Andy Pham

ADT HW #1 Write-Up

**Table 1: Time Values for Each Run**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **File** | **ADT** | **Time #1** | **Time #2** | **Time #3** | **Average** |
| File1 | Linked List | 0.106 | 0.105 | 0.106 | 0.106 |
| File2 | Linked List | 57.152 | 57.22 | 57.156 | 57.176 |
| File3 | Linked List | 0.083 | 0.08 | 0.08 | 0.081 |
| File4 | Linked List | 32.182 | 32.119 | 32.118 | 32.140 |
| File1 | Cursor List | 0.091 | 0.082 | 0.081 | 0.0846 |
| File2 | Cursor List | 302.051 | 302.243 | 302.039 | 302.111 |
| File3 | Cursor List | 0.109 | 0.1 | 0.1 | 0.103 |
| File4 | Cursor List | 150.101 | 160.904 | 160.92 | 157.308 |
| File1 | Stack Array | 0.075 | 0.071 | 0.072 | 0.0726 |
| File2 | Stack Array | 0.072 | 0.072 | 0.071 | 0.0716 |
| File3 | Stack Array | 0.067 | 0.066 | 0.067 | 0.0666 |
| File4 | Stack Array | 0.071 | 0.07 | 0.069 | 0.07 |
| File1 | Stack List | 0.096 | 0.097 | 0.097 | 0.0966 |
| File2 | Stack List | 0.079 | 0.08 | 0.079 | 0.0793 |
| File3 | Stack List | 0.078 | 0.078 | 0.078 | 0.078 |
| File4 | Stack List | 0.083 | 0.083 | 0.083 | 0.083 |
| File1 | Queue Array | 0.076 | 0.076 | 0.076 | 0.076 |
| File2 | Queue Array | 0.072 | 0.071 | 0.071 | 0.0713 |
| File3 | Queue Array | 0.073 | 0.073 | 0.072 | 0.0726 |
| File4 | Queue Array | 0.075 | 0.074 | 0.074 | 0.0743 |
| File1 | Skip List | 0.324 | 0.326 | 0.324 | 0.325 |
| File2 | Skip List | 0.234 | 0.232 | 0.236 | 0.234 |
| File3 | Skip List | 0.274 | 0.272 | 0.279 | 0.275 |
| File4 | Skip List | 0.409 | 0.428 | 0.432 | 0.423 |

**Table 2:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Individual Insertion** | **Individual Deletion** | **Entire series of insertions** | **Entire series of deletions** | **Entire File** |
| **Linked List** | | | | | |
| File1.dat | O(1) | N/A | O(N) | N/A | O(N) |
| File2.dat | O(1) | O(N) | O(N) | O(N^2) | O(N^2) |
| File3.dat | O(1) | O(1) | O(N) | O(N) | O(N) |
| File4.dat | O(1) | O(N) | O(N) | O(N^2) | O(N^2) |
| **Cursor List** | | | | | |
| File1.dat | O(1) | N/A | O(N) | N/A | O(N) |
| File2.dat | O(1) | O(N) | O(N) | O(N^2) | O(N^2) |
| File3.dat | O(1) | O(1) | O(N) | O(N) | O(N) |
| File4.dat | O(1) | O(N) | O(N) | O(N^2) | O(N^2) |
| **Stack Array** | | | | | |
| File1.dat | O(1) | N/A | O(N) | N/A | O(N) |
| File2.dat | O(1) | O(1) | O(N) | O(N) | O(N) |
| File3.dat | O(1) | O(1) | O(N) | O(N) | O(N) |
| File4.dat | O(1) | O(1) | O(N) | O(N) | O(N) |
| **Stack List** | | | | | |
| File1.dat | O(1) | N/A | O(N) | N/A | O(N) |
| File2.dat | O(1) | O(1) | O(N) | O(N) | O(N) |
| File3.dat | O(1) | O(1) | O(N) | O(N) | O(N) |
| File4.dat | O(1) | O(1) | O(N) | O(N) | O(N) |
| **Queue Array** | | | | | |
| File1.dat | O(1) | N/A | O(N) | N/A | O(N) |
| File2.dat | O(1) | O(1) | O(N) | O(N) | O(N) |
| File3.dat | O(1) | O(1) | O(N) | O(N) | O(N) |
| File4.dat | O(1) | O(1) | O(N) | O(N) | O(N) |
| **Skip List** | | | | | |
| File1.dat | O(log N) | N/A | O(N log N) | N/A | O(N log N) |
| File2.dat | O(log N) | O(log N) | O(N log N) | O(N log N) | O(N log N) |
| File3.dat | O(log N) | O(log N) | O(N log N) | O(N log N) | O(N log N) |
| File4.dat | O(log N) | O(log N) | O(N log N) | O(N log N) | O(N log N) |

