מבנה נתונים – תרגיל בית מעשי 2

תיעוד:

הערה: משום שזו ערימה d-ארית, לכל צומת פנימי d עלים (פרט לצומת האחרון שאינו עלה). מאחר והעץ המייצג את הערימה הינו עץ d-ארי כמעט מלא, אז המסלול הארוך ביותר משורש לעלה הינו $\lfloor \log_d n \rfloor$, ולכן לפונקציות שזמן הריצה שלהן תלוי בגובה העץ תהיה סיבוכיות זמן ריצה התלויה ב- $O(\log_d n)$.

`public class DHeap`	
הסבר	מימוש ערימה d-ארית עם מפתחות שלמים לא ייחודיים וערכים מתאימים
שדות	`private int size` מספר האיברים בערימה
	`private int max_size` מספר האיברים המקסימלי שהערמה מסוגלת להכיל
	`private int d` פרמטר הערימה (לכל צומת d ילדים, מלבד אולי לעלה האחרון, שיש לו עד d ילדים)
	`private DHeap_Item[] array` מערך של איברי `DHeap_Item`, המייצג את הערימה
פונקציות	`DHeap(int m_d, int m_size)` תיאור: בנאי המחלקה – מייצר עצם חדש מסוג ערימה d-ארית. הערימה הנוצרת הינה ריקה. אופן הפעולה: המתודה מייצרת מערך ריק בגודל `m_size` של איברי `DHeap_Item` המייצג את הערימה. `m_size` יהיה מספר האיברים המקסימלי (`max_size`) ו-d יהיה פרמטר הערימה. מאפסת את גודל הערימה. סיבוכיות: כל הפעולות בעלות זמן ריצה קבוע ולכן - $O(1)$
	`public int getSize()` תיאור: המתודה מחזירה את מספר האיברים בערימה. אופן הפעולה: מחזירה את השדה `size`. סיבוכיות: $O(1)$
	`public int arrayToHeap(DHeap_Item[] array1)` תיאור: הפונקציה בונה ערימה חדשה מהמערך `array1`, תוך דריסת האיברים הקודמים בערימה (באם היו). מחזירה את מספר ההשוואות שנעשו בתהליך. אופן הפעולה: הכנסת כל האיברים שבמערך `array1` לערימה לפי סדר הופעתם, תוך דריסת האיברים הקודמים (באם היו כאלו), ועדכון שדה המיקום `pos` של כל איבר, לפי מיקומו בהכנסה למערך הערימה. עדכון שדה ה-`size` לגודל המערך `array1`. מעבר על איברי הערימה מהסוף להתחלה (לא כולל עלים) וביצוע `heapifyDown` על כל אחד מהם. מוחזר מספר ההשוואות – סכום התוצאות המוחזרות מ- `heapify`. סיבוכיות: הכנסת כל האיברים מהמערך `array1` למערך הערימה - $O(n)$ (כאשר n מייצג את גודל המערך `array1`). ביצוע `HeapifyDown` מהסוף להתחלה – כפי שראינו בכיתה, לוקח $O(n)$ בלי תלות בפרמטר d (♦ הסבר מפורט יותר בסוף הטבלה). סה"כ $O(n)$.
	`private int heapifyDown(int i)` תיאור: מורידה את האיבר במקום ה-`i` במערך במורד הערימה, עד אשר הוא אינו מפר את חוק הערימה. אופן הפעולה: כפי שראינו בכיתה – פונקציה רקורסיבית: במידה ואחד מהבנים של האיבר במקום ה-`i` בעל מפתח קטן יותר, נחליף עם הילד בעל המפתח המינימלי, ונקרא לפונקציה רקורסיבית עם האינדקס החדש. נחזיר את מספר ההשוואות שנעשו. סיבוכיות: בכל פעם עוברים על כל d הילדים של האיבר. במקרה הגרוע נבצע כך לאורך כל גובה העץ. לכן בסה"כ נקבל $O(d \cdot \log_d n)$.
	`private int heapifyUp(int i)` תיאור: מעלה את האיבר במקום ה-`i` במערך במעלה הערימה, עד אשר הוא אינו מפר את חוק הערימה. אופן הפעולה: כפי שראינו בכיתה – בלולאה כל עוד לא הגענו לשורש, אם המפתח של ההורה של האיבר במקום ה- `i` גדול מהמפתח של האיבר עצמו, נחליף ביניהם ונעדכן את `i`. נחזיר את מספר ההשוואות שנעשו. סיבוכיות: במקרה הגרוע נעלה עד השורש, כלומר $O(\log_d n)$ פעמים. בכל איטרציה מבצעים פעולה מסדר גודל של קבוע $O(1)$ בלי תלות בפרמטר d. לכן סה"כ $O(\log_d n)$.
	`public boolean isHeap()` תיאור: מחזירה `true` אם ורק אם הערימה המיוצגת במערך `array` (שדה של הערימה) חוקית. אופן הפעולה: קוראת למתודה `isHeapAux` עם האינדקס `0`, ומחזירה את הערך המוחזר ממנה. סיבוכיות: זהה לזו של `isHeapAux` עבור הערימה כולה - $O(n)$.
	`private boolean isHeapAux(int i)` תיאור: המתודה מחזירה `true` אם ורק אם הערימה ששורשה ב-`array[i]` חוקית (או ריקה לגמרי).

