Algorithms and Analysis Assignment 1

Multiset Implementation

Sriram Senthilnathan(S3812810) & Yogeshwar Girish Chaudhari(S3828116)

AUGUST,2020

Abstract:

An analysis of implementation of Multiset(bag) abstract data structure using Arrays, Binary Search Tree (BST), Ordered Linked List and Dual Ordered Linked List in Java is carried out and evaluated their performance in terms of four common operations such as growing the multiset, shrinking the multiset, intersecting two multisets and printing the multisets. The algorithm for these four scenarios of all 4 implementations are evaluated by running the regression tests against them using different datasets of various sizes and exact timings are recorded for each test scenarios and finally average them individually for each operation of different implementations.

From the obtained results of our devised algorithms, we state that the Multiset using Binary search tree is efficient in case of adding, removing and intersecting operations. When it comes to print operation, the performance is not at its best on comparing with other implementations, but it performs reasonably well in overall.

Introduction:

A set is unordered collection of elements with no repetition in it. But if a set allows repetition of elements then it is known as Multiset or Bag. For this assignment we have implemented the Multiset abstract data type in Java to perform various operations (like create, add, remove, print, search, union, intersect, difference) using four data structures, such as

- Array
- Ordered singly linked list
- Binary search tree
- Dual ordered linked list

We have evaluated the performance and complexities of our implementation, both theoretically and empirically for four basic operations such as Growing Multiset, Shrinking Multiset, Intersection of two Multisets, Print Multiset. This report contains the results of our analysis and evaluation of the implemented Multiset.

Theoretical Analysis:

We have evaluated the complexity of our implementation of Multiset using four data structures in terms of the scenarios (given below) and the results are tabulated.

- Growing Multiset - **Scenario 1**
- Shrinking Multiset - **Scenario 2**
- Intersection - **Scenario 3**
- Print - **Scenario 4**

n – Size of the multiset *n1, n2 – Sizes of the multisets for intersect operation*

Multiset Implementation using Array			
Scenarios	**Best Case**	**Worse Case**	**Big O Notation Asymptotic Complexity**
Scenario 1 n = 5	**Time Estimate:** $t(n) = C_{op}$ **Example & Explanation:** ▪ ***Multiset = {apple, banana, car, Elf, Duck}*** ▪ ***Operation = add apple*** If an element to be added is already present at the first position of the set, then there will be only one comparison and incrementation operation for add operation. Hence it is O (1).	**Time Estimate:** $t(n) = (n * C_{op1}) + (n * C_{op2})$ **Example & Explanation:** ▪ ***Multiset = {Elf, Duck, car, banana, bike}*** ▪ **Operation = *add apple*** Irrespective of distribution of elements in the multiset, if the element to be added to the multiset is not present in the multiset, then the element will be inserted at the end of the array after 'n' comparisons. Also, the array size needs to be increased by one and copy the 'n' elements already present in the array before adding the new one at the end. So, its complexity is 2n.	O(n)
Scenario 2 n = 5	**Time Estimate:** $t(n) = C_{op}$ **Example & Explanation:** ▪ ***Multiset = {apple, apple, car, Elf, Duck}*** ▪ ***Operation = removeOne apple*** If the element to be removed is present at the first position of the array and has more than 1 instances of it, then there is only one comparison and decrement operation. Hence it is O (1).	**Time Estimate:** $t(n) = n * C_{op}$ **Example & Explanation:** ▪ ***Multiset = {apple, apple, car, Elf, Duck}*** ▪ ***Operation = removeOne Duck*** If the element to be removed is present at the anywhere in the array other than first or last position of the array and has only one instance, then iterate through array till the element is found and then swap the element to be removed with last element in the array. Then finally delete the last element (which is the element to be deleted). Then post removal of that element, new array of size reduced by '1' will be created and copy operation runs for n-1 elements. So, its complexity is O(n).	O(n)
Scenario 3 n1 = 5, n2 = 5	**Time Estimate:** $t(n) = n^2 * C_{op}$ **Example & Explanation:** ▪ ***Multiset1 = {apple, banana, car, Elf, Duck}*** ▪ ***Multiset2 = {Zebra, titanium, rock, kite, jump}*** ▪ ***Operation = intersect nultiset1 multiset2*** Both the sets have no common elements. In this case there will be 'n^2' comparison operations. Complexity is $O(n^2)$.	**Time Estimate:** $t(n) = n^2 * C_{op1} + 2n * C_{op2}$ **Example & Explanation:** ▪ ***Multiset1 = {apple, banana, car, Elf, Duck}*** ▪ ***Multiset2 = {apple, banana, car, Elf, Duck}*** ▪ ***Operation = intersect nultiset1 multiset2*** Both the sets have same set of elements and in same order. Even in this case there will be 'n^2' Comparisons and 'n' no. of add operations to create new multiset containing common set of elements. Complexity is $n^2 + 2n \Rightarrow O(n^2)$	$O(n^2)$

