מבני נתונים 234218 חורף תשפ"ג
גיליון רטוב מספר 1

מגישים:

דניאל בן שלוש – 316468685

עמית - 207422650

הסבר כללי למבנה:

ראשית נתחיל בתיאור של טיפוסי הנתונים בהם נשתמש:

Player :

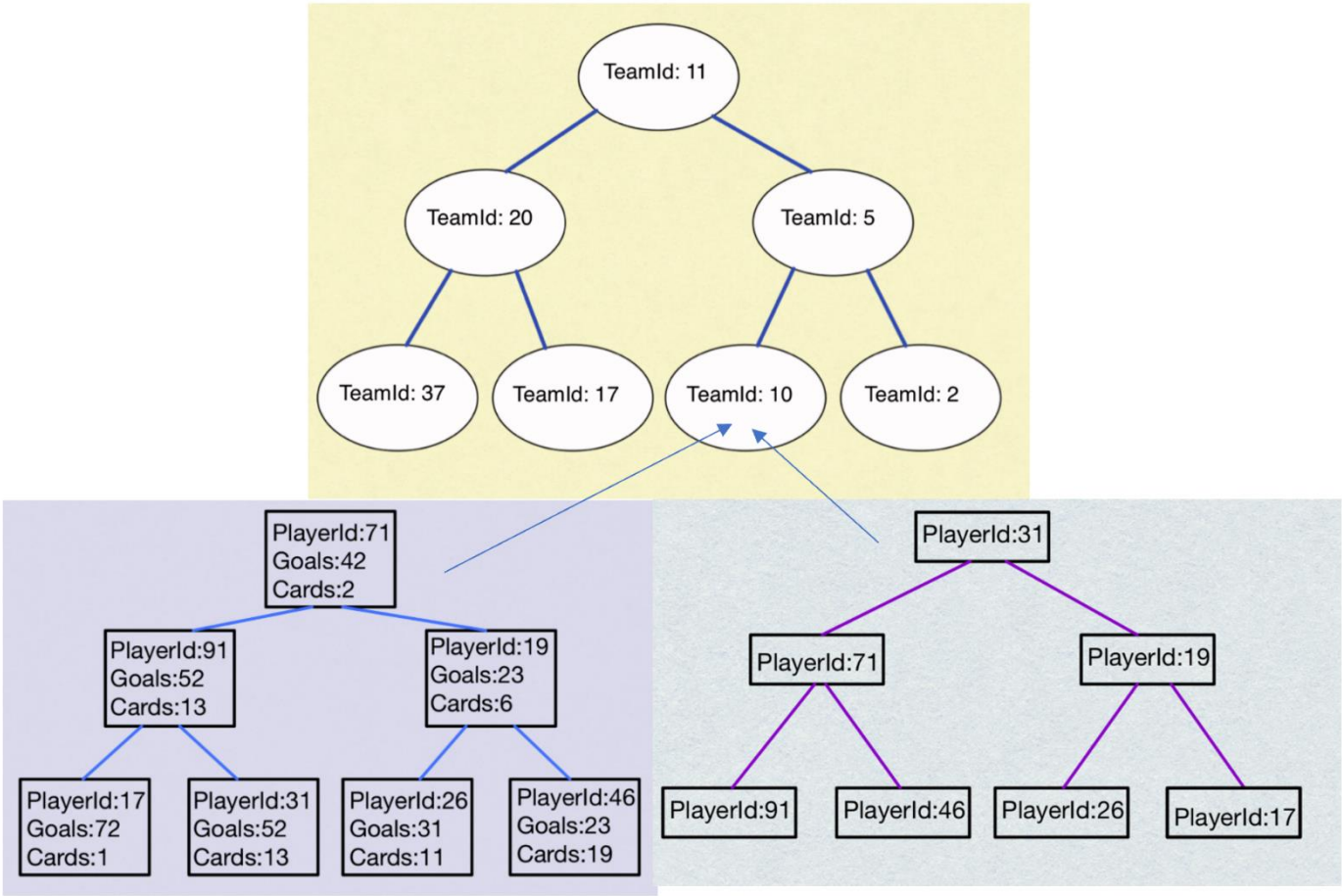
- Int ID – מספר מזהה של שחקן
- Int Goals – מספר הגולים של אותו שחקן
- Int Cards – מספר הכרטים של אותו שחקן
- Int initialGamesPlayed – מספר המשחקים שאותחל לשחקן בעת יצירה או כניסה לקבוצה.
- Int gamesPlayedAtJoin – הערך של הgamesCounter של הקבוצה ברגע שהצטרף אליה
- Bool goalkeeper – האם השחקן יכול לשחק כשוער או לא.
- Team* team – פויטר לקבוצה
- Int teamId – מספר הזהות של הקבוצה

