<u>מסמך תיעוד חיצוני</u>

מגישים:
אליקה גרייב – alikagaraev – 323222141
ישי ימיני – ishayyem – 322868852

מחלקת AVLNode

מייצגת צומת בודד בעץ AVL, היא כוללת את השדות:
key – מפתח של הצומת מסוג int
value – הערך של הצומת
left – מצביע לבן השמאלי של הצומת
right – מצביע לבן הימני של הצומת
height – שדה הגובה של הצומת
parent – מצביע לאב של הצומת

פונקציות במחלקה:

<u>init:</u> הבנאי מאתחל צומת חדש בעץ, מקבל כפרמטר את הkey והvalue של הצומת.
אם מדובר בצומת שאינו וירטואלי – הבנאי מייצר לו בן וירטואלי ימני ושמאלי, ומגדיר את הגובה להיות 0 (עלה).
במידה ומדובר בצומת וירטואלי – הבנאי מגדיר כי אין לו בן ימני או שמאלי (None) והגובה הוא 1- כדרוש.
בכל מקרה עבור צומת חדש הבנאי את השדה parent כNone.
סיבוכיות זמן הריצה של הבנאי היא $O(1)$.

<u>get_left</u>: הפונקציה מחזירה את הבן השמאלי של הצומת עליו היא מופעלת, אם לצומת אין כזה למעשה יוחזר בן וירטואלי.
הפונקציה מבצעת זאת על ידי החזרת השדה left של הצומת.
סיבוכיות זמן הריצה של הפונקציה זו היא $O(1)$.

<u>get_right</u>: הפונקציה מחזירה את הבן הימני של הצומת עליו היא מופעלת, אם לצומת אין כזה למעשה יוחזר בן וירטואלי.
הפונקציה מבצעת זאת על ידי החזרת השדה right של הצומת.
סיבוכיות זמן הריצה של הפונקציה זו היא $O(1)$.

<u>get_parent</u>: הפונקציה מחזירה את צומת ההורה של הצומת עליו היא מופעלת, אם לצומת אין כזה (שורש) יוחזר None.
הפונקציה מבצעת זאת על ידי החזרת השדה parent של הצומת.
סיבוכיות זמן הריצה של הפונקציה זו היא $O(1)$.

<u>get_key</u>: הפונקציה מחזירה את המפתח של הצומת עליו היא מופעלת, אם לצומת אין כזה (ובעצם מדובר בצומת וירטואלי) יוחזר None.
הפונקציה מבצעת זאת על ידי החזרת השדה key של הצומת.
סיבוכיות זמן הריצה של הפונקציה זו היא $O(1)$.

<u>get_value</u>: הפונקציה מחזירה את הערך של הצומת עליו היא מופעלת, אם לצומת אין כזה (ובעצם מדובר בצומת וירטואלי) יוחזר None.
הפונקציה מבצעת זאת על ידי החזרת השדה value של הצומת.
סיבוכיות זמן הריצה של הפונקציה זו היא $O(1)$.

get_height: הפונקציה מחזירה את הגובה של הצומת עליו היא מופעלת, אם מדובר בצומת וירטואלי יוחזר 1-.
הפונקציה מבצעת זאת על ידי החזרת השדה height של הצומת.
סיבוכיות זמן הריצה של הפונקציה זו היא $O(1)$.

get_bf: הפונקציה מחשבת ומחזירה את הבאלנס פקטור של הצומת עליו היא מופעלת.
הפונקציה בודקת האם מדובר בצומת וירטואלי על ידי קריאה ל-is_real_node ואם מדובר בצומת וירטואלי היא מחזירה 0.
במידה והצומת אינו וירטואלי היא מחשבת את הבאלנס פקטור על ידי קריאה לפונקציות get_left, get_right ו-get_height ומבצעת חישוב מתמטי של גובה הבן השמאלי של הצומת פחות גובה הבן הימני של הצומת.
מפני שמדובר בפונקציות עם זמן ריצה $O(1)$ ופעולה אריתמטית סיבוכיות זמן הריצה של הפונקציה זו היא $O(1)$.

