מבני נתונים 1

234218

תרגיל רטוב 1

מספר

הוגש ע"י:

עמרי בן ארי	209540483
שם	מספר זהות
גל שור	208253559
שם	מספר זהות

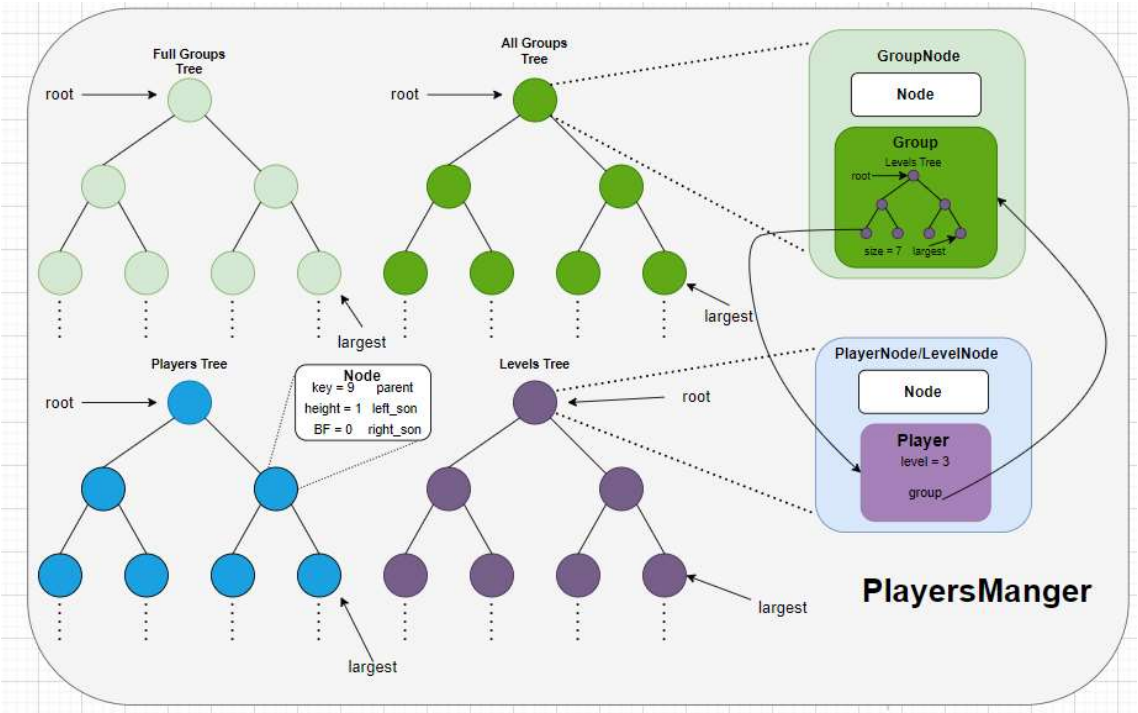
ציון:

לפני בונוס הדפסה:

כולל בונוס הדפסה:

נא להחזיר לתא מס':

חלק א' – הצגת מבנה הנתונים באמצעות ציור



חלק ב' – הצגת רכיבי מבנה הנתונים באופן מילולי

על מנת להבין את מבנה הנתונים שלנו אותו נכנה בשם PlayersManager יש צורך להזכיר תחילה כמה מחלקות עזר הממשות מבני נתונים פנימיים.

1. **Node** : המחלקה הבסיסית ביותר במבנה שלנו היא Node שהינה מחלקה אבסטרקטית.
מחלקה זו מייצגת צומת כללי בעץ ומאגדת בתוכה את השדות שרלוונטיים לכלל סוגי הצמתים והם : הערך השמור בצומת (key), גובה הצומת (height), גורם האיזון של הצומת (balance factor) ומצביעים לבן השמאלי, הימני והאבא של הצומת (left_son, right_son, parent).

2. **4 מחלקות היורשות מ- Node** : כל אחת מהן מממשת את השדות והמתודות הפנימיות של Node אך לכל אחת יש שדות או מתודות שמאפיינות רק אותה. המחלקות האלה משמשות בפועל כצמתי עצי ה-AVL במבנה שלנו.

4 המחלקות היורשות מ-Node הן :
   a. **PlayerNode** – צומת המכילה בנוסף לשדות הכלליים גם שדות הרלוונטיים לשחקן בלבד והם הרמה (level) של השחקן ומצביע לקבוצה אליה השחקן משתייך.
   b. **GroupNode** - צומת המכילה בנוסף לשדות הכלליים גם שדות הרלוונטיים לקבוצה בלבד והם המזהה (id) של הקבוצה ומצביע לעץ AVL של כלל השחקנים החברים בקבוצה זו.
   c. **LevelNode** - בדומה ל-PlayerNode צומת זו מאגדת בתוכה שדות הרלוונטיים רק לשחקן אך בשונה מה-PlayerNode יחס הסדר המוגדר על שני צמתים מטיפוס זה נקבע קודם כל לפי הרמה שלהם (הגדול יותר הוא בעל הרמה הגבוהה יותר) ורק אז לפי ה-id שלהם (הגדול יותר הוא בעל ה-id הקטן יותר).

3. **AVLTree** : מחלקה המתארת את מבנה הנתונים עץ AVL שנלמד בהרצאות ובתרגולים. מחלקה זו אינה אבטרקטית וצמתיה הן מטיפוס *Node.
המחלקה שמימשנו מכילה גישה לשורש (root) של העץ ובנוסף גם לצומת הימנית ביותר בכל עץ שהיא הצומת הגדולה ביותר בעץ מכיוון וזה עץ חיפוש (largest_node). בנוסף בכל עץ קיים שדה המתאר את גודל העץ (size).
על עץ ה- AVL שלנו מוגדרות במחלקה מתודות המקבילות לפעולות על עצי AVL שנלמדו בהרצאות ובתרגולים (פירוט נוסף בהמשך).