	<u>אופן הפעולה</u>: נעשית השוואה בין מפתח כל אחד מן הילדים של האיבר ב- `array[i]` לבין מפתח האיבר ב- `array[i]`. באם נמצא ילד אשר הינו בעל מפתח קטן יותר משל ההורה, יוחזר FALSE. אחרת, המתודה תקרא רקורסיבית על כל אחד מן הילדים. אם לא נמצא אף איבר אשר מפר את חוק הערמה, יוחזר TRUE. <u>סיבוכיות</u>: המתודה עוברת על כל האיברים בערימה ששורשה באינדקס `i`, ועבור כל אחד מהאיברים מבצעת פעולת השוואה בודדת הלוקחת $O(1)$ (האם המפתח גדול מהמפתח של האבא). לכן, סיבוכיות זמן הריצה הינה כגודל תת-הערימה, $O(k)$, כאשר k מספר האיברים בתת-הערימה. עבור חסם עם גודל הערימה המקורית, נקבל $O(n)$.
	`public static int parent(int i, int d)` <u>תיאור</u>: מחזירה את מיקום ההורה של האיבר באינדקס ה-i במערך המייצג את הערימה (עם פרמטר d). <u>אופן הפעולה</u>: נחזיר את האינדקס של ההורה לפי תכונות ערימה המיוצגת במערך. <u>סיבוכיות</u>: חישוב פשוט בזמן ריצה קבוע - $O(1)$.
	`public static int child (int i, int k, int d)` <u>תיאור</u>: מחזירה את מיקום הילד ה-k של האיבר באינדקס ה-i במערך המייצג את הערימה (עם פרמטר d). <u>אופן הפעולה</u>: נחזיר את האינדקס של הילד ה-k לפי תכונות ערימה המיוצגת במערך. <u>סיבוכיות</u>: חישוב פשוט בזמן ריצה קבוע - $O(1)$.
	`public int Insert(DHeap_Item item)` <u>תיאור</u>: הפונקציה מכניסה את האיבר `item` לערימה ומחזירה את מספר ההשוואות שנעשו בתהליך. <u>אופן הפעולה</u>: הכנסה במקום האחרון בערימה, עדכון שדה ה-size, וקריאה ל- `Heapify-Up` עם אינדקס האיבר שזה עתה הכנסנו לשם שמירה על תכונות הערימה. נחזיר את מספר ההשוואות שנעשו – מוחזר מהפונקציה `heapifyUp`. <u>סיבוכיות</u>: כפי שראינו בכיתה – ההכנסה עצמה ועדכון השדה לוקחים $O(1)$, והתיקון באמצעות `Heapify-Up` שמטפס קומה בערימה בכל איטרציה, במקרה הגרוע ייקח כגובה הערימה. כלומר, בסה"כ $O(\log_d n)$.
	`public int Delete_Min()` <u>תיאור</u>: הפונקציה מוחקת את איבר המינימום של הערימה ומחזירה את מספר ההשוואות שנעשו בתהליך. <u>אופן הפעולה</u>: קריאה לפונקציה `Delete` עם שורש הערימה (האיבר באינדקס 0) שהוא האיבר המינימלי. <u>סיבוכיות</u>: זהה לזו של `Delete`. כלומר, בסה"כ $O(d \cdot \log_d n)$.
	`public DHeap_Item Get_Min()` <u>תיאור</u>: הפונקציה מחזירה את איבר המינימום של הערימה. <u>אופן הפעולה</u>: החזרת האיבר באינדקס `0` במערך, כלומר השורש. <u>סיבוכיות</u>: $O(1)$.
	`public int Decrease_Key(DHeap_Item item, int delta)` <u>תיאור</u>: הפונקציה מקטינה את המפתח של האיבר `item` ב- `delta` ומחזירה את מספר ההשוואות בתהליך. <u>אופן הפעולה</u>: עדכון שדה המפתח של `item` לפי `delta`. תיקון הערימה באמצעות `Heapify-Up` על האינדקס של `item`, שכן יתכן והמפתח החדש שלו נמצא גבוה יותר בערימה. נחזיר את מספר ההשוואות שנעשו, המוחזר מהפונקציה `heapifyUp`. <u>סיבוכיות</u>: עדכון שדה המפתח לוקח $O(1)$, והתיקון באמצעות `heapifyUp` יכול לקחת במקרה הגרוע כגובה הערימה. לכן, בסה"כ נקבל $O(\log_d n)$.
	`public int Delete(DHeap_Item item)` <u>תיאור</u>: הפונקציה מוחקת את האיבר `item` מהערימה ומחזירה את מספר ההשוואות שנעשו בתהליך. <u>אופן הפעולה</u>: באם בערימה רק איבר אחד – נעדכן את שדה ה-`size` ל-0. אחרת: דריסת האיבר `item` עם האיבר האחרון בערימה. עדכון שדה המיקום של איבר זה ועדכון גודל הערימה. תיקון מיקומו בערימה באמצעות `heapifyDown` על האינדקס שלו. נחזיר את מספר ההשוואות שנעשו כפי שמוחזר מ- `heapifyDown`. <u>סיבוכיות</u>: במקרה הגרוע - החלפה עם האיבר האחרון, עדכון המפתח וגודל הערימה - $O(1)$, תיקון הערימה באמצעות `heapifyDown` - $O(d \cdot \log_d n)$. סה"כ קיבלנו $O(d \cdot \log_d n)$.
	`public static int DHeapSort(int[] array, int d)` <u>תיאור</u>: הפונקציה ממיינת `in-place` את המערך `array` באמצעות ערימה עם פרמטר d. מחזירה את מספר ההשוואות שנעשו בתהליך. <u>אופן הפעולה</u>: יצירת מערך של `DHeap_Item` ע"י מעבר על כל איברי מערך `array`. יצירת ערימה ריקה ומילויה באמצעות הפונקציה `arrayToHeap` עם המערך שבנינו. באם יש n איברים בערימה, נבצע n פעמים: נדרוס במקום המתאים במערך `array` עם האיבר הבא באמצעות `Get_Min` ונמחק אותו מהערימה באמצעות `Delete_Min`. נחזיר בסוף את מספר ההשוואות כפי שקיבלנו מ- `arrayToHeap` ומ- `Delete_Min`. <u>סיבוכיות</u>: יצירת המערך לוקחת $O(n)$. מילוי המערך באמצעות `arrayToHeap` – גם $O(n)$. אבל מחיקת האיבר המינימלי n פעמים תיקח $O(nd \cdot \log_d n)$. לכן בסה"כ קיבלנו $O(nd \cdot \log_d n)$.

