תרגיל מעשי מספר 1 במבנה נתונים

נופר לוי nufarlevy 203432372 יהונתן זומר somer 307923383

תיעוד המחלקות

תיעוד מחלקת FibonacciHeap

מימשנו מחלקה בשם FibonacciHeap. מחלקה זו הינה יישום שלערימת פיבונאצ'י כפי שנלמד בקורס. הערימה שמימשנו מכילה רשימה של שורשים (הרשימה ממומשת ע"י מחלקה בשם MyList אשר תפורט בהמשך) של עצים ונודים כצמתים בתוך העצים אשר מיושמים על יידי מחלקה נוספת בשם HeapNode (מחלקה זו מפורטת בהמשך המסמך). כל נוד בעץ מכיל מפתח ועוד שדות אחרים שיפורטו בהמשך.

שדות המחלקה

private HeapNode min;
שדה זה מחזיק פוינטר למינימום של הערימה, המינימום הוא נוד מסוג HeapNode. בעת אתחול עץ השדה מקבל את הערך null.

private static int *totalLinks*;
שדה זה מחזיק מספר (int) שמייצג את מספר הלינקים שבוצעו במהלך ריצת התכנית בעץ הנוכחי. בעת אתחול השדה מקבל את הערך 0.

private static int *totalCuts*;
שדה זה מחזיק מספר (int) שמייצג את מספר החיתוכים שבוצעו במהלך ריצת התכנית בעץ הנוכחי. בעת אתחול השדה מקבל את הערך 0.

private MyList roots = **new** MyList();
שדה זה מחזיק רשימה של שורשי הערימה כאשר בכל שורש אנו מחזיקים את המינימום של אותו העץ. בעת אתחול הרשימה מאתחילם אותה לרשימה ריקה (עם 0 איברים).

private int size;
שדה זה מחזיק מספר (int) שמייצג את גודל הערימה. גודל העץ הינו מספר הנודים בערימה. בעת אתחול השדה מקבל את הערך 0.

private int marks;
שדה זה מחזיק מספר (int) שמייצג את מספר האיברים המסומנים שקיימים בערימה לפי מה שנלמד בכיתה איבר מסומן זה איבר שחתכנו ממנו ילד אחד. בעת אתחול השדה מקבל את הערך 0.

פונקציות המחלקה

public boolean empty()

הפונקציה בודקת האם הערימה מכילה נודים. הפונקציה מחזירה true אם הערימה ריקה (לא מכילה נודים), ומחזירה false אם הערימה לא ריקה (מכילה נוד אחת לפחות).

סיבוכיות: O(1)

public HeapNode insert(**int** key)

הפונקציה מקבלת כקלט מפתח key ומוסיפה את המפתח ל HeapNode חדש מדרגה 0. אותו היא מכניסה כשורש מדרגה 0 (בעל איבר אחד) לרשימת השורשים של הערימה. הפונקציה תעדכן את המצביעים next ו-prev של אותו הנוד לפי הצורך. בנוסף היא תעדכן את גודל העץ ואת נוד המינימום (אם יש צורך). הפונקציה מחזירה את אותו הנוד שהוספנו כHeapNode .

סיבוכיות: O(1)

public void deleteMin()

הפונקציה מוחקת מהעץ את הNode בעל המפתח הקטן ביותר ומבצעת consolidate כפי שלמדנו. תחילה נמחק הנוד הנ"ל ורשימת בניו מצורפת לרשימת השורשים concate + removeMin Ologn Worst Case (ראה ניתוח של הפונקציות הנ"ל) לאחר מכן consolidate : amortized (O (log n . .

סיבוכיות: O(logn)

public MyList removeMin()

הסרת השורש והחזרת רשימה של בניו. במקרה הרע O(logn) בנים. נדרש מעבר על הבנים כדי לעדכן את האבא שלהם.

סיבוכיות: O(logn)

public void consolidate()

פונקציה הפועלת בדיוק כפי שנלמד בכיתה. ניתוח סיבוכיות זהה. תוך שימוש בפונקציות עזר toBuckets, fromBuckets

סיבוכיות – amortized O(log n)

O(n) Worst case

public HeapNode findMin()

הפונקציה מחזירה את המינימום של הערימה כHeapNode.

סיבוכיות: O(1)

public void meld (FibonacciHeap heap2)

הפונקציה עושה meld לערימה הנוכחית ולערימה נוספת. מה שהיא בעצם עושה זה לקנקט 2 רשימות מקושרות. בנוסף היא מעדכנת את השדות שרלוונטים לערימות הללו (גודל,marks, min).

סיבוכיות: O(1)

public int size()

הפונקציה מחזירה את גודל הערימה (מספר הנודים בערימה).