Scenario 4 n = 5	**Time Estimate:** $t(n) = n^2 * C_{op1} + n* C_{op2}$ **Example & Explanation:** ▪ ***Multiset = {apple, apple, car, car, Duck}*** ▪ ***Operation = print multiset*** Even when the elements in multiset are already in decreasing order of instance count, the sort function is called before print function, hence the complexity is always n^2+n (traversal for printing)	**Time Estimate:** $t(n) = n^2 * C_{op1} + n* C_{op2}$ **Example & Explanation:** ▪ ***Multiset = {zebra, titanium, car, apple, apple}*** ▪ ***Operation = print multiset*** Elements in multiset are in ascending order of instance count, the sort function is called before print function, hence the complexity is always n^2 (sorting) + n (Traversal).	$O(n^2)$

Multiset Implementation using Ordered singly linked list			
Scenarios	**Best Case**	**Worse Case**	**Big O Notation Asymptotic Complexity**
Scenario 1 n = 5	**Time Estimate:** $t(n) = C_{op}$ **Example & Explanation:** ▪ ***Multiset = {banana, car, Duck, Elf, Zebra}*** ▪ ***Operation = add apple*** If an element to be added is less than the list head element, then there is only one operation to add that new element as new list head and make its next node pointer to point to the old list head. Hence complexity is O (1).	**Time Estimate:** $t(n) = n*C_{op}$ **Example & Explanation:** ▪ ***Multiset = {apple, banana, car, Elf, Duck}*** ▪ ***Operation = add Zebra*** If the element to be added is not present and greater than all the elements that exists already in the list, then the new element will be added at the end of the list after 'n' comparisons. Hence complexity is O(n)	O(n)
Scenario 2 n = 5	**Time Estimate:** $t(n) = C_{op}$ **Example & Explanation:** ▪ ***Multiset = {apple, banana, car, Duck, Elf}*** ▪ ***Operation = removeOne apple*** If the element to be removed is the head of the list, there is only one operation to remove it by making the second element in the list as new list head. Complexity is O (1).	**Time Estimate:** $t(n) = n*C_{op}$ **Example & Explanation:** ▪ ***Multiset = {apple, banana, car, Duck, Elf}*** ▪ ***Operation = removeOne Elf*** If the element to be removed is in the last position of the list, then 'n' number of comparison operations are needed to arrive at the element by sequential search and then remove it. Hence the complexity is 'n'.	O(n)
Scenario 3 n1 = 5, n2 = 5	**Time Estimate:** $t(n) = n*C_{op}$ **Example & Explanation:** ▪ ***Multiset1 = {apple, banana, car, Duck, Elf}*** ▪ ***Multiset2 = {Frog, Kite, Lamp, Sun, Truck, Zebra}*** ▪ ***Operation = intersect nultiset1 multiset2*** Both Multisets have no elements in common and are of same sizes (n1 = n2), then there will be n1(or n2) number of comparison operation. So, complexity is n1(or n2).	**Time Estimate:** $t(n) = n*C_{op} + n*C_{op1}$ **Example & Explanation:** ▪ ***Multiset1 = {apple, banana, car, Elf, Duck}*** ▪ ***Multiset2 = {apple, banana, car, Elf, Duck}*** ▪ ***Operation = intersect nultiset1 multiset2*** If both Multisets have same set of elements in same order, then number of comparison operations needed is 'n' as both lists are being traversed simultaneously and the n number of add operations to create new list. Complexity is 2n.	O(n) Where n = Min (n1, n2)
Scenario 4 n = 5	**Time Estimate:** $t(n) = n*C_{op} + n^2*C_{op1} + n*C_{op2}$ **Example & Explanation:** ▪ ***Multiset = {apple, apple, car, car, Duck}*** ▪ ***Operation = print multiset*** Elements in the multiset are already in the decreasing order of instance count. Even in this case no. of operations needed to print the list is n (Traversal) + n^2(Sort using Array Multiset) + n (print). Hence complexity is $O(n^2)$	**Time Estimate:** $t(n) = n*C_{op} + n^2*C_{op1} + n*C_{op2}$ **Example & Explanation:** ▪ ***Multiset = {apple, banana, banana, car, car}*** ▪ ***Operation = print multiset*** Even the worst case will be same as best case. The list will be first converted to Array Multiset. Thereafter it follows the same path of Array print operation. Hence the complexity is always n^2+ 2n.	$O(n^2)$

