Ben-Gurion University of the Negev

Data Structures
202.1.1031

Assignment No. 3

Responsible staff members:
Or Sattath (sattath@bgu.ac.il) Dor Amzaleg(amzalegd@post.bgu.ac.il)

Authors:
Roni Gonen (207195868) Hadas Fire (315233676)

Publication date: May 4th, 2023
Submission date: May 18th, 2023

0 Integrity Statement

I, Roni Gonen 207195868, certify that the work I have submitted is entirely my own. I have not received any part of it from any other person, nor have I given any part of it to others for their use.
I have not copied any part of my answers from any other source, and what I submit is my own creation. I understand that formal proceedings will be taken against me before the BGU Disciplinary Committee if there is any suspicion that my work contains code/answers that is not my own.
If you have relied on or used an external source, you must cite that source at the end of your integrity statement. External sources include shared file drivers, Large Language Models (LLMs) including ChatGPT, forums, websites, books, etc.

I, Hadas Fire 315233676, certify that the work I have submitted is entirely my own. I have not received any part of it from any other person, nor have I given any part of it to others for their use.
I have not copied any part of my answers from any other source, and what I submit is my own creation. I understand that formal proceedings will be taken against me before the BGU Disciplinary Committee if there is any suspicion that my work contains code/answers that is not my own.
If you have relied on or used an external source, you must cite that source at the end of your integrity statement. External sources include shared file drivers, Large Language Models (LLMs) including ChatGPT, forums, websites, books, etc.

1 AVL trees

1. *(8 points)* Let x be a leaf in an AVL tree. Prove that $d(x) \geq \lceil \frac{h}{2} \rceil$. Here, $d(x)$ is the depth of x, and h is the height of the tree.

2. *(2 points)* Give an example for a class of AVL trees (this class of trees was mentioned in class) in which the bound above is sharp (namely, there exists a leaf x with $d(x) = \lceil \frac{h}{2} \rceil$). There is no need to justify your answer; the answer should take at most 1 line.

3. *(10 points)* Explain how it is possible, by using additional space, to support the operation $find(x, k)$ that returns the first k elements greater than the key x in an AVL tree, in time complexity $O((\log n) + k)$. If the number of elements with a key greater than x is less than k, the operation should return all these elements.

Answer to Question 1:

1. Let x be a leaf in an AVL tree. We need to prove that $d(x) \geq \lceil \frac{h}{2} \rceil$. Here, $d(x)$ is the depth of x, and h is the height of the tree. We will prove by induction on n.
   Base: for h=0 the tree is only the root node and therefore its depth is 0 so $0 \geq \lceil \frac{0}{2} \rceil = 0$ as required.
   for h=1, there are 3 options of trees- the first option is the root node and its left child only, the second option is the root node and its right child only and the third option is the root node with its two children. Therefore, for each leaf, its depth is 1 and $1 \geq \lceil \frac{h}{2} \rceil = 1$ as required.
   Induction assumption: Let $n \in N$. We suppose that for each leaf x at the lowest level in the AVL tree it holds that $d(x) \geq \lceil \frac{h}{2} \rceil$. And for every leaf y one level before the lowest in the AVL tree it holds that $d(y) \geq \lceil \frac{h}{2} \rceil$. (We know for sure that all the leaves in AVL tree are in these two lower levels only from the AVL feature which says that: $|h(z.left) - h(z.right)| \leq 1$ when z is a node in the tree, and it follows that the difference in the depth of each leaf is less than/equal to 1.
   The induction step: We need to prove that $d(z) \geq \lceil \frac{h+1}{2} \rceil$ for every leaf z in the tree. the depth of each leaf z in the tree is $d(x) = d(y) + 1$ or $d(x) + 1$ (from the saved AVL tree format).
   We divide into cases and prove:
   case one: if $d(z) = d(x) + 1$ then the height is $h + 1$ and therefore from the induction assumption it holds that $d(x) + 1 \geq \lceil \frac{h}{2} \rceil + 1 \implies d(x) + 1 \geq \lceil \frac{h}{2} \rceil + \frac{2}{2} \implies d(x) + 1 \geq \lceil \frac{h+2}{2} \rceil \geq \lceil \frac{h+1}{2} \rceil \implies d(z) \geq \lceil \frac{h+1}{2} \rceil$ as required.
   case two: if $d(z) = d(x) = d(y) + 1$ then the height is $h + 1$ and therefore from the induction assumption it holds that $d(y) + 1 \geq \lceil \frac{h}{2} \rceil + 1 \implies d(y) + 1 \geq \lceil \frac{h}{2} \rceil + \frac{2}{2} \implies d(y) + 1 \geq \lceil \frac{h+2}{2} \rceil \geq \lceil \frac{h+1}{2} \rceil \implies d(y) + 1 \geq \lceil \frac{h+1}{2} \rceil \implies d(z) \geq \lceil \frac{h+1}{2} \rceil$ as required.