Team :

- Int ID – מספר מזהה של הקבוצה
- Int goalsCounter – סכום של כל הגולים של כל השחקנים בקבוצה (מתעדכן בעת הכנסת שחקן ויציאת שחקן)
- Int cardsCounter – סכום של כל הכרטיסים של כל השחקנים בקבוצה (מתעדכן בעת הכנסת שחקן ויציאת שחקן)
- Int playerCounter – סופר כמה שחקנים יש בקבוצה (מתעדכן בהכנסה או הוצאה של שחקן)
- Int goalkeeperCounter – סופר כמה שוערים יש (מתעדכן בהכנסה או הוצאה של שוער)
- Int gamesCounter – סופר כמה משחקים הקבוצה שיחקה
- AVLtree<PlayerKey*> playersByScore נגדיר יחס סדר הדרגתי לפי הרמות הבאות:
  1. לפי השחקן בעל מספר הגולים הגדול ביותר
  2. במקרה של שיוויון במספר הגולים נעבור לבדוק למי יש הכי פחות כרטיסים.
  3. במקרה של שיוויון גם במספר הגולים וגם במספר הכרטיסים נבדוק לפי מספר המזהה של כל שחקן מהגדול לקטן.
- AVLtree<Player*> - עץ מסודר לפי מספר הזהות של השחקנים.

מבנה הנתונים world_cup_t יכל בתוכו:

- Int numberOfPlayers – משתנה מקומי שיאמר מהו מספר השחקנים הכולל בכל זמן נתון.
- Int numberOfTeam – משתנה מקומי שיאמר כמה קבוצות יש בכל זמן נתון.
- AVLtree<Team*> teams – עץ שיחזיק את כל הקבוצות וידרג לפי מספר הזהות של הקבוצות.
- AVLtree<Player*> playersId -עץ שמכיל את כל השחקנים(פוינטר לכל שחקן) ומדורג לפי מספר המזהה של כל שחקן.
- AVLtree<Player*> playersByScore -עץ שמכיל את כל השחקנים((פוינטר לכל שחקן) נגדיר יחס סדר הדרגתי לפי הרמות הבאות: נסדר לפי מספר הגולים של השחקנים במקרה בו לכמה שחקנים אותו מספר גולים נסדר אותם לפי מספר הכרטיסים שלהם (זה שיש לו הכי פחות כרטיסים הוא הכי טוב) ובמקרה בו לשחקנים שונים יש אותו מספר גולים וכרטיסים נסדר אותם לפי המספר המזהה.



ברקע הצהוב מתואר עץ של קבוצות מסודר לפי מספר מזהה. לכל איבר בעץ הנל יש קישור לשני עצים נוספים עבור השחקנים בכל קבוצה.

העץ ברקע הסגול מכיל את כל השחקנים בקבוצה מסודר לפי הדרישות של topScorer.

העץ ברקע הירוק מכיל את כל השחקנים בקבוצה ומסודר לפי המספר המזהה של השחקנים.

דרישות מיוחדות מהעץ:

בכל עץ (במיוחד בעץ של ה-score) יש צורך בשדה שמחזיק פוינטר לנוד במקום הכי ימני- זאת על מנת להחזיר ב(O(1 את האיבר המקסימלי. כדי לשמור כזה פוינטר ולעמוד בתנאים עלינו לבצע מספר פעולות בכל הכנסה והוצאה של איבר מהעץ.

כאשר מכניסים איבר חדש לעץ – בעט ההכנסה יש להשוות את הKey של האיבר החדש עם האיבר שאליו אנו מצביעים. אם נקבל שלאיבר החדש Key גדול יותר נעדכן את הפוינטר, אחרת לא נשנה.

כאשר מוציאים איבר יש שני אפשרויות:

1. הוצאנו את האיבר המקסימלי בעץ – נעדכן את הפוינטר להצביע לאבא של האיבר שמוציאים.
2. הוצאנו איבר שונה מהאיבר המקסימלי – אין צורך לעדכן .

כל הפעולות הנל לוקחות (O(1 ולכן לא פוגעות בסיבוכיות.

