סיכום חומר לקראת מבחן – מבנה נתונים

עדכנו אותי על טעויות או אי-דיוקים! תודה, אפרת.

נושאים חסרים: פתרון נוסחאות נסיגה, ניתוח פחת, וחלק קטן מהאלגוריתמים שנלמדו.

בצהוב: עידכונים מה-23/6

1. טבלת מיונים:

שם המיון	הסבר המיון	In place?	מיון יציב?	זמן ריצה + הסבר	הערות חשובות ווריאציות של המיון
מיון הכנסה	מיון השוואה פשוט. לולאה חיצונית עוברת על כל תא במערך מתחילתו עד סופו. בלולאה פנימית, כל תא נכנס במקומו בין כל האיברים **עד אליו** ע"י השוואות והחלפות.	✓	✓	O(n^2) במקרה הגרוע ובמקרה הממוצע.	מיון יעיל במקרה של מערכים קצרים או במקרה שרוב המערך כבר ממוין.
מיון מיזוג	מיון השוואות. מיון רקורסיבי. בכל שלב מפצלים את המערך לשניים, ממשיכים לפצל עד קבלת תאים בודדים, ובונים את המערך חזרה כלפי מעלה כאשר ממזגים בכל שלב כל שני חלקים עד קבלת המערך בשלמותו. פעולת המיזוג מתבצעת ע"י השוואת שני האיברים הראשונים בשני המערכים שרוצים למזג והוצאת הקטן ביותר בכל פעם.	X	✓	O(nlogn) – O(n) פעולת המיזוג בכל שלב. logn – מספר הפעמים שמיזוג זה מתבצע.	לאלגוריתם רקורסיבי ניתן לחשב את זמן הריצה באמצעות זמן נסיגה. במקרה זה: $T(n) = 2*T(n/2)+O(n)$ $T(1)=O(1)$
עץ חיפוש בינארי	(פירוט מטה)				
Heap-sort	מיון המתבסס על ערימת מקסימום/מינימום. בונים ערימה מהמערך הנתון (ללא צריכת זיכרון נוסף). אח"כ עוברים בלולאה מ-n (מס' האיברים) עד 1: מושכים כל פעם את איבר המקסימום, מציבים במקום האחרון הפנוי במערך ומתקנים את הערימה (פחות האיבר שכבר במקומו).	✓	X	O(nlogn) במקרה הגרוע. הסבר: בניית ערימת-מקסימום – O(n), תיקון הערימה אחרי כל הוצאת מקסימום – O(logn).	
Quick-Sort מיון מהיר	מיון השוואה. מממש אסטרטגיה של הפרד ומשול. בהינתן סדרת מפתחות: 1. בוחרים איבר-ציר, pivot, לרוב באופן אקראי. 2. מסדרים את המערך כך שכל הקטנים מהציר יופיעו לפניו וכל הגדולים יופיעו אחריו. (איך עושים זאת? פונקציית עזר – partition: עוברים על המערך מתחילתו. עבור איבר קטן מהציר ממשיכים הלאה. עבור איבר גדול מהציר מחפשים את האיבר הבא הקטן מהציר ומחליפים ביניהם. שומרים "חוצצים" במערך המסמלים: כל האיברים שכבר התגלו כקטנים מהציר, כל הגדולים מהציר, ואלו שטרם נבדקו. את הציר עצמו נשים בינתיים בסוף המערך הנבדק). זמן ריצה: O(n). 3. קוראים באופן רקורסיבי לגבי כל צד של הציר. 4. תנאי עצירה: כאשר המערך עליו השיטה מופעלת מכיל איבר אחד בלבד.	✓	X	O(n^2) במקרה הגרוע ו O(nlogn) במקרה הממוצע. זמן הריצה תלוי בבחירת הציר.	לבחירת ה-pivot חשיבות מכרעת בקביעת זמן הריצה. אם הציר יהיה תמיד החציון, נקבל זמן ריצה אידיאלי, אם הציר יבחר להיות האיבר הקטן או הגדול ביותר, נקבל O(n^2). שיטות לבחירת הציר: בחירת 3 איברים באקראיות ולקיחת האמצעי מתוכם.

שם המיון	הסבר המיון	In place?	מיון יציב?	זמן ריצה + הסבר	הערות חשובות ווריאציות של המיון
מיון מנייה – counting sort	כל איברי הקלט הם מספרים שלמים בתחום k-1. רעיון : מחשבים עבור כל איבר כמה איברים קטנים ממנו או שווים לו. אם יש y איברים קטנים מאיבר x, הרי שx יהיה במקום ה-y במערך הממוין. האלגוריתם משתמש ב-2 מערכי עזר : מערך C באורך k לשמירת המנייה לכל איבר, ומערך פלט B באורך מערך הקלט A. פירוט האלגוריתם : מאתחלים את מערך C באפסים. עוברים פעם ראשונה על מערך C. אם i=A[i], כלומר מצאנו איבר במערך הקלט השווה לאינדקס הנוכחי, נגדיל באחד את ערך התא C[i]. עוברים פעם שנייה על C, הפעם סוכמים לכל איבר את עצמו וכל אלה שלפניו (כלומר המספר בתא C[i] מייצג עתה את מסי האיברים הקטנים/שווים ל-i). כעת עוברים על מערך הקלט מהסוף להתחלה, כל איבר עליו עוברים מועבר למקום המתאים ב-B לפי המספר השמור ב-C, וערך C מתעדכן ע״י הפחתת 1.	X	✓	זמן הריצה הוא O(n+k) כאשר אם k=O(n) אז זמן הריצה הוא לינארי – O(n).	
מיון בסיס – radix sort	מיון לפי ספרות. מסתמך על כך שמספר הספרות בייצוג המספר חסום ע״י d. האלגוריתם מבצע מיון של המערך לפי ספרת האחדות של המספר (כשהייצוג הוא בבסיס 10), לאחר מכן מיון לפי ספרת העשרות, וכן הלאה – עד מיון הספרה ה-d. כל אחד מהמיונים יתבצע ע״י מיון יציב כלשהו. לצורך העניין – מיון מנייה.	תלוי במיון שבחרנו להשתמש בו.	✓	O(d(n+k)) – מכיוון שמבוצע מיון מנייה, כפול מסי הספרות d. אם d קבוע ו-k=O(n) נקבל זמן ריצה לינארי O(n).	יש לנו גמישות בבחירת הבסיס. משפט : נתונים n מספרים בעלי b ביטים, ונתון מסי שלם חיובי r הקטן מ-b. מיון בסיס ממיין את המספרים הנ״ל בזמן : $O(\frac{b}{r}(n+2^r))$. כלומר, עבור n וb נתונים נרצה לבחור r כך שזמן הריצה יצא מינימלי.
מיון דלי – bucket sort	הנחה : המספרים הם ממשיים בתחום $[0,1)$ ומתפלגים באופן יחיד בתחום זה. (ניתן להשתמש גם על כל טווח מספרים סופי אחר). נשתמש במערך עזר B של רשימות מקושרות שייצג את ה״דליים״. כל דלי מייצג טווח בגודל 1/n. עוברים על מערך הקלט A (באורך n). כל איבר A[i] ״מכניסים״ לדלי המתאים : $B[n\cdot A[i]]$ (לוקחים את הערך התחתון של nA[i]). לאחר מכן עוברים על כל רשימה במערך B וממיינים אותה ע״י מיון השוואות רגיל כלשהו. כעת מעבר על מערך B ייתן את המערך כולו ממוין.	X	✓	O(n), <u>בהנחת התפלגות אחידה.</u> O(n^2) במקרה הגרוע.	ללא הנחת ההתפלגות האחידה, זמן הריצה עולה אך האלגוריתם עדיין נשאר נכון.

