בס"ד

סוכם ע"י: אלמוג יעקב

סיכום - מבני נתונים

סמסטר קיץ 2020

מבני נתונים:	הסבר מקוצר:	שימושים:
רשימות מקושרות (Linked List)	רשימה שבה כל איבר מצביע על האיבר הבא אחריו. רשימה מקושרת דו כיוונית כוללת הצבעה של איבר לאיבר הקודם לו.	מעקב אחר נתונים המקושרים אחד לשני בצורת שרשרת (כמו בנגני מוזיקה עם לחצן "הקודם" ו"הבא").
מחסנית (Stack)	מבנה נתונים מופשט שמזכיר מחסנית של רובה: האיבר שנכנס ראשון למחסנית יוצא ממנה אחרון (נכנס אחרון יוצא ראשון - LIFO).	ביטול פעולה/ביצוע מחדש במעבדי תמלילים, הערכת ביטוי וניתוח תחביר.
תור (Queue)	מבנה נתונים מופשט שמזכיר תור של בני אדם: האיבר שנכנס ראשון לתור יוצא ממנו ראשון (נכנס ראשון יוצא ראשון - FIFO).	אחסנה לעיבוד וביצוע פעולות מאוחר יותר. לדוג', הדפסת מסמכים (מסמכים שנשלחו להדפסה בסוף יאוחסנו עד לביצוע בסוף).
גרף (Graph)	גרף הוא מבנה נתונים בו הנתונים נשמרים באוסף של קודקודים (צמתים) מקושרים, וקצוות (נתיבים).	משמש לדוג' לייצוג רשתות (עשוי לכלול שבילים כמו ברשת טלפונית), רשת חברתית (כאשר אדם מיוצג ע"י צומת ופרטיו בצומת).
טבלת גיבוב (Hash Table)	מימוש של מילון המשתמש בפונקציית גיבוב על-מנת להקטין את תחום המפתחות ולשמרם במערך לצורך שליפה מהירה.	משמש לחיפוש נתונים מהיר - טבלת סמלים עבור מהדרים, אינדוקס של מסדי נתונים, מטמונים, ייצוג נתונים ייחודי, מילונים.
• עץ בינארי - דרגתו של כל קודקוד בעץ היא לכל היותר 2 (שני בנים לכל היותר). דרגת השורש לכל היותר 1. • עץ חיפוש – מבנה נתונים ממוין העונה להגדרת עץ (גרף לא מעגלי). • עץ חיפוש בינארי – מקיים את תכונות עץ חיפוש ועץ בינארי יחדיו. • עץ מאוזן - עץ חיפוש בינארי השומר על גובה מינימלי תחת פעולות הכנסה והוצאה של צמתים. • ערימה - מבנה נתונים בצורת עץ מכוון המקיים תכונה בסיסית, הנקראת תכונת הערימה: המפתח של כל צומת בעץ קטן ממפתח אביו (ערימת מקסימום) או שהמפתח של כל צומת בעץ גדול ממפתח אביו (ערימת מינימום).		
עץ AVL (משפחת עצי חיפוש) (משפחת עצים מאוזנים)	עץ חיפוש בינארי שמתקן את עצמו תוך כדי בנייה באמצעות "גלגולים" כך שגובהו יישאר נמוך יחסית למספר האיברים בו.	אחסון נתונים בצורה מאוזנת לצורת שליפה מהירה (חיפוש, הוספה, מחיקה, הוצאת ערך מינימלי/מקסימלי וכו').
עץ אדום-שחור (משפחת עצי חיפוש) (משפחת עצים מאוזנים)	עץ חיפוש בינארי הבנוי לפי הגבלות שמצמצמות גובה הבנויות סביב חלוקה של צמתיו לשתי קבוצות. (שומר גובה ע"י "גלגולים" ו"צביעה").	אחסון נתונים בצורה מאוזנת לצורת שליפה מהירה (חיפוש, הוספה, מחיקה, הוצאת ערך מינימלי/מקסימלי וכו').
עץ +B (משפחת עצי חיפוש)	עץ חיפוש שבו לכל צומת מספר גדול של בנים, כך שגובהו קטן יחסית למספר האיברים שהוא מכיל (גישה מהירה ויעילה למידע).	אינדוקס מסדי נתונים (במיוחד מסדי נתונים גדולים). מייעל חיפוש (משמעותית) כאשר ישנם ערכות גדולות לא ממוינות של נתונים.
ערימה (heap) **בינארית**. (משפחת עצים בינאריים)	בנוסף לתכונת הערימה, ערימה בינארית היא עץ כמעט שלם (מלאה בכל הרמות פרט אולי לאחרונה המלאה מצד שמאל עד לנק' מסוימת).	הקצאת זיכרון דינמי. [בזכות תכונתה (עץ כמעט שלם) ניתן לאחסן את הערימה במערך ולגשת לאב/בנים באמצעות נוסחה].
ערימה (heap) **בינומית**.	בנוסף לתכונת הערימה, בערימה הבינומית יש לכל היותר עץ אחד שזהוי דרגת שורשו (כלומר, כל העצים בערימה הם מאורך שונה).	הקצאת זיכרון דינמי. [ערימה בינומית מסוגלת לאפשר מיזוג עם ערימה נוספת מסוגה בזמן מהיר].
איחוד קבוצות זרות (Union Find)	מבנה נתונים המאפשר מעקב אחר קבוצות זרות וביצוע איחוד שלהם, וחיפוש הקבוצה המתאימה לאיבר ביעילות גבוהה מאוד.	שימושי עבור הבדלה מסוימת בין נתונים ואיחוד שלהם. לדוג', בעיצוב תמונה מבוססת שכבות נוכל להבדיל בין אובייקט לרקע.

מיונים

	מקרה טוב	מקרה ממוצע	מקרה רע
מיון מהיר Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
מיון מיזוג Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
מיון בועות Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
מיון הכנסה Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
מיון בחירה Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
מיון דליים Bucket Sort	$O(n+k)$	$O(n+k)$	$O(n^2)$
מיון בסיס Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$
מיון מניה Counting Sort	$O(n+k)$	$O(n+k)$	$O(n+k)$

N/A

מבני נתונים

	מקרה ממוצע				מקרה רע			
	גישה	חיפוש	הכנסה	מחיקה	גישה	חיפוש	הכנסה	מחיקה
מערך Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
רשימה מקושרת Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
מחסנית Stack	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
תור Queue	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
טבלת גיבוב Hash Table	N/A	$O(1)$	$O(1)$	$O(1)$	N/A	$O(n)$	$O(n)$	$O(n)$
עץ חיפוש בינארי Binary Search Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
עץ AVL AVL Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
עץ אדום-שחור Red-Black Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
ערימה בינארית Binary Heap	$O(1)$	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$
ערימה בינומית Binomial Heap	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$?	$O(\log n)$
איחוד קבוצות זרות Union Find	N/A	$O(n)$	N/A	N/A	N/A	$O(n)$	N/A	N/A
עץ B+ B+ Tree	N/A	$O(\log n)$	$O(\log n)$	$O(\log n)$	N/A	$O(\log n)$	$O(b\log n)$	$O(b\log n)$

עץ חיפוש בינארי

<u>טענה 1</u>: בעץ חיפוש בינארי מושלם ברמה שמרחקה עד השורש הוא k יש 2^k צמתים - הוכחה באינדוקציה.

<u>הוכחה באינדוקציה</u>:

א) בסיס אינדוקציה: $k = 0$, העץ מורכב משורש בלד, מספר צמתים הוא $2^0 = 1$, עבור $k = 0$ הטענה נכונה

ב) הנחת אינדוקציה: הטענה נכונה עבור k , כלומר מספר צמתים ברמה שמרחקה עד השורש הוא k שווה 2^k.

ג) צריך להוכיח כי הטענה נכונה עבור $k + 1$, כלומר מספר צמתים ברמה שמרחקה עד השורש הוא $k + 1$ שווה 2^{k+1}. אָמנָם, לכל צומת ברמה k יש שני בנים, לכן מספר צמתים ברמה $k + 1$ שווה (לפי הנחת אינדוקציה) $2 \cdot 2^k = 2^{k+1}$.

<u>מטענה זו נובע</u> כי לעץ מושלם בגובה h יש 2^h עלים.

<u>טענה 2</u>: בעץ חיפוש בינארי מושלם בגובה h יש $2^{h+1} - 1$ צמתים - הוכחה ע"פ טענה 1.

<u>הוכחה:</u> מספר צמתים n בעץ בינארי מושלם שווה לסכום צמתים בכל הרמות.

לפי טענה 1 ניתן לכתוב כי $n = 1 + 2 + \cdots + 2^h = \frac{2^{h+1}-1}{2-1} = 2^{h+1} - 1$.

מכאן נובע כי בהינתן מספר צמתים n של עץ מושלם, הגובה שלו שווה:

$$h = \log_2(n + 1) - 1 \quad \rightarrow O(\log_2 n)$$

<u>מסקנה:</u> מספר מקסימאלי של צמתים בעץ חיפוש בינארי לא עלה על 2^{h+1} , כאשר h הוא גובה העץ.

לכן סיבוכיות של חיפוש איבר בעץ חיפוש בינארי מושלם היא $O(\log_2(n))$.

המקרה הטוב

כאשר עץ חיפוש בינארי קרוב למושלם,

סיבוכיות של חיפוש איבר והוספת/מחיקת איבר היא $\mathbf{O(\log_2 n)}$.

המקרה הגרוע

בעץ שאינו מושלם, הסיבוכיות משתנה. לדוגמה, בעץ שכל איבריו מסודרים כבנים ימניים (באלכסון ממש), גובה העץ הוא כמספר האיברים (פחות אחד). לפיכך, הסיבוכיות היא כמו במבנה נתונים ליניארי $\mathbf{O(n)}$.

מחיקת איבר

קיימות 4 אפשרויות למיקומו:

1. האיבר <u>אינו נמצא</u> - אין צורך לבצע מחיקה.

2. האיבר הוא <u>עלה</u>, צריך רק לשחררו ובמקומו לשים **null**.

3. האיבר הוא צומת פנימי, שיש לו <u>בן אחד בלבד</u> - יש למחוק אותו ולהעביר את בנו במקומו

4. האיבר הוא צומת פנימי, שיש לו <u>שני בנים</u>, לא ניתן להעביר את אחד הבנים במקומו, מכיוון שהעברה כזו תגרום לשיבוש המיון כלפי מטה. אנו נשאף להעביר במקום הצומת שעומד להימחק, איבר שיקיים את התנאי שיהיה גדול מתת-העץ הימני שלו וקטן מתת-העץ השמאלי שלו.

