Data Structure and Programming

HW5

Name: 吳睿哲
Student ID: r06921095

Implementation

For the first part is the implementation of three ADTs: **Double linked list**, **Dynamic array**, and the **binary search tree**. The detail is given as follow.

➢ Double Linked List

Double linked list is a data abstract structure that composed of many nodes. The information of data, next node, and previous node is also store in the node. Thus, we can obtain the previous node and next node easily. For the linked list, the information of the head node is also stored, the we can traverse the whole list by using iterator. The iterator is overloaded in the class of Dlist. We can overload some arithmetic operator such as ++, -- etc. Thus, we can use the overloaded operator to perform some operation on the double linked list. The most important part in the double linked list, I think is to remember to link the next node as well as previous node after performing some operation on the adt. Last but not least, to place the head node in the right position.

➢ Dynamic Array

For the dynamic array, since the memory is continuous allocated. Thus, it is needless to remember the previous node and the next node, which become much easier in implementation. Some implementation part is similar to the double linked list, class of iterator is also utilized in the class of array, so that the array can be traversed easily, and performing several operations. The most important part is to increase/decrease the size of the array when an element is added/deleted. Last but not least, when the size is equal to the capacity of the array, do not forget to new a bigger dynamic array, and copy the older array into the newer one.

➢ Binary Search Tree

For the binary search tree, due to the fact that this part is not finished yet, the experimental result of bst is from the given reference program.

Experiment

In this part, several operations are tested, including add, deletion, query and sort. And the result is also tested in different size, including 100, 1000, 10000, 100000, and 100000. To normalize the scale of the size, log is utilized, where the size will become 2, 3, 4, 5, 6 in the figures.

➢ Add Data

In this part, different size of the element is added to the adt. As we can see in figure 1, due to the fact that the time complexity of inserting new element in **bst** is O(lgn), while the other 2 adt are both O(1). Thus, the time utilized in **bst** is long when the input size is big.

For the memory utilization as shown in figure 2, it is observable that the 3 adts are almost the same.

✓ Random add

time (s) size / adt	100	1000	10000	100000	1000000
Dlist	0	0	0.01	0.03	0.1
Array	0	0	0.01	0.04	0.16
Bst	0	0	0.01	0.08	1.18

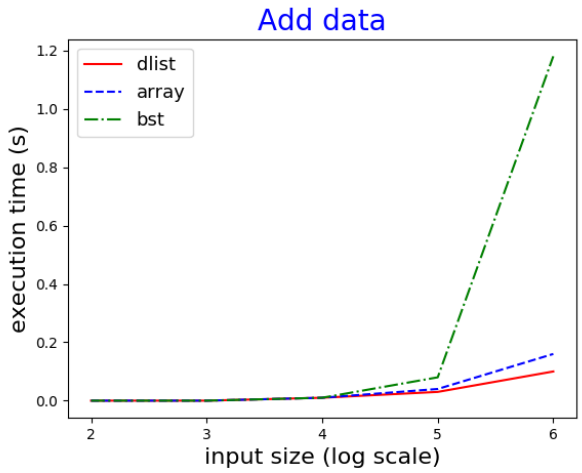

Fig. 1

✓ Memory utilization

mem (M) size / adt	100	1000	10000	100000	1000000
Dlist	0	0	0.194	5.918	60.85
Array	0	0	0.7422	6	48.2
Bst	0	0.03	0.97	6.098	48.2

