בנינו את מחלקות העזר הבאות:

Player

Name	Type
ID	Int
Level	Int
Group_ptr	Pointer to group

השדה Group יצביע לקבוצה אליה שייך השחקן בתוך עץ כל הקבוצות של PlayersManager (ראה מימוש PlayersManager).

Level

Name	Type	Sorted by
ID	Int	-
Player_tree	AVL tree with key: int and info: Player*	Player ID

השדה Player_tree יכיל את כל השחקנים השייכים ל-level מסויים. מספר הצמתים בעץ Player_tree חסום ע"י מספר השחקנים בעלי רמה ID כלשהי.

Group

Name	Type	Sorted by
ID	Int	-
Size	Int	-
Level_tree	AVL tree with key: int and info: Level*	Level_ID
Max_player_info	Struct: int id and int level	

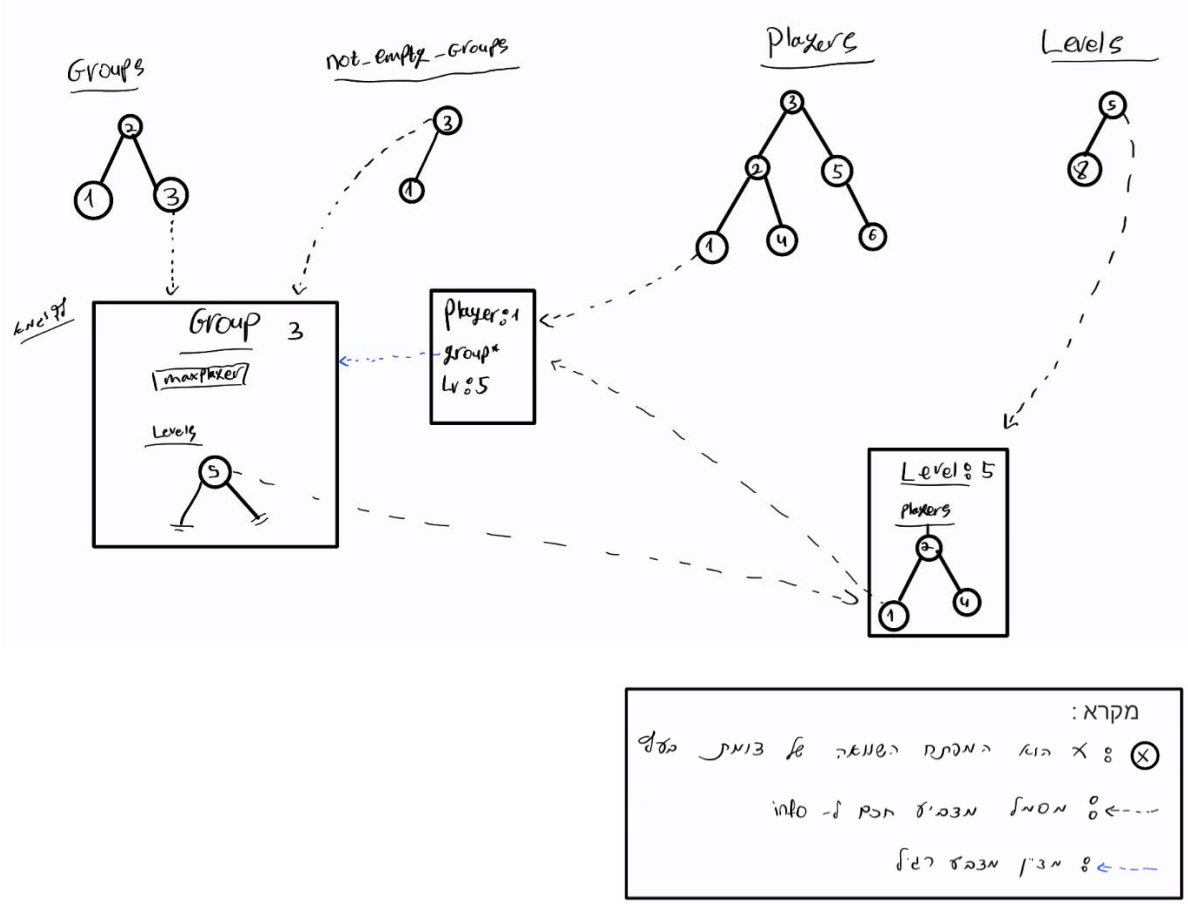
השדה Level_tree יכיל את כל הרמות הקיימות בקבוצה כלשהי. בכל רמה יש עץ של כל השחקנים השייכים לרמה. מספר הצמתים בעץ Level_tree חסום ע"י מספר הרמות בקבוצה, שזה חסום ע"י מספר השחקנים בקבוצה.

