section 3: theoretical questions

<u>תשובה עבור שאלה 1.</u>

Type	**Usage**	**Space Complexity**
$int\ (RotationTimes)$	סופר לנו כמה רוטציות קורות בעת הכנסת כל איבר	$O(1)$ קבוע
$Deque < Object >$	שומרים בתוכו את מספר הרוטציות, את המצביע של האיבר שנכנס($node$) , את האיבר שקרתה בו הרוטציה ,איזו רוטציה קרתה(R, L) ועוד האם יש רוטציה בכלל או לא. (כלי אחסון)	מאחסנים בתוכו 5 נתונים שמסופרים ב $usage$, Space Complexity של כל נתון = $\theta(n)$ ולכן סכה $5 * \theta(n) = \theta(n)$
$int\ (size)$ בתוך מחלקת ה $node$ שנמצאת בתוך מחלקת ה AVL	סופר לנו כמה בנים יש ל$node$.	$O(1)$ קבוע

__תשובה עבור שאלה 2.__

type	**Usage**	**Space Complexity**
ArrayDeque<Object>	To store at it four items : the “splitcounter”, the last node that was split, the last value that was inserted, median index, Boolean value.	For the initialization its take $O(1)$ For each insertion action and split during it we insert five items, the worst case is when we need to do split for each step until the node we need to add the key to, it will take as much as the height of the tree : $5 * O(h) = O(5h)$ Add this for each key we do insertion for which means: $n * O(5h)$
int (splitcounter)	To count the number of split actions that did happened all the way until the node that split	$O(1)$
Node<T>	The last node that was inserted due to get it back at the same spot like before	$O(1)$
T (value)	The value that we added recently We need to save it so we can deleted after and represent the tree as it was before this key	$O(1)$
Int (median index)	Saving the median index gives us the ability to know the value that have been chosen to go up during the split function to put it back to his correct place, In addition, we can also remove the children that we add in the split function using this index.	$O(1)$
Boolean value	The first thing that goes out to check if the last function that happened was the split one or add key	$O(1)$

תשובה עבור שאלה 3.

operation	Number of Repetitions	Total Time Complexity
Backtrack ()	מוצאים ממבנה הנתונים deq שיש לנו את מספר הרוטציות שעשינו בעת ההכנסה ואת הצומת שממנה העץ השתנה (התאזן) בעזרת שיטת העזר שמימשנו $findmynode$ ואז פועלים לפי מספר הרוטציות שיש לנו כלומר אם לא היו רוטציות אזי פשוט מוחקים את הצומת, אחרת נכנסים ללולאה שמתארת את מספר הרוטציות שהיו לנו ופועלת לפי הרוטציה שהייתה לנו, דהיינו אם הייתה רוטציה מימין אז עושים רוטציה לשמאל וכנ״ל לגבי שאר המקרים. ולבסוף מעדכנים את הגבהים.	נראה כי פונקציית העזר $findmynode$ שמשתמשים בה בעת מציאת הצומת שחלה בה השינוי כלומר קרתה רוטציה, לוקחת $\theta(\log n)$ במקרה הגרוע, פונקציית עדכון הגבהים לוקחת $\theta(\log n)$ כיוון שבמקרה הגרוע עדכון הגבהים יתחיל מהשורש עד האיבר האחרון בעץ, וכל שאר הקוד בעל זמן ריצה $\theta(1)$ ולכן בסכה $T(n) = \theta(1) + 2 * \theta(\log n) = \theta(\log n)$ הערה : הלולאה שעובדת כמספר הרוטציות + פעולות הרוטציה בעלות זמן קבוע .
$findmynode$	משתמשים בה פעם אחת בתחילת פעולת ה $backtrack$ כך שבאמצעותה מביאים את הצומת (המצביע) של האיבר.	נראה כי המקרה הגרוע בחיפוש הצומת הוא $\theta(\log n)$ כיוון שהערך יכול להיות עלה ונצטרך לעבור על גובה העץ עד שנגיע אליו.

תשובה עבור שאלה 4.