איבר זה הוא: (או)

א) האיבר הקטן ביותר בתת-העץ הימני,

ב) האיבר הגדול ביותר בתת-העץ השמאלי.

<u>במחיקת איבר שיש לו שני בנים יש שני מקרים</u>: (לפי שיטה א')

4.1 <u>לבן הימני יש רק בן אחד</u> (לא משנה שמאלי או ימני). במקרה כזה מחליפים אותו <u>בבן היחיד</u> שלו.

4.2 לבן הימני שלו יש <u>שני בנים</u> – רצים עד לאיבר הקטן ביותר בתת עץ ימני (הולכים לבן הימני ואז רצים שמאלה עד הסוף) מחליפים את ערכו של האיבר אותו רוצים למחוק בערכו של האיבר הקטן ביותר. אם לאיבר הקטן ביותר יש בן ימני אזי את האיבר הקטן ביותר יחליף הבן הימני שלו. אחרת נמחק את האיבר הקטן ביותר.

(מימוש כל הקודים בסוף המצגת שלאליזבט תחת הנושא: "עץ חיפוש בינארי").

עץ AVL

כל צומת בעץ שומר, פרט למידע הסטנדרטי של עץ חיפוש, גם מאפיין נוסף הנקרא "גורם האיזון" **(Balance factor)**. מאפיין זה הוא ההפרש (בערך מוחלט) בין גובהו של התת-עץ הימני של הצומת וגובהו של התת-עץ השמאלי של הצומת:

$$|\text{height}(v.\text{right}) - \text{height}(v.\text{left})| \le 1$$

חישוב מספר מינימאלי m_h של קדקודים בעץ AVL בגובה h:

במקרה כללי לשורש של עץ AVL בעל גובה h יש תת-עץ ימני בגובה $h-1$ בעל מספר צמתים מינימאלי ותת-עץ שמאלי בגובה $h-2$ גם בעל מספר צמתים מינימאלי.

לכן, $m_h = m_{h-1} + m_{h-2} + 1$, וגם מתקיים אי-שוויון: $m_h > m_{h-1}$, כאשר $h \ge 1$.

קיבלנו נוסחה רקורסיבית למספר קודקודים מינימאלי של עץ AVL בעל גובה h.

נשתמש בנוסחה זו להוכחת המשפט הבא.

משפט: (עץ מאוזן) גובה h מקסימאלי של עץ AVL בעל n קודקודים מקיימת את האי-שיווין הבא:

$$h < 2 * \log_2 n$$

<u>הוכחה:</u> נסמן $n = m_h$:

$$n = m_h = m_{h-1} + m_{h-2} + 1 > 2m_{h-2} > 2^2 m_{h-4} > \ldots > 2^{h/2}$$

לכן $h < 2\log_2 n$ או $h = O(\log_2 n)$

כלומר, העץ מאוזן.

(ניתן להחליף את m_{h-2} ב-m_{h-k} ולהוכיח איזון באופן דומה עבור עץ בעל לכל היותר k צמתים)

דוגמה: $h = 8$, $n = m_8 = m_7 + m_6 + 1 > 2m_6 > 2^2 m_4 > 2^3 m_2 > 2^4 m_0 = 2^{8/2}$

הערה. בגלל שנוסחה לחישוב m_h דומה לחישוב מספרי פיבונאצ'י F, אפשר להשוות את הסדרות ולראות כי: $m_h = F_{h+2} - 1$.

$$m_0 = 1,\ m_1 = 2,\ m_2 = 4,\ m_3 = 7,\ m_4 = 12,\ m_5 = 20,\ m_6 = 33,\ m_7 = 54, \ldots$$

$$F_0 = 1,\ F_1 = 1,\ F_2 = 2,\ F_3 = 3,\ F_4 = 5,\ F_5 = 8,\ F_6 = 13,\ F_7 = 21,\ F_8 = 34,\ F_9 = 55, \ldots$$

טענה: $m_h = F_{h+2} - 1$

<u>הוכחה:</u> (באינדוקציה)

בסיס: עבור $h = 0$ טענה נכונה: $m_h = F_{h+2} - 1$.

הנחה: $m_h = F_{h+2} - 1$

שלב ההוכחה: $m_{h+1} = m_h + m_{h-1} + 1 = F_{h+2} - 1 + F_{h+1} - 1 + 1 = F_{h+3} - 1$.

עד כאן ההוכחה.

מבנה קודקוד העץ מכיל (בנוסף למשתנים הקיימים בעץ חיפוש בינארי) את המשתנה Balance.

הפונקציה set balance מבצעת עדכון משתנה הBalance של כל קודקוד ע"י שימוש בפונקציה height:

$$x.\text{balance} = \text{height}(n.\text{right}) - \text{height}(n.\text{left})$$

קריאות לפונקציה Set Balance:

א. בסוף סיבוב ימינה/שמאלה סביב קודקוד x נקרא לפונקציה set balance עם קודקוד x והאבא החדש שלו.
ב. במימוש הפונקציה הרקורסיבית rebalance עבור קודקוד x הפונקציה rebalance קוראת לset balance ואז מבצעת "סיבובים" ע"י פעולות האיזון (ראה טבלה למטה) בהתאם לbalance העדכני. לאחר מכן, במידה והאב של x אינו null הפונקציה rebalance תקרא לעצמה שוב עם האבא של x עד לשורש.

פעולות האיזון:

<table>
<tr><td colspan="4">מקרה 1: גובה של התת-עץ הימני של קדקוד p גדול ב-2 מגובה של תת-עץ השמאלי.
מקרה זה מתפצל לשני מקרים התלויים בq (הבן הימני של p):</td></tr>
<tr><td><u>מקרה 1.א:</u> (סיבוב אחד)
<u>הבן הימני של q גדול מהבן השמאל של q</u>

$\text{if}\left(\text{height}(\text{p.right.left}) \le \text{height}(\text{p.right.right})\right)\{$
$\quad \text{rotateLeft}(\text{p})$
$\}$</td><td>P
/ \
A Q
/ \
B S
/ \
C D</td><td>←</td><td>Q
/ \
P S
/ \ / \
A B C D</td></tr>
<tr><td><u>מקרה 1.ב:</u> (שני סיבובים)
<u>הבן השמאלי של q גדול מהבן הימני של q</u>

$\text{if}\left(\text{height}(\text{p.right.left}) > \text{height}(\text{p.right.right})\right)\{$
$\quad \text{rotateRight}(\text{p.right})$
$\quad \text{rotateLeft}(\text{p})$
$\}$</td><td>P
/ \
A Q
/ \
S D
/ \
B C</td><td>←</td><td>P
/ \
A S
/ \
B Q
/ \
C D</td></tr>
<tr><td colspan="2">N/A</td><td>←</td><td>S
/ \
P Q
/ \ / \
A B C D</td></tr>
<tr><td colspan="4">מקרה 2: גובה של התת-עץ השמאלי של קדקוד p גדול ב-2 מגובה של התת-עץ הימני (דומה למקרה קודם).</td></tr>
</table>

<u>חיפוש איבר</u>: באופן דומה כמו בעץ חיפוש בינארי.
<u>הוספת איבר</u>: מבוצע באופן זהה כמו בעץ חיפוש בינארי ההבדל היחיד הוא שנפעיל פונקציית איזון (ע"י סיבובים) מהאב של האיבר החדש באופן רקורסיבי עד לשורש. (קוראים לפונקציית האיזון עם הNode של האב).
(ביצוע ההוספה ע"י פוינטרים (nodes) על אב ובן עד למטה שהבן הוא null ואז מוסיפים לאב את האיבר החדש).
<u>מחיקת איבר</u>: מאתחלים 3 פוינטרים (nodes) סמוכים על סבא על בן ועל נכד. בנוסף מאתחלים node נוסף בשם delNode שיצביע על null וכאשר נמצא את האיבר אותו נרצה למחוק, delNode יצביע עליו. נרוץ בצעדים עד למטה (כל עוד הנכד אינו null) כאשר אם קודקוד מסוים קטן או שווה לערך אותו אנו רוצים למחוק אזי נלך צעד ימינה. אחרת שמאלה. בסוף כשנגיע למטה במידה והdelNode אינו null זה אומר שמצאנו את האיבר אותו רוצים למחוק. (בודקים אם delNode אינו null ואם כן אז נבצע את כל הפעולות המופיעות מתחת. אחרת, סיימנו).
* נשים לב כי באופן וודאי נגיע לאיבר הקטן ביותר מימין לקודקוד אותו אנו רוצים למחוק (לפי שיטת מחיקה א').
כעת, נבצע את הפעולות הבאות:

א. את ערכו (המשתנה עצמו בnode) של הקודקוד נחליף בערכו של הבן.
ב. במידה וקיים לבן בן ימני אזי נפנה את הנכד (שכאמור מצביע על null) להצביע על בנו הימני של הבן. אחרת, הנכד יצביע על בנו השמאלי של הבן (בנו השמאלי הוא null).

כעת נבצע בדיקת if ואם ערכו של השורש שווה לאיבר אותו נרצה למחוק אזי נשווה את השורש לנכד.
אחרת, נבדוק עם if פנימי נוסף אם בנו של הסבא הוא באמת הבן אזי נפנה את בנו שמאלי של הסבא להיות הנכד (הנכד הופך להיות בן שמאלי של הסבא). אחרת, נפנה את בנו הימני של הסב להיות הנכד.
(הסיבה לכך היא בגלל שאנו מתחשבים במקרה בו לבן הימני של הdelNode אין בן שמאלי).
נקרא לפונקציית rebalance עם קודקוד הסבא.

עץ אדום-שחור

במבנה הנתונים עץ אדום שחור לכל צומת מוענק שדה חדש המכונה "צבע" (color) בעל הערך אדום (red) או שחור (black).

בנוסף לדרישות הרגילות של עצי חיפוש בינאריים, עץ אדום שחור מקיים גם את הדרישות הבאות:

1. צומת הוא שחור או אדום (אחד מן השניים).
2. השורש הוא שחור.
3. כל העלים שחורים.
4. שני ילדיו של צומת אדום הם שחורים.
5. כל מסלול פשוט מצומת מסוים לכל אחד מהצאצאים העלים שלו מכיל אותו מספר של צמתים שחורים .

