**FibonacciHeap**

**תיעוד הקוד**

**מחלקת FibonacciHeap**

מחלקה זו תממש ערימת פיבונא'צי מינימום. במימוש זה נניח שהאיברים בערימה הינם מספרים שלמים שונים זה מזה ונייצג את הערימה באמצעות רשימה מקושרת דו כיוונית ומעגלית של אובייקטים מטיפוס HeapNode.

שדות:

* first – שדה השומר מצביע לצומת הראשונה ברשימת העצים.
* min – שדה השומר מצביע לצומת בעלת המפתח המינימלי בעץ.
* size – שדה השומר את מספר האיברים בערימה בכל שלב בערימה.
* num\_trees – שדה השומר את מספר העצים בערימה בכל שלב בערימה.
* num\_marked – שדה השומר את מספר הצמתים המסומנות בכל שלב בערימה.
* link\_counter – שדה סטטי השומר את מספר החיבורים שעשינו לאורך כל התוכנית.
* cuts\_coutner – שדה סטטי השומר את מספר החתכים שעשינו לאורך כל התוכנית

מתודות:

* **FibonacciHeap()**  – בנאי המחלקה.

**אלגוריתם:** מאתחל את שדות size, num\_trees, num\_marked להיות 0. שאר השדות מאותחלים באופן אוטומטי להיות הערכים הדיפולטים.

**סיבוכיות:** הפונקציה מבצעת מספר קבוע של פעולות ולכן סיבוכיותה היא .

* **isEmpty()**– פונקצית מופע על אובייקט מסוג FibonacciHeap, תחזיר T אם הערימה ריקה, אחרת F.

**אלגוריתם:** הפונקציה תחזיר True אם מספר הצמתים בערימה שווה לאפס. אחרת, תחזיר False.

**סיבוכיות:** הפונקציה מבצעת מספר קבוע של פעולות ולכן סיבוכיותה היא .

* **insert(int key)**– פונקצית מופע על אובייקט מסוג FibonacciHeap, תכניס לערימה מפתח בעל ערך key.

**אלגוריתם:** הפונקציה יוצרת אובייקט חדש מטיפוס HeapNode שערך שדה המפתח שלו הוא key. אם הערימה ריקה, נכניס את הצומת החדשה לרשימה ונעדכן את שדות ה next וה prev שלה להצביע על עצמה.בנוסף, נעדכן את שדה ה min להצביע על צומת זו. אם הערימה אינה ריקה, נכניס את הצומת לתחילת הערימה נעדכן את שדות הprev וה next בהתאם. כמו כן, במקרה זה יש לבדוק האם המפתח קטן מערך המפתח המינמלי, אם כן נעדכן את שדה min להצביע על הצומת החדשה שהוכנסה. לבסוף נגדיל את שדה size ו num\_trees ב 1.

**סיבוכיות:** הפונקציה מבצעת מספר קבוע של פעולות ולכן סיבוכיותה היא .

* **insertTreeAtStart(HeapNode rootToInsert)**– פונקצית מופע על אובייקט מסוג FibonacciHeap, תכניס לתחילת הערימה את צומת rootToInsert.

**אלגוריתם:** בדומה לפונקציה insert, הפונקציה בודקת האם הערימה ריקה או לא לפני ההכנסה. אם כן, נכניס את הערימה, נעדכן את שדה min להצביע עליה ונעדכן את שדות prev, next להצביע על הצומת עצמה. אחרת, נכניס את הצומת בתחילת הערימה, נעדכן את next, prev ובהתאם ונבדוק האם המינימום של הערימה השתנה. כמו כן נעדכן את num\_trees לגדול ב 1.

**סיבוכיות:** הפונקציה מבצעת מספר קבוע של פעולות ולכן סיבוכיותה היא .

* **deleteMin() -** פונקצית מופע על אובייקט מסוג FibonacciHeap, פונקציה זו תמחק את הצומת בעלת המפתח המינימלי בערימה.