2. The answer is an AVL tree with minimum nodes for a certain height.

3. In order to successfully perform the find operation at this runtime, we will have an AVL tree and a LinkedList that will contain all the keys of the nodes of the tree sorted in ascending order according to their key, where each Link in the LinkedList holds a key of a specific node in the tree and the Links are sorted according to the keys in ascending order. In addition, each node in the tree will have a pointer to its corresponding Link in the LinkedList. In terms of implementation, every time we add a node to the tree, we will also add a Link with its key to the appropriate place in the LinkedList.
   We need the K elements greater than X, so we will run through the tree and look for X (this happens in time of $O(\log n)$- search function in an AVL tree). When we find X in tree, we will access through its pointer to its place in the LinkedList and then we will return the K elements that follow it in the LinkedList (they will necessarily be elements whose keys are greater than the key of X because our preserved assumption is that the Linkedlist is sorted in ascending order). It will happen in $O(k)$. In order to return these elements, we will create a new LinkedList and every time we go over a certain Link from our existing LinkedList (an element from the K elements), we will create a new Link with its data and insert it into our new LinkedList. And finally we will return the new LinkedList. Likewise, the existing linked list will not change. If there will be less elements than K then we will run to the end of our LinkedList and return them in the same way.$\Longrightarrow$ In any case the worst running time will be: $O((\log n) + k)$ -search for the node of X and then run through the following K (or less) Links in the LinkedList and put them into a new LinkedList that will be returned at the end of the operation. $\Longrightarrow O((\log n) + k)$.

2 Data structure by requirements and time complexity constrains *(25 points)*

Implement, for a client who owns a products factory, a data structure that supports the following operations in the given time complexities:

Operation	Description	Time complexity
$Init()$	Initiate an empty data structure.	$O(1)$
$Insert(id, quality)$	Insert a product with a unique identification number id (unbounded natural number), and its quality $quality$ (integer between 0 to 5), to the data structure.	$O(\log n)$
$Delete(id)$	Delete the product with the unique identification number id from the data structure (if there exists such an element in the data structure).	$O(\log n)$
$MedianQuality()$	Return the median quality of the products in the data structure. The median is defined as the $\lceil \frac{n}{2} \rceil$-th smallest element.	$O(1)$
$AvgQuality()$	Return the average quality of the products in the data structure.	$O(1)$
$JunkWorst()$	Delete all the elements with quality 0 from the data structure. This operation should return a linked list with all the deleted elements.	$O(1)$
$Printout()$	Print all the elements in the data structure, ordered by identification number, from the smallest to the biggest.	$O(n)$

Describe briefly your data structure.
Give an algorithm and a short time-complexity analysis to every operation.

Answer to Question 2:

- The data structure consists of:
  - One AVL tree containing all products with quality=0 and sorted by id. In addition, each node in the tree has a pointer to its Link in the LinkedList (explained later).
  - Another AVL tree containing all the products with a quality other than 0 (a quality of 1,2,3,4 or 5) and sorted according to the id.
  - A two-way LinkedList containing only the products with quality=0 (without regard to sorting).
  - An array of size 6 where each index in the array (0-5) represents the quality. The array stores in each index i the amount of products with quality i.
  - A field of type int that stores the number of products in the data structure (for the sake of explanation below we will call it Int AmountOfProducts).
- $Init()$:
  We create two AVL trees (in a time of $O(1)$), a LinkedList (in a time of $O(1)$), an array of size 6 and initialize it with zeros (the creation of the array and its initialization takes $O(7)$ because the size of array is constant(6), which is $O(1)$), and create a variable of type int that is initialized with zero (in a time of $O(1)$) $\Longrightarrow$ total is $O(1)$
- $Insert(id, quality)$:
  First we will update our AmountOfProducts field and increase it by one (in a time of $O(1)$).
  After that we will update our array in the index of our given quality and increase it by one (accessing the position in the array and updating the value in it happen in a time of $O(1)$).
  And then, the continuation of the insertion depends on the quality of the product we want to insert (whether the quality is 0 or different from zero), so we will check the quality of the product:
  - If the product quality is zero then:
    We will create a node of a product with the given id and quality and insert it into the AVL tree containing the products with quality 0 using the insert operation of the AVL tree (in a time of $O(\log n)$).
    After that we will create a new Link with our product details (id and quality) and insert it to our LinkedList which only contains products with 0 quality (in a time of $O(1)$) and we will update the pointer of our node in the tree to be the link we added in the linked list (in a time of $O(1)$).

