# תרגיל מעשי 2 - קובץ תיעוד ובדיקות


מגישים:

גיא לוי, 318645694, gl2

ליעד אילוז, 208427435, liadiluz


## תיעוד המחלקה והפונקציות

להלן תיעוד הפונקציות שממומשות תחת המחלקה FibonacciHeap, בחלוקה לפי סוג הפונקציה – אם נדרשנו לממש אותה במסגרת התרגיל או אם זו פונקציית עזר שהוספנו בעצמנו.

לא יופיע תיעוד על אופן הפעולה של פונקציות getter או setter שכל שהן עושות הוא לשלוף שדה או לעדכן שדה של אובייקט בערך ידוע מראש.

מופיע גם תיעוד קצר עבור השדות של שתי תתי המחלקות.

שדות של FibonacciHeap

| | |
|---|---|
| **מספר הצמתים בערימה** | `private int size;` |
| **מצביע לצומת הראשונה בערימה** | `private HeapNode first;` |
| **מצביע לצומת עם המפתח המינימלי בערימה** | `private HeapNode min;` |
| **מספר הצמתים המסומנים בערימה** | `private int numOfMarked;` |
| **מספר העצים בערימה** | `private int numOfTrees;` |
| **משתנה סטטי: מספר פעולות link שבוצעו מתחילת התכנית.** | `private static int totalLinkes;` |
| **משתנה סטטי: מספר פעולות cut שבוצעו מתחילת התכנית.** | `private static int totalCuts;` |

שדות של HeapNode

| | |
|---|---|
| מפתח של צומת | `public int key;` |
| דרגה של צומת (מספר הילדים של הצומת) | `private int rank;` |
| משתנה בוליאני לקביעת הסימון של הצומת (true אם הצומת מסומן). | `private boolean mark;` |
| מצביע לילד של הצומת (מאותחל ל-null). | `private HeapNode child;` |
| מצביע לצומת הבאה של הצומת (מאותחל  לצומת עצמו). | `private HeapNode next;` |
| מצביע לצומת הקודמת של הצומת (מאותחל  לצומת עצמו). | `private HeapNode prev;` |
| מצביע להורה של הצומת (מאותחל ל-null). | `private HeapNode parent;` |
| מצביע לצומת המקורית של הערימה לצורכי פעולת kMin בלבד. | `private HeapNode origin;` |

**פונקציות שנדרשנו לממש בשלד התרגיל:**

| שם תת המחלקה | חתימת הפונקציה | סיבוכיות הפונקציה |
|---|---|---|
| FibonacciHeap | public boolean isEmpty() | $O(1)$ |
| FibonacciHeap | public HeapNode insert(int key) | $O(1)$ |
| FibonacciHeap | public void deleteMin() | $amort: O(\log n)$<br>$WC: O(n)$ |
| FibonacciHeap | public HeapNode findMin() | $O(1)$ |
| FibonacciHeap | public void meld (FibonacciHeap heap2) | $O(1)$ |
| FibonacciHeap | public int size() | $O(1)$ |
| FibonacciHeap | public int[] countersRep() | $O(n)$ |
| FibonacciHeap | public void delete(HeapNode x) | $amort: O(\log n)$<br>$WC: O(n)$ |
| FibonacciHeap | public void decreaseKey(HeapNode x, int delta) | $amort: O(1)$<br>$WC: O(\log n)$ |
| FibonacciHeap | public int potential() | $O(1)$ |
| FibonacciHeap | public static int totalLinks() | $O(1)$ |
| FibonacciHeap | public static int totalCuts() | $O(1)$ |
| FibonacciHeap | public static int[] kMin(FibonacciHeap H, int k) | $O(k \cdot \deg H)$ |

פונקציות getter ו setter נוספות שנתבקשנו לממש עבור הממשק HeapNode ופונקציות getter ו-setter נוספות בשתי המחלקות:

| שם תת המחלקה | חתימת הפונקציה | סיבוכיות הפונקציה |
|---|---|---|
| HeapNode | public int getKey() | $O(1)$ |
| HeapNode | public int getRank() | $O(1)$ |
| HeapNode | public boolean isMarked() | $O(1)$ |
| HeapNode | public HeapNode getChild() | $O(1)$ |
| HeapNode | public HeapNode getNext() | $O(1)$ |
| HeapNode | public HeapNode getPrev() | $O(1)$ |
| HeapNode | public HeapNode getParent() | $O(1)$ |
| HeapNode | public void setKey(int k) | $O(1)$ |
| HeapNode | public void setRank(int r) | $O(1)$ |
| HeapNode | public void setMark(boolean m) | $O(1)$ |
| HeapNode | public void setChild(HeapNode node) | $O(1)$ |
| HeapNode | public void setNext(HeapNode node) | $O(1)$ |
| HeapNode | public void setPrev(HeapNode node) | $O(1)$ |
| HeapNode | public void setParent(HeapNode node) | $O(1)$ |
| HeapNode | private HeapNode getOrigin() | $O(1)$ |
| FibonacciHeap | public HeapNode getFirst() | $O(1)$ |

