**תיעוד**

***מחלקת FibonacciHeap***

public boolean empty() – הפונקציה בודקת האם הערימה ריקה ע"י השוואת השדה של גודל רשימת העצים. סיבוכיות O(1).

public HeapNode insert(int key) – הפונקציה מוסיפה צומת חדשה לערימה ומעדכנת את המינימום. סיבוכיותO(1).

private int numberOfTrees() – הפונקציה מחזירה את מספר העצים בערימה ע"י שימוש בפונקציה size של HeapList. סיבוכיות O(1).

public HeapNode findMin() – מחזירה את הצומת המינימלי בערימה ע"י החזרה של השדה min אם קיימים איברים בערימה, אחרת מחזירה null. סיבוכיות O(1).

public void meld(FibonacciHeap heap2) – הפונקציה ממזגת 2 ערימות לערימה אחת ע"י שרשור 2 הרשימות שמכילות את העצים ברשימה באמצעות meld ולאחר מכן מעדכנת את המינימום ואת מספר העצים. סיבוכיות O(1).

private void increaseSize(int size2) – מגדילה את הערך של מספר העצים בערימה ב-size2. סיבוכיות O(1).

public int size() – מחזירה את מספר העצים ברשימה ע"י החזרה של השדה size. סיבוכיות O(1).

***מחלקת HeapNode***

private HeapNode parent – שדה שמכיל מצביע לאבא של הצומת

private HeapList children – שדה שמכיל מצביע לרשימת הילדים של הצומת

private HeapNode next – שדה המכיל מצביע לאיבר הבא ברשימה

private HeapNode prev – שדה המכיל מצביע לאיבר הקודם ברשימה

public int key – שדה המכיל את המפתח של הצומת

private boolean mark – שדה שהוא אינדיקטור האם הצומת מסומן (נמחק לו ילד)

\*להוסיף תיעוד של הפונקציות כאן, בעיקר אלו שקשורות לסימון\*

***מחלקת HeapList***

מחלקה שמממשת רשימה מעגלית דו-כיוונית בסיבוכיות שנלמדה בכיתה בתחילת הקורס. הכנסה ומחיקה של צומת ב-O(1) (כשהצומת מועבר כפרמטר) ושרשור של שתי רשימות ב-O(1). המחלקה מממשת גם את iterator כך שניתן לעבור עליה בלולאה.

**מדידות**

תוצאות של סדרת הפעולות הראשונה:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **m** | **Run-Time (ms)** | **totalLinks** | **totalCuts** | **Potential** |
| 1000 | 0.751076 | 0 | 0 | 1000 |
| 2000 | 1.351584 | 0 | 0 | 2000 |
| 3000 | 1.975717 | 0 | 0 | 3000 |

הסבר:

לפי מה שלמדנו פעולת insert בערימת פיבונאצ'י לוקחת amortized(O(1)), לכן אנו מצפים שסדרת פעולות תיקח O(m). זה אכן מתקיים גם כאן, בכל פעם שהגדלנו את m ב-1000 גם זמן הריצה גדל בקצב לינארי. בנוסף מתקיים:

Potential = #trees + 2 #marked

נשים לב שבסדרת פעולות זו אין צמתים שהפכו להיות מסומנים בגלל שלא בוצעו פעולות delete ולכן הפוטנציאל הוא בדיוק מספר העצים.

מספר ה-links וה-cuts הוא 0 כי לא בוצעו פעולות delete או decrease-key.

תוצאות של סדרת הפעולות השנייה:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **m** | **Run-Time (ms)** | **totalLinks** | **totalCuts** | **Potential** |
| 1000 | 11.252029 | 991 | 0 | 6 |
| 2000 | 14.572277 | 1990 | 0 | 6 |
| 3000 | 18.408404 | 2990 | 0 | 7 |

הסבר:

פעולת insert לוקחת amortized(O(1)) (כמו קודם) ו-delete-min לוקחת amortized(O(logm)) בגלל שה-deleteMin הראשון יתבצע ב-O(m) כי מבצעים consolidating על m עצים בגודל 1. כעת נקבל ערימה בינומית. מעתה והלאה שאר פעולות ה-deleteMin יתבצעו ב-O(logm) משום שיוצרים מערך בגודל O(logm) בזמן ה-consolidating ועוברים עליו כדי למצוא את המינימום החדש, לכן בסה"כ זמן הריצה של סדרת הפעולות הוא m\*1+1\*m+((m/2)-1)\*logm=O(mlogm).

מספר פעולות ה-link המבוצעות הוא O(m) בגלל שב-deleteMin הראשון בתא הראשון נבצע m/2 פעולות link, בתא השני m/4, בשלישי m/8 וכו'. בשל סדר ההכנסה, האיברים עם המפתחות הגדולים נמצאים בעצים הגדולים כך שפעולות ה-deleteMin הבאות ימחקו את השורש של העץ המינימלי ולכן לא יהיו יותר פעולות link. סה"כ O(m).

מספר פעולות ה-cut המבוצעות הוא 0 כי לא התבצעו פעולות decreaseKey.

אין צמתים מסומנים ולכן הפוטנציאל הוא למעשה מספר העצים שהוא O(log(m)) משום שביצענו m/2 פעולות deleteMin והערימה היא ערימה בינומית בסוף סדרת הפעולות ומספר העצים בערימה הוא מספר הפעמים שהספרה 1 מופיעה בייצוג הבינארי של כמות הצמתים, כלומר מספר העצים מסדר לוגריתמי, וזה log(m/2) = logm-log2=O(logm).