- If the product quality is different from zero (1,2,3,4 or 5) then:
  We will create a node of a product with the given id and quality and insert it into the AVL tree containing the products with quality different than 0, by using the insert operation of the AVL tree (in a time of $O(\log n)$).

$\Longrightarrow$ total is $O(\log n)$

- *Delete*(*id*):
  First we will update our AmountOfProducts field and decrease it by one (in a time of $O(1)$).
  After that, the deletion continued depending on the quality of the product we want to delete (whether the quality of the product is 0 or different from 0). We do not receive a pointer to the node in the tree, but the id of the product, so we will have to search the tree for the node with the given id, so when we find it, we can access its quality. We will look for the id in our two trees in advance, because we do not know what its quality is. (A search in each tree takes $O(\log n)$).
  If we did not find the given id in the trees, we are done and therefore the operation is $O(\log n)$).
  If we found the appropriate node in one of the trees (ie our product) we will first check its quality and update our array in the index of quality and decrease it by one (in a time of $O(1)$).
  And then:
  - If the product quality is zero then:
    We found the node with given id so we will use its pointer to access the LinkedList that contains only products with quality 0 and delete it from there (our LinkedList is two-way, so this will happen by updating pointers and therefore in a time of $O(1)$).
    Then, we will delete this node (that is, this product) from the tree that contains the products with quality 0, by using the delete operation of the AVL tree (in a time of $O(\log n)$).
  - If the product quality is different from zero (1,2,3,4 or 5) then:
    We will delete the node we reached in the search (by the id) from the tree where we are, the tree of the products with the quality different from 0. We will do this using the delete operation of the AVL tree (in a time of $O(\log n)$).

  $\Longrightarrow$ total is $O(\log n)$

- *MedianQuality*():
  We have an AmountOfProducts field, so we divided it by 2 (if it is an odd number, then we add 1), and we save it in a new variable- for example int helpMedian. Next, we'll start running our array that contains the amount of products of each quality. Each cell we pass through during the run through the array, we will subtract from helpMedian the value in the cell and when we reach 0 or -1 we will know that we have reached the cell in the array whose index is the median and we will return it. Our array is size 6, so the running time is $O(1)$. The subtractions/comparisons are $O(1)$ $\Longrightarrow$ total is $O(1)$

- *AvgQuality*():
  We will run on our array (which holds the quantity of products of each quality) starting from index 0. We will initialize a new variable "Output" with 0 and each cell we pass through in the array, we will multiply the value in its cell with its index and add it to the output. After finishing the run on the entire array, we will divide the output we got by the general amount of products (we have such a field in the data structure) and return what we received- this will be the average of the qualities. We only run on a constant size array $O(1)$. In addition, the multiplication and addition operations takes $O(1)$ $\Longrightarrow$ total is $O(1)$

- *JunkWorst*():
  We have a LinkedList that contains the products with quality 0 so we will initialize a new LinkedList with a pointer to the first Link of our existing LinkedList so we can return it ($O(1)$). Then we will update the root of the AVL tree that contains the products with quality 0 to be Null ($O(1)$) and reset our LinkedList- we will initialize its "First" field with null ($O(1)$). And finally, the new LinkedList we defined contains the products with quality 0 so we will return it ($O(1)$) $\Longrightarrow$ total is $O(1)$

- *Printout*():
  We will perform an InOrder scan of our two trees, inserting each scan into a LinkedList (each scan is performed at the time of $O(n)$ and therefore total time is $O(n)$). So we will have two LinkedLists, which sorted in ascending order by id. After that we will run through the two LinkedLists at the same time, until we reach the end of the longest LinkedList between them ($O(n)$). During the run through the LinkedLists,

we will make comparisons between the id's products of the two different LinkedList and print the products in ascending order (every comparison and every print is $O(1)$) $\Longrightarrow$ total is $O(n)$

3 Merge/Split

1. *(10 points)* Let T_1, T_2 be two AVL trees with heights h_1, h_2 respectively, such that $\max(T_1) < \min(T_2)$ (every key in T_2 is greater than every key in T_1). Let x be an additional key, which satisfies $\max(T_1) < x < \min(T_2)$. Describe an algorithm for merging T_1, x, T_2 to a new AVL tree (the new AVL tree should contain all the keys in T_1, all the keys in T_2, and x), in time complexity $O(|h_1 - h_2|)$. **Guidance:** If $h_1 > h_2$, try to insert the node x and the tree T_2 to the appropriate point in the rightmost path from the root to a leaf in the tree T_1. Show that the tree we get by this insertion can be balanced by rotations. Think about the symmetric case ($h_1 \leq h_2$).

