Big O Table F

ADTs	**Actions**	**Files**			
		File1.dat	**File2.dat**	**File3.dat**	**File4.dat**
LinkedLists	**Individual Insert**	O(1)	O(1)	O(1)	O(1)
	Individual Delete	N/A	O(n)	O(1)	O(n)
	Total Insert	O(n)	O(n)	O(n)	O(n)
	Total Delete	N/A	O(n^2)	O(n)	O(n^2)
	Entire File	**O(n)**	**O(n^2)**	**O(n)**	**O(n^2)**
CursorList	**Individual Insert**	O(1)	O(1)	O(1)	O(1)
	Individual Delete	N/A	O(n)	O(1)	O(n)
	Total Insert	O(n)	O(n)	O(n)	O(n)
	Total Delete	N/A	O(n^2)	O(n)	O(n^2)
	Entire File	**O(n)**	**O(n^2)**	**O(n)**	**O(n^2)**
Stack Array	**Individual Insert**	O(1)	O(1)	O(1)	O(1)
	Individual Delete	N/A	O(1)	O(1)	O(1)
	Total Insert	O(n)	O(n)	O(n)	O(n)
	Total Delete	N/A	O(n)	O(n)	O(n)
	Entire File	**O(n)**	**O(n)**	**O(n)**	**O(n)**
Stack Linked List	**Individual Insert**	O(1)	O(1)	O(1)	O(1)
	Individual Delete	N/A	O(1)	O(1)	O(1)
	Total Insert	O(n)	O(n)	O(n)	O(n)
	Total Delete	N/A	O(n)	O(n)	O(n)
	Entire File	**O(n)**	**O(n)**	**O(n)**	**O(n)**
Queue Array	**Individual Insert**	O(1)	O(1)	O(1)	O(1)

	Individual Delete	N/A	O(1)	O(1)	O(1)
	Total Insert	O(n)	O(n)	O(n)	O(n)
	Total Delete	N/A	O(n)	O(n)	O(n)
	Entire File	**O(n)**	**O(n)**	**O(n)**	**O(n)**
Skip List	**Individual Insert**	O(log n)	O(log n)	O(log n)	O(log n)
	Individual Delete	N/A	O(log n)	O(log n)	O(log n)
	Total Insert	O(nlog n)	O(nlog n)	O(nlog n)	O(nlog n)
	Total Delete	N/A	O(nlog n)	O(nlog n)	O(nlog n)
	Entire File	O(nlog n)	O(nlog n)	O(nlog n)	O(nlog n)

Average Times (s)

	File1	File2	File3	File4
LinkedList	0.065764	85.3703	0.055188	58.9260
CursorList	0.046729	390.650	0.059986	221.482
StackAr	0.038057	0.037099	0.036868	0.039441
StackLi	0.055955	0.044492	0.044733	0.047863
QueueAr	0.039741	0.050322	0.039665	0.044133
SkipList	0.173963	0.128394	0.146860	0.471937

Write Up

To preface, File1.dat inserted values from 1 to 250,000 in that order with no deletions. File2.dat had the same, but also deleted 250,000 to 1, in that order. File3.dat had the same inserts as File1.dat and then deleted 1 to 250,000 in that order. File4.dat inserted values between 1 and 250,000 randomly and deleted them randomly.

To start with, from the average time table, we can see that the times for StackAr are relatively consistent throughout, not changing much regardless of the file used. This is because stacks operate on first in, last out. Meaning the only thing can be removed from the stack is the most recent entry. What this means is that when parsing through the delete commands, it doesn't matter what value the integer the file tells the stack to delete, the stack will always pop the most recent value added. The same applies to StackLi, which is slightly slower than StackAr due to increased memory allocation requirements, and QueueAr, which operates on a first in, first out basis.

Thus, we can also see why the big O for all four files for StackAr, StackLi, and QueueAr were the same throughout - O(n). Inserting an individual integer is O(1) as the integer is inserted at the end no matter what and an individual deletion is O(1) as the integer is deleted from the end or the beginning for stacks and queue respectively.

Linked lists and cursor lists also behave similarly. An individual insert takes O(1) as the value is inserted where the pointer is, and both of these lists are unsorted. An individual delete then takes O(n). The delete itself is O(1) as the value is simply removed from the list. But the search takes O(n) as the pointer must navigate through every node until it finds the correct value. With that said, the reason why File3's individual delete is O(1) is because all the deletions occur where the pointer starts - the pointer is never moved so insertions are made to the beginning of the list. Thus, when deletions are made, deletions are made from the front of the list, where the pointer is already at. Whereas in File2, the pointer must navigate through the entire list before removing the value at the opposite end of the linked list. File3 is an example of the best case scenario for deletions in a linked list while File2 is an example of the worst case scenario for deletions in a linked list. File4 falls in between File2 and File3 as the deletions occur in the middle of the list rather than at the ends. For this file, the pointer must start at one end of the linked list and search the nodes one by one for the value to delete somewhere in the middle of the list. This is ultimately faster than File2 however, as File2 requires the pointer to go to the other end of the list for every deletion.

When comparing the run times for linked list and cursor lists, we also see that the times for the two are comparable for File1 and File3. In fact, for File1, cursor lists are faster than linked lists, likely due to less memory usage. However, for File2 and File4, cursor lists are far slower. This is because each value of a cursor list contains an index to another value in the list, rather than a pointer like in linked lists. So when the value is removed, the indices have to be reassigned.

Lastly, skip lists are a set of multiple linked lists that allow for binary search. Skip lists are also ordered, meaning that value cannot just be inserted anywhere. Thus, an individual insert for a skip list has O(log n). For the same reason, an individual delete has O(log n). Thus the upper bound of the entire file would be O(nlog n).

Skip lists having multiple connected linked lists that can be used to search also explains why File2 and File3 have similar run times for Skip lists as the skip list can quickly navigate from one end of the linked list that contains all of the values to the other end of the linked list using the linked list with the fewest values. File 2 is slightly faster than File 3 due to the organized nature of the list. And since the pointer starts at the beginning of the skip list, it is ever so slightly faster to remove the elements right at the beginning of the list than it is to move elements at the very end of the list, as it does in File3. In comparison, File4 takes longer than File2 and File3. This is due to the randomized nature of the inserts and deletions, and requiring more search time within the linked list before something is added or removed. Likewise, it also explains why the File4's randomized integers is faster for skip lists than linked lists and cursor lists as it can use binary search to search and delete values whereas the linked list and cursor list must search each individual node for the value to delete.