DSnP hw5 array, dlist, bst compare

B07901184 陳映樵

PART 1 Implementation

Array:

For array, most functions are easy.

In iterator, if we meet some form like ++i, we just plus 1 to the pointer, if it is i++, we will copy a iterator and then plus one to the pointer, and return the copy at last. --i and i-- is similar.

In begin, we just find the first of the data, but for end, we have to go through all array to find the last data, so it takes longer time to process.

For erase, actually, pop_front, pop_back, erase(val), find are also related. First, we use find to find the value we want, then call erase(iter) to delete it. And pop_front and pop_back call begin and end respectively and then call erase(iter)

Dlist:

The implementation of dlist is a little tricky than array.

There is a priciple in dlist: if we want to go through the list, we keep finding the next(or prev) of the node until it reach end(or begin). If we want to add(or delete) node, we first get the pointer of the added(or deleted) data, and then change previous next to the successor, the next previous to the predecessor, then delete the pointer if needed.

In iterator, we do almost the same as array.

In begin, we return head anyway, since if no data, dummy = head, it having data, head = head, so there is no problem. But in end, we first check if it is empty, if true, return dummy(head), else return head->prev(because we have data now). In empty, we check if head->next = head(only dummy node will have this property).

To get size, we should run through all list for counting the size.

The concept of erase, pop_front, pop_back, and find are almost the same as array.

In sort, we implement bubble sort, the time complexity is O(n^2), so we cannot sort large data since it take up too many time for sorting.

BST:

In bst, I choose to implement parent instead of trace for some reasons, I think parent is easier than trace. Also, in iterator's constructor, we put not only the wanted node but begin and end node in it. By doing so, we can check whether it reach the edge of all tree. But it still have to maintain begin and end in the iterator when the data had been changed.

In bst iterator, we add a private member dummy to recognize if it reach end. If we want to go through the tree, we call function successor(different parameter for ++ and --). Take ++ for example. It is like recursion, if a node has right child, then we find the smallest node in right tree, else we go back through parent link until we find parent is smaller than it. It will terminate if it reach end. -- is almost the same.

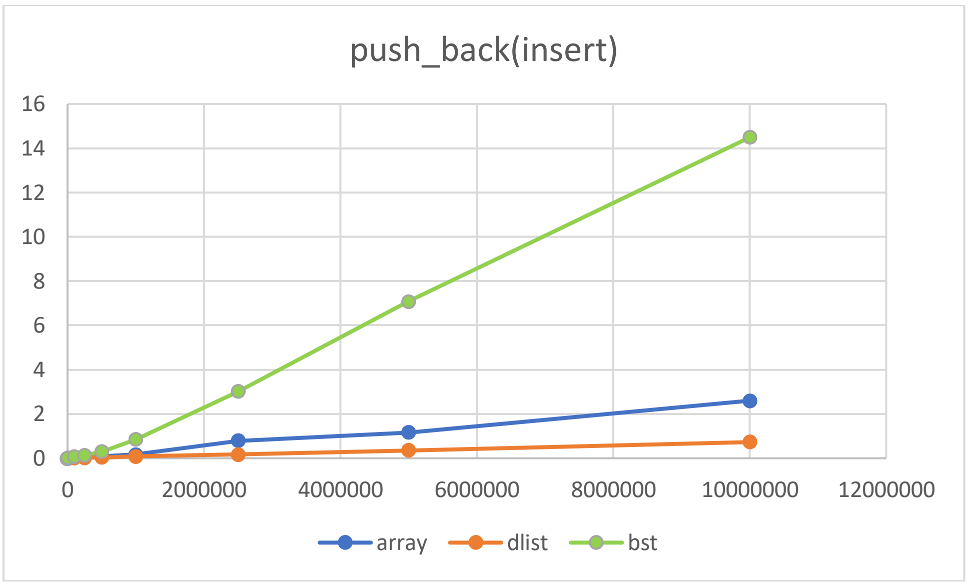
In bst insert, we iteratively compare the current node with given data, if it’s smaller, find its left child, else find its right child. This process will terminate if the target(left or right) is null, then we insert the node at that place.