2. *(5 points)* What is the height of the merged tree in your solution?

   (a) $\max\{h_1, h_2\}$

   (b) $\max\{h_1, h_2\}$, or $\max\{h_1, h_2\}+1$

   (c) Can be as high as $\max\{h_1, h_2\} + |h_1 - h_2|$

   (d) None of the above.

3. *(20 points)* The operation $Split(T, k)$, on an AVL tree T and a key k, returns two AVL trees T_1, T_2, such that T_1 contains all the keys in T which are less or equal to k, and T_2 contains all the keys in T which are greater than k. Describe an implementation to the operation $Split(T, k)$ in time complexity $O(\log n)$, where n is the number of elements in the tree T.
   Guidance:

   - Think of a way to get a list of nodes and sub-trees that should be contained in T_1, and a list of nodes and sub-trees that should be contained in T_2.
   - Find an efficient ordering to merge those sub-trees and nodes, using your algorithm from the previous section. Note that a wrong ordering might give you a running time of $O(\log^2(n))$.
   - Analyze the time complexity of your algorithm. The time complexity of the merge algorithm from the previous section and the upper bound you found for the height of the merged AVL tree, should be used in this analysis.

4. *(10 points)* Describe a data structure that stores elements, where every element has a key (natural number) and a color (black or white). The data structure should support the following operations:

Operation	**Description**	**Time complexity**
$Add(x)$	Insert the element x (where x has fields $x.key$ and $x.color$).	$O(\log n)$
$Color(k)$	Return the color of the element with the key k (return null if no such exists).	$O(\log n)$
$FlipColors(k)$	Flip the colors of all the elements with a key less or equal to k. You may assume that there is an element with key k in your data-structure.	$O(\log n)$

   There are various solutions. Some of them use the previous sections. Of course, all correct solutions will be accepted.

Answer to Question 3:

1. Suppose $h_1 > h_2$. In this case we will traverse down the tree T_1 from the root to the right $|h_1 - h_2|$ times. We will call this node V. Now we will place the node X where V was placed, and place the node V with the rest of his sub-tree as X's left son. In addition, we will place the root of the tree T_2 (and the rest of his sub-tree) as X's right son.
   The sub-tree that his root is X, is keeping the properties of an AVL tree because the node V and all the nodes in his sub-tree are smaller than X, and are placed to the left of X. Moreover, all the nodes in the tree T_2 are bigger than X and are placed to the right of X.
   At present, the height of the new full tree is $h_1 - h_2 + 1 + h_2$ (+1 because we added the node X), that is $h_1 + 1$. Now, if we violated the balance of the AVL tree, we can perform rotations in order to rebalance the tree. These rotations are legal because we violated the balance only in one node (X's father) and two of his sons are legal AVL trees. Additionally, the height of X's right son is h_2 because originally this is the tree T_2, and the height of X's left son is h_2 or $h_2 - 1$ because it was part of a legal AVL and we traversed down $|h_1 - h_2|$ times. So the maximum difference between the heights of both trees is 1.
   In the case where $h_2 > h_1$ we will perform the same stages, only this time we will traverse down and to the left of the tree T_2, $|h_1 - h_2|$ times, call this node V and place X in his place. And this time V will be X's right son and T_1 will be X's left son.
   The time complexity of inserting the node is the time it takes to get to the wanted place so it is $O(|h_1 - h_2|)$ as we explained before and changing the pointers is $O(1)$.
   The time complexity of the rotations is $O(1)$ because the number of rotations is constant, therefore the total time complexity is $O(h_1 - h_2)$.