כעת כשסיימנו לתאר בקצרה את מחלקות העזר במבנה הנתונים שלנו נוכל לעבור לתיאור מבנה הנתונים המרכזי, ה-PlayersManager.

מבנה הנתונים שלנו יכיל את השדות הבאים :

1. **AVLTree* all_players_tree** :

- מי אני: עץ AVL המאגד את כלל השחקנים המשתתפים במשחק.
- מי הם צמתיי: השחקנים עצמם (ובאופן טכני יותר כל צומת במבנה הוא מטיפוס PlayerNode).
- יחס הסדר בין צמתיי: בהינתן שני צמתים המתארים שחקנים נקבע ששחקן א' גדול משחקן ב' אם ה-id של שחקן א' גדול מזה של שחקן ב'.
- למה אני דרוש: עץ שחקנים המונחה על ידי ה-id מאפשר לנו לממש פעולות בסיסיות בעץ כמו חיפוש והכנסה של מידע אודות שחקנים במבנה הנתונים PlayersManager בסיבוכיות הנדרשת.

2. **AVLTree* levels _tree**:
- מי אני: עץ AVL המאגד את כלל השחקנים המשתתפים במשחק.
- מי הם צמתיי: השחקנים עצמם (ובאופן טכני יותר כל צומת במבנה הוא מטיפוס LevelNode).
- יחס הסדר בין צמתיי: בהינתן שני צמתים המתארים שחקנים נקבע ששחקן א' גדול משחקן ב' אם ה-level של שחקן א' גדול מזה של שחקן ב'. במידה ומתקיים שיוויון בין ה-levels אז שחקן א' ייחשב גדול משחקן ב' אם ה- id שלו קטן מזה של שחקן ב'.
- למה אני דרוש: מכיוון ואני מבחין בין השחקנים השונים לפי ה-level שלהם אני מאפשר לממש את הפעולות שבהם ערכי החזרה הרלוונטים תלויים ברמת השחקן ולא ב-id שלו בסיבוכיות המתאימה.

הערה: מיד נציג לפניכם שני עצי AVL נוספים שנועדו לתאר את הקבוצות במבנה שלנו. המשותף לשני העצים הוא שצמתיהם יהיו מטיפוס GroupNode. בעמוד הקודם תיארנו את המבנה הכללי של צומת מטיפוס GroupNode והזכרנו שהוא בין היתר מכיל מצביע לעץ AVL. אותו עץ AVL שנכנה בשם group_players_tree הוא עץ של כלל השחקנים המשויכים לקבוצה.

וכעת בחזרה לשדות במבנה הנתונים שלנו:

3. **AVLTree* all_groups_tree**:
- מי אני: עץ AVL המאגד את כלל הקבוצות (הן הריקות והן המלאות) המשתתפות במשחק.
- מי הם צמתיי: הקבוצות עצמם (ובאופן טכני יותר כל צומת במבנה הוא מטיפוס GroupNode).
- יחס הסדר בין צמתיי: בהינתן שני צמתים המתארים קבוצות נקבע שקבוצה א' גדולה מקבוצה ב' אם ה-id של קבוצה א' גדול מזה של קבוצה ב'.
- למה אני דרוש: לאיגוד מידע ומימוש פעולות הדורשות מאיתנו להכיר את כלל הקבוצות (הן המלאות והן הריקות במבנה) בסיבוכיות הנדרשת.

4. **AVLTree* full_groups_tree**:
- מי אני: עץ AVL המאגד רק את הקבוצות המלאות (אלה המכילות לפחות שחקן אחד) המשתתפות במשחק.
- מי הם צמתיי: הקבוצות (המלאות) עצמם (ובאופן טכני יותר כל צומת במבנה הוא מטיפוס GroupNode).
- יחס הסדר בין צמתיי: בהינתן שני צמתים המתארים קבוצות נקבע שקבוצה א' גדולה מקבוצה ב' אם ה-id של קבוצה א' גדול מזה של קבוצה ב'.
- למה אני דרוש: מימוש פעולות הדורשות מאיתנו להכיר רק את הקבוצות המלאות. בפרט את הפעולה GetGroupLeaders שדורשת מאתנו להחזיר את המובילים בכל קבוצה תוך מעבר רק על הקבוצות המלאות. שימוש בעץ הקבוצות המלאות מאפשר לנו לממש פעולות מסוג זה בסיבוכיות הזמן הדרושה.
- הערה: נשים לב כי ניתן לחסום את מספר הקבוצות בעץ הקבוצות המלא בשני דרכים:
  - דרך א': על ידי מספר הקבוצות הכוללות שכן כל קבוצה מלאה היא בפרט קבוצה (מלאה או ריקה).
  - דרך ב': על ידי מספר השחקנים במערכת. זאת מכיוון ואם קבוצה היא מלאה אז יש בה לפחות שחקן אחד ולכן לא ייתכן תרחיש שבו יש יותר קבוצות מלאות משחקנים כי זה יעמוד בסתירה להגדרה של קבוצה מלאה ולכן מספר הקבוצות המלאות חסום במספר השחקנים.

סיכום השדות במבנה הנתונים שלנו:

השדות המרכיבים את המבנה שלנו הם:

- **AVLTree* all_players_tree**
- **AVLTree* levels _tree**
- **AVLTree* all_groups_tree**
- **AVLTree* full_groups_tree**

כאשר כל אחד מהם מכיל את השדות והפעולות שציינתי בעמודים הקודמים.

במבנה הנתונים שלנו יש שתי מחלקות נוספות והם group, player המוכלות בתוך הצמתים השונים בהתאם.

