו מוצג באתר הקורס.

24. <u>תרגיל כיתה 4</u>: בנה את הייצוג המתקבל עם מימוש זה, עבור הגרף הבא:



פתרון:



הסבר: מ- A יש שתי קשתות, אחת ל- B ואחת ל- D, ולכן מ- A יש לנו רשימה אופקית ובה שתי קשתות: המבנה הראשון מייצג את הקשת הראשונה, ע"י זה שיש בו מצביע ל- B, והמבנה השני מייצג את הקשת השנייה, ע"י זה שיש בו מצביע ל- D. (ניתן להניח למשל שאת הקשת אל D צרפנו ראשונה, ולכן היא בסוף הרשימה.)

באופן דומה, שאר הרשימות האופקיות. בנוסף, הרשימה האנכית היא רשימת הקדקודים עצמם.

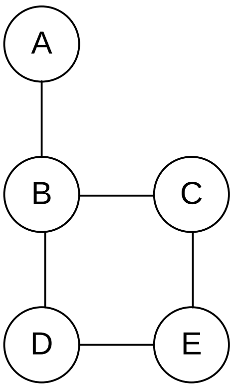
מפגש מס' 13. נושא המפגש: גרפים - BFS, DFS.

1. חיפוש לרוחב *BFS* (**Breadth-first search**)**:** רוצים לעבור על כל הצמתים של הגרף, אולם באופן הבא:
נבחר צומת מקור s. רוצים לסרוק את הגרף, ככה שקודם נעבור את כל הצמתים שרחוקים רק מרחק של צלע אחת מ- s, אח"כ את כל הצמתים שרחוקים מרחק של שתי צלעות מ- s וכך הלאה.
האלגוריתם משמש כאב טיפוס של אלגוריתמים אחרים, כגון מציאת מסלולים קצרים ביותר ממקור יחיד.
ניתן להפעיל אותו על גרפים מכוונים ולא מכוונים, כאשר לכל הקשתות אותו משקל.
2. נראה עכשיו אלגוריתם BFS פשוט:

BFS(G, s)

1. הכנס את s לתור Q.
2. כל עוד התור אינו ריק בצע:
2.1. ראש התור(Q) ← u
2.2. לכל צומת שכן של u, אם לא ביקרנו בו, אזי סמן שביקרנו בו והכנס אותו לתור Q.
2.3. הוצא מהתור(Q) (את u) והדפס אותו.

3. <u>תרגיל כיתה 1</u>: מה פלט האלגוריתם עבור הגרף הלא מכוון הבא (A הוא המקור):



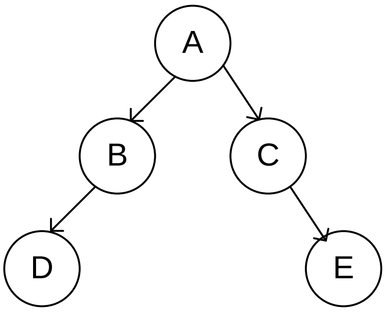
פתרון: משמאל לימין: A, B, C, D, E.
שים לב, ש- B מרוחקת מ- A קשת אחת, C ו- D שתי קשתות, ואז E המרוחקת שלוש קשתות.

4. אלגוריתם BFS יותר מפורט: שומר לנו גם את המסלול הכי קצר מהמקור לכל קדקוד כמו גם את המרחק הזה, וכן פירוט יותר רחב של מצב הסריקה בכל רגע.
האלגוריתם מחזיק את המשתנים הבאים:
Color[v] - צבע של קדקוד. לבן: עוד לא ביקרנו בו. אפור: ביקרתי בו אבל לא בשכנים שלו.
שחור: ביקרתי גם בו וגם בשכנים שלו.
d[v] - מרחק הכי קצר של קשתות מ- קדקוד המקור אל v.
Π[v] - שומר את האבא של v במסלול. (כך נדע לבסוף מה המסלול הכי קצר בין השורש לצומת כלשהי.)

