מבני נתונים
231218

תרגיל רטוב 1- חלק יבש

הוגש ע"י:

ניר אברמוביץ 206562589

אריאל ברוק 318268265

תיאור כללי של מבנה הנתונים:

מבנה הנתונים שלנו מכיל 4 עצי **AVL** ראשיים ועץ **AVL** נוסף לכל קבוצה של שחקנים במערכת, כאשר בכל צומת בעץ יש 2 סוגי מידע: מפתח, ומידע. הצמתים בעץ מסודרים על פי האופרטורים <, > של המפתחות.

- **players**: עץ של כלל השחקנים במערכת שממויין לפי השלבים של השחקנים בסדר עולה והמספר המזהה של השחקנים בסדר יורד, בו בכל חולייה יש את המידע של השחקן שמכיל: את השחקן עצמו ומצביע לקבוצה אליה הוא משתייך. המפתח הוא זוג סדור כאשר השלב הוא הערך הראשון של הזוג, והמזהה הוא הזוג השני. בהמשך מידע נוסף על מבנה הזוג הסדור.
- **levels**: עץ שחקנים שממויין לפי המספר המזהה של השחקנים, ובו יש את השלב של כל שחקן.
- **groups**: עץ קבוצות שממויין לפי המספר המזהה של הקבוצות, בכל קבוצה יש עץ **players** נוסף המכיל רק את השחקנים הנמצאים באותה קבוצה, ממויין כמו העץ **players** הכללי, ובכל צומת מצביע למידע של השחקן שנמצא בעץ **players** הכללי.
- **Non_empty_groups**: עץ של הקבוצות שבהן יש לפחות שחקן אחד, ממויין לפי המספר המזהה של הקבוצות, ומכיל מצביעים לקבוצות הלא ריקות (שבהן יש לפחות שחקן אחד).

תיאור המחלקות:

- **PlayersManager-** המחלקה שמכילה את כלל המידע: מכילה את כל העצים שתוארו למעלה ובנוסף מחזיקה מצביע לשחקן שנמצא בשלב הכי גבוה במשחק.
- **Group-** מחלקה המתארת קבוצה ומכילה: את המזהה של הקבוצה, עץ **AVL** של שחקני הקבוצה ומצביע לשחקן שנמצא המקסימלי ע"פ המיון שהוגדר למעלה.
- **Player** – מחלקה המתארת שחקן ומכילה: את המספר המזהה שלו ואת השלב בו השחקן נמצא.
- **PlayerData** – מחלקה המכילה מידע מסוג **Player**, ובנוסף מחזיקה מצביע לקבוצה אליה השחקן משתייך.
- **Tuple**2-מחלקה המכילה זוג עצמים (שיכולים להיות גם מסוגים שונים) ואופרטורים <,>,=.
  - האופרטור = מוגדר להחזיר **true** אמ"מ בשני זוגות סדורים האיבר הראשון בזוג הראשון שווה לאיבר השני בזוג השני ואם האיבר השני בזוג הראשון שווה לאיבר השני בזוג השני.
  - האופרטור **t1>t2** מוגדר להחזיר **true** אמ"מ כאשר האיבר הראשון ב**t1** **<u>גדול</u>** מהאיבר הראשון ב**t2** ואם הם אז כאשר האיבר **<u>השני</u>** ב**t1** **קטן** מהאיבר השני ב**t2**.
  - האופרטור **t1<t2** מחזיר **true** אמ"מ **t1,t2** לא שווים וגם לא מתקיים **t1>t2**.
- **AVLTree** – עץ מאוזן ע"פ הגדרת עץ **AVL**.

מימוש הפעולות הנדרשות:

1. $void^{*}\ init()$- הפונקציה יוצרת מבנה נתונים ממחלקת **PlayersManager** על ידי קריאה לבנאי הדיפולטיבי. כלומר נוצרים 4 עצים ריקים ועוד מצביע מאותחל. הפונקציה מחזירה מצביע למבנה הנתונים החדש.

   <u>סיבוכיות:</u> הפונקציה מבצעת מספר קבוע של פעולות ולכן הסיבוכיות $O(1)$ כנדרש.

2. $StatusType\ AddGroup(void^{*}\ DS,\ int\ GroupID)$- ראשית הפונקציה בודקת אם הערכים שהתקבלו תקינים, ואם הקבוצה קיימת כבר במבנה הנתונים. אם כן- מחזירה את הערכים המתאימים. אם לא- הפונקציה יוצרת קבוצה חדשה וקוראת לפונקציה ההכנסה של העץ כאשר המפתח או המספר המזהה של הקבוצה והמידע הוא הקבוצה עצמה, ע"פ האלגוריתם שתואר בהרצאה.

סיבוכיות: במקרה הגרוע הפונקציה מבצעת פעולת חיפוש על עץ AVL כדי לבדוק האם הקבוצה כבר קיימת ופעולת הכנסה של הקבוצה אם היא לא קיימת ולכן כפי שנלמד בהרצאה, מבצעת מספר פעולות בסדר גודל של $2 \cdot \log k$. כלומר, הסיבוכיות היא $\boldsymbol{O(\log k)}$ כנדרש.

3. $\boldsymbol{StatusType\ AddPlayer(void^{*}\ DS, int\ PlayerID, int\ GroupID, int\ level)}$-

הפונקציה בודקת אם הערכים שהתקבלו תקינים, אם הקבוצה לא קיימת במערכת ואם השחקן קיים כבר במערכת ומחזירה את הערכים המתאימים אם כן.

אם כל הערכים תקינים, הפונקציה מוסיפה את מזהה השחקן לעץ ה*levels*, מחפשת את הקבוצה המתאימה בעץ ה*groups* ומוסיפה את השחקן לעץ ה*players* של אותה קבוצה עם מצביע ל*nullptr* ו מוסיפה את השחקן לעץ ה*players* הכללי. לאחר ההוספה לעץ ה*players* הכללי מעדכנים את המצביע בעץ השחקנים של הקבוצה. בנוסף, מעדכנת ב**PlayersManager** ובקבוצה אליה השחקן נוסף את המצביע לשחקן שהמפתח שלו הוא המקסימלי בעץ (השחקנים הכללי והשחקנים בקבוצה, בהתאמה). אם השחקן הוא הראשון בקבוצה, מצביע לקבוצה יתווסף לעץ *non_empty_groups*.

סיבוכיות: ההוספה של השחקן מתבצעת על ידי בדיקה של אי הימצאות השחקן בעץ השחקנים הכללי, והימצאות הקבוצה בעץ הקבוצות. לאחר מכן, מציאת הקבוצה המתאימה בעץ הקבוצות, הוספה לעץ השחקנים בקבוצה, הוספה לעץ ה*levels* ולעץ ה*players*, בהם יש *n* שחקנים, וחיפוש השחקן המקסימלי בקבוצה ובעץ השחקנים הכללי. סדר הגודל של מספר כל הפעולות הוא:

$existence\ checks = \log k + \log n_{total}$, $inserts = 2\log k + \log n_{group} + 2\log n_{total}$, $search = \log n_{group} + \log n_{total}$, $total = 3\log k + 4\log n_{total} + 2\log n_{group}$. מספר השחקנים בקבוצה קטן או שווה למספר השחקנים הכולל ולכן הסיבוכיות שמתקבלת היא:

$\boldsymbol{O(\log k + \log n_{total})}$.

4. $\boldsymbol{StatusType\ RemovePlayer(void^{*}\ DS, int\ PlayerID)}$- גם הפעם הפונקציה בודקת אם הערכים תקינים ומחזירה ערכי שגיאה במידת הצורך.

אם הקלט תקין, והשחקן נמצא במערכת, הפונקציה מוצאת את השחקן בעץ ה*levels* לפי המזהה שלו ולוקחת את ה*level*, מוצאת את השחקן בעץ ה*players* וכך ניגשת לקבוצה אליה הוא משתייך (כאמור, בחוליה של עץ ה*players* יש מצביע לקבוצה) ומסירה אותו מעץ השחקנים של הקבוצה. לבסוף, מסירה אותו מעץ השחקנים ומעץ ה*levels* ומעדכנת את השחקן שנמצא בשלב הכי גבוה הן בקבוצה והן בעץ השחקנים הכללי. אם לאחר מחיקת השחקן יש בקבוצה 0 שחקנים, הקבוצה תימחק מהעץ *non_empty_groups*.

סיבוכיות: בודקים שהשחקן קיים בעץ *level* בסדר גודל של $\log n_{total}$. לאחר מכן החיפוש של השחקן בעצי ה *levels* וה*players* (הכללי) מתבצע בסדר גודל של $\log n_{total}$ פעולות. מחיקת השחקן בעץ השחקנים בתוך הקבוצה מתבצע בסדר גודל של $\log n_{group}$ פעולות. מציאת הקבוצה מתבצעת על ידי מספר סופי של פעולות בזכות המצביע לקבוצה שנשמר בעץ ה*players*. לאחר מכן תתבצע מחיקת השחקן מעץ השחקנים הכללי ומעץ השלבים. עדכון השחקן המקסימלי ידרוש מספר פעולות כעומק העץ- כלומר, סדר גודל של $\log n_{group}$

פעולות או $\log n_{total}$ בעץ הכללי. מחיקת הקבוצה מעץ ה*non_empty_groups* תיקח $\log n_{total}$ פעולות (יש לכל היותר *n* קבוצות בעץ כאשר יש שחקן בכל קבוצה) ולכן סדר הגודל של מספר כל הפעולות הוא :

$$existence\ check: \log n_{total}\ , search: 2\log n_{total}\ , deletion: \log n_{group} + 3\log n_{total}\ , check\ new\ max: \log n_{group} + \log n_{total}$$

מספר השחקנים בקבוצה קטן או שווה למספר השחקנים הכולל ולכן הסיבוכיות היא: $\boldsymbol{O(\log n)}$

5. $\boldsymbol{StatusType\ ReplaceGroup(void^{*}\ DS, int\ GroupID, int\ ReplacmentID)}$-

הפונקציה בודקת אם הקלט תקין ואם הקבוצות קיימות במערכת. אם הקלט תקין, הפונקציה יוצרת לכל קבוצה מערך של השחקנים באותה קבוצה ממוינים לפי השלב שלהם על ידי ריצה *in-order* על עצי השחקנים של כל אחת מהקבוצות. כעת, היא יוצרת מערך ממוין של כלל השחקנים על ידי *merge*, ומהמערך החדש יוצרת עץ חדש עם השחקנים של שתי הקבוצות. שתי הקבוצות יימחקו מעץ הקבוצות, ונוסיף מחדש את הקבוצה אליה אנחנו מעבירים את השחקנים כך שעץ השחקנים שלה יהיה העץ המעודכן. השחקן בשלב הגבוה ביותר מעודכן בקבוצה.

<u>סיבוכיות:</u> חיפוש הקבוצות בעץ הקבוצות על מנת לבדוק אם הן קיימות יתבצע בסדר גודל של $2 \cdot \log k$ פעולות, ריצה על עצי השחקנים ויצירת מערך תתבצע בסדר גודל של $n_{group} + n_{replacement}$ פעולות משום שריצה *in-order* על איברי העץ דורשת מספר פעולות בסדר גודל של מספר הצמתים בעץ.

Merge של המערכים ידרוש מספר השוואות כמספר השחקנים המינימלי בשני המערכים ועוד השמה של מספר האיברים הנותרים ולכן גם הוא יתבצע בסדר גודל של $n_{group} + n_{replacement}$ פעולות. יצירת עץ מהמערך המאוחד תדרוש מעבר על כל אחד מהתאים במערך ושיבוצו בעץ (המערך כבר ממוין ולכן ניתן לשבץ ממרכז המערך לצדדים ואין צורך בגלגולים/ חיפוש המקום המתאים) ולכן יצירת העץ תתבצע בסדר גודל של $n_{group} + n_{replacement}$ פעולות. עדכון השחקן המקסימלי מתבצע תוך כדי יצירת העץ החדש ולכן דורש מספר קבוע של פעולות.

בסך הכל נקבל שהסיבוכיות היא: $\boldsymbol{O(\log k + n_{group} + n_{replacement})}$

6. $\boldsymbol{StatusType\ IncreaseLevel(void^{*}\ DS, int\ PlayerId, int\ LevelIncrease)}$-

הפונקציה בודקת את תקינות הקלט ובודקת האם השחקן קיים במערכת, לאחר מכן מעדכנת את השלב של השחקן בעץ השלבים, מוחקת אותו מהעצים בהם הוא מופיע (רק אלו הממוינים לפי השלב), ומוסיפה אותו מחדש עם השלב המעודכן. בנוסף היא מעדכנת את השחקן המקסימלי בקבוצה אליה השחקן שייך ובעץ השחקנים הכללי.

<u>סיבוכיות:</u> בדיקה האם השחקן קיים דורשת סדר גודל של $\log n_{total}$ פעולות,

חיפוש השחקן בעץ *levels* לטובת גישה ל*data* שלו דורשת גם כן סדר גודל של $\log n_{total}$ פעולות.

עדכון השלב של השחקן דורש מספר קבוע של פעולות, מחיקת השחקן מהעץ *players* דורשת סדר גודל של $\log n_{total}$ פעולות, והוספתו מחדש לעץ ובדיקת השחקן המקסימלי גם כן דורשות סדר גודל של $\log n_{total}$ פעולות. באופן דומה מחיקה והוספה של השחקן לעץ השחקנים של הקבוצה דורשת סדר גודל של

$\log n_{group}$ פעולות (קיים מצביע לקבוצה של השחקן בעץ השחקנים הכללי). עדכון השחקן המקסימלי של הקבוצה דורש סדר גודל של $\log n_{groups}$.

המספר הכולל של הפעולות הוא מסדר גודל של $5 \cdot \log n_{total} + 3 \cdot \log n_{group}$.

מספר השחקנים בכל קבוצה קטן או שווה למספר השחקנים הכולל ולכן הסיבוכיות שמתקבלת היא: $\boldsymbol{O(\log n)}$.

7. $\boldsymbol{StatusType\ GetHighestLevel(void^{*}\ DS, int\ GroupID, int * PlayerID)}$-

הפונקציה בודקת את תקינות הקלט, ואז בודקת אם הערך של *GroupID* חיובי או שלילי.

אם חיובי- הפונקציה בודקת אם הקבוצה קיימת ולאחר מכן אם יש שחקנים בקבוצה ומחזירה את המספר המזהה של השחקן המקסימלי לפי המצביע השמור במחלקת הקבוצה.

אם שלילי, הפונקציה בודקת אם יש שחקנים במערכת ואם כן מחזירה את המספר המזהה של השחקן המקסימלי לפי המצביע השמור במחלקת המערכת.

<u>סיבוכיות-</u> במערכת ניהול השחקנים יש מצביע לשחקן שנמצא בשלב הגבוה ביותר מבין כל השחקנים במערכת, באופן דומה גם בקבוצה יש מצביע לשחקן שנמצא בשלב הגבוה ביותר מבין כל השחקנים בקבוצה.

לכן, הגישה לשחקן זה מתבצעת במספר קבוע של פעולות.

עבור המקרה בו *GroupID* חיובי, על הפונקציה למצוא את הקבוצה הרלוונטית בעץ הקבוצות- דבר המתבצע בסדר גודל של $\log k$ פעולות.

לכן, הסיבוכיות שמתקבלת היא:

- $\boldsymbol{O(\log k)}$ עבור *GroupID > 0*
- $\boldsymbol{O(1)}$ עבור *GroupID < 0*

8. $\boldsymbol{StatusType\ GetAllPlayersByLevel(void^{*}\ DS, int\ GroupID, int^{**} Players, int^{*}\ numOfPlayers)}$

– הפונקציה בודקת את תקינות הקלט, לאחר מכן בודקת אם GroupID חיובי או שלילי, אם חיובי היא מקצה מערך לפי מספר השחקנים בקבוצה המבוקשת, ורצה על עץ השחקנים של הקבוצה בשיטת in-order הפוך (כלומר, מהערכים הגדולים לקטנים) ומכניסה את איברי הקבוצה למערך.

אם GroupID שלילי, אותן פעולות מתבצעות על עץ השחקנים הכללי. המערך המוקצה הוא בגודל מספר השחקנים הכולל במערך. הערך שאליו מצביע numOfPlayers משתנה להיות גודל העץ הרלוונטי(אם GroupID חיובי- גודל עץ השחקנים של הקבוצה ואחרת גודל עץ השחקנים במערכת כולה)

<u>סיבוכיות:</u> בכל עץ במבנה הנתונים שמור משתנה המחזיק את גודל העץ, לכן, בדיקת גודל העץ נעשית על ידי מספר קבוע של פעולות.

הריצה על העץ והעתקת האיברים למערך דורשת מספר פעולות בסדר גודל של מספר האיברים בעץ, ולכן עבור GroupID > 0 הפעולה תתבצע בסדר גודל של n_{group} פעולות, ועבור GroupID < 0 היא תתבצע בסדר גודל של n_{total} פעולות. חיפוש הקבוצה המתאימה שאת שחקניה נרצה לשים במערך מתבצע בסדר גודל

של $\log k$ פעולות.

לכן, בסך הכל נקבל שהסיבוכיות היא:

- $\boldsymbol{O(log\ k + n_{group})}$ עבור *GroupID > 0*
- $\boldsymbol{O(n_{total})}$ עבור *GroupID < 0*.

9. $\boldsymbol{StatusType\ GetGroupsHighestLevel(void^{*}DS, int\ numOfGroups, int^{**}Players)}$ –

הפונקציה בודקת את תקינות הקלט, לאחר מכן בודקת האם גודל העץ של הקבוצות הלא ריקות קטן ממספר הקבוצות הנדרש בקלט. אם כן מוחזר ערך שגיאה.

אחרת, היא מקצה מערך בגודל *numOfGroups*, ואז רצה על העץ בו נמצאות רק הקבוצות הלא-ריקות בשיטת *in-order* ומסיימת כשעברה על *numOfGroups*, ועבור כל קבוצה מכניסה את המספר המזהה של השחקן המקסימלי למערך.

<u>סיבוכיות:</u>

על מנת להתחיל את ההעתקת המספרים המזהים של השחקנים, על הפונקציה להגיע לעלה הכי שמאלי בעץ הקבוצות הלא ריקות, לצורך כך מספר הפעולות הנדרש הוא כעומק העץ ולכן הוא מסדר גודל של $\log k$, לאח מכן הריצה על הקבוצות דורשת מספר פעולות בסדר גודל של מספר הקבוצות שנרצה לבדוק עבורן את השחקן המקסימלי- כלומר $numOfGroups$. הגישה לשחקן המקסימלי מתבצעת במספר פעולות קבוע.

הסיבוכיות המתקבלת היא $\boldsymbol{O(\log k + numOfGroups)}$ כנדרש.

10. $\boldsymbol{void\ Quit(void^{**}DS)}$-

הפונקציה קוראת להורס של מבנה הנתונים (בעקבות כך נקראים ההורסים של כלל המחלקות הממומשות), ומשנה את הערך אליו *DS* מצביע להיות *nullptr*.

<u>סיבוכיות:</u>

כאשר ההורס של מבנה הנתונים נקרא, ההורסים של העצים נקראים ואז ההורסים של ה*node*-ים נקראים.

במבנה הנתונים יש מספר קבוע של עצים. עץ אחד בגודל מספר הקבוצות ויתר העצים בגודל מספר השחקנים (כולל *non_empty_groups* שלכל היותר בגודל מספר השחקנים).

קריאה להורס של כל אחד מהעצים דורשת מספר פעולות בסדר גודל של מספר האיברים בעץ.

בנוסף לעצים הכלליים, יש צורך במחיקת עצי השחקנים של הקבוצות, אך נשים לב שסכום הגדלים של עצים אלו שווה לסך כל השחקנים במערך, לכן המחיקה שלהם תהיה בסדר גודל של n פעולות.

בסך הכל נקבל שהמחיקה היא מסדר גודל של $k + 4 \cdot n$ ולכן, הסיבוכיות היא $\boldsymbol{O(n + k)}$

<u>סיבוכיות מקום:</u>

במבנה הנתונים שלנו יש 5 עצים כללים- 1 בגודל k_{groups} ו- 4 בגודל n_{total} (כפי שהוסבר בסעיף קודם).

במהלך הריצה, במידה ונקראה הפונקציה *getGroupPlayersByLevel*, הקצאת המקום במקרה הגרוע היא של מערך בגודל n_{total} .

הקצאת המקום של הפונקציה *getGroupHighestLevel* היא בגודל הפרמטר *numOfGroups*, אך גודלו קטן או שווה למספר הקבוצות הכולל ולכן הקצאת המקום במקרה הגרוע היא בגודל k.

במידה ונקראה הפונקציה *increaseLevel*, הקצאת המקום הנוספת היא במקרה הגרוע *2n* (במקרה ששתי הקבוצות מכילות את כל השחקנים, אז גודל שני מערכי ה*in-order* בחיבורים הם *n* והמערך הממוזג שנוצר בגודל *n*).

בכל קריאה לפונקציה רקורסיבית (של העץ), סיבוכיות המקום היא כעומק העץ. כלל העצים מאוזנים ולכן סיבוכיות המקום תהיה במקרה הגרוע $O(\log n)$ (או $O(\log k)$ אם זה עץ הקבוצות).

$\log n < n$, $\log k < k_{groups}$ ולכן סיבוכיות המקום שנקבל היא $\boldsymbol{O(k_{total} + n_{total})}$ כנדרש.