מבנה הנתונים שבחרנו לעבוד איתו בנוי בצורה הבאה:

PlayersManager

Name	Type	Sorted by
Group_tree	AVL tree with key: int and info: Group	Group ID
Not_empty_group_tree	AVL tree with key: int and info: Group	Group ID
Player_tree	AVL tree with key: int and info: Player	Player ID
Level_tree	AVL tree with key: int and Info: Level	Level
Max_player_info	Struct: int id and int level	

שרטוט של מבנה הנתונים:



סיבוכיות מקום:

נסמן ב-k את מספר הקבוצות וב-n את מספר השחקנים.

כל שחקן מופיע לכל היותר בשלושה עצים:

- Players_tree - כשחקן במערכת.
- Level_tree - כשחקן ברמה שבה הוא נמצא.
- Group_tree - כשחקן בעץ הרמות של הקבוצה שלו.

כלומר כל שחקן מופיע לכל היותר 3 פעמים במערכת, ולכן סיבוכיות המקום עבור n שחקנים היא O(n).
כל קבוצה מופיעה לכל היותר בשני עצים:

- Group_tree - כקבוצה במערכת.
- Not_empty_group_tree - כקבוצה לא ריקה במערכת

נעיר כי כל שחקן רק מצביע על קבוצה ולכן לא מחזיק את הקבוצה עצמה, כלומר המצביע תופס מקום קבוע בכל שחקן ולכן סיבוכיות המקום של עץ השחקנים לא תלוייה במספר הקבוצות.
אז כל קבוצה מופיעה לכל היותר פעמיים במערכת, ולכן סיבוכיות המקום עבור k קבוצות היא O(k).

בסך הכל סיבוכיות המקום של המערכת עבור k קבוצות ו-n שחקנים: O(n+k).

הערות נוספות לגבי גדלי העצים:
כמות הצמתים המקסימלית של עץ הקבוצות הלא ריקות Not_empty_group_tree קטנה או שווה למספר המינימלי בין כמות הקבוצות וכמות השחקנים, כי במקרה הגרוע יש בכל קבוצה שחקן אחד בלבד ולא ייתכן יותר קבוצות לא ריקות מאשר קבוצות.

כמות הצמתים המקסימלית בעץ הרמות Level_tree הוא כמות השחקנים במערכת, כאשר כל השחקנים נמצאים בשלבים שונים, ולכן ביצוע פעולות על רמה בעץ לוקח לכל היותר O(logn) וחיפוש בתוך רמה לוקח גם לכל היותר O(logn). כלומר בסה"כ פעולות על שחקנים ב Level_tree מתבצעות ב O(logn) .
באופן דומה כמות הצמתים המקסימלית בעץ רמות Level_tree של קבוצה היא ככמות השחקנים בקבוצה - כאשר כל השחקנים נמצאים ברמה שונה וגם כולם שייכים לאותה קבוצה. לכן גם פה פעולה על שחקן בקבוצה מתבצעת ב-(O(log(num_of_players_in_group)) ובפרט ב-O(logn).

סיבוכיות זמן:

Init():

יוצרת PlayersManager ריק. כדי לעשות זאת צריך לאתחל את כל העצים לעצים ריקים. כל עץ מחזיק מצביע לשורש העץ ולכן כדי לאתחל אותו לעץ ריק נדרש לתת למצביע ערך nullptr (נעשה בבנאי של העץ).
כדי לאתחל את ה-struct ניצור struct חדש עם ערך דיפולטי מינוס אחד (1-).
בסך הכל מתבצעות ארבעה פעולות ולכן סיבוכיות זמן של O(1) כנדרש.

AddGroup(void *DS, int GroupID):

מוסיפה קבוצה ריקה למבנה הנתונים.
ראשית נבדוק האם הקבוצה כבר קיימת - חיפוש בעץ Group_tree, כפי שלמדנו חיפוש בעץ AVL בעל k צמתים הוא בסיבוכיות O(logk).