**אלגוריתם:** ראשית נבדוק אם הערימה ריקה, אם כן נסיים את הפונקציה. לאחר מכן נבדוק האם הפונקציה מכילה צומת אחת בלבד, אם כן – נעדכן את השדות הרלוונטים ונוציא את הצומת מהרשימה ע"י עדכון first להצביע על null ואתחול השדות. אחרת, נתחיל בביצוע תהליך מחיקת המינימום והכנסת ילדיו לרשימה. נחלק למקרים:

**Case A** : האיבר המינימלי הינו תחת שדה “first” בערימה – אם מס' העצים בערימה הוא סה"כ 1, אזי בשלב זה נידע בוודאות שלאיבר המינמלי יש ילדים (אחרת גודל הערימה היה 1), ולכן נגדיר את השדה first להיות הילד של המינימום. אחרת, נבדוק אם יש למינימום ילדים או לא, נכניס אותם לערימה ונשנה את המצבעים בהתאם.

**Case B**: האיבר המינימלי אינו האיבר ה"first" בערימה – ננתק אותו מאחים שלו השורשים, ובמידה ויש לו ילדים, נכניס אותם בין 2 האיברים שהאיבר המינימלי היה בינהם.  
  
לבסוף, נעדכן את מס' העצים בעץ ע"י כך שנוסיף לו את rank האבא שמחקנו, נעדכן את size להיות קטן ב 1 ונשלח את הערימה כולה לפונקציה consolidate שתממש את תהליך ה successive linking. (בתוך הconsolidate נדאג שההורים של השורשים החדשים שהכנסנו, יהיו null)

**סיבוכיות:** כאשר הערימה ריקה או בגודל 1 לפני המחיקה נבצע מספר קבוע של פעולות ולכן הסיבוכיות תהיה במקרים אלה . בשאר המקרים במהלך הכנסת הילדים של הצומת הנמחק לרשימת השורשים, נבצע שינויי מצביעים בלבד ומס' קבוע של פעולות שגם כן יעלו . לבסוף נקרא לפונקציה consolidate שסיבוכיות ה WC שלה היא . לכן סה"כ קיבלנו שסיבוכיות ה WC של הפונקציה היא . סיבוכיות consolidate בAmortized הינה ולכן סיבוכיות פונקציה זו באמורטייזד הינה גם כן . (מפורט בהמשך).

* **consolidate()**– פונקצית מופע על אובייקט מסוג FibonacciHeap, הפונקציה תבצע תהליך successive linking לערימה.

**אלגוריתם:** נאתחל מערך בגודל לפי בסיס יחס הזהב שישמש כתאים בהם נשתמש בביצוע האלגוריתם. נעבור על רשימת העצים שיש בערימה. עבור כל עץ נבדוק את הדרגה שלו באמצעות שדה rank כדי לקבוע את דרגת העץ. נבדוק האם התא באינדקס rank של העץ תפוס – אם לא תפוס, נכניס את העץ לתא rank במערך. אם כן תפוס – נבצע link בינו לבין העץ שבתא באמצעות הפונקציה link, נעדכן את דרגת העץ שהתקבל להיות rank+1 ונכניס אותו לתא באינדקס rank+1 תוך אתחול התא באינדקס rank. לבסוף נעבור על כל המערך שלנו מהסוף להתחלה ונשרשר את העצים לפי הסדר לתחילת ערימה חדשה ונעדכן את הערימה שלנו להצביע על הערימה החדשה. במהלך המעבר על המערך, נעדכן את השורשים החדשים בתכונות הנדרשות ( אם יש להם אבא, אז נגדיר אותו null ואם הם מסומנים נבטל את הסימון).

**סיבוכיות:** לפני ביצוע ה consolidate יכולים להיות בערימה n עצים שונים, לכן נצטרך לסרוק n עצים. עבור כל עץ נעשה לכל היותר מספר קבוע של פעולות (בדיקה האם התא של דרגת העץ תפוס, חיבור(כפי שנראה בהמשך זה סיבוכיות והכנסה לתא אחר במקרה הצורך). לכן סה"כ נקבל שסיבוכיות WC של הפעולה היא . כעת נסביר למה סיבוכיות האמורטייז היא : ההנחות בהן השתמשנו בכיתה כדי להוכיח בכיתה שהאמורטייז הוא אכן היו:

1. מספר הילדים של השורש שנמחק הוא לכל היותר וזה אכן מתקיים במימוש שלנו שכן אנו שומרים על האינוורינטה שבסוף כל successive linking יש רק עץ אחד לכל היותר מכל דרגה, כאשר הדרגה המקסימלית היא . לכן, בשום שלב בתוכנית לא נקבל עץ שבו לשורש יש יותר ילדים מ .
2. פעולת ה link עבור 2 עצים היא וזה אכן מתקיים כפי שהסברנו בניתוח הסיבוכיות של פונקציה זו.
3. מספר העצים לאחר ביצוע הפעולה יהיה לכל היותר וזה אכן קורה מכיוון שאנחנו משמרים את האינוורינטה של ערימת פיבונאצ'י לאורך התוכנית לפיה, העץ בדרגה הגבוהה ביותר הוא מדרגה

הנחות אלה מתקיימות ולכן ניתן להשתמש בהוכחה כפי שהוכחנו בכיתה באמצעות הגדרת פונק' פוטנציאל כך ש: Potential = Number of Trees.

* **link(HeapNode H1, HeapNode H2)**– פונקצית סטטית המקבלת שני אובייקטים מטיפוס HeapNode עם אותה הדרגה ומחזירה טיפוס HeapNode שהוא החיבור של העצים הנ"ל.

**אלגוריתם:** נשווה את השורשים של שני העצים כדי לקבוע מי יהיה השורש של העץ החדש (כדי לשמור על האינוורנטה שהשורש יהיה האיבר הכי קטן בעץ), נסמן את עץ זה כ smaller ואת השני כbigger . נעדכן את bigger להיות הבן של smaller ונעדכן את השדות של הבנים כדי ליצור ערימה מקושרת דו כיוונית ומעגלית בין הילדים הישנים והחדשים של smaller. לבסוף נעדכן את הדרגה של השורש של smaller ונחזיר את smaller.

**סיבוכיות:** הפונקציה מבצעת מספר קבוע של פעולות ולכן סיבוכיותה היא .

* **findMin() -** פונקצית מופע על אובייקט מסוג FibonacciHeap, תחזיר את הצומת עם הערך המינימלי בערימה.

**אלגוריתם:** הפונקציה תחזיר את הצומת עליה שדה min מצביע.

**סיבוכיות:** הפונקציה מבצעת מספר קבוע של פעולות ולכן סיבוכיותה היא .

* **meld(HeapNode heap2)**– פונקצית מופע על אובייקט מסוג FibonacciHeap, תקבל ערימה נוספת ותבצע מיזוג של שתי הערימות.

**אלגוריתם:** אם heap2 ריקה – סיימנו. אם הערימה שעליה מבצעים את המיזוג ריקה – נעדכן את הערימה הזו להצביע על heap2 תוך עדכון השדות הרלוונטים וסיימנו. אחרת, נשרשר את ערימת העצים של heap2 אחרי הרשימת עצים של הערימה הנוכחית. כאשר נעשה את זה נשווה בין הצמתים המינימלים שלהם כדי לתחזק את שדה ה min בעת ביצוע meld. כמו כן נעדכן את שדה ה size ושדה ה num\_trees להיות הסכום של שתי הערימות.

**סיבוכיות:** הפונקציה מבצעת מספר קבוע של פעולות ולכן סיבוכיותה היא .

* **Size() -** פונקצית מופע על אובייקט מסוג FibonacciHeap, תחזיר את מספר הצמתים בערימה.

**אלגוריתם:** הפונקציה תחזיר את הערך השמור בשדה size של הערימה.

**סיבוכיות:** הפונקציה מבצעת מספר קבוע של פעולות ולכן סיבוכיותה היא .

* **countersRep() -** פונקצית מופע על אובייקט מסוג FibonacciHeap, תחזיר מערך של מספרים המוגדר כך: באינדקס ה i של המערך יהיה שמור מספר העצים מדרגה i בערימה.

