ECS60 Programming Homework 1 Write Up

Zhen Zhang (#913435403, ezzhang@ucdavis.edu)

Time for ADT for each File

File	ADT#	Time#1	Time#2	Time#3	mean
File1	1	0.060276	0.053819	0.053532	0.055875667
File2	1	73.9704	73.377	73.3614	73.5696
File3	1	0.041992	0.038394	0.037542	0.039309333
File4	1	48.4649	48.1956	48.211	48.2905
File1	2	0.039841	0.037563	0.03751	0.038304667
File2	2	342.878	342.126	342.818	342.6073333
File3	2	0.053573	0.048772	0.048758	0.050367667
File4	2	172.607	208.42	207.857	196.2946667
File1	3	0.032802	0.032232	0.03198	0.032338
File2	3	0.031091	0.030581	0.030425	0.030699
File3	3	0.032102	0.031483	0.030575	0.031386667
File4	3	0.035066	0.03411	0.032606	0.033927333
File1	4	0.045851	0.045568	0.045576	0.045665
File2	4	0.036368	0.036213	0.036305	0.036295333
File3	4	0.036665	0.036206	0.036199	0.036356667
File4	4	0.039455	0.038634	0.038752	0.038947
File1	5	0.033876	0.033224	0.032675	0.033258333
File2	5	0.033023	0.03238	0.032099	0.032500667
File3	5	0.033741	0.03297	0.032127	0.032946
File4	5	0.035637	0.034635	0.034414	0.034895333
File1	6	0.143875	0.146164	0.148562	0.146200333
File2	6	0.111638	0.109829	0.112652	0.111373
File3	6	0.128556	0.125547	0.126286	0.126796333
File4	6	0.317571	0.32901	0.33372	0.326767

Time complexity for ADT for each file

File	ADT#	individual insertion	individual deletion	entire insertions	entire deletions	entire file
File1	1	O(1)	NA	O(N)	NA	O(N)
File2	1	O(1)	O(N)	O(N)	O(N^2)	O(N^2)
File3	1	O(1)	O(1)	O(N)	O(N)	O(N)
File4	1	O(1)	O(N)	O(N)	O(N^2)	O(N^2)
File1	2	O(1)	NA	O(N)	NA	O(N)
File2	2	O(1)	O(N)	O(N)	O(N^2)	O(N^2)
File3	2	O(1)	O(1)	O(N)	O(N)	O(N)
File4	2	O(1)	O(N)	O(N)	O(N^2)	O(N^2)
File1	3	O(1)	NA	O(N)	NA	O(N)
File2	3	O(1)	O(1)	O(N)	O(N)	O(N)
File3	3	O(1)	O(1)	O(N)	O(N)	O(N)
File4	3	O(1)	O(1)	O(N)	O(N)	O(N)
File1	4	O(1)	NA	O(N)	NA	O(N)
File2	4	O(1)	O(1)	O(N)	O(N)	O(N)
File3	4	O(1)	O(1)	O(N)	O(N)	O(N)
File4	4	O(1)	O(1)	O(N)	O(N)	O(N)
File1	5	O(1)	NA	O(N)	NA	O(N)
File2	5	O(1)	O(1)	O(N)	O(N)	O(N)
File3	5	O(1)	O(1)	O(N)	O(N)	O(N)
File4	5	O(1)	O(1)	O(N)	O(N)	O(N)
File1	6	O(logN)	NA	O(NlogN)	NA	O(NlogN)
File2	6	O(logN)	O(logN)	O(NlogN)	O(NlogN)	O(NlogN)
File3	6	O(logN)	O(logN)	O(NlogN)	O(NlogN)	O(NlogN)
File4	6	O(logN)	O(logN)	O(NlogN)	O(NlogN)	O(NlogN)

Explanation of why ADTs took longer or shorter with different files

1. LinkedList: For Linked List, File 1 just insert items one by one, but for File 2, after inserting, the delete need to be done by delete 1 first, then 2 second, so on so forth. But the principle of the LinkedList is that, when inserting a new node, it pushes the previous header and then insert the element as the new header. So at last the LinkedList's order 250000, 249999, . . . , 2, 1. Since we can only look for the element we want to delete from the header, it takes time 250000 to find the element 1, and then delete it, so it comes a time complexity of O(N). When it comes to File3, the insert is identical, but the delete comes as the order of 250000, 249999, . . . , 2, 1, so it only takes O(1) each time to do the delete, because the element is where the header lies. For the last file, File 4, the time is half of File 2, since on average, it will take about N/2 time to find the element we want to delete. When comparing File1 with File3, it is just comparing insertion time with the delete time. Since insert involves creating a ListNode and then let the header pointer points to the new Node and the new Node points to the previous Node which is pointed to by the header, then give the value to the new Node, while delete only involves deleting the ListNode and then let the previous Node points to the afterwards Node (Now the find time is only 1 in File3), we know insert does more jobs than delete does, and that is why File1 costs more time than File3.

2. CursorList: The underlying time complexity difference is the same as that of LinkedList. The difference is, instead of creating a new Node each time, it updates a Node's value and then let the header Node points to that one, and the new one to the previous one, so on so forth. That means the first inserted element is always the last one to get, following the chain. So when deleting in a order of 1, 2, ..., 250000, it will cost O(N) time to the end. This is the difference between File2 and 3, in different delete orders. For File4, also half the time of that of File2, since the find time is half of File2. Since insertion does less operation than deletion, then File3 costs more time than File1.

3. Stack Array, StackList, Queue Array: For the stack, it will not take too much time, since when inserting, it will push (enqueue) on the top, and when deleting, it will pop (enqueue) the element on the top (last), regardless of the value. (the words in parenthesis is for the queue array).

4. SkipList: For all of them, they have the same time complexity, but the result is different. The reason is, for File1, the time is almost 2Nlog(2N), since the links (number=500000) doubles when compared to that of File2 and 3, which is 2*Nlog(2N) (number = 250000). So File1 costs more time than File2 and 3. For File4, when the order is random, it is a little complicated. It has to go thought all the levels to find a place to insert, since its place is undetermined, which time is at least log(N). Compared with what File1 does, when inserting an element that is larger than anyone else in the SkipList, sometimes it can get to a very large element directly, omitting some levels, so it costs less time. Then in conclusion we know random costs more time than ordered elements when inserting into a SkipList.

Comparison for ADT

1. Stack Array, Stack List, Queue Array all have time complexity O(1), and Linked List, Cursor List with insertion and delete in reverse order have time complexity O(1) as well.
2. SkipList have time complexity O(logN), so it needs more time.
3. Stack Array, StackList with deletion in ascending order have time complexity O(N).

The reason for same complexity have different time value

1. When comparing the running time of Stack Array, Stack List and Queue Array, I noticed that Stack List is implemented on a Linked List while Stack Array and Queue Array are implemented on a array. Given a fixed size (in this question is 500000), array outperformance Linked List, in the sense of random access, no extra storage needed for reference and the use of local data caches. So Stack Array and Queue Array will almost identical time, and both spend less time than Stack List.
2. StackList, Linked List are all Linked Lists, so their time are almost the same.
3. Now discuss why CursorList is slower than normal LinkedList. In terms of deleting in CursorList and LinkedList, the delete operation can be decomposed into two parts: first find the node needs to be deleted, then delete it. The operation delete spends almost the same time between CursorList and

LinkedList. Now discuss the find operation. The process of find in LinkedList is, use the pointer to find the next, check the value using pointer, if it is not the value we are currently looking for, use the pointer to find the next one again. It only uses pointer to find next. For the CursorList, things become much more complicated. This is because CursorList use vector index instead of pointer or iterator to extract the element value. To be concrete, we use curSorSpace[x].element to get the element value, and use curSorSpace[x].next to get the next index if this value is not we are looking for. Since pointer or vector iterator is usually faster than vector index, it turns out that CursorList will spend more time to do a find operation, which needs more time to delete.