הגובה השחור של צומת העץ:

מספר צמתים שחורים מקודקוד מסוים לכל אחד מהצאצאים העלים שלו מכונה "הגובה השחור" של הצומת.
הצמתים הפנימיים של העץ הם קודקודים שאינם NIL.

1. אורכו של המסלול הארוך ביותר מהשורש לכל אחד מן העלים הוא לכל היותר פעמיים אורכו של המסלול הקצר ביותר מהשורש לאחד העלים.

הוכחה: נגדיר X צמתים שחורים וכעת נוסיף צמתים אדומים בין לבין ולכן ע"פ תכונה 4 נוכל להוסיף בין שני צמתים שחורים רק צומת אחד אדור ולכן נוכל להוסיף לכל היותר עד X צמתים אדומים. ונקבל 2x צמתים.

2. תת-עץ של עץ אדום-שחור בעל n קודקודים פנימיים, ששורשו הוא x מכיל לפחות $2^b - 1$ קודקודים פנימיים, כאשר b הוא הגובה השחור של התת-עץ, כלומר $n \geq 2^b - 1$.

<u>הוכחה</u> באינדוקציה לפי **b**.

<u>בסיס</u>: $b = 0$, כלומר בתת-עץ יש רק עלה אחד שהוא NIL, אז מספר צמתים פנימיים בתת-עץ זה שווה:
$n = 2^b - 1 = 2^0 - 1 = 0$.

ניקח $b = 1$, השורש הוא שחור והוא הצומת היחיד שהוא שחור. כאן יש שלוש אפשרויות:

א) שורש הוא צומת יחיד בעץ $\underline{n = 1}$. **ב)** לשורש יש בן אדום אחד $\underline{n = 2}$. **ג)** לשורש יש שני בנים אדומים $\underline{n = 3}$.

בשלושה המקרים מתקיים: $n \geq 2^b - 1 = 2^1 - 1 = 1$.

<u>הנחת אינדוקציה</u>: מספר צמתים פנימיים של העץ בעל גובה שחור $b > 0$ מקיים את האי-שוויון: $n \geq 2^b - 1$.

<u>הוכחה</u> עבור גובה שחור של $b + 1$.

א) שורש של התת-עץ x הוא <u>שחור</u> ונניח שיש לו שני בנים. אז הגובה השחור של כל אחד מהבנים שווה ל-**b** בגלל שהגובה של כל אחד מהבנים הוא קטן ב-1 מגובה של **x** עצמו, כלומר שווה ל-**b**. לפי הנחת האינדוקציה מספר צמתים פנימיים של כל בן הוא לפחות $2^b - 1$. לכן העץ המקורי מכיל לפחות:

$$n \geq 2^b - 1 \ + \ 2^b - \ 1 + 1 \ = \ 2^{b+1} - 1$$

ב) שורש של התת-עץ x הוא <u>אדום</u>. לפי תכונה 4 יש לו שני בנים שחורים שהגובה השחור של כל אחד מהם הוא $b + 1$. לפי סעיף א) מספר קדקודים לכל אחד מהבנים לא עולה על $2^{b+1} - 1$.
לכן, עבור מספר הקודקודים של תת-עץ ששורשו הוא **x** אדום מתקיים האי-שוויון:

$$n \geq 2^{b+1} - 1 + 2^{b+1} - 1 + 1 = 2 \cdot 2^{b+1} - 1 > 2^{b+1} - 1.$$

ולכן:

$$b \leq \log(n+1) - 1$$

3. גובה h של עץ אדום-שחור בעל n קודקודים פנימיים לא עולה על $2 \cdot \log(n+1)$, כלומר:

$$\mathbf{h \leq 2 \cdot \log(n+1)}$$

<u>הוכחה</u>: הגובה השחור של העץ הוא לפחות $h/2$. כלומר, $b \geq h/2$. לכן, $n \geq 2^b - 1 \geq 2^{h/2} - 1$. ונקבל:
$h \leq 2 \cdot \log_2(n+1)$.

הוספת איבר חדש בעץ אדום-שחור: (ע"י שימוש ב"סיבובים" ו"צביעה מחדש" של הקודקודים בהתאם).
נשים לב כי תמיד לצומת ברמה האחרונה יהיו עלים שחורים שלא מכילים מידע.
אלגוריתם של הוספת איבר חדש: באלגוריתם יש שני מקרים עקרים הקשורים לצבע של "דוד" של איבר חדש.
נסמן את האיבר החדש ב-x. שלבי ההוספה הם:

1) מבצעים הוספת איבר סטנדרטית של עץ חיפוש בינארי וצובעים את x באדום.
2) אם העץ היה ריק אז x הוא שורש העץ, משנים את הצבע שלו לשחור. גובה העץ עולה ב-1, כלומר הופך ל-0.
(נזכור שגובה של עץ ריק הוא 1-, בגלל שגובה של קדקוד כלשהו NIL שווה 1-).
3) קדקוד האב של x הוא שחור. במקרה זה כל התנאים מתקיימים, אין מה לתקן.
4) קדקוד האב של x אדום ו- x לא שורש. (הסב של x חייב להיות שחור ע"פ תכונה 4).
א) <u>הדוד של x הוא אדום</u>**.**
i) משנים את הצבע של קדקוד האב וקודקוד הדוד לשחור.
ii) צובעים את קדקוד הסב לאדום.
iii) חוזרים רקורסיבית לשלבי הבדיקה (שלב 4) עם הסב של x במקום x. (בסוף נוודא שהשורש שחור).
ב) <u>הדוד של x הוא שחור (או שאין לו דוד)</u>. (המקרים הבאים דומים למקרים בעץ AVL).
במקרה זה יש 4 מקרים המתייחסים לתצורות שונות של x, האב של x הוא p והסב של x הוא g.
i) Left-Left Case (p הוא הבן השמאלי של g ו- x הוא הבן השמאלי של p).
ii) Left-Right Case (p הוא הבן השמאלי של g ו- x הוא הבן הימני של p).
iii) Right-Right Case (p הוא הבן הימני של g ו- x הוא הבן הימני של p).
iv) Right- Left Case (p הוא הבן הימני של g ו- x הוא הבן השמאלי של p).

מימוש שלב 4) ב):

<u>מקרה 1:</u> <u>האיבר החדש נמצא בתת עץ שמאלי של הסבא שלו.</u>
מקרה זה מתפצל לשני מקרים התלויים במיקום האיבר החדש X לעומת אביו:
נגדיר: **X** - קדקוד חדש. **P** - קדקוד האב של X. **U** - קדקוד הדוד (השחור כאמור) של X.

<u>Left-Left Case</u>: (סיבוב אחד)
<u>(p הוא הבן השמאלי של g ו- x הוא הבן השמאלי של p)</u>
פעולות:
* סיבוב ימני סביב הסבא (g).
* החלפת צבעים בין הסבא לאבא (בין g ל-p).

```
        G                        P
       / \                      / \
      P   U                    X   G
     / \  / \        ←        / \  / \
    X  T3 T4 T5              T1 T2 T3 U
   / \                                / \
  T1  T2                             T4  T5
```

<u>Left-Right Case</u>: (שני סיבובים)
<u>(p הוא הבן השמאלי של g ו- x הוא הבן הימני של p)</u>
פעולות:
* סיבוב שמאלי סביב האבא (p)
[קבלנו מקרה דומה למקרה קודם כאשר x אבא של p].
* סיבוב ימני סביב הסבא (g),
* החלפת צבעים בין הסבא לאבא (בין g ל-x).

```
        G                        G
       / \                      / \
      P   U                    X   U
     / \  / \        ←        / \  / \
    T1  X T4 T5              P  T3 T4 T5
       / \                  / \
      T2  T3               T1  T2

                                 X
                                / \
                               P   G
                     ←        / \  / \
                             T1 T2 T3 U
                                      / \
                                     T4  T5
```

N/A

מקרה 2: <u>האיבר החדש נמצא בתת עץ שמאלי של הסבא שלו</u>. (דומה למקרה קודם).

<u>ערימה בינארית</u>

עץ בינארי כמעט שלם (nearly complete binary tree) - הוא עץ שבו צריכים להתקיים שלושה תנאים:

1. לכל הצמתים של גובה העץ פחות 2 יש בדיוק שני בנים.
2. אם לצומת **p** ברמת גובה העץ פחות 1 יש בן שמאלי אז לכל צומת שמשמאל יש שני בנים.
3. אם לצומת **p** ברמה גובה העץ פחות 1 יש בן ימני אז גם יש לו בן שמאלי.

ערמה (**min-heap**) היא עץ בינארי כמעט שלם, שבו כל צומת קטן משני בניו. אין חשיבות לסדר הבנים.
שיטה זאת נקראת <u>min-heap property</u>, דהיינו שכל צומת יהיה קטן משני בניו.
<u>הפונקציה minHeapify</u>: $\text{minHeapify}(a[], v, \text{heapSize})$ שומרת על תקינות הערמה.
הארגומנטים הם v - אינדקס של קודקוד ו- heapSize הוא גודל הערמה. <u>תנאי לקריאת הפונקציה: v גדול מבניו</u>.
הפונקציה תחליף את v עם הבן הקטן יותר מכיוון שהוא יהפוך לקודקוד החדש שבהכרח גדול מבניו.
מכאן, הפונקציה תמשיך באופן רקורסיבי עד להגעה לתחתית הערימה.
מקובל לממש ערמה באמצעות מערך - זאת מכיוון שאין מחשיבות לסדר הבנים (עץ כמעט שלם), ולכן גם איברי המערך יתמלאו באופן סדרתי – ללא מקומות ריקים.
במימוש באמצעות מערך, נשתמש בכלל הבא:
כאשר אינדקס של קדקוד האב הוא p:
<u>אינדקס של הבן השמאלי (של p) הוא</u>:

$$\text{left}(p) = 2 * p + 1$$

<u>אינדקס של הבן הימני (של p) הוא</u>:

$$\text{right}(p) = 2 * p + 2$$

<u>אינדקס של קודקוד האב (p)</u> (כאשר אינדקס של אחד מהבנים [לא משנה שמאלי או ימני] הוא i):

$$p = \lfloor (i-1)/2 \rfloor$$

מהו מספר עלים בעץ ערמה?
בעץ ערמה קדקוד אחרון הוא עלה והאינדקס שלו הוא $n-1$ לכן האינדקס של קודקוד האב שלו הוא:

$$p = \lfloor (n-2)/2 \rfloor = \lfloor n/2 \rfloor - 1$$

וזה האינדקס של הקודקוד האחרון שיש לו בנים. לכן מספר קודקודים שיש להם בנים הוא $\lfloor n/2 \rfloor$.
שאר הקודקודים הם עלים, ומספר עלים הוא $n - \lfloor n/2 \rfloor$.
בדוגמה זו $n = 10$, מספר עלים הוא 510=⌊10/2⌋.