**פונקציות עזר שהוספנו בעצמנו:**

| שם תת המחלקה | חתימת הפונקציה | סיבוכיות הפונקציה |
|---|---|---|
| FibonacciHeap | private int getMaxTreeOrder() | $O(this.numOfTrees) \leq O(n)$ |
| FibonacciHeap | private void updateMin() | $O(this.numOfTrees) \leq O(n)$ |
| FibonacciHeap | private void cascading_cut(HeapNode x, HeapNode y) | $amort: O(1),$ $WC: O(\log n)$ |
| FibonacciHeap | private void cut(HeapNode x, HeapNode y) | $O(1)$ |
| FibonacciHeap | private void successiveLink() | $amort: O(\log n)$ $WC: O(n)$ |
| FibonacciHeap | private HeapNode linkTrees(HeapNode x, HeapNode y) | $O(1)$ |
| FibonacciHeap | private void insertNodePointer(HeapNode node) | $O(1)$ |
| FibonacciHeap | private void cutChildrenDeleteMin(HeapNode currMin) | $O(\log n)$ |

**תיעוד פונקציות השלד תחת FibonacciHeap**

<u>הפונקציה ()isEmpty:</u>

תכלית: הפונקציה מחזירה true אם הערימה ריקה ואחרת false.

אופן פעולה: הפונקציה מחזירה true אם שדה ה size של הערימה שווה לאפס ואחרת מחזירה false.

סיבוכיות: $O(1)$.

<u>הפונקציה (insert(int key:</u>

תכלית: הפונקציה מכניסה צומת חדש בעל מפתח $key$ לערימה.

אופן פעולה: הפונקציה יוצאת צומת חדש בעל מפתח $key$. אם יש כבר איברים בערימה, הפונקציה משרשרת את הצומת החדש משמאל אליהם ומטפלת במצביעים. בכל מקרה הפונקציה מגדירה את הצומת החדש כ-$first$ של הרשימה, מעדכנת את שדה ה-$min$ במידת הצורך, מגדילה את שדה ה- $size$ ב-1 ואת שדה ה- $numOfTrees$ ב-1 (שכן הצומת החדש מצטרף בתור שורש לעץ $B_0$ חדש בערימה).

סיבוכיות: $O(1)$.

יצירת הצומת החדש לוקחת $O(1)$ ושאר העבודה שהפונקציה מבצעת, גם היא קבועה.

הפונקציה ()deleteMin:

תכלית: הפונקציה מוחקת את הצומת בעל המפתח המינימלי בערימה.

אופן פעולה: אם הערימה מכילה עד איבר 1, הפונקציה לכל היותר מעדכנת את השדות הנדרשים.

אחרת, הפונקציה קוראת לפונקציה cutChildrenDeleteMin(minNode) אשר חותכת את הילדים של הצומת המינימלי וממקמת אותם מימינו. (סיבוכיות $O(\log n)$). לאחר מכן הפונקציה מעדכנת את שדה ה first במידת הצורך ומנתקת את הצומת המינימלי מהערימה בעזרת עדכון המצביעים של השורשים הסמוכים. הפונקציה מעדכנת את השדות של הערימה לפי הצורך.

לבסוף הפונקציה מעדכנת את המינימום בעזרת הפונקציה $updateMin()$ $(WC: O(n)\ , amort: O(\log n))$

וקוראת ל-$successive-link$ $(WC: O(n)\ , amort: O(\log n))$.

סיבוכיות: $WC: O(n)\ , amort: O(\log n)$

פרט לפונקציות העזר, הפונקציה מבצעת עבודה קבועה.

הפונקציה ()findMin:

תכלית: החזרת המצביע לצומת עם המפתח המינימילי בערימה.

אופן פעולה: החזרה ישירה של השדה min של הערימה.

סיבוכיות: $O(1)$.

הפונקציה (meld (FibonacciHeap heap2:

תכלית: הפונקציה מאחדת שתי ערימות לכדי ערימה אחת (הערימה שעליה קוראים לפונקציה).

אופן פעולה: אם הערימה השנייה ריקה, לא יבוצעו פעולות. אם הערימה הראשונה ריקה, נבצע עדכון לשדות כך שהיא תכיל כעת את הערימה השנייה. אחרת, הפונקציה משרשרת את הערימה השנייה מימין לראשונה בעזרת עדכון מצביעי $Prev\backslash Next$ של השורשים המתאימים ומעדכנת את השדות של הערימה המקורית.

סיבוכיות: $O(1)$.

הפונקציה מבצעת עבודה קבועה ולא קוראת לאף פונקציית עזר.

הפונקציה ()size:

תכלית: החזרת כמות האיברים בערימה.

אופן פעולה: החזרה ישירה של השדה size של הערימה.

סיבוכיות: $O(1)$.

הפונקציה ()countersRep:

תכלית: הפונקציה מחזירה מערך של כמות העצים בעץ כך שבתא ה-$i$ נמצאת כמות העצים בדרגה ה-$i$ (מהדרגה 0 ועד לדרגה המקסימלית של עץ בערימה).

