מעשית 2 - תיעוד

HeapNode

fields:

string **value** - המידע/הערך
int **key** - המפתח
int **rank** - הדרגה
boolean **mark** - אינדיקטור לסימון
HeapNode **child** - הבן השמאלי
HeapNode **next** - האח מימין
HeapNode **prev** - האח משמאל
HeapNode **parent** - ההורה
HeapNode **realHeap** - kmin יוסבר בהמשך תחת המתודה

המתודות

המתודות הן Set-ריות או Get-ריות, לכן בבירור כולן רצות בזמן קבוע.

שם המתודה	תיאור המתודה
public HeapNode(String info,int key)	בנאי צומת על פי תוכן ומפתח
public HeapNode(int key)	בנאי צומת על פי מפתח בלבד (התוכן יהיה null)
public int getKey()	מחזירה את המפתח של צומת
public void setKey(int key)	מעדכנת את המפתח של צומת להיות key (הארגומנט)
public Boolean isMarked()	מחזיר true אם ורק אם הצומת מסומן
public void increaseRank()	מגדילה את דרגת הצומת ב1
public int getRank()	מחזירה את דרגת הצומת
public void setRank(int rank)	מעדכנת את דרגת הצומת להיות rank (הארגומנט)
public void mark()	מסמנת את הצומת
public void unmark()	מסירה סימון מהצומת

public void setPrev(HeapNode prev)	מקבלת צומת כארגומנט ומגדירה את prev של הצומת להיות הצומת שהתקבל
public void setNext(HeapNode next)	מקבלת צומת כארגומנט ומגדירה את next של הצומת להיות הצומת שהתקבל
public void setParent(HeapNode parent)	מקבלת צומת כארגומנט ומגדירה את parent של הצומת להיות הצומת שהתקבל
public void setChild(HeapNode child)	מקבלת צומת כארגומנט ומגדירה את child של הצומת להיות הצומת שהתקבל
public HeapNode getParent()	מחזירה את האבא של הצומת בעץ או null אם הצומת הוא שורש
public HeapNode getChild()	מחזיר את אחד הבנים של צומת או null אם הצומת הוא עלה
public HeapNode getNext()	מחזיר את הצומת הבא ברשימה המקושרת של צומת: אם הצומת הוא שורש של עץ אז את שורש העץ הבא בערימה. אחרת, מחזיר את הבן הבא של האב של הצומת. בשני המקרים אם הצומת הוא יחיד ברשימה הוא הצומת המוחזר.
public HeapNode getPrev()	מחזיר את הצומת הקודם ברשימה המקושרת של צומת: אם הצומת הוא שורש של עץ אז את שורש העץ הקודם בערימה. אחרת, מחזיר את הבן הקודם של האב של הצומת. בשני המקרים אם הצומת הוא יחיד ברשימה הוא הצומת המוחזר.

FibonacciHeap

fields:

HeapNode **min** - מצביע לאיבר בעל המפתח המינימאלי בערימה
int **Trees** - מספר העצים בערימה
int **mark_number**- מספר העצים המסומנים בערימה
int **size** - מספר הצמתים בערימה
int **links**- כלל פעולות הלינק שהתבצעו
int **cuts** - כלל פעולות הניתוק שהתבצעו

מתודות:

public FibonacciHeap() - בנאי ריק של הערימה. זמן קבוע
public FibonacciHeap(HeapNode min) - בנאי ערימה שמקבל צומת ומגדירו כמינ' הערמה. זמן קבוע
public boolean IsEmpty() - הפונ' מחזירה 'אמת' אמ"מ הערימה ריקה. זמן קבוע.
public void setMin(HeapNode m) - עדכון שדה הצומת המינימאלי לזה שהועבר. זמן קבוע
public void insert_helper(HeapNode h) - הכנסת העץ עם השורש המוזן אל הערימה. זמן קבוע
public HeapNode insert(int i) - מייצרת צומת עם המפתח שהועבר, ומכניסה אותו לערימה
זמן קבוע.

public void deleteMin()
מוחקת את הצומת שמכיל את המפתח המינימלי בערימה. המתודה מבצעת successive linking כפי שראינו בכיתה (ישנה מתודת עזרת של sucsseive linking). סיבוכיות WC ליניארית.
Amortized: O(logn).