<u>מיון מערך בעזרת עץ ערימה:</u> (ע"י השימוש בכך שהשורש הוא האיבר הכי קטן באופן חד משמעי)
בונים מאיברי המערך עץ ערמה על-ידי קריאה לפונקציה buildMinHeap(a[]).
עוברים בלולאה על כל איברי הערמה.
בכל איטרציה:

1) האיבר מסוף הערמה a[heapSize-1] עובר לראש הערמה: הוא מתחלף עם האיבר המינימאלי a[0] שנמצא בראש הערמה. ובכך דחפנו את האיבר הקטן ביותר (שהיה בשורש) לסוף הטווח הנתון.
2) בעזרת מתודה minHeapify(a,0, heapSize) מעבירים את השורש החדש ממקום 0 למקום הנכון שלו.
3) מקטינים את heapSize ב-1: כיוון שהאבר המינימאלי ביותר הועבר לסוף הוא במקום הנכון ולכן נתעלם.

<u>סיבוכיות של האלגוריתם $O(n\log_2 n)$</u>.

יצירת עץ ערימה ממערך: נלך לאינדקס של הקודקוד האחרון לו יש בן (ע"י הנוסחה [size/2]-1) ואז נפעיל עליו את פונקציית minHeapify ולאחר מכן נעבור לאינדקס הקטן ממנו באחד עד שנגיע לשורש (אינדקס שווה ל-0).
במהלך הזה אנו עוברים על כל האבות מימין לשמאל ולמעלה מכיוון שהם מסודרים היטב לפי הסדר במערך.

סיבוכיות (של יצירת עץ ערימה ממערך):
כאמור, מריצים את minHeapify על הרמה אחת לפני אחרונה עבור האב האחרון לו יש בן.
נניח שהעץ שלם. בעומק $h-1$ יש 2^{h-1} צמתים וכל אחד צריך לרדת פעם אחת בלבד.
בצורה כללית בעומק $h-j$ יש 2^{h-j} צמתים וכל אחד צריך לרדת j פעמים.
לכן, כאשר עושים חישוב של זמן ריצה $T(n)$ רמה-אחרי-רמה
מרמה התחתונה כלפי מעלה מקבלים:

$$T(n) = \sum_{j=0}^{h} j * 2^{h-j} = \sum_{j=0}^{h} j * \frac{2^h}{2^j} = 2^h * \sum_{j=0}^{h} \frac{j}{2^j}$$

כדי לחשב את הסכום שקבלנו נכתוב נוסחה לסכום של סדרה
הנדסית אינסופית כאשר $0 < x < 1$:

$$\sum_{j=0}^{\infty} x^j = \frac{1}{1-x}$$

נגזור את שני האגפים של המשוואה האחרונה:

$$\sum_{j=0}^{\infty} j * x^{j-1} = \frac{1}{(1-x)^2}$$

ונכפיל את שני האגפים ב-x:

$$\sum_{j=0}^{\infty} j * x^j = \frac{x}{(1-x)^2}$$

השוויון האחרון מתקיים עבור כל $0 < x < 1$, נציב $x = \frac{1}{2}$. מקבלים:

$$\sum_{j=0}^{\infty} \frac{j}{2^j} = \frac{1/2}{(1-1/2)^2}$$

ולבסוף:

$$T(n) = 2^h * \sum_{j=0}^{h} \frac{j}{2^j} \le 2^h * \sum_{j=0}^{\infty} \frac{j}{2^j} \le 2^h * 2 = 2^{h+1} = n+1 \cong O(n)$$

<u>המסקנה</u>: סיבוכיות של אלגוריתם בניית עץ ערמה היא **O(n)**. (ולא $n\log n$ (!)).

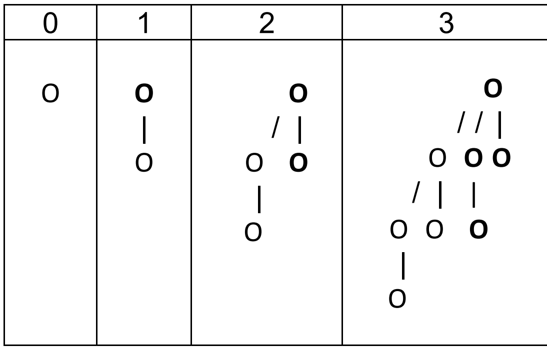
תור עדיפויות: (Priority Queue) או בשם אחר, תור קדימויות. הינו מבנה נתונים המיישם לוגיקה של תור.
אולם, אינו מבוסס כתור רגיל על סדר הכניסה בלבד (First In First Out-FIFO), אלא גם על קוד עדיפות (priority), המוסף לאובייקט המוכנס לתור.
ככל שערך קוד העדיפות של האובייקט גבוה יותר (לפי סדר חלקי כלשהו על קבוצת הערכים המשמשים לסמן את העדיפות), כך יקודם מקומו בתור (מיד עם כניסתו).

<u>מתודת עזר המשמשת בפונקציה להוספת איבר חדש</u>: **heapDecreaseKey(a[], I ,key)** [סיבוכיות $O(\log_2 n)$].
כאשר i הוא אינדקס האיבר אותו רוצים להקטין ו-key הוא ערכו החדש. ראשית המתודה מעדכנת את ערכו החדש לאחר מכן, המתודה מעבירה את הערך החדש כלפי מעלה למקום הנכון. כלומר, לשמירת min-heap property של הערימה היא משווה את איבר החדש עם קדקוד האב שלו וכך הלאה עד השורש.
הוספת איבר חדש לעץ ערמה:
1. מוסיפים לסוף המערך איבר שערכו הוא ∞ (אינסוף).
2. מקטינים את האינסוף. הערך החדש שהאיבר האחרון מקבל במקום ∞ הוא האיבר החדש.
ההקטנה נעשתה ע"י מתודה heapDecreaseKey.

* מחיקת מינימום: החלפת מינימום עם האיבר האחרון. קריאה ל-minHeapify עם השורש. הקטנת size = size-1.

ערימה בינומית

עץ בינומי (binomial tree):
עץ המוגדר רקורסיבית לכל k.
B_0 - עץ שמורכב מקודקוד אחד, שהוא שורש העץ.
B_k – עץ שמורכב משני עצי B_{k-1},
כך ששורש של עץ אחד הוא הבן השמאלי של העץ השני.



גובה של קדקוד הוא מרחק (מספר צלעות) מקודקוד עד העלה הרחוק ביותר.
עומק (רמה) של קדקוד הוא מרחק (מספר צלעות) מקודקוד עד שורש העץ.

1. לעץ בינומי B_k יש 2^k קודקודים (כאשר k הוא מספר השכפולים כמו בטבלה).

<u>הוכחה באינדוקציה.</u>

בסיס האינדוקציה: עבור $k = 0$ הטענה נכונה: $2^0 = 1$.
הנחת האינדוקציה: הטענה נכונה עבור k כלשהו.
צעד האינדוקציה: נוכיח את הטענה עבור $k + 1$. עץ B_{k+1} מורכב משני עצים Bk לפי ההגדרה.
לכך יש לו $2^k + 2^k = 2^{k+1}$ קודקודים.

<u>מסקנה</u>: לעץ בינומי בעל n קדקודים יש $\log_2 n$ רמות: $numLevels(B_k) = k = \log_2 n$.

2. גובה (height) של עץ בינומי B_k הוא k.

<u>הוכחה באינדוקציה.</u>

בסיס: עבור $k = 0$ הטענה נכונה: $height = 0$.
הנחת האינדוקציה: הטענה נכונה עבור k כלשהו.
צעד האינדוקציה: נוכיח את הטענה עבור k+1. B_{k+1} מורכב משני עצים B_k, כך ששורש של עץ אחד הוא הבן השמאלי של העץ השני, לכן גובה של B_{k+1} גדול ב-1 מגובה של B_k ושווה $k + 1$.

3. דרגה של שורש של עץ בינומי B_k היא k. דרגות של שאר הקודקודים קטנה ממש מ- k.

<u>הוכחה באינדוקציה.</u>

בסיס: בעץ B_0 דרגה של השורש היא 0.
הנחת האינדוקציה: הטענה נכונה עבור k כלשהו.
צעד האינדוקציה: נוכיח את הטענה עבור k+1. דרגה של שורש של עץ B_{k+1} שווה לדרגה של B_k ועוד 1.
לפי הנחת האינדוקציה דרגה של B_k היא k לכן הדרגה של B_{k+1} היא $k + 1$.
בגלל שבמעבר מ- B_k ל- B_{k+1} רק דרגה (מספר צלעות היוצאות מקודקוד) של השורש גדלה, אזי הדרגות של שאר הקודקודים של העץ הבינומי קטנה מדרגה של השורש, כלומר קטנה ממש מ-$k + 1$.

4. מסקנה - מהתכונות 1, 3 נובע כי בעץ בינומי בעל n קדקודים הדרגה הגבוהה ביותר היא שווה למספר רמות של העץ ומתקיים: $maxLevel(B_k) = k = \log_2 n$.

5. מספר הקודקודים של עץ בינומי B_k הנמצאים ברמה i הוא $\binom{k}{i}$. (מכאן מגיע השם עץ בינומי).

<u>הוכחה באינדוקציה.</u>

בסיס: עבור $k = 1$ הטענה נכונה: $\binom{1}{i} = 1$ $(i = 0,1)$, .
הנחת אינדוקציה: הטענה נכונה עבור k.
צעד האינדוקציה: נוכיח את הטענה עבור $k + 1$. B_{k+1} מורכב משני עצים B_k, כך ששורש של עץ אחד הוא הבן השמאלי של העץ השני. ברמה i לעץ B_{k+1} יש $\binom{k+1}{i} = \binom{k}{i} + \binom{k}{i-1}$, בגלל שלעץ B_{k+1} הצטרפו קודקודים הנמצאים ברמה $i - 1$ של עץ B_k. ולכן ע"פ הנוסחה הוכחנו את המבוקש.

