**בן שפירא – 313261067 – benshapira**

**רועי כהן – 208460725 – roi1**

**עבודה מעשית – ערימות פיבונאצ'י**

שדות –

**private** HeapNode **first**; - מצביע לתחילת רשימת שורשי העצים  
**private** HeapNode **min**; - מצביע לאיבר המינימלי  
**private int size**; - מספר האיברים בערימה  
**private int marked**; - מספר האיברים שהמצב שלהם הוא marked  
**private int trees**; - מספר העצים בערימה  
**private static int** *links*; - שדה סטטי – כמות הלינקים הכללית במחלקה  
**private static int** *cuts*; - שדה סטטי – כמות ה-cuts הכללית במחלקה

מטודות –

**public boolean** isEmpty()

בודקת האם המצביע לאיבר הראשון ברשימת שורשי העץ הינו null

סיבוכיות – O(1)

**public** HeapNode insert(**int** key)

המטודה יוצרת node חדש המכיל את המפתח הנתון key.

בנוסף המטודה יוצרת ערימה חדשה, המכילה את ה-node החדש בלבד.

לאחר מכן המטודה קוראת למטודה meld אשר מאחדת את שתי הערימות ובכך האיבר מתווסף לערימה.

לאחר מכן מתעדכן המצביע לאיבר הראשון, שכן meld מוסיפה את האיבר החדש להיות בסוף רשימת השורשים הקיימת, לכן נרצה שהוא יהיה בהתחלה, מכיוון שזו רשימה מקושרת דו כיוונית, רק נשנה את המצביע של first.

ה-node מוחזר.

סיבוכיות – O(1). מדובר בסך הכל במספר קבוע של שינויי מצביעים ושינויי שדות.

**public void** deleteMin()

מוחקת את האיבר בעל המפתח המינימלי.

הפעולות שמתבצעות – השורש המינימלי נמחק מרשימת השורשים.

רשימת הילדים של אותו השורש מתווספת להיות ברשימת השורשים.

מתבצע consolidating.

סיבוכיות – בדיוק כפי שחישבנו בכיתה, כל הפעולות הן שינויי מצביעים ומחיקות קלות, אך עלות ה- consolidating היא: Worst case cost – O(*n*)

Amortized cost – O(log n)

**private** HeapNode deleteMinRootFromList(HeapNode x)

בעזרת מספר שינויי מצביעים ועדכוני שדות קלים, נמחק את השורש המינימלי מרשימת השורשים, ונחזיר את ה-child שלו לשימוש עתידי. זאת, תוך תשומת לב למקרים מיוחדים כגון – אם מחקנו את האיבר שהוא first, או אם האיבר הוא השורש היחידי.

סיבוכיות – O(1) כמות קבועה של שינויי מצביעים.

**private void** insertChildrenListToRootList(HeapNode x){

המטודה הופכת את הילדים של השורש המינמלי שנמחק, להיות שורשים בערימה.

מטפלת במקרה שבו מחקנו את השורש היחידי בערימה, במקרה כזה מתבצע unmarkAndUnparent, כלומר כל השורשים החדשים הופכים להיות לא מסומנים (כי הם שורשים) ומתבצע שינוי מצביע כך שלא יצביעו יותר לאביהם שנמחק.

כעת מתבצעת לולאה שמשנה שמכניסה שורש שורש להיות האיבר הראשון ברשימת השורשים.

סיבוכיות – O(logn) מתבצעים רק שינויי מצביעים פשוטים וכן לולאה שרצה על רשימת הילדים ומכניסה אותם להיות שורשים (שמספרם הוא לכל היותר כדרגה המקסימלית של שורש – O(logn)).

**private void** unmarkAndUnparent(HeapNode x){

המטודה מוחקת את המצביע לאבא שנמחק ומבצעת unrank.

סיבוכיות – O(logn) לולאה שרצה על רשימת הילדים ומבצעת שינוי מצביעים פשוט (שמספרם הוא לכל היותר כדרגה המקסימלית של שורש – O(logn)).

**private void** consolidating(){

מתבצעת קריאה לtoBucket ואז fromBuckets.

סיבוכיות – בדיוק כמו delete-min (זוהי הפונקציה בעלת הסיבוכיות הגבוהה ביותר שנקראת בה.

Worst case cost – O(*n*)

Amortized cost – O(log n)

**private** HeapNode[] toBuckets(){

בדיוק כמו שנלמד בכיתה - העצים מוכנסים לתאים – כל אחד מוכנס לתא שהוא כמספר הדרגה שלו. במקרה שבתא יש כבר עץ אחר, מתבצע link ביניהם והם עוברים לתא הבא בתור. כמות התאים היא O(log n).

סיבוכיות – בדיוק כמו consolidating.

**private void** fromBuckets(HeapNode[] buckets){

הפונקציה לוקחת את כל העצים מתוך התאים, ומחזירה אותם להיות שורשים חדשים. תוך כדי מתעדכן המצביע למינימום והעצים יוצאים לפי סדר הדרגות ומוכנסים לתוך הערימה. כך נקבל ערימה חדשה בה לכל היותר O(logn) שורשי עצים.

