Algorithms and analysis - Assignment 1

Student ID(s): s3720145, s3795200

Student Name(s): Ivan Bulosan, Sheen Wey Chua

I (or We) certify that this is all my (our) own original work. If I took any parts from elsewhere, then they were non-essential parts of the assignment, and they are clearly attributed in my (our) submission. I (We) will show we I (we) agree to this honor code by typing ``Yes'': *Yes*.

Task B: Evaluate your Data Structures

Theoretical Analysis

Given a multiset(s) of size n, (n1 & n2 for intersect)

Array Multiset

Scenarios	Best Case	Worse Case	Big O
Growing Multiset	C(n) = n Using a multiset with 3 elements {a:1, b:1, c:1}, the best case is when the element to be added is “a” since it is located at the first position. It will not execute the for loop for n times, it executes once and increments the number of elements by 1, and exits the loop. Hence {a:2, b:1, c1}.	C(n) = 2n Using a multiset with 3 elements {a:1, b:1, c:1}, the worst case is when the element to be added does not exist in the multiset. For example, when elem = “d”, since the array is fixed size, it has to create a new array and copy all the values back into each respective position and add the new elem behind. Another scenario is or when the element to be added is located at the end of the array, using the multiset example, when elem = “c”, it has to loop through n size to increment.	O(n)
Shrinking Multiset	C(n) = 1 Using a multiset with 5 elements {a:2, b:1, c:1, d:3, e:5}, the best case is when the elem to be	C(n) = n^2 Using a multiset with 6 elements {a:2, b:1, c:1, d:3, e:5, f:1}, the worst case is when the elem to be removed is located	O(n^2)

	removed is located at the first position and it has to be more than 1, which is “a” in this case as its number of instances = 2. This means it will not execute the for loop for n times, it stops after decrementing the number of instances of elem “a”. The result would be {a:1, b:1, c:1, d:3, e:5}.	at the end and the elem . This means it will execute the first for loop for n times and if the number of instances is 1, it has to copy all the values except the deleted element into a new array which also executes n times of second for loop. The result would be {a:2, b:1, c:1, d:3, e:5}.	
Intersection	C(n) = $n_1 \times n_2 = n^2$ Using a multiset, s1={a:1, b:1, c:1}, s2={d:1, e:1,f:1}, where $n_1 = 3$ and $n_2 = 3$ the best case is when set 2 does not contain the element in set1, this means it will not execute the for loop after that. However, it will still execute the first for loop and the for loop in the contains method nonetheless.	C(n) = $(n_1 \times n_2 \times n) + n_2 = n^3 + n_2$ Using a multiset, s1={a:3, b:1, c:1}, s2={a:2, b:1, c:3, d:1, e:1}, where $n_1 = 3$ and $n_2 = 5$. The worst case is when set 2 contains the element in set 1, this means it will execute the for loop under to add the number of elements into the new set.	O(n^3)
Print	C(n) $= 2n + 2n^2$ Using a multiset with 5 elements {a:4, b:3, c:2, d:1, e:1}. For my implementation, there is no best case as the nested for loops for sorting are always running until the end of the array. However if the array is already sorted, it will not be executing the assignment operations. The result in the end would still be {a:4, b:3, c:2, d:1, e:1}.	C(n) $= 2n + 2n^2 + 3n = 5n + 2n^2$ Using a multiset with 5 elements {a:2, b:1, c:1, d:3, e:5}. For my implementation, the nested for loops for sorting will always be running until the end of the array. If the array is not in descending order yet, it will execute the assignment operations. The result in the end would be {e:5, d:3, a:2, b:1, c:1}.	O(n^2)

Bst Multiset

Scenarios	Best Case	Worse Case	Big O
Growing Multiset	C(n) = 1 Using a multiset of 3 elements {a:1, b:1, c:1}. The best case is when the item to be added is located at the root of the tree. In this case, “a” is the element stored in the root node. The while loop will stop executing after incrementing the number of instances of “a” in the root node. The result would be {a:2, b:1, c:1}.	C(n) = n Using a multiset of 5 elements {a:1, b:1, c:1, d:1, e:1}. The worst case is when the item to be added is not in the tree and when the tree is either left or right-skewed. This means it has to traverse to the bottom of the skewed tree with height = n and add a new node for the new item. When item = “f”, the result would be {a:1,b:1, c:1, d:1, e:1, f:1}	O(n)
Shrinking Multiset	C(n) = 1 Using a multiset of 5 elements {d:3, a:1, b:1, e:1, f:1}. The best case is when the item to be removed is located at the root with number of instances > 1. The while loop will stop executing after that. In this case, the number of instances of item “d” will be decremented by 1. The result would be {d:2, a:1, b:1, e:1, f:1}	C(n) = 4n Using a multiset of 5 elements {a:1, b:1, c:1, d:1, e:2}. The worst case is when the item to be removed is not in the tree and when the tree is either left or right-skewed. This means it has to traverse to the bottom of the skewed tree with height = n. For example, the item to be deleted is “f”. The result would still be {a:1, b:1, c:1, d:1, e:1}.	O(n)
Intersection	C(n) = $2n_1 + log(n_2)$ Using multisets, s1={a:1, b:1, c:1}, s2={e:1, d:1, f:1} The best case is when set 2 does not contain the element in set 1 and also when set 2 is not a skewed tree. This means for the contains method it only requires log(n) time to search for the item, and also it will not execute the for loop after that.	C(n) = $2n_1 + 5n$ Using multisets, s1={a:1, b:1, c:1, g:1}, s2={d:1, e:1, f:1, g:1}. The worst case is when set 2 contains the element in set 1 and also when set 2 is a skewed tree, this means the contains and search method will traverse through the skewed tree to look for the required item. Moreover, the for loop for adding the elements will also be executed. The result would be	O(n)

	However, it will still execute the for loop in the contains method nonetheless. The result would be newSet = {}	newSet = {g:1}	
Print	C(n) = 2n For this implementation it will still run for n times for the for loop where the n is the highest number of the elements in the multiset.	C(n) = n The worst case will be the same as the best case as for this implementation, it will still run for n times for the for loop where the n is the highest number of the elements in the multiset.	O(n)

Ordered singly linked list Multiset

Scenarios	Best Case	Worse Case	Big O
Growing Multiset	$C(n) = 1$ The best case occurs when inserting a node into an empty LL. This triggers only one comparison and inserts the node.	$C(n) = 3 + 3n$ Example: inserting 3 into multiset {1:2, 2:1}. Worst case occurs when inserting a node that has a larger value than anything currently in the list. The list of size n must traverse all the way to the end to suitably insert.	$O(n)$
Shrinking Multiset	$C(n) = 1$ Example: removing 1 from multiset {1:1,2:3}. Occurs when deleting the head node when it only has an instance count of 1. Only one comparison is made.	$C(n) = 2 + 2n$ Example: removing 3 from multiset {1:1,2:3}. If the item to be removed is not in the list, the whole list of size n must be traversed.	$O(n)$
Intersection	$C(n) = n$ Example: finding the intersect of multiset {1:1,3:3,4:2} and {a:1,b:5}. If an intersect does not exist, the list will keep traversing to the end with only 1 comparison per loop.	$C(n) = 2n$ Example: finding the intersect of multiset {1:1,2:2,3:3} and {1:1,2:2,3:3}. If both multisets are identical 2 comparisons per loop are made until the list reaches	$O(n)$

		the end, resulting in $C(n) = 2n$	
Print	$C(n) = 3n + n\ log(n)$ Best and worst case complexity when printing is the same. $3n$ assignments are made. $n\ log(n)$ is derived from the merge sort algorithm which sorts the nodes in descending order of number of instances. Merge sort also has the same best and worst case time complexity.	$C(n) = 3n + nlog(n)$ Best and worst case complexity when printing is the same. $3n$ assignments are made. $n\ log(n)$ is derived from the merge sort algorithm which sorts the nodes in descending order of number of instances. Merge sort also has the same best and worst case time complexity.	$O(n\ log(n))$

Dual ordered linked list Multiset

Scenarios	Best Case	Worse Case	Big O
Growing Multiset	$C(n) = 2$ The best case is when we insert into an empty list. Only 2 comparisons are made, one for each linked list.	$C(n) = 7n + 5$ Example: inserting z into {a:3,b:3,z:1}. If the node is both alphabetically the largest and also has the least number of instances, even after insertion, the worst cast can occur. The first list has $3n$ comparisons, while the second has $2n - 1.$ After updating the node instance count, it is taken out of the list and reinserted again, traversing starting from the head. Again, if it still has the smallest value by this point, it travels all the way to the back of the list, having $3n$ comparisons.	$O(n)$
Shrinking Multiset	$C(n) = 2$ Example: removing 1 from multiset {1:1}. In the best case, where we remove the	$C(n) = 7n + 5$ Example: removing z from multiset {a:3,b:3;z:2}. Similarly to growing the	$O(n)$