<u>הערה:</u> מיוני השוואות חסומים ע״י O(nlogn). המיונים בזמן ריצה לינארי אינם מיוני השוואות.

2. גרפים

2 דרכים לייצוג גרפים:

א. מטריצת שכנויות. • זוהי מטריצה בגודל |V|X|V|. • תא $a_{i,j}$ יכיל 1 אם $(i,j)\in E$ ו-0 אחרת. • מקום בזיכרון: O(V^2). • זמן ריצה למציאת כל שכני קודקוד u: O(V). • זמן ריצה לבדיקת קיום הצלע (u,v): O(1).	ב. רשימת שכנויות: • זהו מערך שגודלו כמספר הקודקודים בגרף |V| • מכל איבר במערך יוצאת רשימה מקושרת של שכני קודקוד זה. הרשימה של קודקוד v למשל תכיל את כל קודקודי u המקיימים $(v,u)\in E$. • עובד גם על גרפים מכוונים וגם על לא-מכוונים. • מקום בזיכרון: O(V+E). • זמן ריצה למציאת כל שכני קודקוד u: O(deg(u)). • זמן ריצה לבדיקת קיום הצלע (u,v): O(deg(u)).	

G=(V,E) |V|=n, |E|=m	רשימות סמיכות	מטריצת סמיכויות
מקום	O(n+m)	O(n²)
בדיקה האם (v,u)∈E	O(degree(v))	O(1)
מעבר על כל הקשתות הסמוכות לקודקוד v	O(degree(v))	O(n)
הכנסת קשת חדשה (u, v)	O(1)	O(1)
מחיקת קשת (u,v)	O(degree(v))	O(1)

הערה כללית לגבי שאלות על גרפים: ישנן שתי גישות לפתרון בעיות גרפים – האחת, התייחסות לאלגוריתמים מטה כאל "קופסאות שחורות" וביצוע מניפולציות על הקלט כדי שיתאים לאלגוריתם. גישה שנייה – התאמת האלגוריתם לבעיה הספציפית ע"י ביצוע שינויים באלגוריתם עצמו.

MST = עץ פורש מינימלי. יכול להיות יותר מאחד כזה בגרף. יהיו בו \|V\|-1 צלעות. עבור גרף ממושקל – מינימאליות העץ תתבטא בסכום משקלי צלעות העץ.

שם האלגוריתם	**הסבר**	**זמן ריצה + הסבר**
BFS	לא עובד על גרפים ממושקלים. עובד גם על מכוונים וגם על לא מכוונים. בהינתן <u>גרף</u> G=(V,E) ו<u>קודקוד מקור s</u> האלגוריתם מגלה את כל הקודקודים אליהם ניתן להגיע מ s ומחשב את המסלול הקצר ביותר מבחינת מספר קשתות לכל קודקוד שניתן להגיע אליו. בניסוח אחר, האלגוריתם בונה "עץ רוחב" ששורשו s. הוא עושה זאת ע"י עידכון שני שדות בכל קודקוד בגרף: Pred[v] - הקודקוד הקודם ל-v בעץ הרוחב (=מי "גילה" אותו). מאותחל בnull. d[v] – המרחק המינימלי מקודקוד המקור ל-v. מאותחל ב- ∞. במהלך האלגוריתם נשתמש בשדה נוסף: color[v] – לבן (כך מאותחל) = קודקוד שטרם "גילינו". אפור = קודקוד שגילינו וממתין בתור, כלומר טרם גילינו את שכניו, שחור = היה ויצא מהתור, גמרנו "לטפל" בו. האלגוריתם משתמש בתור. בכל שלב מוציאים את הקודקוד הראשון בתור, x, ועוברים על שכניו הצבועים בלבן. לכל שכן לבן מעדכנים את צבע לאפור, מכניסים אותו לתור, מעדכנים את ה pred שלו להיות x ואת מרחק d מקודקוד המקור להיות d[x]+1. מתחילים מלהכניס את קודקוד המקור לתור Q. בגמר האלגוריתם, הדפסת המסלול מקודקוד מסוים v ל-s תתבצע ע"י שיטה רקורסיבית שקוראת לשיטה על pred[v] ואז מדפיסה את v.	O(V+E). הסבר: כל קודקוד נכנס לתור פעם אחת ויוצא ממנו פעם אחת. כש"מטפלים" בכל קודקוד עוברים על כל שכניו, כלומר סה"כ עוברים על בדיוק |E| הצלעות.

