## מבנה הנתונים של התוכנה

מבנה הנתונים שלנו יכיל טרולים ופוסטים.

אנו רוצים להתאים את מבנה הנתונים שלנו כדי שיעבוד בסיבוכיות הנדרשת של התרגיל ושל הפונקציות. על כן, החלטנו להשתמש במבנה של עץ AVL אותו ראינו בהרצאות ואותו נממש בתרגיל זה, לצורך החזקת הנתונים.

מבנה הנתונים ייבנה באופן הבא:

- עץ AVL אשר יכיל את הפוסטים שנמצאים ברגע מסוים במערכת. המיון של נתוני העץ יעשה לפי postId. נוודא שהפעולות שלנו לא פוגעות במיון זה.
- עץ AVL של טרולים שנמצאים במערכת, ממוין לפי trollID.
- עץ AVL אשר יכיל את הפוסטים במערכת. הפעם נמיין את הפוסטים לפי כמות הלייקים שלהם. עבור שני פוסטים בעלי אותה כמות לייקים, הגדול יותר הוא זה בעל המזהה הקטן יותר.
- עבור כל טרול, נחזיק עץ AVL המכיל את הפוסטים שפרסם הטרול. עץ זה יהיה ממוין גם לפי כמות לייקים (ועבור פוסטים בעלי אותה כמות לייקים, ההשוואה תתבצע לפי ID, כמתואר לעיל).

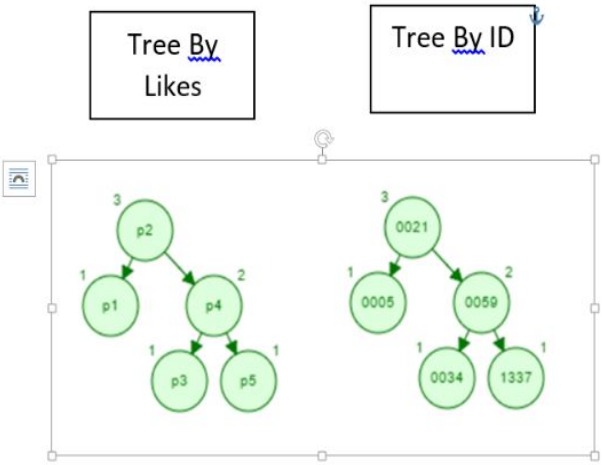
אינווריאנטות:

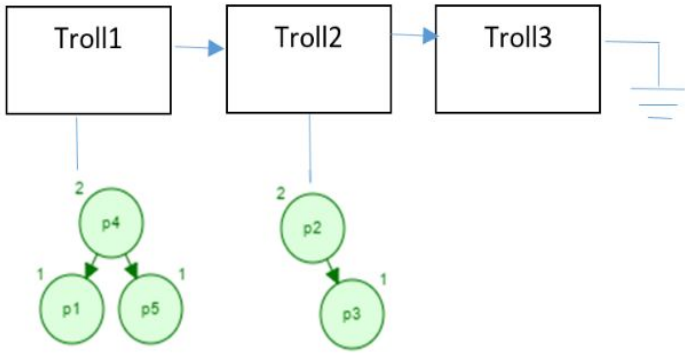
- עץ הפוסטים צריך להישאר ממוין לפי ID ולכן על כל שינוי של נתוני הפוסטים (כולל הכנסה או הוצאה), יש לבדוק ששמורה זו אינה נפגעת.
- עץ הפוסטים שממוין לפי לייקים (כולל העצים אצל הטרולים) צריך להישאר כך לצורך גישה לפי כמות לייקים. על כן, יש לשמור על המיון תוך שמירה על דרישות הסיבוכיות.
- משתני הtopPost, הן אצל כל טרול והן במערכת צריכים להיות מעודכנים בכל רגע נתון. על כן, כל שינוי (הוספה, הוצאה או שינוי נתונים של כל פוסטים), יש לבדוק ששמורה זו אינה נפגעת, גם בשדה כמות הלייקים וגם בשדה הID.

הערות:

- נשמור את מספר הפוסטים שבמערכת בכל עת.
- נשמור במערכת את פרטי הפוסט הפופולרי היותר – בעל כמות לייקים מקסימלית (את כמות הלייקים ואת המזהה שלו).
- לכל טרול נשמור את פרטי הפוסט הפופולרי ביותר שפרסם (את כמות הליקיים ואת המזהה שלו).
- לכל טרול נשמור גם את מספר הפוסטים שמפורסמים אצלו בוול בכל עת.
- ההשוואה בין המפתחות של הצמתים בעצים שממוינים לפי לייקים ולפי מזהה יעשו לפי אופרטור השוואה או לפי Comparer.

תיאור מבנה הנתונים באמצעות איור:




כאשר כל טרול הוא צומת בעץ של הטרולים במערכת.

<u>הוכחת סיבוכיות של פונקציות של מבנה הנתונים:</u>

עבור דרישות הסיבוכיות בתרגיל, הסימונים הבאים מתקיימים:

- k – מספר הטרולים במערכת ברגע ביצוע הפעולה.
- n – מספר הפוסטים במערכת ברגע ביצוע הפעולה.
- $n_{trollID}$ – מספר הפוסטים במערכת ששייכים לטרול בעל המזהה trollID ברגע ביצוע הפעולה.
- עבור הכנסה/חיפוש של טרול לעץ, הפעולות נעשות בזמן O(k=O(logk).

1. <u>Init:</u> דרישת סיבוכיות זמן: O(1).
אתחול מבני הנתונים שלנו נעשה במספר קבוע של פעולות כיוון שיצירת עצי AVL ריקים דורשות מספר קבוע של פעולות. לכן, סה"כ, מספר הפעולות שנעשות בפונקציה זו הוא מסדר גודל של O(1).

2. AddTroll: דרישת סיבוכיות זמן: O(k).
הפונקציה כוללת יצירת Troll חדש – אתחול מספר קבוע של שדות ויצירת עץ AVL ריק – מספר פעולות בסדר גודל של O(1). בנוסף, יש לבדוק שלא קיים טרול עם ID כמו שהתקבל כפרמטר. זה דורש סיור על עץ הטרולים ולכן נעשה בO(k). לבסוף, הכנסת הטרול לעץ בהינתן שאכן לא קיים הטרול במערכת, נעשית באמצעות הפונקציה "insert" ולכן נעשית ב-O(k). סה"כ, מספר הפעולות שנעשות בפונקציה זו הוא מסדר גודל של O(k).

3. PublishPost: דרישת סיבוכיות זמן: O(log(n)+k).
הפעולות הנדרשות לפונקציה זו הן יצירת הפוסט עם ערכי השדות שהתקבלו – אתחול של מספר קבוע של שדות נעשה ב-O(1). בדיקה אם קיים הפוסט במערכת נעשה בחיפוש בינארי על העץ במערכת – נעשה בO(log(n)). בדיקה אם לא קיים טרול עם המזהה שהתקבל – O(k). מציאת הטרול הנ"ל – גם כן נעשית ב-O(k).
הכנסת הפוסט לעץ של הטרול נעשה ב-O(log($n_{trollID}$=O(log(n)). הכנסת הפוסט לעצים שמחוץ לעץ הטרולים נעשים ב-O(log(n)) כל אחד.
יש לעדכן את משתני ה-TopPost במידת הצורך, הן של הטרול והן של כלל המערכת. זה נעשה ב-O(1). יש לציין כי ההכנסה נעשית באופן כזה ששומר על שמורות מבנה הנתונים וההשוואות שנעשות בעת ההכנסה שומרות על מיון העצים. סה"כ, מספר הפעולות שנעשות בפונקציה זו הוא מסדר גודל של O(log(n)+k).

4. DeletePost: דרישת סיבוכיות זמן: O(log(n)+k).
הפעלות הנדרשות לביצוע פונקציה זו הן חיפוש הפוסט בעץ הפוסטים הכללי, ואם קיים, שמירת המזהה של הטרול ומחיקתו מעץ זה – זה נעשה ב-O(log(n)) עבור המחיקה ועבור החיפוש ו-O(1) עבור חילוץ הID של הטרול. יש למחוק את הפוסט גם מהעץ הממוין לפי לייקים – פעולה זו גם לוקחת O(log(n)). חיפוש הטרול לפי המזהה שחילצנו קודם לכן נעשה ב-O(k). מחיקת הפוסט מהעץ של הטרול נעשה ב-O(log($n_{trollID}$=O(log(n)).
ההוצאות מהעצים שומרות על המיון לפי הגדרת עץ AVL וככה שומרות על שמורות מבנה הנתונים שלנו. יש גם לעדכן את משתני ה-TopPost שלנו. במקרה הגרוע ביותר, אם צריך להחליף את נתונים אלו, מספר הפעולות שקול לפעולה של גישה לצומת הימני ביותר בעץ (של הטרול ואו של כלל המערכת), כדי לגשת לפוסט הטוב ביותר החדש. זה נעשה ב-O(log(n)). סה"כ, מספר הפעולות שנעשות בפונקציה זו הוא מסדר גודל של O(log(n)+k).

5. FeedTroll: דרישת סיבוכיות זמן: O(log(n)+k).
הפעולות הנדרשות לביצוע פונקציה זו הן חיפוש הפוסט בעץ הפוסטים הכללי – O(log(n)). יצירת פוסט חדש עם פרטים מעודכנים וחילוץ המזהה של הטרול שפרסם אותו נעשים ב-O(1). שינוי כמות הלייקים של הפוסט בעץ זה גם לוקחת O(1). עץ זה נשאר ממוין כי אינו ממוין לפי לייקים. יש כעת להוציא את הפוסט מהעץ הממוין לפי לייקים של המערכת ולהכניס את הפוסט המעודכן שיצרנו קודם – שתי הפעולות נעשות ב-O(log(n)). חיפוש הטרול לפי המזהה שחילצנו קודם לכן נעשה ב-O(k). מחיקת הפוסט מהעץ של הטרול והכנסת הפוסט עם הנתונים המעודכנים שיצרנו קודם גם לעץ זה נעשות ב-O(log($n_{trollID}$))(O(log(n))).
עצים אלו הממוינים לפי לייקים נשארים ממוינים בתום הפעולה כיוון שהשתמשנו בהוצאה והכנסה של עץ AVL שלפי הגדרה שומרים על המיון. בנוסף, יש גם לעדכן את משתני ה-TopPost שלנו. אם הפוסט החדש גבוה בכמות הלייקים שלו מזה ששמור במערכת או אצל הטרול, יש לשנות את פרטיהם בהתאם. זה דורש O(1) פעולות. סה"כ, מספר הפעולות שנעשות בפונקציה זו הוא מסדר גודל של O(log(n)+k).

6. GetTopPost: דרישת סיבוכיות זמן: O(1 ← trollID<0). אחרת אם O(k :trollID>0).
נפריד למקרים לפי trollID:
אם trollID<0 אז אנו צריכים להחזיר את מזהה הפוסט הטוב ביותר בכל המערכת. כלומר, יש לגשת לשדות הTopPost של המערכת. פעולה זאת נעשית ב-O(1).
אם trollID>0 יש לחפש את הטרול בעץ הטרולים. פעולה זו לוקחת O(k). בין אם הטרול קיים במערכת ובין אם לא, שאר הפעולות שנותרו הן בסדר גודל של O(1): אם הטרול אינו קיים, יש להחזיר ערך שגיאה כשלון. אם הוא אכן קיים, יש להחזיר את פרטי TopPost שלו. בפונקציה זו לא משתנים נתוני העצים במערכת ולכן המיון לא נפגע והשמורות שלנו נשארות. סה"כ, מספר הפעולות שנעשות בפונקציה זו הוא מסדר גודל של O(k) אם trollID>0 או O(1) אם trollID<0.

7. GetAllPostsByLikes: דרישת סיבוכיות זמן: אם O(n ← trollID<0). אחרת אם O($n_{trollID}$+k :trollID>0).
נפריד למקרים לפי trollID.
אם trollID<0 אז יש להחזיר את כל הפוסטים במערכת ממוינים לפי מספר לייקים. זה דורש מעבר על כל הפוסטים בעץ שלנו אשר ממוין לפי לייקים בגישת in-order הפוכה (כלומר ניגש קודם לתת העץ הימני, אחר כך לצומת האב ולאחר מכן לתת העץ השמאלי). פעולה זו אכן נעשית ב-O(n).
אם trolld>0 יש לחפש את הטרול בעץ הטרול, פעולה שלוקחת O(k). אם הטרול אינו קיים, יש להחזיר ערך שגיאה, לכן O(k)=O($n_{trollID}$+k).
אם הטרול אכן קיים, יש לסרוק את עץ הפוסטים שלו באותה שיטה (In-order הפוך) ולהכניס למערך. גם פה העץ ממוין לפי לייקים והסדר יהיה נכון. כיוון שיש ברשות טרול זה (ובתוך העץ שלו) $n_{trollID}$ פוסטים, פעולה זו נעשית בO($n_{trollID}$). לכן, סה"כ עבור מעבר זה, O($n_{trollID}$+k). העצים שלנו לא נפגעים מפעולות פונקציה זו ולכן המיון נשמר בהם. סה"כ, מספר הפעולות שנעשות בפונקציה זו הוא מסדר גודל של O($n_{trollID}$+k) אם trollID>0 או O(n) אם trollID<0.

8. UpdateLikes: דרישת סיבוכיות זמן: O(n+k).
אנו רוצים לעדכן את מידע הפוסטים המתאימים לפרמטר שקיבלנו בשלושת העצים בהם הם נמצאים. נרצה לעשות זאת ולשמור על מיון העצים, כל זאת תוך שמירת על הסיבוכיות הנדרשת. כיוון שאין הגבלה על סיבוכיות מקום, נחשוב על פתרון לא טרויאלי (O(nlog(n))) אשר יכיל שימוש ביותר זכרון. עבור העץ הממוין לפי ID – אין בעיה – נעבור על הצמתים בגישת in-order ונשנה את כמות הלייקים בהתאם תוך בדיקה ששמורת הTopPost נשמרת (כלומר, במידת הצורך, לשנות את שדות אלו). כיוון שמיון העץ לא תלוי בלייקים,

העץ נשאר ממוין. שינוי זה לוקחת O(n). עבור העצים הממוינים לפי לייקים (החיצוני ואלו של הטרולים), נחשוב על האלגוריתם הבא:

- ניצור שני מערכים של פוסטים, כל אחד בגודל n (או A, $n_{trollID}$) ו-B.
- נעבור על העץ בגישת in-order ונכניס כל אחד מהפוסטים שאותם יש לשנות למערך A וכל אחד מהפוסטים שלא צריך לשנות למערך B. בנוסף, נספור כמה פוסטים אנו מכניסים לכל אחד מהמערכים. שני המערכים ממוינים בגלל שיטת האיטרציה In-order ומהסיבה שהכנסנו את הפוסטים לפי סדר למערכים. עד כה, הסיבוכיות היא O(n).
- עבור כל פוסט במערך A, נעדכן את כמות הלייקים שלו בתוך המערך. המערך ישאר ממוין לפי הקריטריון כיוון שהכפלנו את כל הלייקים של כל הפוסטים באותו פקטור ולכן הגדלנו (או הקטנו) את כולם באותו יחס.
- נבצע שילוב של המערכים באמצעות באלגוריתם merge-sort לכדי מערך אחד בגודל n אותו ראינו בקורס מבוא, וראינו שפועל בסיבוכיות של O(n).
- המערך החדש ממוין לפי הסדר שנרצה ש"ייבנה" העץ שלנו.
- נכניס את הפוסטים לעץ בגישת In-order, כלומר, נסרוק את העץ בגישה זו ועבור כל צומת "נחליף" את הערכים שבתוך הצומת באלו של הפוסט ה"נוכחי" לפי האיטרציה במערך. ההחלפה לוקחת O(1) לכל צומת, וסה"כ, ההכנסה לוקחת O(n). כיוון שאכן אנו מכניסים את הפוסטים לפי סדר המערך ולפי גישה in-order, העץ שלנו יישאר ממוין.
- סה"כ, זמן הריצה של האלגוריתם הוא מסדר גודל של O(n).

נפעיל את אלגוריתם זה על העץ הכללי הממוין לפי לייקים ועל כל העצים הרלוונטיים בעץ הטרולים. זמן הריצה עבור העצים של הטרולים: $O(\sum n_{trollID}) = O(n)$ . צריכים להחשיב את האיטרציה לגבי העץ ולכן סה"כ O(n+k).

יש לציין כי כבר בעץ הממוין לפי ID הבטחנו את יציבות השמורה של משתני הTopPost ולכן היא נשמרת.

סה"כ, מספר הפעולות שנעשות בפונקציה זו הוא מסדר גודל שלO(n+k)=O(n).

9. Quit: דרישת סיבוכיות זמן: O(n+k).
דרישת סיבוכיות מקום: O(n+k).

יש למחוק את כל הנתונים במערכת. נעבור על שני העצים החיצוניים בגישת In-order ונהרוס את הצמתים ואת המידע. עד כה, O(n) עבור סיבוכיות זמן ועבור סיבוכיות מקום. נסרוק את העץ של הטרולים ועבור על טרול, נמחק את עץ הפוטסים שלו ונמחק את המידע שהוא מכיל, בנוסף נהרוס את הטרול עצמו ונעבור לטרול הבא בעץ. אנו עושים זאת לכל טרול בעץ, שלו יש מספר מסוים של פוסטים שסה"כ נסכמים ל-n. על כן, הריסת הפוסטים בעץ הטרולים נעשית ב O(n)עבור סיבוכיות זמן ועבור סיבוכיות מקום, והריסת הטרולים והמידע בהם בכל העץ נעשית בO(k) עבור סיבוכיות זמן ועבור סיבוכיות מקום. על כן, נעשית מחיקת עץ הטרולים והמידע שלה בסיבוכיות מקום וזמן של O(n+k). סה"כ, מספר הפעולות שנעשות בפונקציה זו הוא מסדר גודל של O(n+k) והגישה החוזרת למחסנית היא מסדר גודל של O(n+k).