**כדי לחסוך מקום- עבור כל הפונקציות : לפני כל פעולה בודקים שה input-שהתקבל אכן תקין ותואם את הציפיה שלנו לוקח (O(1

הסבר פרקטי לכל פונקציה:

1. world_cup_t()

בפועל:

מאתחל עץ של קבוצות מסודר לפי ID ב-$O(1)$.

מאתחל עץ של שחקנים מסודר לפי score ב- $O(1)$.

מאתחל עץ של שחקנים מסודר לפי ID ב- $O(1)$.

2.virtual ~world_cup_t ()

בפועל:

צריך למחוק את המבנה שהקצנו ב()World_cup_t . למעשה צריך לשחרר את כל העצים ולמחוק את כל הקבוצות והשחקנים שמוכלים במבנה. מיכוון שיש n שחקנים ו-k קבוצות נקבל שהסיבוכיות היא $O(n+k)$.

3. StatusType add_team(int teamId,int points)

בפועל:

ראשית נרצה לוודא שהערכים שקיבלנו תקינים (לוקח $O(1)$. במקרה וקיבלנו ערכים תקינים נרצה לוודא שאין קבוצה קיימת עם המספר המזהה שהתקבל- לכן נשתמש בפונקציה של העץ מסוג treeStatusType אשר תבצע חיפוש על עץ הקבוצות (בעל K צמתים) לפי ה-Id שמקבלים , אם נמצא תחזיר true ואנחנו בהתאם נחזיר failure חיפוש כזה לוקח $O(\log(k))$

במקרה והכל תקין נוסיף את הקבוצה:

נייצר משתנה team חדש נעדכן את כל ה-counters ל- 0 ואת כל השדות הנחוצים לפי מה שנותנים לנו ונכניס אותו לעץ הקבוצות , הוספת איבר לעץ קיים עם K הקבוצות לוקח $O(\log(k))$

סהכ קיבלנו $2 \cdot \log(k)$ לכן מדובר ב-$O(\log(k))$.

4.StatusType remove_team(int teamId)

בפועל:

ראשית בודקים אם אם *teamId<=0* אם כן מחזירים *INVALID_INPUT* סהכ $O(1)$.

אחרת קיבלנו ערך תקין : נרצה לוודא שיש קבוצה עם המספר המזהה שהתקבל ורק אם כן תבצע את הפעולה- לכן מימשנו פונקציה של העץ להסרת איבר כך שהיא קודם מוודא שאכן האיבר קיים , במקרה בו האיבר אינו קיים היא מחזירה שגיאת fail, חיפוש הקבוצה לוקח log(K) .

אם היא מצאה נחפש אותה בעץ הקבוצות וניגש לdata של הnode נקרה לפונקציה מה-class של הteam שתגיד האם הקבוצה ריקה (בודקים עם עץ השחקנים של הקבוצה ריק כלומר עם פויינטר לשורש מצביע ל-null לוקח O(1)אם היא לא ריקה נחזיר failure , אם הקבוצה ריקה נסיר אותה מהץ ולאחר מכן נקרא להורס של ה class ונמחק אותה. כל זה יקח $\log(k)$

סהכ נקבל $O(\log(k))$

5.StatusType add_player(int playerId,int teamId,int gamesPlayed,int goals,int cards,bool goalKeeper)

בפועל:

אם הקלט תקין- נרצה לוודא שאין במערכת שחקן עם המספר המזהה שהתקבל- לכן נשתמש בפונקציה של העץ להוספת איבר מסוג treeStatusType אשר תבצע חיפוש על עץ השחקנים (בעל n צמתים) לפי ה-Id שמקבלים , אם תמצא תחזיר הודעת fail ואנחנו בהתאם נחזיר failure חיפוש כזה לוקח $O(\log(n))$

אחרת לא נמצא לכן נרצה לוודא שיש קבוצה עם המספר המזהה שהתקבל- לכן נשתמש בפונקציה של העץ לחיפוש מסוג treeStatusType אשר תבצע חיפוש על עץ הקבוצות (בעל K צמתים) לפי ה-Id שמקבלים , אם לא תמצא תחזיר fail ואנחנו בהתאם נחזיר failure חיפוש כזה לוקח $\log(k)$.

כעת במקרה בו אין עוד שחקן עם אותו מספר מזהה וקיימת קבוצה עם המספר המזהה המתאים נרצה להוסיף את השחקן.

הוספת שחקן לאחר בדיקות:

ראשית נייצר שחקן חדש עם כל הנתנוים שהתקבלו , נרצה להוסיף אותו לעץ השחקנים של הid ו-score הכללי זה יקח $2 \cdot \log(n)$

נחפש את הקבוצה אליו הוא משוייך בעץ הקבוצות – לוקח $\log(k)$

בעת הוספת שחקן לקבוצה , לוקחים את ערך הגולים שיש לשחקן ומוסיפים אותו לסכום הגולים שבקבוצה , אותו דבר עבור הכרטיסים.

אם הוא שוער מעלים את מס השוערים של הקבוצה ב1 , ובכל מקרה מעלים את מספר השחקנים של הקבוצה ב-1 לוקח- $O(1)$.

כעת נרצה להוסיף את השחקן לעצים של הקבוצה שני עצים לכן $2 \cdot \log(n_{team})$ כאשר $n_{team} \leq n$ לכן $2 \cdot \log(n_{team}) \leq 2 \cdot \log(n)$

סה"כ:

$$2 \cdot \log n_{team} + 3 \cdot \log n + 2 \cdot \log(k) \sim O(\log k \ + \log n)$$

6.StatusType remove_player(int playerId)

אם הקלט תקין נרצה לוודא שאכן קיים שחקן עם המספר המזהה הנל - לכן נשתמש בפונקציה של העץ לחיפוש מסוג treeStatusType אשר תבצע חיפוש על עץ השחקנים (בעל n צמתים) לפי ה-Id שמקבלים , אם לא תמצא תחזיר fail ואנחנו בהתאם נחזיר failure חיפוש כזה לוקח $O(\log(n))$

במקרה בוא היא מצאה והכל תקין נעשה חיפוש בעץ *playerId* אחר השחקן ניגש לשדה *data* של הנוד שמחזיר פוינטר לשחקן , ניגש דרך שדה *team* (שמחזיק פוינטר לקבוצה אליו הוא משוייך) לקבוצה של השחקן נוריד מסכום של הכרטיסים והגולים את הערכים של השחקן (כדי לשמור אותם עדכניים) נוריד את מספר השחקנים בקבוצה ב-1 , נבדוק האם השחקן שוער , אם כן נוריד את מספר השוערים בקבוצה ב-1 .

לאחר מכן נרצה לעשות הסרה של השחקן משני עצי השחקנים של הקבוצה – לוקח

$$2 \cdot \log n_{team} \leq 2 \cdot \log n$$

לאחר מכן נרצה להסיר את השחקן מהעצים הכללים של השחקנים לוקח $2 \cdot \log n$

סהכ $\log(n) + \ 2 \cdot \log n_{team} + 2 \cdot log\, n \ \sim O(\log(n))$

7.StatusType update_player_stats(int playerId,int gamesPlayed,int scoredGoals,int cardsReceived)

בפועל:

אם הקלט תקין נרצה לוודא שאכן קיים שחקן עם המספר המזהה הנל - לכן נשתמש בפונקצית חיפוש של העץ מסוג treeStatusType אשר תבצע חיפוש על עץ השחקנים (בעל n צמתים) לפי ה-Id שמקבלים , אם לא תמצא תחזיר fail ואנחנו בהתאם נחזיר failure חיפוש כזה לוקח $O(\log(n))$

במקרה בוא היא מצאה והכל תקין:

נעשה חיפוש בעץ *playerId* אחר השחקן ניגש לשדה *data* של הנוד שמחזיר פוינטר לשחקן ואז ניגש דרך השדה *team* נסיר את השחקן מעץ ה*scor* לפני עדכון השדות- לוקחת $\log(n_{team})$

בנוסף עם הנתנוים של השחקן לפני העדכון נרצה לעשות חיפוש של ה-*node* שלו בעץ של *playerScore* הכללי ולהסיר אותו – לוקח $\log n$

נחזור לעדכן את השחקן על ידי הוספה לשדה הגולים את הערך של *scoredGoals* ולשדה הכרטיסים את הערך *cardsReceived*, כמו כן נרצה לעדכן את השדה initialGamesPlayed ולהוסיף לו *gamesPlayed*.