שם האלגוריתם	הסבר	זמן ריצה + הסבר
DFS	סריקה לעומק: יוצאים מנקודה מסוימת ומנסים להגיע הכי "עמוק" שניתן. האלגוריתם מתחיל את החיפוש מצומת שרירותי בגרף ומתקדם לאורך הגרף (עפ"י סדר השכנים ברשימת השכנויות) עד שנתקע. לאחר מכן חוזר על עקבותיו (backtracking) עד שהוא יכול לבחור דרך אלטרנטיבית להתקדמות. כאשר עבר על כל הדרכים האפשריות ברכיב קשירות זה, מגריל קודקוד חדש שטרם עברו בו. אלגוריתם זה, בניגוד לקודם, עובר על כל קודקודי G. האלגוריתם מגלה מעגלים, מצייר את הגרף כיער עצים, מאפשר מיון טופולוגי (בהמשך). שדות שמעודכנים במהלך ריצת האלגוריתם בכל אחד מהקודקודים: d[v] – "חותמת זמן" – הגעה לקודקוד. f[v] – "חותמת זמן" – סיום טיפול בקודקוד. (מתקיים: $1 \le d[v] < f[v] \le 2\|V\|$) $\pi[v]$ - שומר את הקודקוד ממנו הגענו ל-v. ה-predecessor. לכל קודקוד יש גם צבע: לבן = קודקוד שטרם גילינו, אפור = קודקוד שגילינו אבל לא סיימנו לבדוק את כל הדרכים היוצאות ממנו, שחור = גילינו וסיימנו לעבור על שכניו. מיון צלעות: Tree edge – קשת המרכיבה את אחד העצים ביער, כלומר קשת שבה "גילינו" קודקוד חדש. קשת אחורה – back edge – קשת שמגלה מעגל, כלומר קשת מצאצא לאחד מאבותיו הקדמונים. צלעות נוספות, מופיעות רק בגרפים מכוונים: קשת קדימה – forward edge - קשת מאב קדמון לצאצא שכבר גילינו קודם לכן. קשת חוצה – cross edge – כל צלע אחרת, לרוב מחברת בין שני עצים ביער. הבדל בין קשת חוצה לקשת קדימה: בקשת חוצה, לקודקוד אליו הגענו תהיה חותמת זמן סגירה קטנה מחותמת זמן הפתיחה של הקודקוד ממנו יצאנו. משפט: קשת אחורה מורה על מעגל והפוך.	O(V+E). הסבר: הלולאה החיצונית ביותר עוברת על כל קודקוד, למקרה שכל קודקוד נמצא ברכיב קשירות אחר. מכל קודקוד סורקים את כל המסלולים האפשריים, סה"כ |2|E. החישוב הוא בהנחה שייצוג הגרף נעשה ע"י רשימת שכנויות.
מיון טופולוגי	זהו מיון המבוסס על DFS כאשר הגרף שניתן כקלט הוא גרף מכוון ללא מעגלים – DAG. מיון זה יוצר סידור ליניארי של הקודקודים כך שאם גרף מכיל קשת (u,v) אזי u נמצא בסידור לפני v. לכל גרף יש יותר מסדר טופולוגי יחיד, הדבר תלוי בקודקוד ממנו התחלנו ב- DFS. הסבר האלגוריתם: מפעילים את DFS, אך בכל השחרה של קודקוד, מצרפים קודקוד זה בראש רשימה מקושרת נפרדת. הרשימה שתתקבל בסוף היא רשימת הקודקודים הממוינת. או: לאחר הפעלת DFS מסדרים את כל הקודקודים בשורה לפי השדה f[v] בסדר יורד.	כמו DFS
GenericMST	בהינתן גרף, האלגוריתם הבסיסי למציאת MST יבנה סדרה ריקה A של צלעות ה-MST ויוסיף בכל שלב צלע לסדרה אס"ם צלע זו היא "צלע בטוחה". "צלע בטוחה" היא כזו שלא סוגרת מעגל עם שאר הצלעות בסדרה, כלומר גם אחרי הוספתה, הצלעות ב-A מהוות תת-גרף של MST.	
האלגוריתם של קרוסקל	נועד לגרף ממושקל לא מכוון וקשיר. מוצא עץ פורש מינימלי. זהו אלגוריתם חמדן, כלומר כזה המבצע בכל שלב את מה שנראה "טוב ביותר" עבורו בטווח הקצר. באלגוריתם זה, בכל שלב מוכלים ב-A כל צלעות תת-הגרף של ה-MST, כלומר יער של עצים. בהתחלה A ריקה – כל קודקוד מהווה "עץ" בפני עצמו. בכל שלב בלולאה מוצאים "צלע בטוחה" ומצרפים ל A, עד שA הופכת לMST. מימוש האלגוריתם נעשה ע"י שימוש בקבוצות זרות – union find, כל קבוצה מכילה קודקוד. מציאת צלע בטוחה: ממיינים את הצלעות לפי משקלן, זה סדר סריקתן בלולאה. לגבי כל צלע בודקים אם קודקודיה נמצאים בקבוצות זרות. אם כן – מצרפים אותה. אם לא – ממשיכים הלאה. מפסיקים כאשר סיימנו לסרוק את הצלעות.	זמן ריצה: בניית הקבוצות הזרות – O(V). מיון הצלעות: O(ElogE). בניית העץ – עוברים פעם אחת על כל צלע, כלומר E, ועל כל צלע שאנו רוצים להוסיף לעץ עלינו לחבר בין שתי קבוצות זרות דרך union-find, כלומר בזמן ריצה ממוצע של logv. – O(ElogV) סה"כ: $O((V+E\log E+E\log V)$ |E|>|V|-1 ובנוסף O(loge)=O(logV) (משיקולים מתמטיים) ולכן ניתן לרשום סה"כ: $O(E\log V)$

האלגוריתם של פרים	אלגוריתם חמדן גם כן. מוצא עץ פורש מינימלי בגרף ממושקל קשיר (לא מכוון). A מהווה עץ בכל שלב. נתייחס לכל הקודקודים המופיעים ב-A כחלק מ"ענן" אחד. מתחילים בקודקוד רנדומאלי כלשהו ומצרפים לA בכל שלב את הצלע הקלה ביותר שמחברת בין הענן לבין שאר הגרף מחוצה לו. מימוש: נשתמש בתור עדיפויות עבור הקודקודים מחוץ לענן. לכל קודקוד יהיה שדה key ובו המשקל המינימלי של צלע המחברת בינו לבין הענן. $key[v]=\infty$ כאשר אין צלע ישירה מv לענן. אם המפתח אינו 0, סימן שיש צלע כלשהי שמחברת ישירות בין הענן לקודקוד הזה. תחילה – נאתחל את כל ערכי key באינסוף, מלבד קודקוד השורש שהוגרל, ונכניס לתור עדיפויות (נניח שימוש בערימה בינארית). כל עוד יש קודקודים בתור – נוציא את הקודקוד שמפתחו מינימלי (extractMin) ונצרף לA. נעבור על כל שכניו של קודקוד זה ונעדכן את ערכי המפתח key שלהם, רק בהנחה שמשקל הצלע הנבדקת קטן מהמפתח שהיה שם קודם. בסוף התהליך נקבל עץ פורש מינימלי. ***אלגוריתם זה כמעט זהה לאלגוריתם דייקסטרה***	זמן ריצה: אתחול והכנסה לתור עדיפות: O(V). (רק כי אנחנו כבר יודעים שכל המפתחות שוות אינסוף). עידכון מפתח השורש: O(logV). בניית MST – הוצאת קודקוד מינימלי בכל פעם – Vקודקודים, logV עלות עידכון - O(VlogV). עידכון שדות השכנים – על כל צלע עוברים פעם אחת (סה"כ E) ועדכון המפתח שלו בערימה לוקח logV. O(ElogV) סה"כ (משיקולים דומים לשיקולים בפרים): $O(E\log V)$
האלגוריתם של דייקסטרה (לא למבחן)	אלגוריתם למציאת המסלול הקצר ביותר מקודקוד מקור כלשהו לשאר הקודקודים. האלגוריתם עובד על גרפים מכוונים או לא מכוונים בעלי משקולות אי-שליליות. w(u,f) היא פונקצית המשקל של הגרף. רעיון: נשמור בכל רגע קבוצת קודקודים מהגרף, S, נתייחס אליהם כאל "ענן" קודקודים שבכולם כבר חישבנו את המסלול הקצר ביותר. מימוש: נשתמש בתור עדיפויות עבור הקודקודים מחוץ לענן. לכל קודקוד v יהיה שדה d[v] ובו המשקל המינימלי של מסלול בינו לבין קודקוד המקור s. $d[v]=\infty$ כאשר אין מסלול מs לv או שמסלול זה טרם נתגלה. בנוסף יהיה שדה π[v] ובו נשמור את ה"אבא" של כל קודקוד (כלומר הקודקוד ממנו הגענו ל-v השייך למסלול הקל ביותר). תחילה – נאתחל את כל ערכי $d[v]=\infty$, מלבד קודקוד השורש שהוגרל (d[s] מאותחל ב-0), ונכניס לתור עדיפויות (נניח שימוש בערימה בינארית). כל עוד יש קודקודים בתור – נוציא את הקודקוד שמפתחו מינימלי (extractMin) ונצרף לענן S. נעבור על כל שכניו של קודקוד זה ונעדכן את ערכיהם באופן הבא: if(d[u]>d[v]+w(u,v)) then d[u]←d[v]+ w(u,v). כלומר, אם המסלול שמצאנו עכשיו טוב יותר מכל מסלול שמצאנו עד כה, נעדכן את השדה למשקל המסלול החדש, ונעדכן את האבא, π[u], גם כן. בסופו של התהליך יכיל כל קודקוד בגרף שדה ובו המסלול ה"קל" ביותר מ- s עד אליו ומצביע לקודקוד הקודם לו במסלול זה.	חישוב זהה לאלגוריתם פרים