אופן פעולה: תחילה מבוצעת קריאה לפונקציה העזר ()getMaxTreeOrder אשר עוברת על כל שורשי העצים בעץ ומחזירה את הדרגה המקסימלית של עץ בערימה. לאחר מכן מאותחל מערך כמתואר מעלה. לבסוף ישנו מעבר על כל שורשי העצים בערימה ועדכון התא הרלוונטי במערך בהתאם לדרגה של כל עץ בערימה.

סיבוכיות: $O(this.numOfTrees)$.

הפונקציה delete(HeapNode x):

תכלית: הפונקציה מוחקת את הצומת $x$ מהערימה.

אופן פעולה: הפונקציה תחילה מבצעת הפחתה של המפתח של הצומת לערך מינימלי בערימה. זאת באמצעות קריאה ל-decreaseKey(x, x.getKey() - *this*.min.getKey() + 1) .

סיבוכיות קריאה זו היא $O(1)$ אמורטייזד ו- $O(\log n)$ במקרה הגרוע. בשלב השני מבוצעת קריאה לפונקציה ()deleteMin המוחקת את האיבר המינימלי בערימה שהוא הצומת x.
סיבוכיות פעולה זו היא $(WC: O(n)\ ,amort: O(\log n))$.

סיבוכיות: $(WC: O(n)\ ,amort: O(\log n))$.

הפונקציה decreaseKey(HeapNode x, int delta):

תכלית: הפונקציה מפחיתה את ערכו של המפתח של הצומת x ב-delta.

אופן פעולה: מבוצעת תחילה הפחתה של הערך כמתואר מעלה. לאחר מכן, אם כלל הערימה בעץ שבו בוצעה הפחתה הופר (הפחתה לצומת שהופך להיות קטן מאביו), מבוצעת קריאה לפונקציה cascading_cut(x, x.getParent()), אשר מבצעת חיתוכים של צמתים בעץ הרלוונטי באופן סדרתי כפי שראינו בכיתה. סיבכיות פעולה זו היא ($amort: O(1),\ WC: O(\log n)$). לבסוף אם הערך המינימלי של הערימה השתנה ע"י הפחתת הצומת x מבוצע עדכון של הערך המינימלי לצומת x.

סיבוכיות: $amort: O(1), WC: O(\log n)$.

הפונקציה potential():

תכלית: הפונקציה מחזירה את הפוטנציאל ע"פ החישוב הבא:

Potential = #trees + 2*#marked

אופן פעולה: -

סיבוכיות: $O(1)$.

הפונקציה kMin(FibonacciHeap H, int k):

תכלית: הפונקציה מחזירה מערך ממוין של k המפתחות הקטנים ביותר בערימה. (בהנחה שהערימה מכילה עץ בינומי יחיד).

אופן פעולה: הפונקציה עושה שימוש בערימת עזר h2 שמכילה צמתים שלהם שדה origin שמצביע לצומת המתאים בערימה המקורית.

הפונקציה מתחילה מהמינימום, כלומר מהשורש של העץ היחיד בערימה (המקורית).

במהלך כל איטרציה הפונקציה מבצעת הכנסה של כל הילדים של הצומת הנוכחי לערימה h2 בתצורה של צמתים עם מצביע origin לילדים המקוריים (בערימה המקורית) בעזרת הפונקציה insertNodePointer(currChild) (שהיא $O(1)$).

לאחר מכן, הפונקציה מבצעת $deleteMin()$ על ערימת העזר h2 וממשיכה מאיבר זה את האלגוריתם. (בעצם אנו קוראים ל-origin של המינימום של h2 על מנת להגיע לצומת המקורי בערימה המקורית).

סיבוכיות: $O(k \cdot Deg(H))$.

במהלך הריצה של הפונקציה אנו מבצעים $k$ איטרציות. בכל איטרציה אנו מכניסים את כל הילדים של צומת מסויים בעץ הבינומי שבערימה המקורית לערימה $h2$ ואז מבצעים $deleteMin()$ על $h2$.

בגלל שאנו מכניסים ל-$h2$ רק איברים מהערימה המקורית, גודל הערימה $h2$ הוא לכל היותר כגודל הערימה המקורית, כלומר, $2^{Deg(H)}$. (הערימה המקורית מכילה עץ בינומי יחיד).

כל הכנסה תעלה $O(1)$ (נזכיר ש-$h2$ היא ערימת פיבונצ'י). בכל איטרציה יש $Deg(H)$ הכנסות לכל היותר. עלות ההכנסות של כל איטרציה תהיה $Deg(H)$ לכל היותר.

כל $deleteMin()$ יעלה לכל היותר $Deg(H) + Deg(H)$. הסיבה לכך היא שבסיום האיטרציה הקודמת ביצענו $deleteMin()$ ולכן נשארנו עם ערימה בינומית תקינה שמכילה $Deg(H)$ עצים לכל היותר. באיטרציה הנוכחית הוספנו לכל היותר $Deg(H)$ צמתים ולכן אנו מבצעים $deleteMin()$ על ערימת פיבונצ'י שמכילה לכל היותר $2Deg(H)$ עצים. נסיק שזו גם העלות של ה-$successive - linking$ שנבצע ב- $deleteMin()$.