Multiset Implementation using Binary search tree (BST) *figures in brackets indicates the instance count. * default is 1. *(root) = Root of the tree			
Scenarios	**Best Case**	**Worse Case**	**Big O Notation Asymptotic Complexity**
Scenario 1	**Time Estimate: t(n)** = c_{op} **Example & Explanation:** ▪ ***Multiset = {e (root), c, a, b, i, f, j}*** ▪ ***Operation = add z*** In case of full binary tree, the search operation for addition would perform more like binary search. Hence the best-case complexity would be O (log n) as addition of node can be done in constant time O (1) Hence the overall complexity is O (log n).	**Time Estimate: t(n) =** n*c_{op} **Example & Explanation:** ▪ ***Multiset = {a (1), b (1), c (1), d (1), e (1), f (1)}*** ▪ ***Operation = add g*** If the BST is a left-skewed or right-skewed (right skewed in this case), then BST traversal will be require n comparisons as it will behave like list only. There are only 2 operations involved - Create new node and attach it as right child of f (g > f) Set 'f' as parent of 'g' which has constant time complexity. Hence complexity is O (n).	O(n)
Scenario 2	**Time Estimate: t(n) = c_{op}** **Example & Explanation:** ▪ ***Multiset = {e (4)(root), c, a, b, i, f, j}*** ▪ ***Operation =*** remove One e Element to be removed is root node of tree having a more than 1 instance. O (1) time for finding the element to be removed. Hence complexity is O (1)	**Time Estimate: t(n) = n*c_{op}** **Example & Explanation:** ▪ ***Multiset = {a (1), b (1), c (1), d (1), e (1), f (1)}*** ▪ ***Operation =*** remove One f Since we are not having a height balanced tree, we will get a right skewed tree for given multiset. For removing an instance of f, we need to traverse to the leaf node of the BST by visiting all nodes in between which requires O(n) time. Decrement instance count of 'f'by 1 requires O (1) Removing f (leaf node) if instance count becomes 0 requires O (1) time. Hence complexity is O (n)	O(n)
Scenario 3 n1 = 6, n2 = 6	**Time Estimate:** t(n) = n*c_{op} **Example & Explanation:** ▪ ***Multiset1 = {f, a, g, b, c, h}*** ▪ ***Multiset2 = {z, g, k, l, m, n}*** ▪ ***Operation = intersect multiset1 multiset2*** If both multisets have uncommon elements, then running time would be O(n) where n = min (n1, n2). This is because loop will run only till 1 of the given trees is traversed completely. Additionally, there would be no add operations involved due to no common elements. Hence the time complexity is O(n).	**Time Estimate:** t(n) = n*c_{op} + n*c_{op2} **Example & Explanation:** ▪ ***Multiset1 = {f, a, g, b, c, h}*** ▪ ***Multiset2 = {f, a, g, b, c, h}*** ▪ ***Operation = intersect multiset1 multiset2*** If both Multisets have same set of elements in same order, then number of comparison operations needed is 'n' as both trees are being traversed simultaneously and the n number of add operations to create new list. However, since we are traversing the trees in in order fashion, the resulting multiset would be right-skewed. Hence the time complexity for each insertion operation would be O(n). Hence overall complexity would be – O(n) as every insertion would require complete traversal of resultant BSTMultiset.	O(n) Where n = Min(n1, n2)
Scenario 4	**Time Estimate:** t(n) = n*c_{op1} + n2*c_{op2} + n*c_{op3} **Example & Explanation:** ▪ ***Multiset = {apple, apple, car, car, Duck}*** ▪ ***Operation = print multiset*** A tree multiset is converted into an array multiset which would require O(n) for tree traversal and $O(n^2)$ for forming ArrayMultiset and $O(n^2)$ for sorting elements based on instance count. And Additional O(n) for printing	**Time Estimate:** t(n) = n*c_{op1} + n2*c_{op2} + n*c_{op3} **Example & Explanation:** ▪ ***Multiset = {apple, apple, car, car, Duck}*** ▪ ***Operation = print multiset*** Worst case is similar to best case. A tree multiset is converted into an array multiset which would require O(n) for tree traversal and $O(n^2)$ for forming ArrayMultiset and $O(n^2)$ for sorting elements based on instance count. And Additional O(n) for printing	$O(n^2)$

Multiset Implementation using Dual Ordered linked list			
Scenarios	**Best Case**	**Worse Case**	**Big O Notation Asymptotic Complexity**
Scenario 1 n = 5	**Time Estimate:** t(n) = C_{op1} + C_{op2} **Example & Explanation:** ▪ ***Multiset = {banana, car, Duck, Elf, Zebra}*** ▪ ***Operation = add apple*** If an element to be added is less than the list1 head element which means there is no previous instances of it, then there is only one operation for each list(list1 and list2) to add that new element as new list head and make its next node pointer to point to the old list head. Hence complexity is 1 + 1.	**Time Estimate:** t(n) = n *C_{op1} + C_{op2} **Example & Explanation:** ▪ ***Multiset = {apple, banana, car, Elf, Duck}*** ▪ ***Operation = add Duck*** If the element to be added is already present in last position of list1 and list2, then 'n' comparison operation is needed to add the element in list1 and after adding the element in list2, its instance count is increased hence swap operation is required to maintain the list 2 in descending order of instance count. Complexity is n + (1 + no of swaps required in list2).	O(n)