**אלגוריתם:** ראשית נרוץ על כל העצים בערימה כדי למצוא את העץ הדרגה המקסימלית. לאחר מכן נגדיר מערך של int באורך הדרגה של העץ המקסימלי (ועוד אחד). כעת נרוץ שוב על רשימת העצים, על כל עץ שנעבור נסמן את הדרגה שלו ב rank ונעדכן את התא במיקום ה rank לגדול ב 1.

**סיבוכיות:** הפונקציה מבצעת לולאה על כל רשימת השורשים בערימה פעמיים. בערימה יכולים להיות לכל היותר n שורשים בזמן נתון ולכן סיבוכיות ה WC תהיה .

* **Delete(HeapNode X) –** פונקציית מופע על אובייקט מסוג FibonacciHeap, הפונקציה מוחקת את הצומת x מהערימה ולא מחזירה כלום.

**אלגוריתם:** נממש את המחיקה באמצעות ביצוע decreaseKey לצומת המיועדת למחיקה כך שערך האיבר שלה יהיה הקטן ביותר בעץ, ואז נבצע deleteMin. נסמן בk את ערך המפתח המינימלי, ובאמצעותו נחשוב את הdelta שאיתה אנו רוצים לשלוח את הצומת המיועד למחיקה לפונקציה decreaseKey. הdelta תהיה ערך האיבר פחות ערך האיבר המינימלי בעץ + 1. ע"י חישוב זה, ערך איבר X החדש בפונקציית decreaseKey יהיה המינימום בעץ ולכן יימחק כאשר נפעיל deleteMin.

**סיבוכיות:** הפונקציה delete משתמשת ב2 פונקציות: decreaseKey וdeleteMin שהסיבוכיות שלהן (כפי שיפורט בהמשך) היא עבור Delete-min ו עבור Decrease-Key (בזמני Amortized) ולכן סה"כ סיבוכיות פונקציה Delete הינה O(logn) באמורטייזד. במקרה הגרוע, deleteMin וdecreaseKey הן ולכן סיבוכיות פונקציה זו במקרה הגרוע הינה גם כן .

* **decreaseKey(HeapNode X, int delta) –** פונקציית מופיע של אובייקט מסוג FibonacciHeap. הפונקציה מקבלת צומת בעץ, ומספר שלם delta. הפונקציה מקטינה את ערך מפתח הצומת שקיבלה בערך המספר השלם delta.

**אלגוריתם:** ראשית, נקטין את ערך המפתח של הצומת שקיבלנו בdelta. נחלק ל3 מקרים:  
Case A: הצומת שקיבלנו הינה שורש - במידה וכן, נבדוק אם המפתח המעודכן קטן מהמינימום הישן של העץ. אם כן, נעדכן את המינימום וניצא מהפונקציה.

Case B: לאחר הפחתת ערך המפתח, ערך מפתח ההורה של הצומת עדיין קטן מהמפתח של הצומת המעודכנת. במקרה כזה, אינווריאנטת העץ לא הופרה, וניצא מהפונקציה.

Case C: הצומת שלה עידכנו את ערך המפתח אינה שורש, ולאחר העדכון הערך החדש של הצומת קטן משל אבא שלו – נקרא לפונקציה הרקורסיבית cascading\_cut עם הצומת המעודכנת ואבא שלה.

**סיבוכיות:** סיבוכיות Case A וCase B הינה שכן כלל הפעולות המבוצעות במקרים אלו הינן פעולות בעלות זמן קבוע. בCaseB אנו קוראים לפונקציה cascading\_cut שנעשית בסיבוכיות של אמורטייזד (יפורט בהמשך) ו במקרה הגרוע. לכן, סיבוכיות פונקציה זו הינה אמורטייזד ו במקרה הגרוע.

* **cascading\_cut(HeapNode X, HeapNode Y) –** פונקציית מופע רקורסיבית של אובייקט מסוג FibonacciHeap. הפונקציה מקבלת 2 צמתים כך ש Y אבא של X וחותכת את X מאבא שלו Y. במהלך מימוש הפונקציה נשמור על האינווריאנטה שלצומת בעץ (שאינה שורש) מותר לאבר לכל היותר בן אחד. אם נחתוך בן של צומת שכבר איבדה בן, נחתוך גם אותה. שורשי הערימה תמיד לא מסומנים (כמי שאיבדו כבר ילד), כי אין מאיפה לחתוך אותם וכן מכיוון שלאחר ששורש מאבד ילד דרגת העץ יורדת. (לעומת חיתוכים "בפנים העץ" שבהן דרגת העץ נותרת גבוהה, אך צמתים נאבדים (שזה החלק שפוגע לנו באנליזיה וחסימת הדרגות ע"י logn).

**אלגוריתם:** נשלח לפונקציה cut את X וY, ולאחריה נקבל ערימת פיבונאצ'י חדשה שבה X כבר לא בן של Y, והוא נהיה עץ "עצמאי" כחלק מהרשימה של העצים המרכיבים את הערימה (פירוט רחב יותר בפונקציה cut). במידה וy הוא שורש, נסיים את הרקורסיה, אחרת, נבדוק אם Y לא מסומן (עדיין לא איבד ילד), במידה ואכן הוא אינו מסומן, נסמן אותו, נעלה את הcounter שסופר לנו את מס' הצמתים המסומנים, ונסיים את הרקורסיה. אחרת (הוא כן מסומן), נבצע קריאה רקורסיבית עם y ואבא שלו (נחתוך כעת את y מאבא שלו).

**סיבוכיות:** במקרה הגרוע, כל הצמתים מעלה בעץ ועד השורש מסומנים כמי שאיבדו ילד, ומתבצע מפל חיתוכים מעלה ועד השורש. ראינו בתרגול, שבערימת פיבונאצ'י עומק העצים לא חסום ולכל n קיימת סדרת פעולות תקינות על ערימת פיבונ'צי שיוצרת עץ "שרוך" באורך n. מכאן, שבמקרה הגרוע ייתבצעו כn קריאות רקורסיביות, שבכל איטרציה מבוצעות פעולות בעלות זמן קבועה (כולל הפונקציה cut שהסיבוכיות שלה מפורטת בהמשך), כלומר סה"כ הסיבוכיות הWC היא . מכיוון שבכל שלב שמרנו על האינווריאנטה שבה כל צומת מאבד לכל היותר ילד אחד ומכאן למה 1 למה 2 ולמה 3 שהוכחנו בכיתה מתקיימות. ובעזרת פונקציית הפוטנציאל (מס' הצמתים המסומנים) ראינו כי מספר החיתוכים שמתבצע בAmortized הוא 2, כלומר . ולכן סיבוכיות הפונקציה בAmortized היא .

* **cut(HeapNode X, HeapNode Y) –** פונקציית מופע על אובייקט מסוג FibonacciHeap. הפונקציה מקבלת 2 צמתים X וY (שY אבא של X), חותכת את X מY, ומוסיפה את X כשורש נוסף בערימה.

**אלגוריתם:** תחילה, נעלה את counter הcuts ב1. נגדיר את אבא של X להיות null, כי אנו רוצים לשים אותו כשורש בערימה. במידה וX היה מסומן, נוריד את קאונטר הצמתים המסומנים שלנו. כמו כן, נהפוך את X להיות לא מסומן (ערך הmark 0) כי הוא יהיה שורש. נחלק למקרים:  
Case A: X הינו בן יחיד – אזי נכדיר את הילד של Y להיות null ונשלח את X לפונקציה insertTreeAtStart שמכניסה אותו כשורש לערימה שלנו.

Case B: לX יש "אחים" –נבצע תהליך ניתוק של X: נגדיר את הילד של Y להיות הnext של X, נגדיר את הnext של האח הprev של X להצביע על הnext של X, ונגדיר את הprev של הnext של X להצביע על הprev של X. לאחר הניתוקים נשלח את X לפונקציה insertTreeAtStart.

**סיבוכיות:** כמתואר למעלה, כלל הפעולות המבוצעות הינן בסיבוכיות זמן קבועה, וכן הפונקציה insertTreeAtStart הינה מסיבוכיות (כמפורט בתיעוד הפונקציה), לכן סה"כ הסיבוכיות של הפעולה היא .