סיבוכיות: O(1)

public int[] countersRep()

בוני מערך המתרחב בהתאם לדרגה הגבוהה ביותר שנתקלים בה. מוסיפים 1 למקום המתאים לדרגה של כל עץ ברשימת השורשים של העץ.

public void arrayToHeap(**int**[] array)

הפונקציה מאתחלת את הערימה ואז מוסיפה את רשימת האיברים איבר אחר איבר לערימה.

סיבוכיות: O(n) (n=מספר האיברים ברשימה)

public void delete(HeapNode x)

הפונקציה מוחקת איבר מהרשימה על ידי decreaseKey למינוס אינסוף ואז עושה deleteMin (זה יצביע לאותו האיבר ולכן ימחק).

(לכן הסיבוכיות תהיה הסיבוכיות של decreaseKey ועוד הסיבוכיות של deleteMin)

סיבוכיות: O(logn)

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

הפונקציה מקבלת HeapNode (node) ועושה לו decreaseKey למפתח קטן ממנו (delta). הפונקציה משתמשת בפונקציה cuscadingCuts פונקציה רקרוסיבית שעושה את החיתוכים לפי הצורך (נפרט עליה בהמשך). הפונרציה שלנו חותכת את האיבר שעשינו לו decreaseKey במידה והוא גדול מאבא שלו ומעדכנת את רשימת השורשים, את הסימונים ועושה חיתוכים נוספים לאבות במידת הצורך.

סיבוכיות: O(1)

private void cascadingCut(HeapNode node, HeapNode parent)

פונקצית עזר של decreaseKey, הפונקציה מקבלת node HeapNode ו parent HeapNode קודם כל חותכת אותו מהאבא שלו (הבדיקה אם לחתוך או לא נעשתה בdecreaseKey) מעדכנת את המצביעים ואת רשימת הילדים של האבא ולאחר מכן עוברת לאבא ובאופן רקרוסיבי במידה והוא מסומן חותכת אותו וממשיכה הלאה לאבא שלו.

כך היא חותכת את כל האיברים המסומנים באותה דרך עד לשורש העץ.

סיבוכיות: O(1)

public int potential()

הפונקציה מחזירה את הפוטנציאל של הערימה לפי הנוסחא שנלמדה בכיתה: (מספר העצים ועוד פעמיים מספר האיברים המסומנים) Potential = #trees + 2*#marked

סיבוכיות: O(1)

public static int totalLinks()

הפונקציה מחזירה את מספר הלינקים שבוצעו בריצת התוכנית של הערימה הנוכחית.

סיבוכיות: O(1)

public static int totalCuts()

הפונקציה מחזירה את מספר החיתוכים שבוצעו בריצת התוכנית של הערימה הנוכחית.

סיבוכיות: O(1)

תיעוד מחלקת MyList

מימשנו מחלקה פנימית למחלקת FibonacciHeap בשם MyList . מחלקה זו הינה יישום רשימה מקושרת שבה כל הפונקציות הדרושות לנו על מנת לתפעל את הפעולות על רשימת השורשים בסיבוכיות זמן שדרושה לנו.

שדות המחלקה

private HeapNode head;
שדה זה מחזיק בפוינטר לHeapNode שהוא האיבר הראשון ברשימה. בעת אתחול השדה מקבל את הערך null.

private HeapNode tail;
שדה זה מחזיק בפוינטר לHeapNode שהוא האיבר האחרון ברשימה. בעת אתחול השדה מקבל את הערך null.

private int size;
שדה זה מחזיק במספר int שהוא אורך הרשימה. בעת אתחול השדה מקבל את הערך 0.

פונקציות המחלקה

public boolean isEmpty()

פונקציה שמחזירה ערך בולאיני אם הרשימה ריקה או לא, אם אין בה איברים מחזירה true אם יש בה לפחות איבר אחד מחזירה false.

סיבוכיות: O(1)

public void addLast(HeapNode node)

הפוקנציה מקבלת HeapNode ומוסיפה אותו לסוף הרשימה, מעדכנת גם את הtail של הרשימה ואת כל המצביעים לאותו איבר. במידה והרשימה ריקה מעדכנת את האיבר להיות גם הtail וגם הhead.

סיבוכיות: O(1)

public void concate(MyList heap2)

הפונקציה מקבלת רשימה נוספת heap2 ומקנקטת אותה לרשימה הנוכחית שלנו, היא פשוט מחברת את האיבר האחרון של הרשימה שלנו עם האיבר הראשון של הרשימה החדשה ומעדכנת את הגודל של הרשימה החדשה שנוצרה.

סיבוכיות: O(1)

public void delete(HeapNode node)

הפונקציה מקבלת איבר מתוך הרשימה HeapNode (node), ומוחקת אותו מהרשימה המקושרת על ידי שינוי הפוינטרים של prev וnext של האיברים שיושבים על ידו. במידה והוא האיבר היחידי ברשימה או הtail או הhead מעדכנת את הרשימה בהתאם.

