**מבני נתונים – פרויקט מעשי 2**

# Fibonacci Heap

## תיעוד המחלקות – ניתוחי סיבוכיות, מתודות ושדות

**מחלקת HeapNode –**

מחלקה זו מטרתה לתאר צומת בערימת פיבונאצ'י, נשמור בה שדות רלוונטי ונכתוב מתודות שיעזרו לנו לגשת ולעדכן את השדות הנ"ל.

שדות :

private int key – שדה השומר את המפתח של צומת נתון
private int rank – שדה השומר את הדרגה של צומת נתון.
private boolean mark – שדה בוליאני הבודק אם צומת נתון מסומן.
priavte HeapNode child – שדה המחזיק את הבן השמאלי של צומת נתון.
private HeapNode parent – שדה המחזיק את צומת האב של צומת נתון.
private HeapNode next – שדה המחזיק את הצומת הקרוב מימינו של צומת נתון ( במידה ואין כזה, יצביע אל עצמו ).
private HeapNode prev - שדה המחזיק את הצומת הקרוב משמאלו של צומת נתון ( במידה ואין כזה, יצביע אל עצמו ).

private HeapNode corresponding_kmin_node - שדה עזר המיועד לשימוש במתודת kMin, יחזיק שדה מסוג heap-node שיצביע על צומת אחר.

בנאי :

public HeapNode(int key):
נעדכן את המפתח של הצומת החדש שיצרנו, כמו כן את שדות הצומת הקודם לו והעוקב לו להיות הוא עצמו כי אינו משורשר לשום דבר. שאר הערכים יעודכנו בצורה הדיפולטית שג'אבה מעדכנת –
key = 0, rank = 0, mark = false, child = null, parent = null
בבנאי זה אנו רק מעדכנים שדות לכן זמן הריצה יהיה קבוע.
**time complexity : worst case O(1)**

public boolean getMarked():
מתודה המחזירה האם צומת נתון מסומן, מחזירה ערך ששמור בשדה לכן זמן הריצה יהיה קבוע.
**time complexity : worst case O(1)**

public int getKey():
מתודה המחזירה את מפתחו של צומת נתון, שמור בשדה ולכן זמן הריצה יהיה קבוע.
**time complexity : worst case O(1)**

public void SetNext(HeapNode next), public void setPrev(HeapNode prev):

המתודות הנ״ל מקבלים צמתים עוקבים וקדומים בהתאמה ומעדכנות את שדות הצומת נתון בהתאם. זמן הריצה של שתי המתודות יהיה קבוע כיוון שמדובר בעידכון שדות.

**time complexity : worst case O(1)**

**בדומה למתודות קודמות כל המתודות הבאות במחלקה מעדכנות או בודקות שדות ולכן זמן הריצה של כולן יהיה קבוע.**

public int getRank():

מתודה המחזירה את דרגת הצומת.

**time complexity : worst case O(1)**

public HeapNode getChild():

מתודה המחזירה את הבן השמאלי של הצומת הנתון.

**time complexity : worst case O(1)**

public HeapNode getParent():

מתודה המחזירה את צומת האב של הצומת הנתון

**time complexity : worst case O(1)**

public HeapNode getNext():

מתודה המחזירה את הצומת העוקב של הצומת הנתון

**time complexity : worst case O(1)**

public HeapNode getPrev():

מתודה המחזירה את הצומת הקודם של הצומת הנתון

**time complexity : worst case O(1)**

**מחלקת FibonacciHeap –**

מחלקה זו מתארת ערימת פיבונאצ'י, ניעזר במחלקת heap-node על מנת לתאר איברים בערימה.

שדות :

private HeapNode first - שדה שיצביע לצומת הראשון בערימה, נדאג לתחזק אותו בעת הכנסה מחיקה ושאר הפעולות.

private HeapNode last - שדה היצביע לצומת האחרון בערימה, יתוחזק בצורה דומה לfirst

private HeapNode min - שדה היצביע לשורש המינימלי בערימה, זה יהיה האיבר הקטן בערימה

private int size - שדה שישמור את גודל הערימה, נתחזק בהכנסה מחיקה וכו'

private int marked_cnt - שדה שישמור את מספר הצמתים המסומנים בערימה.

private int tree_cnt - שדה שיימור את מספר העצים שיש בערימה.

private int max_rank - שדה שישמור את הדרגה הכי גדולה של צומת בערימה.

private static int link_cnt - שדה סטטי ששומר את כמות השרשורים שביצענו.

private static int cuts_cnt - שדה סטטי ששומר את כמות החיתוכים שעשינו בעת מחיקה או decrease-key

בנאי : נשתמש בבנאי הדיפולטי של ג'אבה, נשים לב שזו אינה בעיה כיוון שכל הערכים הדיפולטים שג'אבה תתן לנו עבור השדות מתאימים להם, נחדד ונראה את הערכים של השדות הדיפולטים :

HeapNode first = null, HeapNode last = null, HeapNode min = null, int size = 0, int marked_cnt = 0, int tree_cnt = 0, int max_rank = 0, int link_cnt = 0, int cuts_cnt = 0

כיוון שכל ערכים אלה מסתדרים עם מה שהיינו רוצים במקרה בו היינו ממשים את הבנאי בעצמנו, אין צורך בלממש את הבנאי ולכן זה אכן בסדר להשאר עם הבנאי הדיפולטי שג'אבה מציעה.

