AADS Project 1 Report

Syed Mustafa Umar
A0209677N

This report will mainly encompass my design choices, the time and space complexities of those choices, and a discussion about how the implementation can further be expanded upon.

Some Thoughts

While the assignment prompt sounds quite challenging, I think it was important to take a step back for a bit and understand what is happening underneath the structure. While working on the project, I came to realise that a lot of certainties that I had developed from the previous assignment (in which we were to implement the data structure without the amortized time constraint) were, in fact, not the case. Furthermore, while implementing the algorithm for append, I had to tweak around quite a bit to get just the right condition for the heavy cost operation. Since the main purpose of the assignment was the append operation, I'll go over apply and update briefly, and then move over to append. However, before I do that, I'll go over the structure you obtain when you call IArr. This is important to know before we discuss other functions because it plays a role in all of them.

Structure after Append

There are two main distinctions I want to create here. One being the IArrLeafs and the other IArrNodes. For the leaves, the append operation is quite simple. All we do is create an IArrNode2 and then add the leaf as one element and the array to be appended as the other. That is, the result looks like a pyramid structure wherein the top of the pyramid is the IArrNode2 itself and the previous leaf and new array is on the left and right respectively. The reason we do this is because the array cannot be added as an element to the leaf since a leaf is meant to consist of nodes. Note that the algorithm is designed in such a way that the height of the tree will always be according to the highest array slice attached to the node. The reason I chose to do

this implementation is because I thought it was simple to understand and it would be very rare for us to be looking at the leaves in an array. So, even if it wasn't the most ideal solution (such as in the case of a very large array being appended to the leaf causing a huge dis-balance), it is a rarity to occur.

The nodes work a bit more differently. If we have an IArrNode1 as the root of the array and append another array to it, we can try and re-use some of the code. That is, we create an IArrNode2 where the first element is simply the element of the previous IArrNode1 and the second element is the second array. This way, we don't create extra higher roots but rather manipulate the existing structure. That is, instead of creating a pyramid-like structure similar to the leaf case, we just increase the capacity of the root by making a new one and then adding the previous and new elements to that. We do something similar for IArrNode2 where we create an IArrNode3. The same goes for IArrNode3. However, in the case of IArrNode4, since our bucket sizes are limited to 4, we are forced to create an IArrNode2 and then construct the pyramid-like structure mentioned in the leaf case. Now, we move towards the apply and update operation for an appended or unappended array.

Apply and Update

These functions are relatively easy to understand once you get whats happening. We'll be discussing two cases, the IArrLeafs and IArrNodes. In the previous assignment, we could use modulus to send information down the array tree (the apply function, that is, the array(index) the brackets we use to access an element in the array). We could do this because the array tree was balanced from the left. That is, the array was always filled from left to right. So, we could use the already-given **withinBoundsWithIndexSplit** to determine the bounds of the array and send it to the respective child accordingly. However, since the tree is not left-balanced anymore (due to the way we've implemented the append operation), we need to make changes accordingly.

In the case of the IArrLeafs, the operations of apply and update remain the same as in the previous assignment. That is, we can simply find the element being looked for by calling the modulus in the apply or update index and adjust the retrieval or new value accordingly. The reason we don't need to do any fancy calculations here is because we already know that there won't be anything below the leaf, so we need not determine which node to send the value down to. This is the job for the IArrNodes, which we'll now move to.

The IArrNodes, as mentioned, work a bit differently than the leaves since the nodes need to decide which child to send the apply or update index to. The main

value that'll help is here is the length (or size) of each child of the node. We know that the node now is not necessarily left-balanced, so we cannot use the **withinBoundsWithIndexSplit** function to determine which child to send the retrieval or update to. However, we do know that the length of each sub-array is stored in the children nodes. So, using that and the input index, we can determine which sub-array to call the function array. Since this implementation uses information that's already stored in the sub-array, this is essentially of constant time for each node and logarithmic overall. So, I won't try to argue that there's a better way to do this because I can't think of anything better. Now, we'll move towards the main implementation (which is a relatively short but significant one), and that is the append operation.

Append

Again, we'll split the append operation into two cases, one of IArrLeafs and the other of IArrNodes. The append operation for the leaves is all the same but there are slight differences for the nodes.

For the leaves, the append operation first checks if the array-to-be-appended is empty. If it is, we can simply return the current array. If not, we move forward to create the new array (as discussed in the Structure after Append section). In this array, it makes an extra check to ensure the array structure is not too distorted (the name of the condition is **condForAppend**). If the new array is too distorted (meaning that it is too unbalanced), a cleanup takes place where the iterator of the array is used to reconstruct the whole structure. The new array is now left-balanced and ready to be manipulated. Note that I'll talk about the specification of the condition in the next section. Also, notice that this check is where the amortization comes into play. Again, we'll talk about this a bit later.

For the nodes, the append operation also first checks if the array-to-be-appended is empty. It it is, we can just return the current array. If not, we move forward to create the new array (again, discussed in the Structure after Append section). We again make an extra check to see distortion within the array tree. If it is too distorted, we use the iterator to construct it from scratch.

There are probably ways in which the append operation could be better done. The main problem I foresee with the current implementation is that there are no modifications within the previous or the new array. That is, we could possibly use points within an array so instead of always creating a new node, if the input were the right height, we could just add it to one of the existing sub-trees in the array so that we do not keep increasing the height of the array. However, while this implemen-

tation does somewhat makes sense, it'll be difficult to figure out all the cases we'd need to account for. There are probably other ways to better solve this problem but I think the current implementation is one that is easy to grasp, and works efficient enough. Now, let's talk about the condition that we've discussed previously that's used to determine the cleanup operation.

Condition in Append

The **condForAppend** function in the IArr trait is the one that is used for all the append operations in the leaves and nodes. This condition basically returns true if twice the log (base 4) of the length of the tree is is less than the height of the tree. Now, I tried a lot of values for this and this is the final formula I came upon. It gave me the fastest time for appending and then trying to retrieve some items from the array. Lets try to understand the formula now. The first part before the less than basically uses the size of the current items stored in the array (the length of the array) and then takes the log of that size. Then, if the height of the tree (which determines the capacity) is greater than twice the log, then we need to adjust the array structure such that it stops being too costly. Note that if this condition fails (which will be the case most of the time) the cost of the append operation is constant. If it does not, then we need to re-construct the tree which probably takes $O(nlogn)$ (I might be a bit incorrect in this but I'm using the already provided for array construction which shouldn't be too costly). However, what is of interest here is that we are using log and height as a determinant. That is, as the array size grows, the need to re-construct the array decreases. This is where amortization takes place. When the length is relatively low, the condition might be executed more often since major appends will distort the array tree heavily. However, as the length grows, these appends will have lesser impact so the cleanup will adjust accordingly. That is, a bigger array will have greater capacity so a lesser possibility of high distortion. Furthermore, since the dependence of determining the distortion is according to the $log2$ function, we can argue that the algorithm is near logarithmic amortized complexity.

Again, there are many ways this could have been implemented. There are other methods for cleanup as well and one possible improvement I can think of is cleaning up sub-arrays rather than the whole array itself. If a few bad nodes were appended to an array (bad nodes meaning nodes that caused a lot of distortion), it might be smarter to construct a tree from those nodes to better balance the tree overall than to disturb all of the existing nodes and re-work the whole array. However, since this assignment does not require for us to dig too deep into sub-trees and more specific (and so, complex) cases, the algorithm that is good enough.

Furthermore, one part that I did find confusing and still haven't figured an answer to is why the exact formula that I've written above. At first I thought that the log of the length + 1 less than the height would be a good condition since this ensures that the height does not deviate too much from the length. However, I later realized that this condition is too restricting and will keep on cleaning up the tree even when it is not needed. After running a few values, I came upon the addition of 10 as the best case scenario at the time. However, upon digging deeper, I understood that this would be fine for smaller arrays but the constant 10 might cause problems for arrays with millions and billions of data points. So, then I finally came upon the two times log (base 4) of the length of the array as a determinant to compare to the height of the tree. It is also important to note that we'll need to adjust the formula for different log bases as well, which was something interesting I noticed.

Conclusion

Similar to the other assignments in the course, I had to research a bit on the subject and had to try and draw images of how the solution would look like. I always appreciate it when an assignment allows me to visualize what I am trying to code out, and this was no different. This was a different assignment though, where it was asking us to use the bare minimum functionality to construct a very simple - yet complex - data structure. I liked how I was able to implement one data structure using another and, also, what persistent data structures are. Furthermore, it was nice trying out different solutions until I came upon one that works for me. I do think that this might have different results for different machines, so I am a bit concerned on that end. However, it provided the fastest results on my machine so I am hopeful that it will perform well for others. Overall, this was an interesting project where I had to draw a lot of diagrams to weed out any possible edge cases and play around with the formula to find the right mix of ingredients.