Scenario 2 n = 5	**Time Estimate:** $t(n) = C_{op1} + C_{op2}$ **Example & Explanation:** ▪ ***Multiset = {apple, banana, car, Duck, Elf}*** ▪ ***Operation = removeOne apple*** If the element to be removed is the smallest and has largest number of instance in multiset, then it will be head in both list1 and list2, there is only one operation to remove it by making the second element in both the lists as new list head. Complexity is 1 + 1.	**Time Estimate:** $t(n) = n * C_{op1} + C_{op2}$ **Example & Explanation:** ▪ ***Multiset = {apple, car, car, Elf, Elf}*** ▪ ***Operation = removeOne Elf*** If the element to be removed is the largest in the multiset and has greater least number of instances, then it will be in last position of both list1 and list2, so 'n' number of comparison operations are needed to arrive at the element by sequential search and then remove it from both the lists, additionally swap is required in list2 if after remove operation the order of element needs to be changed based on instance count. Hence the complexity is n + (n + no. of swaps required in list2).	O(n)
Scenario 3 n1 = 5, n2 = 5	**Time Estimate:** $t(n) = n * C_{op}$ **Example & Explanation:** ▪ ***Multiset1 = {apple, banana, car, Duck, Elf}*** ▪ ***Multiset2 = {Frog, Kite, Lamp, Sun, Truck, Zebra }*** ▪ ***Operation = intersect nultiset1 multiset2*** Both lists have no elements in common and are of same sizes (n1 = n2), then there will be n1(or n2) number of comparison operation. Complexity is n1(or n2).	**Time Estimate:** $t(n) = 3n * C_{op}$ **Example & Explanation:** ▪ ***Multiset1 = {apple, banana, car, Elf, Duck}*** ▪ ***Multiset2 = {apple, banana, car, Elf, Duck}*** ▪ ***Operation = intersect nultiset1 multiset2*** If both lists have same set of elements in same order with same instance count, then number of comparison operations needed is 'n' as both lists are being traversed simultaneously and the (n + n) number of add operations to create new lists. Complexity is 3n.	O(n1) or O (n2) When (n1 = n2)
Scenario 4 n = 5	**Time Estimate:** $t(n) = n * C_{op}$ **Example & Explanation:** ▪ ***Multiset = {apple, apple, car, car, Duck}*** ▪ ***Operation = print multiset*** Elements in list2 are already maintained in ascending order of instance count. Hence using recursion to print that list in reverse order will always require 'n' number of operations. Complexity is n.	**Time Estimate:** $t(n) = n * C_{op}$ **Example & Explanation:** ▪ ***Multiset = {apple, banana, banana, car, car}*** ▪ ***Operation = print multiset*** Elements in list2 are already maintained in ascending order of instance count. Hence using recursion to print that list in reverse order will always require 'n' number of operations. Complexity is n.	O(n)

Experimental Setup:

Data Generator:

We have created a DataGenerator.java class to do the experimentation. The file setup is described as follows.

- Analysis is performed on 5 different dataset sizes.

Dataset name	Size(No. of elements)
Small	100
Double Small	2500
Medium	5000
Double Medium	7500
Large	10000

- All datasets are initialised with multiple words containing random 5 letters.
- There are 16 functions for performing best, practical, and worst cases on the each of the ADT implementation (ArrayMultiset, BSTMultiset, OrderListMultiset, DualOrderedListMultiset) for each of the scenario (insertion, removal, intersection, printing).
  For more information on the parameter and configuration settings of the data generator java file, please refer Appendix.

Data Generator Algorithm:

1.Generate datasets(small, double small, medium, double medium, large)
2.perform testing for(array, bst, orderedll, dualll)
3.for(add, remove, intersect, print)
4.for(small, double small, medium, double medium, large)
5.for(best, worst, average)
6.Record the time
7.Repeat process from 1 – 6 to get results of 50 runs.

s

Initial Population of the multisets:

Multisets are populated with the datasets at the beginning on each test cycle. These datasets are restored for intersection operation as insertion and deletion operations changes the initial multisets.

.

Fixed Datasets:

Common insertion dataset for analysing the insertion cases { "aaaaa", "ddddd", "hhhhh", "kkkkk", "vvvvv", "rrrrr", "zzzzz"}. The elements spaced at sufficient distance and possibly cover scenarios of insertion operation in every 1/7th portion of multiset. Common cases for analysis include but not limited to.