זמן הריצה כמובן קבוע עבור בנאי שרק מאתחל שדות.

**time complexity : worst case O(1)**

public boolean isEmpty():

פונקציה בוליאנית שבודקת אם הערימה ריקה או לא – נממש ע''י לבדוק אם שדה הגודל שווה לאפס, אם כן נחזיר true אחרת נחזיר false .

זמן הריצה הוא קבוע, כיוון שאנו רק בודקים שדה .

**time complexity : worst case O(1)**

public HeapNode insert(int key):

מתודה המכניסה איבר חדש לערימה – במימוש יצרנו צומת חדש עם המפתח שקיבלנו, השתמשנו במתודת nodeInsert אשר נפרט עליה בהמשך שמכניסה צומת חדש לעץ. דאגנו להגדיל את שדה הגודל ולסיום החזרנו את הצומת החדש שיצרנו.

זמן הריצה יהיה קבוע כיוון שכל הנעשה במתודה הוא עידכון שדה וקריאה ל node-insert שזמן הריצה שלה קבוע גם כן.

**time complexity : worst case O(1)**

private void nodeInsert(HeapNode node):

מתודת עזר שמטרתה להכניס טיפוס מסוג heapnode לערימה.
במתודה אנו מכניסים את הצומת שהתקבל כצומת הראשון בעץ, ומעדכנים שדות רלוונטים ( למשל בודקים אם הוא המינימום, אם כן דואגים לשנות את שדה המינימום לצומת החדש ), בנוסף נדאג לשנות את כל המצביעים הרלוונטים של השדות שהיו קיימים בעץ לפני ( ואם לא היו קיימים נטפל במקרה גם שהצומת הוא הצומת הראשון והיחיד שהכנסנו לעץ ) .
סה''כ כל הפעולות במתודה הן עידכוני ובדיקת שדות ולכן זמן הריצה קבוע.
**time complexity : worst case O(1)**

private void deleteMin():
פונקציית ה delete-min, בה אנו בודקים שהערימה אינה ריקה, אם תהיה ריקה נקרא לפונקציה delete עם המינימום הנתון בערימה שלנו ( שמתוחזק כשדה ), על דרך פעולת delete נפרט בהמשך.
זמן הריצה כפי שנלמד בביתה הוא O(n) במקרה הגרוע, ו(amortized cost O(log(n))
**time complexity : worst case O(n), Amortized-cost O(log(n))**

public HeapNode findMin():
מתודה שמחיזרה את המינימום בערימה – נחזיר את המצביע למינימום שאנו שומרים כשדה.
מחזירים ערך שמתוחזק כשדה לכן זמן הריצה הוא קבוע.
**time complexity : worst case O(1)**

public HeapNode getLast() & public HeapNode getFirst():
זוג מתודות שפועל באופן זהה ומטרתן להחזיר את האיבר האחרון והאיבר הראשון בערימה בהתאמה. נחזיר את המצביעים שאנו מתחזקים כשדות באופן דומה ל find-min.
זמן הריצה יהיה קבוע לשתי המתודות כיוון שאנו מחזירים ערך השמור בשדה.
**time complexity : worst case O(1)**

public int getMarkedCount():
מתודה שנועדה להחזיר את כמות הצמתים המסומנים, ערך המתוחזק בשדה ( ומשתנה בזמני מחיקות ).
זמן הריצה יהיה קבוע כיוון שאנו מחזירים ערך ששמור בשדה.
**time complexity : worst case O(1)**

public int getTreeCount():
מתודה שנועדה להחזיר את כמות העצים שיש בערימה, ערך המתוחזק כשדה.
זמן הריצה יהיה קבוע כיוון שאנו מחזירים ערך ששמור בשדה.
**time complexity : worst case O(1)**

public void meld(FibonacciHeap heap2):

מתודה שמטרתה לשרשר זוג ערימות. ברצוננו לשרשר את הערימה המתקבלת במתודה לערימה שלנו. נעשה זאת ע"י עידכוני המצביעים בסוף ותחילת הרשימה ושדותיהם, בנוסף נדאג לעדכן את המינימום, גודל הערימה כמות הצמתים המסומנים וכמות העצים.

זמן הריצה יהיה קבוע כיוון שכל מה שאנו עושים זה לעדכן את שדות העץ ומצביעים.

**time complexity : worst case O(1)**

public int size():

מתודה המחזירה את גודל הערימה, נתון שאנו מתחזקים כשדה.

זמן הריצה של החזרת ערך הנתון כשדה הוא קבוע.

**time complexity : worst case O(1)**

public int[] countersRep():

מתודה המחזירה מערך ובו האיבר במיקום ה -i הוא כמות השורשים מדרגה i .

נעבור על כל השורשים בלולאה, לכל שורש מדרגה k נדאג להגדיל באחד את האיבר ה- k במערך. כמו כן נשמור את הדרגה המקסימלית במעבר על השורשים על מנת שנוכל בסיום לייצר מערך חדש מהגודל של הדרגה המקסימלית ולהעתיק אליו את האיברים.

ניתוח זמן הריצה: במקרה הגרוע ייתכן שיש n שורשים ונצטרך לעבור על כולם בלולאה, בגוף הלולאה אנו רק ניגשים לשדה של השורשים לכן זמן הריצה לכל איטרציה קבוע. לכן זמן הריצה במקרה הגרוע יהיה לינארי באורך הקלט, כלומר O(n)

**time complexity : worst case O(n)**

public void delete(HeapNode x):

מתודה שמטרתה למחוק איבר מהערימה.

נחלק למקרים הבאים : הערימה מכילה איבר אחד, במקרים האחרים הערימה מכילה יותר מאיבר אחד – נפרט מהם : אנו מוחקים שורש ואנו מוחקים צומת שאינו שורש.

במקרה הראשון והפשוט, נדאג לעדכן את כל שדות הערימה לערכים הדיפולטים כך שלא יהיה בערימה שום צומת או מידע.

במקרה בו אנו מוחקים צומת שהוא שורש : נבדוק אם לצומת יש ילדים, אם לא המקרה קל לטיפול, רק נרצה לדאוג למצביעים של האיברים הסמוכים לשורש. אם לשורש יש ילדים, נדאג לחתוך אותם מהעץ ולהכניס אותם במקום השורש אותו מחקנו, נעזר במתודה העזר cutAddToMiddleOfList ולעדכן שדות כמו כמות העצים וגודל הערימה.

במקרה בו אנו מוחקים צומת שאינו שורש : נצטרך לבצע cascadingCuts על הצומת שהתבקשנו למחוק כפי שלמדנו, בסיום המתודה cascading-cuts תדאג להכניס את הצומת x שהתבקשנו למחוק לערימה כשורש.

במקרה זה נקרא שוב לdelete על הצומת שהתבקשנו למחוק וכעת כשהוא נכנס כשורש המקרה יטופל והצומת יימחק כפי שפירטנו.

בכל אחד מהמקרים, נבצע בסוף קריאה לconsolidate.

ניתוח זמן הריצה : במקרה הראשון, זמן הריצה יהיה קבוע, במקרה השני הסיבוכיות תהיה חסומה ע''י סיבוכיות זמן הריצה של cut-add-to-middle-of-list שכפי שנתאר זמן ריצתה חסום במקרה הגרוע ע''י O(log(n)) .
במקרה השלישי נבצע cascading-cuts, ניתחנו את המתודה הנ''ל בביתה וראינו שזמן הריצה שלה במקרה הגרוע יהיה לוגריתמי באורך הקלט, אך חסם הamortized הוא קבוע.
פעולת ה-consolidate רצה בזמן O(log(n)) כפי שנתאר בהמשך.

**time complexity : worst case O(log(n))**

**time complexity for deleting node which is not a root : worst case O(log(n)), amortized cost O(1)**

cutAddToMIddleOfList(HeapNode child_to_disconnect):

מתודה שמטרתה לנתק ילד של צומת שעתיד להמחק מאביו, ולהכניס אותו במיקום שצומת האב היה קודם לכן.
נבצע מתודה זאת בצורה הבאה : נדאג כעת לשרשר את הצומת שקיבלנו אחרי הצומת שקדם לצומת האב ובצורה דומה לשרשר את הילד האחרון של האב לצומת שקודם לכן היה עוקב לצומת האב.
כעת נעבור בלולאה כמות איטרציות כגודל דרגת האב – בכל איטרציה נעבור על ילד אחר מילדי האב שאנו רוצים למחוק מהרשימה, נדאג לעדכן את שדה האב שלו וכמו כן כיוון שכעת הצומת הופך לשורש בפני עצמו נרצה לבטל את ה mark עליו אם יש כזה ( בו בזמן נזכור לעדכן שדות כמו marked_cnt ).
ניתוח זמן הריצה : אנו עוברים בלולאה כמות איטרציות כדרגת צומת האב – דרגה זו חסומה ע''י log(n), בכל איטרציה אנו מעדכנים שדות ולכן זמן הריצה של איטרציה קבוע. מלבד הלולאה אנו אכן רק מעדכנים שדות ומצביעים, פעולות בעלות זמן ריצה קבוע.
לכן זמן הריצה במקרי הגרוע יהיה O(log(n)).

**time complexity : worst case O(log(n))**

public void decreaseKey(HeapNode x, int delta):

מתודה שמטרתה להפחית ערך עבור צומת x בdelta .
נרצה לבדוק אם המפתח שלנו הינו שורש או לא – אם כן, פשוט נפחית את ערכו ורק נרצה לבדוק אם יש צורך בעידכון שדה המינימום, במקרה זה לעולם לא נפר את תכונת הערימה.
אם המפתח שלנו אינו שורש, נרצה לבדוק שערכו החדש אינו מפר את תכונת הערימה, אם אינו אז רק נוכל לשנות את ערכו, אם כן ייפר את תכונת הערימה נרצה לעשות cascading-cuts על הצומת.
זמן הריצה כפי שנלמד בביתה יהיה במקרה הגרוע O(log(n)) כיוון שזה החסם העליון על cascading-cuts אך זמן הריצה amortized יהיה O(1).

**time complexity : worst case O(log(n)),Amortized-cost O(1)**

public int nonMarked():

מתודה המחזירה את כמות הצמתים הלא מסומנים בערימה. כמות הצמתים הלא מסומנים זו בדיוק כמוך סך העצים פחות אלו שכן מסומנים, ואלו אנו שומרים בשדות לכן נוכל להחזיר את הנדרש ע''י חיסור שתי הערכים הללו השמורים בשדות.
זמן הריצה יהיה קבוע כיוון שאנו סך הכל ניגשים לשני שדות.

**time complexity : worst case O(1)**

public int potential():

מתודה המחזירה את הפוטנציאל של הערימה שהוא מחושב ע''י כמות העצים ועוד פעמיים הכמות המסומנים.
זמן הריצה יהיה קבוע כיוון שאנו סה''כ מבצעים חישוב אריתמטי בעזרת מידע שיש לנו בשדות.

**time complexity : worst case O(1)**

public void cascadingCuts(HeapNode node):

cascading-cuts פועלת כפי שלמדנו בביתה, מקבלת צומת וחותכת אותו מהוריו ( נשים לב שמתודה זו אנו מפעילים רק על צמתים שאינם שורשים ), בנוסף מתודה זו תדאג לסמן את ההורה במידת הצורך, ואם ההורה היה מסומן קודם לכן – נפעיל עליו את cascading-cuts.
זמן הריצה במקרה הגרוע יהיה חסום ע''י כמות הפעמים שנטפס למעלה – כיוון שגובה העץ הכי גדול לוגריתמי בגודל הקלט (מצב המתקבל כאשר יש לנו עץ יחיד מגודל n ואז גובהו יהיה log(n) ), נקבל ש cascading-cuts תפעל במקרה הגרוע בסיבוכיות זמן ריצה של O(log(n)). את חסם ה amortized ראינו בביתה ניתוח מדויק וכיוון שהביצוע זהה למה שראינו בביתה אז חסם ה amortized יהיה כפי שראינו, בזמן ריצה קבוע.

**time complexity : worst case O(log(n)), Amortized cost O(1)**

private void consolidate():

המתודה consolidate הינה מתודה שמטרתה לאחד שורשים מאותה דרגה לאחר מחיקת צומת. ניצור מערך בשם buckets מגודל log(n) – מערך זה יהיה המערך כך שבאינדקס i נשים צמתים מדרגה i, הדבר מסתדר עם גודל המערך כיוון שהצומת בדרגה הכי גדולה שיכול להיות בעץ יהיה בגודל log(n).
נעבור בלולאה על כל הצמתים שיש לנו בעץ, בכל איטרציה נפעיל בצורה הבאה : נשמור את דרגת הצומת הנוכחי במשתנה ונבדוק אם המערך במיקום זה אינו ריק. במידה וריק אז עוד כה לא מצאנו צמתים מאותה דרגה, נוכל לשים את הצומת הנוכחי במיקום זה במערך. במקרה בו המערך במקום זה תפוס – נבצע על הצומת הנוכחי והצומת במערך שדרגתן זהה link כפי שלמדנו בביתה ( נפרט בהמשך על המתודה ).
לסיום בלולאה נוספת נעבור על המערך buckets שמכיל את הצמתים החדשים המשורשרים מסודרים לפי הדרגות ( כלומר יש לנו מכל דרגה עץ יחיד ), במעבר בלולאה זו נדאג להכניס לפי הסדר את השורשים החדשים שלנו מהמערך buckets לערימה שלנו ולעדכן בהתאם שדות.
את ניתוח זמן הריצה של consolidate ראינו בביתה, אנו יודעים שבמקרה הגרוע זמן הריצה יהיה O(# of trees + log(n) = O(n) כיוון שייתכנו n עצים שונים בעת הקריאה למתודה זו, אך זמן הריצה amortized יהיה לוגריתמי באורך הקלט.

**time complexity : worst case O(n), Amortized cost O(log(n))**

private void cut(HeapNode node):

מתודת עזר שמטרתה לחתוך צומת מעץ. נרצה לעדכן את הורי הצומת הנחתך להיות null, לבטל לו סימון mark אם יש כזה, ולעדכן את שדות אחיו אם קיימים לו כאלה, כמו כן לעדכן את שדות ההורה ( rank,child ) במידת הצורך. נעדכן את כמות החיתוכים בעת ונקרא ל node-insert עם הצומת שחתכנו.
סיבוכיות זמן הריצה : נשים לב שכל מה שקורה ב cut הינו שינוי של שדות, מצביעים וקריאה ל node-insert, כל אלה לוקחות זמן ריצה קבוע ולכן המקרה הגרוע cut תעבוד ב O(1) .

**time complexity : worst case O(1)**

private HeapNode link(HeapNode x, Heapnode y):

מתודה לינק תעבוד כמו שלמדנו בכיתה, בהנתן זוג צמתים נבדוק מי בעל הערך הקטן מביניהם – את הצומת בעל הערך הגדול יותר נשרשר כבן שמאלי של הצומת בעל הערך הקטן, ולאחר מכן נעדכן את שדות הבנים הרלוונטים.

נרצה שהצומת שהכנסנו כבן שמאלי יהיה משורשר יחד עם הבנים הקודמים של הצומת הקטן שביצענו עליו link.

בנוסף נדאוג לעדכן שדות כמו דרגת הצומת הקטן, link_cnt ועוד.

סה''כ מתודה זו רק משרשרת צמתים ע''י עידכון שדותיהם ועל כן זמן הריצה יהיה קבוע.

**time complexity : worst case O(1)**

public static int totalLinks():

מתודה סטטית שתחזיר את כמות השרשורים שבוצעו מתחילת ריצת התוכנית – כיוון שאנו מחזיקים שדה סטטי שסופר קישורים, כל מה שנצטרך לעשות זה להחזיר אותו. מדובר בהחזרת ערך של שדה ולכן זמן הריצה יהיה קבוע.

**time complexity : worst case O(1)**

public static int totalCuts():

מתודה סטטית שתחזיר את כמות החיתוכים שבוצעו מתחילת ריצת התוכנית – כיוון שאנו מחזיקים שדה סטטי שסופר חיתוכים, כל מה שנצטרך לעשות זה להחזיר אותו. מדובר בהחזרת ערך של שדה ולכן זמן הריצה יהיה קבוע.

**time complexity : worst case O(1)**

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

מתודה שמטרתה להחזיר את k הערכים הקטנים ביותר בערימת פיבונאצ'י נתונה ( נשים לב שערימה זו תהיה עץ יחיד ע''פ הנחיות התרגיל ). נסביר את האלגוריתם באופן כללי ראשית ולאחר מכן נכנס יותר לפרטים :

ראשית נבדוק את מקרי הקצה בהם k = 0 או שקיבלנו עץ ריק ובמקרים הנ''ל נחזיר מערך ריק.

נאתחל מערך בשם k_min בגודל k אשר יהיה המערך שמכיל את כל הערכים של הצמתים שנרצה להחזיר.

ניצור ערימת מינימום חדשה שתהיה ערימת העזר שלנו ( מעתה ואילך נקרא לה בשם ערימת העזר ).

נשמור את המינימום של הערימה המקורית במשתנה עזר, נוסיף לערימת העזר שלנו צומת חדש – צומת עם הערך של המינימום שעכשיו שמרנו מהערימה המקורית.

**פרט טכני שחשוב להסביר : במהלך המתודה אנו נרצה למצוא צומת בערימה המקורית ע''י צומת במתודת העזר, את הדבר ביצענו ע''י הוספת שדה בשם corresponding_kmin_node שטיפוסו יהיה heapnode למחלקת heap-node, בכל פעם שנרצה להוסיף צומת לערימת העזר נעשה זאת כאשר אנו עוברים על צומת בערימה המקורית והצומת החדש שאנו יוצרים זה הערך של הצומת שעברנו עליו בערימה המקורית. באותו רגע אנו מעדכנים את השדה corresponding_kmin_node של הצומת שהתווסף לערימת העזר כך שיצביע על הצומת בערימה המקורית. בדרך זו בכל פעם שאנו מביטים בצומת בערימת העזר ורוצים לדעת איפה ברשימה המקורית הגענו לצומת שיש לו ערך זהה – פשוט נוכל לבדוק את שדה זה.**

כעת נתחיל לבצע לולאה שתיקח כ-k איטרציות, כל איטרציה נסמן באינדקס i : בכל איטרציה i נוסיף למערך במקום ה i את המינימום מערימת העזר. לאחר מכן נשמור במשתנה את הצומת עם הערך הזהה בערימה המקורית, ואז נמחק את המינימום מערימת העזר. אם לצומת ששמרנו כעת במשתנה ישנם ילדים, נוסיף את כולם לערימת העזר.

בכל איטרציה אנו מוסיפים איבר חדש למערך לכן בסיום יהיו בו k איברים, כאשר אנו מוסיפים את האיבר הקטן ביותר מערימת העזר שאליה אנו תמיד מוסיפים את הצמתים הקטנים ביותר מהערימה המקורית.

ניתוח הסיבוכיות :
ראשית אנו מוסיפים ומשנים מצביעים, אין מה להעמיק בתוכן הקודם ללולאה שכן זמן הריצה של כל פעולה בו הוא קבוע.
ניתוח גוף הלולאה :
בכל איטרציה אנו מגדירים משתנים ועיקר העבודה נעשה לצמתים שיש להם בנים – במקרה זה אנו עוברים על כל הבנים ומוסיפים אותם לערימת העזר. כל הוספה לערימת העזר קורית בזמן ריצה קבוע.
כעת, כיוון שבעץ בערימת פיבונאצי הדרגה המקסימלית היא של השורש, כמות הפעמים שהלולאה הפנימית רצה חסום ע''י דרגת השורש – deg(h).
לכן לסיום אנו מבצעים k איטרציות בהן בכל חזרה אנו מבצעים לכל היותר deg(H)*O(1) = O(deg(H)) פעולות ולכן זמן הריצה יהיה O(k*deg(H)) כנדרש.
לסיום בסוף אנו מחזירים את המערך k_min

**time complexity : worst case O(deg(H) * k )**

## חלק תיאורטי -

<u>שאלה 1</u>

א. סיבוכיות זמן הריצה תהיה O(m) -

ראשית אנו מבצעים בלולאה הראשונה m איטרציות שבכל חזרה אנו מכניסים איבר לערימה – זמן הריצה של הכנסת איבר לערימת פיבונאצ'י הוא קבוע ולכן m חזרות ייקחו O(m)

שנית, אנו מבצעים פעולת deleteMin - כפי שראינו בהרצאה זמן הריצה גם במקרה הגרוע ביותר יהיה לינארי באורך הקלט וamortized לוגריתמי באורך הקלט, כיוון שאנו גם ככה מבצעים את הלולאה הקודמת בזמן ריצה לינארי, נוכל להתייחס למקרה הגרוע ולכן זמן הריצה יהיה O(m)

לסיום אנו מבצעים log(m) פעולות בלולאה, כאשר בכל איטרציה אנו עושים decreaseKey שייקח O(1) – חסם זה הדוק, נפרט מדוע :
תחילה, נבחין בדבר הבא – decreasekey עובדת תמיד עם delta = m+1 > m-1 = maximum key in the heap
לכן, תמיד פעולה זו חותכת את הצומת שאנו מבצעים עליו את הפעולה – כלומר תמיד נקבל שהצומת שמבצעים עליו את decrease key יפר את תכונת הערימה ולכן נבצע cascading cuts , בפעולה זו נוסיף אותו כשורש חדש לרשימת השורשים.

באיטרציה הראשונה בלולאה, אנו מבצעים decreasekey על הבן הימני של השורש, הדבר ייגרום לכך שהוא ייצא מהעץ וייתווסף מחדש לרשימת השורשים.
באיטרציה הבאה נבצע decreasekey על הבן הימני של הבן השמאלי של השורש, שוב נוציא את הצומת מהעץ ונכניס אותו מחדש .
אנו עושים זאת באופן אינדוקטיבי וניתן לשים לב שבדרך זו לשום צומת לא ייחתכו שני בנים ( לכן לעולם לא נחתוך בן של צומת שמסומן ב marked ) - כלומר לעולם לא נמשיך לטפס למעלה בcasacading cuts ולכן כל פעולת decreasekey תעלה O(1).
לכן לולאה זו מתבצעת log(m) פעמים ובכל פעם זמן הריצה קבוע – מכך נקבל שזמן הריצה של לולאה זו סה"כ לוגריתמי באורך הקלט.

סה"כ קיבלנו שזמן הריצה חסום ע"י O(m) .

ב.

| m | Run-Time (ms) | totalLinks | totalCuts | Potential |
|---|---|---|---|---|
| $2^5$ | 0.6 | 31 | 5 | 14 |
| $2^{10}$ | 0.9 | 1023 | 10 | 29 |
| $2^{15}$ | 4.2 | 32767 | 15 | 44 |
| $2^{20}$ | 76 | 1048575 | 20 | 59 |

ג. כמות פעולות ה links מתבצעות בdeletemin שאנו עושים פעם אחת. כיוון שאנו מבצעים deletemin על ערימה עם m צמתים, וכיוון שm הוא חזקה של 2 נקבל עץ בינומי. אנו יודעים שבמקרה זה כמות פעולות השרשור הן m-1 כיוון שאנו יוצרים עץ יחיד.
כמות פעולות ה cut תהיה log(m) כיוון שאנו חותכים בלולאה רק צומת אחת כל פעם, והלולאה רצה log(m) פעמים ( אנו מדברים על הלולאה השנייה, בנוסף לכך הסברנו את כמות החיתוכים בכל איטרציה בסעיף א' ).
הפוטנציאל הוא מספר העצים ועוד פעמיים העצים המסומנים.

העצים בסיום יהיו העץ המקורי, וכל העצים שחתכנו – נקבל $\# of\ trees = \# of\ cuts + 1 = \log(m) + 1$
מספר העצים המסומנים יהיה כמספר החיתוכים פחות אחד כיוון שאיננו מסמנים את השורש של העץ המקורי בחיתוך הראשון, נקבל $\# of\ cuts - 1$

לסיום הפוטנציאל יהיה: $2 \cdot marked\ trees + \# of\ trees = 2(\log(m) - 1) + \log(m) + 1 = 3\log(m) - 1$

ד. נסביר את התהליך לאחר השינוי :
באיטרציה הראשונה אנו מבצעים פעולת decrease key על השורש (שורש של עץ בינומי יחיד) , הדבר לא ישנה כלום במבנה העץ כיוון שתכונת הערימה תשמר (רק הפחתנו מהמפתח של צומת שכבר לפני כן היה קטן משל בניו).
לאחר מכן בכל איטרציה נעשה את ה decrease key הבא לבן השמאלי, כיוון שאנו מפחיתים כל פעם באותו ערך הבן השמאלי תמיד יהיה קטן מההורה שלו ולכן תכונת הערימה תשמר – מכך נקבל כי אנו לעולם לא צריכים לחתוך צמתים מהעץ.

לסיכום, כמות ה links כמו מקודם תהיה רק בפעולת ה delete-min הראשונה נבצע m-1 שרשורים.
לא נבצע חיתוכים כלל.
בסיום רצף הפעולת העץ נשאר זהה, לכן מספר העצים הוא 1 ומספר הצמתים המסומנים הוא 0 – לא ביצענו שום חיתוך בשום שלב.

לסיום הפוטנציאל יהיה : $2 * marked\ trees + \# of\ trees = 2 * 0 + 1 = 1$

ה. אם נמחק את שורה 2# איננו עושים delete-min ולכן איננו עושים consolidate בשום שלב, כלומר מההתחלה ועד הסוף נהיה עם m+1 עצים ( ככמות הצמתים שהכנסנו ) .

כמות ה links תהיה 0 כיוון שאיננו עושים consolidate.
כמות ה cuts תהיה 0 כיוון שאנו עושים רק decrease key על שורשים של עצים בגודל 1, כלומר אין מה לחתוך ותכונת הערימה נשמרת באופן טריוויאלי.

הפוטנציאל יהיה: $\# of\ trees + 2 * marked\ trees = m + 1 + 2 * 0 = m + 1$
הדבר נובע מכך שכפי שהסברנו נהיה עם m+1 עצים בסוף לאחר m+1 הכנסות, אנו תמיד עושים decrease-key לשורשים ולכן לא נסמן צמתים לעולם.

ו. ראשית, נשים לב שבסעיף זה, עד הוספת הפעולה אנו מבצעים את אותן הפעולות כפי שמתוארות בסעיף ג', כלומר אם נבין מה מוסיפה הפעולה החדשה נוכל להוסיף את הערכים שחישבנו בג'.

הפעולה החדשה היא פעולת decrease-key על הצומת האחרון שחתכנו לו בן – זה יהיה הצעצע השמאלי ביותר בעץ שהתחלנו איתו (כרגע הוא השורש האחרון ברשימת השורשים). נשים לב שכל הצמתים במסלול מהשורש עד לצומת הנ''ל מסומנים כי חתכנו להם קודם לכן בן, ולכן נבצע cascading cuts עד לשורש. נשים לב שאלו ואלו בלבד הצמתים המסומנים ולכן בסוף פעולת הdecrease-key הנוספת לא יישארו צמתים מסומנים.
מכיוון שנצטרך לחתוך כל צומת עד השורש, מספר פעולות הcut שיתווספו לאחר הפעולה יהיה log(m)-1 (באורך המסלול עד לשורש). כלומר אם נוסיף לכמות החיתוכים שהיו בסעיף ג' נקבל שסך כל החיתוכים עבור מקרה זה הוא 2log(m) + 1.
מספר הlinks יישאר זהה לאחר הפעולה, מכיוון שפעולת decrease-key לא קוראת לconsolidate/link, כלומר מספר הlinks עדיין m-1 כמו בסעיף ג'.
נחשב את הפוטנציאל. כפי שציינו, מספר העצים המסומנים בסוף הפעולות יהיה 0. מספר העצים יגדל ב- log(m) – 1 לאחר פעולת הdecrease-key האחרונה, כלומר הוא יהיה 2log(m). לכן הפוטנציאל יהיה

$$\#trees + 2 * \#marked\ trees = 2\log(\mathrm{m})$$

נשים לב כי פעולת הdecrease-key היקרה ביותר תהיה האחרונה, מכיוון שהיא תצטרך לעבור על כל גובה העץ כדי לבצע cascading cuts עד למעלה, לעומת שאר פעולות הdecrease-key שבהן ההורה של הצומת שנחתך לא מסומן ולא נעלה עד למעלה. כפי שאמרנו פעולה זו תרוץ בזמן O(log(m)).

טבלה מסכת עבור שאלה 1:

| Question Number | Case | totalLinks | totalCuts | Potential | decrease-key max cost |
|---|---|---|---|---|---|
| c | Original | m-1 | log(m) | 3log(m) - 1 | - |
| d | decreaseKey(m-2^i) | m-1 | 0 | 1 | - |
| e | Remove line #2 | 0 | 0 | m+1 | - |
| f | Add line #4 | m-1 | 2log(m)-1 | 2log(m) | log(m) - 1 |

שאלה 2

א.

| m | Run-Time (ms) | totalLinks | totalCuts | Potential |
|---|---|---|---|---|
| 728 | 1 ms | 723 | 0 | 6 |
| 6560 | 4 ms | 6555 | 0 | 6 |
| 59048 | 20 ms | 59040 | 0 | 9 |
| 531440 | 133 ms | 531431 | 0 | 10 |
| 4782968 | 1257 ms | 4782955 | 0 | 14 |

ב. זמן הריצה האסמיפטוטי הוא O(mlog(m)).
נבצע ניתוח סיבוכיות עבור הפעולות. אנו מבצעים m הכנסות כאשר זמן הריצה להכנסה קבוע. לאחר מכן אנו מבצעים O(m) פעולות delete-min וראינו חסם amortized על סדרת פעולות delete-min של log(m) לפעולה, לכן נקבל שזמן הריצה הכולל של סדרת הפעולות של delete-min הוא O(mlog(m)) ולכן זה חוסם גם את זמן הריצה הכולל.

ג. מספר פעולות ה-**cut** יהיה 0 מכיוון שאנחנו לא מבצעים cascading cuts (delete-min תמיד מוחק שורש). נחשב את **הפוטנציאל**. נשים לב כי מספר הצמתים המסומנים הוא 0 (כפי שציינו קודם, אנחנו חותכים רק שורשים). לכן הפוטנציאל יהיה כמספר העצים בערימה. מספר העצים בסוף התהליך יהיה כמספר האחדים בייצוג הבינארי של m/4. נסביר: בערימה בינומית רגילה, מספר העצים הוא תמיד תואם את מספר האחדות בייצוג הבינארי שך כמות האיברים בעץ. זאת מכיוון שכל אחד מעיד על כך שיש עץ בדרגה של המיקום שלו. כלומר, עבור ערימה עם 20 איברים, היא תיוצג ע"י עץ בינומי מדרגה 4 ועץ בינומי מדרגה 2, מכיוון שיש $2^k$ איברים בעץ מדרגה k, אז $2^4 + 2^2 = 16 + 4 = 20$. ניתן לראות שזה בדיוק תואם ייצוג בינארי. מכיוון שאנחנו מסתכלים על סדרה של delete-min, אחרי כל פעולה יהיה consolidate ולכן אנחנו יכולים להתייחס לערימה כערימה בינומית, כי לא יהיו 2 עצים מאותה דרגה לאחר סיום הפעולה (וכל העצים יהיו עצים בינומים כי לא נבצע cascading cuts).

כלומר, מספר העצים בסיום, שהוא גם הפוטנציאל במקרה הזה, יהיה כמספר האחדים בייצוג הבינארי של m/4.
מספר פעולות ה**link** הוא m-#trees at end.