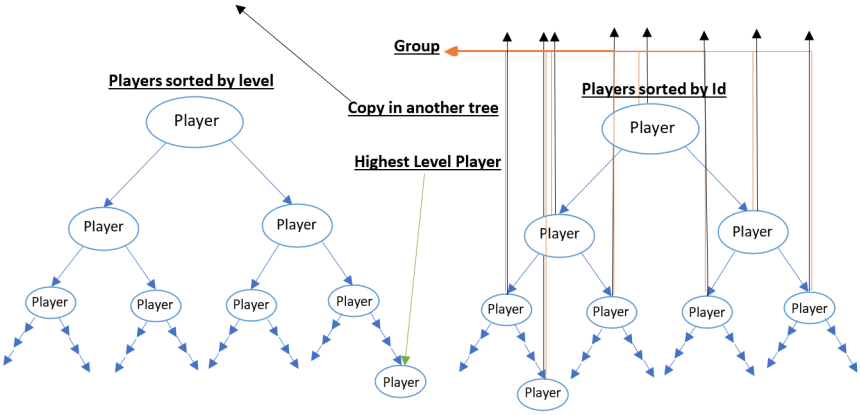
תחילה נציג את מבני הנתונים בהם נשתמש בפתרון התרגיל:

1. ''שחקן'' - מבנה שישמור את המספר המזהה של השחקן את רמתו, את הקבוצה לה הוא שייך, מצביע לעותק שלו הנמצא בעץ אחר ומזהה האם הוא שייך לעץ הממיין אותו לפי השלב או לפי המספר המזהה.
2. ''קבוצה''- מבנה שישמור שני עצים מסוג AVL של שחקניו, אחד לפי המספר המזהה ואחד לפי רמתם, את גודלו (כמות השחקנים בו), מצביע לשחקן שרמתו הגבוהה ביותר, מצביע לעותק של הקבוצה הנמצא בעץ אחר ואת המספר המזהה של הקבוצה עצמה.
3. ''מנהל השחקנים'' - מבנה שישמור שני עצים של קבוצות , אחד של כל הקבוצות ואחד של הקבוצות הלא ריקות וקבוצה ששחקניה הם כל השחקנים במשחק.
4. כמו כן, נממש מבנה מבנה נתונים של עץ AVL גנרי $(template)$ בתור מבנה עזר.