Sr	Operation	Scenario	Description
1	Insertion	Best	Inserting duplicate element which is present at 0th index (Array), root node (BST), Head (Orderlies, DualOrderedList). Requires constant time.
		Worst	Inserting an element which is not present in a multiset, inserting ordered elements for tree resulting into skewed trees
		Average	Inserting 5 random duplicates, inserting elements in between (applicable for lists only)
2	Removal	Best	Removing head node for lists, duplicate starting element from all data structures
		Average	Removing random nodes in between.
		Worst	Removing last element for lists, array, and skewed trees
3	Intersection	Best	No common elements lead to no additional add operations
		Worst	Common elements, overhead of performing add operations for resulting multiset
4	Print	Best	Every ADT other than Dual Ordered List requires sorting elements based on instance count.
		Worst	Same as best case.

- These tests are then run on each dataset (Small, Double Small, Medium, Double Medium, Large) for 4 individual operations (Add, Remove, Intersect, Print) of each data structures mentioned above. We have included 50 test runs total and respective times (in seconds) are recorded.
- The datasets are refreshed in between 2 consecutive operations, so that every time an operations runs, it runs with new set of elements.
- The time recorders are placed just before and after the primitive function call making sure that we are not capturing time spent in preparing datasets or cases. All times are captured in nanoseconds which are then converted to seconds with up to 10 precision places.
- Once the timings are recorded, then the average timings are calculated for four operations(add, remove, intersect, print) individually for all 4 data structures.

Results of Experimental Analysis:

