**מבני נתונים – פרוייקט מספר 1 – עץ מאוזן**

מגיש 1:

יעקב אודסר

iakovodesser

209860188

מגיש 2:

איל ספיר

eyalsapir

206405417

להלן תיעוד חיצוני של המחלקות "צומת" ו"עץ" ושל הפונקציות שבהן:

**המחלקה AVLNode**

בכל צומת אנחנו מתחזקים מצביעים לשני הבנים ולאבא ולגובה (המרחק מהעלים). המצביע לאבא דרוש, לדוגמה, עבור פעולות האיזון. הגובה דרוש עבור שמירה על איזון תתי העצים.

פונקציות עזר: הגדרנו שלוש פונקציות עזר שישמשו אותנו בהמשך: is\_real\_node, max\_children\_height, ו-balance. כולן ניתנות לחישוב בהסתמך על ערכי הבנים שלהן, ולכן לא יפגעו בסיבוכיות של הפונקציות שוקראות להן.

**המחלקה AVLTree**

בכל עץ אנחנו מתחזקים מצביעים לשורש, למקסימום (עבור פעולות finger) לגודל העץ (עבור הפונקציה שמחזירה את גודל העץ) ולעלה וירטואלי בגובה מינוס 1 (כדי למנוע מצב של עלה ללא בנים, וככה להקל על ביצוע איזונים).

**פונקציית search:** הפונקציה קוראת לפונקציית עזר שעוברת על העץ בלוגיקה של חיפוש בינארי מהשורש עד המפתח הנחוץ. בחרנו במימוש איטרטיבי לשם הפשטות, ולכן ניתן לראות שמדובר בחיפוש בסיבוכיות O(log*n*). לאחר מכן הגדרנו פונקציית עזר בשם pointer\_only המחזירה את הצומת המבוקש ללא המסלול לשם פשטות המימוש בפונקציות שקוראות ל- search. פונקציית העזר ניגשת לערך הראשון ב-tuple בזמן קבוע.

**פונקציית finger\_search:** הפונקצייה נעזרת במצביע השמור למקסימום ומתקדמת ממנו איטרטיבית עד הצומת הראשון שהמפתח המבוקש k נמצא בתת העץ שלו (במימוש שלנו: פשוט בודקים מתי הגענו למפתח קטן מ-k). הילוך כזה לוקח O(log*k*) פעולות, ולאחר מכן ההגעה למפתח בעזרת חיפוש רגיל לוקחת זמן קבוע בלבד. בפונקציות הבאות נראה איך שומרים על המצביע למקסימום מבלי לפגוע בסיבוכיות הפעולות האחרות.

**פונקציית insert:** קודם כל אנחנו יוצרים צומת חדש. אם העץ ריק, אז רק צריך לעדכן את ערכי המקסימום ואת גודל העץ ל-1. אחרת, צריך להכניס אותו בתור בן של צומת קיים. בשביל זה בנינו את פונקציית העזר insert\_as\_child. בפונקציית העזר אנחנו שומרים שני ערכים: צומת שנצטרך לאזן בסוף התהליך, ומספר פעולות ה-promotion שביצענו. לאחר מכן אנחנו מחלקים למקרים:

ראשית אנחנו מבצעים חיפוש איטרטיבי של הצומת שאליו צריך לחבר את הערך החדש. חיפוש זה עולה O(log*n*). אנחנו מחברים את הערך לצומת שמצאנו ומגדירים אותו כעלה (גובה 0 ועל עלים וירטואליים כבנים), ואז מתחילים לבדוק אם צריך לעשות promotions. בתהליך אנחנו עולים לכל היותר עד השורש, לכן הוא לוקח O(log*n*) פעולות. בכל עלה אנחנו קוראים ל- max\_children\_height שמוצאת בזמן קבוע את הגובה של הילד המקסימלי ומצבעים promotion אם התגלה שהגובה של הילד שווה לגובה של הבן. בנוסף אנחנו קוראים ל- balance שבודקת את הפרש הגבהים בזמן קבוע, ואם מתגלה שיצרנו הפרש גבהים לא תקין, שומרים את הצומת לפונקציית האיזון.

בסוף התהליך אנחנו מעדכנים את הערך המקסימלי של העץ (בזמן קבוע) וקוראים לפונקציית האיזון אם היה צורך. זאת פונקציית עזר בשם rebalance. הפונקצייה ארוכה אבל חוזרת על עצמה אז נסביר רק את הרעיון. אנחנו שומרים מצביעים לצומת שצריך לאזן ולאבא שלו, ומתקדמים בשלבים: בודקים איזה צד "כבד" יותר – נניח במקרה הזה שהימני – ואז בודקים בתוכו: אם תת העץ הימני שלו מאוזן לחלוטין או עם הפרש גבהים 1 כך שהחלק הימני גדול יותר, זה המקרה ה"פשוט" שבו צריך לבצע רק רוטציה שמאלה. אחרת, זה המקרה ה"קשה" שבו צריך לבצע רוטציה ימינה ולאחר מכן רוטציה שמאלה. בסוף כל הפעולות אנחנו מעדכנים את הגבהים בפעולות שעולות זמן קבוע בלבד, וכאמור המקרה השמאלי סימטרי.

לסיכום: בתהליך ההכנסה לעץ אנחנו מבצעים חיפוש בסיבוכיות O(log*n*), לאחר מכן מחפשים צמתים לאזן ב- O(log*n*) ולבסוף מאזנים צומת אחד בלבד בזמן קבוע, ולכן פעולת החיפוש כולה לוקחת בסך הכול O(log*n*).

**פונקציית finger\_insert:** באופן אנלוגי לחלוטין ל- finger\_search אנחנו מתקדמים איטרטיבית מהמקסימום עד הצומת הראשון שהערך שאנחנו מעוניינים להכניס קטן ממנו, ואז קוראים ל- insert הרגילה, כמובן תוך התחשבות במקרה הקצה של עץ ריק. לפיכך הסיבוכיות תהיה O(log*k*) כמו finger\_search.

**פונקציית delete:** אנחנו מחפשים את הצומת, נעזרים במצביע לאבא, מחלקים למקרים ובסוף התהליך בודקים אם צריך לאזן את העץ. במקרה שהצומת שרצינו למחוק הוא עלה – פשוט מנתקים אותו מהעץ ומחברים את אבא לעלה הווירטואלי. אם לעלה יש רק בן אחד, אנחנו נעזרים בפונקיית עזר פשוטה בשם switcheroo בשביל להחליף בין הצומת לבן שלו ולמחוק את הצומת כעלה.

האפשרות האחרונה היא שלצומת יש שני בנים. במקרה הזה אנחנו נעזרים בפונקציית עזר בשם get\_successor. הפונקצייה מבצאת בצורה איטרטיבית את האלגוריתם שלמדנו בהרצאה למציאת יורש במקרה הגרוע צריך לעלות עד השורש, ולכן היא בסיבוכיות O(log*n*). אחרי שמצאנו את היורש אנחנו משנים את ערכי הצומת שלנו לערכי היורש ומוחקים את היורש.

בסך הכול, כמו שהוכחנו בכיתה, פעולת המחיקה דורשת במקרה הגרוע למצוא את הערך, למצוא יורש ולאזן את העץ – כולן פעולות שחסומות על ידי גובה העץ, ולכן עלות המחיקה כולה היא O(log*n*).

**פונקציית join:** הגדרנו שתי פוקנציות עזר, key\_size\_matters ו- height\_mattersשבודקות בזמן קבוע מי העץ בעל המפתחות הגדולים יותר ומי העץ הגבוה היותר. אנחנו מעדכנים את הגודל והערך המקסימלי של העץ בזמן קבוע ואז מחלקים למקרים: במקרה ששני העצים בגובה דומה, אנחנו מכניסים את שניהם כבנים של ערך הביניים ומגדירים את ערך הביניים כשורש. במקרה הקשה, אנחנו מטיילים למטה מהשורש של העץ הגבוה (מצד ימין או שמאל לפי גודל הערכים) ועוצרים בנקודה שבה הגבהים כמעט שווים. אנחנו מחברים את ערך הביניים כבן של הצומת שבו עצרנו, וכתתי עצים של ערך הביניים את העץ הקטן ואת מה שניתקנו מהצומת שאליו חיברנו את צומת הביניים.

לאחר התהליך אנחנו מעדכנים גבהים בזמן גבוה ובודקים אם צריך לבצע איזון. בשונה מהפעולות האחרות, במקרה הנ"ל ייתכן שנצטרך לבצע שני איזונים ולא רק אחד, ולכן אנחנו לא מגבילים את מספר האיזונים, אבל עדיין זה יוצא זמן קבוע, ולכן הסיבוכיות עולה לנו בסך הכול כמו שעולה המעבר מהשורש של העץ הגדול עד נקודת החיבור, כלומר O(heightT1 – heightT2 + 1).

**פונקציית split:** הגדרנו פונקצייה רקורסיבית rec\_split שמקבלת צומת ומפתח ומחזירה את העץ המפוצל. אנחנו מזינים את השורש לפונקציה הרקורסיבית ובודקים אם הוא שווה למפתח ההפרדה. אם לא, אנחנו הולכים שמאלה או ימינה לפי הערך, ומחפשים את מפתח ההפרדה רקורסיבית (לכל היותר log*n* איטרציות). כשמצאנו את המפתח, אנחנו מנתקים את הבנים שלו ממנו,