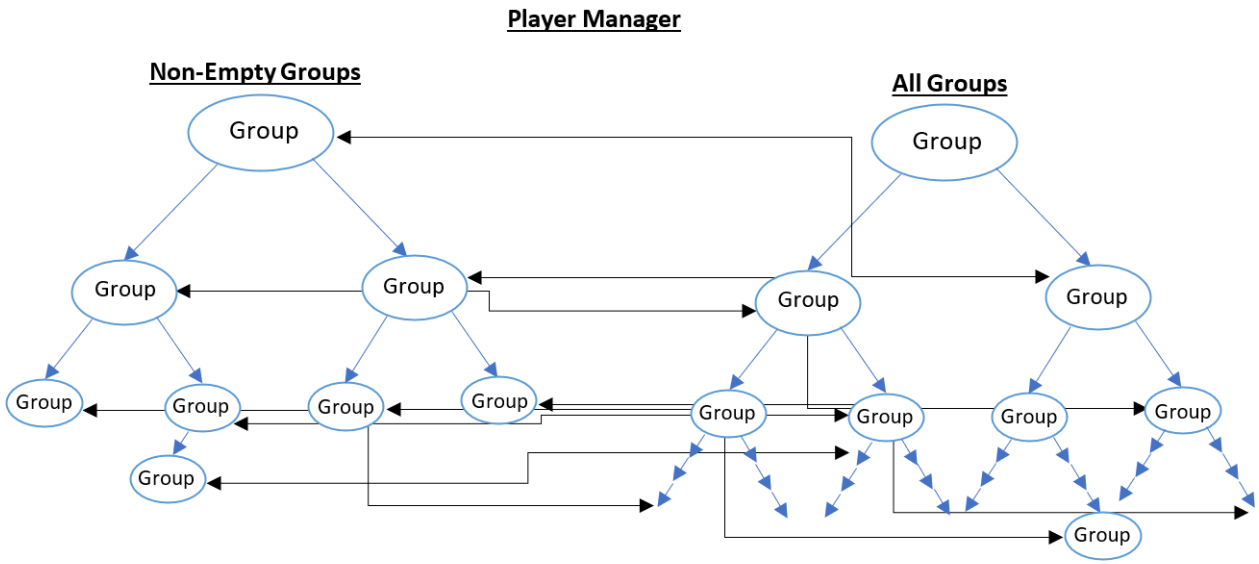
תיאור מפורט של מבנה הנתונים: לכל שחקן קיימים ארבעה עותקים - בקבוצת כל השחקנים ובקבוצותו, נכניס אותו לכל עץ בנפרד (לפי מספר מזהה ולפי רמה). לכל שחקן המזוהה לפי מספרו, מאותחל מצביע לקבוצה בה הוא נמצא במשחק (לא לקבוצת כל השחקנים גם אם שם העותק) ומצביע לעותק שלו המזוהה לפי המספר המזוהה ונמצא בקבוצה השנייה (אם העותק שאנו מסתכלים עליו בקבוצת כל השחקנים, המצביע לשחקן בקבוצה שלו ולהפך). דבר זה מאפשר לנו לתמוך בכל שינוי הקורה באחד העותקים - לדוגמה, עליית שלב או שינוי קבוצה, ולעדכן את העותק השני ובכך לא לפגוע בנכונות התוכן. לא נדאג לעדכן את המצביעים של השחקנים השמורים לפי רמתם וגם לא נסתמך על נכונותם במהלך התוכנית.

כמו כן, לכל קבוצה, במידה והיא לא ריקה, קיים מצביע ששומר על העותק שלה בעץ הקבוצות הלא ריקות. הקבוצות נמצאות בעץ הקבוצות הלא ריקות אינן מאותחל לדבר מלבד המספר המזהה שלהן ומצביע לעותק המקורי של קבוצתן. כך אין חשש לפגיעה בנכונות התוכן (הכול מעודכן בקבוצה המקורית) ובמידה ונרצה משהו מהקבוצות הללו, כמו גישה לכל השחקנים הגבוהים ביותר במשחק, נעבור דרך המצביע לקבוצות המקוריות. נשים לב שבעץ הקבוצות הלא ריקות יש לכל היותר ככמות השחקנים, דבר זה יעזור לעמוד בדרישות הסיבוכיות בהמשך.

מצורפות תמונות הממחישות כיצד יראה מבני הנתונים הכולל של מנהל השחקנים. התמונה הראשונה מתארת את מבנה הקבוצה והשנייה את עצי הקבוצות.



בתמונה מתאורת קבוצה מעץ כל הקבוצות הלא ריקות. החצים הכתומים הם מהשחקנים למבנה עצמו של הקבוצה, הירוק הוא לשחקן הגבוהה ביותר והחצים השחורים מצביעים לעותקים המתאימים מחוץ לקבוצה. ניתן לשים לב כי שני העצים ממויינים בקבוצה שונה ועל כן המבנה שלהם יהיה אולי שונה.



בתמונה מעלה אפשר לראות את המצביעים בין הקבוצות הלא ריקות לכל הקבוצות. בניגוד לעצים בתוך קבוצה, שני העצים פה ממויינים לפי המספר המזהה של הקבוצות. לא יהיו בעץ הקבוצות הלא ריקות עותקים של קבוצות ריקות.

כעת נראה שכל אחת מהפעולות עומדת בדרישות סיבוכיות הזמן הנדרשת: נשתמש בסימונים מדף ההנחיות לכמות הקבוצות וכמות השחקנים.

