פרויקט מספר 1- AVL ורשימות

עמית אליאסי 316291434

אורן לוי 208410183

סמסטר אביב 2020

תיעוד הפרויקט

הגדרות:

n= מספר האיברים במבנה

h= גובה העץ

מתכונות עץ AVL -h=O(logn) $= \log_2 n < h < \log_\Phi n = O(logn) \Rightarrow h = O(logn)$

עץ AVL תקין= עץ בינרי בו גורם האיזון של כל צומת לא גדול מ1 ולא קטן מ-1.

AVLTree

שדות

root – עצם מסוג IAVLNode המצביע לשורש העץ. מחזיק null כאשר העץ ריק.

size – עצם מסוג int. מחזיק את ערך מספר הצמתים בעץ. מאותחל ל0.

מתודות

Empty

מבצעת פעולה אחת בלבד כיוון שבמחלקה מתוחזק גודל העץ בשדה size , כלומר בודקת אם גודל העץ הוא 0. לכן, פועלת בסיבוביות $O(1)$.

Search

משתמשת בפונקציית העזר searchNode(k) הפועלת בסיבוביות $O(\log n)$ ומחזירה את הצומת שהמפתח שלה הוא k, כל שאר הפעולות פועלות בזמן קבוע, כלומר סך הכל המתודה פועלת בסיבוביות $O(\log n)$. מחזירה את ערך של הצומת המוחזרת מפונקציית העזר.

Insert

הפונקציה מחפשת את המיקום הנכון (שישאיר את העץ, עץ חיפוש בינארי חוקי) לצומת החדשה באמצעות חיפוש בינארי, בזמן $O(\log n)$, ומכניסה את הצומת במקום זה באמצעות פעולות בזמן קבוע. לאחר מכן הפונקציה מתקנת את העץ חזרה להיות AVL תקין באמצעות מעבר על כל הדרך מהצומת החדש ועד לשורש, ובדרך מעדכנת את הגובה והגודל של כל צומת. כלומר התיקון והעדכון עובר לא יותר מ hn צמתים בדרך, כלומר $O(\log n)$ צמתים בסיבוכיות זמן של $O(\log n)$. סך הכל, הפונקציה פועלת בסיבוכיות זמן של $O(2\log n) = O(\log n)$.

Delete

כפי שנלמד בכיתה, בכדי למחוק צומת, במידה ואין לצומת ילדים, ניתן פשוט למחוק את הצומת, במידה ויש ילד אחד, ניתן לבצע מעקף, כלומר שהילד של הצומת יהיה עתה הילד של ההורה של הצומת. במידה ולצומת יש שני ילדים יש להחליף את מיקומו עם מיקום הקודם שלו, (לו בטוח אין בן ימני כי הוא המקסימלי מבין הצמתים הקטנים ממנו, כלומר אין גדול ממנו בתת העץ הנוכחי) ומשם למחוק אותו.

לאחר פעולות אלה נחזור ונעדכן את העץ חזרה להיות AVL תקין בעזרת גלגולים, ונעדכן את השדות הרלוונטים.

חיפוש הצומת למחיקה באמצעות פונקציית עזר searchNode הפועלת בסיבוכיות $O(logn)$.

מציאת הקודם של הצומת שצריך למחוק באמצעות פונקציית העזר predecessor הפועלת בסיבוכיות $O(logn)$.

מחיקת הצומת + פעולות קבועות (כמו *switchPosithion* אם צריך) בזמן קבוע.

תיקון ועדכון גובה וגודל של הצמתים שנפגעו, כמו בפונקציית הכנסה (מוסבר שם לעומק) יתבצע בסיבוכיות $O(logn)$

סך הכל הפונקציה פועלת בסיבוכיות זמן של $O(logn)$.

Min

המפתח המינימלי בעץ תמיד ימוקם בצומת השמאלית ביותר, כלומר בכדי להגיע אליו נרד מהשורש לבן השמאלי וממנו לבן השמאלי שלו וכן הלאה עד שלא יהיה בן שמאלי לרדת אליו.

המתודה מבצעת שימוש בפונקציית העזר minNode (ניתוח הסיבוכיות שלה מתואר בהמשך) הפועלת בסיבוכיות זמן של $O(logn)$, כלומר סך הכל הפונקציה פועלת בסיבוכיות זמן של $O(logn)$.

Max

המפתח המקסימלי בעץ תמיד ימוקם בצומת הימנית ביותר, כלומר בכדי להגיע אליו נרד מהשורש לבן הימני וממנו לבן הימני שלו וכן הלאה עד שלא יהיה בן ימני לרדת אליו.

מבצעת שימוש בפונקציית העזר maxNode (ניתוח הסיבוכיות שלה מתואר בהמשך) הפועלת בסיבוכיות זמן של $O(logn)$, כלומר סך הכל הפונקציה פועלת בסיבוכיות זמן של $O(logn)$.

keysToArray

בכדי להכניס את כל המפתחות של העץ בזמן מינימלי בחרנו להשתמש בפונקציית עזר רקורסיבית (מתוארת בהמשך). הרעיון הוא לעבור על הצמתים לפי סדר in-order ולהכניס את המפתח שלהם בסדר זה למערך.

לאחר יצירת מערך חדש באורך n המתודה מבצעת שימוש בפונקציית העזר keysToArrayRec (ניתוח הסיבוכיות שלה מתואר בהמשך) הפועלת בסיבוכיות זמן של $O(n)$, כלומר סך הכל הפונקציה פועלת בסיבוכיות זמן של $O(n)$. המתודה תחזיר את המערך המוחזרת מפונקציית העזר.

infoToArray

בכדי להכניס את כל הערכים של העץ בזמן מינימלי בחרנו להשתמש בפונקציית עזר רקורסיבית (מתוארת בהמשך). הרעיון הוא לעבור על הצמתים לפי סדר in-order ולהכניס את ערכם בסדר זה למערך.

לאחר יצירת מערך חדש באורך n מבצעת שימוש בפונקציית העזר infoToArrayRec (ניתוח הסיבוכיות שלה מתואר בהמשך) הפועלת בסיבוכיות זמן של $O(n)$, כלומר סך הכל הפונקציה פועלת בסיבוכיות זמן של $O(n)$.

size

מחזירה את הערך הנמצא בשדה size בזמן קבוע, כלומר פועלת בסיבוכיות זמן של $O(1)$.

getRoot

מחזירה את הצומת הנמצאת בשדה root בזמן קבוע, כלומר פועלת בסיבוכיות זמן של $O(1)$.

פונקציות עזר

keysToArrayRec

הפעולה מקבלת בכל פעם מערך בגודל n שבו אנו שומרים את המפתחות מהקטן לגדול, אינדקס שמסמל את המיקום הבא שיש למלא במערך, מצביע לnode הנוכחי ומצביע לnode הקודם. כל ריצה של הרקורסיה מבצעת פעולות קבועות בזמן קבוע ופועלת בסיבוכיות זמן של $O(1)$, משום שמתודה רק בודקת אם יש בן ימני/ שמאלי או הורה, מי הprev שלנו, ובמידת הצורך מוסיפה את מפתח הצומת הנוכחי למערך. הפעולה תבוצע מקסימום 3 פעמים על כל node, לכן סה"כ תבוצע לא יותר מ 3n פעמים כאשר n מספר הצמתים בעץ, ולכן סה"כ פועלת בסיבוכיות זמן של $O(n)$. (תבוצע מקס' 3 פעמים, במקרה הגרוע נעבור על צומת נתון פעם ראשונה כאשר הprev הוא

ההורה שלו, פעם שנייה כאשר prev הוא הבן השמאלי שלו, ופעם שלישית כאשר prev הוא הבן הימני שלו).

infoToArrayRec

הפעולה מקבלת בכל קריאה מערך בגודל n שבו אנו שומרים את הערכים (info) מסודרים מהערך של הצומת שמפתחה הקטן ביותר לגדול, אינדקס שמסמל את המיקום הבא שיש למלא במערך, מצביע לnode הנוכחי ומצביע לnode הקודם. כל ריצה של הרקורסיה מבצעת פעולות קבועות בזמן קבוע ופועלת בסיבוכיות זמן של $O(1)$, משום שרק בודקת אם יש בן ימני/ שמאלי או הורה, מי הprev שלנו, ובמידת הצורך מוסיפה את ערך הצומת הנוכחי למערך. הפעולה תבוצע מקסימום 3 פעמים על כל node, לכן סה"כ תבוצע לא יותר מ 3n פעמים כאשר n מספר הצמתים בעץ, ולכן סה"כ פועלת בסיבוכיות זמן של $O(n)$. (תבוצע מקס' 3 פעמים, במקרה הגרוע נעבור על צומת נתון פעם ראשונה כאשר הprev הוא ההורה שלו, פעם שנייה כאשר prev הוא הבן השמאלי שלו, ופעם שלישית כאשר prev הוא הבן הימני שלו).

searchNode

מחזירה את הצומת עם המפתח k באמצעות חיפוש בינארי. במידה והמפתח לא קיים בעץ תחזיר null. במקרה הגרוע, הצומת הדרושה תהיה רחוקה מהשורש מרחק של כ-h קשתות, כלומר במקרה הגרוע הפונקציה תעבור על פני $O(logn)$ צמתים ותבצע $O(logn)$ פעולות.

סך הכל הפונקציה פועלת במקרה הגרוע בסיבוכיות זמן של $O(logn)$.

minNode

המתודה מחזירה את הצומת בעלת המפתח המינימלי בעץ. עבור עץ ריק המתודה תחזיר null.

המפתח הקטן ביותר נמצא תמיד בצומת שמאלי ביותר של העץ, כלומר בכדי להגיע אליו יש לרדת שמאלה מהשורש עד שמגיעים שאין בן שמאלי לרדת אליו. נשים לב שירידה זו חסומה בגובה העץ, ששווה ל $O(logn)$, כלומר סך הכל הפונקציה פועלת בסיבוכיות זמן של $O(logn)$.

maxNode

המתודה מחזירה את הצומת בעלת המפתח המקסימלי בעץ. עבור עץ ריק המתודה תחזיר null.

המפתח הגדול ביותר נמצא תמיד בעלה הכי ימני של העץ, כלומר בכדי להגיע אליו יש לרדת ימינה מהשורש עד שאין בן ימני לרדת אליו. נשים לב שירידה זו חסומה בגובה העץ, ששווה ל $O(logn)$, כלומר סך הכל הפונקציה פועלת בסיבוכיות זמן של $O(logn)$.

Predecessor

המתודה מקבלת צומת, ומחזירה את הצומת הקודמת לה בעץ, לפי סדר in-order.

במקרה הגרוע בבדי למצוא את הקודם של צומת, נרד לבן השמאלי שלה, וממנו נרד ימינה עד שנגיע לעלה, שיהיה הקודם של הצומת. הירידה חסומה בגובה העץ, כלומר סך הכל הפונקציה פועלת במקרה הגרוע בסיבוכיות זמן של $O(logn)$.

Successor

המתודה מקבלת צומת, ומחזירה את הצומת הבאה בעץ, לפי סדר in-order.

במקרה הגרוע בבדי למצוא את הבא של צומת, נרד לבן הימני שלה, וממנו נרד שמאלה עד שנגיע לעלה, שיהיה הבא של הצומת. הירידה חסומה בגובה העץ, כלומר סך הכל הפונקציה פועלת במקרה הגרוע בסיבוכיות זמן של $O(logn)$.

switchPositions

הפונקציה מחליפה את המיקום של 2 צמתים-צומת 1 וצומת 2 בעזרת צומת זמנית: נעביר את המצביעים של הילדים ואת כל ערכי צומת 1 לצומת הזמנית, (בבדי שכשאר נעביר את הערכים מצומת 2 לצומת 1, לא נאבד את הערכים שהיו בצומת 1) לאחר מכן נעביר את הערכים והמצביעים של צומת 2 לצומת 1 ואז מהזמנית לצומת 2. פעולות אלה פועלות בזמן קבוע כל אחת, סך הכל הפונקציה פועלת בסיבוכיות זמן של $O(1)$.

Rebalance

המתודה מאזנת מחדש את הצומת אותו מצאנו כ"עבריין AVL" כאשר גורם איזון של צומת מופר, ומחזירה את הצומת להיות צומת AVL תקין.

הפונקציה מבצעת בין רוטציה אחת לשתיים כפי שנלמד בביתה, כאשר כל רוטציה פועלת בזמן קבוע.

בנוסף הפונקציה מתקנת את העץ על ידי שימוש בפונקציית העזר updatHeightAndSize הפועלת בסיבוכיות זמן של $O(logn)$.

סך הכל הפונקציה פועלת במקרה הגרוע בסיבוכיות זמן של $O(logn)$.

leftRotation

המתודה מקבלת צומת ומבצעת גלגול שמאלה עבור הצומת הזו.

מבצעת פעולות קבועות (אין לולאה בפונקציה) כלומר פועלת בסיבוכיות זמן של $O(1)$.

rightRotation

המתודה מקבלת צומת ומבצעת גלגול ימינה עבור הצומת הזו.

מבצעת פעולות קבועות (אין לולאה בפונקציה) כלומר פועלת בסיבוכיות זמן של $O(1)$.

updateHeightAndSize

המתודה מקבלת צומת, ומעדכנת את הגובה והגודל של כל הצמתים בדרך מן הצומת עד השורש של העץ.

במקרה הגרוע הפונקציה תעבור על המסלול הארוך ביותר בין עלה לשורש, כלומר תעבור על h או h+1 צמתים, כלומר פועלת בבסיבוכיות זמן של $O(logn)$.

nodeInIndex

מתודת עזר בשביל המחלקה TreeList – מקבלת מספר שלם i ומחזירה את הצומת בעץ שדרגתה i+1.

הפונקציה מחפשת צומת מסויימת לפי הדרגה שלה, החל מהשורש יורדת עד למציאת הצומת, באמצעות חיפוש בינארי.

עבור כל צומת $node$ בה נעבור נחשב את הדרגה שלה על ידי הדרגה הקודמת (עבור כל חישוב נבצע פעולות בזמן קבוע):

- הדרגה של השורש היא $node.getLeft().size() + 1$ (כלומר, גודל העץ השמאלי שלו פלוס עצמו)
- אם ירדנו שמאלה בחיפוש, הדרגה של הצומת שירדנו אליה תהיה הדרגה הקודמת, פחות גודל תת העץ הימני, פחות 1.
- אם ירדנו ימינה, הדרגה תהיה הדרגה הקודמת, פלוס גודל תת העץ השמאלי פלוס 1
- (נשים לב שבכל ירידה עלינו לשמור את הדרגה הקודמת בלבד, אין צורך לשמור את כל השרשרת)

במקרה הגרוע, הצומת הדרושה תהיה רחוקה מהשורש מרחק של כ-h קשתות, כלומר במקרה הגרוע הפונקציה תעבור על פני $O(logn)$ צמתים ותבצע $O(logn)$ פעולות.

סך הכל הפונקציה פועלת במקרה הגרוע בסיבוכיות זמן של $O(logn)$.

itemInIndex

מתודת עזר בשביל המחלקה TreeList – מקבלת מספר שלם ומחזירה את הItem של הצומת בעץ שדרגתה i+1.

הפונקציה משתמשת בפונקציית העזר nodeInIndex הפועלת בסיבוכיות זמן של $O(logn)$, כלומר פועלת במקרה הגרוע בסיבוכיות זמן של $O(logn)$.

insertInIndex

מתודת עזר בשביל המחלקה TreeList – מקבלת מספר שלם i וצומת ומכניסה את הצומת לעץ כך שדרגתה תיהיה i+1.

בבדי למצוא את המיקום המתאים לצומת החדשה, הפונקציה משתמשת בפונקציית העזר nodeInIndex הפועלת בסיבוכיות $O(logn)$ במקרה הגרוע. במקרה הצורך הפונקציה משתמשת בפונקצייות העזר maxNode ו- predecessor הפועלות בסיבוכיות $O(logn)$ במקרה הגרוע.

לאחר מציאת המיקום, פעולת ההכנסה מתבצעת בזמן קבוע (מספר קבוע של פעולות).

לאחר ההכנסה, הפונקציה מתקנת את העץ חזרה להיות AVL תקין באמצעות מעבר על כל הדרך מהצומת החדש ועד לשורש, ובדרך מעדכנת את הגובה והגודל של כל צומת. כלומר התיקון והעדכון עובר לא יותר מh צמתים בדרך, כלומר $\mathrm{O(logn)}$ צמתים בסיבוכיות זמן של $\mathrm{O(logn)}$. סך הכל במקרה הגרוע, הפונקציה פועלת בסיבוכיות זמן של $\mathrm{O(3logn)} = \mathrm{O(logn)}$.

deleteInIndex

מתודת עזר בשביל המחלקה TreeList – מקבלת מספר שלם i ומוחקת את הצומת שדרגתה תיהיה i+1 מהעץ.

בבדי למצוא את הצומת למחיקה, הפונקציה משתמשת בפונקציית העזר nodeInIndex הפועלת בסיבוכיות $O(logn)$, ואז מוחקת את הצומת שנמצאה בעזרת הפונקציה delete הפועלת בסיבוכיות זמן של $O(logn)$. כלומר, סך הכל במקרה הגרוע, הפונקציה פועלת בסיבוכיות זמן של $\mathrm{O(2logn)} = \mathrm{O(logn)}$.

AVLNode

שדות

Item – עצם מסוג Item המחזיק את הערך והמפתח של הצומת.

parent – עצם מסוג IAVLNode. מצביע להורה של הnode. במידה והnode הוא שורש, מצביע לnull.

leftSon – עצם מסוג IAVLNode. מצביע לבן השמאלי של הnode. במידה ואין לה בן שמאלי – מצביע לnull.

rightSon – עצם מסוג IAVLNode. מצביע לבן הימני של הnode. במידה ואין לה בן ימני – מצביע לnull.

height –עצם מסוג int. ערך גובה התת העץ ששורשו הצומת הזאת. מאותחל ל0.

size – עצם מסוג int. ערך מספר הצמתים בתת העץ ששורשו הצומת הזאת, כולל הצומת עצמה, דהיינו מאותחל ל1.

מתודות

החזרת הערך משדה מסויים או השמת ערך בשדה (get/set) של המחלקה דורש פעולות המתבצעות בזמן קבוע בלבד, כלומר כל פונקציה ברשימה הבאה מתבצעת בסיבוכיות זמן של $O(1)$ במקרה הגרוע:

- getItem
- getKey
- getValue
- setLeft
- getleft
- setRight
- getRight
- setParent
- getParent
- setHeight
- getHeight
- setSize
- getSize

balanceFactor

המתודה מקבלת צומת ומחזירה את גורם האיזון שלה.

חישוב גורם האיזון של צומת מתבצע על ידי חישוב אריתמטי פשוט: גובה תת העץ השמאלי פחות גובה תת העץ הימני (שאת ערכם ניתן לקבל בסיבוכיות זמן של $O(1)$) . בסך הכל הפונקציה מבצעת מספר פעולות קבוע בסיבוכיות זמן של $O(1)$ במקרה הגרוע.

isCriminal

המתודה מקבלת צומת, מחזירה true אם היא עברייינית AVL (צומת שגורם האיזון שלה גדול מ1 או קטן מ-1-), false אחרת.

הפונקציה משתמשת בפונקציית העזר balanceFactor הפועלת בסיבוכיות זמן קבועה, ומבצעת פעולות קבועות בזמן קבוע בסיבוכיות זמן של $O(1)$.

TreeList

שדות

tree – עצם מסוג AVLTree המחזיק את העץ המייצג את הרשימה.

מתודות

Retrieve

המתודה מקבלת מספר שלם i ומחזירה את הItem הנמצא במקום הi ברשימה.

הפונקציה מבצעת שימוש בפונקציית העזר itemInIndex של המחלקה AVLTree הפועלת בסיבוכיות זמן של $O(\log n)$ כמוסבר בניתוח הסיבוכיות של המחלקה AVLTree . חוץ משימוש זה, המחלקה מבצעת פעולות קבועות בסיבוכיות זמן קבועה. סך הכל הפונקציה פועלת במקרה הגרוע בסיבוכיות זמן של $O(logn)$.

Insert

המתודה מקבלת מספר שלם i ועצם מסוג item, ומכניסה את הItem למקום הi ברשימה.

בנוסף מחזירה את ערך הפלט של פונקציית העזר insertInIndex של המחלקה AVLTree הפועלת בסיבוכיות זמן של $O(\log n)$ כמוסבר בניתוח הסיבוכיות של המחלקה AVLTree . סך הכל הפונקציה פועלת במקרה הגרוע בסיבוכיות זמן של $O(logn)$.

Delete

המתודה מקבלת מספר שלם i ומוחקת את הItem הנמצא במקום הi ברשימה.

בנוסף מחזירה את ערך הפלט של פונקציית העזר deleteInIndex של המחלקה AVLTree הפועלת בסיבוכיות זמן של $O(\log n)$ כמוסבר בניתוח הסיבוכיות של המחלקה AVLTree . סך הכל הפונקציה פועלת במקרה הגרוע בסיבוכיות זמן של $O(logn)$.

CircularList

שדות

circularArr – מערך של עצמים מסוג Item בו נשתמש כמערך מעגלי, בו נתחזק את הרשימה.

maxLen – מספר שלם, גודל המערך המעגלי. מומש כfinal כיוון שבשום שלב אין צורך או אפשרות לשנותו.

size – מספר שלם, גודל הרשימה, כלומר, מספר האיברים ה"אמיתיים" ברשימה (שאינם ערכי זבל או שאינם ריקים) . מאותחל להיות 0, כל הכנסת איבר תגדיל אותו ב1, וכל מחיקה תקטין אותו ב1.

start – מספר שלם, ערך האינדקס במערך circularArr בו מתחילה הרשימה. כלומר circularArr[start] יחזיק מצביע לאיבר הראשון ברשימה. עבור רשימה ריקה יאותחל ל1-.

end – מספר שלם, ערך האינדקס במערך circularArr בו נגמרת הרשימה. כלומר circularArr[end] יחזיק מצביע לאיבר האחרון ברשימה. עבור רשימה ריקה יאותחל ל1-.

מתודות

Retrieve

הפונקציה מקבלת אינדקס i, ומחזירה את הitem שבמיקום הi ברשימה. הפונקציה פועלת בסיבוכיות זמן של $O(1)$, משום שבידוע גישה לאיבר ספציפי במערך מתבצעת בסיבוכיות $O(1)$.

Insert

הפונקציה מקבלת שני מספרים שלמים: מפתח וערך (מהם ניצור אייטם חדש), ואינדקס i. הפונקציה יוצרת אייטם חדש עם המפתח והערך, מכניסה אותו לרשימה במיקום i , כלומר מכניסה למערך circularArr של הרשימה, את האיבר במיקום start+i. הפעולה בעצם בודקת האם i גדול או קטן מחצי מsize, דהיינו האם האיבר ייכנס למיקום שקרוב יותר לתחילת הרשימה או לסופה, ולאחר מכן מזיזה את האיברים שבין קצה הרשימה לאינקס i (הקצה הקרוב יותר לו) מיקום אחד, ומכניסה את האייטם החדש במיקום i. בעצם, הפונקציה מזיזה במקרה הגרוע (שבו i=size/2) size/2 איברים, לכן הסיבוכיות במקרה הגרוע היא $O(n)$, כאשר n הוא מספר האיברים ברשימה. כמובן שלאחר הפעולה start יקטן ב1 או end יגדל ב1, וsize יגדל ב1.

Delete

הפונקציה מקבלת אינדקס i, ומוחקת את האיבר שבמיקום i מהרשימה, כלומר את האיבר במקום start+i מהמערך circularArr . בדומה לפונקציית insert, הפונקציה מזהה לאיזה מקצוות הרשימה i קרוב יותר, ומזיזה את כל האיברים שבין קצה זה לבין i מיקום אחד לכיוון i, וכך בעצם "דורסת" את האיבר שהיה במיקום i. בעצם, הפונקציה מזיזה במקרה הגרוע (שבו i=size/2) size/2 איברים. לכן במקרה הגרוע נצטרך להזיז n/2 איברים, ולכן הסיבוכיות במקרה הגרוע היא $O(n)$, כאשר n הוא גודל הרשימה. כמובן שלאחר הפעולה start יגדל ב1 או end יקטן ב1, וsize יקטן ב1.

מדידות

1) בבדי להדגים את היתרון של רשימה מעגלית על פני רשימה עצית בחרנו לבצע הכנסה בסוף הרשימה, כיוון שברשימה מעגלית הממומשת על ידי מערך, הכנסה בסוף לוקחת סיבוכיות זמן קבוע כי לא נדרשות פעולות עדכון או סידור מחדש כמו שנדרשות ברשימה העצית בהכנסה דומה. הכנסה בסוף ברשימה מעגלית מתבצעת ב$O(1)$ במקרה הגרוע בעוד שברשימה עצית הדבר ייקח $O(logn)$. הערכים להם ציפינו היו שהזמן הממוצע להכנסה עבור הכנסת k איברים לרשימה עצית יהיה גדול פי בערך $\log(k)$ מההכנסה לרשימה מעגלית.
תוצאות המדידות:

מספר סידורי	מספר פעולות	זמן הכנסה ממוצע עבור רשימה מעגלית	זמן הכנסה ממוצע עבור רשימה עצית (בns)	כמות גלגולים ימינה ממוצעת עבור רשימה עצית	כמות גלגולים שמאלה ממוצעת עבור רשימה עצית
1	10,000	167	2072	0	0.999
2	20,000	610	5084	0	0.999
3	30,000	954	14352	0	0.999
4	40,000	2327	14796	0	0.999
5	50,000	1110	20110	0	0.999
6	60,000	1599	23142	0	0.999
7	70,000	2224	30422	0	0.999
8	80,000	2863	40704	0	0.999
9	90,000	3639	52470	0	0.999
10	100,000	2909	68930	0	0.999

מן הטבלה ניתן להסיק שבמידה והמשתמש צריך רשימה שתייעל הכנסת איברים לסוף או להתחלה (כלומר הכנסה לקצוות), עדיף יהיה לבחור להשתמש ברשימה המעגלית.

2) בבדי להדגים את היתרון של רשימה עצית על פני מעגלית בחרנו להכניס את האיברים באמצע הרשימה, כיוון שסיבוכיות הזמן של רשימה מעגלית במקרה הזה יהיה $O\left(\frac{n}{2}+1\right)=O(n)$ בעוד שהכנסה לרשימה עצית תיקח סיבוכיות זמן של $O(logn)$.
הסיבה לכך היא שברשימה מעגלית עלינו להעביר את כל האיברים הנמצאים באינדקס גדול מאינדקס ההכנסה, אינדקס אחד ימינה (או הקטנים אינדקס שמאלה), כלומר להזיז $\frac{n}{2}$ איברים, בעוד שברשימה עצית כל הכנסה לוקחת במקרה הגרוע $O(logn)$.
הערכים להם ציפינו היו שהזמן הממוצע להכנסה עבור הכנסת k איברים לרשימה מעגלית יהיה גדול אקספוננציאלית מהכנסה דומה לרשימה העצית.

מספר סידורי	מספר פעולות	זמן הכנסה ממוצע עבור רשימה מעגלית (בns)	זמן הכנסה ממוצע עבור רשימה עצית (בns)	כמות גלגולים ימינה ממוצעת עבור רשימה עצית	כמות גלגולים שמאלה ממוצעת עבור רשימה עצית
1	10,000	45441	1857	0.812	0.81
2	20,000	370473	4910	0.813	0.811
3	30,000	1069232	11794	0.813	0.812
4	40,000	2990509	15353	0.813	0.812
5	50,000	6591306	22459	0.813	0.812
6	60,000	1.02940966194E7	25989	0.813	0.811
7	70,000	1.6893739310000002E7	20603	0.813	0.812
8	80,000	2.431379264E7	25999	0.813	0.812
9	90,000	3.66340303791E7	38199	0.813	0.812
10	100,000	5.63826182E7	43483	0.813	0.812

הערכים שקיבלנו אכן תאמו את הציפיות. ניתן להסיק שבבדי לייעל הכנסת איבר באמצע הרשימה נעדיף להשתמש ברשימה עצית.

3) נבדוק את המקרה הממוצע. לפי המדידות בסעיפים 1 ו2 אפשר להניח שבממוצע, הכנסת איברים לרשימה עצית תהיה יותר יעילה מההכנסה לרשימה מעגלית, כיוון שבממוצע המספר הרנדומלי ייפול קרוב לאמצע (התפלגות אחידה- עקומת הפעמון), ושם לפי סעיף 2 קיים יתרון משמעותי לרשימה העצית. בנוסף האינטואיציה אומרת שהמרחק בין 1 ל logn קטן יותר מהמרחק בין logn לn , כלומר היתרון של הרשימה העצית יעפיל על היתרון של המעגלית.

מספר סידורי	מספר פעולות	זמן הכנסה ממוצע עבור רשימה מעגלית (בns)	זמן הכנסה ממוצע עבור רשימה עצית (בns)	כמות גלגולים ימינה ממוצעת עבור רשימה עצית	כמות גלגולים שמאלה ממוצעת עבור רשימה עצית
1	10,000	20277	2743	0.35	0.35
2	20,000	190195	7091	0.35	0.35
3	30,000	456514	21851	0.35	0.35
4	40,000	1114478	21748	0.35	0.35
5	50,000	2207756	27258	0.35	0.35
6	60,000	3911407	24833	0.35	0.35
7	70,000	7052082	37795	0.35	0.35
8	80,000	1.063046552E7	45081	0.35	0.35
9	90,000	1.61332197291E7	62214	0.35	0.35
10	100,000	2.1609636200000003E7	75553	0.35	0.35

גם כאן הערכים שקיבלנו אכן תאמו את הציפיות. ממדידה זו ניתן להסיק שבאופן כללי עבור רשימה בה נכניס איברים במיקומים רנדומלים, עדיף יהיה להשתמש במימוש העצי.