In the linked list, file 1 was quick because it was all insertions and since the linked list inserts the values from the beginning of the list in this case, the process is very quick. However, because the linked list inserts from the beginning of the list, deletions that start at the end of the list will go slowly. This is the case in file 2 where insertions are done 1 to 250000 and then deletions are done 1 to 250000. Since the first elements inserted get moved to the end of the list, the subsequent deletions have to go backwards up the list from the end, leading to a large amount of time. File 3 takes little time because the deletion values are found at the beginning of the list and follows the list order. File 4 takes a decent amount of time since the insertions and deletions are randomized, so the iterator has to locate the deletions that could be located in the middle of the list or at the end. However, the iterator does not have to go backwards up the list in order to delete all the values so it takes less time than File 2 did.

In cursor list, file 1 was quick because it was all insertions and all the list needs to do is change the index of the inserted element. Values are inserted from the beginning of the list and File 2 takes substantially long because the deletion happens from the end of the list to the beginning, so the iterator has to go through the whole list to delete the values. File 3 takes little time because the deletion values are found at the beginning of the list and follows the list order. File 4 takes a decent amount of time since the insertions and deletions are randomized, so the iterator has to locate the deletions that could be located in the middle of the list or at the end. However, the iterator does not have to go backwards up the list in order to delete all the values so it takes less time than File 2 did. The last 2 runs for file 4 are faster than the first run because the “cursorSpace” list has already been created. In the first run, the program has to create the 250000 long array and it takes awhile for the computer to find such a continuous segment of memory to allocate.

In both stack array and stack list, insertions happen at the top of the stack, so file 1 goes very fast. The three files containing deletions also go very fast for stacks because the stacks do not care about the actual values being deleted since they can only remove values from one end. Thus all 4 files in stack array and stack list execute quickly.

In queue array, insertions happen at the beginning of the queue, so file 1, being all insertions, executes quickly. Queues also do not care about the actual values being deleted and just removes values at the end of the queue, so all 3 files with deletions execute fast as well. Thus all 4 files in queue array have short times.

In skip list, a form of binary search is used to find values. This cuts the time needed to find deletions at any part of the list by a lot, especially if the deleted values are located at the boundaries of the list. This is why the 2 files with ordered deletions (file 2 and file 3) run faster than file 4 because the deletions are always happening at either the head or the end of the list, meaning half the numbers can just be skipped automatically. File 4 has randomized deletions, so numbers can't be skipped as easily.

Compared to cursor lists and linked lists, skip lists perform faster because their complexity differences for whole file insertions and deletions are O(N log N) which grows at a much slower rate than the whole file insertions and deletions for the other lists, which are usually O(N^2). Skip list insertion may be slower than cursor list and linked list since it has to search for the position for the value to be inserted, but its deletion speed is much faster due to the same search used to insert values. Skip lists can ignore numbers within a certain threshold, but can't ignore the actual value, which makes it slower than queues or stacks, but faster that its list brethren. Stacks and queues perform the fastest for insertions and deletions because they just ignore the actual values for deletions. Cursor list is slower than the linked list because for both insertions and deletions, an iterator must be used to point to the position, whereas the linked list just inserts the value at one end. Additionally, the space allocation for the free list that keeps track of free and allocated nodes for the Cursor List is created in its constructor, rather than the space being dynamically allocated on the fly like the linked list, so that could take up time to create as well, especially if the space is large, like 250000. The computer has to keep track of both the available indexes on the “cursorSpace” and the user created list, continuously moving the cursor back and forth for insertions and deletions, thus making cursor list slower to run than the linked list. Additionally, the deletion method for cursor list requires a lot more steps to execute than the deletion for linked list. The compiler has to handle exceptions thrown by the brackets in the cursor list as well as the procedure of finding the node, deleting the node from the user list, and reinserting the node back into the “cursorSpace” list for reuse. In linked list, the deletion just descontructs the node and removes it from memory, which takes far fewer steps.