* כל פעולות הכנסה, הוצאה ומציאת איבר בכל אחד מהעצים היא ב- $O(log(m))$ כאשר m היא כמות האיברים בעץ. כמו כן, סיבוכיות הזמן של פונקציות סיור על העץ היא $O(m)$. נסתמך על כך בהמשך ההסבר, הפונקציות בעץ ממוממשות בצורה שעומדת בדרישות הסיבוכיות הללו.

1. $void^{*}init()$ הפעולה אינה מקבלת קלט ולכן בכל מקרה סיבוכיות הזמן שלה היא $O(1)$. הפעולה יוצרת שני עצים ריקים של קבוצות, וקבוצה ריקה של השחקנים במשחק ומחזירה את מנהל השחקנים במידה ואין בעיה של הקצאת זיכרון.

2. $StatusType\ AddGroup(void^{*}\ DS, int\ GroupID)$: הפעולה יוצרת קבוצה חדשה ומכניסה אותה לעץ כל הקבוצות. סיבוכיות הזמן היא הסיבוכיות הדרושה למציאת מיקום הקבוצה בעץ הקבוצות והיא $O(log(k))$ כפי שנלמד בהרצאה. במקרה של ערך הכנסה לא תקין, במקרה של הקצאת זיכרון או במקרה שבו קיימת כבר קבוצה עם המזהה הנ"ל, תחזיר הפונקציה ערך מתאים בהתאם לדרישות ומבנה הנתונים אינו ישתנה.

3. $StatusType\ AddPlayer(void^{*}\ DS, int\ PlayerID, int\ GroupID, int\ level)$: הפעולה יוצרת שחקן חדש, מוצאת את הקבוצה שאליה יש לצרפו ומכניסה את השחקן לקבוצה הנ"ל ולקבוצה של כל השחקנים במשחק. לפני הכנסת השחקן, נייצר שני עותקים שלו שיצביעו אחד על השני, אחד יצביע יכנס לקבוצת כל השחקנים והשני לקבוצה של השחקן. כמו כן, במידה וקבוצת השחקן הייתה ריקה, נייצר קבוצה חדשה שתייצג את קבוצתו, נקשר בין שתי הקבוצות ונכניס את החדשה לעץ הקבוצות הלא ריקות. כך נדע לגשת אליה או ממנה במידת הצורך. במידה ויש צורך, הקבוצה תעדכן את המצביע לשחקן הגבוהה ביותר ברמתו. סיבוכיות הזמן של מציאת מיקום השחקן בקבוצה של כל השחקנים ובקבוצתו היא $O(log(n))$ ומציאת הקבוצה לה יש להוסיפו בעץ כל הקבוצות היא $O(log(k))$ והכנסת הקבוצה החדשה לעץ הקבוצות הלא ריקות גם היא ב- $O(log(k))$ במידת הצורך ולכן סיבוכיות הזמן הכוללת של הפעולה היא $O(log(n) + log(k))$. במידה ונכנס ערך לא תקין, השחקן כבר נמצא במשחק, הקבוצה שאליה הוא אמור להיכנס לא קיימת או של הקצאת זיכרון שלא צלחה, יוחזר ערך מתאים ומבנה הנתונים אינו ישתנה.

4. $StatusType\ RemovePlayer(void^{*}\ DS, int\ PlayerID)$: הפעולה תמצא את השחקן בקבוצת כל השחקנים לפי המספר המזהה שלו, תיגש

לקבוצתו דרך המצביע אותו מחזיק השחקן ותסיר אותו משתי הקבוצות. כמו כן, במידה וקבוצתו ריקה כעת, נסיר את העותק שלה מעץ הקבוצות הלא ריקות. יש לכל היותר n כנ"ל (ככמות השחקנים במשחק) ולכן הסרתה תבוצע ב- $O(log(n))$. במידה ויש צורך, הקבוצה תעדכן את המצביע לשחקן הגבוהה ביותר ברמתו. סה"כ, חוץ ממציאת השחקן בקבוצה של כל השחקנים ואולי הסרת עותק הקבוצה מעץ הקבוצות הלא ריקות, כל הפעולות קבועות ולא תלויות בקלט ולכן סיבוכיות הזמן הכוללת של הפעולה היא $O(log(n))$. במקרה של ערך הכנסה לא מתאים, השחקן אינו במשחק, או במקרה של בעיה בהקצאת זיכרון, יוחזר ערך מתאים ומבנה הנתונים אינו ישתנה.