- המחלקה group מאגדת בתוכה שדות הקשורים בקבוצה בלבד. שדות אלה הם ה- id של הקבוצה ומצביע לעץ המכיל את כל השחקנים בקבוצה אשר מסודרים בעץ לפי הרמות שלהם.
- המחלקה player מאגדת בתוכה אינפורמציה הקשורה בשחקן. אינפורמציה זו היא ה-level שלו ומצביע ל-group שלו.

חלק ג' – הצגת הפעולות במבנה הנתונים

בחלק זה נעבור על כלל הפעולות הדרושות ונסביר:

א. מה המטרה של כל פעולה.
ב. מה דרישת סיבוכיות הזמן בפעולה.
ג. כיצד מימשנו את הפעולה במבנה הנתונים שלנו.
ד. כיצד הפעולה משפיעה על מבנה הנתונים שלנו.
ה. מדוע המימוש של הפעולה עומד בדרישות הסיבוכיות.

נעיר כי המעברים בהוכחת סיבוכיות הזמן נובעים ממשפטים על חסמי סיבוכיות שהוכחו בתרגולים.

1) **הפעולה:**

```
void* Init()
```

- **מטרה**: מאתחל מבנה נתונים ריק.
- **דרישות סיבוכיות זמן**: O(1).
- **מימוש**: קוראים לבנאי של המחלקה PlayersManager.
בנאי זה אחראי להקצאת זיכרון עבור ארבעת העצים הראשיים שציינו קודם לכן. במצב ההתחלתי אנחנו מאתחלים את כל העצים להיות ריקים באופן הבא: ה-root שלהם מצביע ל-nullptr, ה-largest_node שלהם מצביע ל-nullptr וה-size שלהם מאותחל להיות 0.
- **מצב מבנה הנתונים לאחר ביצוע הפעולה**: המערכת מכילה ארבעה עצים ריקים.
- **מדוע אנחנו עומדים בסיבוכיות הזמן הדרושה**: הפעולה היחידה המבוצעת היא קריאה לבנאי שמבצע מספר סופי של פעולות הקצאה ואתחול ערכים שונים במבנה שלנו.

2) **הפעולה:**

```
StatusType AddGroup(void *DS, int GroupID)
```

- **מטרה**: הוספת קבוצת שחקנים חדשה עם המזהה GroupID.
- **דרישות סיבוכיות זמן**: O(log k) במקרה הגרוע, כאשר k הוא מספר הקבוצות.
- **מימוש**:
בפועל מה שנרצה לעשות זה להכניס את הקבוצה שלנו כצומת ב-all_groups_tree. (לא נרצה להכניס אותה ל-full_groups_tree כי היא כרגע ריקה).
נתאר את עיקרי המימוש בשלבים:
  - בדיקה שהקלט שקיבלנו עומד בדרישות הקלט והחזרת ערך מתאים אם לא – **O(1)**.
  - יצירת צומת מתאים לקבוצה מסוג GroupNode - **O(1)**.
  - הכנסת הצומת לעץ all_groups_tree על ידי שימוש בפעולה insert של העץ המאוזן הממומשת כפי שלמדנו בשיעור בסיבוכיות **O(log(k))**.
- **מצב מבנה הנתונים לאחר ביצוע הפעולה**: זהה למה שהיה קודם לכן פרט לעץ - all_groups_tree אליו נוספה צומת חדשה המתארת את הקבוצה.
- **מדוע אנחנו עומדים בסיבוכיות הזמן הדרושה**: O(1) + O(1) + O(log(k)) = O(log(k))

3) **הפעולה:**

```
StatusType AddPlayer(void *DS, int PlayerID, int GroupID, int Level)
```

- **מטרה**: הוספת שחקן חדש שעבר את שלב Level במשחק, ומשתייך לקבוצה בעל המזהה GroupID.
- **דרישות סיבוכיות זמן**: O(log n + log k) במקרה הגרוע, כאשר n הוא מספר השחקנים ו-k הוא מספר הקבוצות.
- **מימוש**: בעת הוספת שחקן למערכת נרצה להוסיף אותו בתצורה כזאת או אחרת לארבעת העצים המרכזיים במבנה הנתונים שלנו.

נתאר את עיקרי המימוש בשלבים:

- בדיקה שהקלט שקיבלנו עומד בדרישות הקלט והחזרת ערך מתאים אם לא – **O(1)**.
- חיפוש הקבוצה (כלומר הצומת המתאימה לקבוצה) אליה משתייך השחקן ב-all_groups_tree על ידי שימוש בפעולה find של העץ הממומשת כפי שלמדנו בשיעור בסיבוכיות של **O(log(k))** (אם קבוצה עם מזהה group_id לא קיימת אז נעצור את הפעולה כאן ונחזיר ערך מתאים).
- יצירת צומת מסוג PlayerNode המתאר את השחקן – סיבוכיות **O(1)**.
- הכנסת הצומת המתארת את השחקן לעץ all_players_tree על ידי שימוש בפעולה insert של העץ המאוזן הממומשת כפי שלמדנו בשיעור בסיבוכיות **O(log(n))**, פעולה זו אף תתריע אם קיים שחקן עם מזהה player_id ותעצור את הפעולה אם כן.
- יצירת צומת מסוג LevelNode המתאר את השחקן – סיבוכיות **O(1)**.
- הכנסת צומת זו לעץ levels_tree על ידי שימוש בפעולה insert של העץ הממומשת כפי שלמדנו בשיעור בסיבוכיות **O(log(n))**.
- הכנסת צומת זו גם לעץ group_players_tree היושב בתוך הצומת המתאימה לקבוצה בעלת המזהה group_id אותה מצאנו קודם לכן בעץ הקבוצות המלא גם כאן עושים שימוש בפעולה insert ולכן הסיבוכיות היא **O(log(n))**.
- בדיקה האם הצומת היא הצומת היחידה בעץ group_players_tree בסיבוכיות **O(1)**.
- במידה והיא אכן היחידה המשמעות היא שלפני זה קבוצה זו הייתה ריקה משחקנים ולכן כעת יש להוסיף את הקבוצה לעץ הקבוצות המלאות. פעולת הכנסה זו מבוצעת על ידי המתודה insert של העץ full_groups_tree בסיבוכיות **O(log(k))**.

- **מצב מבנה הנתונים לאחר ביצוע הפעולה**: השינוי במבנה הנתונים הוא שכעת יש צומת המאפיינת את השחקן הן ב- levels_tree, הן ב- all_players_tree והן בתוך העץ group_players_tree שאליו יש גישה מהצמתים עם ה-group_id המתאים הן ב-all_groups_tree והן ב- full_groups_tree כאשר ייתכן ונוספה צומת חדשה ל-full_groups_tree אם היא הייתה ריקה קודם לכן והשחקן הוא החבר הראשון בקבוצה הזו.
- **מדוע אנחנו עומדים בסיבוכיות הזמן הדרושה**:

$$4*O(1) + 2*O(\log(k)) + 3*O(\log(n)) = O(\log(n) + \log(k))$$

4) **הפעולה:**

```
StatusType RemovePlayer(void *DS, int PlayerID)
```

- **מטרה**: השחקן בעל המזהה PlayerID "נפסל" מהמשחק, וניתן למחוק אותו מהמערכת.
- **דרישות סיבוכיות זמן**: O(log n) במקרה הגרוע, כאשר n הוא מספר השחקנים.
- **מימוש**: בעת הסרת שחקן מהמערכת נרצה להסיר את כל המופעים שלו מארבעת העצים המרכזיים במבנה הנתונים שלנו.

נתאר את עיקרי המימוש בשלבים:

- בדיקה שהקלט שקיבלנו עומד בדרישות הקלט והחזרת ערך מתאים אם לא – **O(1).**
- חיפוש השחקן (כלומר הצומת המתאימה לשחקן) ב-all_players_tree על ידי שימוש בפעולה find של העץ הממומשת כפי שלמדנו בשיעור בסיבוכיות של **O(log(n))** (אם שחקן עם מזהה player_id לא קיים אז נעצור ונחזיר ערך מתאים).
- נשמור את הצומת שמצאנו במשתנה זמני –**O(1) .**
- הסרת השחקן מה-levels tree על ידי שימוש בפעולה remove של העץ הממומשת כפי שלמדנו בשיעור, בסיבוכיות של **O(log(n)).**
- ניגש מהצומת של השחקן ששמרנו במשתנה זמני והינה מטיפוס PlayerNode דרך השדה player אל השדה המשותף של group (הקיים בצומת של הקבוצה המתאימה לשחקן ב-all_groups_tree וב-full_groups_tree). הגישה מתבצעת ב- **O(1)** אודות לדרך בה מימשנו את מבנה הנתונים שלנו.
- בתוך השדה של group נרצה להסיר את השחקן מ-group_players_tree על ידי שימוש בפעולה remove של העץ בסיבוכיות **O(log(n))** .
- בדיקה האם השחקן היה היחידי בקבוצה שלו – **O(1).**
- אם הוא אכן היה נרצה להסיר את הצומת המתאימה לקבוצה אליה הוא משתייך מעץ הקבוצות המלאות שכן הקבוצה אליה הוא משתייך ריקה. נשתמש בפעולת remove של ה-full_groups_tree שמתקיימת בסיבוכיות של **O(log(n))** כתוצאה מכך שמספר הקבוצות המלאות חסום במספר השחקנים כפי שתיארנו קודם.
- הסרת השחקן מהעץ all_players_tree על ידי שימוש ב-remove בסיבוכיות **O(log(n))**.

- **מצב מבנה הנתונים לאחר ביצוע הפעולה**: השינוי במבנה הנתונים הוא שכעת אין צומת המאפיינת את השחקן המוסר הן ב- levels_tree והן ב- all_players_tree. בנוסף אין צומת המאפיינת את השחקן גם בתוך העץ group_players_tree של הצומת של הקבוצה אליה השתייך השחקן. ייתכן והסרנו את הצומת עם המזהה group_id המתאים מה -full_groups_tree אם השחקן היה החבר היחידי בה.

- **מדוע אנחנו עומדים בסיבוכיות הזמן הדרושה**:

$4*O(1) + 5*O(\log(n)) = O(\log(n))$

5) **הפעולה:**

StatusType ReplaceGroup(void *DS, int GroupID, int ReplacementID)

- **מטרה**: הקבוצה בעלת המזהה GroupID חודלת מלהתקיים, והקבוצה הקיימת במערכת בעלת המזהה ReplacementIDמקבלת את השחקנים שהיו בקבוצה GroupID (בנוסף לשחקנים שכבר נמצאים בקבוצה).
- **דרישות סיבוכיות זמן**: O(log(k)+n_group+n_replacement)
מימוש: נתאר את עיקרי המימוש בשלבים:
  - בדיקה שהקלט שקיבלנו עומד בדרישות הקלט והחזרת ערך מתאים אם לא – **O(1)**.
  - חיפוש הקבוצה GroupId בעץ – all_groups_tree ע"י שימוש ב- ()find בסיבוכיות - **O(log(k))**.
  - חיפוש הקבוצה ReplacementId בעץ all_groups_tree ע"י שימוש ב- ()find בסיבוכיות - **O(log(k))**.
  - במידה ואחת מן הקבוצות הללו לא נמצאה נחזיר שגיאת FAILURE - **O(1)**.
  - נבדוק אם גודל הקבוצה של GroupId הוא 0. במידה וכן נבצע אך ורק הסרה של הקבוצה מעץ הקבוצות הכללי על ידי שימוש בפעולה ()remove כפי שנלמדה בהרצאה ונסיים את הפעולה כאן.
נימוק: אין שחקנים חדשים להוסיף לקבוצה ReplacementId. אין צורך להסיר את הקבוצה בעלת המזהה GroupId מעץ הקבוצות המלאות שכן היא ריקה. מבחינת עדכון השחקנים, אין צורך לבצע עדכון של אף שחקן שכן בפועל לא העברנו אף שחקן לקבוצה חדשה - סיבוכיות **O(log(k))**.
  - אם הגענו לכאן אז קיים לכל הפחות שחקן אחד בקבוצה GroupId.
כעת נעבור על השחקנים שנמצאים בקבוצה GroupId ונעדכן את שדה הקבוצה שיצביע לקבוצה ReplacementID באמצעות מעבר Inorder על group_players_tree ב-groupId . הסיבוכיות של פעולה זו היא **O(n_group)**.
  - כעת נבצע פעולת מיזוג עצים על העץ הפנימי של GroupID והעץ הפנימי של ReplacementId לפי האלגוריתם שהוצג בשיעור. נעיר כי במהלך אותו אלגוריתם ישנו שלב בו יוצרים עץ כמעט שלם ריק בגודל n_group+n_replacement בכך שקודם יוצרים עץ שלם ורק אז מסירים צמתים. בשלב זה נמענו מלהשתמש ב-remove כדי לא לפגוע בסיבוכיות וביצענו הסרה ממוקדת. הסיבוכיות כאן היא – **O(n_group+n_replacement)** כפי שהראו בתרגול.
  - משימים את העץ הממוזג שהתקבל מהאלגוריתם לתוך הקבוצה ReplacementID בסיבוכיות **O(1)**.
  - אם בעץ GroupId היו שחקנים לפני ההסרה נסיר את groupID מעץ הקבוצות המלאות על ידי שימוש בפעולה ()remove של all_groups_tree בסיבוכיות - **O(log(k))**
  - אם ReplacementID היה ריק לפני ההסרה נבצע הכנסה של ReplacementID לתוך עץ הקבוצות המלאות על ידי שימוש בפעולה ()insert של all_groups_tree בסיבוכיות - **O(log(k))**.
  - לבסוף נעשה הסרה של GroupID מעץ הקבוצות הכללי ולמעשה נמחק לחלוטין את GroupId מהמבנה PlayersMangers. גם כאן נשתמש ב-remove של all_groups_tree בסיבוכיות **O(log(k))**.
- **מצב מבנה הנתונים לאחר ביצוע הפעולה**: הקבוצה GroupID הוסרה מהעץ all_groups_tree ומהעץ full_groups_tree (אם הייתה שם מלכתחילה). הקבוצה ReplacementID מכילה כעת את כלל השחקנים משתי הקבוצות ועל כן השדות שלה מעודכנים בהתאם (העץ הפנימי מכיל בתוכו את כלל השחקנים בצורה ממוינת והגודל מעודכן). השחקנים שהיו בקבוצה GroupId מצביעים כעת לקבוצה ReplacementID.
- **מדוע אנחנו עומדים בסיבוכיות הזמן הדרושה-**

$6*O(\log(k)) + O(n\_group)+5*O(n\_group\_n\_replacement)+2*O(1)=\underline{O(\log(k)+n\_group+n\_replacement)}$

6) **הפעולה:**

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

- **מטרה**: הגדלת השלב במשחק של השחקן בעל המזהה PlayerID ב- LevelIncrease.
- **דרישות סיבוכיות זמן**: O(log n) במקרה הגרוע, כאשר n הוא מספר השחקנים.
- **מימוש**: כשנרצה להעלות את הרמה של שחקן במערכת נרצה לעשות זאת על ידי הסרת המופעים הקודמים שלו מארבעת העצים המרכזיים שלנו ואז להכניסו מחדש לכל אחד מהעצים לאחר עדכון ה-level שלו.
הסיבה למימוש זה היא שמירה על שמורת החיפוש בעצים שלנו.
נתאר את עיקרי המימוש בשלבים:
  - בדיקה שהקלט שקיבלנו עומד בדרישות הקלט והחזרת ערך מתאים אם לא – **O(1)**.
  - חיפוש השחקן (כלומר הצומת המתאימה לשחקן) ב-all_players_tree על ידי שימוש בפעולה find בסיבוכיות של **O(log(n))** (אם שחקן עם מזהה player_id לא קיים אז נעצור ונחזיר ערך מתאים).
  - נשמור את הצומת שמצאנו במשתנה זמני –**O(1)** .
  - הסרת השחקן מה-levels tree על ידי שימוש בפעולה remove בסיבוכיות של **O(log(n))**.

- ניגש מהצומת של השחקן ששמרנו במשתנה זמני והינה מטיפוס PlayerNode לשדה group של הקבוצה המתאימה. הגישה מתבצעת ב- **O(1)** אודות לדרך בה מימשנו את מבנה הנתונים שלנו.
- בתוך השדה של group נרצה להסיר את השחקן מ-group_players_tree על ידי שימוש בפעולה remove של העץ בסיבוכיות **O(log(n))** .
- יצירת LevelNode הזהה לLevelNode- הקודם עם ה-level העדכני של השחקן ששווה ל-level הקודם בתוספת LevelIncrease – **O(1)**.
- הוספת אותו LevelNode ל-levels_tree תוך שימוש בפעולה insert של העץ בסיבוכיות **O(log(n))** .
- הוספת אותו LevelNode ל-group_players_tree תוך שימוש בפעולה insert של העץ בסיבוכיות **O(log(n))** .

- **מצב מבנה הנתונים לאחר ביצוע הפעולה**: אין שינוי מבחינת המבנה, אך ה-level של השחקן בכל צומת המתארת את השחקן מעודכן להיות ה- level החדש שהוא ה-level הקודם בתוספת LevelIncrease.
- **מדוע אנחנו עומדים בסיבוכיות הזמן הדרושה**:

$$4*O(1) + 5*O(\log(n)) = O(\log(n))$$

- **הערה:** מכיוון והשדה level הוא שדה פנימי של המחלקה player אליה מצביעים במשותף הצמתים המתארים את השחקן הן ב- all_players_tree והן ב-levels_tree. שינוי של ה-level ועדכונו בעץ אחד בלבד משפיע ישירות גם על המידע בעץ השני. ולכן גם ה-level ב-all_players_tree מעודכן.

7) **הפעולה:**

StatusType GetHighestLevel(void *DS, int GroupID, int *PlayerID)

- **מטרה**: יש להחזיר את מזהה השחקן שנמצא בשלב (Level) הגבוה ביותר מבין אלו ששייכים ל-GroupID.
- **דרישות סיבוכיות זמן**: אם GroupID<0 אז O(1) במקרה הגרוע.
  אחרת, O(log(k)) במקרה הגרוע. כאשר k הוא מספר הקבוצות.
- **מימוש**: לאחר שנבצע בדיקת קלט ב- **O(1)** והחזרת ערך מתאים אם הקלט אינו תקין נרצה לטפל בנפרד בשני המקרים.
  מימוש המקרה בו GroupID < 0 :
  - בדיקה האם אין שחקנים במערכת על ידי בדיקת ה- size של levels_tree ב- **O(1)**.
  - אם אין – מחזירים ערכים בהתאם לדרישה – **O(1)**.
  - אם יש – ניגשים לצומת הימנית ביותר בעץ (השמורה כשדה largest_node בעצי ה-AVL שלנו) בעץ levels_tree הממיין את השחקנים לפי רמותיהם ומחזירים את המפתח המתאים – **O(1)**.

  מימוש המקרה בו GroupID > 0 :
  - מחפשים האם הקבוצה בעלת המזהה group_id בכלל קיימת על ידי חיפוש הצומת המתאימה לה ב-all_groups_tree תוך שימוש ב-find בסיבוכיות **O(log(k))**.
  - אם היא אינה קיימת – מחזירים ערך מתאים - **O(1)**.
  - אם היא קיימת – בודקים האם היא ריקה משחקנים על ידי בדיקת ה- size של group_players_tree אליו ניגשים מהצומת שמצאנו ומחזירים ערכים מתאימים - **O(1)**.
- **מצב מבנה הנתונים לאחר ביצוע הפעולה**: אין כל שינוי.
- **מדוע אנחנו עומדים בסיבוכיות הזמן הדרושה**:
  - אם GroupID < 0 אז: $4*O(1) = O(1)$
  - אם GroupID > 0 אז: $3*O(1) + O(\log(k)) = O(\log(k))$

8) **הפעולה:**

StatusType GetAllPlayersByLevel (void *DS, int *GroupID*, int **Players, int *numOfPlayers)

- **מטרה**: יש להחזיר את כל השחקנים ששייכים לקבוצה בעלת המזהה GroupID ממוינים לפי השלב שלהם.
- **דרישות סיבוכיות זמן**: אם GroupID<0 אז O(n) במקרה הגרוע, כאשר n הוא מספר השחקנים במערכת.
  אחרת, O(n_GroupID+log(k)) במקרה הגרוע, כאשר n_GroupID הוא מספר השחקנים ששייכים לקבוצה בעלת המזהה GroupID ו-k הוא מספר הקבוצות.
- **מימוש**:
  לאחר שנבצע בדיקת קלט ב- **O(1)** והחזרת ערך מתאים אם הקלט אינו תקין נרצה לטפל בנפרד בשני המקרים.
  מימוש המקרה בו GroupID < 0 :
  - בדיקה האם אין שחקנים במערכת על ידי בדיקת ה- size של levels_tree ב- **O(1)**.
  - אם אין – מחזירים ערכים בהתאם לדרישה – **O(1)**.

- אם יש – נבצע סיור reverse inorder על העץ levels_tree הממיין את השחקנים לפי רמותיהם. נכניס כל צומת אליה נגיע בסיור למערך players* בגודל n שהקצנו קודם באופן כזה שאת הצומת הראשונה אליה נגיע בסיור (הצומת המתארת את השחקן המוביל) נכניס במקום הראשון, את השחקן השני הגדול ביותר במקום השני וכך הלאה. למדנו בשיעור ובתרגול כי סיור inorder מבוצע בסיבוכיות O(n) תוך מעבר על צמתי העץ מהקטן לגדול. סיור reverse_inorder ממומש בדיוק באותו אופן רק שהמעבר מבוצע מהגדול לקטן ולכן גם כאן הסיבוכיות היא **O(n)**.

מימוש המקרה בו GroupID > 0 :

- מחפשים האם הקבוצה בעלת המזהה group_id בכלל קיימת על ידי חיפוש הצומת המתאימה לה ב-all_groups_tree תוך שימוש ב-find בסיבוכיות **O(log(k))**.
- אם היא אינה קיימת – מחזירים ערך מתאים - **O(1)**.
- אם היא קיימת – בודקים האם היא ריקה משחקנים על ידי בדיקת ה- size של group_players_tree אליו ניגשים מהצומת שמצאנו - **O(1)**.
- אם היא ריקה משחקנים מחזירים ערכים מתאימים – **O(1)**.
- אם היא אינה ריקה משחקנים נקצה שוב מערך בגודל n_group_id שהוא מספר הצמתים בעץ נבצע את אותו סיור reverse_inorder אך הפעם על העץ group_players_tree שהוא העץ של השחקנים בקבוצה ממויינים לפי רמותיהם. מאותה סיבה כמו קודם הסיבוכיות היא **O(n_group_id)** שכן מספר הצמתים בעץ הוא n_group_id.

- **מצב מבנה הנתונים לאחר ביצוע הפעולה** : אין כל שינוי.
- **מדוע אנחנו עומדים בסיבוכיות הזמן הדרושה** :
  - אם GroupID < 0 אז : 3*O(1) + O(n) = O(n)
  - אם GroupID > 0 אז : 4*O(1) + O(log(k)) + O(n_group_id) = O(log(k) + n_group_id)

9) **הפעולה:**

StatusType GetGroupsHighestLevel (void *DS, int numOfGroups, int **Players)

- **מטרה** : יש להחזיר עבור כל אחת מ-numOfGroups הקבוצות בעלות המזהה GroupID הכי קטן שיש בהן לפחות שחקן אחד את השחקן שנמצא בשלב (level) הגבוה ביותר (אם יש יותר משחקן אחד באותו level מאותה קבוצה, יהיה זה השחקן בעל ה-playerID הקטן יותר). כל השחקנים יוחזרו במערך ממוינים לפי מזהה הקבוצה GroupID בסדר עולה.
- **דרישות סיבוכיות זמן** : O(numOfGroups+log(k)) במקרה הגרוע, כאשר numOfGroups הוא הפרמטר לפונקציה (כמות הקבוצות הלא ריקות להן יש להחזיר את השחקנים בשלב הכי גבוה) ו-k הוא מספר הקבוצות.
- **מימוש** : נרצה להשתמש ב-full_groups_tree המכיל רק קבוצות עם שחקן אחד לפחות ובכך שבתוך כל צומת של קבוצה בעץ יש עץ נוסף של כל השחקנים בקבוצה הממויינים באותו אופן שהתבקשנו כאן כתוצאה מהאופן שבו הגדרנו את צמתי הצומת.
  נתאר את עיקרי המימוש בשלבים :
  - בדיקה שהקלט שקיבלנו עומד בדרישות הקלט כולל בדיקה האם מספר הקבוצות עליהן צריך לעבור גדול ממספר הקבוצות המלאות והחזרת ערך מתאים אם לא – **O(1).**
  - נעבור על הקבוצות המלאות שהן צמתי העץ full_groups_tree בסיור inorder המאפשר לנו לעבור על הקבוצות בעלות ה- id הקטן ביותר קודם. נחזיק מונה המעודכן להיות מספר הצמתים עליהן עברנו ונעצור את הסיור ברגע שהמונה יהיה שווה ל-numOfGroup . לכל קבוצה מלאה אליה נגיע לסיור נשלוף ב- **O(1)** את השחקן המוביל לפי הדרישות שכן בתוך כל קבוצה ישנו עץ של שחקנים הממויינים לפי הדרישות בשאלה ויש לנו מצביע לצומת הימנית ביותר בעץ שהיא השחקן המוביל בקבוצה.
- **מצב מבנה הנתונים לאחר ביצוע הפעולה** : אין כל שינוי.
- **מדוע אנחנו עומדים בסיבוכיות הזמן הדרושה** : הגישה לצומת הראשונה בסיור inorder בעץ AVL נעשית בסיבוכיות log(k) כאשר k הוא מספר הקבוצות הכולל ולכן לכל היותר מספר הקבוצות המלאות בעץ שכן מדובר בגישה לצומת השמאלית ביותר. לאחר מכן אנחנו מבצעים numOfGroups מעברים ולכן הסיבוכיות במעבר בין הצמתים היא o(numOfGroups) בכל צומת מבצעים מספר סופי של פעולות ב- O(1) כתוצאה מגישה לשדות שמורים ללא צורך בחיפוש. בסך הכל קיבלנו כי הסיבוכיות היא O(numOfGroups+log(k)).

10) **הפעולה:**

void Quit(void **DS)

- **מטרה** : הפעולה משחררת את המבנה.
- **דרישות סיבוכיות זמן** : O(n+k).

- **מימוש**: לאחר בדיקת קלט ב- **O(1)**, קוראים להורס של המחלקה PlayersManager. ההורס של PlayersManager מכיל קריאה להורס של ארבעת העצים שציינו בפתיחה (נשים לב שכל העצים הם מטיפוס *AVLTree ולכן ההורס שלהם זהה). ההורס של העץ מבצע מעבר על צמתי העץ באמצעות סיור postorder שהוצג בתרגול ומבוצע בסיבוכיות לינארית למספר הצמתים בסיור. עבור כל צומת אליה מגיע בסיור הוא מבצע קריאה להורס של הצומת.
  - העץ all_players_tree מכיל צמתים מטיפוס *PlayerNode והעץ levels_tree מכיל צמתים מטיפוס *LevelNode. ההורסים של הצמתים הללו מבצעים מספר סופי וחסום של פעולות שחרור כלומר ההורס הפנימי מבצע פעולות בסיבוכיות **O(1)**. לכן, הרס כל אחד מהעצים מבוצע בסיבוכיות **O(n)** - מעבר על כלל השחקנים בעץ ב- O(n) והרס כל שחקן ב- O(1).
  - גם העץ full_groups_tree וגם העץ all_groups_tree מכילים צמתים מטיפוס *GroupNode אשר כל אחד מהם מכיל בתוכו את עץ השחקנים המשתייכים לקבוצה. נשים לב לכן כי כחלק מפעולת ההריסה של העצים הללו יש לעבור על כל הצמתים בעץ ועבור כל צומת כזאת לדאוג לעבור על העץ שחקנים הפנימי שלה ולהרוס אותו ואז את הצומת.
  - טענה: הרס ה- full_groups_tree, all_groups_tree מבוצע בסיבוכיות **O(n+k)**.
  - נימוק: בהינתן שחקן במבנה שלנו הוא משתייך בדיוק לקבוצה אחת. בהינתן קבוצה במבנה שלנו אז קיים בדיוק צומת אחד לכל היותר בעץ קבוצות המתאר אותה (אולי ריקה ואז אין צומת בעץ הקבוצות המלאות המתאר אותה). הרס של עץ קבוצות המשתמש בסיור PostOrder כולל מעבר על כלל הצמתים (יש k כאלה לכל היותר כמספר הקבוצות) בדיוק פעם אחת ומעבר על כלל השחקנים במבנה (המפוזרים ביניהם בין הקבוצות השונות) בדיוק פעם אחת. סך הכל מעבר על n+k צמתים כאשר פעולת השחרור עצמה נעשית ב-**O(1)** . לכן הסיבוכיות של שחרור עץ קבוצות כלשהו הוא **O(n+k)**.
  - לסיום, מבצעים השמה ל nullptr ל-DS*.

מצב מבנה הנתונים לאחר ביצוע הפעולה: המבנה לא מוגדר ומכיל ערכי זבל.

- **מדוע אנחנו עומדים בסיבוכיות הזמן הדרושה**: פעולת ה-Quit כוללת שתי פעולות:

$$2*O(1) + 2*O(\log(n)) + 4*O(\log(n + k)) = O(\log(n+k))$$

חלק ד' – סיבוכיות המקום של המבנה

דרישת סיבוכיות מקום: $O(n + k)$ במקרה הגרוע, כאשר n הוא מספר השחקנים ו-k הוא מספר הקבוצות.

מדוע אנחנו עומדים בדרישה זו:

במבנה ישנם 4 עצים משני סוגים: עצי קבוצות ועצי שחקנים. נשים לי כי כל קבוצה מלאה מיוצגת בדיוק על ידי שני צמתים במבנה (צומת ב-all_groups_tree וצומת ב-full_groups_tree), כל קבוצה ריקה מיוצגת בדיוק על ידי צומת אחת במבנה שלנו (צומת ב- all_groups_tree) וכל שחקן מיוצג בדיוק על ידי שלושה צמתים במבנה (צומת ב- all_players_tree צומת ב- levels_tree וצומת ב- group_players_tree). סה"כ לכל קבוצה יש להקצות לכל היותר שני צמתים בגודל קבוע כלומר נקבל סיבוכיות מקום O(k), לכל שחקן נקצה בדיוק שלושה צמתים בגודל קבוע, כלומר נקבל סיבוכיות מקום O(n). בסה"כ מבנה הנתונים שלנו בסיבוכיות מקום של O(n+k) מכיוון שכל הקצאה מתאימה לשחקן יחיד או לקבוצה יחידה במבנה הנתונים.

נציין כי קיימות פעולות לאורך הקוד שיש להן סיבוכיות מקום גדולה מ- O(1), נראה כי בכל זאת, בכל שלב לאורך הקוד שלנו סיבוכיות המקום עומדת בדרישות. בפרט, בכל מקום בקוד בו ישנה פעולת חיפוש, הכנסה, הוצאה וסיור מכל סוג שהוא מתבצעות קריאות רקורסיביות כתלות בגובה העץ (כלומר log(n) עבור עצי שחקנים ו- log(k) עבור עצי קבוצות). נעבור בקצרה על הפעולות השונות:

- Init- סיבוכיות מקום O(1) כי יש מספר סופי של הקצאות זיכרון במבנה.
- AddGroup, AddPlayer, RemovePlayer, IncreaseLevel- המשותף לפעולות אלה הוא שמלבד מספר סופי של הקצאות לטובת הכנסת ו/או הסרת שחקן ו/או קבוצה מהמבנה, יש שימוש חוזר בפעולות insert, remove ו- find באופן בלתי תלוי (אין רקורסיה מקוננת), קרי סיבוכיות מקום של O(log(k)+log(n)) במקרה הגרוע כנדרש.
- ReplaceGroup- בפעולה זו ישנה הקצאה של שלושה מערכים בגודל המתאים למספר השחקנים בקבוצות, כלומר הקצאות אלו חסומות על ידי O(n). בנוסף ישנה הקצאה של עץ חדש בגודל של מספר השחקנים בשתי הקבוצות כלומר הקצאה החסומה על ידי O(n). בנוסף על כל זה מתבצעים בפעולה מספר סיורי Inorder שזו סיבוכיות מקום של O(log(n)) במקרה הגרוע.
- GetHighestLevel- שימוש בפעולה find בסיבוכיות מקום של O(log(k)).
- GetAllPlayersByLevel- בפעולה זו ישנה הקצאת זיכרון של מערך בגודל של מספר השחקנים בכל המשחק במקרה הגרוע, כלומר בסיבוכיות מקום של O(n). בנוסף, יש שימוש בפעולה find בעץ הקבוצות וסיור inOrder בעץ השחקנים כלומר סיבוכיות מקום של O(log(k)+log(n)).
- GetGroupsHighestLevel- בפעולה זו יש הקצאת זיכרון של מערך בגודל של מספר הקבוצות שקיבלנו כקלט, כלומר במקרה הגרוע O(k). בנוסף ישנו סיור Inorder על עץ הקבוצות המלאות, כלומר סיבוכיות מקום של O(log(k)) במקרה הגרוע.