3. **<u>מבנים ושונות:</u>**

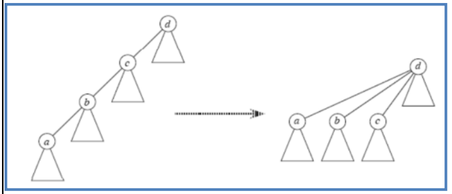
שם מבנה	הסבר כללי	ייצוג גרפי +הערות	תיאור פעולות עיקריות/ מבנה אלגוריתם ומשתניו/ זמני ריצה	הערות חשובות ווריאציות של המבנה
ADT רשימה מקושרת	סדרת אובייקטים המחוברים ע"י מצביעים.		Create(list) – יצירת רשימה ריקה. O(1) Search(k,list) – מוצאת את האיבר הראשון ברשימה עם מפתח k. O(n) במקרה הגרוע. Insert(item, list) – מכניס איבר ספציפי לראש הרשימה. O(1). Delete(item,list) – מוחק איבר ספציפי מהרשימה ומחבר בין זה שלפניו לזה שאחריו. O(n) במקרה הגרוע.	וריאציה נוספת – רשימה מקושרת דו כיוונית – בה לכל איבר יש מצביע לזה שאחריו ולה שלפניו. רשימה מעגלית – הסוף מחובר להתחלה.
ADT תור - Queue	מבני נתונים אבסטרקטי המתנהג כתור ממתינים: הראשון שנכנס הוא הראשון לצאת (FIFO).	רשימה/מערך	Create(Q) – יצירת תור ריק Q. IsEmpty(Q) – מחזיר ערך בוליאני האם התור ריק. enqueue(Q,item) – מכניס את האיבר item לסוף התור. dequeue(Q) – מוחק ומוציא את האיבר ה"ישן" ביותר בתור.	ניתן למימוש או בעזרת מערך או בעזרת רשימה מקושרת. זמני הריצה נותרים זהים. במימוש בעזרת מערך נתייחס למערך כאל רשימה מעגלית, כלומר כשהתור מגיע לסוף המערך מתחילים למלא אותו מראשיתו, כל עוד נשמר התנאי שהתור לא יכיל יותר מגודל המערך פחות אחד.
ADT מחסנית-Stack	מבנה נתונים אבסטרקטי המתנהג כמחסנית: האחרון שנכנס הוא הראשון שיוצא (LIFO).	רשימה/מערך	Create(S) – יצירת מחסנית ריקה S. IsEmpty(S) – מחזיר ערך בוליאני האם המחסנית ריקה. Push(S,item) – דוחף את האיבר item לראש המחסנית. Pop(S) – מוציא מהמחסנית את האיבר האחרון שהוכנס לתוכו ומחזירו.	ניתן למימוש או בעזרת מערך או בעזרת רשימה מקושרת. פעולה מיוחדת: multi-pop – מרוקנת את כל תכולת המחסנית. בחישוב של <u>ניתוח פחת</u>, עבור מחסנית בעלת הפעולות push, pop, multi-pop זמן הריצה הממוצע לפעולה הוא O(1).

שם מבנה	הסבר כללי	ייצוג גרפי +הערות	תיאור פעולות עיקריות/ מבנה אלגוריתם ומשתניו	הערות חשובות ווריאציות של המבנה
עצי חיפוש בינאריים	עץ = מבנה היררכי בו לקודקודים יש יחסי אב-בן. מושגים (חלק): עומק = מס' קשתות משורש לקודקוד מסוים. דרגה של צומת = מספר הבנים. גובה = מס' קשתות מקודקוד מסוים לעלה שהוא הצאצא הכי רחוק ממנו. גובה עץ ריק מוגדר (-1). עץ בינארי = עץ בו לכל צומת יש לכל היותר 2 בנים. עץ בינארי מלא = עץ בו לכל צומת יש 2 או 0 בנים. עץ בינארי מושלם = לכל העלים אותו עומק. עץ חיפוש בינארי מקיים תנאי נוסף: לכל קודקוד, המפתח של בנו הימני גדול משלו והמפתח של בנו השמאלי קטן משלו. עץ זה משמש למימוש מילון דינאמי.	עץ	נסמן ב-h את גובה העץ. במקרה הטוב, ובמקרה הממוצע h=O(logn) (לפי משפט). במקרה הגרוע h=O(n). Search(T,k) – חיפוש מפתח k בעץ T. O(h). ניתן לממש רקורסיבית או איטרטיבית. Minimum/maximum(x) – חיפוש מפתח מקסימלי/מינימלי בתת העץ המושרש ב-x. זמן ריצה: O(h). Insert(T,x) – הכנסת איבר x לעץ. זמן ריצה: O(h). Delete(T,x) – מחיקת איבר x לעץ. זמן ריצה: O(h). תיאור אלגוריתם: תחילה מוצאים את x בעזרת search. אם x הוא עלה – פשוט מוחקים. אם לx בן יחיד – פשוט מעבירים את המצביע מאביו של x אל אותו בן יחיד. אם ל-x שני בנים: נמצא את הקודקוד המינימלי בתת העץ הימני של x ונחליף ביניהם, ואז נמחק את x (בוודאות לא יהיו לו שני בנים ולכן המחיקה תהיה פשוטה). Successor/predecessor(T,x) – מציאת האיבר העוקב/קודם לקודקוד x. תיאור אלגוריתם: אם הבן הימני קיים, קוראים לפונקציית מציאת מינימום על תת העץ המושרש בבן הימני. אם לא – מחפשים את האב הקדמון הקרוב ביותר כך ש-x נמצא בתת העץ המושרש משמאלו. סה"כ – O(h).	קיימות 3 סריקות של איברי עץ חיפוש בינארי: Inorder – בן שמאלי, אני, בן ימני. Preorder – אני, בן שמאלי, בן ימני. Postorder – בן שמאלי, בן ימני, אני.
AVL-tree	זהו עץ חיפוש בינארי בעל תכונה נוספת: לכל קודקוד בצומת מתקיים שגובה הבן הימני וגובה הבן השמאלי נבדלים ב1 לכל היותר. משפט: גובה של AVL בעל n צמתים הוא O(logn).	כל צומת בעץ מחזיק שדה נוסף – גובה תת העץ המושרש בצומת זו.	Insert – הכנסה מתבצעת כמו בעץ חיפוש בינארי רגיל, עם תוספת: לאחר שהקודקוד החדש מוכנס, עולים חזרה למעלה עד השורש ובמידה ומוצאים קודקוד לא מאוזן מבצעים רוטציה יחידה. הרוטציות יכולות להיות מ- 4 צורות: right, left, right-left, left-right, לפי המצב בעץ. זמן ריצה(הכנסה): חיפוש רגיל – O(logn), טיפוס חזרה עד השורש – O(logn), תיקון במידה ונדרש – O(1). סה"כ: O(logn). Delete – מתבצע כמו בעץ חיפוש בינארי רגיל רק שגם הפעם נצטרך לעלות חזרה עד השורש ולבצע רוטציות מתקנות בכל צומת בו התגלה חוסר איזון. בניגוד להכנסה, במחיקה ייתכן ונצטרך לבצע יותר מרוטציה אחת בדרך חזרה לשורש. במקרה הגרוע ביותר נצטרך לתקן כל שלב ושלב בדרך לשורש. זמן ריצה(מחיקה): מחיקה רגילה – O(logn), טיפוס חזרה עד השורש – O(logn) ותיקון בכל שלב אם נחוץ – O(1). סה"כ: O(logn). Search – כמו בעץ חיפוש רגיל. O(logn).	

שם מבנה	הסבר כללי	ייצוג גרפי+ הערות	תיאור פעולות עיקריות/ מבנה אלגוריתם ומשתניו	הערות חשובות
Hash-tables	טבלאות גיבוב נועדו לאפשר מערך דינאמי בעל הפעולות: הכנסה, חיפוש ומחיקה, כאשר טווח ערכי המפתחות גדול בהרבה ממספר המפתחות הנתונים. נתייחס למפתחות כאל מספרים טבעיים. <u>הנחה מקדימה: כל המפתחות שונים זה מזה.</u> פונקציית ה- hash, נסמנה ב- h, תיקח כל מספר טבעי ותצמיד לו ערך בטווח המערך, כלומר מ-0 עד m-1 שהגדרנו. זאת ב-O(1). הבעיה: מפתחות שונים עלולים לקבל אותו ערך (h איננה חח"ע). שני פתרונות: שירשור (chaining) ו- open addressing. מבנה זה נותן זמני ריצה טובים ב **<u>ממוצע</u>**, כאשר מתקיימת הנחת הגיבוב האחיד (ההסתברות ליפול בתא כלשהו במערך שווה ל- 1/m). חישוב זמני הריצה נעזר ב"מקדם העומס". מקדם העומס זהו ממוצע באיברים לכל תא, כלומר: $\alpha = \frac{n}{m}$. <u>Chaining:</u> מכניס לתא i במערך את כל המפתחות שקיבלו את הערך i בפונקציה ושומר אותם כרשימה מקושרת. <u>Open addressing:</u> הרעיון הוא שאם התא אליו איבר מסוים נשלח ע"י h תפוס כבר, האיבר יוכוון לתא אחר. במילים אחרות הפונקציה תקבל גם את המפתח וגם את מספר הבדיקה ותיתן כל פעם אינדקס אחר במערך, באחת משלושת דרכים אלו: 1. linear probing – התקדמות ליניארית לתא הבא במערך עד שמגיעים לתא ריק. שיטה פשוטה אך יוצרת גושים. פרמוטציות אפשריות להצעות למיקומים: O(m). 2. quadratic probing – התקדמות בקפיצה קבועה, למשל: $h(k,i)=(h'(k)+c_1 i+c_2 i^2) \bmod m$. כאשר i מייצג את מספר הקפיצות שביצענו (כלומר מספר הפעמים שהגענו לתא שהיה כבר מלא) ו- c1,c2 קבועים. פועל מעט טוב יותר מהקודם. אותו סדר גודל של פרמוטציות אפשריות. 3. double hashing – משלבים בין שתי פונקציות hash: $h(k,i)=(h_1(k)+i\cdot h_2(k)) \bmod m$. את הפונקציה השנייה נכנה "פונקציית הצעד". כך, שני מפתחות שקיבלו אותו ערך, יוכלו לקבל צעדים שונים במידה ומיקום זה במערך כבר תפוס. פרמוטציות: O(m^2), כלומר הסיכוי להיווצרות גושים קטן. זוהי שיטה עדיפה <u>כאשר אין צורך במחיקה.</u> זמן הריצה של מחיקה בשיטה זו לא יהיה תלוי במקדם העומס α. אם נרצה להוסיף אפשרות מחיקה נוסיף "דגל" בשם "deleted" שיסמן לנו מאיפה מחקנו בעבר איבר. בנוסף המקום בזיכרון מנוצל באופן "חסכוני" יותר.	<u>Chaining</u> מערך באורך m שכל תא בו הוא רשימה מקושרת.	Search(T,k) – נחפש את מפתח k ברשימה המקושרת ב-T[h(k)]. חיפוש לא מוצלח יצרוך זמן ריצה של O(1+α). חיפוש מוצלח יצרוך **<u>ב</u>**<u>ממוצע</u> O(1+α/2), כלומר O(1+α) גם כן. (מכאן שאם n=O(m) אזי זמן הריצה הוא קבוע O(1). Insert(T,k) – מכניס את איבר x בראש הרשימה שבתא T[h(x.key)]. זמן ריצה: O(1). Delete(T,k) – מחיקת האיבר x מהרשימה שבתא T[h(x.key)]. זהה לזמני חיפוש.	הערה – בחירת פונקציית hash – יש שתי שיטות עיקריות: שיטת החילוק – $h(k)=k \bmod m$ (יעבוד טוב עבור m ראשוני שרחוק ככל הניתן מחזקה של 2). שיטת הכפל – $h(k)=floor[m\cdot(kA \bmod 1)]$ כאשר A בין 0 ל-1. (שיטה זו איטית יותר אך ערך m כאן אינו קריטי ואף יעיל כאשר m=2^p). בחירת פונקציית הצעד: צריך שלא תהיה בעלת מחלק משותף עם m. בפרט, אם m זוגי עליה להיות אי-זוגית. ---- יתרונו הגדול של מבנה זה הוא אך ורק <u>במקרה הממוצע</u>, תחת הנחת הגיבוב האחיד. (בשרשור למשל, במקרה הגרוע ביותר החיפוש יעלה לנו O(n)). ---- וריאציה לצורך הקטנת ההסתברות ליפול באותו התא: עבודה עם מס' טבלאות זהות בגודלן במקביל, לכל טבלה פונקציה משלה. איבר קיים אך ורק אם הוא נמצא בכל אחת מהטבלאות.
		<u>Open addressing</u> מערך באורך m	Search(T,k) – חיפוש לא מוצלח ייקח בממוצע $O\left(\frac{1}{1-\alpha}\right)$ וחיפוש מוצלח ייקח בממוצע $O\left(1+\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)\right)$. Insert(T,k) – זמן ריצה בממוצע: $O\left(\frac{1}{1-\alpha}\right)$	

שם מבנה	הסבר כללי	ייצוג גרפי +הערות	תיאור פעולות עיקריות/ מבנה אלגוריתם ומשתניו	הערות חשובות
Huffman code	קוד הופמן הוא שיטה לדחיסת נתונים ע"י הצגת סימנים נפוצים באמצעות מספר סיביות קטן לעומת סימנים נדירים במספר סיביות גדול יותר. הקוד הוא "קוד תחיליות"--prefix code, כלומר כל מחרוזת ביטים שמייצגת אות, איננה יכולה להיות תחילית של מחרוזת ביטים אחרת. משמעות: פענוח הקוד הוא חד-ערכי.	עץ בינארי. כל קודקוד מכיל את סך השכיחויות של העלים תחתיו (ומייצג 0 או 1). כל עלה מייצג תו ומכיל את שכיחות תו זה בטקסט/קלט הנתון. ה"דרך" לתו מסוים הוא הקידוד שלו. ניתן לממש בעזרת מערך.	C = סדרה של n תווים. f(c) = השכיחות של התו c. d(c) = עומק העלה c בעץ. = אורך קידוד התו. האלגוריתם של הופמן בונה "עץ" המייצג את הקידוד: מתחילים מתחתית העץ, כל עלה הוא תו. מכניסים לערימת מינימום את כל סדרת התווים ושכיחויותיהם. מושכים מהערימה את שתי השכיחויות המינימליות. יוצרים קודקוד חדש שהוא סכום שני הקודקודים שהוצאנו ומכניסים אותו לערימה. מבצעים את הלולאה עד שנשאר איבר אחד בערימה – הוא שורש העץ. זמן ריצת האלגוריתם: O(nlogn).	- באמצעות קוד הופמן ניתן לחסוך 20-90% מקום בזיכרון. - טענה: <u>כאשר הקוד הוא אופטימלי, העץ יהיה עץ מלא.</u> כלומר, לכל צומת שאינו עלה יהיו בדיוק שני בנים. לדוגמה, נניח כי בקוד כלשהו לשורש של העץ יש רק בן אחד, שאליו מובילה הקשת 0. פירוש הדבר הוא שבקוד לא יהיו כלל אותיות שקידודן מתחיל ב-1, וזהו כמובן בזבוז של מקום (כי אפשר היה, לכל הפחות, לתת לאות כלשהי את הקידוד "1", ובכך לחסוך במקום).
B-tree	זהו מבנה המבוסס על עץ ונועד לצמצם את מספר הפניות לדיסק החיצוני כאשר בסיס הנתונים גדול מדי. (כלומר לקרוא כמה שיותר מהדיסק בבת אחת במקום לגשת אל הדיסק כמה פעמים). הדבר נעשה ע"י דחיסת כמה שיותר מפתחות ומצביעים בקודקוד יחיד. לכל עץ קיים קבוע t>=2 שלפיו נקבעים החסמים העליונים והתחתונים למספר המפתחות בצומת: כל איבר מלבד השורש יכיל לפחות (t-1) מפתחות ולכל היותר (2t-1) מפתחות.	עץ. n[x] – מס' המפתחות שבכל צומת של העץ. המפתחות עצמם ממוינים מקטן לגדול ומצביעים מוחזקים לפני ואחרי כל מפתח. בעץ בעל n[x] מפתחות מוחזקים (n[x]+1) מצביעים: c1[x], c2[x], ... **<u>כל העלים תמיד באותו גובה.</u>**	Leaf[x] – שיטה החזירה ערך בוליאני. אמת – אם הצומת הוא עלה, שקר – אחרת. Search(x,k) – החיפוש יתחיל מקודקוד x, (או מהשורש) ויסרוק את המפתחות שבקודקוד עד מציאת k או עד הגעה לערך הגדול מ-k. לפי הסריקה – או שמחזירים את המיקום הרלוונטי, או שקוראים לשיטה רקורסיבית על הקודקוד אליו יש הצבעה לפני המפתח שגדול מk, או שמחזירים null אם k לא נמצא. זמן ריצה: O(t*h) = O(tlogn). מס' גישות לדיסק: O(h) = O(logn). Create – יוצר עץ ריק ב-O(1) זמן ריצה וגישות לדיסק. splitChild(x,I,y) – פונקציית עזר שמפצלת את הקודקוד y המלא, שהוא בנו של x. i הוא אינדקס הבן. זמן ריצה: O(t). Insert(t,k) – מתחילים מהשורש ויורדים למטה. כל קודקוד שמכיל מקסימום מפתחות (2t-1) מפוצל לשניים על ידי העברת המפתח האמצעי לאביו (לשם כך יש לשמור הצבעה). רק כאשר נגיע לקודקוד שהוא עלה ואינו מלא – נכניס את המפתח במקום המתאים. זמן ריצה: O(tlogn). מס' גישות לדיסק: O(logn). Delete – חלוקה למקרים לפי המצאות המפתח בעלה או בצומת פנימית ולפי מס' המפתחות בצומת (יש רף תחתון). פירוט – בתירגול. זמן ריצה – כמו הכנסה.	- <u>משפט:</u> גובה העץ, h, חסום ע"י $h \le \log_t \frac{n+1}{2}$. כאשר n מספר המפתחות סה"כ בעץ. - שורש העץ תמיד ישמר בזיכרון הראשי כדי שקריאתו לא תצריך disk-read.

שם מבנה	הסבר כללי	ייצוג גרפי +הערות	תיאור פעולות עיקריות/ מבנה אלגוריתם ומשתניו	הערות חשובות
Skip-list	דרך למימוש מילון דינאמי. זהו אלגוריתם רנדומאלי. רשימת דילוגים עבוד קבוצת מפתחות זרים היא סדרה של רשימות מקושרות המקיימת: (1) כל רשימה מתחילה ב- $-\infty$ ומסתיימת ב- ∞. (2) הרשימה הראשונה S0 מכילה את כל המפתחות. (3) כל רשימה היא תת רשימה של הקודמת לה. (4) הרשימה האחרונה בסדרה תכיל רק את שני המפתחות המיוחדים $-\infty$, ∞. (5) לכל איבר יש סיכוי של 50% להופיע ברשימה מעליו.	רשימות מקושרות זו מעל זו. מימוש בעזרת קודקודים עם 4 מצביעים (אחד לכל כיוון).	חיפוש – נתחיל באיבר $-\infty$ ברמה העליונה וכל פעם שנתקל באיבר הגדול מk נרד רמה אחת מטה ונמשיך להתקדם בחיפוש.אם אין יותר לאן לרדת – החיפוש הסתיים לא מציאת המפתח k. זמן ריצה במקרה הגרוע: O(n), במקרה ה**ממוצע**: O(logn). הכנסה – תחילה מוצאים את המיקום המתאים (משתמשים בחיפוש). מכניסים את המפתח לרמה התחתונה ומגרילים האם יעלה לרמה שמעליו. אם יצא שכן – מגרילים שוב לרמה הבאה שמעליה, כך עד הרמה העליונה. **יש לקבוע מדיניות לגבי הגעה לרמה העליונה בעת הכנסה. גישה 1: נגביל את מספר הרמות למספר מסוים (אולי כתלות בn). גישה 2: לא נעצור את התהליך ונסתמך על כך שההסתברות לכך תהיה נמוכה. זמן ריצה במקרה הגרוע: O(n), במקרה ה**ממוצע**: O(logn). מחיקה – נמצא את האיבר ונמחק אותו ואת המגדל מעליו. זמן ריצה – כמו של חיפוש/הכנסה. פעולות ב-O(1): מציאת מינימום, מציאת מקסימום, עוקב ומקדים. סיבוכיות מקום: תלויה בתהליך הרנדומאלי של ההכנסות. לפי חישובי הסתברות, סיבוכיות המקום ה**ממוצעת** היא O(n).	עובדות הסתברותיות: - ההסתברות שאיבר יעלה לרמה הבאה היא 1/2. - ההסתברות שאיבר מסוים יכנס לרמה i היא $1/2^i$. - ההסתברות שרשימה i תהיה בעלת איבר אחד לפחות: $n/2^i$.
ערימה - heap	ערימות יכולות להיות מסוג מקסימום או מינימום, מיוצגת ע"י עץ. ערימת מקסימום: כל קודקוד גדול מבנו הימני וגדול מבנו השמאלי. השורש הוא האיבר המקסימלי בערימה. ניתן להצגה כמערך.	עץ (לרוב בינארי) מלא, כך שהרמה האחרונה בו מלאה משמאל ועד נקודה מסוימת. גובה העץ – logn	Length(A) = מס' איברים מקסימלי בערימה = אורך המערך. Heap-size(A) = מס' איברים בערימה. A(1) = שורש הערימה. לכל אינדקס i: ערימת מקסימום – A(parent(i))>=A(i) Parent(i) = i/2 (ערך תחתון) Left(i) = 2i Right(i) = 2i+1 פעולות: בניית הערימה – חישוב נאיבי: O(nlogn). **בחישוב הדוק מקבלים: O(n).** (בהינתן מערך כלשהו, מיון ערימה <u>על מערך זה</u>, ללא בניית ערימה נפרדת, ייקח O(nlogn) במקרה הגרוע). insert. רעיון הכנסה – מוסיפים עלה ע"י הוספת האיבר החדש במקום הראשון הפנוי במערך ואז מטפסים עד השורש ומחליפים בין 2 קודקודים בכל פעם שאינם מקיימים שהעליון גדול מהתחתון. – O(logn) הוצאה – מחליפים את הקודקוד האחרון עם זה שרוצים להסיר ואז מבצעים שורת תיקונים אחת כלפי מטה ואחת כלפי מעלה. O(logn) Heap-max - מציאת ערך מקסימלי – O(1) Heap-extract-max - שליפת ערך מקסימלי - O(logn). Max-heapify: עידכון הערימה לערימת מקסימום כאשר איבר אחד אינו במקומו: לכל איבר נבדקים שני בניו. אם אחד נמצא גדול יותר – הוא מוחלף עם אביו. וממשיכים לרדת דרך האב. – O(logn).	משפט: מספר ה<u>עלים</u> בערימה הוא n/2 (ערך עליון). ניתן להוסיף שיטה המאפשרת הגדלת אחד המפתחות שכבר בעץ – heap-increase-key. אם המפתח כבר גדול יותר – לא משנים כלום. יידרש תיקון כלפי מעלה (ערימת מקסי').

<table>
<tr>
<td>ADT
תור – עדיפות
Priority queue</td>
<td>תור בו לכל איבר יש עדיפות מסוימת וניתן להכניס איבר חדש עם עדיפות ספציפית, כלומר הסדר לא מבוסס רק על סדר הכניסה.</td>
<td></td>
<td>פעולות: Insert(S,x) מקבל מערך S ומכניס אליו את האיבר עם מפתח x. O(logn) (או: enqueue)
Maximum(S) – מחזיר את האיבר בעל המפתח הגדול ביותר במערך. O(1)
extractMax(S) – מוציא ומחזיר את האיבר המקסימלי. O(logn) (או: dequeue).
increaseKey(S,x,k) – מגדיל את ערך המפתח x ל-k. O(logn)
(באופן מקביל פועל תור מינימום).</td>
<td></td>
</tr>
<tr>
<td rowspan="2">קבוצות – זרות
disjoint sets</td>
<td rowspan="2">מבנה הממשש אוסף קבוצות זרות. כל קבוצה בעלת נציג כלשהו (לרוב אין חשיבות לנציג). שתי דרכי מימוש עיקריות: יער (כל קבוצה מיוצגת ע"י עץ) ורשימות מקושרות. דרך המימוש מכתיבה את זמן הריצה במקרה הגרוע.</td>
<td>רשימות מקושרות.
כל קבוצה היא רשימה (עם מצביעים לראש ולזנב) וכל איבר בקבוצה בעל מצביע לאיבר הבא ובעל מצביע לנציג קבוצתו.</td>
<td>makeSet(x) – יצירת רשימה עם איבר אחד O(1).
union(x,y) – מחבר בין ראש הקבוצה של y לבין זנב הקבוצה של x ומעדכן בכל איברי הקבוצה של y את הנציג להיות הנציג של x.
שיפור: איחוד לפי משקל. שומרים לכל קבוצה שדה גודל וכאשר מבצעים איחוד מחברים תמיד את הקבוצה הקטנה לזו הגדולה. שינוי בזמני ריצה: הפעולה עצמה עדיין יכולה להיות בסדר גודל O(n/2) כלומר ללא שינוי, אבל בניתוח פחת מקבלים שזמן הריצה הממוצע לכל פעולה מתוך סדרה של m פעולות יהיה O(logn).
find(x) – מחזיר מצביע לנציג הקבוצה ע"י השדה המתאים ב-x O(1).</td>
<td rowspan="2">ניתוח זמן ריצה של מבנה זה מתבצע ע"י ניתוח פחת: חישוב זמן הריצה של סה"כ m פעולות משלושת הסוגים,find כאשר נעשו n פעולות makeSet מתוכן.
m>=n. (מכאן שלכל היותר נעשו n-1 פעולות union).</td>
</tr>
<tr>
<td>יער עצים.
כל קבוצה היא עץ. שורש כל עץ הוא נציג הקבוצה.
כל צומת בעץ מצביע אך ורק לאביו.</td>
<td>makeSet(x) – יצירת עץ עם איבר אחד והוא השורש O(1).
union(x,y) – הופך את שורש העץ הראשון לבן של שורש העץ השני. O(1).
find(x) – מטפס מקודקוד x אל שורש העץ בו x נמצא. מחזיר את השורש שהוא נציג הקבוצה. במקרה הגרוע – העץ הוא פשוט שרשרת של האיברים ולכן זמן הריצה יהיה O(n).
שיפורים:
1. איחוד לפי דרגה: נחבר את העץ בעל הדרגה הקטנה יותר כילד של שורש העץ בעל הדרגה הגדולה יותר. נקטין בכך את זמן הריצה של פעולת ה-find.
2. path-compression: נשנה את פעולת ה-find באופן הבא: כשאנו מטפסים מקודקוד x לשורש, נהפוך כל קודקוד דרכו אנו עוברים לבן ישיר של השורש.