set_left: הפונקציה מגדירה את הבן השמאלי של הצומת עליו היא מופעלת, מעדכנת את הגובה של הצומת ואת האב של הצומת שמוגדר כבן שמאלי.
תחילה הפונקציה בודקת האם מדובר בצומת וירטואלי על ידי קריאה ל-is_real_node, אם זה צומת שאינו וירטואלי אז היא מגדירה את השדה left של הצומת להיות הnode שהתקבל בקלט.
הפונקציה מגדירה את ההורה של אותו node להיות הצומת עצמו באמצעות הפונקציה set_parent ואז מעדכנת את הגובה של הצומת באמצעות set_height, כאשר חישוב הגובה החדש מתבצע עם קריאה ל-get_height על שדות הבן השמאלי והימני של הצומת, מקסימום וחישוב אריתמטי.
מפני שמדובר בפעולות עם זמן ריצה $O(1)$ ופעולה אריתמטית סיבוכיות זמן הריצה של הפונקציה זו היא $O(1)$.

set_right: הפונקציה מגדירה את הבן הימני של הצומת עליו היא מופעלת, מעדכנת את הגובה של הצומת ואת האב של הצומת שמוגדר כבן ימני.
תחילה הפונקציה בודקת האם מדובר בצומת וירטואלי על ידי קריאה לis_real_node , אם זה צומת שאינו וירטואלי אז היא מגדירה את השדה right של הצומת להיות הnode שהתקבל בקלט.
הפונקציה מגדירה את ההורה של אותו node להיות הצומת עצמו באמצעות הפונקציה set_parent ואז מעדכנת את הגובה של הצומת באמצעות set_height, כאשר חישוב הגובה החדש מתבצע עם קריאה ל-get_height על שדות הבן השמאלי והימני של הצומת, מקסימום וחישוב אריתמטי.
מפני שמדובר בפעולות עם זמן ריצה $O(1)$ ופעולה אריתמטית סיבוכיות זמן הריצה של הפונקציה זו היא $O(1)$.

set_parent: הפונקציה מגדירה את ההורה של הצומת עליו היא מופעלת.
הפונקציה מבצעת זאת באמצעות הגדרת השדה parent של הצומת להיות הnode שהתקבל בקלט.
סיבוכיות זמן הריצה של הפונקציה זו היא $O(1)$.

set_value: הפונקציה מגדירה את הערך של הצומת עליו היא מופעלת.
הפונקציה מבצעת זאת באמצעות הגדרת השדה value של הצומת להיות הערך שהתקבל בקלט.
סיבוכיות זמן הריצה של הפונקציה זו היא $O(1)$.

set_height: הפונקציה מגדירה את הגובה של הצומת עליו היא מופעלת.
הפונקציה מבצעת זאת באמצעות הגדרת השדה height של הצומת להיות הערך שהתקבל בקלט.
סיבוכיות זמן הריצה של הפונקציה זו היא $O(1)$.

is_real_node: הפונקציה בודקת האם הצומת עליו היא מופעלת הוא צומת וירטואלי, מחזירה true אם הצומת לא וירטואלי או false אם הצומת וירטואלי.
הפונקציה מבצעת זאת על ידי בדיקת השדה key והאם הוא לא None.
סיבוכיות זמן הריצה של הפונקציה זו היא $O(1)$.

מחלקת AVLTree:

מייצגת עץ AVL, היא כוללת את השדות:
root – השורש של העץ
tree_size – מספר הצמתים בעץ

init: הבנאי מאתחל עץ חדש.
הערכים הדיפולטיים עבור עץ ריק הם שורש None, וגודל עץ 0.
סיבוכיות זמן הריצה של הבנאי היא $O(1)$.

search: הפונקציה מחפשת בעץ צומת שהשדה key שלו זהה לקלט שהתקבל, מחזירה None במידה ולא קיים צומת כזה.
הפונקציה תחילה בודקת האם העץ ריק – במקרה זה מחזירה None כדרוש.
אחרת היא תקרא לפונקציה tree_position עם הקלט (key) שקיבלנו, הפונקציה הזו תחזיר את הצומת שערך ה-key שלו הוא הקרוב ביותר בערך הקלט המבוקש.
לאחר מכן הפונקציה בודקת האם הkey של הצומת שהתקבל תואם לקלט של הפונקציה או לא – אם כן יוחזר הצומת שנמצא, אם לא - יוחזר None כדרוש.
מפני שסיבוכיות זמן הריצה של tree_position היא $O(\log n)$ גם סיבוכיות זמן הריצה של פונקציה זו הוא $O(\log n)$ משום שהקריאה ל-tree_position מהווה את עיקר העבודה.

insert: הפונקציה מכניסה צומת לעץ ומאזנת אותו כך שישמור על הגדרת עץ AVL, היא מחזירה את מספר פעולות האיזון שנדרשו לשם כך.
הפונקציה מייצרת איבר חדש מסוג AVLNode עם ערכי ה-key וה-value שהתקבלו בקלט, ומגדילה את שדה ה-tree_size של העץ הנוכחי ב-1.
הפונקציה בודקת האם מדובר בעץ ריק – במידה וכן היא מגדירה את שדה root של העץ להיות הצומת החדש שנוצר ומחזירה 0 – מפני שלא נדרשו פעולות איזון כלשהן.
אחרת, הפונקציה מוצאת את המיקום בו צריך להכניס את הצומת על ידי קריאה לפונקציה tree_position עם הקלט key, ושומרת את הגובה הנוכחי של הצומת שהתקבל מהפעלת tree_position.
במידה והמפתח של הצומת שהתקבל גדול יותר ממפתח הצומת שצריך להכניס – הפונקציה מגדירה את הצומת החדש להיות הבן השמאלי שלו על ידי קריאה ל-set_left, אחרת הפונקציה מגדירה את הצומת החדש להיות הבן השמאלי שלו על ידי קריאה ל-set_right (ניתן להניח כי המפתח לא קיים כבר בעץ).
הפונקציה מחזירה את המתקבל מקריאה ל-fix_tree, שזה מספר פעולות האיזון הנדרשות.
נשים לב שסיבוכיות זמן הריצה של הכנסת הצומת עצמו היא $O(\log n)$ בדומה לסיבוכיות זמן הריצה של tree_position, וכי סיבוכיות זמן הריצה של ספירת פעולות האיזון שהתבצעו היא בדומה לסיבוכיות זמן הריצה של fix_tree, שהיא $O(\log n)$, אך אלו רצים זה לאחר זה ולכן סך הסיבוכיות הינו $O(\log n)$.
(הערת חשובה! בהרצאה למדנו כי לאחר insert יש לכל היותר סיבוב אחד (או סיבוב כפול אחד), אבל לפי החישוב שלנו של פעולות איזון (שכולל גם עדכון גובה), אנו אכן עלולים גם ב-insert לעשות $\log n$ פעולות איזון)

tree_position: מחפשת בעץ צומת שהשדה key שלו זהה לקלט שהתקבל, מחזירה את הצומת האחרון שהגיעה אליו בחיפוש (שערכו הוא הקרוב ביותר לקלט המבוקש).
הפונקציה עושה זאת על ידי מעבר על הצמתים של העץ החל מהשורש, כל עוד הצומת אינו צומת וירטואלי הפונקציה בודקת האם המפתח של הצומת תואם לקלט – אם כן היא תחזיר את הצומת, אם לא, הפונקציה כעת תבדוק האם המפתח קטן או גדול מהקלט – אם הוא גדול מהקלט היא תבדוק את הבן השמאלי, ואם הוא קטן אז את הבן הימני.
באופן זה כאשר הפונקציה מגיעה לעלה היא תחזיר אותו מפני שזה היה הצומת האחרון שנבדק בטרם הגענו לצומת וירטואלי.
סיבוכיות זמן הריצה של הפונקציה היא $O(\log n)$ כי במקרה הגרוע הפונקציה תעבור את המסלול הארוך ביותר מהשורש לעלה, ומאחר שמדובר בעץ AVL מדובר ב-$O(\log n)$.

delete: הפונקציה מוחקת ערך מהעץ, כאשר היא מקבלת את המצביע לערך הרלוונטי, ומחזירה את מספר פעולות האיזון הנדרשות.
אם ה-node המתקבל הוא וירטואלי, הפונקציה מסיימת את הריצה ומחזירה 0.
אחרת, היא קובעת ילד חדש להורה במקום ה-node המסופק.
אם ה-node הוא עלה, הילד החדש המסומן ב-new_node, יהיה node וירטואלי.
אם ל-node יש רק ילד אחד, אז ה-new_node יהיה הילד שלו (פשוט "נדלג" על ה-node).
אחרת, ל-node יש שני ילדים, ואז ה-new_node יהיה ה-successor, הפונקציה תמחק אותו מהעץ עם delete (ותוסיף 1 ל-tree_size, כי אינה מבצעת פה מחיקה בפועל), תוסיף את האיזונים שהתקבלו מה-delete למספר האיזונים שבוצעו ותגדיר את הילדים של ה-new_node להיות הילדים של node בהתאמה.
כעת, יש לנו new_node, הפונקציה תחסיר אחד מ-tree_size.
אם ה-node שהתבקשנו למחוק הוא השורש של העץ אז הפונקציה תגדיר את השורש החדש להיות ה-new_node, את ה-parent שלו להיות None ותחזיר את מספר האיזונים שביצענו.
אחרת, הפונקציה תגדיר את ה-new_node להיות הילד החדש של ההורה של node (אם node היה ילד שמאלי אז new_node יהיה ילד שמאלי ולהיפך), תקרא ל-fix_tree עם ההורה של node והגובה הישן שלו, ותחזיר את מספר האיזונים ועוד מה שנקבל מ-fix_tree.
סיבוכיות זמן הריצה של הפונקציה היא $O(\log n)$ מאחר ואנו מבצעים בהתחלה פעולות בסיבוכיות של $O(1)$, או $O(\log n)$ אם לשורש יש שני ילדים: אפילו הקריאה ל-delete היא לא מסובכת, כי ה-successor בהכרח יהיה שורש עם לכל היותר ילד אחד ולכן לא נקרא שוב ל-delete.
כל שאר הפעולות הן $O(1)$, ו-fix_tree בסיבוכיות של $O(\log n)$ ולכן סה"כ הסיבוכיות היא $O(\log n)$.

fix_tree: הפונקציה מתקנת את איזון העץ, החל מה-node המסופק (כאן ה-node הוא מצביע לצומת כלשהי בעץ). הפונקציה מקבלת init_height, שהוא הגובה של הצומת לפני שינויים שהתבצעו, מאחר ובפונקציות שקוראות ל-fix_tree אנו קודם מבצעים שינויים כמו שינויי ילדים שמשנים את הגובה, ורק בסוף מאזנים את העץ – לכן חשוב שם לאחסן קודם את הגובה ולספק אותו ל-fix_tree כמשתנה.
הפונקציה עולה מה-node למעלה עד לשורש העץ או עד היציאה ממנה: תחילה הפונקציה בודקת אם הבאלנס פקטור הוא ב-$\{-1,0,1\}$ וגם אם הגובה של ה-node לא השתנה, אם כן הפונקציה תצא מהלולאה ותחזיר את כמות האיזונים שבוצעו.
אחרת, היא תמשיך בלולאה, ואם הבאלנס פקטור הוא ב-$\{-2,2\}$ (כלומר צריך לבצע סיבוב) היא מבצעת את הסיבוב הרלוונטי בהתאם: אם הבאלנס פקטור הוא 2- אז סיבוב שמאלה (אם ה-BF של הילד הימני הוא 1 אז מבצעת עליו קודם סיבוב ימינה ומוסיפה 1 למונה האיזונים), ואם הבאלנס פקטור הוא 2 אז סיבוב ימינה (אם ה-BF של הילד הימני הוא 1- אז מבצעת עליו קודם סיבוב שמאלה ומוסיפים 1 למונה האיזונים).
כעת, עדיין בתוך הלולאה, הפונקציה מוסיפה 1 למונה האיזונים (את זה היא מבצעת גם אם לא בוצעו פעולות סיבוב, מאחר והייתה הנחייה לספור תיקוני גובה שאינם מסובבים גם בתור פעולת איזון).
הפונקציה מגדירה את ההורה של node (ששמרנו לפני הסיבובים) בתוך node ואם הוא לא None, מאחסנת את הגובה הישן ב-old_height ומעדכנת את הגובה שלו לפי מקסימום הגובה מבין ילדיו ועוד 1.
הפונקציה חוזרת על הלולאה עד שמגיעה לשורש או לאיבר שלא מצריך סיבובים ולא שינה את הגובה שלו.
לבסוף, הפונקציה מחזירה את מספר פעולות האיזון שבוצעו.
סיבוכיות זמן הריצה של הפונקציה היא לכל היותר $O(\log n)$, מאחר ואנחנו עולים מ-node עלה כלשהו עד לשורש במקרה הגרוע, ומבצעים בכל פעם פעולות בסיבוכיות $O(1)$.

avl_to_array: הפונקציה מחזירה רשימה ממויינת המייצגת את עץ הAVL, רשימה של טאפלים של (key, value) המייצגים את צמתי העץ.
הפונקציה בודקת האם מדובר העץ ריק – במצב זה מחזירה רשימה ריקה, אחרת היא קוראת לפונקציה הרקורסיבית in_order_scan.
סיבוכיות זמן הריצה של הפונקציה זהה לסיבוכיות זמן הריצה של in_order_scan כלומר $O(n)$.

in_order_scan: הפונקציה עוברת על צמתי העץ ומחזירה רשימה של טאפלים של (key, value) המייצגים את צמתי העץ.
הפונקציה היא רקורסיבית כאשר תנאי העצירה שלה הוא בהגעה לצומת וירטואלי.
הפונקציה מייצרת טאפל המייצג את הצומת.
הפונקציה מחזירה רשימה משורשרת של קריאה רקורסיבית של הפונקציה על הבן השמאלי של הצומת, הטאפל המייצג את הצומת הנוכחי וקריאה רקורסיבית של הפונקציה על הבן הימני של הצומת – בכך שומרת על הסדר.
נשים לב שמתבצעת $O(1)$ עבודה בכל צומת – ומפני שאנחנו מבקרים בכל צמתי העץ סיבוכיות זמן הריצה של הפונקציה הוא $O(n)$.

size: הפונקציה מחזירה את גודל העץ עליו היא מופעלת – כלומר מספר הצמתים בעץ.
הפונקציה מבצעת זאת על ידי החזרת השדה tree_size של הצומת.
סיבוכיות זמן הריצה של הפונקציה זו היא $O(1)$.

split: הפונקציה מפצלת את העץ לפי ה-node המסופק.
הפונקציה מתחילה משני משתנים של left_subtree ו-right_subtree - תתי העצים שהבנים של הnode שהתקבל כקלט מהווים השורש של כל אחד.
הפונקציה עולה מה-node עד השורש, כאשר בכל הורה היא בודקת את המפתח שלו לעומת המפתח של הnode שקיבלנו:
אם המפתח שלו קטן יותר מהמפתח של node היא עושה join לתת העץ השמאלי של ההורה עם left_subtree וה-key, value של ההורה.
אם המפתח שלו גדול יותר מהמפתח של node אז להיפך – תת העץ הימני של ההורה עם right_subtree.
בסוף הפונקציה מחזירה רשימה של שני העצים החדשים: [left_subtree, right_subtree], כאשר השמאלי הוא כל המפתחות בעץ שקטנים מ-node והימני הם המפתחות שגדולים ממנו.
נדמה כי סיבוכיות זמן הריצה היא בניתוח נאיבי $O(\log^2 n)$, מאחר ומבצעים $O(\log n)$ איטרציות ובכל אחת עושים join וזה בסיבוכיות $O(\log n)$, אבל בפועל הסיבוכיות של הפונקציה היא $O(\log n)$ כי אם נחשב את הפרשי הגבהים של כל ה-join שעושים (שזו בפועל עלות ה-join), נקבל טור טלסקופי שיצטמצם לנו ל-$O(\log n)$ ולכן זו הסיבוכיות הכוללת.

get_sub_tree: הפונקציה מגדירה איבר חדש מסוג AVLTree שהשורש שלו הוא הצומת אותה הפונקציה מקבלת כקלט.
הפונקציה מייצרת איבר חדש במחלקה AVLTree, היא בודקת האם הצומת שהתקבל כקלט הוא צומת וירטואלי – אם כן אז לא מתבצע דבר ומוחזר האיבר הריק הדיפולטי שיצרנו, אחרת אם מדובר בצומת שאינו וירטואלי – הפונקציה מגדירה את השדה root של העץ שיצרנו להיות הצומת שקיבלנו כקלט, ובכך שמעדכנת את ההורה שלו להיות None הפונקציה מוודאת כי מדובר בשורש של העץ.
מפני שמדובר באוסף פעולות בזמן ריצה $O(1)$ אזי נסיק שסיבוכיות זמן הריצה של הפונקציה הוא $O(1)$.

join: הפונקציה מבצעת מיזוג עם עץ tree2 וצומת עם ערכי key, value כאשר ה-key נמצא בין ערכי self ו-tree2.
הפונקציה מחזירה את הפרש הגבהים של העצים ועוד 1.
תחילה הפונקציה מעדכנת את tree_size להיות סכום הגדלים של העצים ועוד 1.
הנחנו ש-tree2 גדול יותר מ-self, אבל אם זה הפוך אז אנחנו מחליפים בין השורשים (ב-a ו-b).
אם אחד (או שניהם) מהעצים שהתקבלו ריקים, היא מוסיפה את השורש שהתקבל לעץ שאינו ריק (אם קיים), מקבעת אותו להיות self, ומחזירה את הפרש הגבהים.
אחרת, אם הגבהים של העצים שווים אז הפונקציה מגדירה את ה-new_node (שמכיל את ה- key, value שקיבלנו) בתור root של העץ החדש.
אם הגבהים שונים אז הפונקציה יורדת מהשורש של העץ הגבוה שמאלה (אם tree2 גבוה יותר, אחרת יורדת ימינה) עד שמגיעה לגובה שווה לעץ הקטן יותר - שם מגדירה את העץ הקטן יותר להיות הילד השמאלי (או הימני בהתאמה) של ההורה של הצומת עם הגובה השווה לעץ הקטן.
כעת הפונקציה מגדירה את הילדים של new_node להיות תתי העצים הרלוונטיים שמצאנו (שהם

באותו הגובה).
אם היו הבדלי גבהים בין self ל-tree2 אז הפונקציה מבצעת fix_tree על ההורה של new_node, כלומר מתקנת את ה-BF עד השורש, ובכל מקרה מחזירה את הפרשי הגבהים בין העצים.
סיבוכיות זמן הריצה של הפונקציה היא $O(\log n)$ מפני שבמציאת מקום המיזוג אנחנו פוטנציאלית עלולים לרדת את כל גובה העץ הגבוה, וזה מתבצע בסיבוכיות $O(\log n)$, ולאחר מכן יש את הפעלת fix_tree וגם היא בסיבוכיות $O(\log n)$.

get_root: הפונקציה מחזירה את שורש העץ עליו היא מופעלת.
הפונקציה מבצעת זאת על ידי החזרת השדה root של הצומת.
סיבוכיות זמן הריצה של הפונקציה זו היא $O(1)$.

<u>rotate</u>: הפונקציה מבצעת סיבוב של הצומת המתקבל כקלט, ימינה או שמאלה בהתאם לקלט ב-right, הפונקציה מחזירה את השורש החדש של אותו תת עץ.
הפונקציה בודקת האם מדובר בצומת וירטואלי – במידה וכן היא מחזירה את הצומת.
הפונקציה מקבלת את ההורה של הצומת x על ידי קריאה ל-get_parent, ומבצעת את הסיבוב (ימינה או שמאלה בהתאם לקלט) על ידי קריאה לפונקציות get_left, get_right והשמה מחדש באמצעות set_right, set_left בהתאם לכיוון הסיבוב.
הפונקציה מגדירה את ההורה שהשתנה בעקבות הסיבוב, תחילה היא בודקת האם בעקבות הסיבוב השתנה השורש של העץ ואז מגדירה לו הורה None, אחרת הפונקציה מגדירה את ההורה של צומת x ששמרה בתחילת התהליך להיות ההורה של השורש החדש שקיבלנו לתת העץ בעקבות הסיבוב ומחזירה אותו.
נשים לב שהפונקציה מבצעת רק קריאות לפונקציות עם סיבוכיות זמן ריצה של $O(1)$, ולכן סיבוכיות זמן הריצה של הפונקציה הוא $O(1)$.

<u>left_rotate</u>: הפונקציה מבצעת סיבוב שמאלה של הצומת המתקבל כקלט.
הפונקציה מבצעת זאת על ידי קריאה לפונקציה rotate עם הקלט של הצומת שהתקבל.
סיבוכיות זמן הריצה של הפונקציה זהה לסיבוכיות זמן הריצה של rotate ולכן היא $O(1)$.

<u>right_rotate</u>: הפונקציה מבצעת סיבוב ימינה של הצומת המתקבל כקלט.
הפונקציה מבצעת זאת על ידי קריאה לפונקציה rotate עם הקלט של הצומת שהתקבל ו-True בקלט הקובע כי יהיה מדובר בסיבוב ימינה.
סיבוכיות זמן הריצה של הפונקציה זהה לסיבוכיות זמן הריצה של rotate ולכן היא $O(1)$.

חלק ניסוי\תיאורטי

1 . מאחר ואנו יודעים כי עלות של join חסומה על ידי $O(height(T_2) - height(T_1) + 1)$ כלומר – הפרשי הגובה של העץ, ופונקציית ה-join שלנו מחזירה בדיוק את זה (פחות 1), יצרנו רשימת counters שמאחסנת את תוצאת ה-join ועוד 1 בכל פעם שאנו קוראים לה מתוך split.
מתוך הרשימה הזאת, לקחנו את הממוצע והמקסימלי בהתאם.

מספר סידורי i	עלות join ממוצע עבור split אקראי	עלות join מקסימלי עבור split אקראי	עלות join ממוצע עבור split של האיבר מקסימלי בתת העץ השמאלי	עלות join מקסימלי עבור split של איבר מקסימלי בתת העץ השמאלי
1	2.33	4	2.27	11
2	2.62	4	3.11	12
3	2	3	2.42	13
4	2.33	3	2.46	14
5	3	4	2.92	17
6	2.67	4	2.6	17
7	3.17	4	2.87	18
8	3	3	2.76	19
9	2.83	5	2.42	19
10	3.43	4	2.67	21

2. נסמן את עומק הצומת ב-d. נשים לב כי מספר פעולות ה-join הדרוש הינו $O(d)$, מפני שאנחנו עולים מהצומת בו אנחנו מבצעים את הפיצול למעלה עד השורש.
בנוסף, מהנתון, סיבוכיות הפיצול האסימפטוטית היא כעומק הצומת ולכן - $O(d)$.
נשים לב שעלות join ממוצע תהיה עלות הפיצול בכללותו חלקי מספר פעולות ה-join שביצענו, לכן נסיק באופן תיאורטי כי סיבוכיות join ממוצע תהיה $O(1)$.
נשים לב כי ניתוח זה זהה עבור שני המקרים, בין אם לקחנו איבר אקראי או את האיבר המסוים, ותתקבל סיבוכיות זמן ריצה של join ממוצע של $O(1)$.
ניתוח זה מתיישב בהחלט עם התוצאות שקיבלנו: אכן התוצאות שקיבלנו בניסויים מתיישבות עם ניתוח סיבוכיות זה – ראינו שגודל העץ לא משפיע על עלות ה-join הרלוונטי, גם לא בהגדלה משמעותית של מספר הצמתים.

3. נשים לב כי אנו מתייחסים לאיבר המקסימלי בתת העץ השמאלי, כלומר ה-predecessor של השורש.
לכן כאשר מבצעים split בצומת זה, אנו עולים למעלה לאורך כל תת העץ השמאלי – כל פעם מבצעים join בעלות של $O(1)$ - מאחר ואנו עושים join לעצים שהפרש הגבהים שלהם הוא לכל היותר 1 (כולם מתבצעים בתוך תת העץ השמאלי שהוא כמובן עץ AVL) עד שמגיעים לשורש.

בשורש קורה דבר מעניין – אנחנו מבצעים join על עץ ריק (תת העץ הימני של הצומת שפיצלנו), השורש של העץ כולו ותת העץ הימני שלו.
ידוע שסיבוכיות זמן הריצה של פעולה זו היא הפרש הגבהים, ובמצב זה הפרש הגבהים הוא log(n) - לכן זהו ה-join המקסימלי והסיבוכיות שלו היא אכן $O(\log n)$.
ניתוח זה מתיישב בהחלט עם התוצאות שקיבלנו: אכן, ניתן לראות שעלות ה-join המקסימלי עבור ה-split של האיבר המקסימלי בתת העץ השמאלי גדלה בהתאם לגידול במספר הצמתים, ולכל i קיבלנו תוצאה קרובה מאוד ל-$\log_2 1000 \cdot 2^i = O(\log n)$ - כדרוש.