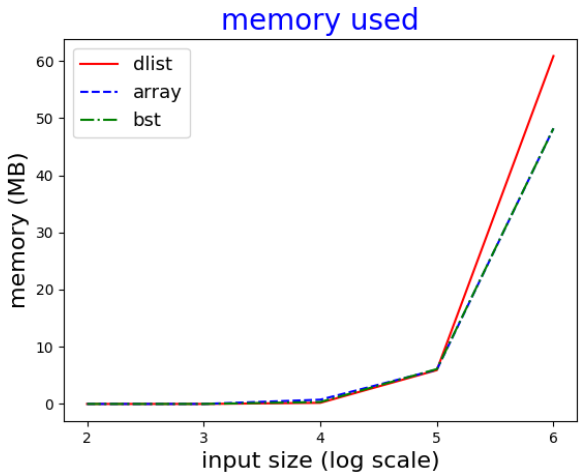

Fig. 2

➢ Delete Data

For data deletion, 3 cases are tested. Case 1, 1% of the elements at the front of the adt is performed. As shown in figure 1, due to the fact that the time complexity of deletion in Dlist and Array are both O(1). Thus, the time utilized are almost 0. And for the binary search tree, due to the fact that the elements to be deleted are located **all** at the front of the adt. Thus, the deletion spedd is also fast.

Same as case 1, in case 2, the complexity of Dlist and Array to delete the 1% of the elements at the back of adt are also O(1). Thus, the deletion operation is also fast. The elements to be deleted in this case are **all** at the end of the adt. Thus, the deletion speed for bst is also fast.

For case 3, random deletion of different size is performed. For Dlist, it takes O(n) time to locate each element to be deleted. For Array, it takes only O(1) time due to the fact of random access property. For bst, it takes a lot of time to delete the element and reconstruct the data structure of tree. Thus, it takes a lot of time.

✓ Delete first 1% data

time (s) size / adt	100	1000	10000	100000	1000000
Dlist	0	0	0	0	0.01
Array	0	0	0	0	0.01
Bst	0	0	0	0	0.01

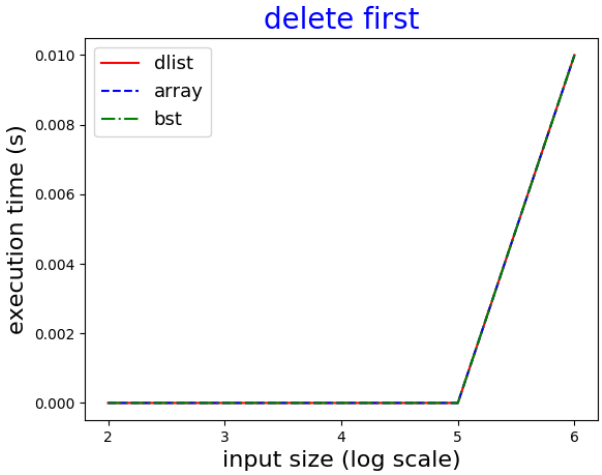

Fig. 3

✓ Delete back 1% data

time (s) size / adt	100	1000	10000	100000	1000000
Dlist	0	0	0	0	0
Array	0	0	0	0	0
Bst	0	0	0	0.01	0.01

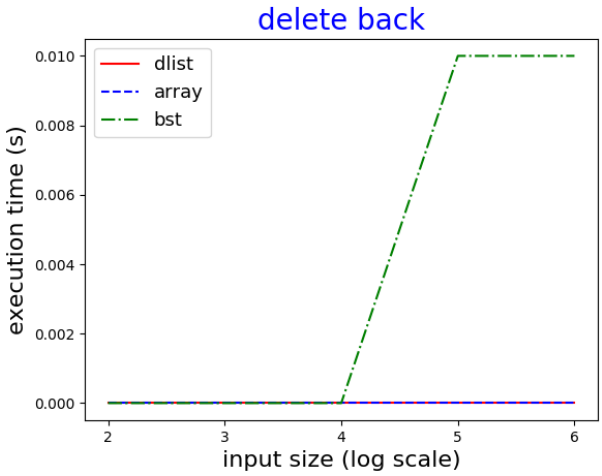

Fig. 4

✓ Random delete 1% data

time (s) / adt \ size	100	1000	10000	100000	1000000
Dlist	0	0	0.01	0.27	28.17
Array	0	0	0	0	0.01
Bst	0	0	0.03	1.68	363.9

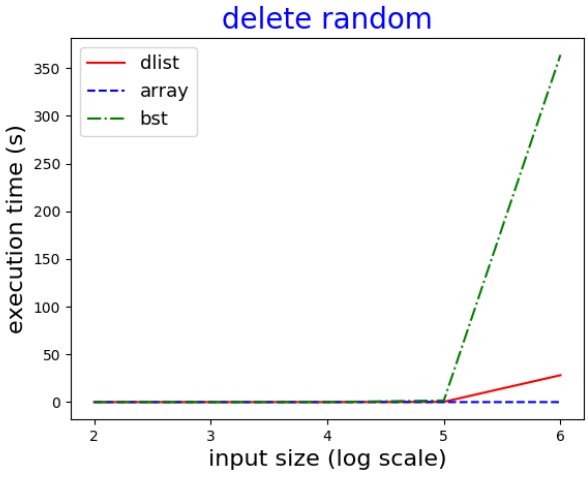

Fig. 5

➢ Query Data

In this part, each adt is trying to locate the last element. For Dlist and Array, the complexity to find the element are both O(n), and the complexity to find the element of bst is only (Olgn). Thus, the bst spends less time than the other two adt.

✓ Query the last element

time (s) / size / adt	100	1000	10000	100000	1000000
Dlist	0	0	0	0	0.01
Array	0	0	0	0	0.02
Bst	0	0	0	0	0

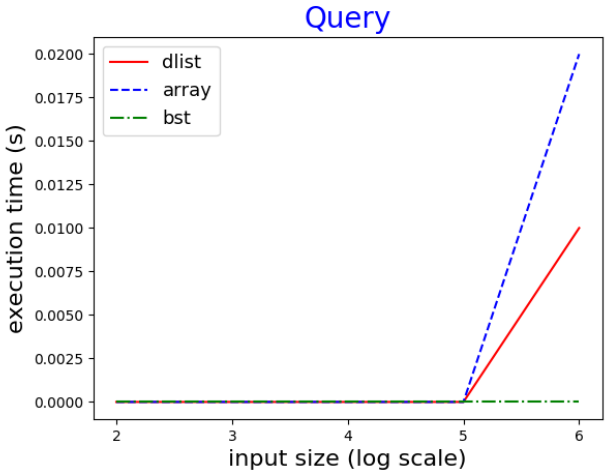

Fig. 6

➢ Sort Data

In this part, the performance of sorting on each adt are evaluated. First, due to the fact of Dlist can not perform random access, thus, it can only perform a slower algorithm such and insertion sort, where the complexity is O(n^2). And for Array, it can use a faster algorithm such as quick sort, where the time complexity is O(nlgn). For bst, the element is placed in a sorted order during insertion. Thus, it takes no time to sorting.

✓ Sorting all the element in each adt

time (s) / size / adt	100	1000	10000	100000	1000000
Dlist	0	0.03	0.97	97.92	1000 up
Array	0	0	0	0	0.37
Bst	0	0	0	0	0

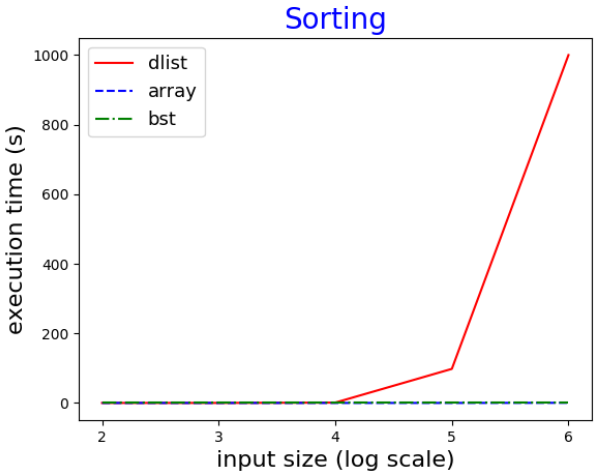

Fig. 7