5. $StatusType\ ReplaceGroup(void^*\ DS, int\ GroupID, int\ ReplacemenetID)$: ראשית נסביר איך עובדת פונקציית עזר של העץ על מנת להסביר בקלות פונקציה זו - הפונקציה $combineTrees$ מקבלת שני עצים, מסיירת בהם "לפי סדר" - $inorder$ יוצרת שני מערכים עם הערכים של כל אחד מהעצים, ממזגת ביחד למערך אחד המכיל את כל האיברים בכך שהיא עוברת לפי סדר על שני המערכים ומכניסה בכל פעם את הקטן בין השניים (פעולה זו קורת ב-$O(n_1 + n_2)$ כאשר בעץ i יש n_i איברים), הופכת את העץ אליו נכניס את הערכים להיות בעל גודל מתאים בכך שהופכת אותו לעץ כמעט שלם מהגודל והגובה המתאימים (גם פעולה זו קורת ב- $O(n_1 + n_2)$) ולאחר מכן מסיירת על העץ $inorder$ ומכיניסה את הערכים מהמערך המשולב למקומות המתאימים בעץ (גם פעולה זו, כפעולת סיור, קורת ב- $O(n_1 + n_2)$).

כעת נסביר את פעולת מיזוג הקבוצות: פעולה זו ממזגת את שני העצים של שתי הקבוצות בהתאמה (שחקנים לפי מספר מזהה עם שחקנים לפי מספר מזהה ושחקנים לפי שלב עם שחקנים לפי שלב), לאחר מכן מעדכנת את השחקן עם השלב הגבוהה ביותר במידת הצורך, עוברת את כל שחקני העץ הממויינים לפי מספר המזהה לפי סדר, מעדכנת את קבוצתם להיות הקבוצה בה כעת כולם נמצאים (כמובן שנעדכן גם את העותקים שלהם בקבוצת כל השחקנים) ופעולה זו, כפעולת סיור על העצים של הקבוצה קורת ב- $O(n_1 + n_2)$ כאשר $n_1 + n_2$ היא כמות השחקנים בשתי הקבוצות יחד, כלומר כמות השחקנים כעת בקבוצה לבסוף, הפעולה תסיר את הקבוצה שהוחלפה מעץ כל הקבוצות ומעץ הקבוצות הלא ריקות במידת הצורך, כמו גם תוסיף את עותק של הקבוצה המחליפה לעץ זה במידה ולא היה שם קודם. במידה ויש צורך, הקבוצה המחליפה תעדכן את המצביע לשחקן הגבוהה ביותר ברמתו. במידה ואחד מערכי הקלט לא תקין, אחת הקבוצות לא נמצאת או במקרה של בעיית זיכרון, יוחזרו ערכים מתאימים והמבנה לא ישתנה. סה"כ, סיבוכיות הזמן של הפעולה היא מציאת שתי הקבוצות בעץ הקבוצות - $O(log(k))$, הכנסה הוצאה מעצי הקבוצות גם כן ב- $O(log(k))$ ומיזוג הקבוצות - $O(n_1 + n_2)$ ולכן סיבוכיות הזמן הכוללת היא $O(n_1 + n_2 + log(k))$.