ערימה בינומית:

ערמה בינומית היא אוסף של עצים בינומיים, המקיימים את התכונות הבאות:

1. כל עץ בינומי מקיים את minimum-heap property, כלומר מפתח של כל קדקוד גדול או שווה למפתח של קדקוד האב שלו. מכן ניתן להשיג כי לכל עץ בינומי שורש העץ הוא האיבר המינימאלי.
2. גבהים של כל תתי-עץ של ערמה בינומית שונים. במילים אחרות בערמה בינומית יכול להיות לכל היותר עץ בינומי אחד בגובה k כלשהו.

```
10  ----------------> 9 -----------------> 5  -----------------> 1
                      |                  / |                    / /|
                     12                17  21                 23  12 77
                                        |                    / |    |
                                       99                  33 24   22
                                                            |
                                                           53
```

תכונות של ערמה בינומית:

מספר עצים בינומיים בערמה בינומית בעלת n קודקודים מכילה לכל היותר $\lfloor \log_2 n \rfloor + 1$ עצים בינומיים.

הוכחה: נזכיר כי ערמה בינומית מורכבת מעצים בינומיים שונים וכל עץ בינומי מכיל 2^k קדקודים, כאשר k הוא גובה העץ. וייצוג של מספר n בצורה בינארית מכיל בדיוק $p = 1 + \lfloor \log_2 n \rfloor$ ספרות בינריות.

כלומר, ניתן לייצג n בצורה הבאה: $n = k_0 \cdot 2^0 + k_1 \cdot 2^1 + \cdots + k_p \cdot 2^p$, כאשר $k_0 \in \{0,\ 1\}$.

בגלל שייצוג מספר בצורה בינרית הוא חד-חד ערכי מספר עצים בינומיים בערמה בינומית בעלת n קדקודים יכולה להכיל מקסימום.

דוגמה $n = 13$, $13 = 1 + 4 + 8 = 2^0 + 2^2 + 2^3$,

כלומר, ערמה בינומית בעלת 13 קודקודים מורכבת משלושה עצים בינומיים: $B_0 + B_2 + B_3$.

מבנה של קודקוד:

1. key - מפתח הקודקוד.
2. parent - קודקוד האב של הקודקוד. root.parent=null.
3. child - הבן השמאלי של הקודקוד. כאשר לקודקוד x אין בנים, x.child=null.
4. sibling - האח הימני של הקודקוד. כאשר x הוא הבן הימני של קודקוד האב שלו x.sibling=null.
5. degree - מספר הילדים של הקודקוד.

שורשים של עצים בינומיים שערמה הבינומית מכילה אותם נמצאים **ברשימת השורשים.**
בערמה בינומית רשימת השורשים ממוינת מקטן לגדול לפי דרגות השורשים.

מבנה של ערמה בינומית:

מחלקת BinomialHeap מכילה משתנה עצם אחד בלבד והוא ראש של רשימת השורשים head.

הפעולות הנתמכות בערימה בינומית:

1. בניית ערמה בינומית חדשה (create new binomial heap).
2. מציאת איבר מינימאלי (find minimum).
3. איחוד שתי ערמות בינומיות (union).
4. הכנסת איבר חדש (insert).
5. מחיקת איבר מינימאלי (delete minimum).
6. הקטנת מפתח (decrease key).
7. מחיקת איבר (delete).

1. **בניית ערמה בינומית חדשה**. [סיבוכיות O(1)]
מכילה פעולה אחת: head = null
2. **מציאת איבר מינימאלי**. [סיבוכיות $O(\log_2 n)$]
למציאת איבר מינימאלי אנו צריכים לעבור על רשימת השורשים ולמצוא בה איבר מינימאלי. מכוון שמספר עצים בינומיים לכל היותר $\log_2 n + 1$ סיבוכיות הפעולה $O(\log_2 n)$.
3. **איחוד שני עצים בינומיים בעלי אותה דרגה (מקרה פרטי) הנמצאים באותה ערמה.** [סיבוכיות O(1)]
שורש של העץ המאוחד הוא הקטן ביותר בין שני השורשים (שורש העץ הוא האיבר הקטן בתוך העץ).
לכן עץ בעל שורש גדול יותר הופך להיות עץ משני.
מקבלים 2 nodes של השורשים שלהם ומקשרים את השורש הגדול מביניהם להיות בנו של השורש הקטן
כעת, עושים השמה למשתני הparent node, sibling, child, degree.
* פעולה זו היא הפעולה הבסיסית בפעולת איחוד של שתי ערימות בינומיות.
4. **הוספת איבר חדש x לערמה בינומית**: (ראה הסבר על פעולת האיחוד למטה). [סיבוכיות $O(\log_2 n)$].
בונים ערמה חדשה שמכילה איבר אחד x, לאחר מכן מאחדים את הערימה הנוכחית עם נערימה חדשה.

פעולת האיחוד של שתי ערמות בינומיות היא המעניינת ביותר בין כל הפעולות. [סיבוכיות $O(\log_2 n)$].
הפעולה הראשונה היא מיזוג של שתי ערמות בינומיות לפי דרגות העצים הבינומיים שלהם.
אחרי המיזוג אנו צריכים לקבל ערמה בינומית המקיימת את כל התכונות של ערמה בינומית.
אם בעץ הממוזג יש שני עצים באותה דרגה, אז שני האלה ימוזגו שוב.
אם בעץ הממוזג יש שנים או שלושה עצים בעלי אותה דרגה ממזגים את השני העצים הימניים.
* בכל אחד מהערמות שאנו מאחדים אין שני עצים בעלי אותו גובה. לכן אחרי המיזוג נקבל לכל היותר שתי ערמות בעלי אותה דרגה. לפיכך, בשלבי אחוד הערמות נקבל לכל היותר שלושה עצים בעלי אותה דרגה.

סיבוכיות פעולת המיזוג: $\log_2(n_1 + n_2)$, כאשר n_1 הוא מספר הקודקודים בערימה H_1, ו- n_2 הוא מספר הקודקודים בערימה H_2.
סיבוכיות של פעולת האיחוד היא גם $O(\log_2 n)$.
הוכחה לשני המקרים (מיזוג ואיחוד): יהי n_1 מס' קודקודים של H_1, n_2 מס' קודקודים של H_2, $n = \mathbf{n_1} + \mathbf{n_2}$.
הערמה המאוחדת מכילה $\mathbf{n}$ איברים ובמקרה הגרוע $\log_2 n$ שורשים.
האיחוד מתבצע על השורשים בלבד, לכן הסיבוכיות של האחוד היא $O(\log_2 n)$.

הוצאת איבר מינימלי: binomialHeapExtractMin(H) [סיבוכיות $O(\log_2 n)$]
1. מחפשים את השורש **x** הקטן ביותר.
2. מנתקים את תת-ערמה שמכילה את **x** מערמה מקורית. מקבלים שתי ערמות.
3. מוחקים את x מתת-הערמה של עצמו. בונים ערמה זמנית שבה הבנים של **x** הופכים לשורשים שלה.
4. ממניינים את סדר השורשים בעץ החדש מדרגה גדולה לקטנה, כי העצים שמרכיבים את הערמה החדשה צריכים להיות ממוינים לפי הדרגה של השורשים. (בעץ בינומי תת-עצים שהם הבנים מסודרים בסדר עולה מימין לשמאל הפוך ממה שצריך). מאחדים את הערמה שניתקנו עם הערמה הזמנית ומקבלים ערמה שלא מכילה את **x.**

הפחתת ערך מפתח: [סיבוכיות $O(\log_2 n)$]
הפחתת מפתח מתבצעת באותה צורה בדיוק כמו בעץ ערמה.
סיבוכיות של הפעולה היא $O(\log_2 n)$, מכוון שהעומק המקסימאלי של קודקוד בעץ בינומי הוא $\log_2 m$ כאשר m הוא מספר הקודקודים בעץ. אבל מספר הקודקודים בערמה הוא גדול (או שווה) ממספר קודקודי העץ שהעץ הוא רק חלק מערמה: $m \le n$.
מחיקת קודקוד x מערמה בינומית: [סיבוכיות $O(\log_2 n)$]
1. מפחיתים את המפתח שלו ל ∞-, הקודקוד שלו יהפוך למינימאלי ויעלה לשורש העץ אליו הוא שייך.
2. מוחקים את האיבר המינימאלי ∞- מהערמה.

איחוד קבוצות זרות

איחוד קבוצות זרות (**Disjoint-Set Data Structure**) הוא מבנה נתונים אשר מבצע מעקב אחרי קבוצה של אובייקטים המחולקים למספר של תתי-קבוצות זרות ולא חופפות. הפעולה העיקרית של מבנה זה היא איחוד של שתי קבוצות זרות. (משמש גם לאחסון רכיבי קשירות בגרף).

פעולות המוגדרות על מבנה זה הן:

יצירה (makeSet): פעולה היוצרת קבוצה חדשה המכילה אובייקט אחד בלבד (סינגלטון).

חיפוש *(*find*)*: קביעה איזו קבוצה מכילה אובייקט ספציפי. פעולה זו יכולה גם לעזור בקביעה האם שני אובייקטים שייכים לאותה הקבוצה.

איחוד *(*union*)*: איחוד שתי קבוצות לקבוצה אחת.

מימוש המבנה (איחוד קבוצות זרות) ע"י עצים:

כל קבוצה מיוצגת על ידי עץ, שורש העץ נחשב כנציג של הקבוצה. כל קדקוד מחזיק את המצביע לקודקוד האב שלו (parent node), שורש (root) העץ נחשב כקודקוד האב של עצמו. העצים שמיצגים את הקבוצות מהווים יער.

פעולת makeSet - יוצרת עץ שמכיל קודקוד אחד בלבד.

פעולת find - הולכת מקודקוד מסוים לשורש העץ לפי מסלול של קודקודי האב.

פעולת union - קושרת 2 שורשים של 2 עצים.

מימוש חיפוש (find): קריאה לקודקוד האב. תנאי העצירה הוא כשנגיע לקודקוד שמצביע על עצמו (שורש).

מימוש איחוד העצים (union):

מימוש נאיבי: נריץ על כל עץ פונקציית find ואז נאחד את השורשים. (בודקים חריגה שזה לא אותו שורש).

היעילות של מימוש זה נמוכה, סיבוכיות של כל קריאה של find במקרה הגרוע היא $O(n)$, ורצף של n פעולות find יתבצע בסיבוכיות של $O(2n) = O(n)$.