לאחר מכן ניגש שוב דרך שדה *team* (שמחזיק פוינטר לקבוצה אליו הוא משוייך) לקבוצה של השחקן נרצה להוסיף לסכום הגולים של הקבוצה את *scoredGoals* ולסכום הכרטיסים את *cardsReceived*. בנוסף נרצה להכניס את השחקן עם הנתונים החדשים לעץ *score* של הקבוצה

לבסוף אחרי כל העדכונים נרצה להוסיף את השחקן חזרה לעץ *playerScore* – לוקח $\log n$

סהכ : $2 \cdot \log(n_{team}) + 4 \cdot \log(n) \sim O(\log(n))$

8.StatusType play_match(int teamId1,int teamId2)

בפועל:

כעת נרצה לוודא שאכן קיימות 2 קבוצות עם מס מזהה כפי שנתון לשם כך נשתמש בפונקציה של העץ מסוג treeStatusType אשר תבצע חיפוש על עץ קבוצות (בעל K צמתים) לפי ה-Id שמקבלים , אם לא תמצא תחזיר fail ואנחנו בהתאם נחזיר failure חיפוש כזה לוקח $2 \cdot \log(k)$. במקרה בו אכן קיימות 2 קבוצות כאלו נבצע חיפוש בעץ עבור כל קבוצה שיחזיר לנו פויינטר לקבוצה – לוקח $2 \cdot \log(k)$, כעת עבור כל קבוצה בנפרד נוודא שהיא אכן יכולה לשחק (כלומר יש לה שוער ויתר מ11 שחקנים) על ידי קריאה לפונקציה מהclass

אם שתיהן יכולות לשחק – קיימת פונקציה ב class של הקבוצה שמכינה עבור הקבוצה את התוצאה הסופית של הנוסחא, משווים בין שתי התוצאות

במקרה בו אין שיוויון: עבור הקבוצה עם הערך היותר גדול – היא תוגדר כמנצחת לכן הניקוד שלה יעלה ב-3 והניקוד של השנייה לא תשתנה

במקרה של שיוויון : הניקוד של שתי הקבוצות עולה ב-1. עבור שניהם ה-counter של המשחקים עולה ב-1 . סהכ $O(\log(k))$.

9.output_t<int> get_num_played_games(int playerId).

בפועל:

אם הקלט תקין נרצה לוודא שאכן קיים שחקן עם המספר המזהה הנל - לכן נשתמש בפונקצית חיפוש של העץ מסוג treeStatusType אשר תבצע חיפוש על עץ השחקנים (בעל n צמתים) לפי ה-Id שמקבלים , אם לא תמצא תחזיר fail ואנחנו בהתאם נחזיר failure חיפוש כזה לוקח $O(\log(n))$

במקרה בו קיים והכל תקין נחפש את השחקן בעץ playerId כך שיוחזר לי פוינטר לשחקן, מהשדה team נשיג את ערך הgamesCounter של הקבוצה , נחסר מערך זה את הערך שמוכל ב- gamesPlayedAtJoin נוסיף את התוצאה של החיסור לערך שמוחזק ב- initialGamesPlayed ונחזיר את התוצאה.

סהכ $O(\log(n))$

10.output_t<int> get_team_points(int teamId)

בפועל:

אם קיבלנו ערך תקין : נרצה לוודא שיש קבוצה עם המספר המזהה שהתקבל- לכן נשתמש בפונקצית חיפוש של העץ מסוג treeStatusType אשר תבצע חיפוש על עץ הקבוצות (בעל K צמתים) לפי ה-Id שמקבלים , אם לא תמצא תחזיר fail ואנחנו בהתאם נחזיר failure חיפוש כזה לוקח $\log(k)$.

במקרה בו יש קבוצה כזו נחפש אותה בעץ הקבוצות וניגש לdata של הnode נקרא לפונקציה מה-class של הteam אשר תחזיר את הpoints של הקבוצה - לוקח $\log(k)$.

סהכ: $O(\log(k))$.

11.StatusType unite_teams(int teamId1,int teamId2,int newTeamId)

בפועל:

מחפשים בעץ קבוצות בעלות מספרים מזהים נתונים. אם לא נמצאה קבוצה עם מספר מזהה (אחד מהם או שניהם) נחזיר FAILURE, כל החיפוש מתבצע ב$O(\log k)$. נייצר קבוצה חדשה בעלת מספר מזהה הנתון ועם מספר נקודות השווה לסכום הנקודות של שתי הקבוצות המקוריות (סיבוכיות של $O(1)$). כעת נבצע מיזוג של עצי הקבוצות. נייצר 4 מערכים של מצביעים לשחקנים בגודל n_{Team1},n_{Team2} עבור קבוצה 1 ו-2 בהתאמה, 2 מערכים לכל קבוצה (אחד עבור כל עץ). לתוך מערכים אלה נכניס את השחקנים inorder מתוך העצים הקיימים בשתי הקבוצות (בכל קבוצה יש עץ ממויין לפי מספר מזהה ועץ ממויין לפי דירוג השחקן, סה"כ ב2 הקבוצות 4 עצים). פעולת הinorder לוקחת $O(n)$, לכן סיבוכיות מילאוי ארבעת המערכים ביא:

$2 \cdot O(n_{Team1}) + 2 \cdot O(n_{Team2}) = O(n_{Team1} + n_{Team2})$. כעת נרוץ על 4 המערכים ונעדכן את כמות המשחקים ששוחקו ע"י כל שחקן ע"י הוספת כמות המשחקים שהקבוצה שיחקה לכל שחקן. גם כאן הסיבוכיות הינה $O(n_{Team1} + n_{Team2})$.
לאחר מכן נמזג את המערכים הממויינים של שתי הקבוצות (המתאימות לעץ מסויים) למערך ממויין אחד. רצים על המערכים של שתי הקבוצות ומכניסים למערך החדש את השחקן הקטן יותר (לפי מספר מזהה או לפי דירוג השחקן). סיבוכיות פעולה זו היא $O(n_{Team1} + n_{Team2})$ כי אנו רצים על שני המערכים פעם אחת. גודל המערך החדש $n_{Team1} + n_{Team2}$. כעת נייצר עצים ממוזגים מתוך המערך הממויין הממוזג. מכיוון שייצרנו קבוצה חדשה, נוצרו שתי עצים ריקים. העצים שלנו מסוגלים, בהנחה שהם ריקים, להיבנות מתוך מערך ממויין בסיבוכיות זמן $O(n)$, כאשר n הוא אורך המערך. העץ עושה זאת על ידי הוספה רקורסיבית של nodes במיקומים המתאימים בעץ. השורש יהיה האיבר המרכזי במערך (מיקום $\frac{n+0-1}{2}$), ואז הבן השמאלי יבצע אותו חישוב על החצי הקן של המערך, הבן הימני על החצי הגדול של המערך וכו'. לאחר בניית העצים שאר משתני הקבוצה החדשה מעודכנים ב$O(1)$ והקבוצות המקוריות מרוקנות משחקנים ב$O(n_{Team1,2})$. לאחר מכן הקבוצות נמחקות מעץ הקבוצות והקבוצה החדשה מתווספת לעץ קבוצות. (סיבוכיות של $O(\log k)$). סה"כ סיבוכיות זמן:

$$O(\log k + n_{Team1} + n_{Team2})$$

כנדרש.

12.output_t<int> get_top_scorer(int teamId)

בפועל:

האם teamId<0 – אם כן :

ניזכר שאנחנו שומרים במבנה של כל העץ פויינטר לאיבר הכי ימני (כלומר האיבר המקסימלי) ניגש לעץ playerScore לשדה max שמחזיק פוינטר לאיבר המקסימלי ונחזיר דרכו את ה-Id של השחקן- סהכ לוקח O(1)

אחרת teamId>0:

נרצה לוודא שיש קבוצה עם המספר המזהה שהתקבל- לכן נשתמש בפונקציה של העץ מסוג bool אשר תבצע חיפוש על עץ הקבוצות (בעל K צמתים) לפי ה-Id שמקבלים , אם לא תמצא תחזיר false ואנחנו בהתאם נחזיר failure חיפוש כזה לוקח $\log(k)$.

במקרה בו יש קבוצה כזו נחפש אותה בעץ הקבוצות וניגש לdata של הnode ניגש לעץ playerScore של הקבוצה דרכו ניגש לשדה max של העץ - שמחזיק פוינטר לאיבר המקסימלי ונחזיר דרכו את ה-Id של השחקן- סהכ לוקח O(log(k))

13.output_t<int> get_all_players_count(int teamId)

בפועל:

אם קיבלנו teamId<0 , אנחנו שומרים משתנה כללי במבנה שמחזיק את המספר הכולל של כל השחקנים(מתעדכן בהכנסת שחקן חדש והוצאה) לכן פשוט נחזיר אותו לוקח- O(1) . אחרת קיבלנו teamId>0 לכן נחפש את הקבוצה בעלת המספר המזהה הנל (אם אין כזו מחזירים fail) לוקח logK ואז לכל קבוצה יש counter של שחקנים, נחזיר אותו.