BFS(G,s)

```
1.  For each vertex u in V[G] - {s} do          // לכל קדקוד בגרף מלבד המקור
        1.1   Color [u] ← white                 // תן לו צבע לבן כי לא ביקרת בו
        1.2   d[u] ←∞                           // מרחק אינסופי מהמקור
        1.3   Π[u] ← NIL                        // וכרגע לא ידוע מי אבא שלו
2.  Color[s] ←gray                              // למקור תן צבע אפור - לא ביקרת בשכנים שלו
3.  d[s] ←0                                     // המרחק שלו מהמקור הוא אפס
4.  Π[s] ← NIL                                  //והוא אין לו אבא במסלול
5.  Enqueue(Q, s)    //Q ←{s}                   //נתחיל בכך שנכניס אותו לתור
6.  While Q is not empty do
    6.1  u ← head[Q]                            //בפעם הראשונה u זה קדקוד המקור
        6.1.1 for each v in adj[u] do if color[v] = white     //לכל אחד משכנים של u שלא ביקרנו
            6.1.1.1   color[v] ← gray           // ביקרנו בו עכשיו אבל לא בשכנים שלו
            6.1.1.2   d[v] ← d[u] +1            //כל שכן כזה המרחק שלו הוא עוד קשת אחת מ- u
            6.1.1.3   Π[v] ← u                  //ואבא שלו הוא u
            6.1.1.4   Enqueue(Q, v)             // עכשיו תכניס את השכן הזה אל התור
    6.2  Dequeue(Q)    //אחרי שהכנסתי את כל השכנים של u לתור, אני מוציא את u מהתור ומדפיס
    6.3  color[u] ←black                        // וסיימתי איתו
```

5. זמן הריצה של BFS הוא $O(|V| + |E|)$. כי במקרה הגרוע נסרוק את כל הצמתים וכל הקשתות.
6. <u>תרגיל כיתה 2 :</u> עבור הגרף הבא : תאר באופן כללי את הריצה של BFS. (המקור הוא A.)



פתרון:

נכניס את A לתור, ו- u יקבל את A.

לכל אחד מהשכנים של A, נסמן שהמרחק שלו הוא 1, שהאבא שלהם הוא A, ונכניס אותם לתור, כלומר התור יראה כעת ככה משמאל לימין: A, B, C.

כעת, נוציא את ראש התור, כלומר את **A**, נדפיס וסיימנו.

כעת, חוזרים לשורה 6: התור עדיין לא ריק, והוא נראה ככה משמאל לימין: B, C.

לכן, u מקבל עכשיו את B שהוא ראש התור.

כל אחד מהשכנים של B צריך לסמן כעת שהמרחק שלו הוא 2, כי ל- B כבר היה מרחק של 1 מ- מקודם.

בנוסף, גם צריך לעדכן מי האבא, צבע אפור, ולהכניס אותו לתור.

אבל השכן היחיד של B הוא D, ולכן רק אותו נכניס לתור.

התור עכשיו נראה כך, משמאל לימין: B, C, D.

עכשיו סיימתי עם **B**, נדפיס אותו ונוציא מהתור. לכן התור עכשיו הוא C, D.

נבדוק כעת מי הם השכנים של C: יש רק את E. לכן נבצע הפעולות הנ"ל עם E, ונכניס אותו לתור.

מצב התור עכשיו הוא: C, D, E. סיימתי עם **C**, נדפיס אותו ונוציא מהתור. התור עכשיו: D, E.

לשניהם אין שכנים, ולכן נוציא ונדפיס את **D** ואז את **E**.

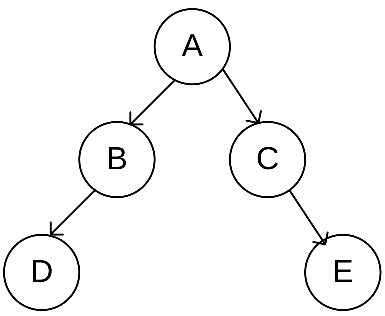
שים לב, שסדר הסריקה שמקבלים הוא לפי רמות!

7. חיפוש לעומק *DFS* (**Depth-first search**): שימושי עבור למשל חיפוש יציאה ממבוך, וכבסיס לאלגוריתמים שונים, כגון חיפוש מעגליות בגרף, ועוד.
8. האלגוריתם סורק את הצמתים החל מצומת מקור כלשהי, ומתקדם לעומק. לאחר מכן, הוא יחזור ויחפש צמתים שעוד לא נסרקו בכדי לחדש את הסריקה לעומק מהם.
   ניתן להפעיל אותו על גרפים מכוונים ולא מכוונים. (משקל קשתות אם יש לא משנה כאן.)
9. נראה עכשיו אלגוריתם DFS פשוט:

DFS(G, s)

1. דחוף את s למחסנית ST. (וסמן שביקרנו בו.)
2. כל עוד אינה ריקה (ST) בצע:
   2.1. שלוף (ST) ← u והדפס.
   2.2. לכל אחד מהשכנים של u, אם לא ביקרנו בו, אזי סמן שביקרנו ודחוף אותו למחסנית ST.

10. <u>תרגיל כיתה 3:</u> מה פלט האלגוריתם עבור הגרף הלא מכוון הבא (A הוא המקור):



11.אלגוריתם DFS יותר מפורט: שומר לנו את מועד הגילוי של קדקוד u במשתנה d[u] (d = discover),
ואת מועד סיום הטיפול בו במשתנה f[u] (f = finish). לכל הקדקודים מתקיים תמיד ש- d[u] < f[u].
חותמות זמן אלו מספקות מידע על מבנה הגרף: למשל, האם יש מעגליות, האם קדקוד מסוים הוא צאצא כלשהו של קדקוד אחר, ועוד.

שים לב: time הוא משתנה **גלובלי**.

<u>DFS(G)</u>

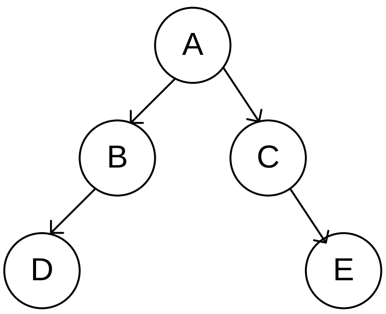
```
1. For each vertex u in V[G]                // עבור כל הקדקודים שבגרף
     1.1  do color[u] ← white               // צבע אותם בצבע לבן
     1.2  Π[v]←NIL                          // לאף אחד מהם אין אבא  מוגדר במסלול
2. time ←0
3. For each vertex u in V[G]                // עבור כל קדקוד שבגרף אם הצבע שלו לבן נבצע אלגוריתם עזר
     3.1 do if color[u] = white then DFS-visit(u)
```

<u>DFS-visit(u)</u>

```
1. Color[u] ←gray                           // ביקרנו בו אבל עדיין לא בשכנים שלו.  אפשר גם להדפיס כאן
2. d[u] ←time ←time+1                       // מעלה לו את חותמת זמן הגילוי שלו
3. For each v in adj[u]                     // לכל קדקוד שהוא שכן שלו
     3.1.  if color[v] = white              // אם עדיין לא ביקרנו בו
          3.1.1    Π[v]←u                   // נסמן מי אבא שלו במסלול
          3.1.2    DFS-visit(v)             // ונקרא רקורסיבית ל- DFS-visit
4. Color[u] ← black                         // סיימנו איתו ועם כל השכנים שלו אז נסמן אותו בשחור
5. f[u]←time←time+1                         // ונציב לו חותמת הזמן של סיום טיפול
```

12.זמן הריצה של DFS הוא O(|V| + |E|).

13.<u>תרגיל כיתה 4:</u> עבור הגרף הבא: תאר באופן כללי את הריצה של DFS. (המקור הוא A.)



פתרון:

נתחיל בקדקוד A ונצבע אותו באפור. חותמת זמן הגילוי שלו היא 1. השכנים שלו הם B ו- C.

נבצע עליהם את DFS-visit. נתחיל קודם עם B שיקבל חותמת זמן גילוי של 2. ואז נעשה DFS-visit לשכן שלו שהוא D, שיקבל חותמת זמן גילוי של 3, וחותמת זמן סיום של 4.

עכשיו יש חותמת זמן סיום עבור B, שיקבל 5.

עכשיו נעבור לשכן C שעדיין ממתין. זמן הגילוי שלו יהיה 6, השכן שלו E יקבל זמן גילוי של 7 וסיום של 8. C יקבל כעת זמן סיום של 9, ולבסוף נשאר לנו את A עם זמן סיום של 10.