אם הקבוצה לא קיימת, נייצר דינמית Group חדש עם ה-GroupID שקיבלנו ונוסיף אותו לעץ Group_tree.
הוספת צומת לעץ AVL עם k צמתים מתבצעת בסיבוכיות O(logk).
לכן בסך הכל סיבוכיות הזמן של הוספת קבוצה היא O(logk).

AddPlayer(void *DS,int PlayerID, int GroupID, int Level):

מוסיפה שחקן חדש לקבוצה.
בהינתן שהעצים במערכת כולם עצי AVL חיפוש הוספה ומחיקה מתבצעים ב log(n) או log(k) בהתאם.
ראשית בדקנו כי הקבוצה הנתונה קיימת ושמרנו את כתובתה - חיפוש בעץ קבוצות ב log(k), ושהשחקן אינו קיים כבר במערכת -חיפוש ב log(n).
שנית יצרנו דינמית את השחקן, הוספנו אותו לעץ השחקנים ב log(n), הוספנו אותו לעץ רמות בקבוצה אליה הוא שייך ב log(n) (כי חיפוש הרמה הרלוונטית לוקחת עד log(n) ואז הוספת השחקן ברמה לוקח עד log(n), ובעץ רמות כל השחקנים במערכת בסיבוכיות זמן log(n) (חיפוש רמה רלוונטית לוקח עד log(n) ואז הוספת השחקן עד log(n)).
תחזקנו את השחקן הגבוהה ביותר במערכת O(1) בעזרת השוואת Int.
תחזקנו את השחקן הגבוהה ביותר בקבוצה O(1) בעזרת השוואת Int.
אם הקבוצה הייתה ריקה בעבר הוספנו אותה אל עץ הקבוצות הלא ריקות ב O(logk).
בסך הכל הוספת שחקן מתבצעת בסיבוכיות זמן O(logk+logn).

RemovePlayer(void *DS, int PlayerID):

מסירה שחקן מהמערכת.
נבדוק אם השחקן קיים: חיפוש בעץ player_tree ולכן מתבצע בסיבוכיות O(logn).
אם השחקן קיים, צריך להסיר אותו משלוש עצים:

- level_tree: צריך למצוא את הרמה אליה השחקן שייך. נחפש בעץ player_tree את השחקן שברצוננו להסיר (חיפוש בעץ AVL עם n צמתים מתבצע בסיבוכיות זמן O(logn)) ונחלץ מה-info שלו את הרמה של השחקן. כעת נמצא בעץ level_tree את הצומת המתאימה לרמה של השחק ונסיר מהשדה player_tree שנמצא ב-info של הצומת את השחקן PlayerID.
  חיפוש ב-level_tree: סיבוכיות זמן O(logn).
  חיפוש בplayer_tree בתוך ה-level הרלוונטי: סיבוכיות זמן O(logn).
  בסך הכל סיבוכיות הזמן של הסרה מ-level_tree היא O(logn).
- player_tree: הסרה מעץ AVL מתבצעת בסיבוכיות זמן O(logn).
- group_tree: בתוך כל קבוצה יש עץ של כל הרמות שקיימות בקבוצה. כבר חיפשנו את את ה-info של השחקן מקודם, יש לגשת לשדה group_ptr כדי להגיע לקבוצה אליו הוא שייך. נשאר להסיר אותו מהעץ level_tree שנמצא בתוך הקבוצה שמצאנו, נעשה בדיוק כמו שכתוב מקודם - סיבוכיות זמן O(logn).

כעת צריך לתחזק את השדות המחזיקים את השחקן בעל הרמה הגבוהה ביותר במערכת ובקבוצה אליה שייך PlayerID.
מציאת השחקן בעל הרמה הגבוהה ביותר במערכת: נלך ל-level_tree ונחפש את הצומת עם הערך הכי גבוהה: הולכים ימינה עד הצומת האחרון - מתבצע בסיבוכיות זמן O(logn). כעת ניגש לשדה player_tree שנמצא ב-info של ה-level שמצאנו עכשיו בחיפוש ונחפש שם את השחקן עם הרמה הכי נמוכה: הולכים שמאלה עד הצומת האחרון - סיבוכיות זמן O(logn).
מציאת השחקן בעל הרמה הגבוהה ביותר בקבוצה: אותו דבר בדיוק כמו מקודם אבל עם ה-level_tree ששייך לקבוצה אליה PlayerID שייך, חילצנו אותה מקודם מתוך השדה group_ptr של השחקן.
עדכון השדה max_player_level בקבוצה ובכללי מתבצעת ע"י שינוי השדות שלו ולכן לא תלוייה באורך הקלט.

בסך הכל כל הפעולות התבצעו ב(O(logn ולכן סיבוכיות הזמן של הפונקצייה היא (O(logn כנדרש.

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

מכניסה את השחקנים בקבוצה GroupID לקבוצה ReplacementID.
ראשית מוצאים את הצומת המייצגת את כל אחת מהקבוצות בעץ Group_tree - פעמיים חיפוש בעץ AVL, סיבוכיות זמן (O(logk. לכל אחת מהקבוצות האלה יש שדה של עץ רמות - level_tree.
נאחד את העצים האלו* - סיבוכיות זמן (O(m1+m2 כאשר m1 זה מספר השחקנים בקבוצה GroupID ו-m2 מספר השחקנים בקבוצה ReplacementID.
ניצור דינמית קבוצה חדשה בעלת מזהה ReplacementID כך שהעץ level_tree שלה הוא עץ האיחוד ממקודם. כמו כן נעדכן את השדה size לגודל עץ האיחוד.
נזכיר כי לכל שחקן יש מצביע לקבוצה אליה הוא שייך. לכן צריך לעדכן את המצביעים של כל השחקנים בקבוצה שיצרנו. כדי לעשות זאת נעבור בעזרת סיור Inorder על עץ האיחוד (עץ הרמות של הקבוצה). לכל צומת שנגיע נסייר סיור Inorder על עץ השחקנים שבאותה רמה. לכל צומת שנגיע נעדכן את הקבוצה עליה השחקן מצביע לקבוצה החדשה שהקצנו. סכום מספר השחקנים של כל רמה על פני כל הרמות בקבוצה הוא בדיוק מספר השחקנים בקבוצה ולכן פעולה זו מתבצעת בסיבוכיות זמן (O(m1+m2.
כעת נסיר מהעץ group_tree את הקבוצות GroupID ו-ReplacementID ונוסיף את קבוצת האיחוד שלהם.
נעשה אותו דבר לעץ not_empty_group_tree כך ששומרים על התכונה בעץ זה שכל קבוצה שנמצאת בו אינה ריקה (ייתכן שאחד הקבוצות GroupID, ReplacementID ריקות ולכן לא קיימות בעץ, יש התייחסות לזה). שתי פעולות אלו קורות ב-(O(logk.
לבסוף מעדכנים את השדה Max_player_info בקבוצה החדשה כך שיכיל את הפרטים הנכונים: חיפוש הרמה הגבוהה ביותר בקבוצה ואז את השחקן בעל המזהה הקטן ביותר בקבוצה זו מתבצעת בסיבוכיות (O(log(m1+m2)), כפי שמתואר בפונקצייה הקודמת.
לכן בסך הכל הפונקציה מתבצעת בסיבוכיות זמן (O(m1+m2+logk כנדרש.
* איחוד עצים: משטחים את העצים למערכים ומאחדים את הרשימה בסיבוכיות זמן (O(m1+m2 כפי שראינו בתרגול. כעת נייצר עץ חדש לפי הרשימה בעזרת האלגוריתם הבא:
0. קריאה לפונקצייה getTreeFromList(Array list, int start, int end) כאשר start זה אינדקס האיבר הראשון במערך ו-end האחרון.
1. חשב mid=(start+end)/2.
2. בצע root=array[mid].
3. בצע root->left=getTreeFromList(list, start, mid-1).
4. בצע root->right=getTreeFromList(list, mid+1, end).
כך עברנו פעם אחת על כל איבר במערך ושמנו במקום מתאים בעץ - סיבוכיות זמן (O(m1+m2.
השארית המקסימלית בחלוקה ל-2 היא 1, לכן לכל צומת, ההפרש (בערך מוחלט) בין כמות הצמתים משמאלו ומימינו יהיה לכל היותר 1. לכן העץ שיצרנו מאוזן.
הוכחה פורמלית לנכונות אלגוריתם זה ולעמידתו בסיבוכיות בסוף גליון זה.
נעיר כי ייתכן שבשלב איחוד הרשימות, יהיו איברים "כפולים" ברשימה (למשל אם לשתי הקבוצות יש רמה 1). במקרה זה נצטרך לאחד את עצי השחקנים של הרמות הכפולות, לכן נקרא לפונקציה שוב. כעת לא ייתכן כי ברשימה יהיה איברים כפולים כי לא ייתכן שאותו שחקן יהיה בשתי רמות שונות ולכן אין סכנה לקריאות אינסופיות לפונקציה. איחוד עצי השחקנים יקח
(O(num_of_players_in_first_level+num_of_players_in_second_level. סכום כל השחקנים בכל הרמות עבור 2 הקבוצות הוא m1+m2 ולכן גם אם נצטרך לבצע איחוד עצים עבור כל רמה ברשימת הרמות המאוחדת, עדיין נעמוד בסיבוכיות זמן של (O(m1+m2.

IncreaseLevel(void* DS , int playerID, int LevelIncrease):

ראשית מוצאים את השחקן במערכת Player_tree - מתבצע ב O(logn).
לפי הרמה הנוכחית של השחקן מסירים אותו מ Level_tree ב O(logn).
בעזרת המצביע של השחקן לקבוצה אליה הוא משתייך ניתן להגיע לקבוצה ב-O(1) ולהסיר באופן דומה אותו מעץ הרמות של הקבוצה ב O(logn) את השחקן.
אם הוא היה השחקן היחיד ברמה הקודמת בקבוצה או במערכת כולה מסירים את הרמה מהעצים הרלוונטיים - כל הסרה של צומת בעץ הרמות לקחת גם היא O(logn) .
מוסיפים לשחקן את levelincrease ובודקים אם הרמה החדשה שבה הוא נמצא קיימת במערכת ובקבוצה שלו - כל חיפוש רמה כזה לוקח O(logn), אם לא ,נוסיף לכל אחד מהעצים במידת הצורך את צומת הרמה החדשה - הוספה של צומת בעץ מתבצעת ב O(logn). לאחר מכן נוסיף את השחקן לעץ הרמות של המערכת ועץ הרמות של הקבוצה גם אלו מתבצעות ב O(logn). נעדכן את השחקן בעל הרמה הגבוהה ביותר בקבוצה ובמערכת במידת הצורך על ידי פעולה שהולך לקצה הימני ביותר בעץ - יתבצע גם הוא ב O(logn).
לכן בסך הכל הפונקציה מתבצעת בסיבוכיות זמן O(logn) כנדרש.

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

אם צריך להחזיר את השחק בעל הרמה הגבוהה ביותר בכל המערכת ניגשים לשדה Max_player_info בתוך Players_manager וב-O(1) נקבל את התוצאה הדרושה. זאת מכיוון ששדה זה הוא אינווריאנט ובכל פעולה אנו דואגים לשמור על נכנותו.
אחרת - נחפש בעץ הקבוצות את הקבוצה בעלת ה ID הנתון ב-O(logk) וניגש לשדה Max_player_info שלה ב-O(1) ונחזיר את התוצאה הדרושה.
לכן סה"כ הפונקציה מתבצעת ב-O(1) במקרה הראשון ו-O(logk) במקרה השני כנדרש.

GetAllPlayersByLevel(void* DS, int GroupID , int* PlayerID):

במידה ונצטרך להחזיר את כל השחקנים במערכת לפי הרמה שלהם (בסדר יורד) נסרוק בinorder "הפוך" על מנת לקבל את הרצוי מהרמה הגבוהה לקטנה ביותר ב Level_tree - מתבצע ב O(n) במקרה הגרוע בו כל השחקנים ברמות שונות.
לאחר מכן בכל רמה נבצע סריקה ב inorder רגיל של עץ השחקנים ברמה כדי לקבל אותם לפי ID מהקטן לגדול.
במקרה הגרוע ביותר כל השחקנים נמצאים ברמה זהה ולכן גם זה יתבצע ב O(n) (אך במקרה זה ב-Level_tree צומת אחת ולכן החיפוש שם יקח O(1)).
סה"כ שני המעברים לא יקחו יותר ממספר השחקנים בכל המערכת ולכן זה יתבצע ב O(n) כנדרש.

אחרת נחפש את הקבוצה הנתונה בעץ הקבוצות - זה יתבצע ב O(logk) ואז נבצע את אותו אלגוריתם אך על עץ הרמות של הקבוצה. בהינתן n כמות השחקנים בקבוצה הנתונה, הסריקה תתבצע ב O(n). ובסך הכל ב-O(n+logk).