סיבוכיות – בדיוק כמו consolidating.

**private void** deleteFromRootList(HeapNode y, HeapNode first){

פונקציית העזר שמוחקת איבר כלשהו מרשימת השורשים – כאשר הוא מוכנס לתא כלשהו.

מתבצעים כמה שינויי מצביעים על מנת למחוק אותו, לכן הסיבוכיות היא O(1).

**private void** link(HeapNode x, HeapNode y){

הפונקציה מחברת שני שורשים, כך שאחד יהיה אבא של השני, פעולה זאת מתרחשת בעת הכנסה לתאים, אם הכנסנו עץ לתא כלשהו שכבר היה בו עץ אחר, מתבצע חיבור ביניהם.

מתבצעים כמה שינויי מצביעים על מנת לחבר ביניהם, לכן הסיבוכיות היא O(1).

**private static int** rankBarrier(**int** n){

פונקציה המחשבת את מספר התאים שבמערך שאליו אמורים להיות מוכנסים כל השורשים.

סיבוכיות – O(1).

**public** HeapNode findMin()

מחזירה מצביע לאיבר המינימלי, ששמור אצלנו בשדות

סיבוכיות – O(1).

**public void** meld (FibonacciHeap heap2(

הפונקציה מאחדת שתי ערימות לאחת.

מתבצעים עדכוני שדות, כך שהערימה שלנו – שאליה מתחברת הערימה החדשה תעדכן את שדותיה להיות סכום השדות עם הערימה החדשה, תוך עדכון מינימום ואיבר ראשון.

מתבצע חיבור שתי הערימות, כך שהערימה שלנו מגיעה ראשונה והערימה השניה לאחריה.

אנחנו בודקים בדרך האם הערימה השניה לא ריקה, אם היא ריקה לא נעשה כלום, אם היא לא ריקה והערימה שלנו ריקה, נעשה שינוי כך שהערימה שלנו תהיה שווה לערימה החדשה.

סיבוכיות – O(1) - שינויי מצביעים ועדכוני שדות בלבד.

**public int** size()

מחזירה את כמות האיברים בערימה, ששמורה אצלנו בשדות

סיבוכיות – O(1).

**public int**[] countersRep()

הפונקציה מחזירה מערך ובו במקום ה-i יש את כמות העצים מדרגה i בערימה. ראשית, נבדוק לאיזה שורש יש את הדרגה המקסימלית, כדי לדעת מהו גודל המערך שלנו. לאחר מכן נרוץ על רשימת העצים שלנו ונספור כמה שורשים יש לנו מכל דרגה ונמקם את הכמות הזו במערך לפי הדרגות שלהם.

סיבוכיות – O(#Trees) שכן אנחנו מבצעים מעבר על שורשי העצים פעמיים.

**public void** delete(HeapNode x)

פונקציה שמשתמשת בפונקציות אחרות – ראשית ערך המפתח שיש למחוק אותו מורד להיות המינימום החדש, לאחר מכן מתבצע delete-min על מנת למחוק את המינימום החדש – שהוא המפתח שלנו.

סיבוכיות – כמו delete-min.

**public void** decreaseKey(HeapNode x, **int** delta)

הפונקציה מפחיתה את הערך של המפתח של הnode x ב-delta.

מתבצעת הפחתה למפתח, אם המינימום השתנה, מתבצע עדכון למצביע למינימום.

אם x הוא שורש, או אם x גדול מההורה שלו (מצבים תקינים) נסיים כאן.

במקרה אחר – נבצע cascadingCuts.

סיבוכיות – O(1) אמורטייזד! כפי שניתחנו בכיתה.

**private void** cascadingCut(HeapNode x , HeapNode y){

ראשית נחתוך את x מאביו y. אם y הוא שורש, סיימנו, אם הוא לא שורש, נבדוק האם הוא מסומן כבר, אם לא, נסמנו ונגדיל את מספר המסומנים.

אם הוא כן מסומן, נמשיך רקורסיבית ב-cascadingCuts הפעם עם y (האב של x) ואבא של y.

סיבוכיות – O(1) אמורטייזד! כפי שניתחנו בכיתה.

**private void** cut(HeapNode x , HeapNode y){

נחתוך את x מאביו y.

נבצע מספר עדכוני מצביעים לשם כך, נבדוק האם x היה marked, אם כן נעשה לו unmark.

נבדוק האם יש לעדכן את ילדו של y (אם x היה מוגדר child של y אז נעביר את הילד שלו להיות הילד הבא. נמחק את x מהרשימה של אחיו ונכניס אותו להיות בתחילת הרשימה על ידי כמה שינויי מצביעים.

סיבוכיות – שינויי מצביעים קבוע בלבד, לכן O(1).

**private boolean** isRoot(HeapNode x){

נבדוק האם אבא של x הוא Null – אם כן אז נחזיר true, שכן הוא שורש.

סיבוכיות – O(1).

