**AVLNode:**

Fields:

* parent – pointer to parent of node
* left – pointer to left child of node
* right – pointer to right child of node
* key – key of node
* value – value of node
* height – height of node
* trueCnt – number of nodes whose value is true in subtree whose root is this node

Functions:

* empty constructor – Creates virtual node. O(1)
* constructor with key, value – Creates node with matching key and value. O(1)
* getKey – Returns key of node. O(1)
* getValue – Returns value of node. O(1)
* setLeft – Sets left child of node to be given node. O(1)
* getLeft – Returns left child of node. O(1)
* setRight – Sets right child of node to be given node. O(1)
* getRight – Returns right child of node. O(1)
* setParent – Sets parent of node to be given node. O(1)
* getParent – Returns parent of node. O(1)
* isRealNode – Returns true if and only if node is node not virtual node (key is -1). O(1)
* setHeight – Sets height of node to be given node. O(1)
* getHeight – Returns height of node. O(1)
* updateHeight – Sets height of node to one plus maximum between the height of its children. O(1)
* getBalanceFactor – Returns difference between height of left child and right child of node. O(1)
* updateTrueCnt – Sets trueCnt of node to sum of trueCnts of its children (plus one if value of node is true). O(1)

**AVLTree:**

Fields:

* virtualNode – pointer to common virtual node for the whole tree
* root – pointer to root node in the tree
* size – counter for the number of nodes in the tree
* min – pointer to node with smallest key in tree
* max – pointer to node with largest key in tree

Functions:

* empty constructor – Creates empty tree. O(1)
* empty – Returns whether root is not a virtual node. O(1)
* search – Starts at root node. Compares k to key of current node. If equal return value of node, if k is smaller continue searching on left child, and if k is larger continue searching on right child. If virtual node reached (end of tree), return null. At each iteration, search is either completed or continued on child node. Therefore, in worst case we walk down height of the tree. O(log n)
* setFirstParentToSecondParent – Sets the parent of node1 to be the parent of node2 and also updates the reciprocal connection. O(1)
* setFirstRightChildToSecond – Sets the right child of node to be rightChild and also updates the reciprocal connection. O(1)
* setFirstLeftChildToSecond – Sets the left child of node to be leftChild and also updates the reciprocal connection. O(1)
* rotationHeightXorUpdate – Updates height and xor of nodes whose values may have changed after a rotation. O(1)
* rotation – Given an AVL criminal node, performs matching rotation and returns the node which is now in the criminal's original location. O(1)
* insert – Searches for proper location of the new node by walking down the height of the tree. If a node with key k is encountered, return -1. Otherwise, if the new node’s key is smaller/larger than the min/max update min/max to point to new node. After that, we add the new node to the tree in its proper location. Lastly, we walk the path from the inserted node to the root fixing AVL criminals and updating the height and trueCnt of the nodes on the path, counting the number of rotations and height updates made. Finally, size is increased by 1 and the count of rotations and height updates made is returned. In the worst case, we walk down and up the height of the tree. O(log n)
* delete – If key to be deleted is same as that of min/max, update min/max to parent. Next, search for node with key k by walking down the height of the tree.

If such node is found, there are three scenarios:

* If node has two children, we remove its successor and put it in its place.
* If node has one child, we bypass node straight to its child.
* If node is leaf, we simply remove it from tree.

If no such node is found, we return -1.

Lastly, we walk the path from the parent of the physically deleted node to the root fixing AVL criminals and updating the height and trueCnt of the nodes on the path, counting the number of rotations and height updates made. Finally, size is decreased by 1 and the count of rotations and height updates made is returned. In the worst case, we walk down and up the height of the tree. O(log n)

* min – Simply returns the value of min or null if min is null. O(1)
* max - Simply returns the value of max or null if max is null. O(1)
* keysToArray – Iterates over tree, starting from min node, by calling successor n times, each time adding the key of the current node to an array, finally returning the array. We proved in class that iterating over tree by calling successor is like iterating inorder. O(n)
* infoToArray – Iterates over tree, starting from min node, by calling successor n times, each time adding the value of the current node to an array, finally returning the array. We proved in class that iterating over tree by calling successor is like iterating inorder. O(n)
* size – Returns size. O(1)
* getRoot – Returns null if tree is empty, otherwise, returns root node. O(1)
* prefixXor – We walk down the height of the tree in the following manner:
* If the key of the current node is larger than k, we move on to the left child.
* Otherwise (since the node and its left subtree contain keys smaller or equal to k), we add to our trueCount the trueCnt of the left child of the current node (plus one if the value of the current node is true). If the current node’s key is equal to k, we break out of the loop, otherwise, we move on to the right child.