מימוש לפי שיטת Union by Weight:

נגדיר משקל העץ כמספר הקודקודים שלו. בשיטה איחוד לפי משקל, כלומר העץ עם מספר קטן יותר של קודקודים מצורף לעץ בעל מספר קודקודים גדול יותר. השיטה דומה לשיטה קודמת.

סיבוכיות שיטת Union by Weight:

טענה: יהי n משקל העץ (מס' הקודקודים), אזי הגובה h שלו לא עולה על $\log_2 n$. כלומר, $h \le \log_2 n$.

הוכחה באינדוקציה על n.

בסיס האינדוקציה: עבור $n = 1$, (יש עץ אחד בלבד) הטענה נכונה: $0 = h \le \log_2 1 = 0$.

כאשר יש שני עצים בגובה 0, $n = 2$, גובה של העץ המאוחד הוא לכל היותר אחד $h = 1 = \log_2 2 = 1$.

הנחת האינדוקציה: הטענה נכונה עבור n כלשהו.

צעד האינדוקציה: בשלב זה מאחדים שני עצים בגובה $n_1 + n_2 = n$. $n_1 < n \ \&\& \ n_2 < n$.

לפי הנחת אינדוקציה $h_1 \le \log_2 n_1$ ו- $h_2 \le \log_2 n_2$

ללא הגבלת כלליות ניתן להניח כי $n_1 \ge n_2$. לאחר האיחוד משקל עץ

יהפוך ל- $n = n_1 + n_2$ וגובה העץ יהיה $h = \max(\log_2 n_2 + 1, \log_2 n_1)$

כדי לסיים את ההוכחה צריך להראות כי $h \le \log_2(n_1 + n_2)$. אמנם:

$$h = \max(\log_2 n_2 + 1, \log_2 n_1) = \max(\log_2 2n_2, \log_2 n_1) \le \max(\log_2 n, \log_2 n) = \log_2 n$$

האי-שיוון האחרון מתקיים כי $n_1 \le n_1 + n_2 = n$ וגם $2n_2 = n_2 + n_2 \le n_1 + n_2 = n$.

באיחוד של שני עצים משתמשים בשיטת איחוד לפי הדרגה (union by rank) שיטה יעילה ביותר.

שיטה זו מאפשרת לעשות עץ מאוזן יותר. הגישה תהיה להפוך את השורש של העץ עם פחות קודקודים להיות בן תחת שורש העץ עם הכמות הגדולה יותר של הקודקודים. עבור כל קדקוד שומרים על דרגה (rank).

ה-rank מהווה חסם עליון על הגובה (height) של הקודקוד.

למרות שבתחילת התהליך rank הקודקוד שווה לגובה שלו, בהמשך התת-עץ יכול להימחק, וחיפוש הגובה החדש היא פעולה די יקרה. לכן לכל שורש רושמים rank שהוא בטוח גדול מגובה השורש.

תכונות union by rank:

טענה 1: כאשר x הוא לא שורש, אזי x.rank < x.parent.rank.

הוכחה: קודקוד בדרגה k נוצר בעזרת מיזוג של שני שורשים בדרגה k-1 בלבד.

טענה 2: כאשר x הוא לא שורש, אזי x.rank לעולם לא ישתנה שוב.

הוכחה: דרגת הקודקוד משתנה לשורשים בלבד, קדקוד שהוא לא שורש לעולם לא יהפוך לשורש.

טענה 3: לכל שורש בדרגה k יש לפחות $n \geq 2^k$ קדקודים בעץ שלו.

הוכחה באינדוקציה.

בסיס האינדוקציה: נכון עבור $k = 0: \ \ 1 = n \geq 2^0 = 1$.

הנחת האינדוקציה: הטענה נכונה עבור $k - 1$.

שלב האינדוקציה: קודקוד בדרגה k נוצר בעזרת מיזוג של שני שורשים בדרגה $k - 1$ בלבד.

לפי הנחת האינדוקציה כל תת-עץ בעל דרגה $k - 1$ מורכב מלפחות 2^{k-1} קודקודים,

לכן לאחר מיזוג לעץ הממוזג יש לפחות $n \geq 2^{k-1} + 2^{k-1} = 2^k$ קודקודים.

טענה 4: הדרגה הגבוהה ביותר של קדקוד היא קטנה או שווה ל- $\log_2 n$.

הוכחה: לפי טענה 3 מתקיים $2^k \leq n$, לכן $k \leq \log_2 n$.

מסקנה 1: סיבוכיות של חיפוש איבר ללא דחיסת מסלול (מבואר בהמשך) היא $O(\log_2 n)$.

מסקנה 2: מכוון שבפונקציית union קוראים פעמיים לפונקציית חיפוש, הסיבוכיות שלה היא גם $O(\log_2 n)$.

מימוש של מבנה נתונים דורש שמירת ה-rank בכל קדקוד בעץ:

כאשר פעולת יצירה **makeSet** יוצרת עץ חדש, נוצר רק שורש העץ שהוא מצביע על עצמו, כלומר, הוא קדקוד האב של עצמו והדרגה שלו שווה אפס. כל הפעולות **makeSet** אינן משנות את הדרגה של הקודקודים.

פעולת **find** גם לא משנה את הדרגה של הקודקודים.

בפעולת **union** יש שני מקרים:

א. דרגות השורשים שונות. במקרה זה שורש העץ בעל דרגה קטנה יותר מצביעה על שורש העץ בעל דרגה גבוהה יותר, אבל הדרגות עצמן אינן משתנות.

ב. דרגות השורשים שוות. במקרה זה אנו בוחרים את העץ בצורה שרירותית והשורש שלו הופך להיות קדקוד האב של שורש העץ השני.
דרגת שורש העץ הראשון עולה באחד.

דחיסת מסלול (path compression):

עוד שיטה שהופכת את פונקציית החיפוש ליעילה ביותר נקראת דחיסת מסלול (path compression).

שיטה זו גורמת לכל קודקוד במסלול החיפוש להפוך לבן של השורש של אותו העץ.

פעולה זו לא משנה את דרגות (rank) הקודקודים.

קלט הפונקציה הוא קודקוד מסוים. הפונקציה היא פונקציה רקורסיבית אשר קוראת לעצמה עם קודקוד האב עד להגעה לשורש ובחזור (אחר תנאי העצירה כאשר הפונקציה הרקורסיבית חוזרת אחורה) הפונקציה מבצעת עבור כל קודקוד x במסלול השמה x.parent = root.

עץ +B

כאשר נפח הנתונים עובר את גודל הזיכרון, הנתונים נשמרים על הדיסק.
• הגישה לדיסק היא איטית מאוד ביחס לזמן הגישה לזיכרון.
• לכן, מספר הגישות לדיסק יקבע את המהירות הכוללת.
המטרה שלנו היא להציע עץ חיפוש שיביא למינימום את מספר הגישות לדיסק.

מספר גישות לדיסקים.
על מנת להקטין את הזמן המושקע להמתנת תנועות הדיסקים הגישה נעשית לא לפריט אחד, אלא למספר פריטים באותו זמן. המידע מחולק למספר דפים (pages), שמופיעים ברצף, של סיביות (bits) באותו גודל ובגישה אחת לדיסק הקריאים\כתיבים דף אחד או מספר דפים. עבור דיסק טיפוסי, דף עשוי להיות בגודל של 2^{11} עד 2^{14} bytes (2^{14} bytes=16KB, 2^{11}bytes = 2KB).

זמני חישוב של המעבד המרכזי.
• עץ +B הוא עץ חיפוש n-ary שלם מאוזן, המיועד לעבוד היטב עם דיסקים או התקנים משניים אחרים. עצי +B דומים לעצי AVL, אבל הם מבצעים פעולות קלט-פלט במהירות גבוהה יותר.
• עץ +B מורכב משורש, צמתים פנימיים ועלים, השורש יכול להיות עלה או צומת עם שני צאצאים או יותר. עץ +B מאפשר פעולות הוספה, מחיקה וחיפוש בצורה יעילה.

הגדרה: עץ +B הוא עץ מושרש מדרגה m - branching factor או גורם הסתעפות.
תכונות עץ +B:
1. כאשר השורש לא עלה יש לו לפחות שני ילדים (מפתח אחד) ועד m ילדים ($\mathrm{m}-1$ מפתחות).
   כאשר השורש הוא עלה הוא מכיל מ-0 עד $\mathrm{m}-1$ מפתחות.
2. כל קדקוד פנימי פרט לשורש מכיל בין $\lceil \mathrm{m}/2\rceil$ ל- m ילדים. (בין $\lceil \mathrm{m}/2\rceil-1$ ל-$\mathrm{m}-1$ מפתחות).
   קדקוד פנימי אינו מאחסן רשומה, אלא מאחסן מפתחות ומצביעים לילדים המשמשים לחיפוש.
3. כל עלה מכיל בין $\lceil (\mathrm{m}-1)/2\rceil$ ל- $\mathrm{m}-1$ מפתחות ורשומות.
   עלים מאחסנים מפתחות ורשומות או מפתחות ומצביעים לרשומות.
4. בכל קודקוד המפתחות ממוינים בסדר עולה: $\mathrm{k}_1 \le \mathrm{k}_2 \le \cdots \le \mathrm{k}_\mathrm{n}$ כאשר n הוא מס' מפתחות בקודקוד.
5. כל העלים נמצאים באותו עומק, כלומר העץ הוא מאוזן.
6. כל העלה מצביע על העלה הנמצא בצד הימני ושמאלי שלא, כלומר העלים מקושרים זה לזה בעזרת רשימה מקושרת דו-כיוונית. זה מאפשר לעבור מהר על כל עלי העץ.

P_1	K_1	P_2	K_2	…	P_{n-1}	K_{n-1}	P_n	K_n	P_{n+1}

מבנה של קדקוד פנימי:
כאשר $\mathrm{K_i}$ הוא מפתח, ו- $\mathrm{P_i}$ הוא מצביע לאחד מהילדים. הקודקוד מכיל $\mathbf{n}$ מפתחות ו-$\mathbf{n+1}$ ילדים.
• הילד הראשון וכל הילדים שלו מכילים מפתחות שערכיהם נמצאים בקטע $(-\infty, \mathrm{K}_1)$.
• לכל ילד שמספרו i $(2 \le \mathrm{i} \le \mathrm{n})$ הילדים שלו מכילים מפתחות שערכיכם נמצאים בקטע $[\mathrm{K}_{\mathrm{i}-1}, \mathrm{K_i})$.
• הילד שמספרו $\mathrm{n}+1$ (אחרון) וכל הילדים שלו מכילים מפתחות שערכיהם נמצאים בקטע $[\mathrm{K_n}, \infty)$.
מבנה של עלה: - העלים מכילים את כל המפתחות.