* **potential() –** הפונקציה היא פונקציית מופע של אובייקט מסוג FibonacciHeap. הפונקציה מחזירה את ערך הפוטנציאל הנוכחי של הערימה, כפי שהגדרנו בשיעור

**אלגוריתם:** הפונקציה מחזירה את מספר העצים + פעמיים מספר הצמתים המסומנים (#trees + 2\*#marked)

**סיבוכיות:** סיבוכיות הפונקציה הינה . מבוצעות פעולות בעלות זמן קבוע על משתנים מעודכנים שתחזקנו במהלך כל התוכנית.

* **totalLinks() –** פונקציה סטטית שמחזירה את מספר כל פעולות הלינק שבוצעו מתחילת ריצת התוכנית.

**אלגוריתם:** הפונקציה מחזירה את המשתנה הסטטי links\_counter שתחזקנו במהלך כל התוכנית ומכיל את מס' כל פעולות הלינק שבוצעו.

**סיבוכיות:** סיבוכיות הפונקציה היא שכן מחזירה משתנה בלבד.

* **totalCuts() –** פונקציה סטטית שמחזירה את מספר כל פעולות הcut שבוצעו מתחילת ריצת התוכנית.

**אלגוריתם:** הפונקציה מחזירה את המשתנה הסטטי cuts\_counter שתחזקנו במהלך כל התוכנית ומכיל את מס' כל פעולות החיתוך שבוצעו.

**סיבוכיות:** סיבוכיות הפונקציה היא שכן מחזירה משתנה בלבד.

* **kMin(FibonacciHeap H, int k) –** פונקציה סטטית שמקבלת עץ בינומי תקין ומספר k ומחזירה מערך ממוין בגודל k המכיל את k האיברים הכי קטנים בעץ.

**אלגוריתם:** נאתחל ערימת פיבונאצ'י ריקה שתשמש לאורך הפונקציה כערימת עזר. נסמן את המערך היכיל את k האיברים הכי קטנים כ result. אם k=0 נחזיר מערך ריק. אחרת, נגדיר את השורש להיות האיבר הראשון ב result ונכניס צמתים חדשים היכילו את ערכי כל הבנים של השורש לערימת עזר. לכל צומת שכזה נעשה שיבוט (לשיבוט יהיה ערך מפתחות שווה לצומת אחרת אך הוא אובייקט שונה). נעדכן את שדה prevMe של כל שיבוט להצביע על הצומת המקורית בעץ הבינומי כדי שנוכל לגשת לצמתים המקוריים לאורך האלגוריתם. כעת, עבור i=1 עד k נבצע את התהליך הבא: בכל פעם נמחק את הצומת המינימלית מהערימת עזר,נכניס את ערך המפתח שלה למקום ה i במערך ונכניס לערימת עזר שיבוטים של כל הבנים שלה מעץ המקורי (אם יש ילדים), כך נבטיח שהאיבר הבא בתור במערך הממוין יהיה חייב להיות בערימת עזר שלנו (מכיוון שהאיבר הבא אחרי שמחקנו את האיבר ה j יכול להיות או אח של j או בן של j. אם j בערימה זה אומר שהאחים שלו גם בערימה לכן נותר רק להכניס את הבנים).

**סיבוכיות:** מספר האיברים בערימת עזר בכל שלב יהיה תמיד קטן ממש מ מכיוון שכל איטרציה (יש K כאלה) ייתכן ונכניס בנים של צומת מסוימת כאשר מספר הילדים יהיה חסום תמיד ע"י . בנוסף, בכל איטרציה מבצעים delete-min. סיבוכיות ה WC של delete-min כפי שהסברנו בניתוח הסיבוכיות שלה, יהיה כמספר העצים בערמה. לאחר הכנסה של ילדים, נקבל שיהיה לכל היותר (ואפילו קטן ממש מ -)

. לכן ביצוע delete-min יעלה ולאחר המחיקה בעקבות תהליך הConsolidating נקבל ערימת פיבונאצ'י חוקית. לכן ניתן להסיק שבכל רגע נתון, מספר העצים יהיה פחות מ -

לסיכום, נבצע k איטרציות כאשר בכל איטרציה נבצע הכנסה של לכל היותר ילדים ומחיקת המינימום מרשימת העזר - . ולסיכום נקבל:

**מחלקת HeapNode**

מחלקה מקוננת של FibonacciHeap שנועדה לממש צומת אשר נמצא בערימה.

