Program 1: Timetest Write-up

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ADT** | **File #** | **Time #1 (s)** | **Time #2 (s)** | **Time #3 (s)** | **Average Time (s)** |
| **LinkedList** | 1 | 0.355023 | 0.32205 | 0.322751 | 0.333275 |
| 2 | 2238.16 | 2226.34 | 2227.05 | 2230.517 |
| 3 | 0.254283 | 0.222329 | 0.225615 | 0.234076 |
| 4 | 1938.95 | 1934.22 | 1935.76 | 1936.31 |
| **CursorList** | 1 | 0.228225 | 0.216317 | 0.21651 | 0.220351 |
| 2 | 9412.58 | 9403.63 | 9405.96 | 9407.39 |
| 3 | 0.288544 | 0.27666 | 0.277114 | 0.280773 |
| 4 | 4712.66 | 9443.71 | 9451.39 | 7869.253 |
| **StackAr** | 1 | 0.183388 | 0.183166 | 0.182921 | 0.183158 |
| 2 | 0.171694 | 0.171322 | 0.171099 | 0.171372 |
| 3 | 0.170842 | 0.1707 | 0.170375 | 0.170639 |
| 4 | 0.179951 | 0.179732 | 0.179711 | 0.179798 |
| **StackLi** | 1 | 0.267989 | 0.266099 | 0.265797 | 0.266628 |
| 2 | 0.212148 | 0.213908 | 0.214038 | 0.213365 |
| 3 | 0.209862 | 0.209504 | 0.210325 | 0.209897 |
| 4 | 0.217025 | 0.217087 | 0.217333 | 0.217148 |
| **QueueAr** | 1 | 0.196364 | 0.196075 | 0.195901 | 0.196113 |
| 2 | 0.184621 | 0.183762 | 0.183211 | 0.183865 |
| 3 | 0.183552 | 0.183228 | 0.182939 | 0.18324 |
| 4 | 0.190132 | 0.189204 | 0.188688 | 0.189341 |
| **SkipList** | 1 | 0.856161 | 0.908896 | 0.893867 | 0.886308 |
| 2 | 0.65389 | 0.66362 | 0.652891 | 0.6568 |
| 3 | 0.76573 | 0.774755 | 0.771929 | 0.770805 |
| 4 | 3.03403 | 3.17835 | 3.21428 | 3.14222 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **ADT** | **File #** | **Individual Insertion** | **Individual Deletion** | **Series of insertions** | **Series of deletions** | **Entire File** |
| **LinkedList** | 1 | O(1) | - | O(n) | - | O(n) |
| 2 | O(1) | O(n) | O(n) | O(n2) | O(n2) |
| 3 | O(1) | O(1) | O(n) | O(n) | O(n) |
| 4 | O(1) | O(n) | O(n) | O(n2) | O(n2) |
| **CursorList** | 1 | O(1) | - | O(n) | - | O(n) |
| 2 | O(1) | O(n) | O(n) | O(n2) | O(n2) |
| 3 | O(1) | O(1) | O(n) | O(n) | O(n) |
| 4 | O(1) | O(n) | O(n) | O(n2) | O(n2) |
| **StackAr** | 1 | O(1) | - | O(n) | - | O(n) |
| 2 | O(1) | O(1) | O(n) | O(n) | O(n) |
| 3 | O(1) | O(1) | O(n) | O(n) | O(n) |
| 4 | O(1) | O(1) | O(n) | O(n) | O(n) |
| **StackLi** | 1 | O(1) | - | O(n) | - | O(n) |
| 2 | O(1) | O(1) | O(n) | O(n) | O(n) |
| 3 | O(1) | O(1) | O(n) | O(n) | O(n) |
| 4 | O(1) | O(1) | O(n) | O(n) | O(n) |
| **QueueAr** | 1 | O(1) | - | O(n) | - | O(n) |
| 2 | O(1) | O(1) | O(n) | O(n) | O(n) |
| 3 | O(1) | O(1) | O(n) | O(n) | O(n) |
| 4 | O(1) | O(1) | O(n) | O(n) | O(n) |
| **SkipList** | 1 | O(log n) | - | O(n\*logn) | - | O(n\*logn) |
| 2 | O(log n) | O(log n) | O(n\*logn) | O(n\*log n) | O(n\*log n) |
| 3 | O(log n) | O(log n) | O(n\*logn) | O(n\*log n) | O(n\*log n) |
| 4 | O(log n) | O(log n) | O(n\*logn) | O(n\*log n) | O(n\*log n) |

First, I’ll explain the differences in times between the 4 files for each data structure. *For File1.dat*, the linked list was very fast because it inserts at the head of the list, which is an O(1) operation (no traversals needed). Inserting n numbers results in O(n). *For File2.dat*, the linked list was very slow because the deletions are done in the same order as insertions. Numbers are inserted at the front of the list so when numbers 1-1250000 are added, 1250000 will be at the head of the list and 1 will be at the tail. Although insertion is O(1) like in File1.dat, deleting 1 from the list means you always have to traverse the entire list just to get to 1 before you can delete it, making deletions an O(n) operation. Doing these insertions and deletions for n items is O(n2). *For File3.dat*, the linked list is very fast because the deletions are done in a reverse order as the insertions. Once again, insertions at the head of the linked list are O(1) and 1250000 ends up at the head of the list. However, this time, the numbers are always deleted from the head of the list since deletions start with 1250000 and end with 1, so deletions are O(1). Repeating the insertions and deletions n times gives us O(n). Notice that File3.dat was a bit faster than File1.dat. One explanation is that File1.dat calls malloc() twice as much as File3.dat which slows it down. Finally, *for File4.dat*, the insertions are again O(1), but deletions are random, so in a worst case scenario, you would have to traverse the entire list like you had to with File2.dat so it would be O(n2). However, chances are that number will be somewhere in the middle of the list (on average) since we are assuming perfectly random numbers giving O(n2/2). That’s why File4.dat finishes a bit faster than File2.dat.

The same arguments that were used for linked list can be used can be used to explain the cursor list times (except a cursor list uses an array). One small difference in the cursor list is that File1.dat was faster than File3.dat. This is probably because the cursor list allocates 2500000 memory ahead of time no matter how many insertions are made, so there’s no difference in speed in terms of allocating space for insertions. However, the extra deletions slow down File3.dat.

For the array implementation of the stack, deletions are O(1) no matter the file because the order of the deletions is ignored since pop() always removes from the top of the stack. Similarly, insertions are O(1) because they add to the top of the stack so there is no traversal needed. So for every file, processing n operations would be O(n). That’s why all the times are about the same – Files 2-4 are identical operations. However, File1.dat is a little bit slower, probably because insertions take longer than deletions. The same argument works for the linked list version of the stack. However, there’s a larger difference in times between File1.dat and the rest of the files because memory allocation for each individual ListNode takes more time than deletion of that ListNode, and there are twice as many insertions in File1.dat. The queue has the same time complexity as the stack and it essentially works the same way, except deletion occurs from the opposite end of the insertions. Deletions and insertions are O(1) and repeating it for n items gives O(n). Therefore, the arguments used for stack also can be made for the array. Again, all the times are similar, however because insertions take more time than deletions, File1.dat is the slowest.

The skip list is essentially a linked list with binary search. Because insertions and deletions use binary search to find where to insert or what to remove, each operation will take O(logn) and n operations will take O(n\*logn), which is why every file has O(n\*logn) complexity. Comparing the times, File1.dat was slower than File2.dat and File3.dat because it makes twice as many insertions (malloc()). Also, because it’s sorting the numbers from smallest to greatest, it must travel to the end of the list every time it makes an insertion. File2.dat was the faster than File3.dat because deletions occur at the head of the list. It never leaves the head node of the skip list. It just goes down the height of the head element until it reaches the first element to delete it. This is the most optimal route since there is no horizontal movement. File3.dat is slower because deletions occur at the end of the list. Reaching the end requires jumping from one node to another until you finally get to the last number, which requires more steps. However, this is the most optimal way to get there so it is still somewhat efficient. File4.dat was the worst because it’s random so it will have the most horizontal movement. Unlike File2.dat and File3.dat which only go forward and take the minimal route, File4.dat will overshoot a lot and spend a lot of time traversing the list at the lowest level.

Now I’ll compare data structures. Comparing the linked list with the cursor list, we see that in File1.dat, cursor list was faster. This makes sense, since the linked list performs costly allocations every time it inserts a new list node, while the cursor list allocates memory only once at the beginning. Also, the cursor list data is contiguous unlike list nodes, so there would be less cache misses. However, the other 3 files were slower for cursor list. Analyzing File3.dat which has no traversals, you can deduce that link list deletions are faster than cursor list deletions, since from File1.dat we found that insertions are faster in the cursor list. However, for File2.dat and File4.dat, there is a huge difference in times. We know that both the linked list and the cursor list will traverse the same number of nodes, so the difference should be in the find() function (how each ADT transverses the list). Both data structures have a single while() loop. However, running it through GDB shows the difference. In cursor list, when the condition in the while loop runs, *cursorSpace[ cursorSpace[ itr ].next ].element != x* (CursorList.cpp, line 139) and cursorSpace tries to access a specific element in the array, the [] operator is overloaded. Instead of simply returning the value, it calls an if statement to make sure the index is not out of bounds *if(index<0 || index>=currentSize) throw ArrayIndexOutOfBounds();* (vector.h, lines 39-40). The linked list does not have this check every time it tries to access an element. This significantly slows down deletions for the cursor list.

Comparing the stack array and the stack list, there isn’t much of a difference. However, the stack list is slower because it has more calls to malloc() / new to create each list node. Also, the memory in an array is contiguous so more can fit into a CPU’s cache at once, resulting in faster speeds overall for the stack array. Similarly, the queue array is faster than the stack list because it uses an array instead of a linked list. However, the queue array was a little bit slower than the stack array probably due to its organization in CPU cache? One strange thing to point out is that the linked list took longer than stack list for File1.dat even though theoretically they are acting the same way for the insertions. However, if we compare the insert() and push() code, the linked list has an extra if() statement (LinkedList.cpp, line 77), while the push() function does not (StackLi.cpp, line 110). This if statement is probably the reason File1.dat is faster for the stack list than the linked list.

Finally, the skip list is not comparable to the other data structures. It fits right in between all the other data structures in terms of speed because of the binary search.