Programming HW 1 Writeup

Dylan M Ang

March 2021

Contents

1 Reference

⇒ File1	Has a series of 500,000 insertions in order from 1 to 500000
⇒ File2	Has a series of 250,000 insertions in order from 1 to 250000 followed by 250,000 deletions 1 to 250000
⇒ File3	Has a series of 250,000 insertions in order from 1 to 250000 followed by 250,000 deletions 250000 to 1
⇒ File4	Has a series of 250000 random, unique insertions from 1 to 250000 followed by 250000 random, unique deletions
⇒ ADT 1	Linked List
⇒ ADT 2	Cursor List
⇒ ADT 3	Stack implemented as an Array
⇒ ADT 4	Stack implemented as a Linked List
⇒ ADT 5	Queue implemented as an Array
⇒ ADT 6	Skip List

2 Times

File	ADT	Time1	Time2	Time3	Average
1	1	0.064501	0.059457	0.059385	0.061114
2	1	73.9228	73.9897	73.9399	73.950800
3	1	0.074317	0.046418	0.041135	0.053957
4	1	11.5934	11.6341	11.6124	11.613300
1	2	0.061247	0.05905	0.058898	0.059732
2	2	340.42	340.321	340.434	340.391667
3	2	0.066172	0.060504	0.06046	0.062379
4	2	42.4495	49.7888	49.9038	47.380700
1	3	0.036738	0.033312	0.032751	0.034267
2	3	0.034676	0.031774	0.031513	0.032654
3	3	0.034417	0.03176	0.031692	0.032623
4	3	0.020147	0.016222	0.016138	0.017502
1	4	0.055112	0.048781	0.048636	0.050843
2	4	0.038699	0.038155	0.03802	0.038291
3	4	0.042772	0.038	0.038024	0.039599
4	4	0.024526	0.01934	0.019345	0.021070
1	5	0.037728	0.035137	0.034607	0.035824
2	5	0.038181	0.034352	0.03414	0.035558
3	5	0.03843	0.034055	0.033914	0.035466
4	5	0.020553 f	0.017409	0.017301	0.018421
1	6	0.154624	0.155144	0.158328	0.156032
2	6	0.120325	0.115589	0.117138	0.117684
3	6	0.136952	0.131517	0.138876	0.135782
4	6	0.13124	0.131421	0.132129	0.131597

3 Time Complexity

File	ADT	Single Insertion	Single Deletion	All Insertions	All Deletions	File
1	1	$O(1)$	N/A	$O(n)$	N/A	$O(n)$
2	1	$O(1)$	$O(n)$	$O(n)$	$O(n^2)$	$O(n^2)$
3	1	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
4	1	$O(1)$	$O(n)$	$O(n)$	$O(n^2)$	$O(n^2)$
1	2	$O(1)$	N/A	$O(n)$	N/A	$O(n)$
2	2	$O(1)$	$O(n)$	$O(n)$	$O(n^2)$	$O(n^2)$
3	2	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
4	2	$O(1)$	$O(n)$	$O(n)$	$O(n^2)$	$O(n^2)$
1	3	$O(1)$	N/A	$O(n)$	N/A	$O(n)$
2	3	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
3	3	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
4	3	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
1	4	$O(1)$	N/A	$O(n)$	N/A	$O(n)$
2	4	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
3	4	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
4	4	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
1	5	$O(1)$	N/A	$O(n)$	N/A	$O(n)$
2	5	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
3	5	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
4	5	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
1	6	$O(log\, n)$	N/A	$O(nlog\, n)$	N/A	$O(nlog\, n)$
2	6	$O(log\, n)$	$O(log\, n)$	$O(nlog\, n)$	$O(nlog\, n)$	$O(nlog\, n)$
3	6	$O(log\, n)$	$O(log\, n)$	$O(nlog\, n)$	$O(nlog\, n)$	$O(nlog\, n)$
4	6	$O(log\, n)$	$O(log\, n)$	$O(nlog\, n)$	$O(nlog\, n)$	$O(nlog\, n)$

4 Discussion

ADT 1 is a singly Linked List. File 1 and File 3 are $O(n)$, where File 2 and 4 are $O(n^2)$. This difference is due to the fact that the linked list inserts at the front, so by the time all 250,000 numbers are inserted, the list is ordered from 250,000 at the front, to 1 at the very end. File 2 and 3 have similar operations, the only difference is the order in which they make the deletions. File 2 deletes in ascending order, from 1 to 250,000. File 3 deletes in descending order, from 250,000 to 1, and this results in shorter times because the next item that requires deletion is always at the front of the list. Whereas, with File 2, the first item to be deleted, 1, is at the very end of the list. This results in $O(n)$ for the insertions, and $1 + 2 + 3 + ... + n = \frac{n(n+1)}{2} = O(n^2)$ for the deletions in File 2. File 4 is also quite slow, but not as slow as File 2. This is because File 4's insertions and deletions are in random order, so it isn't convenient as File 1 or 3, where insertions and deletions are $O(1)$, but not as inconvenient as File 2, which represents the worst-case scenario. File 4 is somewhere in between.

ADT 2 is a Cursor List. It has the slowest time with File 2, then File 4. File 1 and 3 are quite fast. Cursor Lists, are essentially just Linked Lists, but rather than pointers, the Cursor List keeps index values of arrays. For this reason, it has the same time complexity as a normal Linked List. We see the exact same pattern of performance with the Cursor List as we saw in the Linked List. It is slower faster on File 1 and File 3 because they represent ideal conditions. The ideal conditions being due to the fact that Cursor List inserts at the beginning of the list, so values being deleted in File 3 are at the beginning of the list. However, it is slower on File 4, when the order of the insertions and deletions are random, the list needs to be searched, and indices may need to change, thus more indices are being shifted. Finally, it's slowest on File 2 because it represents the absolute worst possible conditions, where the elements to be deleted are always at the end of the list.

ADT 3 is a Stack being implemented as an array. Stacks are LIFO, or first in, last out, data structures. So, operations can only be performed on the top of the stack, the most recently pushed element. Usually, deletion for an array is $O(n)$ worst-case, representing when the first element in the array is deleted, and all subsequent elements must be shifted. However, because this array is implementing a stack, this isn't a concern. The stack's pop function does not take a value to delete, it always deletes the last element in the array, avoiding the need to shift all the array elements. So, the stack has the same time complexity for all files with deletions.

ADT 4 is also a Stack, but implemented as a Linked List. While the Stack array is appending values to the end of the array, the list pushes elements at the front of the list. Because it's a stack, it pops at the front of the list as well. Despite the differences between the two implementations, they have the same time complexity, because they choose the respective $O(1)$ push and pop operations for their data structure. So, the stack array has the same time complexity for all files with deletions, because it isn't searching for the right value to delete, it simply deletes at the top of the stack, which is the beginning of the list.

ADT 5 is a Queue implemented as an array. A queue is a FIFO, or first in, first out, data structure. Like a stack, a Queue only cares about the order in which an element has been enqueued, not the actual value. All of the files have similar times, because there will always be 500,000 operations. The queue takes one operation to dequeue and enqueue, as it ignores the actual value of the element, and an insertion will cost just as much as a deletion. The 500,000 insertions of File 1 will take just as long as the insertions and deletions of FIle 2,3, and 4 because it isn't searching for specific elements.

ADT 6 is a Skip List. It is $O(nlog\ n)$ for all files, with $O(log\ n)$ insertions and deletions. Because the heights of the elements are randomly generated, it doesn't matter the order in which deletions are presented. The random number generator will create the express lanes that ensure $O(log\, n)$ search. Technically, a Skip List could be $O(n)$ in the absolute worst-case scenario, however is is $O(log\, n)$ for search/insertion/deletion in most situations.

Both stack implementations and the queue array has very similar times, which is expected, considering they are the types that avoid costly deletions by simply deleting at the top of the stack, or beginning of the queue. I thought it was interesting how the Skip List isn't first place in any of the files, but it is the most consistent while still conserving deletion of specific values. The other List ADT's beat the Skip List for specific files, but they both have weak points when they recieve a file with slightly less-than-ideal conditions. Finally, the Cursor List and Linked list share the same time complexity. Yet, the Cursor List is shockingly slower for File 2 and File 4. So, same complexity, same underlying structure - they are linked lists - why is the Cursor List so much slower? Well, when a deletion is performed on a Cursor List, there is a lot of array indexing happening that takes a lot of time, especially when compared to the Linked List, which just has to follow a pointer. The extra time comes from the while loop inside Cursor List's implementation of findPrevious(), as this is an $O(n^2)$ operation, and any significant time difference would be here. It can't be anywhere else as we don't see such drastic time differences for File 1 with just insertions, or the best case deletions in File 3. Examining the code for findPrevious(), they are doing in essence the same thing, checking that the next element isn't a null pointer or at index 0, and checking that the element isn't the one we're looking for. Due to this, I believe that the process of deleting is slightly slower in Cursor List, and the large amount of worst-case deletions amplifies the time gap.