2. The height of the merged tree in our solution is: **(b)** $\max\{h_1, h_2\}$, or $\max\{h_1, h_2\}$+1

3. We will create 2 stacks, one will hold the nodes that will eventually be in T_1, we will call it S_1, and the second will hold the nodes that will be in T_2, we will call it S_2.
   We will traverse down the tree while comparing the keys to K. If the current node's key is smaller of equal to K, we will insert this node to the stack S_1, and after that we will insert his left sub-tree to S_1 and move on to his right son. If the current node's key is bigger than K, we will insert this node to the stack S_2 and after that, we will insert his right sub-tree to S_2 and move on to his left son. We will continue traversing down the tree while comparing and inserting the elements to the appropriate stack (like explained before) until we reach a leaf, which we will insert to the appropriate stack as well.
   At this point, we have 2 stacks, each one of them holding nodes and sub-trees.
   In the stack S_1 we will pop the last element out and call it T_1, pop another element out and call it T_2, and pop another element out and call it X. Now we will merge these elements to a single tree using the function Merge that we wrote in section 1 and send T_1, T_2, and X (Merge(T_1, T_2, X)). We will push this new tree back to the stack and continue to do so until the stack holds only one element.
   In the stack S_2 we will perform the same actions only this time the first element we will pop out will be T_2, the second will be T_1 and the third will be X.
   When finishing these steps we will have two stacks, each holding an AVL tree.
   Time Complexity Analysis:
   While traversing don the tree, the number of steps it takes is the height of the tree, therefore it is $O(\log(n))$. Each stack will hold up to $\log(n)$ elements. The MERGE function's time complexity depends on the difference between the height of the trees. The maximum difference is $\log(n)$. This is why in the worst case the time complexity is $O(\log(n))$- this only happens when we merge the whole tree with an empty tree. The biggest tree in the stack will always be added first and merge last so it can only happens once.
   In conclusion, the overall time complexity is $O(\log(n))$.

4. The data structure we will use is a data structure that will hold 2 AVL trees that each of them is sorted by the keys. One tree holds all the keys that their color is black, and the second tree holds all the keys that their color is white.

   The function Add(x):
   We will check the color of x and in accordance we will add his key as a node to the appropriate tree. The insertion to the tree will be like regular AVL insertion.
   Time complexity of AVL insertion is $O(\log(n)$ and that is why our Add function is also $O(\log(n))$.

The function Color(k):
We will search for the key 'k' in both trees until we find it. When we find the node, we will remove it from his current tree, create an element with the key 'k' and the color of his opposite tree and add it to the opposite tree.
The time complexity of searching for a key in an AVL tree is $O(\log(n))$, removing (changing pointers) is $O(1)$ and adding while using the function Add(x) is $O(\log(n))$. The overall time complexity is $O(\log(n))$.

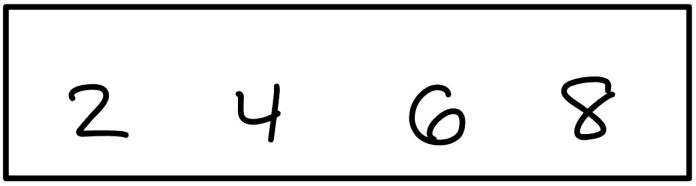
The function Flip Colors(k):
We will call the function Split that we wrote above to each of our trees, in order to get 4 separated trees. One with keys smaller or equal to 'k' and the other one with keys bigger than 'k', in both colors. We will take the trees with the keys smaller or equal to 'k' from both colors and merge each of them with the tree that has the keys bigger than 'k' in the opposite color.
The time complexity of Merge is $O(|h_1 - h_2|)$, and in the worst case the difference between the heights can be $\log(n)$ so together, $O(\log(n))$. Every function will be called twice so overall the time complexity is $O(\log(n))$.

4 B-trees *(10 points)*

Give an example to a B-tree T with minimal degree $t = 3$, and (different) elements x, y which are not in the tree, such that the tree we get by inserting x and then y to T is different than the tree we get by inserting y and then x to T.
Add pictures of the tree T, and the trees after inserting x, y in both cases.
You should give an example with the smallest number of elements possible.

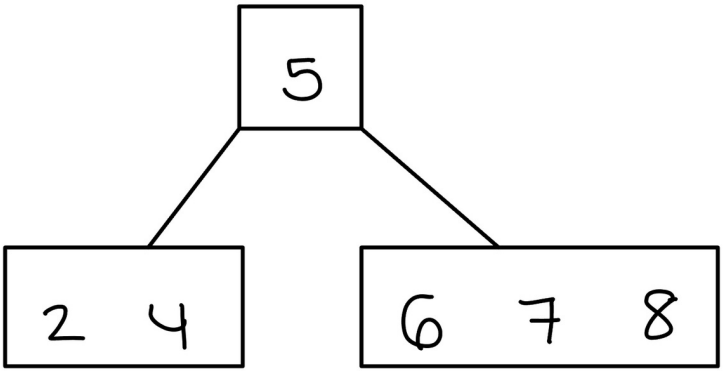
Answer to Question 4:

Picture of the tree T:



The element x is: 5
The element y is: 7

Picture of the tree after inserting x and then y:



Picture of the tree after inserting y and then x: