Algorithms and analysis

Assignment 1 Written Report

S3549962 Yangming An
S3615907 Huirong Huang

Section 1 Analysis Set Up

For this report, we have created different scenarios to evaluate which includes :

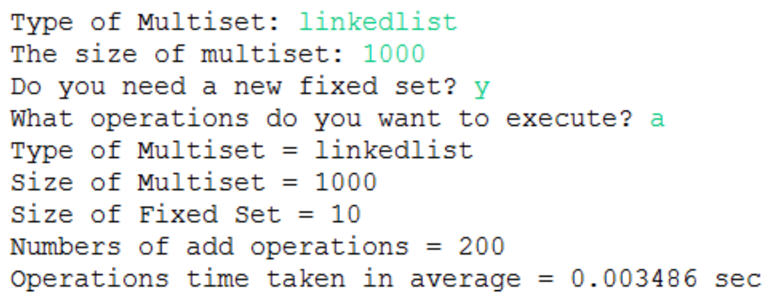
Sizes of the multisets:

For the sizes of the original multisets, we will use the size of 1000, 5000, 10000, 30000 elements.
Reasons to choose above four sizes are that we try to cover the range from a few original data to a lot original data and without making the test too complicated. So we chose 1000 as the minimum size because if the amount of the original is too small, there will not be a huge difference when doing experiment. 5000 as middle sizes, as for 10000 and 30000, we use these two to see the performance of these data structures when dealing with large amount of data.

Scenarios:

We also have different scenarios to apply for above four multisets. So firstly, we apply increment to these sets, which means we only do add element operations. Then we do decrement that is applying only remove element operations. Thirdly we make both operations a roughly equal number.
Lastly, we consider to have two ratio of the number of searches versus number of addition and removal that are 2 : 1 and 4 : 1.
Reason to evaluate on these scenarios is to test the performance of different data structures under a vary number of input and different ways of operations.
These scenarios are generated by using a constant ratio of operations and size of the sets for example 1 : 5, which means for a set of size 1000, we do 200 add or remove operation to obtain the scenario.
In order to obtain the timing for different data structure and scenario, we use a program for experimental part.
We used data generator to generate 4 fixed sets, and the ratio of multiset versus fixed sets is 100 : 1.
We write codes to run a program for experiment.
It is a simple program to measure the time of operations (already included in the zip file)
Eg. we want to measure the time of **doubly linked list** of size **1000** with only **add** operations.
1.We type “linkedlist” as we want the type of the multiset to be a doubly linked list.
2.We type “1000” as the size of the multiset we want.
3.The program will call the DataGenerator class to generate a random linked list of size 1000.
4.We type “y” because we need a new fixed set.

5.The program will call the DataGenerator class to generate a random array of string of size 10.
6.The random array will be stored in a txt file which we may use next time.
7.We type “a” to indicate we want all add operations.
8.The program will measure the time of the operations using System.nanoTime() for 5 times.
9.The program will show the details of the experiment as well as the average time.



```
Type of Multiset: linkedlist
The size of multiset: 1000
Do you need a new fixed set? y
What operations do you want to execute? a
Type of Multiset = linkedlist
Size of Multiset = 1000
Size of Fixed Set = 10
Numbers of add operations = 200
Operations time taken in average = 0.003486 sec
```

Section 2 Structure Analysis

The scenario that we tested for each algorithms are listed below, the results will be shown in the form of charts.
Case 1: For a set that has 1000 elements with only add operation
Case 2: For a set that has 1000 elements with only remove operation
Case 3: For a set that has 1000 elements, with equal number of additions and removals
Case 4: For a set that has 1000 elements, search versus number of addition and removal has a ratio of 1 : 2
Case 5: For a set that has 1000 elements, search versus number of addition and removal has a ratio of 1 : 4
Case 6: For a set that has 5000 elements with only add operation
Case 7: For a set that has 5000 elements with only remove operation
Case 8: For a set that has 5000 elements, with equal number of additions and removals
Case 9: For a set that has 5000 elements, search versus number of addition and removal has a ratio of 1 : 2
Case 10: For a set that has 5000 elements, search versus number of addition and removal has a ratio of 1 : 4
Case 11: For a set that has 10000 elements with only add operation
Case 12: For a set that has 10000 elements with only remove operation
Case 13: For a set that has 10000 elements, with equal number of additions and removals
Case 14: For a set that has 10000 elements, search versus number of addition and removal has a ratio of 1 : 2
Case 15: For a set that has 10000 elements, search versus number of addition and removal has a ratio of 1 : 4
Case 16: For a set that has 30000 elements with only add operation
Case 17: For a set that has 30000 elements with only remove operation