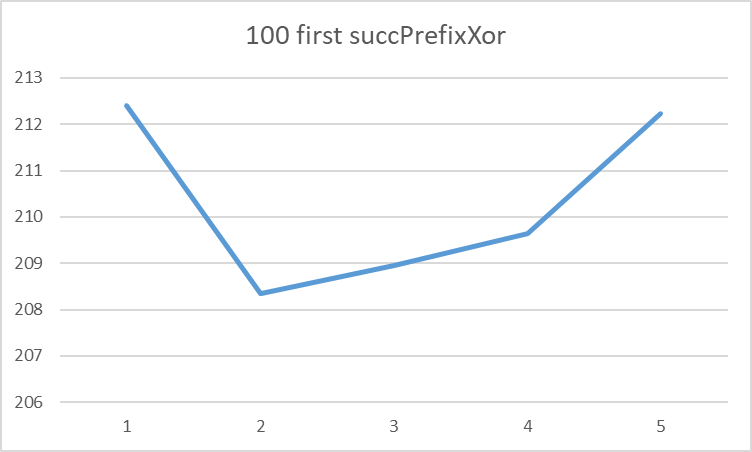
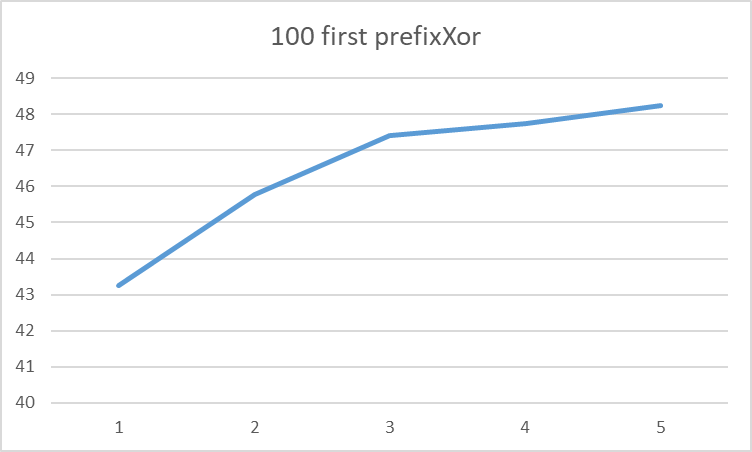
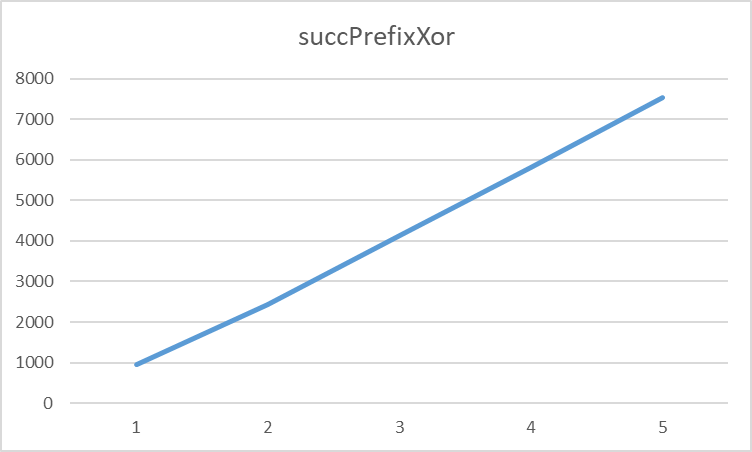
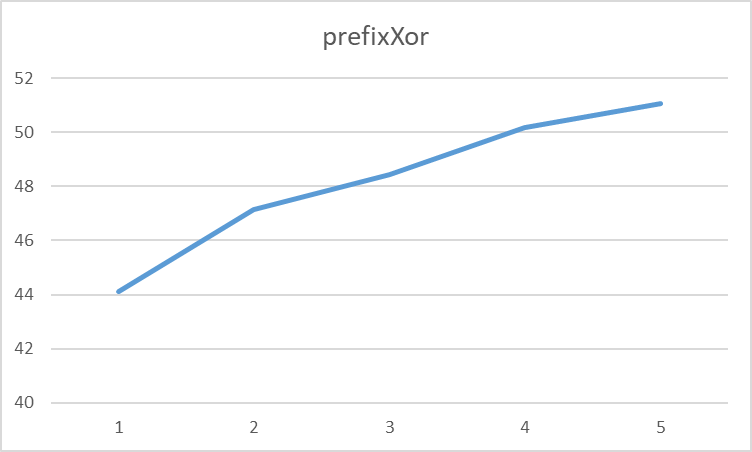
Finally, we return true if and only if trueCount is odd. In worst case, we walk down the height of the tree. O(log n)

* successor – If node has right child, then we move to the right child and left as much as possible finally returning the node which is reached. Otherwise, we move up from the node until the first time we turn right, returning the node reached. In worst case, we walk the height of the tree. O(log n)
* succPrefixXor – Counting number of nodes whose value is true starting from min and calling successor until node with key k is reached. Finally, returning true if and only if number of trues counted is odd. In worst case, node with k is the max, therefore function will call successor n times – which we proved in class is like in-order traversal. O(n)

כיוון שראינו שההרצות הראשונות מניבות תוצאות לא קונסיסטנטיות, הרצנו כל ניסוי 50 פעמים מבלי למדוד את התוצאות ולאחר 1000 פעמים שבהם מדדנו את התוצאות. לבסוף לקחנו את ממוצע של כל קטגוריה. המדידות הם בננו שניות.

**ניסוי 1:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| מספר סידורי | עלות prefixXor ממוצעת  (כל הקריאות) | עלות succPrefixXor ממוצעת  (כל הקריאות) | עלות prefixXor ממוצעת  (100 קריאות ראשונות) | עלות succPrefixXor ממוצעת  (100 קריאות ראשונות) |
| 1 | 44.117 | 955.9 | 43.265 | 212.394 |
| 2 | 47.133 | 2426.102 | 45.766 | 208.351 |
| 3 | 48.444 | 4141.626 | 47.41 | 208.944 |
| 4 | 50.162 | 5817.721 | 47.74 | 209.646 |
| 5 | 51.058 | 7535.881 | 48.234 | 212.23 |



התוצאות מראות:

* מגמה שנראית "לוגריתמית" עבור הסיבוכיות הממוצעת של הפעולה prefixXor (בדומה לניתוח התאורטי).
* מגמה שנראית "ליניארית" עבור הסיבוכיות הממוצעת של הפעולה succPrefixXor (בדומה לניתוח התיאורטי).
* מגמה שנראית "לוגריתמית" עבור הסיבוכיות הממוצעת על 100 הקריאות הראשונות של הפעולה prefixXor. בפעולה של prefixXor מה שקובע את זמן הריצה הינו העומק של האיבר בעל המפתח k בעץ. כיוון שאין קשר בין גודל המפתח לבין העומק שלו בעץ (כאשר ההכנסה היא רנדומלית) הגיוני שהממוצע על 100 הקריאות הראשונות יהיה דומה לזה של כל הקריאות.
* מגמה שנראית "קבועה" (יותר בא לידי ביטוי בהסתכלות על המספרים – הם כמעט ואינם משתנים) עבור הסיבוכיות הממוצעת על 100 הקריאות הראשונות של הפעולה succPrefixXor. הגיוני שהסיבוכיות תהיה קבועה כיוון ש100 האיברים הקטנים ביותר בכל עץ AVL נמצאים באותו מיקום יחסי ביחס לאיבר המינימלי שממנו מתחילים את הפעולה succPrefixXor.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| עלות הכנסה ממוצעת  מספר סידורי | עץ AVL  סדרה חשבונית | עץ ללא מנגנון איזון  סדרה חשבונית | עץ AVL  סדרה מאוזנת | עץ ללא מנגנון איזון  סדרה מאוזנת | עץ AVL  סדרה אקראית | עץ ללא מנגנון איזון  סדרה אקראית |
| 1 | 57.458 | 1051.87 | 42.927 | 24.852 | 81.283 | 47.975 |
| 2 | 58.8 | 2244.526 | 44.083 | 26.508 | 87.244 | 54.475 |
| 3 | 60.071 | 3420.319 | 45.46 | 28.744 | 92.417 | 58.85 |
| 4 | 61.044 | 4585.959 | 46.105 | 29.03 | 98.206 | 63.284 |
| 5 | 62.523 | 5738.651 | 48.058 | 29.171 | 100.566 | 65.614 |

**ניסוי 2:**

**סדרה חשבונית:**

ניתן לצפות שעלות הכנסה ממוצעת בעץ ללא מנגנון איזון יהיה גרוע בהרבה מאשר בעץ AVL כאשר הסדרה המוכנסת היא סדרה חשבונית מכיוון שהעץ ללא מנגנון איזון יראה כמו שרוך וכל הכנסה תיכנס לסוף של השרוך. תוצאות הניסוי מאששות את האינטואיציה.

**סדרה מאוזנת:**

ניתן לצפות שעלות הכנסה ממוצעת בעץ ללא מנגנון איזון יהיה טוב יותר מאשר בעץ AVL כאשר הסדרה המוכנסת היא סדרה מאוזנת מכיוון שהעץ ללא מנגנון איזון יהיה מאוזן ופעולת ההכנסה שלו היא יותר פשוטה מאשר של עץ AVL (שבודק כל מיני תנאים). תוצאות הניסוי מאששות את האינטואיציה.

**סדרה אקראית:**

תוצאות הניסוי עבור סדרה אקראית הן מפתיעות בכך שעלות הכנסה ממוצעת בעץ לא מאוזן היא טובה יותר מאשר בעץ AVL. כנראה שסדרה אקראית מתפלגת בצורה יחסית שוויונית ולכן גם עץ ללא מנגנון איזון יהיה מאוזן יחסית על סדרה אקראית.