סוג צומת	סוג ילד	מינימום ילדים	מקסימום ילדים	דוגמא: m=7	דוגמא: m=100
שורש (שהוא עלה)	רשומות	1	$\mathrm{m}-1$	$1-6$	$1-99$
שורש (שאינו עלה)	צמתים פנימיים או עלים	2	m	$2-7$	$2-100$
צומת פנימי	צמתים פנימיים או עלים	$\lceil \mathrm{m}/2\rceil$	m	$4-7$	$50-100$
עלה	רשומות	$\lceil \mathrm{m}/2\rceil$	m	$4-7$	$50-100$

* לפי העולה מן ההגדרה העלים הם הקודקודים האחרונים המחזיקים את הרשומות (רמה לפני אחרונה).
כלומר, הרשומות שנמצאות בתחתית העץ אינם נחשבים כעלים אך הם נחשבים בנים.

הוספת איבר חדש:
א) מוסיפים קדקוד חדש כעלה.
ב) אם לעלה אין דיי מקום יש לפצל את הקודקוד ולהעתיק את הקודקוד האמצעי (מיקום $\lceil \boldsymbol{m/2} \rceil$ מתחילים מ-0) כלפי מעלה - לקודקוד האב שלו כך שקודקוד האב יישאר ממוין לפי המפתחות.
ג) אם לקודקוד האב אין מספיק מקום יש לפצל אותו ולהעתיק את האלמנט האמצעי למעלה לקודקוד האב שלו.
ד) כאשר מפצלים את השורש - יוצרים שורש חדש בעל מפתח אחד ושני בנים (שני מצביעים).
המפתח של מוסיפים אותו לשורש יימחק מהקודקוד שלו.

מחיקת איבר: כל הרשומות נמצאות בתוך עלים, לכן המחיקה מתבצעת רק בעלים.
1. יש למצוא את העלה L שהאיבר נמצא בו.
2. יש למחוק את האיבר מ-L .
3. אם קדקוד העלה לפחות חצי מלא, סיימנו!
4. אחרת לעלה יש פחות מחצי כניסות (מפתחות). ניסיונות לסדר:
א. מוסיפים לקודקוד L את המפתח השמאלי מהאח הימני של L אם הצלחנו - סיימנו.
ב. אחרת יש להזיז את המפתח הימני של האח השמאלי ל-L .
ג. אחרת יש למזג קודקוד L עם אחיו הימני. אחרי המיזוג יש למחוק את המצביע ל-L מקודקוד האב שלו.
ד. כאשר ממזגים את השורש גובה העץ קטן.

גובה של עץ B:
טענה: לעץ B+ בעל $n \ge 1$ מפתחות, גובה h ודרגה $m \ge 2$ מתקיים:
$$h \le 1 + \log_{\lceil \frac{m}{2} \rceil}\left(\frac{n}{2}\right).$$
הוכחה: כיוון ש- $n \ge 1$, שורש של העץ מכיל לפחות מפתח אחד, ולפחות שני בנים שנמצאים בגובה 1.
כל קדקוד בגובה 1 (שהוא בן השורש) מכיל לפחות $\lceil m/2 \rceil$ מפתחות, לכן בגובה 2 יש לפחות $\lceil m/2 \rceil \cdot 2$ מפתחות. בגובה 3 יש לפחות $\lceil m/2 \rceil^2 \cdot 2$ מפתחות. וכן הלאה.
בגובה h יש לפחות $\lceil m/2 \rceil^{h-1} \cdot 2$ מפתחות. בגלל שכל המפתחות נשמרות בעלים, ל - n מפתחות מתקיים אי-שוויון $n \ge \lceil m/2 \rceil^{h-1} \cdot 2$ או $\frac{n}{2} \ge \lceil m/2 \rceil^{h-1}$. אזי $h \le 1 + \log_{\lceil \frac{m}{2} \rceil} \frac{n}{2}$.

במהלך הכנסת איברים של עץ B+ אנו נבדוק אם מספר האלמנטים בצמתים הפנימיים עבר את הדירוג M.
וגם אם מספר העלים עבר את מספר הL המקסימלי.
אם המקסימום נחצה החוקים הם:
1. לתת את האלמנט לשכן השמאלי.
2. אם לא אפשרי נפצל את הצומת שמצד שמאלי $\lceil m/2 \rceil$ (ערך עליון! עבור מספר זוגי יותר אלמנטים יעברו לעלה הימני החדש). חוק זה מתקיים גם עבור קודקודים פנימיים.
* צומת הפנימי מחזיק את כל הערכים המינימליים של כל אחד מהילדים שלו פרט לילד הכי שמאלי.
* אם אנו מעלים רמה כלפי מעלה ונותנים לשורש החדש את הערך השמאלי שמתחתיו אזי נמחק את אותו ערך מהצומת שמתחת לשורש (פרט לרשומות שבתחתית העץ! אנו לא מוחקים ערכים).
* כאשר פתחנו קודקוד חדש מצד ימין ואנו רוצים לגשר בינו לבין הקודקוד השמאלי שלו ע"י אב אזי נמחק את הערך שבניהם כיוון שהאב מחזיק את כל הערכים המינימליים עבור כל בן פרט לבן השמאלי וכרגע יצרנו אב לשני צמתים אז נוודא שהאב לא מחזיק את הערך של הבן השמאלי (שזה בעצם מה שבניהם).

במחיקה צמתים פנימיים מתנהגים באופן זהה לעלים ולכן אם אפשר לקחת ילדים משמאל או מימין אז ניקח.
אם לשורש יש ילד אחד השורש ימחק והילד יהפוך לשורש! אם יש לשורש ילד ונכד אז הילד יהפוך לשורש בנוסף לפתח מינימלי מהנכד. אם מיזגנו צמתים והוספנו ערך ביניים שהוא זהה למפתח קומה מעל - נמחק את המפתח.

<u>כללי ברזל:</u>

1. לכל צומת פנימי פרט (אולי) לשורש יש c בנים כאשר $\lceil m/2 \rceil \le c \le m$
2. מספר הבנים של השורש הוא $1 \le c \le m$
3. לצומת בעל c בנים יש c-1 מפתחות.

פעולות בסיסיות הקשורות ל- B + עץ:

חיפוש בצומת בעץ B +

בצע חיפוש בינארי ברשומות בצומת הנוכחי.
אם נמצא רשומה עם העונה על ערך החיפוש, החזיר את הרשומה.
אם הצומת הנוכחי הוא צומת שהוא עלה והמפתח לא נמצא, דווח על חיפוש לא מוצלח.
אחרת, עקוב אחר הענף המתאים וחזור על התהליך.

הכנסת צומת בעץ B +:

הקצה עלה חדש והעבירו חצי מרכיבי הדליים לדלי החדש.
הכנס את המפתח והכתובת הקטנים ביותר של העלה החדש להורה.
אם ההורה מלא, חלק גם אותו.
הוסף את המפתח האמצעי לצומת האב.
חזור על הפעולה עד שיימצא הורה שאינו צריך להתפצל.
אם השורש מתפצל, צור שורש חדש שיש לו מפתח אחד ושתי מצביעים. (כלומר, הערך שנדחק לשורש החדש יוסר מהצומת המקורי)

מחיקת צומת בעץ B +:

יורדים אל העלה בו קיים המפתח.

1. הסר את המפתח הנדרש וההפניה המשויכת אליו (המפתח) מהצומת.
2. אם לצומת (הרשומה) עדיין יש מספיק מפתחות (הערכים עצמם) והפניות, סיימנו.
3. א. אם לצומת יש פחות מ-$\lceil m/2 \rceil$ מפתחות, אך לאחיו הבכור או הצעיר הבא באותה רמה יש יותר ממה שצריך, הלווה ממנו מפתח. לאחר מכן תקן את ערכי ההפניות ברמה שלמעלה כדי לייצגם בצורה תקנית כי לצמתים אלה יש כעת "נקודת פיצול" שונה. (זה פשוט משנה את ערכי המפתחות ברמות שלעיל, ללא מחיקה או הכנסה).
   ב. אחרת, אם לאחיו הבכור או הצעיר הבא באותה רמה אין מספיק מפתחות בכדי להלוות מהם, מזג את הצומת עם אחד האחים (ימני לפי סדר הפעולות שהזכרנו לעיל).
   במידה וסעיף ב' התרחש כאשר הצומת אינו עלה (כלומר, עם אחד הצמתים הפנימיים), אזי נצטרך לשלב את "המפתח המפצל" מההורה שלו למיזוג שלנו.

בשני המקרים נצטרך לחזור על אלגוריתם ההסרה בצומת האב כדי להסיר את "המפתח המפוצל" שהפריד בעבר בין הצמתים הממוזגים הללו - אלא אם כן ההורה הוא השורש ואנחנו מסירים את המפתח הסופי מהשורש, ובמקרה זה הצומת הממוזג הופך לשורש החדש (והעץ הפך לרמה אחת פחות ממקודם).

Hash Table

טבלת גיבוב או טבלת ערבול (hash table) היא מבנה נתונים מילוני, שנותן גישה לרשומה באמצעות המפתח המתאים לה. המבנה עובד באמצעות הפיכת המפתח על ידי פונקציית גיבוב, למספר המייצג אינדקס במערך שמפנה אל הרשומה המבוקשת. הפעולה העיקרית שבה היא תומכת ביעילות היא אחזור מידע מתוך מבנה הנתונים: בהינתן מפתח נתון (למשל שם של אדם) מצא את הרשומה המתאימה (למשל מס' הטלפון של האדם).

הנחות:
לכל איבר ישנו שדה מפתח (key) בהתאמה חד-חד ערכי. אין שני איברים עם מפתח זהה.
למשל, עבור רשימת אנשים, המפתח הוא ת.ז. לאיבר המערך המפתח הוא האינדקס שלו, המהווה את מיקומו של האיבר במערך.
טבלאות גיבוב הן טבלאות אינדקסים, המקשרות בין איבר למקומו במערך. באופן זה מתאפשרת גישה ישירה לחיפוש איבר במערך בסיבוכיות של O(1).
מממשים מילון ע"י מערך גישה ישירה (Direct Addressing). המפתח עצמו משמש כאינדקס במערך. אם מרחב המפתחות גדול, נחשב את האינדקס מתוך המפתח ע"י פונקציית ערבול (hash function) שהטווח שלה כתחום האינדקסים במערך.