Case 18: For a set that has 30000 elements, with equal number of additions and removals
Case 19: For a set that has 30000 elements, search versus number of addition and removal has a ratio of 1 : 2
Case 20: For a set that has 30000 elements, search versus number of addition and removal has a ratio of 1 : 4

Following are the charts that represent our results, each chart represents the time used for the 5 algorithms to do addition or removal and search for above 4 different sizes of miltisets.

Chart 1 : addition/removal of set size 1000

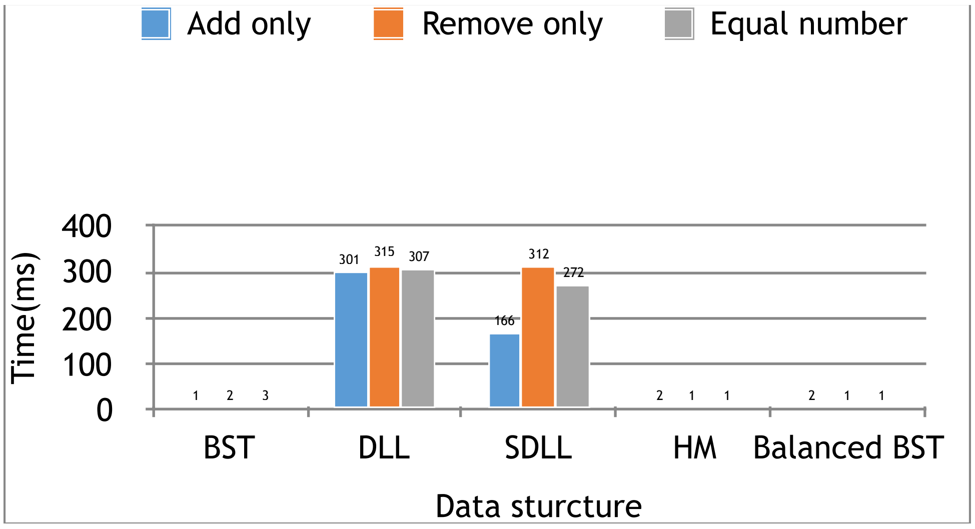


Chart 2 : Search versus addition and removal of set size 1000

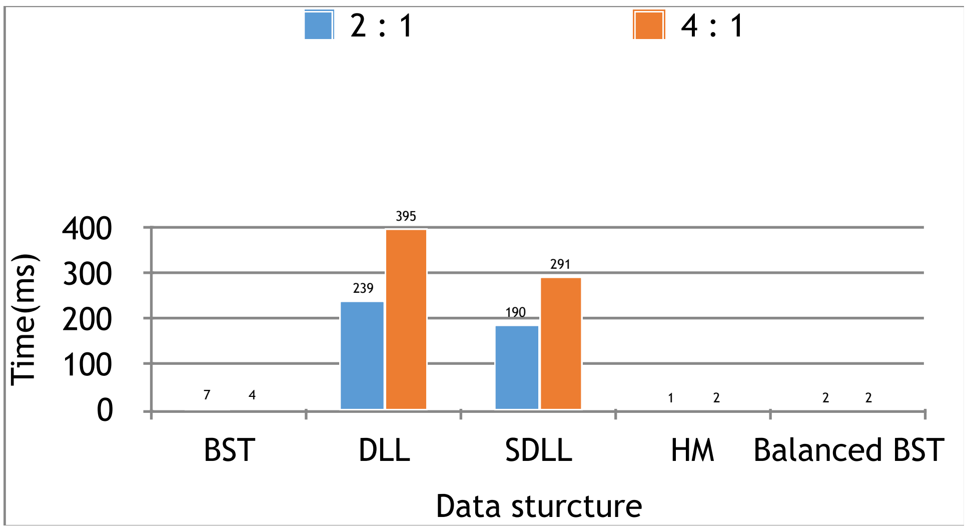


Chart 3: Addition and removal of set size 5000

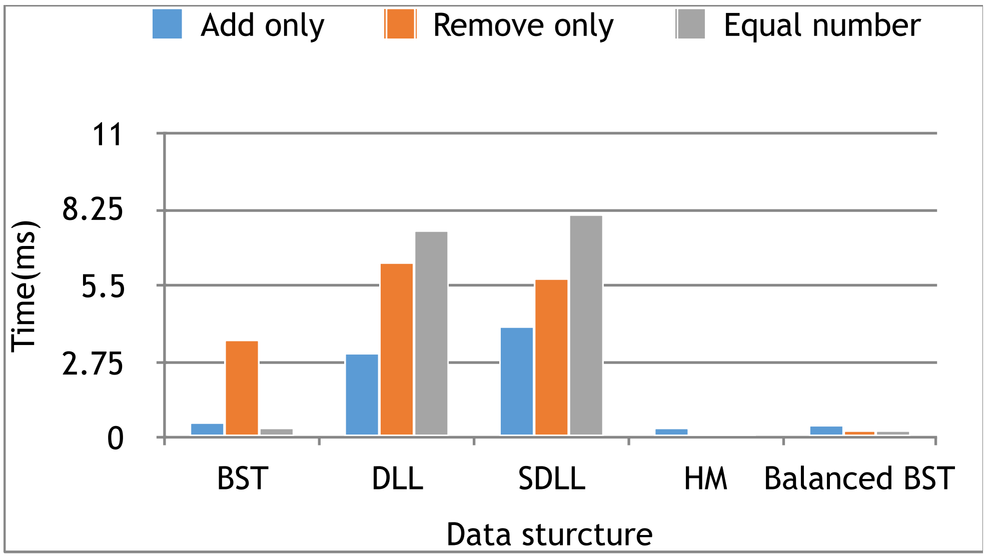


Chart 4 : Search versus addition and removal of set size 5000

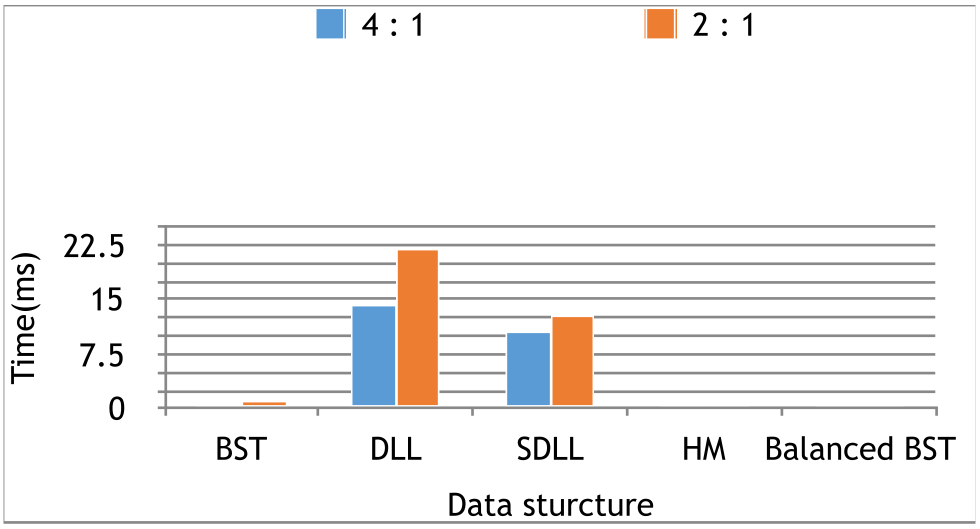


Chart 5 : Addition and removal of set size 10000

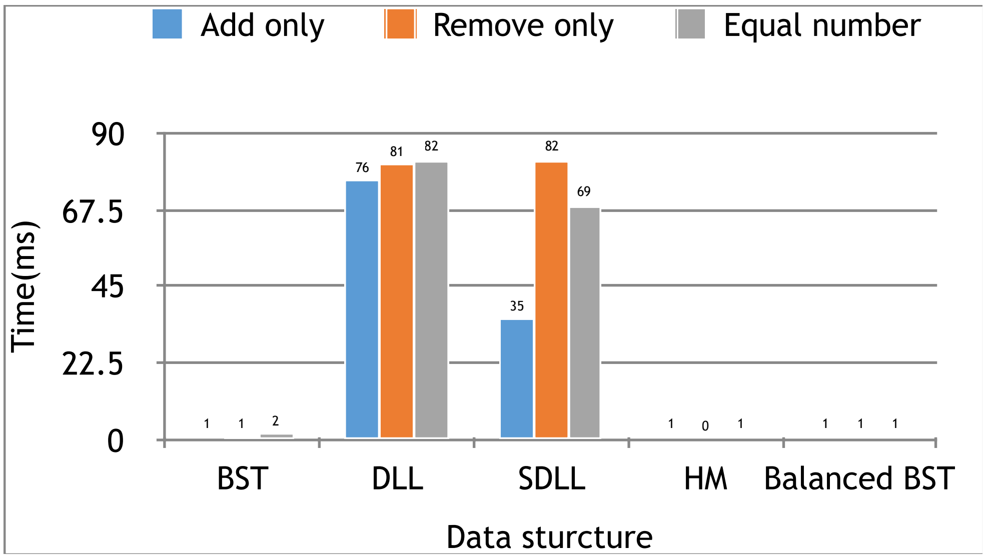


Chart 6: Search versus addition and removal of set size 10000

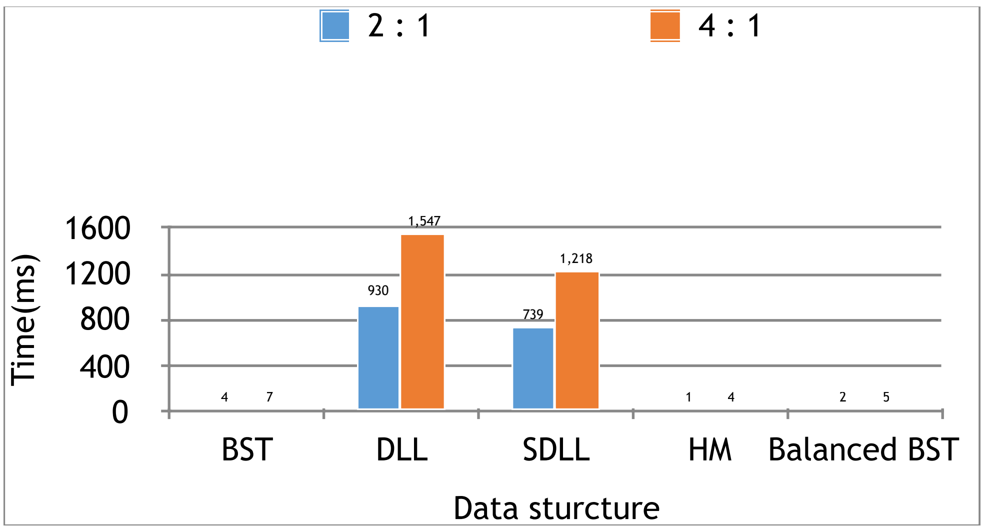


Chart 7 : Addition and removal of set size 30000

