<u>תיעוד פונקציות :</u>

<u>המחלקה AVLNode:</u>

__init__(self,key,val) :

הפונקציה יוצרת צומת חדש ומבצעת השמה דיפולטית להשדות: Self.Key=key , self.value=val המפתח והערך מקבלים אותם

נאתחל ההורה וגם הבנים הימני והשמאלי ל-None , בנוסף נאתחל שדה הגובה ל 1-

get_left(self):

הפונקציה מחזירה את הבן השמאלי .

סיבוכיות זמן : $O(1)$

get_right(self):

הפונקציה מחזירה הבן את הימני .

סיבוכיות זמן : $O(1)$

get_parent(self):

הפונקציה מחזירה את ההורה של צומת אם קיים , במקרה של שורש מחזירה None, **סיבוכיות זמן** : $O(1)$

get_value(self):

הפונקציה מחזירה את ה-value של הצומת , לצומת וירטואלי מחזירה None,

סיבוכיות זמן : $O(1)$

get_key(self):

הפונקציה מחזירה את ה-key של הצומת , לצומת וירטואלי מחזירה None,

סיבוכיות זמן : $O(1)$

get_height(self):

הפונקציה מחזירה את הגובה של הצומת , לצומת וירטואלי מחזירה 1-

סיבוכיות זמן : $O(1)$

set_left(self,node):

הפונקציה מעדכנת את הבן השמאלי של הצומת self לצומת שמקבלת node , בנוסף מעדכנים את ההורה של ה-node שהוא מקבלת ל self .

סיבוכיות זמן : $O(1)$

set_right(self,node):

הפונקציה מעדכנת את הבן הימני של הצומת self לצומת שמקבלת node , בנוסף מעדכנים את ההורה של ה-node שהוא מקבלת ל self .

סיבוכיות זמן: $O(1)$

set_parent(self,node):

הפונקציה מעדכנת את ההורה של הצומת self לצומת שמקבלת node

סיבוכיות זמן: $O(1)$

set_key(self,key):

הפונקציה מעדכנת את המפתח - key של הצומת self ל-key שמקבלת

סיבוכיות זמן: $O(1)$

set_value(self,value):

הפונקציה מעדכנת את הערך – value של הצומת self ל-value שמקבלת

סיבוכיות זמן: $O(1)$

set_height(self,h):

הפונקציה מעדכנת את הגובה – height של הצומת self ל-h שמקבלת

סיבוכיות זמן: $O(1)$

Is_real_node(self):

הפונקציה מחזירה True אם הצומת הוא וירטואלי , אחרת False

סיבוכיות זמן: $O(1)$

Height_calculator(self):

הפונקציה מחשבת את הגובה החדש של צומת ומעדכנת אותו

סיבוכיות זמן: $O(1)$

המחלקה AVLTree:

__init__(self):

הפונקציה מאתחלת את השורש – root ל-None , בנוסף הוספנו שדה חדש Size שמתאר את כמות הצמתים בעץ ומאתחלים אותו ל 0

סיבוכיות זמן: $O(1)$

Search(self,key) :

הפונקציה מקבלת מפתח ,נחפש את הצומת עם המפתח key בעזרת לולאה , נגדיר משתנה current = self.root כל עוד הצומת אינו None ז"א לא הגענו לצומת וירטואלי נבדוק אם המפתח של הצומת שווה ל key אם כן נחזיר את הצומת , אחרת אם המפתח של הצומת גדול נתקדם שמאלה , אחרת ימינה . אם לא מצאנו נחזיר None

סיבוכיות זמן: $O(log\ n)$

ניתוח הסיבוכיות : במקרה הגרוע לא נמצא את המפתח *key* ולכן עלינו לעבור על כמות צמתים כגובה העץ כי בכל שלב או מתקדמים ימינה או שמאלי אז סה"כ: $O(log\ n)$

get_root(self) :

פשוט מחזירים את השדה *root* שמעודכן במהלך הריצה

סיבוכיות זמן: $O(1)$

avl_to_array(self) :

הפונקציה מגדירה רשימה ריקה , מעבירה את השורש והרשימה כפרמטרים וקוראת ל avl_to_array_helper

סיבוכיות זמן: $O(n)$ כי היא קוראת לפעולה שדורשת O(n)

avl_to_array_helper(self,node,arr) :

הפונקציה מקבלת כפרמטרים node ורשימה arr , המימוש של הפונקציה הוא ריקורסיבי תנאי העצירה : אם הצומת הוא וירטואלי , תחזור .

אנו רוצים שהרשימה תהיה ממוינות לכן , הפונקציה קוראת לעצמה ריקורסיבית עם הבן השמאלי , אחר כך מוסיפה את המפתח key והערך value כ touple למערך , אחר כך הפונקציה קוראת לעצמה עם הבן הימני .

סיבוכיות זמן: $O(n)$

ניתוח סיבוכיות : עוברים על העץ in-order ומבקרים בכל צומת פעם אחת , כל ביקור בצומת לוקח זמן קבוע , יש n צמתים בעץ לכן סה"כ $O(n)$ כאלה , בנוסף בכל פעם עלינו להוסיף איבר לרשימה , הוספה איבר לרשימה דורשת זמן קבוע , יש *n* הכנסות ולכן סה"כ $O(n)$ עלות של הכנסות .

לבסוף נקבל : $T = O(n) + O(n) = O(n)$

Successor(self,node) :

הפונקציה מקבלת *node* ומחפשת את ה *successor* שלו , המימוש נעשה ע"פ הפסאדו קוד שנלמד בכיתה , הפונקציה קוראת לפונקציה *Min* במקרה שיש לצומת בן ימני .

סיבוכיות זמן : $O(\log n)$

ניתוח סיבוכיות : ראינו בכיתה שבמקרה הגרוע נבצע $O(h)$ פעמים node.right כאשר h הוא גובה העץ , כיוון שזה עץ AVL אז $h = \log n$ ולכן נקבל שהסיבוכיות : $O(\log n)$

Min(self,node):

הפונקציה מקבלת צומת בעץ self ומחזירה את הצומת עם המפתח הכי קטן בתת העץ השמאלי של הצומת

סיבוכיות זמן : $O(\log n)$

ניתוח סיבוכיות : אותו טיעון מלמעלה אבל במקום node.right נבצע node.left

Delete(self,node):

הפונקציה מקבלת צומת בעץ ומוחקת אותה ע"י קריאה לפונקצית העזר delete_node , בנוסף הפונקציה מחזירה כמות פעולות האיזון הנדרשות אחרי המחיקה .

סיבוכיות זמן: $O(\log n)$.

ניתוח סיבוכיות : הפונקציה רק קוראת לפונקצית העזר לכן יש להן אותה סיבוכיות.

Delete_node(self,node) :

הפונקציה מקבלת צומת בעץ , בודקת אם לצומת יש / אין בן ימני ושמאלי , ובכל מקרה מבצעת את השינויים הנדרשים כמו : עדכון הבנים , עדכון ההורה של הבנים ועדכון הבנים של ההורה . בסוף הפונקציה קוראת לפונקציה Deletion_rebalance ומעבירה לה את ההורה של הצומת כפרמטר .

סיבוכיות זמן: $O(\log n)$.

ניתוח סיבוכיות : כל פעולה של עדכון בנים או הורה לוקחת זמן קבוע , במקרה הגרוע עלינו למחוק צומת עם שני בנים לכן עלינו למצוא את העוקב שלוקח $O(\log n)$ במקרה הגרוע

Deletion_balance(self,node) :

הפונקציה מקבלת צומת שעליה צריך לבדוק את כל השלבים לאיזון שנלמדו בכיתה ומבצעת את השלבים ובסוף מחזירה מספר פעולות האיזון הנדרשים, אם ה*BF* הוא 2 היא קוראת לפונקציה *delete_rotate*

סיבוכיות זמן: $O(\log n)$.

ניתוח סיבוכיות : במקרה הגרוע ביותר עלינו לבצע איזון מהצומת לשורש ולכן הסיבוכיות כעומק העץ

delete_rotate(self,node) :

הפונקציה מקבלת צומת שעליה צריך לבצע פעולות גלגול כפול או רגיל כפי שנלמד בכיתה ,
הפונקציה גם מעדכנת את הגבהים ומחזירה 2 אם ביצענו גלגול כפול אחרת 1

סיבוכיות זמן: $O(1)$.

Split(self,node) :

הפונקציה מקבלת צומת שעושים לפיו פיצול , נגדיר שני עצים left_subtree ו right_subtree שמכילים כל המפתחות הקטנים והגדולים בהתאמה . בתוך לולאת while נבדוק אם הצומת היא בן ימני להורה אז ההורה קטן ממנו ולכן נעשה join בין תת העץ השמאלי של ההורה ו left_subtree אחרת נעשה בין התת עץ הימני של ההורה עם right_subtree .

סיבוכיות זמן: $O(\log n)$.

ניתוח סיבוכיות : במקרה הגרוע ביותר לולאת ה while עולה עד לבן של השורש , כפי שראינו בהרצאה , אנו עושים telescopic joins ולכן הגובה לכל תת עץ Tk יקטן ככל שהתקדמנו , ניתחנו את הסיבכיות בהרצאה לפי 2 למות וקיבלנו סה"כ: $O(\log n)$.

:insert (self, key , val)

תחילה ניצור בעזרת הבנאי של AVLNode צומת חדש node , ונעדכן את הגובה שלו ל- 0. ניצור שני צמתים וירטואליים, ונעדכן את שדות ה- left והוא- right של node לאותם שני צמתים.

אם העץ ריק נעדכן את שורש העץ להיות node ונגדיל את שדה ה- size ב- 1. אחרת נקרא לפונקציית העזר regular_insert שמבצעת הכנסה של node במקום המתאים.

:regular_insert(self, node)

תחילה נשמור משתנה new_p שישמור את מיקום האבא של node (נמצא אותו בעזרת פונקציית node_position שמחפשת את מיקום node, על ידי כך שהיא מתחילה מהשורש ויורדת שמאלה או ימינה בהתאם להשוואות של מפתחות צמתי העץ עם key של node , ואם במהלך הריצה מצאנו מפתח ששווה ל key נחזיר false). כעת כשיש לנו את המיקום של node נשאל אם הוא false, אם כן נחזיר false, אחרת: נשאל אם הוא בן שמאלי או ימני של אבא שלו ונעדכן את שדה ה left/right של האב בהתאם, ונעדכן גם כן את שדה ה- parent של node.

כעת נחזור ל- insert: אם הקריאה ל- regular_insert מחזירה false, זה אומר שהמפתח כבר קיים בעץ ואין צורך להכניס איבר, ובכן נחזיר 0 (לא הצטרכנו לבצע פעולות איזון). אחרת נגדיל את size העץ ב 1 ונקרא לפונקציה Balance(self,p,son) עם node והאבא p של node, שבודקת אם צריך לבצע סיבובים ובמבצעת בהתאם. ונחזיר את מס׳ פעולות האיזון שהיא מחזירה.

: Balance(self,p,son)

נאתחל משתנה balance ל 0 (כדי לחשב את מס׳ פעולות האיזון). נבנה לולאה שרצה כל עוד לא הגענו לצומת None. בלולאה נאתחל משתנה bf שישמור את ה balance factor עבור הצומת p (על ידי הפונקציה BF(self,node) שמחשבת את ה balance factor של node ע״י כך שהיא מחשבת את הפרש הגבהים של תת העץ השמאלי ותת העץ הימני של node, ומחזירה את התוצאה). נאתחל עוד משתנה h_changed שישמור אם הגובה של הצומת p השתנה או לא לאחר הכנסת node, על ידי פונקציית עזר height_has_changed(self, node, son) שבודקת אם הגובה הנוכחי של node (לפני ההכנסה) קטן מהגובה של הבן (שבתת עץ שלו קרו השינויים) ועוד 1, אם כן זה אומר שהגובה של node גדל לאחר ההכנסה (זאת מפני שאז הגובה של תת העץ של son גדול מגובה תת העץ של הבן האחר).לאחר מכן נבדוק שלושה מקרים:

מקרה 1: אם bf בערך מוחלט קטן מ- 2 וגם כן הגובה של p לא השתנה, אז נחזיר את balance (כי זה אומר שכל הצמתים מעליו לא ראו את השינוי).

מקרה 2: אם bf בערך מוחלט קטן מ- 2 אך כן התרחש שינוי בגובה, אז נעלה את שדה גבוה של p ב-1, נוסיף 1 ל- balance (כי שינוי גובה מחוץ ל- rotation נחשב לאיזון), ונעדכן את son להיות p , ואת p להיות האב של p (כך נוכל לעלות במעלה העץ).

מקרה 3: אם bf בערך מוחלט שווה ל- 2, זה אומר שהגובה של p גדל ובכן נוסיף לו 1, (ואז נצטרך להוסיף 1 גם ל balance). נצטרך גם כן לבצע rotation , ובכן נקרא לפונקציית עזר (rotate(self, node, son עם הפרמטרים p ו- son, ונוסיף ל- balance את מה ש rotate מחזירה. ולבסוף הפונקציה תחזיר את balance. ואם לא היה צריך לבצע איזונים הפונקציה תחזיר את balance ששווה אז ל- 0.

:rotate(self, node, son)

נאתחל שני משתנים n_bf ו- s_bf שישמרו את ה balance factors של node ושל son בהתאמה.

נחלק ל- 4 מקרים, ונבצע rotation בהתאם:-

מקרה 1: אם n_bf חיובי (= 2+) ו- s_bf חיובי (=1+) או שווה ל-0 (בשביל לטפל במקרה שב join יש צומת עם balance factor 2 והבן שלו עם balance factor 0), אזי נצטרך לבצע סיבוב ימינה. ובכן נקרא לפונקציית העזר do_right_rotation(self, node, son) עם הערכים node ו- son. ונחשב מחדש את הגבהים שלהם בעזרת הפונקציה (height_calculator(self של AVLNode, ונחזיר 1 (ביצענו פעולת איזון אחת).

מקרה 2: אם n_bf שלילי (=2-) ו- s_bf שלילי (=1-) או שווה ל-0, נבצע סיבוב שמאלי בעזרת פונקציית העזר do_left_rotation(self, node, son) (נקרא לה עם הפרמטרים node ו- son) ונחשב מחדש את הגבהים, ונחזיר 1.

מקרה 3: אם n_bf חיובי(=2+) ו- s_bf שלילי (= 1-), אז נצטרך לבצע שני סיבובים. ראשית נבצע סיבוב שמאלה עם הפרמטרים son ו- grandson (כך ש- grandson זה הבן הימני של son), ונבצע לאחר מכן סיבוב ימינה עם node ו- grandson (כי לאחר הסיבוב השמאלי grandson נהיה הבן של node). נחשב מחדש את הגבהים של שלושת הצמתים node, son , grandson, ונחזיר 2 (היו שני סיבובים).

מקרה 4: אם n_bf שלילי (=2-) ו- s_bf חיובי (=1+), נבצע סיבוב ימינה עם הפרמטרים son ו- grandson (כאשר grandson הוא בן שמאלי של son), ולאחר מכן סיבוב שמאלה עם הפרמטרים node ו- grandson. בסוף, נחשב גבהים ונחזיר 2 .

do_left_ rotation(self, node, son):

נבדוק ראשית אם node הוא שורש העץ, אם כן נעשה השמה ל- son לתוך השדה root (כי נרצה לבצע סיבוב, ואז node ו- son יחליפו תפקידים), ונעדכן את השדה parent של son להיות None.

נעדכן את שדה right של node להיות הבן השמאלי של son (ושאר תתי העצים ישארו במקום) ונעדכן בהתאם את השדה parent של הבן השמאלי להיות node. כעת, node נהיה הבן השמאלי של son ובכן נעשה לו השמה לתוך שדה left של son ונעדכן את ההורה שלו להיות ההורה הישן של node.

אם ההורה הישן של node לא היה None, נבדוק אם node הוא בן שמאלי או ימני שלו (עדיין לא עדכנו את שדה parent של node), אם הוא בן שמאלי נעדכן את שדה left של ההורה הישן להיות son, אחרת נעדכן את שדה right שלו להיות son. ולבסוף נעדכן את שדה parent של node להיות son.

do_right_ rotation(self, node, son):

בדיוק כמו האלגוריתם של do_left_ rotation רק המקרה הסימטרי.

סיבוכיות זמן של insert : O(log n)

ניתוח סיבוכיות של insert:

כל ההשמות והבדיקות שביצענו לוקחות זמן קבוע (בפרט, קריאה לבנאי של AVLNode לוקחת זמן קבוע). קריאה ל regular_insert לוקחת O(log n) , ולבסוף קריאה ל- Balance לוקחת O(log n). ובכן סה״כ insert רצה ב O(log n).

ניתוח סיבוכיות של regular_insert:

בהתחלה קוראים ל- node_position שרצה בזמן (O(log n. הבדיקות שמבצעים לאחר מכן לוקחות זמן קבוע. ובכן סה״כ regular_insert רצה ב O(log n) .

ניתוח סיבוכיות של node_position:

מתחילים מ node ויורדים במורד העץ באמצעות פניות שמאלה/ ימינה, עד שמגיעים לצומת שמחפשים. ובכן לכל היותר נרוץ על גובה גובה העץ, כלומר קיבלנו סיבוכיות של O(log n).

ניתוח סיבוכיות של Balance:

מתחילים מ- p ועולים עד שמוצאים צומת שהגובה שלה לא השתנה, או שעושים סיבוב (או שניים) ועוצרים, או שבכלל מגיעים ל- None. כלומר לכל היותר נצטרך לרוץ למעלה כגובה העץ. ובכן סה״כ סיבוכיות זמן O(log n). ביצענו גם כן קריאות ל- BF ול- height_has_changed ששתיהן רצות ב- O(1) (באחת מבצעים חישוב של הפרש ובשנייה מבצעים בדיקה). בסוף במקרה שקראנו ל- rotate תיקח גם היא O(1).

ניתוח סיבוכיות של rotate:

ביצענו בדיקות והשמות ב O(1). הקריאה ל- height calculator לוקחת O(1). וכל אחת מ- do_left_ rotation ו- do_right_ rotation לוקחת גם היא O(1). ובכן סה״כ קיבלנו סיבוכיות של O(1).

ניתוח סיבוכיות של do_left_ rotation/ do_right_ rotation:

בפונקציות ביצענו רק מס׳ קבוע של בדיקות והשמות, ובכן שתיהן רצות ב-O(1) .

<u>join(self, tree2, key, val):</u>

תחילה נבצע כמה בדיקות , אם העצים ריקים פשוט נעשה insert רגיל (גם נבדק אם אחד או שני העצים ריקים)

כעת, ניצור את צומת x מ- key ו- val , ונחלק לכמה מקרים לפי הגובה והמפתחות ונקרא לפונקציה add עם פרמטר בוליאני אמת או שקר אם מפתחות עץ אחד קטן מאחר .

בסוף נעדכן את שדה הsize של self להיות סכום ה- size ״ים״ של שני העצים ועוד 1. ונחזיר כפלט את הפרש גבהי העצים ועוד 1.

:add(self , t1, t2 , x ,add_to_ left)

נשמור את השורש של t2 במשתנה needed, ואז נרצה לרדת במורד העץ הגדול עד שנמצא צומת שגובהה כגובה t1. ובכן נרוץ בלולאה על הגובה של t2. בתוך הלולאה נבדוק אם צריך לחבר את t1 לשמאל t2. אם כן נעדכן את needed להיות הבן השמאלי, אחרת נעדכן אותו להיות הבן הימני. נבדוק אם הגובה של needed קטן שווה לגובה של t1. אם כן, אז הגענו לצומת שאנו צריכים. נקרא להורה שלו c. ואז נבדוק אם add_to_left. אם כן נעדכן את הבן השמאלי של x להיות השורש של t1, ואת הבן הימני שלו להיות needed, ונעדכן את הבן השמאלי של c להיות x. אחרת שורש t1 יהיה הבן הימני ו- needed יהיה השמאלי, ונעדכן את הבן הימני של c להיות x.

כעת, נשנה את גובה x להיות הגובה של t1 ועוד 1 (הגובה של needed קטן שווה משל t1), נעדכן את השורש של self להיות השורש של t2 (העץ הגבוה), ואז נבדוק אם יש צורך לעשות איזון. נבדוק אם הגובה של c השתנה, אם כן אז נצטרך לעשות איזון ובכן נקרא לפונקציה Balance שמימשנו קודם עם הפרמטרים c -ו x.

סיבוכיות זמן של join: O(log n)

ניתוח סיבוכיות של join: בהתחלה אם ביצענו קריאה למתודה insert (שרצה בזמן O(log n)), אז הסיבוכיות הכוללת תהיה O(log n). אחרת נבצע מס׳ קבוע של השוואות ובדיקות, ונקרא לפונקציה add שרצה ב- O(log n). ובכן סה״כ קיבלנו סיבוכיות של O(log n).

ניתוח סיבוכיות של add:

התחלנו מהשורש של t2 וירדנו למטה עד שמצאנו צומת עם גובה מתאים, ובכן לכל היותר ביצענו מס׳ איטירציות כגובה העץ. בכל איטירציה ביצענו מס׳ קבוע של בדיקות והשמות ובנוסף יש סיכוי שקראנו לפונקציה Balance בשביל לאזן. Balance רצה ב O(log n) לכל היותר. ובכן סה״כ add רצה ב- O(log n).

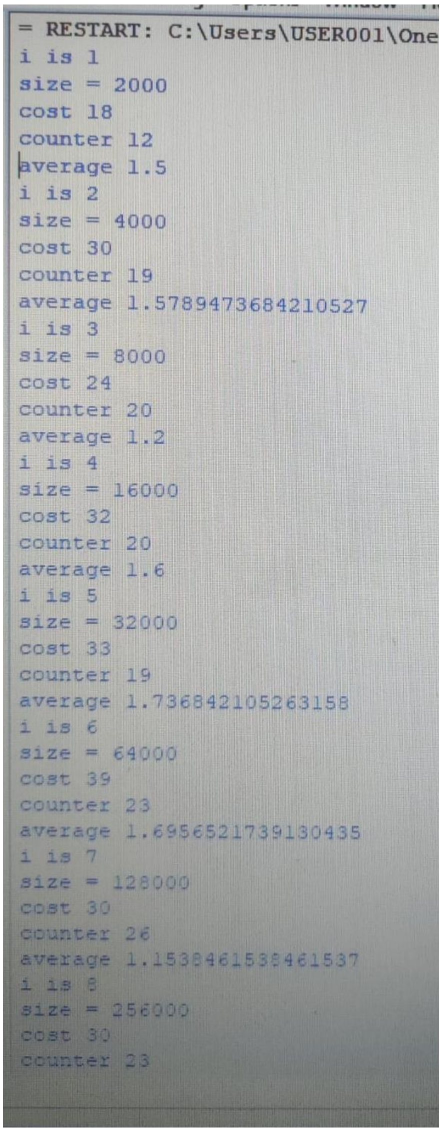
<u>חלק ניסויי / תיאורטי :</u>

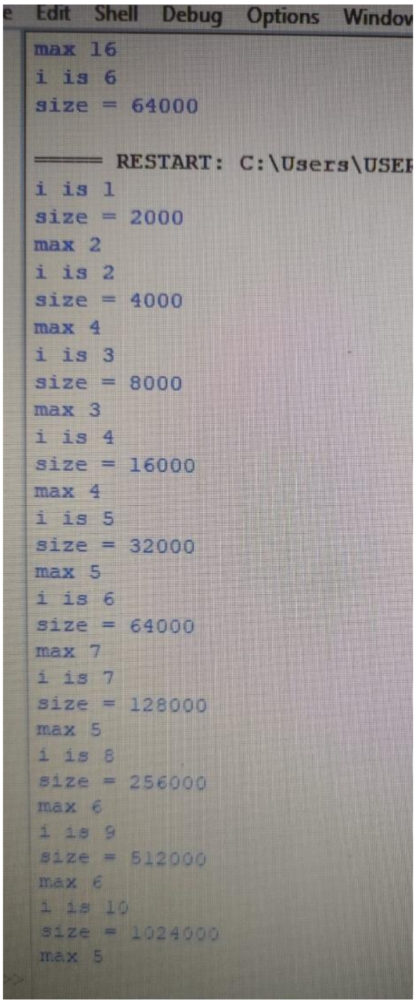
איך עשינו את הניסוי ?

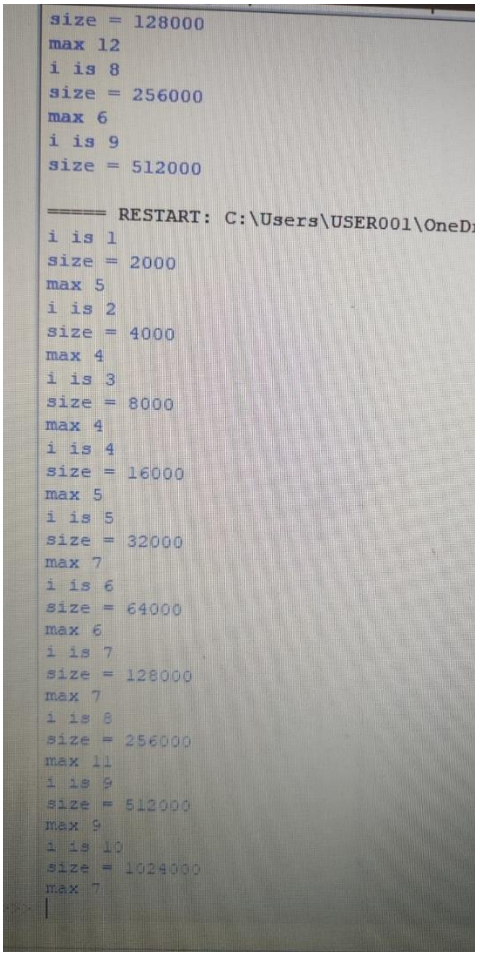
עבור הממוצע הוספנו counter ומשתנה cost שצובר עלויות ובסוף חישבנו ממוצע .

עבור עלות מקסימלית הגדרנו משתנה בשם max שמתעדכן אם העלות גדולה מערכו .

עשינו את הניסוי יותר מפעם אחת כדי שהמספרים יתייצבו , מצורף כמה תמונות (לא כולם)







.1

מספר סידורי i	עלות join ממוצע עבור split אקריא	עלות join מקסימלי עבור split אקראי	עלות join ממוצע עבור split של האיבר המקסימלי בתת העץ השמאלי	עלות join מקסימלי עבור split של האיבר המקסימלי בתת העץ השמאלי
1	1.8	5	1.81	12
2	1.75	6	1.73	13
3	1.71	4	1.65	15
4	1.68	9	1.678	15
5	1.775	9	1.73333	17
6	1.67	7	1.766	18
7	1.65	6	1.742	19
8	1.68	6	1.6995	20
9	1.8	9	1.833	21
10	1.64	7	1.687	22

. 2

Join ממוצע על איבר אקראי ומקסימלי :

בהינתן צומת אקראי v, ננתח את העלות הממוצעת שלה , נסמן את בעומק של הצומת ב d עלות split על צומת v היא $O(d)$ בנוסף מתקיים שמספר הפעמים שאנו קוראים ל join היא בדיוק כעומק של הצומת v וכן סה"כ עלות join ממוצעת של split הינו $O(1)$, נשים לב שקיבלנו שהעלות היא O(1) ללא תלות בצומת , לכן זהו גם חסם עבור join ממוצע על איבר מקסימלי בתת העץ השמאלי .

אכן קיבלנו תוצאות קבועות ללא תלות בקלט n וכפי שהסברנו, הניתוח היה על צומת אקראית ולכן ניתן לראות שהתוצאות של join ממוצע עבור איבר מקסימלי קרובות לתוצאות על איבר כללי או אקראי

.3

נשים לב שהמסלול מהצומת הגדול ביותר בתת העץ השמאלי לשורש הוא $O(\log n)$, עלינו לבצע join עם תת העץ הימני של השורש (עבור המפתחות הגדולים מהשורש) ולכן העלות היא $O(\log n)$, בנוסף כל join של תתי העצים במסלול מהצומת המקסימלי לשורש עולות $O(1)$ כי זהו עץ *AVL* ולכן הפרש הגבהים לכל היותר 1 , ובכן עלות *join* מקסימלית בניסוי הזה הינה $O(\log n)$