ADTs for HW5
B06705023
Information Management
(I can't type Chinese in Linux, so I typed in English instead.)

Array

First of all, let's talk about array.
For insertion, simply put it in the last place. This requires O(1). Whenever the size reaches its capacity, a resize will be performed. Meaning, I will copy all of the elements to a larger array.
For deletion, replace the element with the last element. It is way better than shifting all the elements forward by 1 space. This is also O(1).
For query, I will check if the array is sorted, if sorted, O(logN) (Binary Search), otherwise O(N) (Linear Search).
The basis of array ADT is an array, obviously. I resize it everytime I reach its limit. Making it twice the space as it originally is capable of.
For sorting, *Algorithm's* sort is introduced.

D-List

For insertion, it depends on push_back or push_front. Creating a new element and connect it with the list requires constant time. Both of it requires O(1).
For deletion, going through the list is necessary and deleting the object is constant time. Hence, it is O(N).
For sorting, I use quick sort, because it out-performs Bubble sort. Also, the non-recursive version of quick sort has roughly the same loc(lines of codes) as Bubble sort. Moreover, since quick sort is constantly swapping data. So it is ideal for such data type that each node is only LINKED together. Besides, it requires O(1) space. In this case, time complexity will be O(NlogN).

BST

For insertion, it is just the same as how BST performs insertion. Nothing more needs to talk about. O(logN). However, I do maintain a parent child relationship during insertion. It will benefit us in the development in iterator stage. If there is a duplicate, I keep a variable in each node recording how many duplicates. That way, we won't have to worry about storing duplicate on LHS or RHS.
For deletion, Check for duplicates first. If exists duplicate, decrement it. Otherwise, perform BST deletion. During the deletion, I need to reconnect the link between nodes. That is, if the node to delete has 2 child, I need to take care of the 6 or 8 pointers. The additional 2 or 4 pointers are the one that we want to replace the tod(To Delete) with. O(logN).
For lookups, it requires O(logN) to traverse down the tree. As the height is at worst O(N). However I did some research and calculations $\frac{H_n}{\log(n)} \approx 4.14$. This number shows that some self-balancing tree, like AVL, RB tree, can have height 4 times lower than random BST. It may seem to be a huge difference in numbers though. After thorough contemplation and research, I still choose to use the simplest ADT → BST. Due to every insertion for AVL or RB tree, it will require more time than a standard BST. In the long run,(after hundreds of thousands of insertions), BST runs faster. Even if I want to re-balance it, I can finish it in O(n) time. On the other hand, the constant factor will eventually be diluted in O() notation. On the other hand However the average case is O(logN).
If you're the TA marking grading my program. I found out that

Now, proceeding into the performance section.

PERFORMANCE

As I mentioned above in each ADT, the complexity is for each operation can be described as following.

-Insert randomly (time needed)

Items\ADTs	Array	D-List	BST
10000	0	0	0.01
20000	0.02	0.01	0.02
40000	0.02	0.01	0.04
80000	0.03	0.01	0.07
160000	0.04	0.03	0.14
320000	0.07	0.03	0.69

-delete randomly (time needed)

Items\ADTs	Array	D-List	BST
10000	0	31.15	0.01
20000	0.	63	0.02
40000	0	Too long	0.04
80000	0.01	TOO LONG	0.07
160000	0.01	TOO LONG	0.14
320000	0.03	TOO LONG	0.69

SORTING

Both Array and Dlist uses quick-sort. Therefore, there is no obvious difference between both of them. In addition, BST is already a sorted DS, we don't need to discuss it here.

From the test cases I generate, it will still be array for me to do the work. Since it is much better and much straightforward to use it. Secondly, a BST is still achievable with an array. As for performance issue. Insertion and deletion is O(1) in array, and sorting the whole chunk is O(NlogN). Querying can be O(n) or O(logN). However, for day-to-day use, I will still prefer a mixture of D-list and tree.

Things to cover.
(operations briefly discuss it.)
1. How to implement
2. Why you want to implement in that method.
Performace comparison
  experiment design, Expect, result and discuss