14.StatusType get_all_players(int teamId,int *const output)

בפועל:

אם קיבלנו teamId<0, אנחנו עוברים על עץ playerScore בצורת InOrder ומכניסים תוך כדי המעבר את השחקנים למערך, לאחר מכן עוברים על המערך שנותנים לנו ועל המערך שיצרנו בו זמנית וממלאים במערך שקיבלנו את מספרי הזהות לוקח O(n) . אם קיבלנוteamId >0 אז מחפשים את הקבוצה ב-log(K) (מוודאים שיש קבוצה כזו) ועושים אותם פעולת , סה"כ לוקח O(log(K)+n_{team}).

15.output_t<int> get_closest_player(int playerId,int teamId)

בפועל:

במקרה בו קיבלנו ערכים תקינים, בודקים שאכן קיימת קבוצה עם המספר המזהה שנתנו לנו , אם קיימת מחפשים את הקבוצה בעץ teams לפי id , מהnode ניגשים לקבוצה עצמה , בקבוצה מוודאים שקיים שחקן בקבוצה עם המספר המזהה הזה , מחפשים אותו בעץ השחקנים לפי Id ניגשים לdata של השחקן ומהdata שלו עושים חיופש בעץ playerScore של הקבוצה, בודקים אם יש לילד ילדים (כלומר הוא מצביע על null , אם יש לו בנים עוברים על שני הבנים ועל האבא , בודקים איזה אחד יותר קרוב עליו במספר הגולים – אם יש יותר מ1 אם אותה קרה בודקים לפי מספר הכרטיסים , במקרה הכי גרוע בודקים לפי id. מחזירים את ה-id של השחקן שהכי קרוב עליו מבין השלושה .

חיפוש הקבוצה בעץ לוקח log(k) חיפוש השחקן בעץ של הקבוצה לפי id לוקח$\log(n_{team})$, חיפוש השחקן בעץ score לוקח $\log(n_{team})$ השוואה בין הבנים והאבא לוקחל O(1) סהכ - $O(logk + \log n_{team})$

16. output_t<int> knockout_winner(int minTeamId,int maxTeamId)

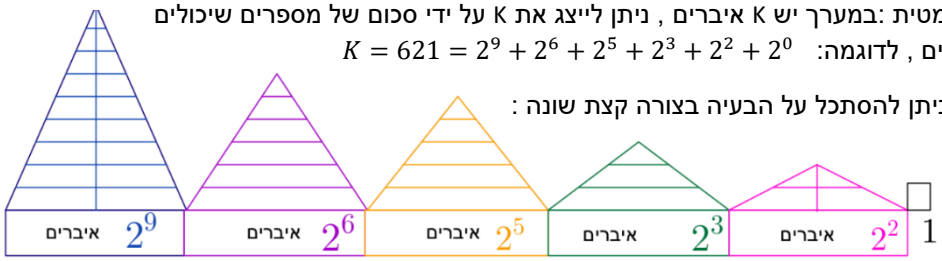
בפועל:

בעץ שלנו קיימת פונקציה המאפשר לעבור inOrder על כל האיברים בטווח נתון שלKey , במקרה שלנו נרצה לעבור על כל הקבוצות בעלות מספר מזהה גדול או שווה ל-**minTeamId** עד לקבוצה בעל מזהה קטן או שווה ל-**maxTeamId** , בזמן המעבר נבדוק עבור כל קבוצה בטווח האם היא חוקית למשחק , אם כן נוסיף אותה למערך ממויין על פי מס מזהה (כך שלכל משתנה במערך יש מספר מזהה ובנוסף יש לו ניקוד למשחק, כאשר ניקוד למשחק שווה לניקוד הקבוצה + סכום הגולים הכולל של כל שחקני הקבוצה – סכום כל הכרטיסי של כל שחקני הקבוצה). אם הקבוצה לא חוקית למשחק נמשיך לבאה .

לאחר בניית המערך שמכיל את כל הקבוצות : נבדוק האם המערך ריק, אם כן נחזיר fail, אחרת נקרא לפונציה עם המערך שתחזיר את החיזוי , נפרט עליה עכשיו:

נרצה להסתכל על הבעיה בצורה מתמטית :במערך יש K איברים , ניתן לייצג את K על ידי סכום של מספרים שיכולים להיות מיוצגים על ידי חזקות של שתיים , לדוגמה: $K = 621 = 2^9 + 2^6 + 2^5 + 2^3 + 2^2 + 2^0$