public void successive linking ()
מבצעת תהליך consolidating כמו שראינו בכיתה. סיבוכיות WC: לינארית.(Amortized: O(log n.

public HeapNode one_link(HeapNode A ,HeapNode B)
מבצעת פעולת link בודדת בין הצמתים שהתקבלו. הערך המוחזר הוא השורש החדש [אחד מהצמתים כמובן], שמתקבל בהתאם לכלל הערימה. זמן קבוע.

public int degrees_max ()
מחזירה חסם עליון על הדרגה המקסימאלית שתיווצר אחרי ביצוע תהליך consolidating על הערימה.
סיבוכיות: זמן קבוע.

public void unmark ()
מבצעת מעבר על כל שורשי העץ. התוצר: כל השורשים יהיו לא מסומנים.
WC של O(n), כאשר בפועל נקראת רק אחרי consolidating, לכן (o(logn.

public HeapNode update_our_father (HeapNode b)
מבצעת מעבר על כל הרשימה המקושרת של הצומת b (על כל האחים שנמצאם באותו "דור") ומגדירה את האב שלהם להיות null; הערך המוחזר הוא הצומת עם המפתח המינ'.
סיבוכיות: WC ליניארית, אך בדומה למתודה הקודמת בפועל נקראת רק על רשימה של צאצאי השורש, ולכן לוגריתמית.

public HeapNode findMin()
מחזירה את האיבר המינ' בערימה. זמן קבוע.

public void meld (FibonacciHeap heap2)
מבצעת מיזוג בין הערימה ל-heap2. זמן קבוע.

public int size()
מחזירה את מספר הצמתים הכולל בערימה. זמן קבוע.

public void delete(HeapNode x)
מוחקת את הצומת x מהערימה. הפונ' קוראת לdecrease key כדי להוריד את ערך המפתח של x להיות
המפתח המינ' בערימה, ואז ל-delete min כדי למחוק את x. סיבוכיות: במקרה הגרוע ליניארית, בזמן ממוצע: log n.

public void decreaseKey(HeapNode x, int delta)
מקטינה את המפתח של x בערך delta. במידת הצורך הפונ' חותכת את הצומת ומתחילה תהליך של cascading cuts, שזמן הamor שלה, לפי ההרצאה, הוא קבוע. סיבוכיות: WC של (O(log n, ואמורטייז של זמן קבוע.

public void cascading_cuts(HeapNode node)
ביצוע התהליך cascading cutss שלמדנו בהרצאות, על הצומת x. נעזרת במתודה single_cut [שבזמן קבוע].
WC: O(logn). Amortized: O(1)

public void single_cut(HeapNode node)
חותכת את העץ ששורשו x מאביו ומוסיפה אותו לרשימת השורשים. זמן קבוע.

public static int totalLinks()
מחזירה את המספר הכולל של פעולות ה link שבוצעו סך הכל. זמן קבוע.

public static int totalCuts()
מחזירה את המספר הכולל של פעולות ה link שבוצעו סך הכל. זמן קבוע.

()public int[] countersRep

המתודה יוצרת מערך כגודל הדרגה המקסימלית של העץ שנמצא בערימה, כאשר דרגה של עץ מוגדרת להיות מספר הילדים של העץ הנתון.

תחילה, נעבור על כל העצים שנמצאים בערימה בשביל למצוא את העץ בעל הדרגה הגבוהה ביותר בערימה.
(אנו עוברים על השורשים של העצים בערימה, ועל כל שורש משתמשים בשדה rank שמחזיר את מספר הילדים של השורש)

לאחר מכן, ניצור מערך int של אפסים כגודל הדרגה המקסימלית.

לבסוף נעבור שוב על כל העצים בערימה, ובכל פעם שנתקל בעץ שדרגתו היא i אז נעלה את הערך של המערך במיקום הi בפלוס 1.

לבסוף נקבל מערך דרגות של הערימה כפי שרצינו.

הסיבוכיות של המתודה היא כמספר העצים בעץ, לכן סיבוכיות WC הוא O(N).

()public int potential

המתודה מחזירה את הפוטנציאל הנוכחי של הערימה, כלומר:

Potential = #trees + 2*#marked

אנו מחזיקים את השדות marked_nodes,Trees של הערימה, לכן סיבוכיות הזמן של מתודה זאת הוא זמן קבוע.

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

המתודה מקבלת עץ בינומי **H** כלשהוא ומספר טבעי **k** כך **k<size(H)** ומחזירה מערך **int** בגודל **k** עם **k** הצמתים הקטנים ביותר ב**H**.

בשביל חישוב הסיבוכיות נסמן:

r – הדרגה של **H**, כלומר מספר הילדים של השורש של **H**.

לכן, על פי הגדרת עץ בינומי נובע שיש ב**H** בדיוק 2^r צמתים.

<u>טיפול במקרי קצה:</u>

אם **k=0** או ש**H** הוא עץ ללא צמתים נחזיר מערך ריק

אם **k=1** אז נצטרך להחזיר רק את האיבר הקטן ביותר ב**H** והוא על פי הגדרה השורש של **H**, לכן נחזיר את המפתח של השורש.

אחרת (**1<k**):

מהנתון ש**k<size(H)** נובע ש**$k<2^r$**, לכן אם נפעיל $\log_2(*)$ על שני האגפים החיוביים נקבל:

$$\log_2(k) < \log_2(2^r) = r$$

כלומר: $\log_2(k) < r$, נשתמש בנתון זה בחישוב הסיבוכיות בהמשך.

ניצור מערך **int** בגודל **k** שעתיד להחזיק את **k** הצמתים הקטנים ביותר בעץ בינומי.

כפי שציינתי קודם, השורש בעץ בינומי הוא תמיד הצומת הקטן ביותר בעץ לפי הגדרה, לכן נכניס את המפתח של השורש למיקום הראשון במערך **int** שלנו, נותר לנו למצוא **k-1** צמתים.

ניצור ערימת פיבוצני חדשה בשם **helperheap**.

נשים לב שהאיבר השני בגודלו חייב להיות ילד של השורש, לכן הוא בהכרח נמצא בין **r** הילדים של השורש.

לכל ילד של השורש ניצור **heapnode** <u>חדש</u> בעל אותו מפתח בדיוק ונכניס ל**helperheap**.

בנוסף, נעדכן את השדה **realheap** של כל צומת שהכנסנו ל**helperheap** להצביע על הצומת המקורי ב**H** בעל אותו מפתח.

realheap: שדה של ה**heapnode** שמטרתו מיועדת רק למתודה **kmin** שבאמצעותה נוכל לדעת עבור כל צומת ב**healpermin** היכן ממוקם הצומת המקורי ב**H**.
הסיבה שאנו צריכים את השדה הזה יפורט בהמשך.

סיבוכיות הזמן לביצוע העדכון הוא זמן קבוע.

איטרציה ראשונה:

הכנסו ל**helperheap** בדיוק **r** צמתים**,** כעת ב**helperheap** יש בדיוק **r** צמתים.
נמצא את המינימום של הערימה – לוקח זמן קבוע.

נוסיף אותו למיקום השני במערך **int**.

לאחר מכן אנו למצוא את הצומת השלישי בגודלו ב**H**,

לכן המיקום האפשרי שלו הוא:

1)או מבין **k-1** הילדים של השורש (ללא האיבר השני בגודלו שמצאנו)

2)מבין הילדים של האיבר השני שמצאנו.

כמה ילדים יש לאיבר השני? לכל היותר **r**, זה נובע על פי הגדרת עץ בינומי השורש הוא בעל הדרגה הגבוהה ביותר בעץ, או בדרך אחרת לראות זאת הוא הילדים של כל צומת הם שורשים של עצים בינומים שדרגתם קטנה ממש הצומת של האבא.

לכן בהכרח לצומת השני יש לכל היותר **r** ילדים, ובאופן כללי לכל צומת בעץ יש לכל היותר **r** ילדים.

לכן ניצור מצבעי זמני ל**realheap** של האיבר השני (שמצביע למיקום של האיבר המקורי בעץ)

נבצע **delete-min** ל**helperheap** סיבוכיות זמן **O(r) WC** אבל אמורטייז $O(\log r)$

ונעבור לאיטרציה השנייה.

עבור האיטרציה הi כאשר i>1:

נוסיף את הילדים של האיבר הi בגודלו לhelperheap.

נמצא את המינימום ונוסיף את המפתח שלו למיקום הi+1 במערך int שלנו

ניצור מצביע לצומת בH שהוא המיקום המקורי של המינימום

נמחוק את המינימום מhelperheap.

לאחר k-1 איטרציות אנו מקבלים מערך int בגודל k שמכיל את k המפתחות הקטנים ביותר בH כרצוי!

<u>חישוב סיבוכיות של האלגוריתם:</u>

אנו מבצעים k-1 לולאות באלגוריתם, נחשב את הסיבוכיות של כל לולאה:

אנו עוברים על לכל היותר r ילדים של צומת ומכניסים אותם לhealperheap, לכן פעולה זאת לוקחת O(r)

אנו מחפשים את הmin על פני העץ – תמיד לוקח זמן קבוע

מוסיף את המפתח של הmin למיקום ידוע מראש במערך הint (למיקום הi+1 כאשר i הוא מספר האיטרציה)

בנוסף, נגשים לשדה של הmin- לוקח O(1)

והפעולה האחרונה מבצעים delete min (נחשב את הסיבוכיות בהמשך בנפרד)

לכן סה"כ הסיבוכיות של האלגוריתם הוא:

$$O\big((k-1)*r\big)+O((k-1)*1))+O(total\ cost\ of\ delete\ \min)$$
$$=O(kr)+O(total\ cost\ of\ delete\ \min)$$

ניתן חסם O גדול לסיבוכיות *total cost of delete min*:

באיטרציה הראשונה יש בעץ r צמתים, לכן הסיבוכיות זמן ב*wc* הוא **O(r)** אבל אמורטייז $O(\log r)$

בכל איטרציה נוספת אנו מוסיפים לעץ לכל היותר r איברים, כלומר שבאיטרציה הi יש לכל היותר $i*r$ צמתים בhelperheap שלנו.

לכן כאשר נבצע delete min באיטרציה הi, אז יש בעץ לכל היותר $i*r$ צמתים, לכן הסיבוכיות זמן הממוצעת הולכת להיות $O(\log i*r)$

כלומר, העלות הכוללת של כל פעולות הdelete min עבור k-1 האיטרציות חסום על ידי:
(הlog ללא בסיס הוא log בבסיס 2)

$$
\begin{aligned}
&O(\log(r)) + \mathrm{O}(\log(2r)) + O(\log(3r)) + \cdots + \mathrm{O}(\log\big(r * (k-1)\big) \\
&\qquad = c1 * \log(r) + c2 * \log(2r) + \cdots + \mathrm{c}_{k-1} * \log\big(r * (k-1)\big) \\
&\qquad \leq C * (\log(r) + \log(2r) + \cdots + \log\big(r * (k-1)\big) \\
&\qquad = O(\log(r) + \cdots + \log\big(r * (k-1)\big) = O(\log(r * \ldots * r(k-1)) \\
&\qquad = O\big(\log\big(r^{k-1} \ * (k-1)!\big)\big) = O\big(\log\big(r^{k-1}\big) + \log(k-1)\,!\big) \\
&\qquad = O(\mathrm{klog}(r) + klog(k))
\end{aligned}
$$

כאשר השתמשנו בכך שמתקיים:

$$\log(k!) = O(klog(k))$$

וגם השתמשנו בכך שמתקיים:

$$\log\big(r^k\big) = klog(k)$$

לכן בסה"כ העלות **<u>הכוללת</u>** של פעולות הdelete min חסום ע"י $O(\mathrm{klog}(r) + klog(k))$

כעת נשתמש בעובדה ש $\boldsymbol{\log_2(k) < r}$ בשביל לקבל שהעלות הכוללת של פעולות **delete min** חסומה ע"י:

$$O\big(\mathrm{klog}(r) + klog(k)\big) = O(k * \log(r) + k * r) = O(k * r)$$

לכן בסה"כ העלות הכוללת של התוכנית:

$$= O(kr) + O(total\ cost\ of\ delete\ \mathrm{min}) = O(kr) + O(kr) = O(kr)$$

כלומר הסיבוכיות היא $O(k{\cdot}\mathrm{deg}(H))$ כרצוי!