In bst erase, if a node have no child, we just delete it and let the parent point to 0. If a node has one child, we cut it and link it parent to its child. If a node has two children, we, instead, swap data with its child until it meet condition 1 or 2 above. It is actually a loop.

In bst clear, we apply breadth first search to push each node into a queue and then delete it.

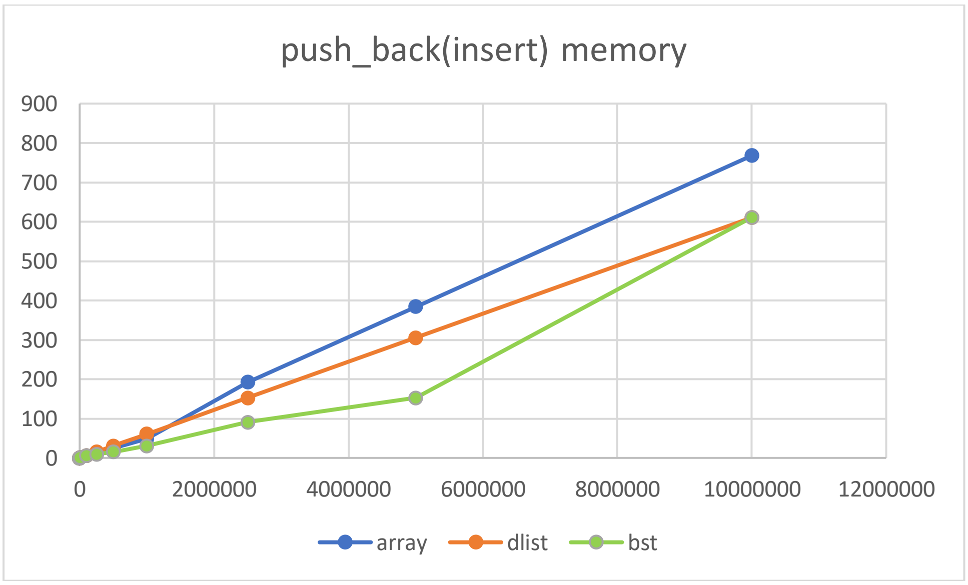
There is nothing to do is bst sort, since the data had already been sorted when inserting into the tree.

Part 2: TEST

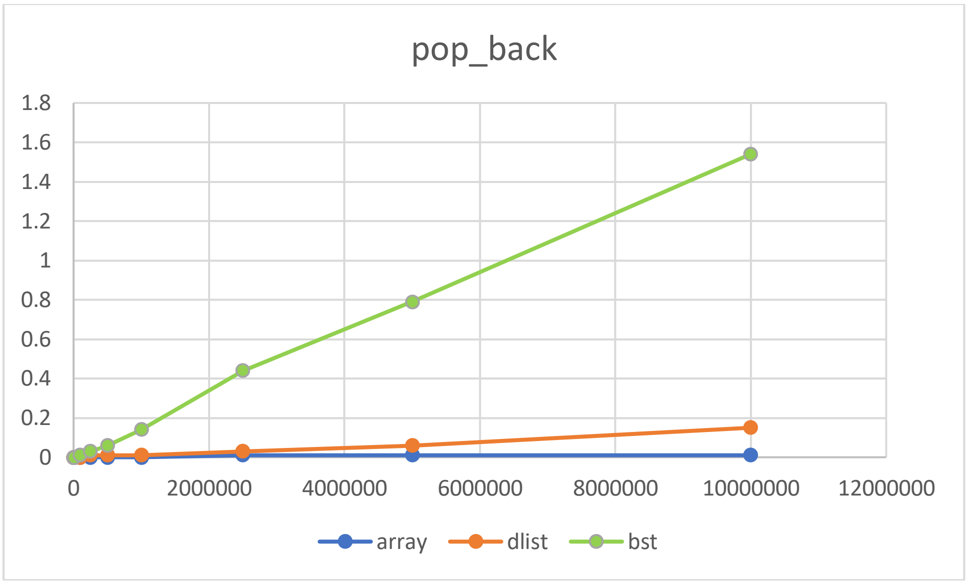


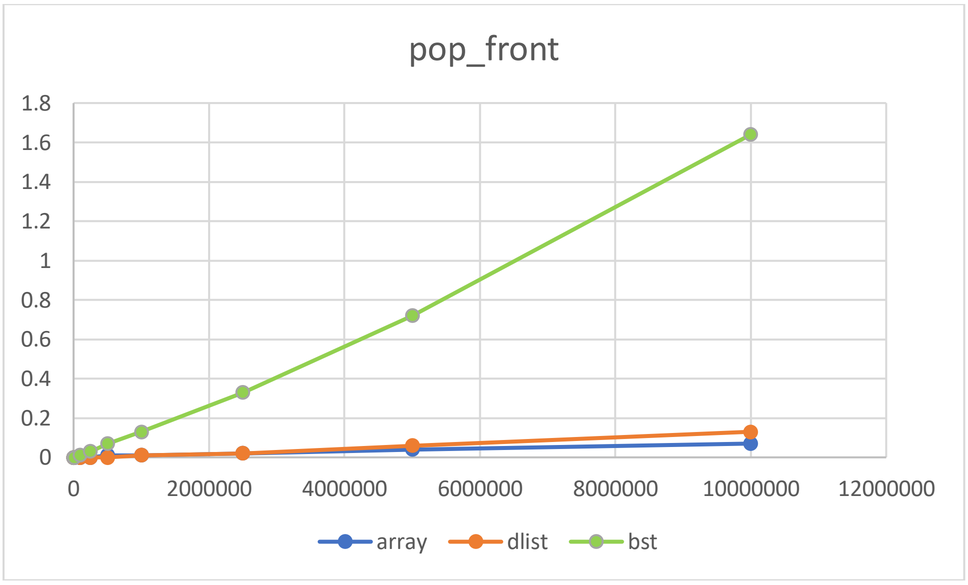
We can observe that bst insert takes much more time than array and dlist. It’s because it has to put the data into the right place, that is automatically sort the tree. So the advantage of bst is that you don’t have to sort the data again, it save a lot of time. If array and dlist is O(1), then bst is O(logN).

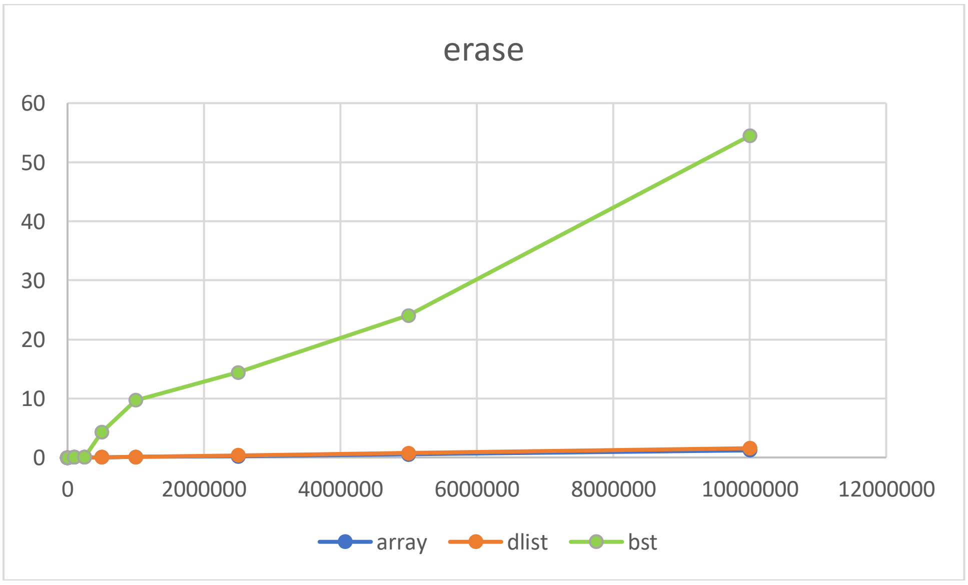
Now let’s compare array and dlist. Although there is almost constant time to insert a data, but since array have to resize if we run out of the capacity. Therefore, some reallocation is needed, and that cause extra time in array push_back.

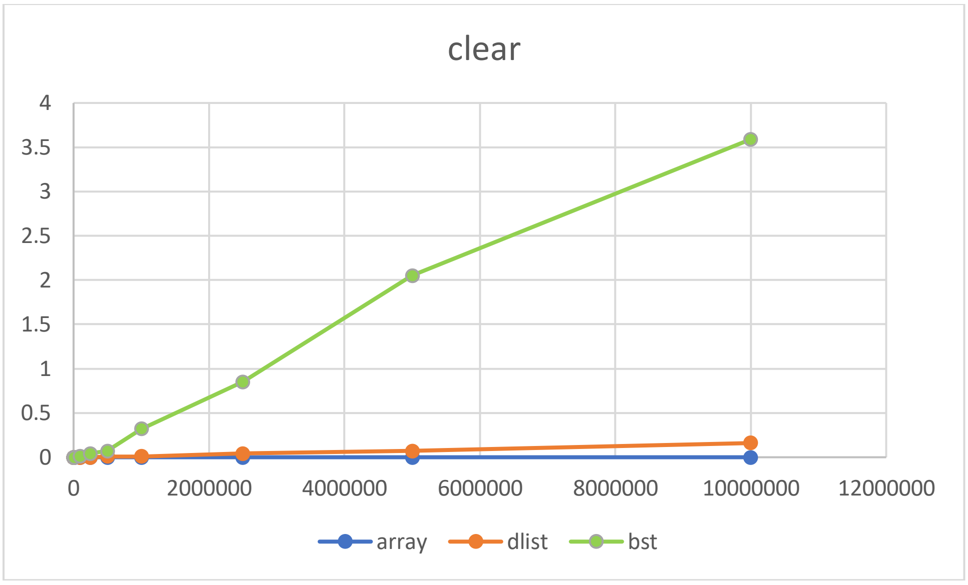


The result fit our intuition. Since when we put more data in a data structure, it consume more memory. Notice that array will need more memory for reallocation, but dlist and bst will just need certain memory for inserting a node.


pop_back
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
2000000
4000000
6000000
8000000
10000000
12000000
array
dlist
bst


pop_front
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
2000000
4000000
6000000
8000000
10000000
12000000
array
dlist
bst


erase
60
50
40
30
20
10
0
0
2000000
4000000
6000000
8000000
10000000
12000000
array
dlist
bst

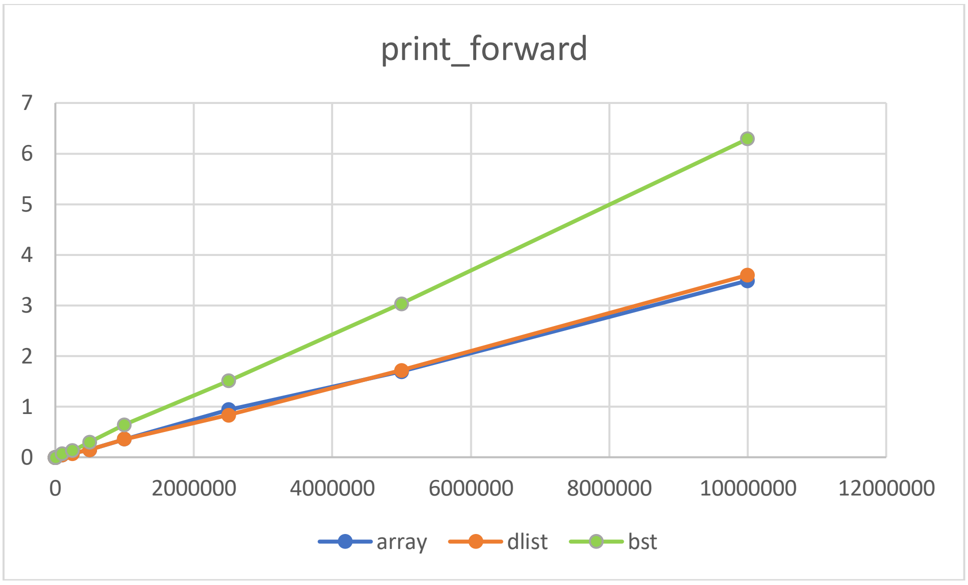


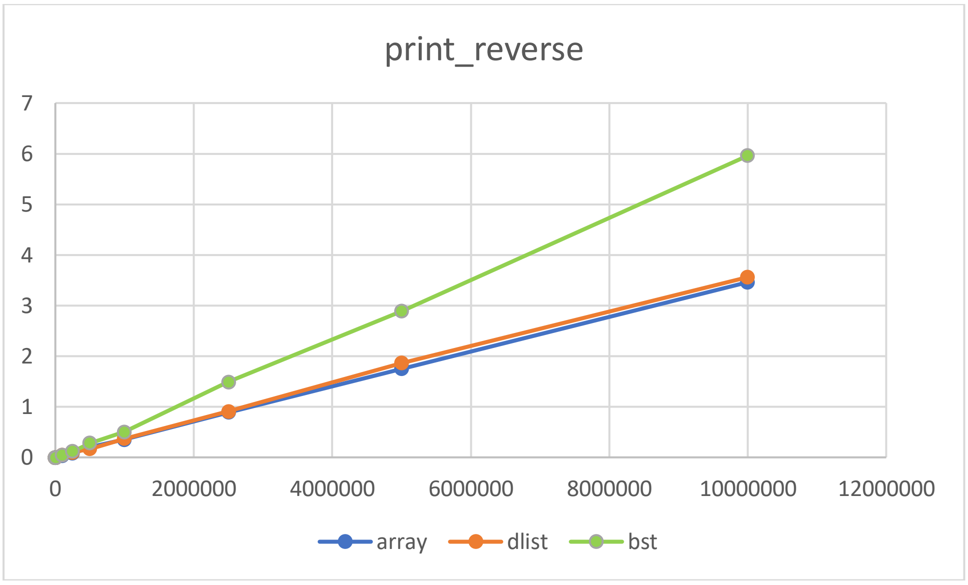
Let’s interpret clear, erase, pop_front, pop_back.
These pattern are nearly the same since it use the same procedure.
First, we implement the erase function. Then for pop_front and pop_back, we just call erase function with parameter begin or end in it. Finally, for clear, we also call erase.
If array and dlist is O(1), then bst is O(logN).

FIND:
For find function in all three data structure, it is hard to quantify since it take too short time. But we can take the average time of a large repetitive same command.

Array: 4000 times in 100000 array, total 2.43s, average is 0.00061 per query
Dlist: 4000 times in 100000 dlist, total 2.89s, average is 0.00072 per query
BST: 100000 times in 100000 bst, total 3.06s, average is 0.000031 per query

By looking for the data, we can find that bst find is much more quickly than array and dlist. That is because is just take O(logN) per query, instead of O(N) in array and dlist.

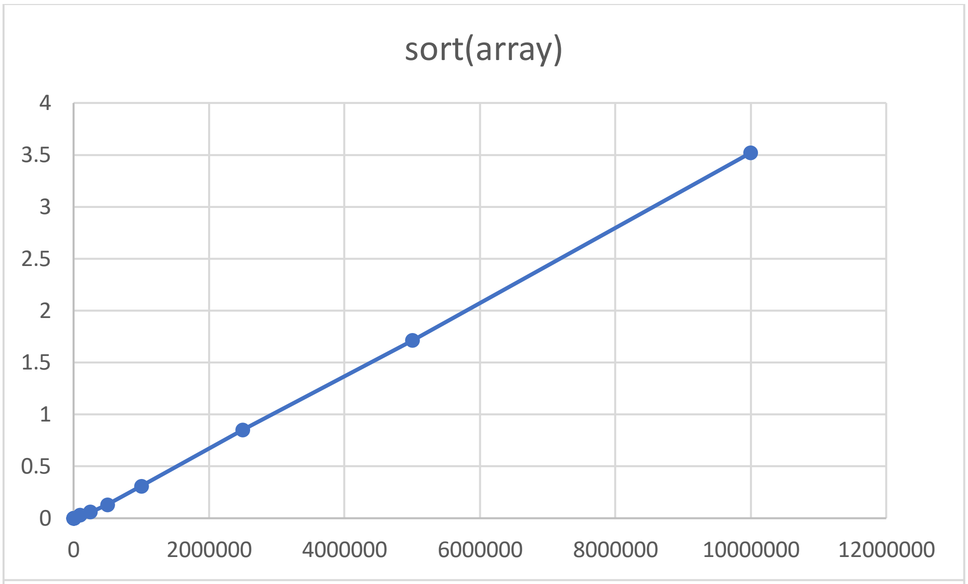


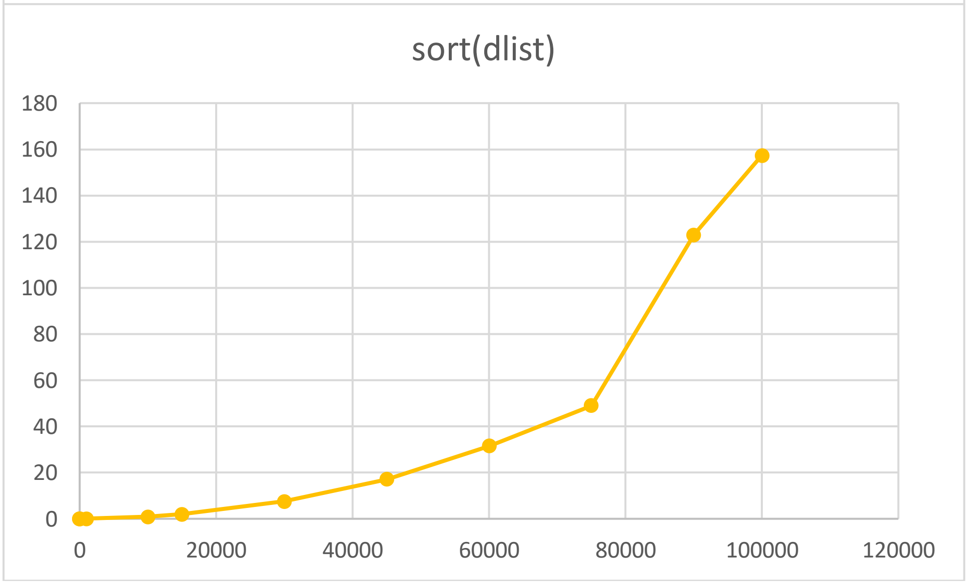


The speed of three data structure are almost the same.

But bst take a little more time than two others.

That is because we have to call the successor function of bst. And it will browse through the tree. Luckily, it will use O(1) time. But sometimes it has to go to another tree and the worst case is O(logN). So the difference between bst and other two will become larger since bst print is not a constant time, but a function of N.





For bst, it is already been sort when inserted, so there's no need do sort again.

For array, we call std::sort to sort the data, and it's nearly linear but it's actually O(NlogN).

For dlist, we implement bubble sort, which is O(N^2), so we cannot sort too large dlist, since it will take too much time. For me, 10000 dlist takes about 3 minute to sort, so it's not a good algorithm for sorting. The better way is to copy the data and to sort it using another sort function.

Appendix: Raw data
Array:

array	**1**	**10**	**100**	**1000**	**10000**	**100000**	**250000**	**500000**	**1000000**	**2500000**	**5000000**	**10000000**
push_back(time)	0	0	0	0	0	0.01	0.06	0.09	0.16	0.79	1.17	2.6
push_back(memory)	0	0	0	0	0.6367	5.996	12.12	24.11	48.1	192	383.9	767.9
pop_back	0	0	0	0	0	0	0	0	0	0.01	0.01	0.01
pop_front	0	0	0	0	0	0	0	0.01	0.01	0.02	0.04	0.07
erase	0	0	0	0	0	0.01	0.01	0.03	0.06	0.19	0.52	1.2
find	4000	times	in	100000	array	2.43	avg	0.00061	per	time		
clear(time)	0	0	0	0	0	0	0	0	0	0	0	0
sort	0	0	0	0	0	0.03	0.06	0.13	0.31	0.85	1.71	3.52
print_forward	0	0	0	0	0	0.04	0.08	0.15	0.36	0.94	1.69	3.49
print_reverse	0	0	0	0	0	0.03	0.08	0.2	0.35	0.89	1.75	3.46

Dlist:

dlist	**1**	**10**	**100**	**1000**	**10000**	**100000**	**250000**	**500000**	**1000000**	**2500000**	**5000000**	**10000000**
push_back(time)	0	0	0	0	0	0.02	0.02	0.03	0.08	0.17	0.35	0.73
push_back(memory)	0	0	0	0	0.6386	6.098	15.12	30.59	61.01	152.5	305.2	610.4
pop_back	0	0	0	0	0	0	0.01	0.01	0.01	0.03	0.06	0.15
pop_front	0	0	0	0	0	0	0	0	0.01	0.02	0.06	0.13
erase	0	0	0	0	0	0.01	0.02	0.05	0.1	0.35	0.72	1.56
find	4000	times	in	100000	dlist	2.89	avg	0.00072	per	time		
clear(time)	0	0	0	0	0	0	0	0.01	0.01	0.04	0.07	0.16
sort	0	0	0	0.01	0.83	1.99	7.53	17.06	31.56	49	122.86	157.4
print_forward	0	0	0	0	0	0.04	0.07	0.15	0.36	0.83	1.72	3.6
print_reverse	0	0	0	0	0	0.04	0.1	0.17	0.37	0.91	1.86	3.56

BST:

bst	**1**	**10**	**100**	**1000**	**10000**	**100000**	**250000**	**500000**	**1000000**	**2500000**	**5000000**	**10000000**
insert(time)	0	0	0	0	0.01	0.06	0.14	0.3	0.85	3.02	7.08	14.49
push_back(memory)	0	0	0	0	0.7188	5.4142	9.027	15.21	30.17	91.04	152.1	611.1
pop_back	0	0	0	0	0	0.01	0.03	0.06	0.14	0.44	0.79	1.54
pop_front	0	0	0	0	0	0.01	0.03	0.07	0.13	0.33	0.72	1.64
erase	0	0	0	0	0	0.01	0.09	4.24	9.68	14.38	24.03	54.47
find	100000	times	in	100000	bst	3.06	avg	0.000031	per	time		
clear(time)	0	0	0	0	0	0.01	0.04	0.07	0.32	0.85	2.05	3.59
sort	0	0	0	0	0	0	0	0	0	0	0	0
print_forward	0	0	0	0	0	0.07	0.13	0.3	0.64	1.51	3.03	6.29
print_reverse	0	0	0	0	0	0.05	0.12	0.28	0.5	1.49	2.89	5.96