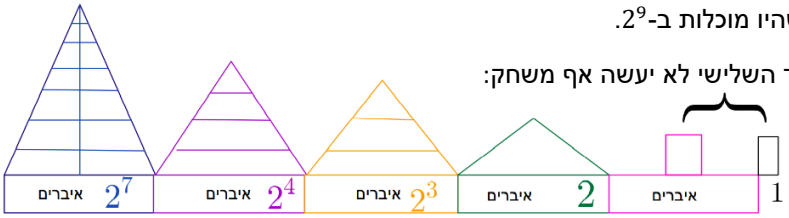
בגלל שכל זוג שכנים מבצע משחק , ניתן להסתכל על הבעיה בצורה קצת שונה :



נשים לב: לכל פירמידה מספר קומות שונה, כל קומה מתארת את מספר הקבוצות שעולות אחרי כל מחזור של משחקים (תמיד עולה חצי מהכמות). הפירמידה הכחולה (מייצגת את 2^9 האיברים הראשונים) יקח 9 מחזורים עד שאחת הקבוצות שמוכלות בה תוכל להכנס למשחק עם קבוצה מ-"פירמידה" אחרת (קבוצה שמבחינת מספר מזהה לא נמצאת ב-2^9 הקבוצות הראשונות), כלומר רק במחזור ה10 יהיה משחק עם קבוצה מהפירמידה הכחולה וקבוצה שלא שייכת עליו.

הסבר אינטיאוטיבי למה זה קורה: מיכוון שכל שחקן משחק עם שכנו ונשמר סדר האיברים לאחר כל מחזור , נקבל שעבור ה2^9 קבוצות הראשונות , מובטח שעד המחזור ה10 , מספר הקבוצות שישרדו מההתחלה ישאר זוגי ולכן מובטח שהמשחקים שיבצע עד המחזור ה-10 יהיו עם קבוצות שהיו מוכלות ב-2^9.

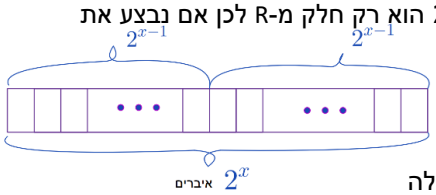
הריבועה השחור מתאר את האיבר האחרון שעד המחזור השלישי לא יעשה אף משחק:



אחרי שלקחנו בחשבון את המודל, נרצה להראות את הדרך בא נמצא את ההימור לקבוצה המנצחת בסיבוכיות זמן הנדרשת.

נסתכל על כל פירמידה בנפרד. נרצה למצוא מנצח עבור כל פירמידה, לשם כך נסתכל על כל פירמידה כמערך. נשים לב מאופן בניית הפירמידות, מספר האיברים במערך הוא -2^X ! . לכן נצפה ל-X שלבים למציאת המנצח של הפירמידה.

נקח את המערך נחצה אותו ועבור כל חצי בנפרד נסכום את כל הנקודות של כל הקבוצות באותו החצי, נשווה בין שני הסכומים, נצפה שאחד יהיה גדול יותר (במקרה של שיוויון החצי העליון יותר חזק) , כעת נסתכל על החצי החזק יותר, יש בו -2^{X-1} איברים , נבצע אותה פעולה באופן רקורסיבי, בסוף נגיע למצב שנשווה בין 2 קבוצות, המנצחת בינהן תיהיה המנצחת של הפירמידה. הצדקה מתמטית לסיבוכיות: $2^X \cdot \sum_{n=0}^{x-1} \left(\frac{1}{2}\right)^n = 2 \cdot (2^x - 1)$, נזכור 2^X הוא רק חלק מ-R לכן אם נבצע את האלגוריתם הנל לכל הפרמידות נקבל שהסיבוכיות היא O(R), כפי שהתבקשנו.



סה"כ : בניית המערך הראשוני שיחזיק את כל הקבוצות שמקיימות את התנאי לוקח $O(\log(K))$, יצירת המערכים החדשים + מציאת המנצח של כל פירמידה לוקח $O(r)$, לסיום נאשר להשוות את המנצח של הפירמידה הכי קטנה עם הצמודה לה וכן האלה ולמצוא את המנצח הכללי.

נשים לב: בפועל הפירמידות מנוהלות ע"י אינדקסים, כלומר אין העתקה של כל פירמידה לתתי מערכים אלא שומרים את אינדקסי ההתחלה והסוף של כל פירמידה במערך הכולל. בכל סיבוב משחקים מתעדכנים אינדקסי ההחלה הסוף של הפירמידות לפי הצד בפירמידה שניצח (הוסבר קודם), עד שאינדקס ההתחלה שווה לאינדקס הסוף (כלומר הגענו לראש הפירמידה).