מבני נתונים - פרויקט מספר 2 – ערמות מתקדמות

Amit Kacen | 212677884 | amitkacen

Dan Remeniuk | 322377490 | danremeniuk

חלק מעשי - תיעוד

בפרויקט זה כתבנו מחלקה בשם Heap המממשת סוגי ערימות שונות מעל מספרים שלמים חיוביים. המחלקה תומכת בגמישות מימושית ומאפשרת לבחור (בעת יצירת הערימה) האם להשתמש ב lazy melds והאם להשתמש ב lazy decrease keys. מטרת המחלקה היא לאחסן זוגות של Key-Value ולתמוך בפעולות ערימה יעילות.

תפקידו של כל חבר במחלקה

- self.lazyMelds – שדה בוליאני הקובע האם הערימה עובדת במצב של "מיזוג עצל". אם הערך true, פעולות מיזוג והכנסה הן מהירות והסידור נדחה. אם false, מתבצע סידור (successive linking) בכל פעולת הכנסה ומיזוג.
- self.lazyDecreaseKeys – שדה בוליאני הקובע את אופן הטיפול בהקטנת מפתח. אם הערך true, מתבצע שימוש במנגנון ה-Cascading Cuts של ערימת פיבונאצ'י. אם false, מתבצעת פעולת Heapify-Up סטנדרטית.
- self.min – שדה המחזיק מצביע לאובייקט בעל המפתח המינימלי הנוכחי בערימה.
- self.head – שדה המחזיק מצביע לאיבר הראשון ברשימת השורשים של הערימה.
- self.last – שדה המחזיק מצביע לאיבר האחרון ברשימת השורשים של הערימה.
- self.size – שדה המחזיק את מספר האיברים הכולל בערימה.
- self.numTrees – שדה המחזיק את מספר העצים הנוכחי ברשימת השורשים.
- self.numMarkedNodes – שדה המונה את כמות הצמתים המסומנים בערימה.
- self.totalLinks – מונה הסופר את סך פעולות הקישור (Link) שבוצעו בין שני עצים לאורך חיי הערימה.
- self.totalCuts – מונה הסופר את סך פעולות החיתוך (Cut) שבוצעו לאורך חיי הערימה.
- self.totalHeapifyCosts – מונה הסופר את סך פעולות ההחלפה (Swap) שבוצעו במסגרת תהליך Heapify-Up.

תיאור הפונקציות

- ***insert*** - הפונקציה מכניסה איבר חדש לערימה עם המפתח והמידע הנתונים. היא מחזירה את האובייקט HeapItem החדש שנוצר, המכיל מצביע לצומת בערימה. מבחינת אופן הפעולה, הפונקציה יוצרת תחילה צומת חדש ומאתחלת את מצביעי ה-next וה-prev שלו להצביע על עצמו. לאחר מכן, היא יוצרת ערימה זמנית חדשה (heap2) המכילה רק את האיבר החדש, ומבצעת קריאה לפונקציה meld כדי למזג את הערימה הזמנית לתוך הערימה הנוכחית.
  <u>ניתוח סיבוכיות:</u> מן הריצה תלוי בערכו של השדה lazyMelds:
  - אם lazyMelds הוא true, פעולת ה-meld מבצעת שרשור של רשימות השורשים בלבד, ללא succesiveLinking. פעולה זו כוללת מספר קבוע של עדכוני מצביעים. לכן, זמן הריצה הוא $O(1)$.
  - אם lazyMelds הוא false, פעולת ה-meld קוראת ל-succesiveLinking. במקרה הגרוע ביותר (בו חיבור העץ החדש גורר רצף של מיזוגים), ייתכנו מיזוגים עד לגובה העץ המקסימלי. מאחר וגובה העץ חסום לוגריתמית, זמן הריצה במקרה הגרוע הוא $O(logn)$.

- ***findMin*** - הפונקציה מחזירה את האובייקט בעל המפתח המינימלי בערימה. הפונקציה ניגשת ישירות לשדה self.min, אשר מחזיק מצביע לאיבר המינימלי הנוכחי. אם הערימה ריקה, מוחזר ערך null.
<u>ניתוח סיבוכיות:</u> המצביע למינימום מתוחזק באופן שוטף ולכן הגישה אליו קוראת בזמן קבוע של $O(1)$.
- ***deleteMin*** - הפונקציה מוחקת את האיבר המינימלי מהערימה. תחילה הפונקציה מטפלת במקרי קצה (ערימה ריקה או ערימה עם איבר יחיד). לאחר מכן, היא מוציאה את איבר המינימום מרשימת השורשים ומעדכנת את המצביעים ואת גודל הערימה בהתאם. אם לאיבר שנמחק היו ילדים, הפונקציה הופכת אותם לשורשים עצמאיים. לבסוף, הפונקציה קוראת ל-*succesiveLinking* כדי לבצע פעולת *consolidate* ולסדר את הערימה מחדש, ואז סורקת את רשימת השורשים החדשה כדי למצוא ולעדכן את מצביע ה-*min* החדש.
<u>ניתוח סיבוכיות</u>: זמן הריצה תלוי בכמות העצים ברשימת השורשים ובדרגת האיבר המינימלי שנמחק.
  - במקרה של *lazyMelds=true*, ייתכן שרשימת השורשים מכילה $O(n)$ עצים (עקב רצף הכנסות ללא סידור). פעולת ה-*succesiveLinking* ועדכון ה-*min* יאלצו לסרוק את כל העצים הללו. לכן, זמן הריצה במקרה הגרוע הוא $O(n)$. עם זאת, נציין כי עלות ה *amortize* במצב זה היא $O(logn)$.
  - במקרה של *lazyMelds=false*, רשימת השורשים תמיד מכילה לכל היותר $O(logn)$ עצים. לכן, תהליך המחיקה, הקישור מחדש ועדכון המינימום חסומים כולם על ידי גובה העץ. זמן הריצה הוא $O(logn)$ ב *WC*.
- ***decreaseKey*** - הפונקציה מקטינה את המפתח של איבר נתון (*x*) בערך מסוים (*diff*) ושומרת על חוקיות הערימה. תחילה הפונקציה מעדכנת את המפתח של האיבר ובודקת אם הוא קטן מהמינימום הנוכחי (ומעדכנת את *min* בהתאם). לאחר מכן, אופן הפעולה תלוי בדגל *lazyDecreaseKeys*:
  - אם הוא *true*, נבדק האם הופר כלל הערימה. במידה וכן, נקראת הפונקציה *cascadingCut* כדי לנתק את הצומת ולטפל בשרשרת הסימונים במעלה העץ.
  - אם הוא *false*, נקראת הפונקציה *heapifyUp* כדי לבצע פעפוע של האיבר כלפי מעלה.

<u>ניתוח סיבוכיות:</u> נראה כי זמן הריצה ב *WC* הוא $O(logn)$

  - במקרה הראשון (*if*): הפונקציה מבצעת *cascadingCut*. במקרה הגרוע, בהתאם לערך של *lazyMelds*, עלות הריצה היא $O(\log^2 n)$ או $O(logn)$.
  - במקרה השני (*else*): הפונקציה מבצעת *heapifyUp* שעלותו חסומה בגובה העץ ולכן זמן הריצה יהיה $O(logn)$.
- ***heapifyUp*** - פונקציית עזר המעלה צומת במעלה העץ עד שחוקיות הערימה משוחזרת. הפונקציה רצה בלולאה כל עוד הצומת אינו שורש ומפתחו קטן ממפתח אביו. בכל איטרציה מתבצעת החלפה (באמצעות קריאה למתודה *swapWithParent*) והצומת מתקדם לאביו.
<u>ניתוח סיבוכיות</u>: במקרה הגרוע, הצומת יעלה מהעלה העמוק ביותר ועד לשורש. על כן, זמן הריצה ב *WC* הוא $O(logn)$.
- ***swapWithParent*** - פונקציית עזר המבצעת החלפה בין צומת לאביו. ההחלפה מתבצעת על ידי החלפת התוכן של ה-*HeapItem* בין שני הצמתים, ועדכון המצביעים המתאימים בתוך ה-*Item*. כמו כן, הפונקציה מעדכנת את המצביעים הגלובליים (*head, last, min*) במידת הצורך.
<u>ניתוח סיבוכיות:</u> הפונקציה מבצעת פעולות בזמן קבוע בלבד, לכן זמן הריצה הוא $O(1)$.
- ***cascadingCut*** - פונקציה רקורסיבית המבצעת את מנגנון הניתוק המדורג. תחילה הפונקציה קוראת ל-*cut* כדי לנתק את הבן (*x*) מאביו (*y*). לאחר מכן היא בודקת את האב: אם האב אינו שורש והוא אינו מסומן (*marked*), נסמן אותו (*marked=true*) והתהליך נעצר. אם האב כבר מסומן, הסימון מוסר (*marked=false*), והפונקציה קוראת לעצמה רקורסיבית כדי לנתק גם את האב מאביו שלו.
<u>ניתוח סיבוכיות:</u> במקרה הגרוע ביותר, שרשרת הניתוקים נמשכת עד לשורש העץ. מכיוון שגובה העץ חסום לוגריתמית, מספר הקריאות הרקורסיביות חסום על ידי גובה העץ. כאשר *lazyMelds=True*, זמן הריצה הוא $O(logn)$, אחרת כל פעולת *cut* היא בזמן ריצה של $O(logn)$ ולכן סה"כ $O(\log^2 n)$.

- ***cut*** - פונקציית עזר המנתקת קשר בין צומת (x) לאביו (y) והופכת את x לשורש חדש בערימה. הפונקציה מעדכנת את מצביעי הילדים של האב, מקטינה את דרגתו, ומאפסת את סימון ה-*marked* של הבן. לאחר מכן, הבן מתווסף לרשימת השורשים של הערימה באמצעות יצירת ערימה זמנית וקריאה ל-*meld*.
<u>ניתוח סיבוכיות:</u> הפונקציה מבצעת מספר קבוע של עדכוני מצביעים וקוראת ל-*meld*. בהנחה שהערימה מוגדרת כעצלה (*lazyMelds=true*), פעולת המיזוג היא $O(1)$ וכך גם זמן הריצה של המתודה. במידה ו *lazyMelds=false*, הסיבוכיות תהיה $O(logn)$.
- ***delete*** - הפונקציה מוחקת איבר כללי (x) מהערימה. המימוש 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.
<u>ניתוח סיבוכיות:</u> הפונקציה מורכבת משני שלבים: *decreaseKey* שעלותו $O(logn)$ במקרה הגרוע, ו-*deleteMin*. בהנחה שהערימה מוגדרת כלא-עצלה (*lazyMelds=false*), גם *deleteMin* חסומה ב-$O(logn)$ אם *lazy*DecreaseKeys=false אחרת חסומה ב $O(\log^2 n)$, אם *lazyMelds=True* חסומה ב $O(n)$. בפרט, ב*WC* זמן הריצה הוא $O(n)$.
- ***meld*** - הפונקציה ממזגת את הערימה הנוכחית עם ערימה נוספת (*heap2*). תחילה הפונקציה משרשרת את רשימת השורשים של הערימה השנייה לקצה רשימת השורשים של הערימה הנוכחית. לאחר מכן היא מעדכנת את השדות הסטטיסטיים (*size, numTrees, numMarkedNodes* וכו') ואת המינימום הגלובלי. לבסוף, נבדק את הערך של *lazyMelds*: אם הוא *false*, מתבצעת קריאה ל-*succesiveLinking* כדי לסדר את הערימה המאוחדת.
<u>ניתוח סיבוכיות:</u> מתחלק לפי ערכו של השדה *lazyMelds*:
  - אם *lazyMelds=true*: מתבצע שרשור מצביעים בלבד ועדכון שדות, ולכן הזמן הוא $O(1)$.
  - אם *lazyMelds=false*: מתבצעת קריאה ל *succesiveLinking* הסורקת את כל השורשים. במקרה הגרוע, מספר השורשים פרופורציונלי ל n. לכן, הזמן הריצה הוא $O(n)$.
- ***succesiveLinking*** - פונקציית עזר המבצעת את תהליך ה *Consolidation*. הפונקציה אחראית על ארגון מחדש של רשימת השורשים כך שלא יישארו שני עצים בעלי אותה דרגה. היא יוצרת מערך עזר בגודל לוגריתמי, מאתחלת אותו ב-*null*, וקוראת לפונקציה *consolidate*. לאחר מכן היא מעדכנת את מצביעי ה-*head* וה-*last* בהתאם לרשימה החדשה שהתקבלה.
<u>ניתוח סיבוכיות</u>: הפונקציה תלויה ישירות בפונקציה *consolidate* הסורקת את כל שורשי הערימה. בפרט, במקרה הגרוע כמות השורשים לינארית בכמות האיברים ולכן זמן הריצה הוא $O(n)$.
- ***link*** - פונקציית עזר המחברת שני עצים בעלי דרגה זהה. הפונקציה קובעת מי מהשורשים הוא בעל המפתח הקטן יותר, והופכת את השורש השני לילד שלו. הפעולה כוללת עדכון מצביעי הורים, ילדים ואחים, והגדלת הדרגה של השורש בעל המפתח הגדול.
<u>ניתוח סיבוכיות:</u> הפונקציה מבצעת מספר קבוע של שינויי מצביעים ופעולות אריתמטיות. זמן הריצה הוא $O(1)$.
- ***toBucket*** - פונקציית עזר המעבירה את כל השורשים לתוך מערך ה"סלים" (*Buckets*) ומבצעת מיזוגים תוך כדי. הפונקציה עוברת על רשימת השורשים. עבור כל שורש, היא בודקת אם קיים כבר עץ באותו הדרגה ב"סל" המתאים. אם כן, היא מבצעת *link* ביניהם, מפנה את הסל הנוכחי, ומנסה להכניס את העץ המאוחד לסל בדרגה הבאה (תהליך החוזר עד שנמצא סל ריק).
<u>ניתוח סיבוכיות:</u> הפונקציה עוברת על כל השורשים ברשימה המקורית. במקרה הגרוע כאמור עבור ערימות עצלות, ייתכנו $O(n)$ שורשים ועל כן זמן הריצה יהיה $O(n)$, אחרת זמן הריצה יהיה $O(logn)$.
- ***fromBucket*** - פונקציית עזר הבונה מחדש את רשימת השורשים מתוך מערך הסלים. הפונקציה סורקת את מערך הסלים, וכל עץ שנמצא בו מחובר לשרשרת חדשה של שורשים. במקביל, היא מעדכנת את מונה העצים (*numTrees*).
<u>ניתוח סיבוכיות:</u> הפונקציה עוברת על מערך הסלים שגודלו חסום לוגריתמית ומבצעת פעולות קבועות לכל תא. לכן, זמן הריצה הוא $O(logn)$.
- ***consolidate*** - פונקציה המאחדת את תהליך הסידור: קריאה ל-*toBucket* לפיזור ומיזוג העצים, ולאחריה קריאה ל-*fromBucket* לאיסוף העצים לרשימה חדשה.

<u>ניתוח סיבוכיות</u>: הפונקציה מבצעת את שני השלבים לעיל. השלב הדומיננטי הוא *toBucket* ועל כן זמן הריצה הכולל ב *WC* יהיה $O(n)$ עבור ערימות עצלות, אחרת $O(logn)$.

- ***getters*** **-** הפונקציות הבאות מחזירות ערכים של שדות המחלקה. כל הערכים מתוחזקים באופן שוטף במהלך פעולות הערימה, ולכן השליפה היא מיידית בזמן ריצה של $O(1)$.
  - *size* – מחזירה את מספר האיברים בערימה.
  - *numTrees* – מחזירה את מספר העצים (השורשים) בערימה.
  - *numMarkedNodes* – מחזירה את מספר הצמתים המסומנים.
  - *totalLinks* – מחזירה את סך פעולות הקישור שבוצעו.
  - *totalCuts* – מחזירה את סך פעולות החיתוך שבוצעו.
  - *totalHeapifyCosts* – מחזירה את כמות הפעולות שבוצע תהליך ה-*heapify*.

חלק ניסויי/תיאורטי

1. נגדיר פונקצית פוטנציאל. בניגוד לערימת פיבונאצ'י רגילה, שבה פונקציית הפוטנציאל היא $T + 2m$, כאן מספר העצים תמיד חסום ע"י $O(logn)$ בגלל שכל פעולה שקוראת ל *meld* מבצעת *successive linking*. כמו כן, מאותה סיבה כל ניתוק עולה $O(logn)$. לכן נגדיר את פונקציית הפוטנציאל להיות $\Phi = T + m * logn$.

$\boldsymbol{insert}$ – *actual cost* הוא *O(logn)* מאחר ומתבצעת הכנסה *(O(1))* ולאחריה *successive linking* ($O(logn)$). כמו כן, $\Delta\Phi = T_1 + m * logn - (T_0 + m * logn) = O(1)$. לכן סה"כ עלות בניתוח לשיעורין היא $O(logn)$.

$\boldsymbol{findMin}$ – לאורך המימוש מתוחזק מצביע לאיבר המינימלי, לכן $O(1)$.

$\boldsymbol{deleteMin}$ - כפי שראינו בהרצאה, *actual cost* חסום ע"י $T_0 + logn$. נקבל כי מתקיים במקרה זה $\Delta\Phi = T_1 + m_1 * logn - (T_0 + m_0 * logn)$. לכן $amortize = T_1 + (m_1 - m_0)logn + logn$. מכיוון ש T_1 חסום ע"י $logn$, ובנוסף מתקיים $m_1 - m_0 \leq 0$ (לא ייתכן שנוספו צמתים מסומנים, יתכן שירדו בכמות) אז נסיק כי $amortize = O(logn)$.

$\boldsymbol{decreaseKey}$ – במקרה זה, *actual cost* הוא $O(clogn)$ כאשר *c* הוא כמות החיתוכים שמבצעים בפעולה, זאת מאחר וכל חיתוך גורר הוספה לרשימה וביצוע *meld* נפרד (כאמור, *meld* היא פעולה בעלות של $O(logn)$). מבחינת ניתוח השינוי בפוטנציאל, ראשית נשים לב כפי שראינו בהצראה כי השינוי בכמות הצמתים המסומנים הוא $\Delta m = 1 - (c - 1) = 2 - c$. לכן קיבלנו כי השינוי בפוטנציאל הוא $\Delta\Phi = \Delta T + \Delta m * logn = c + (2 - c)logn$. מכאן נקבל כי סה"כ ניתוח של זמן הריצה לשיעורין הוא $amortize = clogn + c + 4logn - clogn = 4logn + c = O(logn)$, מאחר ו ΔT חסום ע"י $O(logn)$.

delete - קוראת ל *decrease key* ולאחר מכן ל *delete min* . הראנו כי ניתוח לשיעורין של שתי הפעולות הוא $amortize = O(logn)$.

לכן סך הכל $amortize = O(logn)$.

2.

פעולה	ערימה בינומית	ערימה בינומית עצלה	ערימת פיבונאצי	ערימה בינומית עם ניתוקים
Insert	*O(1)*	*O(1)*	*O(1)*	*O(logn)*
findMin	*O(1)*	*O(1)*	*O(1)*	*O(1)*
deleteMin	*O(logn)*	*O(logn)*	*O(logn)*	*O(logn)*

decreaseKey	*O(logn)*	*O(logn)*	*O(1)*	*O(logn)*
delete	*O(logn)*	*O(logn)*	*O(logn)*	*O(logn)*

.3

	ערימה בינומית	**ערימה בינומית עצלה**	**ערימת פיבונאצי**	**ערימה בינומית עם ניתוקים**
ניסוי 1	*O(n)*	*O(n)*	*O(n)*	*O(nlogn)*
ניסוי 2	*O(nlogn)*	*O(nlogn)*	*O(nlogn)*	*O(nlogn)*
ניסוי 3	*O(nlogn)*	*O(nlogn)*	*O(n)*	*O(nlogn)*

.4

ניסוי 1:

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

	ערימה בינומית	ערימה בינומית עצלה	ערימת פיבונאצ'י	ערימה בינומית עם ניתוקים
זמן ריצה (מילישניות)	1205	888	209	505
גודל הערימה בסיום	46	46	46	46
מספר העצים בסיום	4	4	2	2
מספר חיבורים	7552819	7552691	748761	1405996
מספר חיתוכים	0	0	748717	748756

סה"כ עלויות של *heapify up*	4149301	4148715	0	0
עלות מקסימלית לפעולה	35	464636	464636	39

ניסוי 2:

ניסוי 3:

	ערימה בינומית	ערימה בינומית עצלה	ערימת פיבונאצ'י	ערימה בינומית עם ניתוקים
זמן ריצה (מילישניות)	115	23	24	121
גודל הערימה בסיום	464644	464644	464644	464644
מספר העצים בסיום	8	8	11	11
מספר חיבורים	464670	464653	509247	509262
מספר חיתוכים	0	0	44614	44610
סה"כ עלויות של *heapify up*	114559	114600	0	0
עלות מקסימלית לפעולה	18	464636	464636	18

.5

א. אכן ניתן לראות כי קיימים הבדלים בביצועים של הערימות בניסויים השונים.

בניסוי הראשון, ניתן לראות כי הערימות אשר lazy meld = True (פיבונאצי ועצלה) הפגינו ביצועים טובים יותר. זאת מכיוון ש lazy meld חוסכת זמן על ידי דחיית איחוד העצים לפעולת הdelete min. לעומת זאת, הערמות האחרות מבצעות את האיחודים בכל פעולת Insert ומכאן נובע הפער בזמני הריצה בין שני סוגי הערימות.

בניסוי השני, נשים לב כי הערימות אשר lazy decrease keys = True (פיבונאצי ובינומית עם ניתוקים) הציגו ביצועים טובים יותר. זאת מכיוון שפעולת הdecrease key מתבצעת ב O(1) לשיעורין לעומת הערימה הבינומית והערימה העצלה שבהן מבצעים את פעולת הheapifyUp שלוקחת O(logn) לשיעורין. ניתן לראות בטבלה כי מספר החיתוכים שביצענו בערימות (בערך 750,000) קטן בהרבה ממספר פעולות ה heapifyUp שביצענו (בערך 4,150,000) ובנוסף לכך שפעולה זאת יקרה בהרבה גרם להבדל הגדול בביצועים.

בנוסף גם כאן ניתן לראות הבדלים בין הערימות אשר ביצעו lazy meld הנובעים מהחלק הראשון של הניסוי שזהה לניסוי 1.

בניסוי השלישי, באופן דומה לניסוי 1, הערימות עם הביצועים הטובים יותר היו פיבונאצי ועצלה. זאת מכיוון שמספר החיתוכים, *heapify up* זניח ביחס לכמות הLinking כפי שניתן לראות בטבלה, לכן הערמות עם lazy meld הציגו ביצועים טובים יותר.

ב. תוצאות הנסוי בסעיף 4 מאששות את התוצאות בסעיף 3.

ראשית, נשים לב שלפי ההערות, מימוש הקוד לערימה בינומית מתנהג כמו ערימה בינומית אבל הביצועים שלו אינם זהים לערימה בינומית. לכן בניסויים שביצענו זמני הריצה של ערימה בינומית גבוהים מהמצופה.

בניסוי הראשון אכן ראינו כי ערימות פיבונאצי ועצלה היו מהירות יותר כפי שציינו בסעיף 3. כמו כן ראינו כי ערימה בינומית עם ניתוקים הגיעה לביצועים הנמוכים ביותר. לעומת זאת זמני הריצה הנמוכים של ערימה בינומית בניסוי נובעים מהמימוש בקוד ולכן אינם תואמים את סעיף 3.

בניסוי השני אכן הגענו לסדרי גודל דומים יותר בין הערימות ובניסוי השלישי כפי שצפינו ערימת פיבונאצי הגיעה לביצועים הטובים ביותר (עם הסתייגות על בינומית עצלה).

ג. ניתן לראות כי בכל הניסויים העלות המקסימלית לפעולה בפיבונאצי ועצלה היא O(n) לעומת בינומית ובינומית עם ניתוקים שם העלות המקסימלית לפעולה קטנה משמעותית. זאת מכיוון שבערימות עם lazy meld = True פעולה המחיקה הראשונה הינה הפעולה הכבדה ביותר והיא משלמת על כל הפעולות הזולות שביצענו עד כה. למרות זאת, זמני הריצה שלהן קצרים יותר משל שתי האחרות, שכן העלות לשיעורין שלהן נמוכה יותר כפי שנלמד בהרצאה.