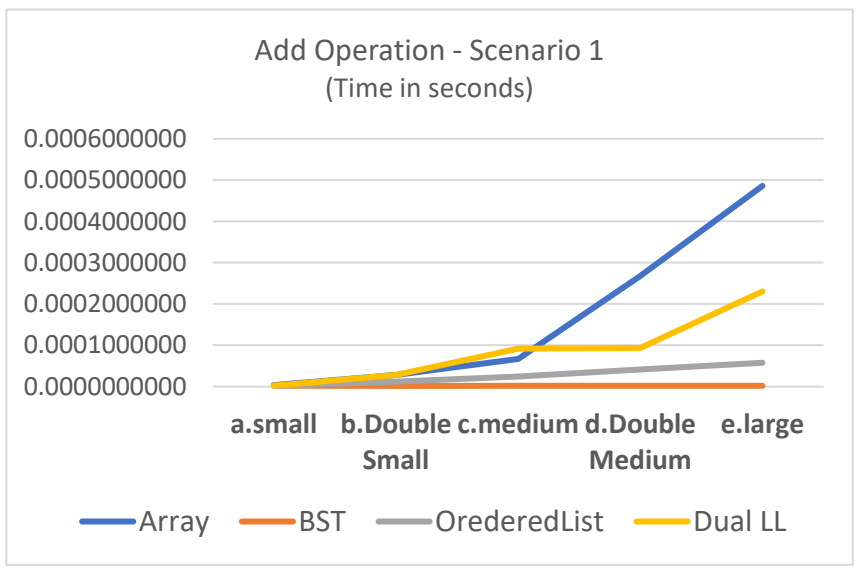

Fig. 1

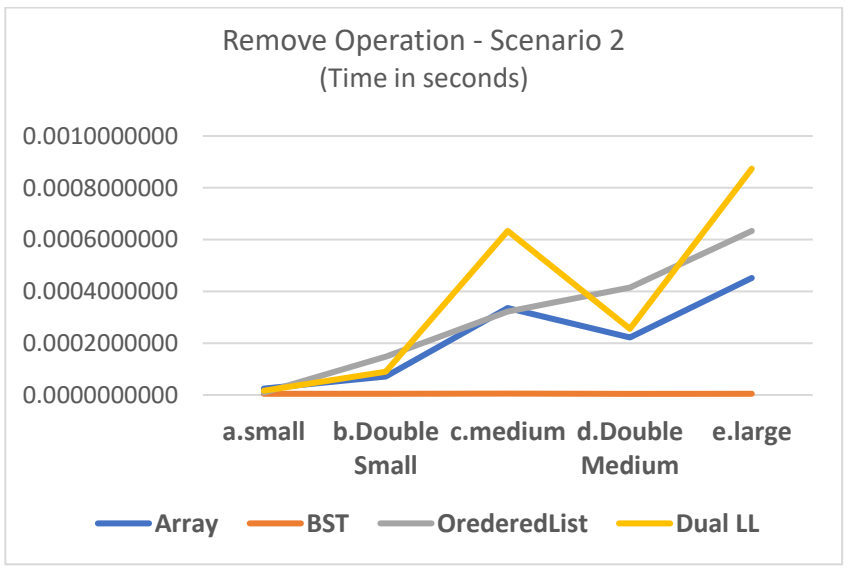

Fig. 2

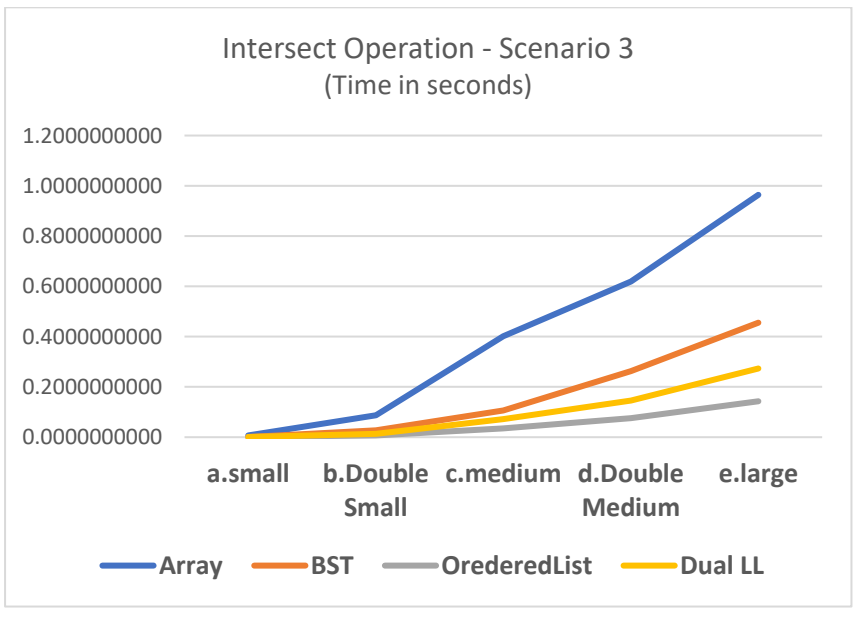

Fig. 3

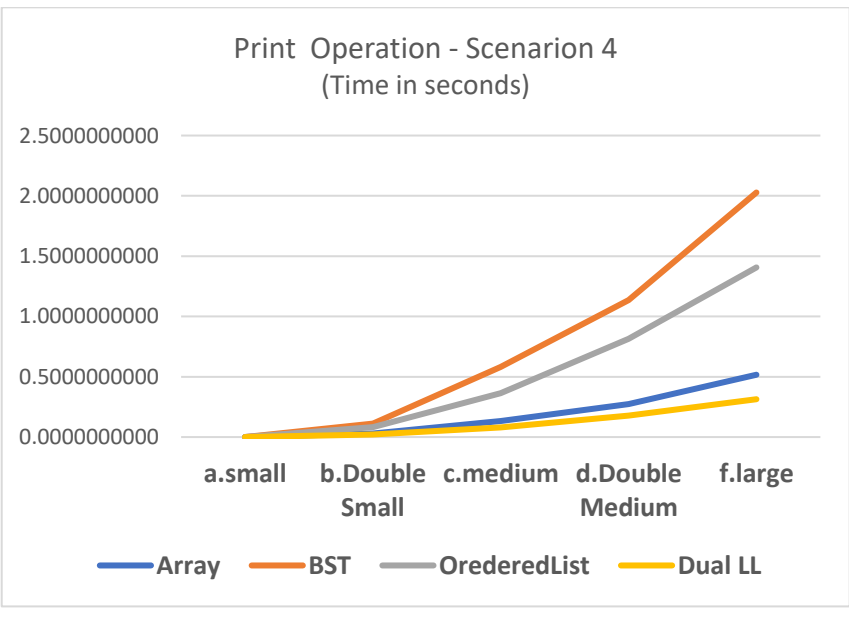

Fig. 4

For detailed and complete timing information of each cases of all four scenarios of four data structures of 50 total runs, please refer the appendix.

Scenario 1 - Add Operation : (Best in Running time – Binary Search Tree)

- BST achieved the fastest running time in all dataset sizes(average), because even in the worst-case scenario (when the tree is left or right skewed tree) it performed maximum of 'n' operations like Ordered List i.e. O(n). For most cases (when tree is sufficiently balanced due to random distribution of data) the binary search brings the complexity to O (log n) to insert a new element or even a duplicate. On average BST performed best in this scenario
- Ordered Linked list achieved the second fastest running time in all dataset sizes(average), as the best case of this scenario is when the element that needs to be added is lesser than the list head it will be inserted as new list head. The list also performs better when elements are small enough to be inserted very close to head node. For worst case, However, entire list is to be traversed to insert an element at end which requires n comparisons i.e. O(n) complexity ignoring the constant time required for creating new node and attaching to list. Hence the time complexity will increase linearly with increase in data size.
- In Dual-ordered linked list, though it performs the same operations as Ordered linked list, it took more running time because of additional overhead of maintaining another list based on descending order of the instance count every time an insertion is made.
- Array insertion is an expensive operation in terms of running time and space usage, when compared to all other 3 data structures. Due to its unordered behaviour, for every insertion, entire array needs to be traversed until the element is found if not already present. New array creation is required to accommodate this new element which increases the overhead of element shifting.

Scenario 2 - Remove Operation: (Best in Running time – Binary Search Tree)

- Again, BST performed best in terms of removing an element on an average. Even its running time is very low when compared to other 3 data structures. Removing the duplicate element has the complexity of O (log n) in best case & average case and O(n) in worst case if tree is skewed. From *Fig. 2,* it is evident the running time of BST grows logarithmic when dataset size increases gradually.
- Surprisingly, from the graph it is evident that array has next better running time, because in the worst case if an element needs to be completely removed then instead of deleting it immediately and then shifting backward all the elements present next to it, our algorithm swaps the element to be deleted with last element in array and then removes it. Hence the avoidance of shifting operations saved a huge amount of running time. But it uses more space for shrinking arrays (by dynamically creating new one of reduced size).
- Ordered linked list consumed more running time than array and for the large dataset its running time went high, due to the random selection of elements and if the element removed has no more instances left then next pointer updation requires constant time.
- Dual-Ordered linked list has the worst running time. Removal operation is like ordered linked list. However, we also need to find the same element in ordered by instance count list and adjusts its position to ensure all elements remain in sorted order. This is an additional overhead that needs to perform with every removal operation which adds complexity of O (n).

Scenario 3 - Intersect Operation: (Best in Running time – Ordered Linked List)

- When it comes to intersect operation, both Ordered linked list and Dual-ordered linked list have better running times, because these data structures traverses through the two multisets at the same time for comparison. Also, if the two multisets are of different sizes then the operation stops when it reaches the end of multiset of minimum size.
- Dual-ordered linked list shows higher running time than ordered linked list and that is because of the overhead of maintaining the second list as well. But it still performs better than array and BST.
- Array has the worst running time in this operation because unordered behaviour. To intersect two arrays, for each element in first array, second array needs to be traversed sequentially till end in worst case. It adds the complexity of $O(n^2)$ and the curve in *Fig. 3* proves the same.

Scenario 4 - Print Operation: (Best in Running time – Dual Linked List)

- For the print operation the dual linked list achieved the best running time because, the list2 in this data structure maintains the order of elements based on decreasing instance count. Hence to print, this list2 is iterated sequentially and printing all elements one by one till the end. So, the complexity is always O(n). In Fig. 4, Dual-linked list curve for increasing data set sizes is O(n) curve.
- Unlike dual ordered list, array elements are sorted using selection method based on their instance count. This adds the additional complexity of $O(n^2)$. After sort, it prints the elements which in turn adds O(n). So, it is n^2+n. That caused the next higher running time.
- For both BST and Ordered linked list, to print the elements, first they are being added to the array multiset using Array Add operation (O(n)). Printing operation is then performed on this new ArrayMultiset which has complexity of $O(n^2)$. So, the complexity is higher in both BST and Ordered linked list due to type conversion involved. Dual ordered list clearly has an advantage here because no sorting is involved.

Conclusion:

From the experimental results, we hereby conclude that Binary Search Tree(BST) Data structure can be used for the efficient implementation of the Multiset(bag) because in terms of running time ,the BST outperforms every other 3 implementation for the operations such as growing multiset, shrinking multiset and intersecting two multisets. BST's performance takes a dig only in printing operation which provides a room for future enhancements, as our algorithm is designed a way such that part of its operation is dependent upon array multiset, but still the performance is not the worst.

Recommendations based on analysis:

- For applications with large datasets, where frequent insertions, deletions and searching is required, BST would be perfect fit. The searching complexity at max would be O(n) which is good when compared to others.
- In applications where data does not change much after initial initialisation or is not exposed to many updates, but requires constant access to highest occurring data, Dual ordered list would be a better fit. The complexity will always be O(n)
- Dual linked list can also be recommended where all data elements are unique and are inserted in ascending order. This way both print and insert would be performed in O(n)
- Applications involving the large sets and where insertions and deletions are frequent, and often required to compare the two lists or get common elements, Ordered List can give best results.
- Where memory is not a constraint, instead of using min-dynamic array, we can use double-dynamic array which would significantly reduce the number of shifting operations required for insertion as well as removal due to availability of space.

Future enhancements based on analysis:

1. Try to achieve the height balanced BST. The BST will behave like list when a sorted order of element is provided while creating BST.
2. While removing an element from the Array, we can replace that element with the last element. This will save the shifting operations required for new insertions as well.
3. A tail pointer can be maintained which can then be effectively used for each operation potential providing constant time complexity for some of the worst cases.
    a. For add operation, if the new element is larger than tail element then inserts at the end.
    b. For remove operation, if the element is equal to tail element, then remove the last element
    c. For intersect operation, if the tail element of first list is smaller than second list, we can softly say that there will be no common elements and hence an empty Multiset can be returned.
4. Instead of going for selection sort algorithm in Array Multiset, more sophisticated sort algorithms can be used to further improve the performance of Array Multiset, which in turn improves the efficiency of BST and Ordered Linked List as well, as their print functions are dependent on array sorting.

References:

Report :

Understanding Time Complexity with Simple Examples - GeeksforGeeks

Understanding Time Complexity with Simple Examples - GeeksforGeeks (2017). Available at: https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/ *(Accessed: 29 August 2020).*

Anon

(2020) Faculty.salina.k-state.edu. Available at: http://faculty.salina.k-state.edu/tmertz/Java/328trees/binarysearchtreeperformance.pdf (Accessed: 29 August 2020).

Code:

[1] Binary Search Tree - GeeksforGeeks

Binary Search Tree - GeeksforGeeks (2020). Available at: https://www.geeksforgeeks.org/binary-search-tree-data-structure/#basic (Accessed: 26 August 2020).

[2] Binary Search Tree | Set 1 (Search and Insertion) - GeeksforGeeks

Binary Search Tree | Set 1 (Search and Insertion) - GeeksforGeeks (2014). Available at: https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/ (Accessed: 26 August 2020).

[3] Binary Search Tree | Set 2 (Delete) - GeeksforGeeks

Binary Search Tree | Set 2 (Delete) - GeeksforGeeks (2014). Available at: https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/ (Accessed: 26 August 2020).

[4] Deletion in Binary Search Tree - javatpoint

Deletion in Binary Search Tree - javatpoint (2020). Available at: https://www.javatpoint.com/deletion-in-binary-search-tree (Accessed: 29 August 2020).

[5] Binary Search Tree (BST) insert, delete, successor, predecessor, traversal, unique trees - Algorithms and Problem Solving
Binary Search Tree (BST) insert, delete, successor, predecessor, traversal, unique trees - Algorithms and Problem Solving (2015). Available at: http://www.zrzahid.com/binary-search-tree-bst-insert-delete-successor-predecessor-traversal/ (Accessed: 29 August 2020).

[6] Inorder Successor in Binary Search Tree - GeeksforGeeks
Inorder Successor in Binary Search Tree - GeeksforGeeks (2011). Available at: https://www.geeksforgeeks.org/inorder-successor-in-binary-search-tree/ (Accessed: 29 August 2020).

[7] Linked List | Set 1 (Introduction) - GeeksforGeeks
Linked List | Set 1 (Introduction) - GeeksforGeeks (2013). Available at: https://www.geeksforgeeks.org/linked-list-set-1-introduction/ (Accessed: 26 August 2020).

[8] Linked List | Set 2 (Inserting a node) - GeeksforGeeks
Linked List | Set 2 (Inserting a node) - GeeksforGeeks (2013). Available at: https://www.geeksforgeeks.org/linked-list-set-2-inserting-a-node/ (Accessed: 26 August 2020).

[9] Linked List | Set 3 (Deleting a node) - GeeksforGeeks
Linked List | Set 3 (Deleting a node) - GeeksforGeeks (2014). Available at: https://www.geeksforgeeks.org/linked-list-set-3-deleting-node/ (Accessed: 26 August 2020).

Appendix is on next page

Appendix :

Parameter configurations:

Sr	Particulars	Details	Why?
1	Word Size	Strictly 5 letters	Keeping test time in check. It also helps in to keep the comparison time same
2	Small Dataset Size	100 unique elements	Small applications
3	Double Small Size	2500 unique elements	
4	Medium Dataset Size	5000 unique elements	
5	Double Medium Dataset Size	7500 unique elements	
6	Large Dataset Size	10000 unique elements	Performance analysis on how the operation complexities scales
7	Sorted Lists (Ascending & Descending)	5 Lists for each mentioned dataset sorted in ascending order.	Insertion of ordered elements would produce left or right skewed
8	Alphabet Set	Unique set of English alphabets producing 1,18,81,376 unique combinations	Analysing the impact of each scenario on unique as well as duplicate elements
9	Time unit	Recorded in seconds	More comprehensible
10	Total Runs	50	
11	Addition (Scenario count) (*Some Scenario has 5 variations)	Array: 4 OrderedLinked List: 4 BST: 5 * 5 DualOrderedList: 3	
12	Removal (Scenario count) (*Some Scenario has 5 variations)	Array: 4 OrderedLinkedList: 3 BST: 5 DualOrderedList: 3	
13	Intersect (Scenario count) (*Some Scenario has 5 variations)	Array: 3 OrderedList : 3 BST: 3 DualOrderedList:	
14	Print (Scenario count) (*Some Scenario has 5 variations)	Array: 2 OrderedLinkedList:2 BST: 2 Dual Ordered List:2	
15	Time unit	Nanoseconds	Precise time recording later converted to seconds

Complete Experimental Results:

Please find the onedrive link to excel workbook to review the complete results of Experimentation.

https://rmiteduau-my.sharepoint.com/:x:/r/personal/s3812810_student_rmit_edu_au/Documents/ExperimentalResults_and_Analysis.xlsx?d=w7197dda5bb2e46c7935f1fbf7195dd0a&csf=1&web=1&e=lu1hiK