סיבוכיות: O(1)

public int size()

הפונקציה מחזירה את אורך הרשימה.

סיבוכיות: O(1)

public HeapNode getTail()

הפונקציה מחזירה את האיבר האחרון ברשימה - tail.

סיבוכיות: O(1)

public HeapNode getHead()

הפונקציה מחזירה את האיבר הראשון ברשימה - head.

סיבוכיות: O(1)

```
public void clear()
```

הפוקנציה מאתחלת את הרשימה (פשוט מאתחלת את הtail והhead להיות null ואת הsize להיות 0)

סיבוכיות: O(1)

תיעוד מחלקת HeapNode

מימשנו מחלקה פנימית למחלקת FibonacciHeap בשם HeapNode . מחלקה זו הינה יישום של נודים בערימת פיבונאצ'י. כל נוד בערימה מכיל מפתח כאשר כל מפתח יכול להופיע כמה פעמים בערימה. רשימת השורשים מכילה מצביעים לנודים של שורשי העצים והם מכילים את הנודים השונים.

שדות המחלקה

```
private int key;
```
שדה זה מחזיק בפוינטר למפתח של הנוד. בעת אתחול האובייקט מקבל את הערך שהוכנס אליו key.

```
private int rank;
```
שדה זה מחזיק בפוינטר לדרגה של הנוד. בעת אתחול השדה מקבל את הערך 0.

```
private boolean mark = false;
```
שדה זה מחזיק ערך הוליאני האם הנוד מסומן או לא, באתחול הדשה מקבל את הערך false.

```
private List<HeapNode> childs = new ArrayList<HeapNode>();
```
שדה זה מחזיק רשימה של כל הילדים של אותו הנוד. באתחול הרשימה היא מאותחלת לרשימה ריקה בעלת 0 איברים.

```
private HeapNode next;
```
שדה זה מחזיק בפוינטר לHeapNode שבא אחרי הנוד שלנו. בעת אתחול השדה מקבל את הערך null.

private HeapNode prev;
שדה זה מחזיק בפוינטר לHeapNode שבא לפני הנוד שלנו. בעת אתחול השדה מקבל את הערך null.

private HeapNode parent;
שדה זה מחזיק בפוינטר לHeapNode האבא של הנוד שלנו. בעת אתחול השדה מקבל את הערך null.

<u>פונקציות המחלקה</u>

public HeapNode(**int** key)
בנאי של המחלקה מאתחל את המפתח להיות הkey שהוכנס.

public int getKey()
פונקציית get שמחזירה את המפתח של ה- HeapNode עליו בוצעה הפעולה.
סיבוכיות: O(1)

public int getRank()
פונקציית get שמחזירה את הדרגה של ה- HeapNode עליו בוצעה הפעולה.
סיבוכיות: O(1)

public boolean getMark()
פונקציה שמחזירה ערך בוליאני אם הHeapNode מסומן או לא.
סיבוכיות: O(1)

public List<HeapNode> getChilds()
פונקציה שמחזירה את רשימת הילדים של הנוד עליו ביצענו את הפעולה.
סיבוכיות: O(1)

public HeapNode getNext()
פונקציה שמחזירה את הHeapNode הבא אחרי הHeapNode עליו ביצענו את הפעולה.
סיבוכיות: O(1)

public HeapNode getPrev()
פונקציה שמחזירה את הHeapNode הבא לפני הHeapNode עליו ביצענו את הפעולה.
סיבוכיות: O(1)

public HeapNode getParent()
פונקציה שמחזירה את הHeapNode האבא של הHeapNode עליו ביצענו את הפעולה.
סיבוכיות: O(1)

public void setKey(**int** key)
פונקציית set שמאתחלת את המפתח של ה-HeapNode להיות המפתח key (הקלט).
סיבוכיות: O(1)

public void setRank(**int** rank)
פונקציית set שמאתחלת את הדרגה של ה-HeapNode להיות הדרגה rank (הקלט).
סיבוכיות: O(1)

public void setMark(**boolean** mark)
פונקציית set שמאתחלת את הסימון של ה-HeapNode להיות true או false לפי הmark (הקלט).
סיבוכיות: O(1)

public void addChild(HeapNode child)
הפונקציה מוסיפה ילד לרשימת הילדים.
סיבוכיות: O(1)

public void deleteChild(HeapNode child)
הפונקציה מוחקת ילד מרשימת הילדים
סיבוכיות: O(1) (לפי המימוש של מחיקה מarrayList)

public void setNext(HeapNode next)

פונקציית set שמקבלת כקלט HeapNode ומאתחלת אותו להיות הבא אחריו של ה-HeapNode הנוכחי.
סיבוכיות: O(1)

public void setPrev(HeapNode prev)
פונקציית set שמקבלת כקלט HeapNode ומאתחלת אותו להיות הבא לפניו של ה-HeapNode הנוכחי.
סיבוכיות: O(1)

public void setParent(HeapNode parent)
פונקציית set שמקבלת כקלט HeapNode ומאתחלת אותו להיות האבא של ה-HeapNode הנוכחי.
סיבוכיות: O(1)

מדידות

להלן תוצאות המדידות שבוצעו על העץ שמימשנו:

Sequence 1:

m	**Run-Time (in miliseconds)**	**totalLinks**	**totalCuts**	**Potential**
1000	1	0	0	1000
2000	3	0	0	2000
3000	5	0	0	3000

הסבר לתוצאות שהתקבלו:

זמן הריצה האסימפטוטי:

ראשית, מכיוון שכל פעולת insert מתבצעת ב$O(1)$. אז הזמן של n פעולות הוא לינארי בm. ניתן לראות לפי תוצאות המדידות כי הזמן הוא בערך: $T = 2m-1$ (milliseconds)

כלומר $O(m)$

מספר לינקס וקאטס:

כמובן 0. אנחנו מבצעים lazy insert. הוספה של node ישירות לרשימת השורשים ללא לינקס וקאטס.

פוטנציאל:

אנו מכניסים m nodes לרשימת השורשים כאשר לכל אחד מהם mark = false. כלומר הפוטנציאל יהיה מספר העצים, שהוא מספר ההכנסות במקרה זה.

בסה"כ: Potential = #T + 2*#marks = m

התשובות כמובן תואמות לטבלא.

Sequence 2:

m	Run-Time (in miliseconds)	totalLinks	totalCuts	Potential
1000	7	991	0	6
2000	14	1990	0	6
3000	12	2990	0	7

זמן ריצה אסימפטוטי:

למעשה ניתן לנתח בנפרד את שלב ההכנסה ושלב המחיקה, כאשר הניתוח של ההכנסות מתבצע בדיוק כמו בסדרה הראשונה. והינו בסיבוכיות **O(m)**

מחיקת מינימום היא כזכור **amortized O(logn)**. ולכן עבור **m/2** של **deleteMin** פעולות כאלו על עץ שיש בו- **m** איברים: **O(mlogm)**

ובסה"כ נקבל סיבוכיות כוללת של **O(mlogm)**

אציין שזמן הריצה שקיבלנו במדידות אינו משקף את הניתוח מכיוון שהשתמשנו במספרים יחסית קטנים. עבור הרצות על מספרים גדולים יותר התוצאות נהיות יותר עקביות עם הניתוח, בפרט זמן הריצה עבור קלטים קטנים יותר הוא קטן יותר מה שלא קרה עבור 2000 -> 3000. בדקנו על קלטים גדולים יותר ושם ראינו את זמן הריצה המצופה.

מספר פעולות לינק וקאט:

ראינו כי בהכנסות לא מתבצעים קאטס ולינקס ולכן נתייחס רק למחיקות.

כאמור לא מתבצעים קאטס במחיקת מינימום. ולכן מספרם 0 (תואם בתוצאות)

מספר פעולות הלינק:

consolidating אשר מורכב מסדרה של לינקס הוא כאמור הפעולה היקרה ב**deletemin**.

לכן מספר הלינקס הוא אמורטייזד דומה לסיבוכיות של מחיקת מינימום והינו **O(logn)**.

סה"כ נקבל **O(mlogm) links**.

והדבר עקבי עם התוצאות: **(log base10)**

log(3000) * 3000/2 = 5215 ואנחנו קיבלנו**2990**

3301 = 2000/2 * (2000)log ואנחנו קיבלנו **1990**.

1500 = 1000/2 * (1000)log ואנחנו קיבלנו**991**

אם היינו לוקחים קבוע **$2*\log_{10} 2=C$** היינו מקבלים חסם עליון של **c*mlogm**

potential :

כאמור גם במקרה זה אין **cuts** ולכן אין שינוי במספר ה**marks** שנשאר עקבי על 0.

לכן הפוטנציאל הוא כמספר העצים ומספר זה הוא **O(logm)**.

בסוף הפעולה כאמור יהיו לנו בעץ **m\2** איברים. והפוטנציאל שיוצא במדידות מתאים:

לדוגמא עבור **M** = 1000 , 2000, 3000 יצא לנו בקירוב:

log 500 = 9 ולנו יצא פוטנציאל 6.

log 1000 = 10 ולנו יצא פוטנציאל 6.

log 1500 = 10.5 ולנו יצא פוטנציאל **7**.

וזה גם תואם את המדידות שלנו כי קיבלנו שהפוטנציאל חסום על ידי החסם התיאורטי.