

מבני נתונים - פרויקט מס' 2 – ערכות מתקדמות

חלק מעשי – תיעוד

בפרויקט זה כתבנו מחלקה בשם Heap המממשת סוג עריםות שונות מעל מספרים שלמים חיוביים. המחלקה תומכת בפעולות מימושית ומאפשרת לבחור (בעת יצירת העירמה) האם להשתמש בlazy melds או האם להשתמש בkeys lazy. מטרת המחלקה היא לאחסן זוגות של Key-Value וلتמוך בעשרות פעולה עירמה עילוות.

תפקידו של כל חבר במחלקה

- self.lazyMelds – שדה בוליאני הקובע האם העירמה עבדת במצב של "מיוזג עצל". אם הערך true, פעולות מיוזג והכנסה הן מהירות והסידור נדחה. אם false, מתבצע סידור (successive linking) בכל פעולה הכנסה ומיוזג.
- self.lazyDecreaseKeys – שדה בוליאני הקובע את אופן הטיפול בהקטנת מפתחה. אם הערך true, מתבצע שימוש במנגנון-hs-Cascading Cuts של עירמת פיבונacci. אם false, מתבצעת פעולה Heapify-Up סטנדרטית.
- self.min – שדה המחזיק מצביע לאובייקט בעל המפתח המינימלי הנוכחי בעירמה.
- self.head – שדה המחזיק מצביע לאיירר הראשון ברשימה השורשים של העירמה.
- self.last – שדה המחזיק מצביע לאיירר האחרון ברשימה השורשים של העירמה.
- self.size – שדה המחזיק את מספר האיברים הכלול בעירמה.
- self.numTrees – שדה המחזיק את מספר העצים הנוכחיים ברשימה השורשים.
- self.numMarkedNodes – שדה המונונה את כמות הצלמתים המסומנים בעירמה.
- self.totalLinks – מונה הסופר את סך פעולות הקישור (Cut) שבוצעו בין שני עצים לאורק חי העירמה.
- self.totalCuts – מונה הסופר את סך פעולות החיתוך (Cut) שבוצעו לאורק חי העירמה.
- self.totalSwap – מונה הסופר את סך פעולות החלפה (Swap) שבוצעו במסגרת תהליך Heapify-Up.

תיאור הפונקציות

- **insert** – הפונקציה מכניסה אייר חדש לעירמה עם המפתח והמידע הנתונים. היא מחזירה את האובייקט HeapItem החדש שנוצר, המכיל מצביע לצומת בעירמה. מבחינה אופן הפעולה, הפונקציה יוצרת תחילת צומת חדש ומאתחלת את מצביעי next וprev שלו להצביע על עצמו. לאחר מכן, היא יוצרת עירמה חדשה (heap2) המכילה רק את האירר החדש, וביצעת קרייה לפונקציה meld כדי למציג את העירמה הזמנית לתוך העירמה הנוכחיית.
ניתוח סיבוכיות: זמן הריצה תלוי בערךו של השדה lazyMelds:
 - אם lazyMelds הוא true, פעולה meld מבצעת שרשור של רשימות השורשים בלבד, ללאsuccesiveLinking. פעולה זו כוללת מספר קבוע של עדכוני מצביעים. לכן, זמן הריצה הוא $O(1)$.
 - אם lazyMelds הוא false, פעולה meld קוראת ל-successiveLinking. במקרה הגראוע ביותר (בו חיבור העץ החדש גורר רצף של מיוזגים), יתכונו מיוזגים עד לגובה העץ המקסימלי. לאחר גובה העץ חסום לוגריתמית, זמן הריצה במקרה הגראוע הוא $(logn)O$.
- **findMin** – הפונקציה מחזירה את האובייקט בעל המפתח המינימלי בעירמה. הפונקציה ניגשת לשדה min, אשר מחזיק מצביע לאירר המינימלי הנוכחי. אם העירמה ריקה, מוחזר ערך null.

ניתוח סיבוכיות: המצביע למינימום מתחזק באופן שוטף ולכן הגישה אליו קוראת בזמן קבוע של $O(1)$.

- **deleteMin** - הפונקציה מוחקת את האיבר המינימלי מהירימה. תחילת הפונקציה מטפלת במקרה קצה (עירמה ריקה או עירמה עם איבר יחיד). לאחר מכן, היא מוציאה את איבר המינימום מרשימת השורשים וمعدכנת את המצביעים ואת גודל העירמה בהתאם. אם לאיבר שנמחק היו ילדים, הפונקציה הופכת אותם לשורשים עצמאיים. לבסוף, הפונקציה קוראת ל-*successiveLinking* כדי לבצע פעולה *consolidate* ולסדר את העירמה מחדש, ועוד שורקת את רשימת השורשים החדשה כדי למצוא ולעדכן את מצביעו *min* החדש.

ניתוח סיבוכיות: זמן הריצה תלוי בכמות העצים ברשימה השורשים ובדרגת האיבר המינימלי שנמחק.

- במקרה של *lazyMelds=true*, יתכן שרישמת השורשים מכילה $(n)O$ עצים (עקב רץ הכנסות ללא סידור). פעולה *successiveLinking* ועדכון *min* יאלצו לסרוק את כל העצים הללו. לכן, זמן הריצה במקרה הגרוע הוא $(n)O$. עם זאת, ניתן כי עלות ה-*amortize* במצב זה היא $O(log n)$.
- במקרה של *lazyMelds=false*, רשימת השורשים תמיד מכילה לכל היוטר $(log n)O$ עצים. לכן, תהליך המחקה, הקישור מחדש ועדכון המינימום חסומים כולם על ידי גובה העץ. זמן הריצה הוא $(log n)O$ ב-WC.

- **decreaseKey** - הפונקציה מקטינה את המפתח של איבר נתון (x) בערך מסוים (*diff*) ושומרת על חוקיות העירמה. תחילת הפונקציה מעדכנת את המפתח של האיבר ובודקת אם הוא קטן מהמינימום הנוכחי (ומעדכנת את *min* בהתאם). לאחר מכן, אופן הפעולה תלוי בדגל *cascadingCut*:
 - אם הוא *true*, נבדק האם הופר כלל העירמה. במידה וכן, נקוראת הפונקציה *cascadingCut* כדי לנתק את הצומת ולטפל בשרשראת הסימונים במעלה העץ.
 - אם הוא *false*, נקוראת הפונקציה *heapifyUp* כדי לבצע פעולה של האיבר כלפי מעלה.

ניתוח סיבוכיות: נראה כי זמן הריצה בWC הוא $(log n)O$

- במקרה הראשון (*if*): הפונקציה מבצעת *cascadingCut*. במקרה הגרוע, בהתאם לערך של *lazyMelds*, עלות הריצה היא $(n^2 log n)O$ או $(log n)O$.
- במקרה השני (*else*): הפונקציה מבצעת *heapifyUp* שעולותה חסומה בגובה העץ ולכן זמן הריצה יהיה $(log n)O$.

heapifyUp - פונקציית עזר המעליה צומת במעלה העץ עד שהחוקיות העירמה משוחזרת. הפונקציה רצתה בולאה כל עוד הצומת אינו שורש ומפתחו קטן ממפתח אביו. בכל איטרציה מתבצעת החלפה (באמצעות קרייה למולדת *swapWithParent*) והצומת מתקדם לאביו.

ניתוח סיבוכיות: במקרה הגרוע, הצומת יעליה מהעליה העמוק ביותר ועד לשורש. על כן, זמן הריצה בWC הוא $(log n)O$.

- **swapWithParent** - פונקציית עזר המבצעת החלפה בין צומת לאביו. החלפה מתבצעת על ידי החלפת התוכן של *HeapItem*-*item* בין שני הצומטים, ועדכון המצביעים המתאים בתוך ה-*item*. כמו כן, הפונקציה מעדכנת את המצביעים/global/*head*, *last*, *min* (במידת הצורך).

ניתוח סיבוכיות: הפונקציה מבצעת פעולות בזמן קבוע בלבד, ולכן זמן הריצה הוא $(1)O$.

cascadingCut - פונקציה רקורסיבית המבצעת את מנגנון הניתוק המדורג. תחילת הפונקציה קוראת *cut* כדי לנתק את הבן (x) לאביו (y). לאחר מכן היא בודקת את האב: אם האב אינו שורש והוא אינו מסומן (*marked*), נסמן אותו (*marked=true*) והתחליך עוצר. אם האב כבר מסומן, הסימון מוסר (*marked=false*), והפונקציה קוראת לעצמה רקורסיבית כדי לנתק גם את האב לאביו שלו.

ניתוח סיבוכיות: במקרה הגרוע ביותר, שרשת הניתוקים נמשכת עד לשורש העץ. כאשר *marked=True*, זמן הריצה הוא $(log n)O$, אחרת כל פעולה *cut* היא בזמן ריצה של $(log n)O$ ולכן סה"כ $(n^2 log n)O$.

- **cut** - פונקציית עזר המנתקת קשר בין צומת (x) לאביו (y) והופכת את x לשורש חדש בעירמה. הפונקציה מעדכנת את מצביעי הילדים של האב, מקטינה את דרגתו, ומפסיקת את סימון ה-*marked*.

של הבן. לאחר מכן, הבן מתווסף לרשימת השורשים של הערימה באמצעות יצרת עירמה זמנית וקירה ל-*meld*.

ניתוח סיבוכיות: הפונקציה מבצעת מספר קבוע של עדכוני מצביעים וקוראת ל-*meld*. בהנחה שהערימה מוגדרת עצלה (*lazyMelds=true*), פועלות המיזוג היא $O(1)$ וכן גם זמן הריצה של המתוודה. במקרה *I*, *lazyMelds=false*, הסיבוכיות תהיה $O(log n)$.

delete - הפונקציה מוחקת איבר כליל (x) מהערימה. הימימוש מתבצע על ידי הקטנת המפתח של האיבר לערך המינימלי האפשרי כדי להבטיח שהוא יהיה הקטן ביותר בערימה. לאחר שינוי המפתח, הפונקציה קוראת ל-*deleteMin* כדי להוציא את האיבר מהערימה.

ניתוח סיבוכיות: הפונקציה מורכבת משני שלבים: *decreaseKey* שועלתו $O(log n)$ במקרה הגרוע, ו- *deleteMin* במקרה שהערימה מוגדרת כל-עצלה (*lazyMelds=false*), גם *deleteMin* חסומה ב- $O(log^2 n)$ אם *lazyDecreaseKeys=false* אחרת חסומה ב $O(n)$, אם *lazyDecreaseKeys=true* חסומה ב $O(n log n)$. בפרט, בWC זמן הריצה הוא $O(n)$.

meld - הפונקציה ממזגת את הערימה הנוכחיית עם עירמה נוספת (*heap2*). תחילת הפונקציה מרשרת את רשימת השורשים של הערימה השנייה לקצה רשימת השורשים של הערימה הנוכחיית. לאחר מכן היא מעדכנת את השדות הסטטיסטיים (*size, numTrees, numMarkedNodes*) וכוכ'ו) ואת המינימום הכלובלי. לבסוף, נבדק את הערך של *lazyMelds*: אם הוא *false*, מתבצעת קריאה ל- *successiveLinking* כדי לסדר את הערימה המאוחדרת.

ניתוח סיבוכיות: מתחילה לפי ערכו של השדה *lazyMelds*:

- אם *lazyMelds=true*: מתבצע שרשור מצביעים בלבד ועדכו'ן שדות, ולן הזמן הוא $O(1)$.
- אם *lazyMelds=false*: מתבצעת קריאה ל- *successiveLinking* הסורקת את כל השורשים.

במקרה הגרוע, מספר השורשים פרופורציונלי ל n. לכן, הזמן הריצה הוא $O(n)$.

successiveLinking - פונקציית עזר המבצעת את ההליך *Consolidation*. הפונקציה אחראית על ארגון מחדש של רשימת השורשים כך שלא ישארו שני עיצים בעלי אותה דרגה. היא יוצרת מערך עזר בגודל לוגריתמי, מתחילה אותו ב-*null*, וקוראת לפונקציה *consolidate*. לאחר מכן היא מעדכנת את מצביעי *head* וה-*last* בהתאם לשיטה החדששה שתתקבלה.

ניתוח סיבוכיות: הפונקציה תלואה ישירות בפונקציה *consolidate* הסורקת את כל שורשי הערימה. בפרט, במקרה הגרוע כמות השורשים לינארית בכמות האיברים ולן זמן הריצה הוא $O(n)$.

link - פונקציית עזר המחברת שני עיצים בעלי דרגה זהה. הפונקציה קובעת מי מהשורשים הוא בעל המפתח הקטן יותר, והופכת את השורש השני ליד שלו. הפעולה כוללת עדכון מצביעי הורים, ילדים ואחים, והגדלת הדרגה של השורש בעל המפתח הגדול.

ניתוח סיבוכיות: הפונקציה מבצעת מספר קבוע של שינוי מצביעים ופעולות אריתמטיות. זמן הריצה הוא $O(1)$.

toBucket - פונקציית עזר המעבירת את כל השורשים לתוך מערך ה-"סלים" (*Buckets*) ובמבצעת מיזוגים תוך כדי. הפונקציה עוברת על רשימת השורשים. עבור כל שורש, היא בזקמת אם קיים כבר עץ באותו הדרגה ב"סל" המתאים. אם כן, היא מבצעת *link* ביןיהם, מפנה את הסל הנוכחי, ומנסה להכניס את העץ המאוחדר לסל בדרגה הבאה (ההילך חוזר עד שנמצא סל ריק).

ניתוח סיבוכיות: הפונקציה עוברת על כל השורשים ברשימה המקורית. במקרה הגרוע כאמור עבור עירומות עצילות, יתכן (n) שורשים ועל כן זמן הריצה יהיה (n) , אחרת זמן הריצה יהיה $O(log n)$.

fromBucket - פונקציית עזר הבונה מחדש את רשימת השורשים מתוך מערך הסלים. הפונקציה סורקת את מערך הסלים, וכל עץ שנמצא בו מחובר לשרשראת חדשה של שורשים. במקביל, היא מעדכנת את מונה העיצים (*numTrees*).

ניתוח סיבוכיות: הפונקציה עוברת על מערך הסלים שגודלו חסום לוגריתמית ומבצעת פעולות קבועות לכל תא. לכן, זמן הריצה הוא $(log n)$.

consolidate - פונקציה המאחדת את ההליך הסידור: קריאה ל-*toBucket* לפיזור ומיזוג העיצים, ולאחריה קריאה ל-*fromBucket* לאיסוף העיצים לרשימה חדשה.

ניתוח סיבוכיות: הפונקציה מבצעת את שני השלבים לעיל. השלב הדומיננטי הוא *toBucket* ועל כן זמן הריצה הכוללת בWC יהיה (n) עבור עירומות עצילות, אחרת $(log n)$.

- - הפונקציות הבאות מחזירות ערכים של שדות המחלקה. כל הערכים מותוחקים באופן שוטף במהלך פעולות העירימה, ולכן השילפה היא מיידית בזמן ריצה של $O(1)$.
 - size* –מחזירה את מספר האיברים בעירימה.
 - numTrees* –מחזירה את מספר העצים (*השורשים*) בעירימה.
 - numMarkedNodes* –מחזירה את מספר הצמתים מסומנים.
 - totalLinks* –מחזירה את סך פעולות הקישור שבוצעו.
 - totalCuts* –מחזירה את סך פעולות החיתוך שבוצעו.
 - totalHeapifyCosts* –מחזירה את כמות הפעולות שבוצעו תחילה *heapify*.

حلק ביסויי/תיאורטי

1. נגיד פונקציית פוטנציאלי. בנויגוד לעירימת פיבונacci' רגילה, שבה פונקציית הפוטנציאל היא $T = 2m + T$, כאן מספר העצים תמיד חסום ע"י $(logn)O$ בגלל שככל פעולה שגוראת ל *meld* מבצעת *successive linking*. כמו כן, מואתיה סיבה כל ניתוק עולה $(logn)O$. לכן נגידior את פונקציית הפוטנציאל להיות $\Phi = T + m * logn$.

actual cost – **insert** לאחר ומבצעת הכנסה $(O(1))$ ולאחריה *successive linking* הוא $O(logn)$. כמו כן, $(1)O(logn) = O(T_0 + m * logn) = O(T_1 + m * logn) - \Delta\Phi = T_1 + m * logn - (T_0 + m * logn)$. לכן סה"כ עלות בנייה לשיעורן היא $O(logn)$.

findMin – לאורך המימוש מתחזק מצביע לאיבר המינימלי, שכן $(1)O$.

- **deleteMin** – כפי שראינו בהרצאה, *actual cost* חסום ע"י $T_0 + logn$. קיבל כי מתקיים במקרה זה $amortize = T_1 + (m_1 - m_0)logn + logn$. לכן $\Delta\Phi = T_1 + m_1 * logn - (T_0 + m_0 * logn)$ ונוסף מתקיים $m_1 - m_0 \leq 0$ (לא יתכן שנוספו צמתים מסומנים, יתכן שרדו בכמהות) אז $amortize = O(logn)$.

decreaseKey – במקרה זה, *actual cost* הוא c הוא כמות החיתוכים שמבצעים בפעולה זאת מאוחר וכל חיתוך גורר הוספה לרשימה וביצוע *meld* נפרד (כאמור, *meld* היא פעולה בעלות של $O(logn)$). מבחינת ניתוח השינוי בפוטנציאל, ראשית נשים לב כי שראינו בהרצאה כי השינוי בכמהות הצמתים המסומנים הוא $c - 1 = 2 - (c - 1) = \Delta m = 1$. לכן קיבלונו כי השינוי בפוטנציאל הוא $\Delta T + \Delta m * logn = c + (2 - c)logn$.

מכאן מקבל כי סה"כ ניתוח של זמן הריצה לשיעורין הוא $O(logn)$, $amortize = clogn + c + 4logn - clogn = 4logn + c = O(logn)$.

delete – קוראת ל *delete min* ולאחר מכן *decrease key* כל c . הראמנו כי ניתוח לשיעורין של שתי הפעולות הוא $O(logn)$.

לכן סך הכל $amortize = O(logn)$.

.2

פעולה	ערימה ביןומית	ערימה ביןומית עצלה	ערימה ביןומית	עירימת פיבונacci'	עירימה ביןומית עם ניתוקים
Insert	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(logn)$
findMin	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
deleteMin	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$
decreaseKey	$O(logn)$	$O(1)$	$O(logn)$	$O(logn)$	$O(logn)$
delete	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$

.3

ניסוי 1	ניסוי 2	ניסוי 3	ערימה בינומית עם ניתוקים	ערימת פיבונacci	ערימה בינומית עצלה	ערימה בינומית
$O(nlogn)$	$O(n)$	$O(n)$	$O(nlogn)$	$O(n)$	$O(nlogn)$	$O(nlogn)$
$O(nlogn)$	$O(nlogn)$	$O(n)$	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$
$O(nlogn)$	$O(nlogn)$	$O(n)$	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$

.4

ניסוי 1:

זמן ריצה (מילישניות)	גודל הערימה בסיום	מספר העצים בסיום	מספר חיבורים	מספר חיתוכים	סה"כ עלויות של $heapify$ up	עלות מקסימלית לפעולה	ערימה בינומית	ערימת פיבונacci'	ערימה בינומית עצלה	ערימה בינומית עם ניתוקים
116	18	19	123							
464645	464645	464645	464645							
9	9	9								
464653	464653	464653	464653							
0	0	0								
0	0	0								
18	464636	464636	18							

ניסוי 2:

זמן ריצה (מילישניות)	גודל הערימה בסיום	מספר העצים בסיום	מספר חיבורים	מספר חיתוכים	סה"כ עלויות של $heapify$ up	עלות מקסימלית לפעולה	ערימה בינומית	ערימת פיבונacci'	ערימה בינומית עצלה	ערימה בינומית עם ניתוקים
505	209	888	1205							
46	46	46	46							
2	2	4								
1405996	748761	7552691	7552819							
748756	748717	0								
0	0	4148715	4149301							
39	464636	464636	35							

עירימה ביןומית עם ניתוקים	עירימת פיבונacci'	עירימה ביןומית עצלה	עירימה ביןומית	
121	24	23	115	זמן ריצה (밀ישניות)
464644	464644	464644	464644	גודל הערימה בסיום
11	11	8	8	מספר העצים בסיום
509262	509247	464653	464670	מספר חיבורים
44610	44614	0	0	מספר חיתוכים
0	0	114600	114559	סה"כ עלויות של <i>heapify up</i>
18	464636	464636	18	עלות מקסימלית לפעולה

.5

א. אkan ניתן לראות כי קיימים הבדלים בביטויים של הערימות בניסויים השונים.

בניסוי הראשון, ניתן לראות כי הערימות אשר $\text{meld} = \text{True}$, (*פיבונacci* ו*עצלה*) הפגינו בביטויים טובים יותר. זאת מכיוון ש *meld lazy* חוסכת זמן על ידי דחית איחוד העצים לפעולת החומרה *delete*. לעומת זאת, הערימות האחרות מבצעות את האיחודים בכל פעולה *Insert* ומכאן נובע הפער בזמן הריצה בין שני סוגי הערימות.

בניסוי השני, נשים לב כי הערימות אשר $\text{True} = \text{decrease keys}$, (*פיבונacci* ובינומית עם ניתוקים) הציגו בביטויים טובים יותר. זאת מכיוון שפעולות *key* מתבצעת ב(1) O לשיעורן לעומת הערימה הבינומית והערימה העצלה שבהן מבצעים את פעולה *heapify up* שלוקחת (logon) O לשיעורן. ניתן לראות בטבלה כי מספר החיתוכים שביצעו בערימות (בערך 750,000) קטן בהרבה ממספר פעולות *key* שביצעו (בערך 4,150,000) ובנוסף לכך שפעולה זאת יקרה בהרבה גורם להבדל הגודל בביטויים.

בנוסף גם כאן ניתן לראות הבדלים בין הערימות אשר ביצעו *meld lazy* הנובעים מהחלוקת הראשון של הניסוי שזיהה לניסוי 1.

בניסוי השלישי, באופן דומה לניסוי 1, הערימות עם הביצועים הטובים יותר היו *פיבונacci* ו*עצלה*. זאת מכיוון שמספר החיתוכים, *dp up* זניח ביחס לכמות ה`logon` כפי שניתן לראות בטבלה, لكن הערימות עם *lazy meld* הציגו בביטויים טובים יותר.

ב. תוצאות הניסוי בסעיף 4 מASHOTOT AT HTOTZAOOT BASUF 3.

ראשית, נשים לב שלפי ההערות, מימוש הקוד לערימה ביןומית מתנהג כמו עירימה ביןומית אבל הביצועים שלו אינם זהים לערימה ביןומית. لكن בניסויים שביצעו זמני הריצה של עירימה ביןומית גבוהים מהצפוי.

בניסוי הראשון אכן רأינו כי ערימות *פיבונacci* ו*עצלה* היו מהירות יותר מאשר כפי שציינו בסעיף 3. כמו כן רأינו כי עירימה ביןומית עם ניתוקים הגיעו לביטויים הנמוכים ביותר. לעומת זאת זמני הריצה הנמוכים של עירימה ביןומית בניסוי נובעים מהIMPLEMENTATION בקוד ולכן אינם תואמים את סעיף 3.

בניסוי השני אכן הגיענו לסדרי גודל דומים יותר בין הערימות ובניסוי השלישי כפי שצפינו ערים פיבונאצ'י הגיעה לביצועים הטוביים ביותר (עם הסטייגות על בינומית עצמה).

ג. ניתן לראות כי בכל הניסויים העולות המקסימלית לפועלה בפיבונאצ'י ועצלה היא (ח) 0 לעומת בינומית ובינומית עם ניטוקים שם העולות המקסימלית לפועלה קטנה ממשמעותית. זאת מכיוון שבערימות עם lazy True = meld פעולה המכילה הראשונה הינה הפעולה הכבודה ביוטר והוא משלמת על כל הפעולות הזרלות שביצעו עד כה. למרות זאת, زمنי הרצאה שלهن קצרים יותר משל שתי האחרות, שכן העולות לשיעורין שלhn נמוכה יותר כפי שנלמד בהרצאה.