זמן ריצה לאחר שיפורים: בניתוח פחת נקבל O(logn) ממוצע לפעולה.</td>
</tr>
</table>

4. **אלגוריתמים נוספים:**

שם האלגוריתם	הסבר	זמן ריצה + הסבר	הערות חשובות ווריאציות
Randomized-select(A,p,r,i)	אלגוריתם למציאת האיבר הk בגודלו, מתוך n איברים. המספר ה-k בגודלו הוא זה שיש בדיוק (k-1) איברים לפניו. חציון הוא המספר במקום ה-floor[n/2]. אלגוריתם זה הוא רקורסיבי ומקבל כקלט את A – המערך, p – ערך ממנו נתחיל לחפש, r – ערך בו נפסיק לחפש, i – מיקום הערך הרצוי. בכל פעם, האלגוריתם יתמקד בטווח מסוים שניתן לו במערך ויבחר באופן אקראי איבר שיתפקד כ-pivot. בשלב הבא יחולקו איברי המערך כך שאלו הגדולים מהפיבוט יהיו אחריו במערך ואלו הקטנים ממנו יהיה לפניו. (באופן זה אנו יודעים שהפיבוט עצמו מונח במקומו). אם מיקום הפיבוט = i, סיימנו. אם לא, נפעיל מחדש את האלגוריתם, הפעם על ערכי הטווח הרלוונטי (מימין או משמאל לפיבוט), ועדכון i במידת הצורך. (הסבר מפורט - http://www.cs.technion.ac.il/~bshouty/DS/OTHER/M10sortB-Slide-Version.pdf)	<u>במקרה הממוצע – O(n).</u> (הוכחת זמן הריצה מבוססת על הוכחה דומה לזו של מיון מהיר) במקרה הגרוע (כלומר הפיבוט הנבחר יקטין את הטווח הנסרק ב-1 כל פעם) – O(n^2).	
select(A,p,r,i)	אלגוריתם זה מהווה תוספת לאלגוריתם מעלה. מחלקים מערך לקבוצות בגודל 5. (בקבוצה האחרונה יהיו 1-5 איברים). כל קבוצה ממוינת וחציונה נשמר במערך חדש (באורך A/5). משווים בין כל החציונים ומוצאים את החציון מביניהם. האיבר שמצאנו ייבחר כפיבוט והמשך האלגוריתם הוא כמו זה הנ"ל. (פירוט בלינק מעלה).	O(n) – במקרה גרוע.	בשאלות בהם אנו נדרשים לבצע מיון, וידוע שיש ערך שמופיע n/2 פעמים ומעלה – נצטרך להסיק שמספר זה הוא בהכרח גם החציון ולכן אלגוריתם זה יוכל למצוא עבורנו את המספר בזמן ריצה לינארי.

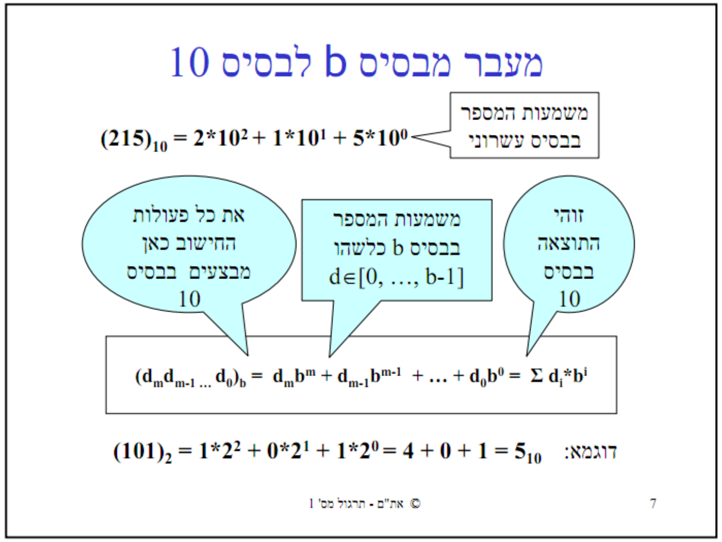
5. **טבלת זמני ריצה מקוצרת:**

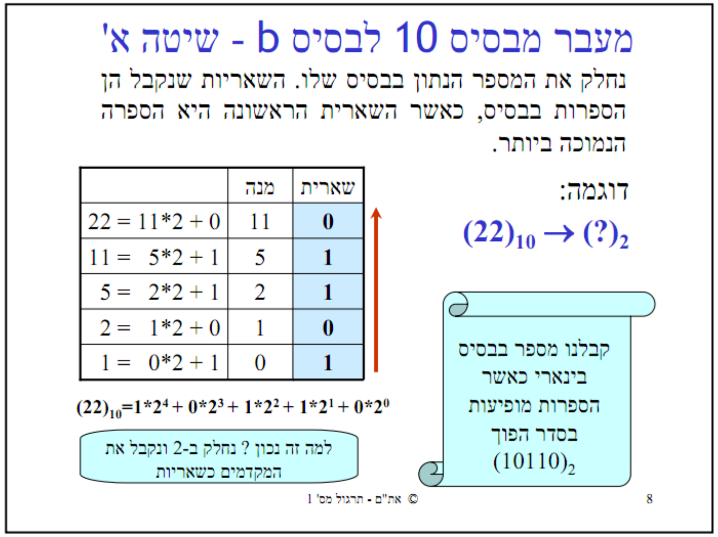
מבנה הנתונים	זמן ריצה: הכנסה	זמן ריצה: חיפוש איבר ספציפי	זמן ריצה: מציאת מינימלי/ מקסימלי	זמן ריצה: חיפוש successor/ predecessor	זמן ריצה: הוצאת מינימלי/ מקסימלי	זמן ריצה: מחיקה	סיבוכיות מקום
עץ חיפוש בינארי	$O(h)$ במקרה הגרוע: $O(h)=O(n)$ במקרה הממוצע: $O(h)=O(\log n)$	$O(h)$ במקרה הגרוע: $O(h)=O(n)$ במקרה הממוצע: $O(h)=O(\log n)$	$O(h)$ במקרה הגרוע: $O(h)=O(n)$ במקרה הממוצע: $O(h)=O(\log n)$	$O(h)$ במקרה הגרוע: $O(h)=O(n)$ במקרה הממוצע: $O(h)=O(\log n)$	$O(h)$ במקרה הגרוע: $O(h)=O(n)$ במקרה הממוצע: $O(h)=O(\log n)$	$O(h)$ במקרה הגרוע: $O(h)=O(n)$ במקרה הממוצע: $O(h)=O(\log n)$	$O(n)$
AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
B-tree	$O(t\log n)$	$O(t\log n)$	-	-	-	$O(t\log n)$	-
Hash-table Open-addressing	במקרה הממוצע (תחת הנחת גיבוב אחיד): $O\left(\frac{1}{1-\alpha}\right)$	חיפוש כושל, בממוצע: $O\left(\frac{1}{1-\alpha}\right)$ חיפוש מוצלח, בממוצע: $O\left(1+\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)\right)$	-	-	-	-	-
Hash-table chaining	$O(1)$	מקרה ממוצע: $O(1+\alpha)$	-	-	-	מקרה ממוצע: $O(1+\alpha)$	-
Skip-list	$O(n)$ מקרה ממוצע: $O(\log n)$	$O(n)$ מקרה ממוצע: $O(\log n)$	$O(1)$	$O(1)$	-	$O(n)$ מקרה ממוצע: $O(\log n)$	מקרה ממוצע: $O(n)$
ערימת מינימום/מקסימום	$O(\log n)$	$O(n)$	$O(1)$	-	$O(\log n)$	$O(\log n)$	$O(n)$
תור עדיפות	$O(\log n)$	-	$O(1)$	-	$O(\log n)$	-	$O(n)$

** אלא אם מצוין אחרת, זמן הריצה הוא worst-case.

6. **נספחים:**

- מעבר מבסיס לבסיס:





- **ניתוח פחת – Amortized analysis**

http://www.cs.bgu.ac.il/~algaf083/wiki.files/Amortized%20Analysis.pdf