מגישים:

רועי קורן, שם משתמש roikoren, ת.ז. 305428369

נדב גסנר, שם משתמש nadavgasner, ת.ז. 204057566

# מחלקת D-Heap

## שדות:

`private int size` – גודל הערימה כרגע.

`private int max_size` – גודל הערימה המקסימלי.

`private int d` – מגדיר את סוג הערימה (ערימה d-ארית).

`private DHeap_Item[] array` – מערך של DHeap_Item בגודל max_size. מכיל את הערימה עצמה.

## מתודות:

`public int getSize()`

מחזירה את הערך שנמצא ב-size.

**סיבוכיות:** O(1).

`public int arrayToHeap(DHeap_Item[] array1)`

בונה ערימה מהמערך שהועבר לה, שומרת ערימה זו כערימה של המופע ממנו הופעלה המתודה, ומחזירה את מספר פעולות ההשוואה שנעשו במהלך ריצתה. עושה זאת ע"י השמה לשדות של המופע של הערכים המתאימים עבור size, max_size ו-array. לאחר מכן מעדכנת את המיקום של כל איבר במערך ע"י קריאה ל-setPos() עליו עם הערך הנכון. לבסוף בשביל להפוך את המערך לערימה, נעבור על כל האיברים שיש להם ילדים, מהימני הנמוך ביותר עד לשורש, וקריאה למתודה Heapify_Down() על כל אחד מהאיברים הללו, ועדכון משתנה השומר את מספר פעולות ההשוואה שנעשו עד כה. לאחר שביצענו Heapify_Down() על השורש, נקבל ערימה תקינה.

**סיבוכיות:** עוברים על כל n האיברים במערך בשביל לעדכן את המצביע למיקום שלהם. לאחר מכן נותר לבצע פעולות Heapify_Down() על כלל הצמתים שיש להם ילדים. יש לכל היותר n/d צמתים עליהם נבצע Heapify_Down() בגובה 1 עבורם נבצע d השוואות לכל היותר לכל צומת, לכל היותר n/ $d^2$ צמתים כאלה בגובה 2 עבורם נבצע d השוואות כפול 2 לכל צומת, וכן הלאה.

סה"כ: $n + 2\frac{n}{d^1} + 3\frac{n}{d^2} + \cdots + dH = \sum_{h=1}^{H} h\frac{n}{d^{h-1}}$

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

`public boolean isHeap()`

מחזירה true עם המערך ששמור ב-array הוא ערימה, ו-false אחרת. עוברת על כל איבר בערימה, בודקת האם הילדים שלו נמצאים עדיין "בתוך" המערך, אם לא מחזירה true שכן כבר עברנו על כלל האיברים. אם אחד האיברים הוא null, מצב שלא אמור לקרות בריצה תקינה, המתודה מוודאת שכל שאר האיברים אחריו במערך גם הם null, מחזירה true אם כן, שכן אז כל האיברים שאינם null כן מהווים ערימה, ו-false אחרת. אם האיבר אינו null, ונמצא "בתוך" המערך, המתודה בודקת האם הוא כולם גדול או שווה לאבא שלו, ומחזירה false אחרת. אם עברנו על כל האיברים במערך, סימן שהוא אכן ערימה תקינה, והמתודה מחזירה true.

**סיבוכיות**: עוברים על כל איבר , ומשווים אותו לכל אחד מהבנים שלו. ישנם לכל היותר n/d + 1 צמתים אותם נצטרך להשוות עם הבנים שלהם, שכן בכל ערימה d-ארית יש לכל היותר n/d + 1 צמתים עם ילדים, ולכן סיבוכיות זמן הריצה היא O(n).

```java
public static int parent(int i, int d)
```

מחזירה את המיקום של האבא של האיבר שנמצא במקום i במערך. אם i קטן או שווה ל-d, האבא שלו הוא השורש, והמתודה מחזירה 0. אחרת אם d מחלק את i, זה אומר ש-i הוא הבן הימני ביותר של האבא שלו, והמיקום שלו במערך הוא i/d – 1. אחרת, האבא של האיבר יושב במקום המתאים לערך התחתון של i/d.

**סיבוכיות:** בכל מקרה מבוצע חישוב פשוט, לכן O(1).

```java
public static int child (int i, int k, int d)
```

מחזירה את המיקום של הבן ה-k של האיבר שנמצא במקום ה-i במערך. עושה זאת ע"י חישוב פשוט של i*d + k.

**סיבוכיות:** O(1).

```java
public int Insert(DHeap_Item item)
```

מכניסה את האיבר item לערימה, ומחזירה את מספר ההשוואות שנעשו בדרך. ראשית המתודה מוודאה שיש מקום להכניס את האיבר החדש לרשימה, אחרת מחזירה 0. אחרת, המתודה מגדילה את הערך השמור ב-size, מכניסה את האיבר במקום האחרון במערך, ומבצעת Heapify_Up() על האיבר שהוכנס, כדי לתקן את הערימה. לבסוף המתודה מחזירה את הערך שהחזירה Heapify_Up(), שהוא מספר ההשוואות שבוצעו בהכנסה.

**סיבוכיות:** במקרה הגרוע נעבור על כל האיברים בדרך מהעלה החדש לשורש, ונשווה ביניהם ובין האיבר החדש. סה"כ נקבל שסיבוכיות זמן הריצה היא: $O(\log_d n) = O(\frac{\log n}{\log d})$.

```java
public int Delete_Min()
```

מוחקת את האיבר המינימלי בערימה, ומחזירה את מספר ההשוואות שנדרשו לשם כך. ראשית המתודה  מוודאה שאכן יש איברים ברשימה, ואחרת מחזירה 0. לאחר מכן היא מקטינה את הגודל של הערימה השמור בשדה size, מעבירה את האיבר האחרון בערימה למקום של השורש, מפעילה עליו את Heapify_Down() ומחזירה את הערך שהחזיר מתודה זו, שהוא מספר ההשוואות שבוצעו במהלך תיקון הערימה.

**סיבוכיות:** מלבד פעולת Heapify_Down(), כל הפעולות במתודה זו מבוצעות ב-O(1). במקרה הגרוע, Heapify_Down() תעבור על כל המסלול מהשורש לאחד העלים, ובכל שלב תבצע סך הכל d השוואות בין האיבר בו אנו נמצאים לבין כל אחד מהילדים שלו. מכיוון והערימה היא עץ d-ארי כמעט מלא, גובהה הוא $O(\log_d n)$ וסיבוכיות זמן הריצה הכוללת היא $O(\frac{d \log n}{\log d})$.

```java
public DHeap_Item Get_Min()
```

מחזירה את האיבר המינימלי בערימה. ראשית מוודאה שהערימה אינה ריקה, ולאחר מכן מחזירה את האיבר שנמצא במקום ה-0.

**סיבוכיות**: O(1).

```java
public int Decrease_Key(DHeap_Item item, int delta)
```

מקטינה את המפתח של item ב-delta, ומתקנת את הערימה לאחר מכן. בעזרת setKey ו-getKey של המחלקה DHeap_Item, מעדכנת את המפתח של האיבר item. לאחר מכן קוראת ל-Heapify_Up() על אותו איבר, לתקן את הערימה במידת הצורך, ומחזירה את מספר ההשוואות שבוצעו סך הכל, אותו ערך אותו תחזיר המתודה Heapify_Up().

**סיבוכיות:** במקרה הגרוע, איבר שהיה אחרון בערימה יהפוך לשורש בסיום ריצת המתודה, ולשם כך נעבור על המסלול מהעלה לשורש, ובכל פעם נבצע מספר קבוע של פעולות. סה"כ סיבוכיות זמן הריצה היא $O(\frac{\log n}{\log d})$.

`public int Delete(DHeap_Item item)`

המתודה מוחקת את האיבר item מהערימה, מתקנת אותה, ומחזירה את מספר ההשוואות שבוצעו במהלך ריצתה. ראשית היא מקטינה את הערך size, ואם הוא שווה ל-0 לאחר ההפחתה, המתודה מחזירה 0, שכן הערימה כעת ריקה (ותקינה). אחרת, המתודה מעבירה את האיבר שהיה במקום האחרון לפני הקטנת הגודל למקום בו נמצא item, ומשווה בין איבר זה לאביו החדש. אם הוא קטן ממנו, המתודה קוראת ל-()Heapify_Up על איבר זה, אחרת ל-()Heapify_Down, ולבסוף מחזירה את הערך שהמתודה שנקראה החזירה, ועוד אחד עבור ההשוואה לאבא החדש.

**סיבוכיות:** עד לבחירה באיזה כיוון יש לתקן את הערימה, כל הפעולות מתבצעות בזמן קבוע, וכבר ראינו (ועוד נראה) את סיבוכיות זמן הריצה של שתי האפשרויות. לכן סה"כ סיבוכיות זמן הריצה היא $O(\frac{d\log n}{\log d})$.

`public static int DHeapSort(int[] array1, int d)`

ממיינת את המערך array1 ומחזירה את מספר ההשוואות שבוצעו במהלך המיון. עושה זאת ע"י הכנסת כל האיברים במערך לערימה חדשה שגודלה כגודל array1, ובעלת ה-d שהועבר למתודה, באמצעות קריאה ל-()Insert על כל האיברים לפי הסדר, ותוך כדי כך מעדכנת משתנה עזר השומר את מספר ההשוואות שבוצעו עד כה. לאחר מכן המתודה מעדכנת את האיברים במערך, אחד אחד, על ידי מציאת האיבר המינימלי בערימה באמצעות קריאה ל-()Get_Min, השמה של המפתח של האיבר שהוחזר במקום הבא במערך array1, מחיקה של האיבר המינימלי ע"י קריאה ל-()Delete_Min, ועדכון מספר ההשוואות שבוצעו בעת מחיקה זו. כשנסיים המערך array1 יהיה ממוין כדרוש.

**סיבוכיות:** אתחול של ערימה חדשה יקח O(n), כל קריאה ל-()Insert תקח $O(\frac{\log n}{\log d})$, ()Get_Min פועלת בזמן קבוע, ו-()Delete_Min תקח $O(\frac{d\log n}{\log d})$. סך הכל, סיבוכיות זמן הריצה היא $O(n\frac{d\log n}{\log d}) = O\left(n\frac{d\log n}{\log d} + n\frac{\log n}{\log d} + n\right)$.

`private int Heapify_Up(DHeap_Item item)`

מתודת עזר, מתקנת את הערימה מ-item כלפי מעלה, ומחזירה את מספר פעולות ההשוואה שבוצעו. ראשית המתודה בודקת האם item הוא השורש, ומחזירה 0 אם כן, שכן אין ל-item אבא שיהיה גדול ממנו. לאחר מכן, מאתחלים משתנה שישמור את מספר ההשוואות ומוצאים את האבא של item. כל עוד item אינו השורש, והמפתח של אבא שלו גדול ממנו, המתודה מגדילה באחד את מספר השוואות, מחליפה בין item ואביו, ומוצאת את האבא החדש של item. אם בסוף הלולאה item אינו השורש, סימן שביצענו השוואה שלא ספרנו, עם השורש עצמו, לכן נוסיף 1 לספירת ההשוואות, ולבסוף נחזיר את מספר ההשוואות שביצענו.

**סיבוכיות:** כפי שראינו, במקרה הגרוע נתחיל בעלה, ונעלה כל הדרך למעלה עד השורש, וסיבוכיות זמן הריצה תהיה $O(\frac{\log n}{\log d})$.

`private int Heapify_Down(DHeap_Item item)`

מתודת עזר, מתקנת את הערימה מ-item כלפי מטה, ומחזירה את מספר פעולות ההשוואה שבוצעו. המתודה מאתחלת משתנה שישמור את מספר ההשוואות שיתבצעו, ומשתנה smallest, שישמור את ה-DHeap_Item עם המפתח המינימלי בכל איטרציה של הלולאה הבאה. בלולאה אינסופית מכוונת, עבור כל אחד מהילדים של item, אם הוא null נצא מלולאת ה-for, שכן הגענו לסוף הערימה ואין ל-item עוד ילדים. אחרת נגדיל את מספר ההשוואות באחד, ואם המפתח של הילד הנוכחי קטן מהמפתח של smallest, מעדכנת את משתנה עזר זה להיות הילד הנוכחי. אם smallest לא שונה באיטרציה הנוכחית, המפתחות של כל הילדים של item גדולים או שווים למפתח שלו, ולכן הערימה תוקנה וניתן לצאת מלולאת ה-while. אחרת, נחליף בין item ובין הקטן שבילדיו, ששמור באותו משתנה smallest, ונחזור על כל התהליך. בסופו של דבר, אם בגלל item עם מפתח שקטן או שווה לכל המפתחות של כל ילדיו, או בגלל item שאין לו ילדים (או שכולם null), נגיע לאיטרציה בה smallest לא ישתנה, ולכן נצא בבטחה מהלולאה האינסופית. לבסוף נחזיר את מספר ההשוואות שבוצעו במהלך ריצת המתודה.

**סיבוכיות:** כפי שראינו, במקרה הגרוע נתחיל בשורש, ונעבור את כל הדרך עד לאחד העלים, כשבכל שלב נבצע d פעולות השוואה, לכן סיבוכיות זמן הריצה היא $O(\frac{d\log n}{\log d})$.

```
private void Swap(DHeap_Item item1, DHeap_Item item2)
```

מתודת עזר, מחליפה בין item1 ו-item2, ע"י עדכון ערך ה-pos של כל item לזה של ה-item השני, והשמה של כל אחד מהם במקום החדש שלו.

**סיבוכיות**: O(1).

# מדידות

| הבדיקה | מספר האיברים | התוצאה |
|---|---|---|
| Sort, d = 2 | 1000 | 17262.8 |
| | 10000 | 239209.5 |
| | 100000 | 3057707.8 |
| Sort, d = 3 | 1000 | 16673.6 |
| | 10000 | 229264.1 |
| | 100000 | 2923129.3 |
| Sort, d = 4 | 1000 | 17829.7 |
| | 10000 | 245102 |
| | 100000 | 3098842.1 |

Heap-sort ייקח במקרה הגרוע תטא של **ndlogn** השוואות (כל הלוגים בסעיף המדידות הם בבסיס d).

בניית הערימה תיקח O(n) ללא תלות בd כפי שהוכח בתרגול 7.

מחיקת המינימום במקרה הגרוע, תיקח O(dlogn) ( d השוואות בכל רמה, logn רמות) ולכן n מחיקות ייקחו O(ndlogn). זהו גם חסם תחתון בתרחיש הגרוע, אם ניקח את n/2 המחיקות הראשונות, והאיבר הרנדומי שנמצא בסוף המערך יפעפע כל פעם גובה log(n/2), הגענו כבר לאומגה של ((n/2)dlogn(n/2)) שזה תטא של (ndlogn). אם נוסיף את סיבוכיות בניית הערימה זה לא ישפיע על הסיבוכיות.

| הבדיקה | דלתא | התוצאה |
|---|---|---|
| Decrease-Key, d = 2 | 1 | 99999 |
| | 100 | 152961.8 |
| | 1000 | 303225.5 |
| Decrease-Key, d = 3 | 1 | 99999 |
| | 100 | 130805.5 |
| | 1000 | 213188.6 |
| Decrease-Key, d = 4 | 1 | 99999 |
| | 100 | 122881.6 |
| | 1000 | 181074.6 |

DECREASE-KEY ייקח במקרה הגרוע תטא של **nlogn** השוואות.

בניית הערימה תיקח O(n) ללא תלות בd כפי שהוכח בתרגול 7.

בתרחיש הגרוע DECREASE-KEY יפעפע כל פעם את האיבר עד למעלה, ויוריד איבר שעוד לא בוצעה עליו הפעולה להיות עלה, שזה logn, ו- n פעולות כאלו יקחו תטא של nlogn.