הגדרה: פונקציית hash - היא פונקציה שמקבלת מפתח כארגומנט ומחשבת אינדקס בטווח המתאים. כלומר:
$\text{hash}: \text{keys} \rightarrow \{1, \dots, m\}$.

בעיות ההתנגשות (Collision):
בעיית ההתנגשות נובעת מכן שלעיתים פונקציית hash נותנת שני ערכים לאותו אינדקס.
למשל: $h(15) = 15 \bmod 10 = 5$ וגם $h(25) = 25 \bmod 10 = 5$ מניבים אותו אינדקס אבל הערכים שונים.

פתרון א' - בדיקה ליניארית (Linear Probing)
אם הפונקציה החזירה אינדקס של מקום במערך שכבר תפוס ע"י ערך אחר, נכניס את הערך החדש באינדקס:
$$\text{hash}(\text{key}, i) = (\text{hash}(\text{key}) + ip) \bmod m$$
כאשר p - קבוע, i - מספר הניסוי, $\text{hash}(\text{key})$ - פונקציית גיבוב (hash code או hash function).
כאשר $p = 1$, בודקים את התא הבא (כלומר, דילוגים של 1), m - מספר התאים בטבלה.
אם המקום הבא תפוס, מחשבים את האינדקס עבור $(i + 1)$ וכן הלאה.
בשיטה זו יש בעיה במחיקת איבר – לא ניתן למחוק איבר ולצמצם מערך - שרשרת החיפוש תתנתק.
יש לסמן את האיבר שנמחק ע"י דגל בוליאני deleted (כך נדע שהאינדקס פנוי).
[שיטה זו קלה לחישוב ולהבנה ושומרת על מקומיות בזיכרון דבר שחוסך גישה לדיסק הקשיח.
אך היא יכולה לצור בעיה של הצטברות רשומות בסדרת אינדקסים ובעקבות כך יהיה מספר גדול של התנגשויות עבור הרשומות באותם באינדקסים.
הערה: ניתן לראות שהערך ההתחלתי קובע את המשך הסדרה, לכן יש בדיוק m רשימות אינדקסים אפשריות.]

פתרון ב' - בדיקה ריבועית - עבור כל התנגשות נבדק מקום מתאים אחר תוך שימוש בהעלאה בריבוע של מספר ההתנגשויות שהיו עד כה עבור אותו מפתח. הפונקציה עבור הבדיקה הריבועית היא:
$$h(\text{key}, i) = (h(k) + i \cdot p_1 + i^2 \cdot p_2) \bmod m$$
ההצטברות עבור הבדיקה הריבועית פחות משמעותית מההצטברות המתרחשת עבור הבדיקה הליניארית אך הבדיקה הריבועית שומרת על מקומיות בזיכרון ברמה פחותה.

פתרון ג' - גיבוב כפול (double Hash)

בשיטת של double hash משתמשים בשתי פונקציות גיבוב שונות: $h_1(\text{key})\ ,\ h_2(\text{key})$

$$h(\text{key},\ i) = (h_1(\text{key}) + i \cdot h_2(\text{key}))\text{mod}\ m$$

כאשר i - מספר הניסוי, $\text{hash}(\text{key})$ - פונקציית גיבוב , m - מספר התאים בטבלה.

ההצטברות בגיבוב כפול מתרחשת רק עבור מפתחות שבאופן נדיר שתי הפונקציות מחזירות עבורם את אותו הערך ובכך עדיפה על הבדיקה הריבועית אבל לא נשמרת המקומיות בכלל.

<u>דוגמה:</u> $m = 13$ ונתונה טבלת גיבוב

$$h(\text{key},\ \ i) = (h_1(\text{key}) + i \cdot h_2(\text{key}))\ \text{mod}\ 13,$$
$$h_1(\text{key}) = \text{key mod}\ 13\ \ \ ,h_2(\text{key}) = 1 + \text{key mod}\ 11$$

מוסיפים 14:

$i=0,\ h(14{,}0) = (h_1(14) + 0 \cdot h_2(14)\ )\ \text{mod}\ 13 = 1$ התא אינו פנוי

$i=1,\ h(14,\ 1) = (h_1(14) + 1 \cdot h_2(14)\ )\ \text{mod}\ 13 = 5$ התא אינו פנוי

$i=2,\ h(14,\ 2) = (h_1(14) + 2 \cdot h_2(14)\ )\ \text{mod}\ 13 = 9$ הניסוי הצליח

פתרון ד' - מערך של רשימות מקושרות - שיטת שרשראות

במערך רגיל אי-אפשר להכניס שני ערכים לאותו מקום, על כן ניצור מערך של רשימות מקושרות שבו אפשר להכניס באינדקס אחד מספר ערכים המוחזקים ברשימה. באופן זה, הערך <u>תמיד</u> יכנס ישירות לאינדקס שפונקציית hash החזירה, ואם הגיע ערך לאותו אינדקס, הוא ישורשר אחרי זה שכבר קיים ברשימה מקושרת. (כלומר, הערך החדש נהיה ראש הרשימה) בשיטה זו רוב הערכים יהיו בפיזור טוב.

סיבוכיות הכנסת איבר וחיפוש איבר בטבלת גיבוב

פונקציית hash מחשבת את האינדקס בזמן $O(1)$ זמן הכנסה והחיפוש יהיה מהיר, שכן באינדקס משורשר יש מספר קטן מאוד של איברים. מספר האיברים שמשורשרים בכל כניסה בממוצע הוא מספר האיברים הכללי מחולק למספר הכניסות.

נסמן את מספר האיברים בפועל ב- n ומספר הכניסות במערך ב-m,

אזי סה"כ מספר האיברים בממוצע עבור כל כניסה הוא $\alpha = n/m$ משתנה זה נקרא load factor.

כדוגמה בעולם האמתי, load factor המוגדר כברירת המחדל עבור HashMap ב-java10 הוא 0.75,

אשר "מציע איזון טוב בין עלויות הזמן וזיכרון". במילים אחרות load factor הוא מספר איברים ממוצע ברשימה.

מציאת איבר ברשימה מקושרת מחייבת מעבר על כל איברי הרשימה באופן ליניארי $O(\alpha)$.

לכן הסיבוכיות של מציאת איבר ברשימה של α איברים היא $O(\alpha)$.

הסיבוכיות של מציאת הכניסה היא $O(1)$ וסה"כ הסיבוכיות של חיפוש איבר היא $O(\alpha + 1)$.

דרישות לפונקציה המחשבת את ה-hashCode:

א) מספקת פיזור טוב.

ב) קלה לחישוב.

ג) m צריך להיות מספר ראשוני רחוק מחזקה של 2. כאשר m שווה לחזקה של 2 הפיזור לא יהיה אחיד.

שיטת הכפל:

- בחר קבוע $0 < \alpha < 1$.
- הכפל את המפתח key בקבוע α.
- מצא את החלק השבור של התוצאה.
- הכפל את החלק השבור ב-m ועגל כלפי מטה.
- הנוסחה: $\text{hash}(\text{key}) = \lfloor m \cdot (\alpha \cdot \text{key mod}\ 1) \rfloor$.
- הערך של m אינו קריטי.
- ערך של α הגורם לפיזור טוב הוא $\alpha = \frac{(1-\sqrt{5})}{2} = 0.61803$.

דוגמא:

עבור $m = 10000$ ו- $\text{key} = 123456$

Hash(k) = $\lfloor 10000 \cdot (123456 \cdot 0.618 \ldots \text{mod}1) \rfloor$
= $\lfloor 10000 \cdot (73600 \cdot 004115 \ldots \text{mod}1) \rfloor$
= $\lfloor 10000 \cdot 004115 \ldots) \rfloor$
= $\lfloor 41.151 \ldots \rfloor$
= 41

דגשים ומושגים שכדאי לזכור

עץ מושלם - מאוזן, לכל הצמתים, מלבד העלים יש שני בנים בדיוק, כל העלים באותה רמה.
עץ שלם - אותו דבר כמו עץ שלם לפי אליזבט.
עץ מלא - יכול להיות לא מאוזן, לכל צומת שאינו עלה יש שני בנים.

* בעץ ערימה מדובר בעץ כמעט שלם.

pre-order: קריאת הקודקוד, ואח"כ קריאת תתי העץ שלו, תחילה תת-עץ השמאלי ואח"כ הימני. בהינתן הדפסת עץ המתבססת על סדר תחילי ניתן אף לשחזר את העץ בצורה חח"ע אם הוא עומד במאפייניו של עץ חיפוש.
in-order: קריאת תת-העץ השמאלי, לאחר מכן קריאת השורש (הצומת ממנו התחלנו) ואח"כ תת-העץ הימני.
post-order קריאת תת-העץ השמאלי, לאחר מכן תת-העץ הימני ולאחר מכן קריאת השורש. בהינתן הדפסת עץ המתבססות על סדר סופי, ניתן אף לשחזר את העץ בצורה חח"ע, אם אכן הוא עומד במאפייניו של עץ חיפוש.

מימוש ההדפסה (הקריאה הראשונית לפונקציה היא עם השורש):

הדפסה בסדר: **Postorder**	<pre>void printPostorder(Node node) {
	    if (node == null) return;
	    /* first recur on left child */
	    printInorder(node.left);
	    /* now recur on right child */
	    printInorder(node.right);
	   /* then print the data of node */
	    System.out.print(node.key + " ");
	}</pre>

הדפסה בסדר: **Inorder**	<pre>void printInorder(Node node) {
	    if (node == null) return;
	    /* first recur on left child */
	    printInorder(node.left);
	    /* then print the data of node */
	    System.out.print(node.key + " ");
	    /* now recur on right child */
	    printInorder(node.right);
	}</pre>

הדפסה בסדר: **Preorder**	<pre>void printPreorder(Node node) {
	    if (node == null) return;
	    /* first print data of node */
	    System.out.print(node.key + " ");
	    /* then recur on left sutree */
	    printPreorder(node.left);
	    /* now recur on right subtree */
	    printPreorder(node.right);
	}</pre>

עץ AVL מאוזן מינימלי בגובה 9: