תרגיל רטוב 2

מבנה נתונים

מגישות:
נתלי אטינגין 209051275
טל פלג 316276955

תיאור כללי של התוכנית:
נשתמש במבנה הנתונים unionFind על מנת לתחזק את הקבוצות. כל קבוצה תכיל מידע על השחקנים שלה, בעזרת עץ level-ים, כך שכל צומת בעץ ישמור נתונים על מספר השחקנים בlevel מסויים.
נשמור גם עץ level-ים כללי עבור כל השחקנים במשחק.
בנוסף נשתמש במבנה הנתונים hashTable כדי לשמור את כל השחקנים במשחק.

<u>מבני הנתונים:</u>
<u>UnionFindGroup:</u>
נשתמש במבנה הנתונים UnionFind ונממש אותו בעזרת עצים הפוכים, ואיחוד וכיווץ מסלולים כפי שראינו בהרצאה. ה-UnionFind יכיל מערך בגודל k+1 כך שכל תא בו יצביע לקבוצה וכל קבוצה תצביע ל-group_info, שיכיל מידע על הקבוצה. בהתחלה נאתחל את ה-father של כל קבוצה להיות nullptr.
בהמשך, כאשר נתחיל לאחד קבוצות, אז תחת כל קבוצה, יהיו מספרי הקבוצות שאיחדנו - שהם מייצגים קבוצה אחת, וכל אחד ממספרי הקבוצה יהיה בעץ הפוך - כך שכל האיברים בעץ זה יתייחסו לאותה הקבוצה. ה-father של השורש יהיה nullptr והשורש של העץ ההפוך יצביע למתאר - groupInfo שהוא יחזיק מידע על קבוצה.

<u>AVLLevelTree:</u>
עץ AVL מסוג level. ה-data של העץ תהיה מסוג LevelInfo.
בנוסף למידע הנ"ל ובנוסף למידע שכל צומת מחזיקה בעץ AVL רגיל,צומת בעץ AVLLevelTree יכיל מידע נוסף:
int sum_num_players- שומר את מספר השחקנים הנמצאים בתת עץ של הצומת.
int* sum_scale- מערך שכל תא בו מייצג score. בכל תא יהיה כתוב מספר השחקנים הנמצאים בscore המתאים בתת עץ של הצומת. בתא 0 לא יהיו כלל שחקנים כי אין שחקנים עם score=0.
int level_sum_players- שומר את מספר השחקנים הנמצאים בתת עץ הנוכחי כפול מספר הצומת.
בכל עדכון של העץ (insert, remove, merge) נעדכן את השדות הנוספים. העדכון יעשה רק על הצמתים הרלוונטים במסלול של הצומת ששונה כפי שראינו בהרצאה, (עדכון של עץ דרגות/סכומים).
כל קבוצה תכיל עץ כנ"ל (בGroupInfo שלה) ובנוסף נחזיק עץ כנ"ל שיכיל את השלבים של כל השחקנים במשחק (ונספור בו את כל השחקנים הקיימים במשחק).

<u>hashTable:</u>
נשתמש במבנה הנתונים hash_table ונממש אותו בעזרת מערך דינאמי.
בהתחלה הגודל של המערך יהיה 10, וכל פעם שהמערך יתמלא, נגדיל אותו פי 2.
כל תא במערך יכיל עץ של שחקנים, AVLTree, שממויין לפי ה-id של השחקנים וכל צומת בעץ תהיה מסוג player_info. כל פעם שנרצה להכניס שחקן, נשתמש בפונקציית הערבול שלנו על ה-id של השחקן. כך נמצא את התא המתאים לשחקן ובתא המתאים נכניס את השחקן לעץ.
פונקציית הערבול של המערך תהיה h(x)=modh כך ש-h מייצג את גודל המערך (לכן כל פעם שהמערך יגדל פונקציית הערבול תשנה בהתאם).
לכן, בעזרת פונקציית העירבול והמערך הדינאמי קיבלנו פיזור אחיד של השחקנים ולכן כפי שראינו בהרצאה, הוצאה והכנסה של שחקנים תהיה בסיבוכיות של O(1) בממוצע על הקלט.
<u>AVLlTree:</u>
עץ AVL , כפי שראינו בהרצאות ובתרגולים, שה-data שלו תהיה מסוג PlayerInfo. כל צומת ב-hashTable יהיה מהסוג של AVLlTree.
בנוסף למבני הנתונים הנ"ל, נשמור מערך ששומר את כל השחקנים בכל המשחק בשלב 0 לפי הscore שלהם, (בתא 0 לא יהיו כלל שחקנים כי אין שחקנים עם score=0).

התוכנית שלנו תכיל את טיפוסי הנתונים הבאים:
<u>Group-</u>
טיפוס נתונים שיכיל את המידע הבא:
<u>group* father</u> - הצומת אליה מצביעה הקבוצה בעץ ההפוך בunionFind
<u>GroupInfo* group_info</u> - מכיל את המידע של כל קבוצה.
<u>int group_id</u> - מספר הקבוצה

GroupInfo -

טיפוס נתונים שיכיל את המידע הבא:

AVLLevelTree <LevelInfo> - עץ level-ים מותאם מסוג LevelInfo שיכיל את כל השלבים במשחק שבהם השחקנים של אותה קבוצה נמצאים.

int num_of_groups- יכיל את מספר הקבוצות שנמצאות מתחת לgroup_info הנ"ל (לאחר איחוד הקבוצות). בהתחלה, נאתחל משתנה זה ל-1 עבור כל הקבוצות.

int arr_size- הגודל של המערך ששומר את השחקנים בשלב 0 לפי הscore שלהם, כלומר scale+1

int* scale_level_0[arr_size]- מערך ששומר את כל השחקנים בשלב 0 (בקבוצה) לפי הscore שלהם, (בתא 0 לא יהיו כלל שחקנים כי אין שחקנים עם score=0).

LevelInfo-

טיפוס נתונים שיכיל את המידע הבא:

int num_level- מספר השלב

int this_num_level- מספר השחקנים שנמצאים ב-level הנוכחי.

int* scale_arr- מערך שכל תא בו מייצג score. בכל תא יהיה כתוב מספר השחקנים הנמצאים בscore המתאים (בlevel הנוכחי). בתא 0 לא יהיו כלל שחקנים כי אין שחקנים עם score=0.

int arr_size- הגודל של המערך ששומר את השחקנים לפי הscore שלהם, כלומר scale+1

PlayerInfo-

טיפוס נתונים שיכיל את המידע הבא:

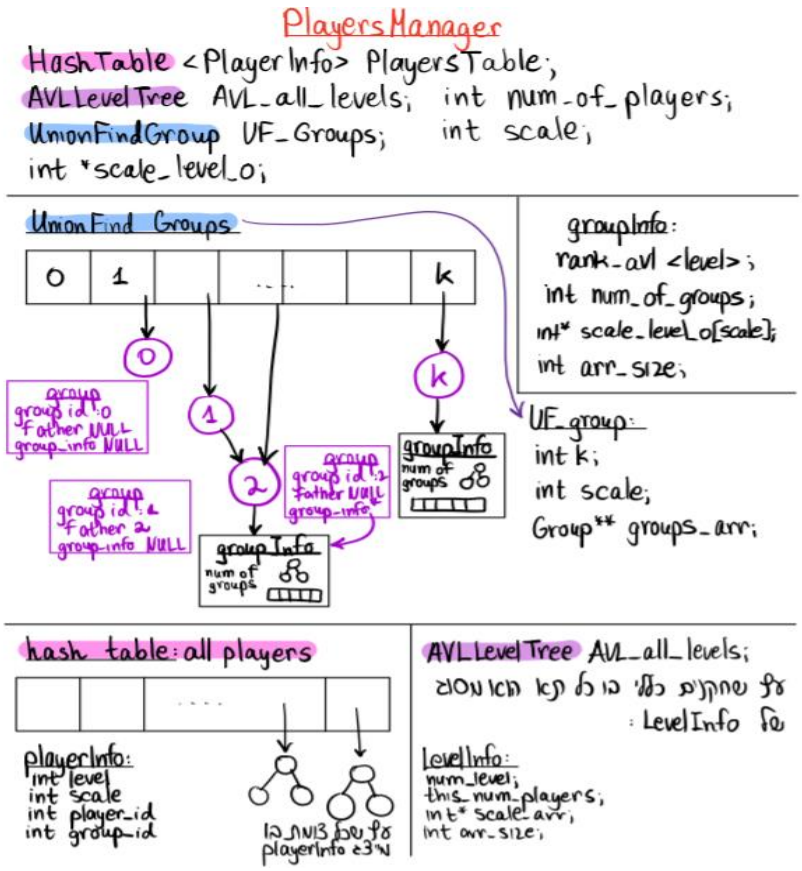
int player_id- ה-id של השחקן

int group_id-מספר הקבוצה שהשחקן נמצא בה

int level- השלב בו השחקן נמצא

int scale- הניקוד של השחקן

תמונה להמחשה:



פירוט על הפונקציות:

void* Init(int k, int scale)

נאתחל את כל המשתנים של ה-PlayersManager:
ב-scale נכניס את ה-scale קיבלנו – פעולה אחת.
ניצור מערך בגודל scale+1 ונאתחל אותו באפסים – מספר סופי של פעולות ולכן בסיבוכיות של O(1)- מערך שישמש לספירת ניקוד השחקנים ברמה 0.
ניצור עץ levelים ריק, כפי שראינו הפעולה תהיה בסיבוכיות של O(1).
נאתחל את ה-hashTable למערך בגודל התחלתי של עשר, כל שכל תא הוא מסוג עץ AVL רגיל – תחילה הוא עץ ריק – זהו מספר קבוע של פעולות, לכן בסיבוכיות של O(1).
ניצור מערך מסוג group* בגודל k כמספר הקבוצות במשחק. כל איבר במערך יצביע לצומת מסוג group, כלומר נאתחל k איברים, ובכל צומת, ה-father יהיה nullptr, וה-group_id יהיה מספר בין 1 לk. בנוסף יהיה מבציע לאיבר מסוג group_info שאותו ניצור כך ש-num_of_groups שלו יהיה 1 וה- level_avl tree יהיה עץ ריק. סה"כ סיבוכיות של O(k).
לכן, סך כל אתחול המערת יהיה בסיבוכיות של הפונקציה תהיה O(k) כנדרש.

StatusType mergeGroups(void *DS, int GroupID1, int GroupID2)

כפי שראינו בהרצאה, נאחד 2 קבוצות ע"י פעולת ה-union, כלומר ניקח את השורש של הקבוצה הקטנה יותר כך שה-father שלו יצביע לשורש של הקבוצה הגדולה יותר.
נחפש את שני השורשים של שני הקבוצות על ידי פעולת ה-find – במהלך פעולה זו נבצע כיוון העצים ההופכים – כפי שנלמד בהרצאה בסיבוכיות של O(log*k).
כעת נרצה לאחד בין שני הgroup_info.
ה-num_of_groups החדש יהיה הסכום של 2 ה-num_of_groups של כל קבוצה.
בנוסף נאחד את 2 עצי הlevelים שלנו:
נהפוך כל עץ למערך ממויין, הפעולה תתבצע בסיבוכיות של O(n), משום שמספר הצמתים בעץ הוא לכל היותר כמספר השחקנים – לכן פעולה זו חסומה על ידי סיבוכיות זו.
כעת נאחד את 2 המערכים למערך ממויין אחד – בסיבוכיות של O(2n) לכל היותר כלומר בסיבוכיות של O(n). נשים לב כי אם יש לנו את אותו ה-level השני המערכים – נאחד אותם ונכניס אותו פעם אחת בלבד.
האיחוד של ה-levelים לוקח מספר סופי של פעולות ולכן בסיבוכיות של O(1). לכן בסה"כ איחוד שני העצים למערך לוקח O(n).
ולבסוף, נהפוך את המערך הממויין לעץ. הפעולה תתבצע בסיבוכיות של O(n).
בסה"כ, כל הפעולות האיחוד של 2 העצים יהיו בסיבוכיות של O(n).
בנוסף נאחד את שני המערכים של ה-scale_level_0 – מספר סופי של פעולות ולכן לוקח O(1).
לאחר מכן נמחק את הgorup_info של הקבוצה הקטנה יותר.
כעת, נעדכן את המצביעים, עבור השורש של הקבוצה הקטנה, נעדכן את המצביע group_info ל-nullptr.
מכיוון שאנו עובדים עם עצים הפוכים, כפי שראינו בהרצאה, סה"כ הסיבוכיות של כל הפונקציה תהיה O(log*k+n).

StatusType AddPlayer(void *DS, int PlayerID, int GroupID, int score)

תחילה ניצור משתנה חדש מסוג player_info ונוסיף אותו ל-hash table – בסיבוכיות של O(1) בממוצע על הקלט – משום שעל ידי פונקציית העירבול שפירטנו עליה למעלה יתבצע פיזור אחיד.
בגלל שהשחקן הוא מ-level 0 בתחילת המשחק, אז אחד לתא במערך scale_level_0 עם הscale של השחקן שקיבלנו.
כעת נוסיף אותו לקבוצה הרלוונטית – ראשית נחפש את הקבוצה בUnionGroup על ידי פקודת ה-Find שתוך כדי עושה כיווץ עצים הפוכים ולכן סיבוכיות זו תהיה O(log*k). שם נעדכן את מערך ה-level_0 באותו האופן שעשינו במערך הכללי.
בנוסף נעדכן את מספר השחקנים בכל המערכת – פעולה אחת.
סה"כ הסיבוכיות במשוערכת בממוצע על הקלט תהיה O(log*k).

StatusType RemovePlayer(void *DS, int PlayerID)

נלך ל-hash table של כל השחקנים במשחק וניצור משתנה זמני מסוג player_info_to_remove ושם נשמור את הlevel ,score ו-group של השחקן שאנו רוצים למחוק. נוכל למצוא אותו בסיבוכיות של O(1) בממוצע על הקלט.

נלך לhash table של כל השחקנים במשחק ונסיר אותו מהעץ הרלוונטי. בעזרת הפונקצית עירבול יצרנו פיזור אחיד ולכן ההסרה תעשה גם כן בסיבוכיות של O(1).

אם הlevel של השחקן הוא 0 – נעדכן את התא המתאים במערך של הscoreים של השחקנים מרמה אפס – פעולה אחת בסיבוכיות של O(1).

אחרת, נלך לעץ הlevelים הכללי. נמצא את הצומת המתאים בסיבוכיות של O(m) כאשר m זה מספר הlevelים שיש במשחק. מספר הlevelים השונים הוא לכל היותר n ולכן הסיבוכיות תהיה O(n) בפעולה זו.

אם נראה שיש שם רק שחקן אחד – אז נסיר את הצומת ונעדכן את כל הצמתים שבהם עברנו בחיפוש צומת זה באופן רקורסיבי את הדרגות בהתאם – העדכון יהיה מספר סופי של פעולות ולכן כל עדכון יקח O(1). אם יש בlevel הזה יותר משחקן אחד – אז רק נעדכן בצומת את מספר השחקנים ואת מספר המערך scaleים ונעדכן גם את מסלול החיפוש באותו האופן כמו שעידכנו למעלה. בסה"כ פעולה זו לוקחת O(m) כאשר n זה מספר השחקנים במשחק.

כעת נלך לקבוצה המתאימה בעזרת המערך של הקבוצות. מכיוון שאנו עובדים עם עצים הפוכים הסיבוכיות המשוערכת תהיה O(log*k) כאשר נשתמש בפקודת הfind. בקבוצה הרלוונטית נלך לgroup_info ולעץ levelים ונעדכן אותו כפי שעידכנו את עץ הlevelים הכללי ולכן הפעולה תיקח גם O(logm) כאשר m זה מספר השלבים.

סה"כ הסיבוכיות הכוללת תהיה O(log*k+logm). בגלל שמספר הlevelים במשחק הוא לכל היותר n מתקיים כי הסיבוכיות הכוללת היא: O(log*k+logn) משוערכת בממוצע על הקלט.

StatusType IncreaseIDLevel (void *DS, int PlayerID, int LevelIncrease)

נמצא את השחקן הרלוונטי ב-hashTable ונעדכן את הlevel שלו ל-level בחדש, ונשמור בצד את המספר קבוצה של השחקן, ה-level הנוכחי (כלומר לפני העדכון) והscore שלו.

אם ה-level הישן של השחקן היה 0, נעדכן את מערך הscore_level_0 הכללי – פעולה אחת ולכן בסיבוכיות של O(1).

כעת נעדכן את עץ ה-levelים הכללי. אם הצומת של הlevel החדש קיים – נחפש אותו באופן רקורסיבי ונעדכן את התא בהתאם. לאחר מכן נעדכן את כל הדרגות שלו שעברנו בהם במהלך החיפוש – בסה"כ פעולה זו תיקח סיבוכיות של O(m) כאשר זה מספר ה-levelים השונים שיש במשחק.

אם הold_level היה שונה מאפס – נחפש אותו בעץ, אם היה בו רק שחקן אחד אז נמחק אותו, אחרת נעדכן אותו. ולאחר מכן נעדכן את כל דרגות הצמתים שעברנו בהם במהלך החיפוש של הצומת המתאים.

פעולות העדכון הן פעולות סופיות בסיבוכיות של O(1) ולכן בסה"כ נקבל שפעולה זו תיקח לנו O(logm).

כעת נעשה באותו אופן מה שעשינו גם בקבוצה המתאימה על ידי המידע שהוצאנו בהתחלה מה-hashTable. פעולה הfind בunionfind לוקחת O(log*k) כי תוך כדי נעשה כיווץ עצים ולכן בסה"כ העדכון הכולל של כל המערכת ייקח O(log*k+logm), m שווה לכל היותר n כמספר השחקנים במערכת ולכן חסם לסיבוכיות זו תהיה O(log*k+logn) ולכן זו תהיה הסיבוכיות המשוערכת בממוצע על הקלט.

<u>בונוס</u>: על מנת לממש את הפעולה הנ"ל בסיבוכיות של O(log*k+logn) משוערכת – לא בממוצע על הקלט, במקום ה-hashTable נוכל להשתמש בעץ trie כאשר הא"ב שלנו הוא כל הספרות מ-0 עד 9. במצב זה, חיפוש שחקן ייקח בסיבוכיות של O(10) שזה יהיה O(1) כי 0 זה מספר סופי, ועדיין החיפוש של הרמה שאותה אנחנו רוצים לשנות יהיה O(logn) בעץ הlevelים הכללי ובעצי הlevelים שבכל קבוצה.

StatusType changePlayerIDScore(void *DS, int PlayerID, int NewScore)

נמצא את השחקן הרלוונטי באמצעות הhash table. נעדכן את הscore שלו ונשמור את המספר קבוצה של השחקן ,הlevel והscore הנוכחי (כלומר לפני העידכון).
אם הlevel שם השחקן הוא 0 – נבצע עדכון בscale_level_0 בלבד – הוספת אחד בתא של הscrore החדש והורדת אחד בתא של הscore הישן. מספר פעולות סופי שלוקח O(1).
אם הlevel שלו אינו אפס – נעדכן גם את עץ הlevelים הכללי.
נחפש שם את הצומת הרלוונטי, כלומר הצומת עם הlevel של השחקן. נעדכן את מערך הscaleים בהתאם לscore החדש של השחקן. הסיבוכיות תהיה O(logm) - חיפוש הצומת בעץ. בנוסף, בגלל שחיפשנו את הצומת באופן רקורסיבי, כל פעם שנעלה שלב אחד למעלה במסלול החיפוש נעדכן את הדרגה המתאימה של מערכך הדרגות באופן דומה.
כעת נלך לקבוצה המתאימה בעזרת המערך של הקבוצות. מכיוון שאנו עובדים עם עצים הפוכים הסיבוכיות המשוערכת תהיה O(log*k) כפי שהסברנו בפונקציות קודמות. בקבוצה הרלוונטית נלך לplayer_info ולעץ levelים ונעדכן אותו כפי שעידכנו את עץ הlevelים הכללי ולכן הפעולה תיקח גם O(logm) כאשר m זה מספר השלבים.
סה"כ הסיבוכיות של הפונקציה תהיה O(log*k+logm) וכפי שהסברנו מקודם, מספר הlevelים הוא לכל היותר n ולכן הסיבוכיות תהיה בסה"כ O(log*k+logn).

StatusType GetPercentOfPlayersWithScoreInBounds (void *DS, int GroupID, int score, int lowerLevel, int higherLevel, double * players)

נבדוק אם הGroupID שווה לאפס. במידה וכן נלך לעץ הlevelים הכללי של כל השחקנים.
תחילה נבדוק אם הlowerLevel שווה לאפס. במידה וכן, נעבור על המערך ששומר את כל השחקנים ב0 level ונסכום את מספר השחקנים בו. מכיוון שהמערך הוא בגודל scale+1, כלומר גודל קבוע, המעבר על המערך יהיה בסיבוכיות של O(1). בנוסף, נלך במערך הנ"ל לתא עם ה-score שקיבלנו ונשמור את מספר השחקנים הנמצאים בשלב זה.
לאחר מכן, נלך לעץ הכללי על מנת למצוא את מספר השחקנים שנמצאים בlevel קטן או שווה ל-higherLevel. נעבור על העץ, במידה והצומת קטן או שווה ל-higherLevel, נוסיף את מספר השחקנים בצומת ואת מספר השחקנים הנמצאים בתת עץ השמאלי של הצומת. לאחר מכן נמשיך לתת עץ הימני ונבצע את אותה בדיקה. נעצור כאשר נגיע לצומת שהיא nullptr. אם הצומת גדול מהצומת higherLevel, נמשיך לתת העץ השמאלי ונבצע את אותה בדיקה. גם כאן נעצור כאשר נגיע ל-nullptr. סה"כ הסיבוכיות של הפעולה הנ"ל תהיה O(logm), כפי שראינו בהרצאה על עץ סכומים וכאשר m הוא מספר השלבים בעץ, מספר השווה או קטן ל-n. נבצע שוב את אותה פעולה, אך כעת במקום לסכום את מספר השחקנים הכללי, נסכום את מספר השחקנים שה-score שלהם שווה ל-score שקיבלנו, בעזרת המערך סכומים. הפעולה תתבצע בדיוק באותה צורה של הסכימה של כל השחקנים ולכן גם הסיבוכיות שלה תהיה O(logm).
לאחר שמצאנו את מספר השחקנים בLevel אפס ואת מספר השחקנים הנמצאים בlevel קטן או שווה ל-higherLevel , נסכום אותם ונבדוק אם הם שווים לאפס. במידה וכן נזרוק שגיאה, אחרת, נחשב את האחוזים של מספר השחקנים בscore הנתון מתוך כלל השחקנים בטווח הנתון ונציב את התוצאה במשתנה players.
אם הlowerLevel שונה מאפס, נלך לעץ הlevel-ים ונסכום את כל השחקנים שנמצאים בLevel קטן ממש מהlowerLevel, ואת מספר השחקנים שה-score שלהם שווה לscore הנתון ונמצאים בשלב קטן ממש מהLowerlevel. הפעולות הנ"ל יבוצעו כפי שביצענו את הפעולות עבור הhigherLevel ולכן יבוצעו בסיבוכיות של O(logm). לאחר מכן נבצע את אותה הפעולה עבור שחקנים בשלב שווה או קטן מה-higherLevel, כפי שראינו יתבצע בסיבוכיות של O(logm).
נחסר בין מספר השחקנים שמצאנו בhigherLevel לבין מספר השחקנים שמצאנו בlowerLevel. אם מספר השחקנים שווה לאפס נחזיר שגיאה, אחרת נחשב את האחוזים של מספר השחקנים בscore הנתון מתוך כלל השחקנים בטווח הנתון ונציב את התוצאה במשתנה players.
במידה והGroupID שונה מאפס, נחפש את הקבוצה המתאימה בUnionFind, הדבר יתבצע בסיבוכיות של O(log*k) משוערכת כפי שראינו בהרצאה ובקבוצה המתאימה נבצע את אותן פעולות שביצענו עבור GroupId ששווה לאפס.
סה"כ הסיבוכיות המשוערכת של הפונקציה תהיה O(log*k+logm) משוערכת כנדרש (מכיוון ש m קטן או שווה לn).

StatusType averageHighestPlayerLevelByGroup(void* DS, int GroupID, int m, double *avgLevel)

נבדוק אם ה GroupIDשווה לאפס. במידה וכן נלך לעץ הlevelים הכללי של כל השחקנים. נשתמש בפונקציה שמוציאה את הסכום של המספר השחקנים הדרוש ולאחר מכן נחלק את הסכום הזה ב-m שקיבלנו. ראשית נבדוק את m גדול ממספר השחקנים שיש לנו בעץ (בהנחה והוא קטן ממספר השחקנים הכולל במערכת). אם כן, אז בעזרת הדרגה ששמרנו בכל צומת – נוציא מהשורש את הסכום הכולל של כל הlevelים שיש לנו בעץ ונבצע את החישוב (בגלל ששאר השחקנים הם מ-level 0 אז השלב שלהם לא תורם לנו לחישוב), אחרת, נלך לפונקציה.

הפונקציה בנוייה באופן הבא: היא מקבלת מצביע למשתנה int שהערך ההתחלתי שלו שווה ל-m – נקרא לו counter, ומצביע למשתנה double שהוא סוכם את הסכום הדרוש.

הפונקציה בנויה באופן רקורסיבי, כל שתנאי העצירה שלה הוא אם ה-node שווה לnullptr או אם ה-counter שווה ל-0. אם לא, היא בודקת האם הבן הימני הוא null. אם כן, היא תיסכום את סכום השלבים הכולל של השחקנים שנמצאים בשורש שבו היא נמצאת בעזרת משתנה הדרגות ששמרנו בהתאם לcounter, לאחר מכן תחסיר ממנו את כמות השחקנים שהיא סכמה, ותיכנס ברקורסיה לבן השמאלי של השורש.

אם הבן הימני הוא לא null – נבדוק קודם האם הדרגה הכוללת של סכום כל שחקנים של הבן הימני גדולה מהcounter – אם כן, ניכנס ברקורסיה עם הבן הימני לפונקציה, אם לא – נבדוק האם הcounter של הבן הימני נמצא בין כלל השחקנים שיש בתת עץ הימני לבין כלל השחקנים שיש בתת עץ הימני ועוד מספר השחקנים בשורש. אם הוא נמצא שם, נבצע את הסכימה של הlevelים ונאפס את הcounter ל-0. אם אף אחד מהתנאים לא מתקיים, זה אומר שלבן הימני יש מספר בנים קטן ממנה שאנחנו צריכים, לכן נסכום את ה-שלבים של כל התת עץ הימני, ונקרא לרקורסיה עם הבן השמאלי. בסה"כ הפוקנציה הזו ניעשת בסיבוכיות של O(logm) כי היא מבצעת מסלול חיפוש עד השורש הרלוונטי. בגלל שm הוא לכל היותר n הסיבוכיות תהיה O(logn). נעשה את אותה הפעולה על העץ של הקבוצה – בגלל שאנחנו משתמשים בפונקציית find ויש לנו עצים הפוכים, כפי שכבר פירטנו הסיבוכיות תהיה O(log*k). ולכן בסה"כ הסיבוכיות המשוערכת תהיה O(log*k+logn).

void Quit(void **DS)

נמחק את כל מבני הנתונים שאנחנו צריכים למחוק בתוכנית:

נמחק את עץ הlevelים הכללי - נשתמש בפונקציית deleteTree() שמוחקת את העץ – נממש אותה בעזרת פונקציה פנימית שעוברת ומוחקת בצורה רקורסיבית את כל הnodeים.

נמחק את הhashTable – מחיקת המערך – פעולה אחת שתיקח לכל היותר O(n) כמספר השחקנים. בכל תא יימחק העץ שנמצא שם – ולכן הסיבוכיות מחיקת הhashTable תיקח O(n).

נמחק את הunionFind על ידי מחיקת כל התאים במערך הקבוצות – שבמחיקתם ייקרא ההורס של הgroupInfo ושם נמחק את עץ הlevelים של כל קבוצת כפי שמחקנו את העץ הכללי.

מעבר על המערך ייקח O(k) ומחיקת כל עץ בכל המבנה תיקח לכל היותר O(m) ובגלל שm קטן יותר מn הסיבוכיות תהיה לכל היותר O(n) ולכן בסה"כ המחיקה תהיה O(n+k).

סיבוכיות מקום:

מערך דינאמי שסיבוכיות המקום שלו היא לכל היותר n ולכן הסיבוכיות תהיה O(n).

בנוסף מערך של k קבוצות של ה-UnionFind שהיא O(k) ולכן בסה"כ סיבוכיות המקום תהיה O(n+k).