שדות:

* key - שדה מסוג int המכיל את מפתח כל צומת.
* rank - שדה מסוג int המכיל את דרגת כל צומת בכל שלב.
* mark - שדה מסוג int שמקבל 0\1 ומסמן אם הצומת מסומנת (1) או לא מסומנת (0) (מסומן = איבד ילד).
* child – שדה השומר מצביע לילד של צומת. אם לא קיים ילד, יכיל null.
* next – שדה השומר מצביע ל"אח" הבא (ימני) של צומת (אח = לשניהם מצביע לאותו אבא)
* prev – שדה השומר מצביע ל"אח" הקודם (שמאלי) של צומת.
* parent – שדה השומר מצביע ל"אבא" של צומת. אם הצומת היא שורש יצביע לnull.
* prevMe – שדה השומר מצביע לצומת המקורי לפני השיבוט (משמש בפונקצית kMin).

מתודות:

* **HeapNode(int key) –** פונקציה זו היא בנאי המחלקה. מקבלת מפתח מסוג int.

**אלגוריתם:** הפונקציה תאתחל את המפתח של הצומת להיות key, את הrank של הצומת להיות 0, ואת שדה mark להיות 0 (צומת לא מסומן). שאר השדות יאותחלו לערכים הדיפולטים שלהם.

**סיבוכיות:** הפונקציה מבצעת מס' קבוע של פעולות קבועות ולכן הסיבוכיות היא .

* **getKey()-** פונקציית מופע על אובייקט מסוג HeapNode. הפונקציה מחזירה את ערך המפתח של הצומת.

**אלגוריתם:** הפונקציה מחזירה את ערך המפתח של הצומת הנוכחית. (פונקציית getter)

**סיבוכיות:** הפונקציה מבצעת פעולות בודדות קבועות בעלות של .

* **isRoot() –** פונקציית מופיע על אובייקט מסוג HeapNode. הפונקציה מחזירה True אם הצומת היא שורש, אחרת False.

**אלגוריתם:** הפונקציה בודקת אם אבא של הצומת הינו null. אם כן, הצומת שורש ומוחזר True. אחרת False.

**סיבוכיות:** הפונקציה מבצעת פעולות בודדות קבועות בעלות של .

מדידות

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **m** | **Run-Time (milliseconds)** | **totalLinks** | **totalCuts** | **Potential** |
|
| 1024 | 0.5739 | 1023 | 18 | 19 |
| 2048 | 1.085 | 2047 | 20 | 21 |
| 4096 | 2.064 | 4095 | 22 | 23 |

שאלה 1

א. בשלב הראשון נכניס את כל המספרים מ m עד 0 ,סה"כ m+1 איברים כאשר כל הכנסה היא זמן קבוע. סיבוכיות חלק זה היא .

בשלב השני נמחק את האיבר המינימלי. בסוף שלב זה נקבל ערימה בינומית תקינה עם עץ יחיד בדרגה . סיבוכיות חלק זה היא , מהסיבה שנבצע m-1 לינקים כפי שנסביר בסעיף ב.

בשלב השלישי מבצעים decrease key לאינדקסים ספציפיים כך שסה"כ תיווצר שרשרת של צמתים marked בכל הענף השמאלי של העץ, מהשורש (לא כולל) עד הצומת השמאלית התחתונה ביותר (לא כולל). כל חתך כזה ייקח , כי לא תתרחש מפולת חתכים (כל פעם חותכים ילד של צומת אחרת). סה"כ יש logm -2 חתכים כאלה לכן סיבוכיות שלב זה היא .

לבסוף, נבצע decrease key לאיבר במקום ה m-1, כלומר האיבר התחתון ביותר בענף השמאלי ביותר של העץ. כאשר נבצע cut לאיבר זה מאבא שלו, תיווצר מפולת חתכים שתמשך עד השורש, כי בשלב הקודם ווידאנו שכל הצמתים בשרשרת זו marked. לכן סיבוכיות פעולה זו תהיה כעומק העץ - logm (מכיוון שהתחלנו עם עץ בינומי תקין, לו עומק של logm, והחתכים שביצענו עד כה, לא הקטינו את העומק). לכן סיבוכיות פעולה זו היא ..