GetGroupsHighestLevel(void* DS, int GroupID ,int NumOfGroups, int PlayerID):**

צריך להחזיר את מזהה השחקן המקסימלי עבור כל אחת מ-NumOfGroups הקבוצות בעלות המזהה הכי קטן.
נבצע סיור inorder על עץ הקבוצות הלא ריקות - not_empty_group_tree.
לכל צומת שנגיע, נכניס למערך (מתבצע ב-O(1)) את המזהה של השחקן המקסימלי - נמצא בשדה MaxPlayerInfo בקבוצה.

נעצור לאחר שנכניס NumOfGroups איברים.
כפי שלמדנו, סיור inorder של NumOfGroups צמתים מתבצע בסיבוכיות O(NumOfGroups) כנדרש.

Quit()

הורסים את כל אחד מהעצים: עץ השחקנים נהרס בסיבוכיות זמן O(n), עץ הרמות נהרס גם כן ב-O(n) כי יש num_of_levels רמות במשחק ולכל רמה יש num_of_players_in_level שחקנים בעץ השחקנים של הרמה, אז כדי להרוס את עץ הרמות מתבצעות num_of_levels*num_of_players_in_level=n פעולות.
עץ הקבוצות נהרס ב-O(n+k) כי יש k צמתים להרוס ובכל צומת יש עץ levels להרוס שהוא נהרס ב-O(num_of_players_in_group) ולכן בסך הכל מתבצעות k+n פעולות.
כנ"ל עבור עץ הקבוצות הלא ריקות.
בסך הכל הפונקציה מתבצעת בסיבוכיות זמן O(n+k) כנדרש.

<u>הוכחה לנכונות האלגוריתם ליצירת עץ בינארי מאוזן מרשימה ממוינת:</u>
ראשית נעיר כי לכל צומת, ההפרש בין מספר הצמתים מימינה לבין מספר הצמתים משמאלה הוא הוא לכל היותר 1 ולכל הפחות 1- (כי שארית מקסימלית בחלוקה ב-2 היא 1). נוכיח באינדוקציה שלמה על עומק העץ כי כל עץ המיוצר באלגוריתם זה הוא מאוזן.
-<u>בסיס:</u> עומק עץ=1. כלומר יש לנו עץ עם צומת אחת, זהו עץ מאוזן. גם עבור עומק עץ=0 מתקבל עץ מאוזן.
-<u>הנחה:</u> נניח כי עץ בעומק n המתקבל מהאלגוריתם הוא מאוזן.
-<u>צעד:</u> נניח כי קיבלנו עץ בעומק n+1 ונוכיח כי הוא מאוזן. נסתכל על השורש ונתבונן בתת העץ השמאלי שלו. זהו עץ בעומק לכל היותר n שהתקבל מהאלגוריתם (כי לפי האלגוריתם השורש הוא האיבר האמצעי ברשימה הממוינת, ותת העץ השמאלי מתקבל מהפעלת האלגוריתם על המחצית השמאלית של הרשימה לא כולל השורש), מההנחה הוא מאוזן. נסמן את גובהו ב-m. באותו אופן עבור תת העץ הימני, נסמן את גובהו ב-k.
מההערה מתחילת ההוכחה, מתקיים $|m-k| \leq 1$.
לכן $-1 \leq BF \leq 1$ עבור השורש, ועבור שאר הצמתים זה מתקיים מהנחת האינדוקציה.
לכן האלגוריתם מייצא עץ מאוזן.

<u>הוכחה לעמידה בסיבוכיות:</u>
אם נסמן את זמן ריצת האלגוריתם עבור קלט n ב-$T(n)$, אזי מתקיים:

$$T(n) = 2T\left(\frac{n}{2}\right) + C$$

כי בכל ריצה של האלגוריתם מפעילים אותו פעמיים על קלט הקטן בחצי.
משיטת המאסטר נקבל: $f(n) = C = O(1),\ a = 2 = b,\ \log_b a = \log_2 2 = 1$
ולכן אנחנו במקרה הראשון עבור $\epsilon = 1$, כלומר $f(n) = O\left(n^{\log_b a - \epsilon}\right) = O(n^{1-1}) = O(1)$.
ולכן לפי המשפט נקבל $T(n) = \theta\left(n^{\log_b a}\right) = \theta(n)$ ובפרט $T(n) = O(n)$ כנדרש.