	head node, only 1 comparison is made in both lists.	multiset, if the item being removed is both the largest alphabetically and has the lowest instance count after removing one instance.The first list will have $2 + 2n$ comparisons, while the second will have $3 + 2n + 3n$.	
Intersection	$C(n) = n$ Uses the exact same algorithm as the ordered linked list. Refer to the table above.	$C(n) = 2n$ Uses the exact same algorithm as the ordered linked list. Refer to the table above.	$O(n)$
Print	$C(n) = n$ The print method in the dual linked list is much simpler compared to the ordered linked list. Because the second list is ordered by instance count, all we have to do is iterate through it at a cost of $2n$ assignments.	$C(n) = n$ The print method in the dual linked list is much simpler compared to the ordered linked list. Because the second list is ordered by instance count, all we have to do is iterate through it at a cost of $2n$ assignments.	$O(n)$

Empirical Analysis

Data generation and experimental setup

For the purpose of testing and evaluating our data, we implemented our own data generator and testing class. DataGenAndTesting.java is located in the generation folder. If you want to use it for your own purposes, drag it out and place it into the root folder of the source.

We attempted to automate as much of the testing process as possible to keep it relatively hands free and reduce as many human variables as possible.

To precisely understand the effectiveness of our data structures, we will be conducting tests on three different sizes of multisets. Starting at 2500 nodes, then 5000 and finally 10,000. Using large input sizes help exaggerate the differences between our results which can help us bring precise conclusions and recommendations. Each test is run five times for each multiset size, which is 15 tests runs in total.

Our fixed set is generated by using random string generators online. Our fixed set consists of 50 random and unique strings, each 10 characters long. A random number generator is then used to determine which string from the fixed set will be inserted in the current moment.

The great feature of our testing class is that all multisets will store the same values when growing, and will remove the same values when shrinking the list. This eliminates variances in data as a potential confounding variable that can affect results.

To accurately measure performance we use System.nanoTime() to measure execution time in nanoseconds. Time is calculated by recording the time before and after method execution. Execution time is then derived, which is then added to a total, in smaller more frequent executed methods such as the case with growing and shrinking the multiset.

DATA - Our test data is available in the spreadsheet accompanying the report.

DISCUSSION

Between the Comparison of Array and BST, Array and BST has much more close result when comparing their overall result, for example, for the comparison of the average runtime between these few scenarios, given, array and BST both has small margin of differences, such as for growing scenario, they contain around 0.002 seconds differences, Shrinking scenario has around 0.006 differences, Intersection scenario has around 0.002 seconds differences too, and lastly Print scenario, which has around 0.02 seconds.

However, even though they had a small margin of differences between Array and BST, based on the result it seems that BST has much faster overall performance then Array. The reason is because thanks to the concept of how BST works, which any inserted nodes then lower than root nodes will be at the left hand side of the tree, while Right hand side of the tree will contain nodes that has higher value than root nodes, which makes it easier for searching for the certain node or adding nodes to the certain location nodes based on the value itself.

Next comparison would be the comparison between Ordered Linked List and Dual Linked List. Based on the result given by the 10000 size of multiset, it seems that in the growing scenario, Ordered Linked List has better performance than Dial Linked List, which has won around 0.05 seconds faster runtime than Ordered Linked List. However, when compared with the other different scenarios, Dual Linked List has

much better performance than Ordered Linked List, which in case of Intersection Scenario and Print Scenario, Dual Linked List has over 50% more faster average result than Ordered Linked List.

From the comparison we got from the average result between Ordered Linked List and Dual Linked List, we can see that Ordered Linked List is faster in terms of inserting additional value, but loses overall performance when trying to search for certain value or when printing all of the data. The reason why Ordered Linked List is better during inserting is thanks to it only contains data field and next link field, which helps get over and insert the nodes into the Linked List. But since it only has one next link field, it became slower to search for the certain node than Dual Linked List, the reason is because in Ordered Linked List, its traversal only can be done using only one next node link, while Dual Linked List can be done its traversal using either next node link, or previous node link, which help when the need to search for the last few value. So it concludes that Ordered Linked List is better when inserting value, but Dual Linked List has much better access to value, which makes it more efficient.

Summary

According to the data test results we have gathered, Binary Search Tree Multiset used the least time for every scenario. Hence, I recommend to use this Binary Search Tree Multiset.