6. $StatusType\ IncreaseLevel(void^*\ DS, int\ PlayerID, int\ LevelIncrease)$: הפעולה תמצא את השחקן לפי מספרו המזהה בקבוצה של כל השקנים, לאחר מכן תיגש דרכו לקבוצה שלו, שם תמצא אותו בתוך שחקני הקבוצה בשני העצים שלה, לאחר מכן תעדכן את שלבו בהתאם לקלט בעץ שבו השחקנים ממויינים לפי השלב (ע"י הסרתו מהעץ, עדכונו ואז החזרתו מחדש בערך המתאים), ואז תעדכן את שלב השחקן בעץ שבו ממויינים השחקנים לפי מספרם מזהה. לאחר מכן, תעדכן הפעולה גם בקבוצת כל השחקנים באותו האופן. במידה ויש צורך, הקבוצה תעדכן את המצביע לשחקן הגבוהה ביותר ברמתו. סה"כ הפעולה היחידה שתלויה בקלט היא מציאת השחקן (בקבוצת כל השחקנים ובקבוצתו) ופעולה זו מבצעת ב- $O(log(n))$. לכן, סיבוכיות הזמן הכוללת של הפעולה היא $O(log(n))$. במידה וערך הכנסה לא תקין/לא קיים שחקן עם מספר מזהה כנ"ל במשחק או במקרה של בעיית זיכרון, יוחזר ערך מתאים ומבנה הנתונים לא ישתנה.

7. $StatusType\ GetHighestLevel(void^*\ DS, int\ GroupId, int^*\ PlayerID)$: נחלק את התנהלות הפונקציה לשתיים לפי הקלט ב- $GroupId$
$GroupId > 0$: במקרה זה, הפונקציה תמצא את הקבוצה בעץ הקבוצות ותחזיר את המספר המזהה של השחקן ששמור אצלה עם השלב הגבוהה ביותר. זמן ריצת התוכנית תלוי במיקום הקבוצה בעץ הקבוצות שכן שאר הפעולות הן קבועות ולא תלויות בקלט, לכן במקרה זה סיבוכיות הזמן היא $O(log(k))$.
$GroupId < 0$: הפעולה תיגש לשחקן ברמה הגבוהה ביותר דרך קבוצת כל השחקנים ותחזיר את מספרו המזהה. סיכוביות הזמן היא $O(1)$ במקרה זה. במידה והקלט אינו תקין, הקבוצה הרצויה ריקה/אין בכלל שחקנים, יוחזר ערך מתאים ומבנה הנתונים לא ישתנה.

8. $StatusType\ GetAllPlayersByLevel(void^*\ DS, int\ GroupID, int^{**}\ GroupID, int^*\ numOfPlayers)$: נחלק את התנהלות הפונקציה לשתיים לפי הקלט ב- $GroupId$:
$GroupId > 0$: במקרה זה, הפונקציה תמצא את הקבוצה בעץ הקבוצות, תמשתמש בפונקציית עזר של העץ, ותקבל את כל השחקנים ממויינים לפי השלב שלהם בסדר עולה, תייצר מערך חדש אליו תכניס את השחקנים בסדר הפוך מזה שבו קיבלה אותם במערך מהעץ. פעולת העזר בעץ היא סיור ושמירת הערכים

והיא קורת ב- $O(n_{group})$ כמו גם אתחול המערך החדש אותו היא מחזירה. מציאת הקבוצה בעץ היא ב- $O(log(k))$ ולכן סה"כ סיבוכיות הזמן של הפונקציה היא $O(log(k) + n_{group})$.

$GroupId < 0$: הפעולה תיגש לקבוצת כל השחקנים, תשתמש באותה פונקציית עזר של העץ, תקבל מערך של כל השחקנים ממויין לפי השלב מהקטן לגדול, תצייר מערך חדש בסדר יורד ותחזיר אותו. סיכוביות הזמן היא $O(n)$ במקרה זה. במשתנה $numOfPlayers$ תוחזר כמות השחקנים בקבוצה או בכל השחקנים כתלות ב- $GroupId$. במידה והקלט אינו תקין, הקבוצה הרצויה ריקה/אין בכלל שחקנים, או במקרה של בעיית זיכרון, יוחזר ערך מתאים לפי הדרישות ומבנה הנתונים לא ישתנה.

9. $StatusType\ GetGroupsHighestLevel(void^{*}DS, int\ numOfGroups, int^{**}\ Players)$: הפעולה תעבור על כל הקבוצות הלא ריקות דרך עץ הקבוצות הלא ריקות, היא תתחיל מהקבוצה הנמוכה ביותר (אותה תמצא דרך פעולות העץ בסיבוכיות של $O(log(k))$) ומשם תעבור למצביע הבא בכל פעם ותיגש לשחקן ברמה הגבוהה ביותר בכל קבוצה (דרך העותק שלה בעץ כל הקבוצות) ועד שסכימת השחקנים תסתכם לערך של $numOfGroups$. במידה ויהיו פחות קבוצות לא ריקות מערך זה, יוחזר ערך החזרה מתאים, כמו כן, במידה והקלט אינו תקין, או במקרה של בעיית זיכרון, יוחזר ערך מתאים לפי הדרישות ומבנה הנתונים לא ישתנה. הפונקציה רצה רק על הקבוצות הלא ריקות אחרי גישה לקבוצה בעלת המזהה הנמוך ביותר, ומכל אחת מהם עושה מספר קבוע של פעולות ולכן סה"כ סיבוכיות הזמן של הפוקנציה היא $O(log(k) + numOfGroups)$.

10. $void\,Quit(void^{*}DS)$: הפעולה תשחרר את הזיכרון של עצי הקבוצות ושל קבוצת כל השחקים דרך כאשר שחרור הזיכרון נעשה ברובו במחיקת העצים שם שמורים השחקנים בקבוצות והקבוצות בעץ. כל עץ נמחק דרך שימוש בסיור $postorder$ ובכך נמחק בסיבוכיות של $O(m)$ כאשר m היא כמות האובייקטים בעץ . כל שחקן נשמר 4 פעמים וכל קבוצה פעמיים לכל היותר (פלוס קבוצת כל השחקנים) ולכן סיבוכיות הזמן של השחרור הכולל היא $O(4n + 2k + 1) = O(n + k)$.

מבנה הנתונים עומד ברשיות סיבוכיות המקום כי:

1. עבור כל שחקן, יש בדיוק ארבע חוליות השומרות את המבנה שלו (תחת הקבוצה בה הוא נמצא ותחת קבוצת כל השחקנים כאשר בכל אחד מהם יש שני עותקים שונים), כל אחת מהן שומרות כמות קבועה (עד כדי שני מצביעים) של זיכרון ולכן כלל הזיכרון הנשמר עבור השחקנים חסום מלמעלה ע"י קבוע (התלוי בכמה מקום נשמר עבור כל מבנה של שחקן) כפול כמות השחקנים. בסה"כ נקבל שסיבוכיות המקום עבור השחקנים בלבד היא $O(n)$.

2. יש סה"כ $2k + 1$ מבנים של קבוצות הנשמרים לכל היותר (שניים לכל קבוצה + קבוצה של כל השחקנים) כאשר עבור כל אחד מהם נשמרת כמות קבועה של מקום (כאשר את השחקנים עבור כל קבוצה ספרנו בנפרד, ובקבוצות הלא ריקות נשמור בסה"כ שני ערכים) ולכן סה"כ סיבוכיות המקום עבור הקבוצות היא $O(k)$.

3. שאר רכיבי הזיכרון בהם נשתמש הם קבועים (מצביעים, מספרים המציינים את גודל הקבוצות וכו, הם זניחים ביחס ל-k, n גדולים) ועל כן לא ישפיעו על סיבוכיות המקום. כמו כן, מבני נתונים זמניים כגון מערכי עזר בהם נשתמש, אינם ישפיעו על סיבכיות המקום הכוללת של התרגיל שכן הם יהיו גם כן במקרה הגרוע ביותר ב-$O(n)$ והם ימחקו בד"כ בסוף הפעולה אלא אם כן נתבקש להחזירם, בפוקנציות $GetAllPlayersByLevel$ ו- $GetGroupsHighestLevel$.

לכן, סה"כ סיכויות המקום של המבנה כולו היא $O(n + k)$ כנדרש.