**סה"כ קיבלנו שזמן הריצה האסימפטוטי של רצף פעולות זה הוא .**

ב.

Cut- כפי שהסברנו בסעיף א', מבצעים חתכים בשלב השלישי, ו חתכים בשלב הרביעי, סה"כ קיבלנו אסימפטוטית חתכים בהתאמה מדויקת לתוצאות הטבלה.

Link – נוכיח באינדוקציה מדוע עבור עץ בדרגה k דרושים לינקים, בהתאם לתוצאות שקיבלנו:

בסיס האינדוקציה – עבור k=1 נצטרך לבצע לינק בין 2 עצים בדרגות 0, כלומר סה"כ לינק 1 כנדרש

הנחת האינדוקציה - עבור עץ בדרגה k דרושים לינקים

צעד האינדוקציה – עבור יצירת עץ בדרגה k+1 נצטרך ליצור עוד עץ בדרגה k, כלומר עוד לינקים (מהנחת האינדוקציה) ועוד לינק 1 לחבר בין 2 העצים (מהגדרת יצירת עץ בינומי), כלומר סה"כ הלינקים שנדרשו עבור יצירת העץ בדרגה k+1 הם: כנדרש!

לכן, עבור עץ בדרגה נקבל שלוקח לינקים כפי שקיבלנו בטבלה ולכן נבצע לינקים.

ג. עלות פעולת הdecrease-key היקרה ביותר תתרחש במפולת החתכים הגדולה ביותר, כלומר כפי שהוסבר בסעיף א', תתרחש בdecrease-key(m-1) כאשר שאר הצמתים במסלול השמאלי ביותר של העץ מסומנות, שם תתרחש מפולת חתכים באורך של חתכים, כלומר עלות פעולה זו היא .

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **m** | **Run-Time (milliseconds)** | **totalLinks** | **totalCuts** | **Potential** |
|
| 1000 | 0.6439 | 1891 | 0 | 6 |
| 2000 | 0.9454 | 3889 | 0 | 6 |
| 3000 | 1.5371 | 5772 | 0 | 7 |

**שאלה 2**

א. תחילה מבצעים m הכנסות לערמה, עלות פעולות אלו היא . לאחר כן, deleteMin הראשון, יעלה מכיוון שיש m עצים, בערימה, ובסופו נקבל ערימת פיבונאצ'י עם לכל היותר עצים, ולכן שאר פעולות הdeleteMin יהיו . לכן סה"כ הסיבוכיות תהיה:

ב. מספר פעולות הLink המבוצעות כפונקציה של m הוא: .

עלות פעולת deleteMin נקבעת ביחידות לפי כמות הלינקים שנבצע במהלכה. לכן, כפי שהסברנו עלות הdeleteMin עבור כל רצף הפעולות הוא ולכן בהתאמה מס' הלינקים יהיה גם כן .

פעולת cuts לא מתבצעות כלל, שכן לא נעשית שום פעולת decrease-key או delete במהלך תוכנית זו.

ג. פונקציית הפוטנציאל של המבנה כפי שהוגדרה הינה: Potential = #trees + 2#marked. בסעיף הקודם הסברנו כי לא יהיו כלל צמתים מסומנים בערימה שלנו מכיוון שלא מבצעים cuts, לכן Potential = #trees. מכיוון שבסוף התהליך נקבל ערימה בינומית תקינה (מכיוון שאין חתכים), מס' העצים בסוף התהליך ודרגתם יהיו בהתאם לייצוג הבינארי של m/2 (כי מחקנו m/2 איברים). כפי שלמדנו בכיתה מס' העצים בערימה, יהיה מס' הביטים הדולקים בייצוג הבינארי. לכן מס' העצים בערימה של m/2 איברים יהיה לכל היותר , ולכן פונקציית הפוטנציאל תהיה לכל היותר .