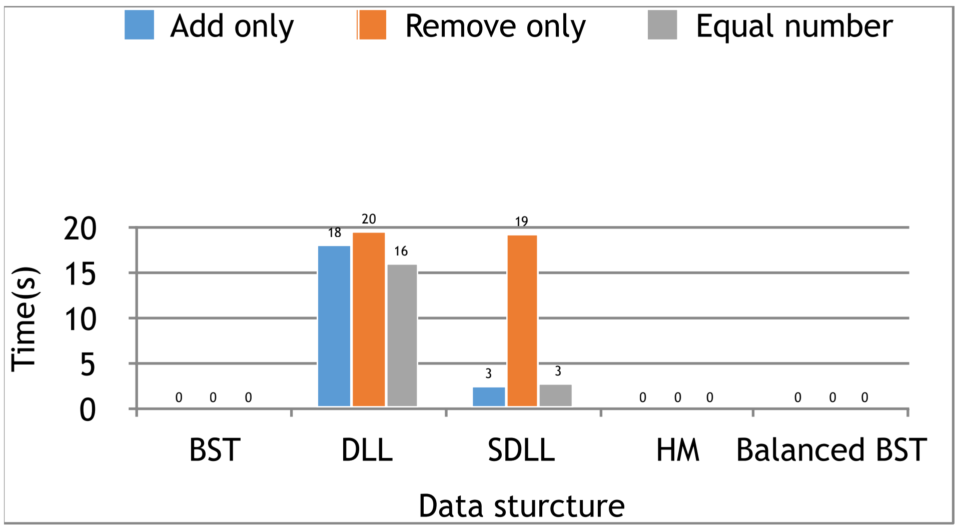
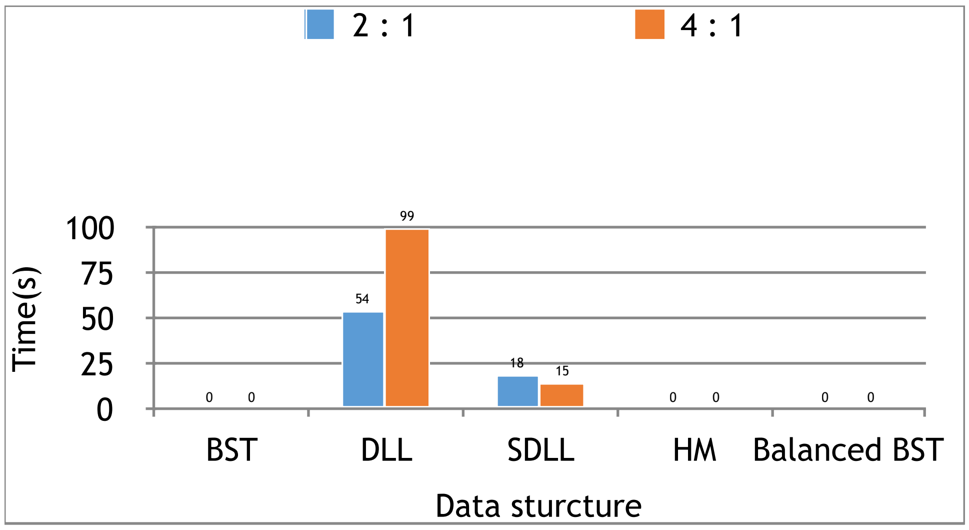


Chart 8 : Search versus addition and removal of set size 30000



We know that hash table algorithm is to use a hash function to assign each "key" a bucket, however, sometimes if the hash function is not well designed, there will be collision. And the theoretical time complexity has an average case of, in Big-O notation O(1) and a worst case of O(n). From the results we could see that this algorithm has a good performance dealing with all these scenarios, reason that we get these results is that hash table is an effective algorithm, essentially when we search for data, we only need to find the hash function of the data and since for each input we can only get one out put unless the hash function is not well designed, the time needed will be short.
For balanced binary search tree, we could see that its performance is better than binary search tree both add or remove and search operations, but not as good as hash table. Theoretically we know that the time complexity of a balanced BST is with both average and worst case of O(log(n)). By using this algorithm, searching time