על כן, כל איטרציה תהיה מסיבוכיות $O(Deg(H))$.

לסיכום, אנו מבצעים $k$ איטרציות כאשר כל אחת מסיבוכיות $O(Deg(H))$.

לכן סיבוכיות זמן הריצה של הפונקציה היא: $O(k \cdot Deg(H))$.

הפונקציה ()totalLinks:

תכלית: פונקציה סטטית המחזירה את השדה הסטטי totalLinkes אשר מונה את כמות ביצועי link מתחילת התכנית.

אופן פעולה: -

סיבוכיות: $O(1)$.

הפונקציה () totalCuts:

תכלית: פונקציה סטטית המחזירה את השדה הסטטי של המחלקה totalCuts המונה את מספר ביצועי cut מתחילת התכנית.

אופן פעולה: -

סיבוכיות: $O(1)$.

**תיעוד פונקציות העזר תחת המחלקה FibonacciHeap**

<u>הפונקציה ()getMaxTreeOrder:</u>

תכלית: הפונקציה מחזירה את הדרגה המקסימלית של עץ בערימה.

אופן פעולה: הפונקציה עוברת על כל שורשי העצים בערימה ומוצאת את הדרגה המקסימלית (מוחזק כשדה rank לכל צומת).

סיבוכיות: $O(this.numOfTrees)$, כלומר $O(n)$ במקרה הגרוע.

<u>הפונקציה ()updateMin:</u>

תכלית: הפונקציה מעדכנת את ערכו של הצומת עם המפתח המינימלי בערימה.

אופן פעולה: הפונקציה עוברת על כל שורשי העצים בערימה ומעדכנת את השדה min בכל פעם שהמפתח של השורש קטן מהמפתח המינימלי הזמני בערימה.

סיבוכיות: $O(this.numOfTrees)$. כלומר $O(n)$ במקרה הגרוע.

<u>הפונקציה (cascading_cut(HeapNode x, HeapNode y:</u>

תכלית: הפונקציה מבצעת חיתוך סדרתי של מהצומת x וההורה שלו y ובמעלה העץ כל עוד יש הצדקה להמשיך לבצע חיתוך: כל עוד בוצע חיתוך על בן של הורה מסומן בעץ.

אופן פעולה: הפונקציה מבצעת קריאה לפונקציה (cut(x,y, אשר מבצעת חיתוך של הצומת x מההורה y ומכניסה את הצומת x ותת העץ שלו כעץ חדש בתחילת הערימה. לאחר מכן, כל עוד ההורה y הוא צומת מסומן בעץ, מבוצעת קריאה רקורסיבית לפונקציה הנוכחית אשר חותכת את הצומת y מההורה שלו. אם y לא היה מסומן (ואינו שורש) הפונקציה מסמנת את y.

סיבוכיות: $amort: O(1),\ WC: O(\log n)$.

הפונקציה cut(HeapNode x, HeapNode y):

תכלית: הפונקציה מבצעת חיתוך יחיד של הצומת x מההורה שלו y (y!=null).

אופן פעולה: הצומת x נחתכת מעץ שבו הוא נמצע והופך להיות שורש של עץ חדש בתחילת הערימה יחד עם כל תת העץ שהיה לו. מבוצעים עדכונים למצביעים ולשדות הרלוונטיים.

סיבוכיות: $O(1)$.

הפונקציה successiveLink():

תכלית: הפונקציה מבצעת את פעולת ה- $successive - linking$ כפי שהגדרנו אותה בהרצאה.

אופן פעולה: הפונקציה משתמשת במערך עזר של $HeapNodes$ שמספר התאים שלו הוא $O(\log n)$.

הפונקציה עוברת באופן סדרתי על שורשי העצים בערימה. לכל שורש הפונקציה בודקת אם המקום במערך העזר שמייצג את הדרגה של אותו השורש תפוס כבר. אם לא, היא תכניס אותו למערך במקום זה ותנתק אותו משרשרת השורשים של הערימה. אחרת, הפונקציה תחבר את השורש הנוכחי לשורש שנמצא בתא במערך בעזרת פונקציית עזר $linkTrees$ (שהיא $O(1)$) ותעביר את שורש העץ המאוחד לתא הבא במערך. אם גם הוא תפוס, הפונקציה תבצע קישור סידרתי עד שהיא תצליח למקם עץ בתא ריק במערך העזר.

לאחר שהפונקציה עברה על כל שורשי העצים בערימה, היא מבצעת שכפול של מערך העזר למערך דומה, שאורכו הוא לכל היותר זהה למערך העזר אך מכיל רק את התאים המלאים של מערך העזר.

הפונקציה תעבור על מערך העזר החדש (שגודלו לכל היותר $O(\log n)$) ותבצע קישור בין כל השורשים שבו באופן סדרתי על מנת לקבל את הערימה החדשה. הפונקציה מעדכנת את השדות $first$ ו- $min$.

סיבוכיות: $O(n)$.

ראינו בכיתה שסיבוכיות הפעולה $successive - link$ היא כמספר העצים בערימה. בערימת פיבונצ'י יכולים להיות לכל היותר $n$ עצים בעת הקריאה ל-$successive - linking$ ולכן היא מסיבוכיות: $O(n)$.

נציין שהמעבר על מערכי העזר לא עולה יותר מ-$O(n)$ וכי הפונקציה שלנו מבצעת לכל שורש לכל היותר את כמות העבודה שראינו בכיתה, עד כדי כפל בקבוע.

הפונקציה linkTrees(HeapNode x, HeapNode y):

תכלית: הפונקציה מקבלת שני שורשים של עצים בערימת פיבונצ'י ומבצעת קישור שלהם לכדי עץ אחד.

הפונקציה מחזירה את השורש של העץ המאוחד.

אופן פעולה: הפונקציה בודקת מי מבין המפתחות של שני הצמתים קטן יותר. מי שקטן יותר יהיה השורש של העץ החדש. לאחר מכן הפונקציה מנתקת את השורש הגדול יותר מהאחים שלו, הופכת אותו לילד של השורש הקטן יותר ומטפלת במצביעים של הילדים של השורש הקטן. הפונקציה מגדילה את הדרגה של השורש הקטן ב-1 מגדילה את השדה $totalLinks$ ב-1 ומקטינה את $numOfTrees$ ב-1. לבסוף הפונקציה מחזירה את השורש הקטן יותר, שהוא כעת השורש של העץ המאוחד.

סיבוכיות: $O(1)$.

הפונקציה מבצעת מספר קבוע של פעולות שלוקחות זמן קבוע כל אחת. לכן סיבוכיות הפונקציה היא: $O(1)$.

הפונקציה insertNodePointer(HeapNode node):

תכלית: הפונקציה מבצעת הכנסה של צומת בעל מצביע לצומת אחר עם אותו המפתח.
פונקציה זאת משמשת באופן בלעדי על מנת להכניס צמתים לערימת עזר במסגרת הפונקציה kMin().
יש צורך בפונקציה הנ"ל שכן אנו רוצים לגשת מהמינימום של ערימת העזר לצומת שהוא מייצג בערימה המקורית.

אופן פעולה: הפונקציה פועלת באופן זהה לפונקציה insert(int key) פרט לכך שבמקום ליצור צומת חדש עם מפתח $key$, היא יוצרת צומת חדש אשר מכיל את המפתח $key$ וגם מכיל מצביע לצומת המקורי בערימה המקורית עם אותו המפתח.

סיבוכיות: $O(1)$.
הפונקציה זהה לפונקציה insert(int key) פרט ליצירה של הצומת החדש, אשר מתבצעת בזמן קבוע.

הפונקציה cutChildrenDeleteMin(HeapNode currMin):

תכלית: פונקציה זו היא פונקציית עזר שנקראת בעת ביצוע $deleteMin()$. התפקיד שלה הוא חיתוך הילדים של הצומת המינימלי בערימה ומיקום שלהם (באותו סדר פנימי) כשורשים של תתי עצים לאחר (מימין) האיבר המינימלי (שגם הוא שורש) בסדרת השורשים שמהווים את ערימת הפיבונצ'י.

אופן פעולה: ראשית הפונקציה עוברת בלולאה על כל הילדים של המינימום. לכל ילד היא מגדירה את ההורה להיות null, מורידה את ה-mark שלו (שכן כעת יהפוך לשורש בעצמו) ומגדילה את השדה $numOfTrees$ ב-1 (שכן הילד הזה יהפוך להיות שורש של תת עץ נפרד חדש בערימה).

לאחר סיום הלולאה, הפונקציה מעדכנת את מצביעי ה-$prev/next$ של הילדים המתאימים על מנת לשרשר את רצף הילדים לאחר (מימין) לצומת המינימלי ומגדירה את הילד של הצומת המינימלי כ-null.

סיבוכיות: $O(\log n)$.

הפונקציה מבצעת כמות עבודה קבועה פרט ללולאה שעוברת על כל הילדים של המינימום.

מספר הילדים של המינימום הוא $\log n$ כדרגה הגדולה ביותר האפשרית בערימת פיבונצ'י עם n איברים.

לכל ילד אנו מבצעים עבודה קבועה.

לכן, סיבוכיות זמן הריצה של הפונקציה היא: $O(\log n)$.

**חלק ניסויי / תיאורטי**

<u>שאלה 1</u>

<u>סעיף א'</u>

נחשב את זמן הריצה האסימפטוטי במונחי "$Big\ O$" של סדרת הפעולות כפונקציה של m.

<u>בשלב הראשון</u> - אנו מבצעים $m+1$ פעמים $insert(k)$. סיבוכיות כל פעולה כזאת היא $O(1)$.

לכן סיבוכיות זמן הריצה של השלב הראשון היא $O(m+1)$, כלומר $O(m)$.

<u>בשלב השני</u> – אנו מבצעים פעולת $deleteMin()$. סיבוכיות פעולה זו היא לינארית במספר העצים.

לכן נקבל $O(m)$.

לאחר פעולה זו הערימה היא עץ בינומי יחיד מדרגה $\log(m)$.

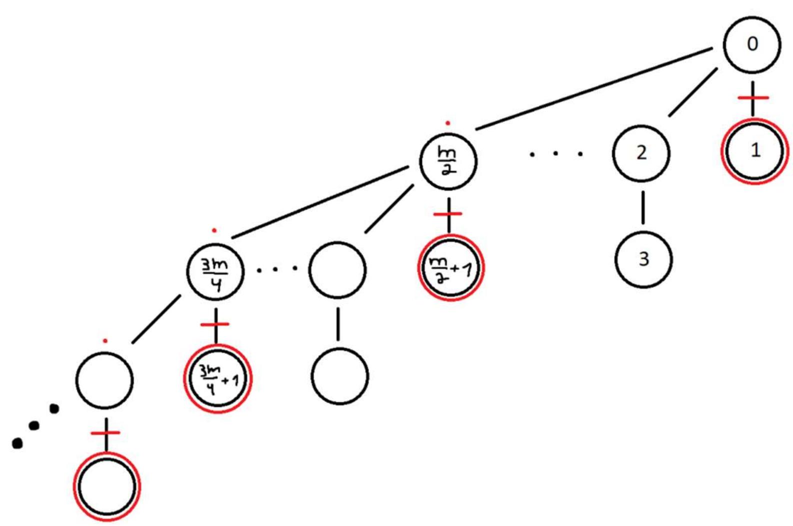
<u>בשלב השלישי</u> – אנו מבצעים $\log(m)$ פעולות של $decreaseKey$. בגלל הסדר של הצמתים שאנו מבצעים עליהם $decreaseKey$ והמבנה של העץ, בכל פעולה כזאת, אנו מבצעים שני דברים:

1. מורידים את הערך של המפתח לערך שיהיה הערך המינימלי החדש בעץ – כלומר, נבצע חיתוך.

2. מורידים את הערך של צומת שהוא ילד להורה לא מסומן ולכן החיתוך יהיה יחיד.

(מכיוון שכל פעולת חיתוך ראשוני מתבצעת רמה אחת מתחת לרמה של מיקום הפעולה הקודמת).

למען המחשה נספק תמונה כללית שמתארת את המצב של העץ ואת הצמתים שאת המפתח שלהם נוריד. (קו אדום אומר חיתוך ונקודה אדומה מעל צומת מסמלת שהוא צומת מסומן).




לכן, סיבוכיות כל פעולת $decreaseKey$ היא $O(1)$ וביצענו $\log(m)$ כאלה.

נקבל שסיבוכיות השלב השלישי של סדרת הפעולות היא: $O(\log(m))$.

<u>בסה"כ סיבוכיות זמן הריצה של סדרת הפעולות היא: $O(m)$.</u>

## סעיף ב'

| m | Run-Time (ms) | totalLinks | totalCuts | Potential |
|---|---|---|---|---|
| $2^{10}$ | 2 | 1023 | 10 | 29 |
| $2^{15}$ | 20 | 32767 | 15 | 44 |
| $2^{20}$ | 177 | 1,048,575 | 20 | 59 |
| $2^{25}$ | 5402 | 33,554,431 | 25 | 74 |

## טבלה עבור סעיפים ג'-ו'

| case | totalLinks | totalCuts | Potential | decreaseKey max cost |
|---|---|---|---|---|
| (c) original | $m-1$ | $\log m$ | $3 \cdot \log m - 1$ | (skip) |
| (d) decKey(m-2^i) | $m-1$ | 0 | 1 | (skip) |
| (e) remove line #2 | 0 | 0 | $m+1$ | (skip) |
| (f) added line #4 | $m-1$ | $2 \cdot \log(m) - 1$ | $2 \cdot \log(m)$ | $\log(m) - 1$ |

## סעיף ג'

מספר פעולות ה-$link$ שנבצע הוא: $m-1$.

הסבר: ראשית נשים לב שפעולות ה-$link$ מתבצעות רק במסגרת ה-$successive - linking$ של פעולת ה-$deleteMin()$.

כל פעולת $link$ מצמצמת את מספר העצים ב-1. מכיוון שהתחלנו עם $m+1$ עצים לאחר פעולות ההכנסה ואז מחקנו עץ אחד, נתחיל את פעולת ה- $successive - linking$ על $m$ עצים ונסיים אותה עם עץ אחד. אם כך, ביצענו $m-1$ פעולות $link$.

מספר פעולות ה-$cut$ שנבצע הוא: $\log(m)$.

הסבר: לפי הניתוח שביצענו בסעיף א' – בכל פעולת $decreaseKey$ נבצע בדיוק חיתוך אחד ולכן כמות פעולות ה-$cut$ שנבצע היא בדיוק $\log(m)$.

גודל הפוטנציאל הוא: $3 \cdot \log(m) - 1$.

הסבר: לפי הניתוח בסעיף א', בכל פעולת $decreaseKey$ אנו יוצרים עץ חדש אחד ומסמנים צומת חדש אחד (חוץ מהפעולה הראשונה שבמסגרתה לא נסמן את שורש העץ).

מכיוון שהתחלנו עם עץ 1 נקבל שמספר העצים הסופי הוא: $\log(m) + 1$.

מכיוון שהתחלנו עם מספר מסומנים 0,
נקבל שמספר המסומנים בסוף סדרת הפעולות הוא: $\log(m) - 1$.

אם כך, הפוטנציאל לאחר סדרת הפעולות הוא: $\#Trees + 2(\#Marked) = 3 \cdot \log(m) - 1$.

סעיף ד'

מספר פעולות ה-$link$ שנבצע הוא: $m-1$.

הסבר: ההסבר הוא זהה לסעיף ג'.

מספר פעולות ה-$cut$ שנבצע הוא: $0$.

הסבר: כעת אנו מבצעים $decreaseKey$ מהשורש לאורך הענף הכי שמאלי בעץ (מלמעלה למטה). בגלל סדר פעולות זה, בכל פעם שנבצע הנמכה של ערך המפתח, ערך זה ישאר גדול יותר מההורה שלו (שאותו גם הנמכנו בצעד הקודם). לכן כלל העץ לא מופר באף שלב ולא מתבצעים חיתוכים.

גודל הפוטנציאל הוא: $1$.

הסבר: מכיוון שלא ביצענו אף חיתוך – מספר המסומנים ישאר 0 ומספר העצים ישאר 1.

אם כך, הפוטנציאל יהיה: $\#Trees + 2(\#Marked) = 1$

סעיף ה'

מספר פעולות ה-$link$ שנבצע הוא: $0$.

הסבר: אנו מבצעים $link$ רק במסגרת פעולת $deleteMin()$. מכיוון שהורדנו את פעולה זו לא נבצע פעולות $link$.

מספר פעולות ה-$cut$ שנבצע הוא: $0$.

הסבר: כעת כל איבר בערימה הוא שורש של עץ $B_0$. לכן בכל פעולת $decreaseKey$ לא יתבצע שום חיתוך.

גודל הפוטנציאל הוא: $m+1$.

הסבר: מספר העצים הוא $m+1$ לאחר ביצוע פעולות ההכנסה ועד לסוף התוכנית. בגלל שלא נבצע שום חיתוך מספר המסומנים ישאר $0$.

אם כך, הפוטנציאל יהיה: $\#Trees + 2(\#Marked) = m + 1$

סעיף ו'

מספר פעולות ה-$link$ שנבצע הוא: $m-1$.

הסבר: סעיף זה זהה לסעיף ג' פרט לכך שבסופו נבצע עוד פעולת $decreaseKey$ שלא גוררת $link$-ים נוספים. (אנו מבצעים $link$ רק במסגרת פעולת $deleteMin()$).

מספר פעולות ה-$cut$ שנבצע הוא: $2\cdot\log(m)-1$.

הסבר: סעיף זה מתחיל בביצוע סעיף ג' במסגרתו היו לנו $\log m$ פעולות $cut$.

לאחר ביצוע של 3 השורות הראשונות, אנו מגיעים למצב שבו הענף השמאלי של העץ המקורי מכיל צמתים מסומנים בלבד (פרט לשורש ולאיבר שהיה העלה השמאלי ביותר בעץ המקורי שאותו חתכנו במהלך הלולאה של שורה 3). כלומר, נישאר עם ענף שמאלי שבו יש $\log(m)-1$ צמתים מסומנים (כולם פרט לשורש). הפעולה בשורה 4 גוררה חיתוך של הצומת שכרגע הוא העלה השמאלי ביותר בעץ וכתוצאה מכך נצטרך לבצע שרשרת של מחיקות לאורך כל הענף השמאלי, עד לשורש.
כלומר, $\log(m)-1$ חיתוכים.

גודל הפוטנציאל הוא: $2\cdot\log(m)$.

הסבר: לפי סעיף ג', לאחר 3 השורות הראשונות נשאר עם $\log(m)+1$ עצים.

כפי שהסברנו קודם, השורה הרביעית תגרור $\log(m)-1$ חיתוכים נוספים ולכן $\log(m)-1$ עצים נוספים.

בנוסף, כל הצמתים שהיו מסומנים לאחר 3 השורות הראשונות היו ממוקמים בענף השמאלי ביותר של העץ וכעת יחתכו ויהפכו לשורשים חדשים ולכן יאבדו את הסימון שלהם. נישאר עם $0$ צמתים מסומנים.

אם כך, לאחר סדרת הפעולות בסעיף זה הפוטנציאל הוא: $\#Trees+2(\#Marked)=2\cdot\log(m)$.

העלות היקרה ביותר של פעולת $decreaseKey$ היא: $\log(m)-1$.

הסבר: לפי הניתוח של מספר פעולות ה-$cut$ שנבצע בסעיף ג', כל פעולות ה-$cut$ שנבצע ב-3 השורות הראשונות עולות $O(1)$ כל אחת. בניתוח של מספר פעולות ה-$cut$ בסעיף זה, ראינו שבשורה 4 יהיה עלינו לבצע $cut$ שיגרור $cascading-cut$ שמהלכו נבצע $\log(m)-1$ חיתוכים.

אם כך, העלות היקרה ביותר של פעולת $decreaseKey$ היא: $\log m-1$.

## שאלה 2

### סעיף א'

| m | Run-Time (ms) | totalLinks | totalCuts | Potential |
|---|---|---|---|---|
| 728 | 3 | 723 | 0 | 6 |
| 6560 | 9 | 6555 | 0 | 6 |
| 59048 | 50 | 59040 | 0 | 9 |
| 531,440 | 280 | 531,431 | 0 | 10 |
| 4,782,968 | 2433 | 4,782,955 | 0 | 14 |

### סעיף ב'

נחשב את זמן הריצה האסימפטוטי במונחי "$Big\ O$" של סדרת הפעולות כפונקציה של $m$.

שורה 1 – בשורה זו אנו מבצעים לולאה של $m$ איטרציות. בכל איטרציה אנו מבצעים פעולת $insert$ שהסיבוכיות שלה היא $O(1)$. לכן סיבוכיות זמן הריצה של שורה 1 היא $O(m)$.

שורה 2 – בשורה זו אנו מבצעים לולאה של $\frac{3m}{4}$ איטרציות. בכל איטרציה אנו מבצעים $deleteMin()$ שבמסגרתו קוראים ל-$successive-linking$. כפי שראינו בכיתה, סיבוכיות הזמן של פעולת $successive-linking$ חסומה על ידי מספר העצים שעליהם קוראים לה.

בפעם הראשונה שנקרא לה, יהיו $m$ עצים. לאחר מכן נקבל ערימה תקינה שמכילה $\log m$ עצים.

מספר זה חוסם את כמות העצים שעליהם נקרא ל-$successive-linking$ לאורך שארית סדרת הפעולות.

אם כך, נקבל שסיבוכיות הזמן של הלולאה של שורה 2 חסומה על ידי:

$$m+\left(\frac{3m}{4}\cdot\log m\right)=O(m\cdot\log(m))$$

בסה"כ נקבל חסם סיבוכיות זמן ריצה עבור סדרת הפעולות של: $O(m\cdot\log(m))$.

סעיף ג'

מספר פעולות ה-$link$ הוא: $m - Bin_1(m)$.

בפעם הראשונה שאנו מבצעים $deleteMin()$, אנו מתחילים עם $m+1$ עצי $B_0$, מוחקים אחד מהם וקוראים ל- $sucessive - linking$ על $m$ עצי $B_0$.

את ה-$successive - linking$ אנו מתחילים על $m$ עצי $B_0$ ומסיימים עם מספר עצים שהוא כמספר הביטים "1" בייצוג הבינארי של המספר $m$ (נסמן מספר זה ב-$Bin_1(m)$).

זאת מכיוון שלאחר ביצוע $successive - linking$ אנו מגיעים לערימה בינומית תקינה וכפי שראינו בכיתה אפשר לבצע התאמה בין העצים בערימה בינומית בעלת $n$ איברים לבין הביטים "1" בייצוג הבינארי של המספר $n$.

בכל פעולת $link$ אנו מפחיתים את כמות העצים ב-1.

לכן מספר פעולות ה-$link$ במסגרת ה-$deleteMin()$ הראשונה הוא: $m - Bin_1(m)$.

כעת נסביר מדוע לא יבוצעו עוד $link$-ים פרט לרצף הראשון שתואר זה עתה.

לאחר ביצוע $successive - link$ ראשון, מהאופן שבו הכנסנו את האיברים לערימה בהתחלה ומהאופן שבו אנו עוברים על העצים בעת ביצוע $successive - link$, האיבר המינימלי יהיה תמיד השורש של העץ מהרמה הנמוכה ביותר. (תכונה שמשמרת את עצמה לאחר כל קריאה של $deleteMin()$).

אם כך, בכל פעם שנבצע $deleteMin()$, אחרי הפעם הראשונה, פעולות ה-$successive - link$ לא תצטרך לבצע אף $link$. זאת בגלל שכל הילדים של המינימום מהווים שורשים לתתי עצים מדרגות שונות שלא נמצאות בעץ לפני כן.

לסיכום, מספר פעולות ה-$link$ שנבצע הוא: $m - Bin_1(m)$.

מספר פעולות ה-$cut$ הוא: 0.

פעולות $cut$ מתבצעות רק במסגרת קריאה לפונקציה $decreaseKey$. בגלל שלא קראנו לה כלל, מספר פעולות ה-$cut$ הוא: 0.

גודל הפוטנציאל הוא כמות העצים בסוף התוכנית = מספר הביטים "1" בייצוג הבינארי של $m/4+1$.

במהלך סדרת הפעולות לא ביצענו $decreaseKey$ כלל. לכן לא סימנו אף צומת.

אם כך הפוטנציאל הוא: $\#Trees + 2(\#Marked) = \#Trees$.

כפי שהסברנו בניתוח של מספר פעולות ה-$link$, מספר זה הוא כמספר הביטים "1" בייצוג הבינארי של המספר $\frac{m}{4}+1$.