**public int** potential()

נחזיר חישוב שדות trees + (2 \* marked) לחישוב הפוטנציאל

סיבוכיות – O(1).

**public static int** totalLinks()

פונקציה סטטית שתחזיר את ערכו של המשתנה הסטטי links שמציין את מספר הלינקים שבוצעו.

סיבוכיות – O(1).

**public static int** totalCuts()

פונקציה סטטית שתחזיר את ערכו של המשתנה הסטטי cuts שמציין את מספר ה-cuts שבוצעו.

סיבוכיות – O(1).

**public static int**[] kMin(FibonacciHeap H, **int** k)

המטודה מחזירה את k האיברים המינימלים בעץ הבינומי שאנו מקבלים.

k האיברים המינימלים בהכרח ימצאו ב-k הרמות הראשונות של העץ.

לכן המטודה מעדכנת ערימת עזר ומכניסה אליה את k הרמות הראשונות של העץ באופן רקורסיבי. כל פעולת insert היא בעלות O(1) ובכל רמה יש לכל היותר איברים, לכן מתבצעות פעולות כאלה לכל היותר, לכן סך הכל העלות של יצירת ערימת העזר היא . לאחר מכן נבצע delete-min k פעמים. כל פעולת delete-min היא בעלות של כאשר n הוא מספר האיברים בערימה, במקרה שלנו כל פעולת delete-min היא בעלות מתבצעות k פעולות כאלה ולכן סך הכל העלות של פעולות delete-min *סך הכל היא - . לכן יחד עם פעולת יצירת ערימת העזר, העלות הכוללת היא –*

*כנדרש.*

**מחלקה – HeapNode:**

**public class** HeapNode **implements** Comparable<HeapNode>{

שדות –

**public int key**; - ערך מספרי של המפתח   
**private int rank**; - ערך מספרי המייצג את מספר ילדיו של node  
**private boolean mark**; - האם הnode מסומן או לא  
**private** HeapNode **child**; - מצביע לילדו של ה-node  
**private** HeapNode **next**; - מצביע לnext של ה-node  
**private** HeapNode **prev**; - מצביע לprev של ה-node  
**private** HeapNode **parent**; - מצביע לאביו של ה-node

מטודות –

getter ו-setter לכל השדות – כולל בנאי שמאתחל node עם ערכים דיפולטיים לכל השדות וערך מפתח לפי האינפוט.

כולן בסיבוכיות O(1).

**מדידות:**

1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **m** | **Run-Time**  **(in miliseconds)** | **totalLinks** | **totalCuts** | **Potential** |
| 1024 | 0.7922 | 2010 | 78 | 19 |
| 2048 | 1.1147 | 4057 | 98 | 21 |
| 4096 | 2.1594 | 8152 | 120 | 23 |

א. נצפה לO(m) משום שיש m פעולות insert שעולות O(1) כל אחת, delete-min אחת שעולה O(log m) וdecrease-key (בעלות O(1)) log m פעמים. לכן סך הכל O(m)+2O(logm)=O(m)

ב. links - פעולת delete-min מתבצעת כאשר יש לנו m עצים, לכן מספר פעולות ה-link אמור להיות O(m).

cuts – נבצע O(1) פעולות cut בכל decrease-key באמורטייזד, משום שאם ביצענו cut, אז הnode היה מסומן ובכל סימון של node נשלם שני מטבעות – אחד שילך לפעולת ה-cut העתידית. לכן סך הכל נצפה ל-O(log m) כיוון שיש O(log m) פעולות decrease-key.

ג. עלות פעולת ה-decrease-key היקרה ביותר היא O(log m) כי במקרה זה, נצטרך לבצע cascading-cuts במסלול מעלה ועד השורש. כיוון שלכל היותר סימנו O(log m) צמתים ולכן לא נוכל לבצע cut לצמתים שלא סימנו.

run for m=1000

time ms: 27.8314

totalLinks=10043

totalCuts=120

potential=6

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **M** | **Run-Time**  **(in miliseconds)** | **totalLinks** | **totalCuts** | **Potential** |
| 1000 | 27.8314 | 1891 | 0 | 6 |
| 2000 | 2.5549 | 5780 | 0 | 6 |
| 3000 | 2.7703 | 11552 | 0 | 7 |

א. זמן הריצה הוא משום שביצענו m פעולות insert בעלות O(1) ועוד m/2 פעולות delete-min שעולות כל אחת.

ב. link – . אחרי הdelete-min הראשון, בו נבצע O(m) פעולות links ישארו לנו m/2-1 פעולות delete-min, בהן כל פעם נוסיף לכל היותר עוד עצים ואז לאחר ה-consolidating כמות העצים תהיה שוב . לכן בכל שאר הפעולות נבצע פעולות link ומכאן שזוהי הכמות.

cuts – לא יהיו cuts בכלל, משום שלא ביצענו כל decrease-key

ג. פונקציית הפוטנציאל מושפעת רק מכמות העצים (שכן לא ביצענו decrease-key ולכן אין לנו צמתים marked) וכמות העצים לאחר delete-min היא חסומה על ידי