complexity is reduced compared to normal BST since the structure of the tree is stable. However, the insertion and removal time is increased because when doing this, algorithm still need to keep the tree in a balanced manner. A normal binary search tree has search time complexity of O(log(n))for average case and O(n) for worst case. So it is just to compare with the node and the value that we looking for and decide to go whether left or right subtree of the binary tree which means for each comparison it will skip half the data, but since it is not sorted, the worst case complexity is due to the height of the tree.
Obviously, sorted and normal doubly linked list do not have a good performance over all scenarios, We know that doubly linked list has search time complexity of O(n) and insertion/removal time complexity of O(1). For insertion and removal, the complexity is constant because if we know at which index we want to insert or delete the element, it will go to that index directly. As for searching a specific element, it will go over the list and to see if there is a match. So that it is quite inefficient when the input is too large.
Sorted linked list is just an improved version of doubly linked list, results shows that has a better performance than the unsorted one but still slow when compare to other data structures, but since it is sorted, searching an element in this list will not go over whole list and time used is therefore reduced.

Summary

From the results we know that hash map is good at dealing with complex situation and large amount of data, but the hash function need to be designed perfectly so that it could then be an algorithm of best choice. Then for both binary trees, it will have a good performance when you need to have a lot of search operations over both large and small number of data input.
As for doubly linked list, it is obvious that their performance has a huge difference between others. So they are not suitable for large input, and search operation is also their limitation. I personally recommend to use these two algorithms when input is not too large and mainly adding and

removing elements from the input, but though it is slow in performance, it has an advantage of easy implement means it is sometimes more convenient to use than other algorithms.