♦ <u>חישוב מפורט עבור סיבוכיות arrayToHeap</u>:

נתבסס על החישוב שראינו בכיתה: אנו מבצעים פעולות heapifyDown לכל איברי הערימה (פרט לעלים) מהסוף להתחלה. נספור את מספר הצמתים שעוברים heapifyDown בסה"כ בכל רמה של העץ: בגובה 1 של העץ (קומה מעל העלים), יעברו לכל היותר $\frac{n}{d}$ צמתים heapify (שכן, במקרה הגרוע, כל הצמתים פרט לעלים יעברו heapifyDown "דרך" רמה זו). בגובה 2 לכל היותר $\frac{n}{d^2}$, וכך הלאה (החלוקה ב- d נובעת מכך שזהו מספר הצמתים הנמצאים ברמה זו ומעלה, לכל היותר, שכן לכל צומת d ילדים). מספר הפעולות לכל פעולת heapifyDown הוא לכל היותר גובה הצומת לו מבצעים heapify כפול מספר הילדים שיש לכל צומת - d (כי הצומת יכול לרדת לכל היותר עד גובה 0 ולהפוך לעלה). נסמן את גובה הערימה ב- H ונקבל בסה"כ:

$$Total\ time = d \cdot \frac{n}{d} + 2d \cdot \frac{n}{d^2} + 3d \cdot \frac{n}{d^3} + \cdots + Hd \cdot 1 =$$

$$= \sum_{h=1}^{H} h \cdot \frac{n}{d^{h-1}} < n \cdot \sum_{h=1}^{\infty} \frac{h}{d^{h-1}} \leq n \cdot \sum_{h=1}^{\infty} \frac{h}{2^{h-1}} = O(n)$$

כאשר המעבר האחרון נובע מההוכחה שהראינו בכיתה עבור ערימה בינארית (והמעבר שלפניו נובע כמובן מהעבודה כי $d \geq 2$). ולכן, בסה"כ סיבוכיות זמן הריצה של arrayToHeap תהיה $O(n)$ ללא תלות בפרמטר d.

- **חלק המדידות – בעמוד הבא.**

מדידות:

חלק א':

מספר סידורי	גודל המערך m	פרמטר הערימה d	מספר ההשוואות במיון המערך בעזרת הערימה
1	1,000	2	16,855.6
2	1,000	3	16,415.1
3	1,000	4	17,604.9
4	10,000	2	235,394.0
5	10,000	3	226,610.2
6	10,000	4	242,798.1
7	100,000	2	3,018,772.1
8	100,000	3	2,896,798.5
9	100,000	4	3,077,129.6

ניתוח מספר ההשוואות האסימפטומטי עבור מיון המערך

מיון מערך כולל הכנסת המערך לערמה ואז הסרת המינימום והכנסתו למקום הפנוי הראשון במערך עד אשר לא נותרו איברים בערמה. ניתוח סיבוכיות זמן הריצה של הפעולות הללו בחלק התיעוד שלעיל מבוסס על מספר ההשוואות שיש לבצע, ולכן מהווה אומדן גם למספר ההשוואות האסימפטומטי - $O(nd \cdot \log_{\mathrm{d}} n)$. מכאן שיש בידינו חסם עליון למספר ההשוואות אשר יתבצעו במהלך מיון באמצעות ערימה במקרה הגרוע ביותר. נראה כי זה הוא גם חסם הדוק.

במקרה הגרוע ביותר, מיון המערך ידרוש שנבצע את פעולת heapify-down על כל אחד ואחד מאיברי הערימה. כמובן, לאחר כל פעולת delete-min מספר האיברים בעץ יורד, ולעיתים גם מספר הרמות. תהי ערימה בעלת n איברים ו-k רמות. באם הרמה האחרונה מלאה, נקבל: $n = 1 + d + d^2 + \cdots + d^k = \frac{d^{k+1}-1}{d-1}$. נסדר ונקבל $d^k = \frac{n(d-1)+1}{d}$. משום ש- d^k מהווה את מספר האיברים ברמה האחרונה (במידה והערימה מלאה), נקבל כי מספר העלים בערימה מלאה $\frac{n(d-1)+1}{d}$. עבור ניתוח .w.c, נתייחס לערימה מלאה עם מספר עלים אסימפטוטי שכזה.

באותו האופן (וכפי שנהגנו בהוכחת זמן הריצה של arrayToHeap), בערימה מלאה, מספר האיברים ברמה מעל האחרונה יהיה פשוט חלוקה ב- d של מספר האיברים ברמה זו. כמו כן - הדבר תקף גם לכל שאר הרמות. לכן, ניתן להתייחס למספר האיברים בעומק h (כאשר עומק השורש 0) כ- $\frac{n(d-1)+1}{d^{H-h+1}}$, כאשר $H = \Theta(\log_d n)$ עומק העץ (מספר הרמות + 1). כפי שהוסבר לעיל, המקרה הגרוע ביותר יתרחש כאשר נאלץ לבצע heapify-down לכל אחד מאיברי המערך. איבר שבסופו של דבר יגיע לעומק h יצטרך לבצע d השוואות h פעמים. סה"כ, נקבל את הסכום –

$$Hd \cdot \frac{n(d-1)+1}{d} + (H-1)d\frac{n(d-1)+1}{d^2} + \cdots + d \cdot \frac{n(d-1)+1}{d^H} = \sum_{i=1}^{H} (H+1-i)\frac{n(d-1)+1}{d^{i-1}}$$
$$= \sum_{i=0}^{H-1} (H-i)\frac{n(d-1)+1}{d^i} = (n(d-1)+1)\sum_{i=0}^{H-1}\frac{(H-i)}{d^i}$$
$$= (n(d-1)+1)\left(H\sum_{i=0}^{H-1}\frac{1}{d^i} - \sum_{i=0}^{H-1}\frac{i}{d^i}\right) \geq (n(d-1)+1)\left(H\sum_{i=0}^{H-1}\frac{1}{d^i} - \sum_{i=0}^{\infty}\frac{i}{d^i}\right)$$
$$= (n(d-1)+1)\left(H\frac{1-\frac{1}{d^H}}{1-\frac{1}{d}} - O(1)\right) = \Theta\left(n(d-1)H\frac{d^H-1}{d^{H-1}(d-1)}\right)$$
$$= \Theta\left(n \cdot \log_d n \cdot \frac{n-1}{\frac{n}{d}}\right) = \Theta(nd \cdot \log_d n)$$

כאשר המעבר מהסכום $\sum_{i=0}^{H-1}\frac{i}{d^i}$ לאיבר קבוע נובע מההוכחה שהראינו בניתוח סיבוכיות arrayToHeap.

הדבר תואם לתוצאות שקיבלנו בטבלה, שכן ככל שיש יותר איברים בערימה, בפעולת heapify-down המתבצעת כתוצאה מה- delete-min, נצטרך לעשות יותר השוואות אסימפטוטית, שכן גובה הערימה גדול יותר.

חלק ב':

מספר סידורי	x - דלתא עבור Decrease-Key	d פרמטר הערימה	מספר ההשוואות בכל פעולות Decrease-Key
1	1	2	99,999.0
2	1	3	99,999.0
3	1	4	99,999.0
4	100	2	152,827.4
5	100	3	130,845.5
6	100	4	122,956.3
7	1,000	2	303,683.0
8	1,000	3	213,195.5
9	1,000	4	181,140.4

ניתוח מספר ההשוואות האסימפטומטי עבור ביצוע decrease-key על כל איברי המערך

בחלק התיעוד שלעיל ראינו כי סיבוכיות זמן הריצה (אשר מבוססת על מספר ההשוואות האסימפטוטי) של decreaseKey הינה $O(\log_d n)$. מכאן שחסם עליון על ביצוע n פעולות שכאלו יהיה $O(n \cdot \log_d n)$. נוכיח עתה כי זה הוא חסם הדוק.

במקרה הגרוע ביותר, בכל פעם שנקטין את מפתחו של איבר, הדבר יגרום לכך שהוא יהפוך לאיבר המינימום החדש ויאלץ לטפס במעלה כל העץ המייצג את הערימה. הדבר מתבצע ע"י heapify-up, אשר מבצע השוואות רק עם ההורה של האיבר. מכאן שבמקרה הכי גרוע עבור כל איבר תתבצענה השוואות כגובה העץ, כלומר $\log_d n$ השוואות. מאחר וכל איבר עובר תהליך שכזה, יתבצעו $\Theta(n \cdot \log_d n)$ השוואות.

דוגמה לקלט שיגרום למספר השוואות שכזה: ניקח מערך arr ממוין הפוך (האיבר הכי גדול – הוא הראשון) עם איברים שונים. נכניס את כל איברי המערך לערימה. נבצע Decrease-Key לפי סדר ההכנסה (מהאיבר הכי גדול להכי קטן), בגודל של $x = arr[0] - arr[n-1] + 1$ (כאשר: $arr[0]$ האיבר המקסימלי ו-$arr[n-1]$ האיבר המינימלי). לכן, בכל פעולת Decrease-Key, האיבר לו מקטינים את המפתח יהפוך להיות האיבר המינימלי בערימה (שכן עוברים מהאיבר המקסימלי למינימלי) – נצטרך לעשות לו heapify-up עד לשורש העץ. קיבלנו $\log_d n$ השוואות לכל איבר, כנדרש. כלומר - הראינו קלט הגורם למספר השוואות של $\Theta(n \cdot \log_d n)$, חסם עליון שהתקבל מהפונקציה שבנינו הינו $O(n \cdot \log_d n)$, ולכן חסם הדוק למספר ההשוואות יהיה $\Theta(n \cdot \log_d n)$.

ניתן לראות בטבלת המדידות כי הדבר תואם את ניתוח החסם שעשינו: ככל ש- x גדול יותר כך יש יותר סיכוי כי האיבר יהפוך למינימלי, ובהתאם יעשו יותר השוואות במהלך ה- heapify-up. כמו כן, ככל ש- d גדול יותר, בהתאם לחסם, גובה הערימה נמוך יותר ולכן יעשו פחות השוואות. עבור המקרה הראשון $(x = 1)$, ניתן בקלות להוכיח כי חייב להתקבל המספר 99,999 (אחת פחות ממספר איברי הערימה), שהרי פעולת ה decrease-key מפחיתה את מפתחו של כל איבר ב- 1. מאחר והאיברים מופחתים לפי סדר הכנסתם לערימה, מובטח לנו כי לפני כל הפחתה האיבר יהיה תמיד גדול מההורה שלו (באם יש איברים זהים, סדר ההפחתה שקבענו תמיד יגרום לכך שהאיבר ברמה הכי גבוהה יופחת קודם), ולכן לאחר ההפחתה כל איבר יהיה לכל היותר שווה להורה שלו, ותדרש השוואה אחת בלבד לכל איבר. השורש, כמובן, אינו מושווה לאף איבר, ולכן מספר ההשוואות יהיה אחת פחות ממספר האיברים.