הסבר לפונקציית BTree Backtrack

ראשית נוציא את הערך הבוליאני כדי לדעת מה הפעולה האחרונה שהתבצעה

במידה והייתה הפעולה האחרונה היא הוספת ה key אז פשוט נעשה עליו מחיקה ונשאל האם הערך של split counter שווה לאפס? אם כן אז סיימנו אחרת נצטרך לעשות קריאה רקורסיבית לפעולת ה backtrack.

הפעם כן הפעולה האחרונה שהתבצעה היא split ואז נוציא מה deque שלנו את הקדקוד ששמרנו כבר נסמנו X (זהו הקדקוד שהתבצעה פעולת הפיצול) נחזיר אותו למקומו כלומר נחזיר את כל בניו שהיו לו לפני הפיצול ונחזיר את אבא שלו

שני הבנים החדשים שייצרנו תוך כדי פעולת ה split נשמיט אותם ונחזיר את הערך הבינוני שהפך להיות בקדקוד מלמעלה ונסיר אותו משם

**חשוב מאוד לא לשכוח את המקרה שבו יוצר אבא חדש שונה מאבא של הקדקוד X במקרה זה גם נשמיט אותו ונחזיר את אבא שהיה לפניו(כבר שמור אצל הבן X עדיין יהי מצביע עליו)

הפעולה הזו מתבצעת באופן רקורסיבי כך שכל פעם אנו שואלים אם עדיין קיימת פעולת split שלא שיפרנו אותה עד כדי לעבור על כל גובה העץ.

Split counter מחשב את פעולות הפיצול שחלו רק על המסלול עד ההגעה לקדקוד שבו נמצה ה key האחרון שהוסף, לאחר מכן ה split counter מתאפס.

מבחינת זמן ריצה כל פעולת מחיקה של key תצטרך $O(1)$ לעומת זאת כל פעולת פיצול תצטרך (2t) לכל היותר וכתלות במספר הילדים של הקדקוד שחלה עליו פעולת הפיצול.(כי אנו בשלב זה רצים על כל הילדים של הקדקוד X ומשפרים את מקומם בעץ)

הפעולות הרקורסיביות ייקחו לכול היותר כאורך העץ כיוון שהשינויים שהתבצעו הם תמיד יהיו לפי פעולות הפיצול כמהלך אורך העץ כלומר $\theta(\log n)$.

לבסוף ניתן להסיק שסך כל זמן הריצה של פעולת ה $backtrack$ הינו $\theta(2t\log n)$.

Operation	Number of Repetitions	Total Time Complexity
Remove child	It takes (2t) times for the worst case repetitions because its depends on the number of children that the split node have and it have at most (2t) children.	$\theta(t)$
Remove key	It depends on the Height of the tree, because we remove the median value, so in the worst case if the split function happened a lot of times it will take in upper bound as the height of the tree which is $\theta(\log n)$	$\theta(\log_t n)$
Add child	we add a child (which is the node that split in the last insert function)in every time we discovered that the last function was the split one.so It takes at most as the height of the tree.	$\theta(\log_t n)$
Backtrack (recursive function)	The number of split function that happened till we just add the key in its correct place. Which depends on the number of the children that every split node have which is ((2t)children) so it will take as much as the height of the tree and the children $\theta(\log_t n)$	$\theta(\log_t n)$

the recursive function repeated log n times in every time there is a function that cost o(t) so in total it will take at most $O(t * \log_t n)$

תשובה עבור שאלה 5.

נראה כי זמן הריצה הכללי של רעיון דני ל backtrack דורש $\theta(1)$ זמן קבוע כיוון שהוא רק מכניס למבנה, שולף מהמבנה ומעדכן מצביע, ולכן אם הוא רוצה לעשות n פעמים backtrack זמן הריצה יהיה $\theta(n)$.

תשובה עבור שאלה 6.

לא, כיוון שדני בכל $insert$ הוא ישתמש בזיכרון נוסף ענק של $O(n^2)$ כדי לשמור את העץ, האלגוריתם שממששנו בחלק א הוא טוב יותר כיוון שהזיכרון הנוסף הוא $O(n)$ וזמן הריצה $\theta(2t\log n)$ (יש שוויון בין זמן הריצה לזיכרון).