Janak Jobanputra

998189389

ECS 60 – UC Davis

**P1 Write-Up**

Times

*ADT 1 – Linked List*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | File1.dat | File2.dat | File3.dat | File4.dat |
| Run 1 | 0.111 | 59.206 | 0.084 | 35.041 |
| Run 2 | 0.110 | 59.271 | 0.084 | 36.165 |
| Run 3 | 0.109 | 59.222 | 0.083 | 35.407 |
| **Average** | **0.110** | **59.233** | **0.084** | **35.537** |

*ADT 2 – Cursor List*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | File1.dat | File2.dat | File3.dat | File4.dat |
| Run 1 | 0.094 | 302.660 | 0.110 | 151.069 |
| Run 2 | 0.084 | 303.001 | 0.101 | 172.774 |
| Run 3 | 0.085 | 302.560 | 0.101 | 173.896 |
| **Average** | **0.087** | **302.740** | **0.104** | **165.913** |

*ADT 3 – Stack Array*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | File1.dat | File2.dat | File3.dat | File4.dat |
| Run 1 | 0.077 | 0.070 | 0.068 | 0.070 |
| Run 2 | 0.074 | 0.070 | 0.068 | 0.070 |
| Run 3 | 0.075 | 0.070 | 0.068 | 0.070 |
| **Average** | **0.075** | **0.070** | **0.068** | **0.070** |

*ADT 4 – Stack List*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | File1.dat | File2.dat | File3.dat | File4.dat |
| Run 1 | 0.097 | 0.080 | 0.080 | 0.083 |
| Run 2 | 0.098 | 0.080 | 0.081 | 0.083 |
| Run 3 | 0.098 | 0.081 | 0.081 | 0.084 |
| **Average** | **0.098** | **0.080** | **0.081** | **0.083** |

*ADT 5 – Queue Array*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | File1.dat | File2.dat | File3.dat | File4.dat |
| Run 1 | 0.076 | 0.075 | 0.071 | 0.074 |
| Run 2 | 0.076 | 0.072 | 0.071 | 0.074 |
| Run 3 | 0.077 | 0.073 | 0.072 | 0.073 |
| **Average** | **0.076** | **0.073** | **0.071** | **0.074** |

*ADT 6 – Skip List*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | File1.dat | File2.dat | File3.dat | File4.dat |
| Run 1 | 0.331 | 0.235 | 0.286 | 0.461 |
| Run 2 | 0.337 | 0.233 | 0.280 | 0.484 |
| Run 3 | 0.330 | 0.241 | 0.297 | 0.487 |
| **Average** | **0.333** | **0.236** | **0.288** | **0.477** |

Results

*ADT 1 – Linked List*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Individual Insertion | Individual Deletion | Series of Insertion | Series of Deletion | Entire File |
| File1.dat | O(1) | - | O(n) | - | O(n) |
| File2.dat | O(1) | O(n) | O(n) | O(n2) | O(n2) |
| File3.dat | O(1) | O(1) | O(n) | O(n) | O(n) |
| File4.dat | O(1) | O(n) | O(n) | O(n2) | O(n2) |

*ADT 2 – Cursor List*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Individual Insertion | Individual Deletion | Series of Insertion | Series of Insertion | Entire File |
| File1.dat | O(1) | - | O(n) | - | O(n) |
| File2.dat | O(1) | O(n) | O(n) | O(n2) | O(n2) |
| File3.dat | O(1) | O(1) | O(n) | O(n) | O(n) |
| File4.dat | O(1) | O(n) | O(n) | O(n2) | O(n2) |

*ADT 3 – Stack Array*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Individual Insertion | Individual Deletion | Series of Insertion | Series of Insertion | Entire File |
| File1.dat | O(1) | - | O(n) | - | O(n) |
| File2.dat | O(1) | O(1) | O(n) | O(n2) | O(n2) |
| File3.dat | O(1) | O(1) | O(n) | O(n) | O(n) |
| File4.dat | O(1) | O(1) | O(n) | O(n2) | O(n2) |

*ADT 4 – Stack List*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Individual Insertion | Individual Deletion | Series of Insertion | Series of Insertion | Entire File |
| File1.dat | O(1) | - | O(n) | - | 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) |

*ADT 5 – Queue Array*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Individual Insertion | Individual Deletion | Series of Insertion | Series of Insertion | Entire File |
| File1.dat | O(1) | - | O(n) | - | 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) |

*ADT 6 – Skip List*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Individual Insertion | Individual Deletion | Series of Insertion | Series of Insertion | Entire File |
| File1.dat | O(log n) | - | O(n log n) | - | 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) |

The Linked List (LL) and Cursor List (CL) are identical in their time complexities, because they are very similarly implemented. The code inserted at the front for both of them, so both of them had insertions of O(1). The deletions had to find the element and then delete it, which, worst case scenario, would be at the end of the list every time. This was shown by File2.dat. However, since File3.dat was deleting in reverse order, it’s time complexity for CL and LL was O(1). But generally speaking a CL and/or LL deletion is on time complexity of O(n). File 4.dat was randomly inserting and deleting so it has the ‘truest’ of all the time complexities, which clearly shows that the time complexity for inserts should be O(1) and deletions should be O(n) for CLs and LLs. On average this takes about half the time as the complete worst-case scenario, as represented by File2.dat, which is the worst-case scenario, since it always deletes from the end.

The Stack Array (SA) and Stack List (SL) work very similarly too, since they both only push and pop and work exactly like any other stack. The time complexity for an insert (push) is an O(1) because it can just put it right at the top without searching, ordering, or anything. The deletions(pop) are also of O(1), again, because it just takes off the first element on the stack, which requires no searching or reordering. The SA is slightly faster than is the SL, most likely due to the extra data that has to be carried around when dealing with a list (ie pointers to next node) as opposed to an array implementation. The Queue Array (QA) is the same as a SA except for the deletions are occurring at the end instead of the front. Therefore an insert (enqueue) for a QA is of time complexity O(1), and a delete (dequeue) is also on O(1), since it doesn’t have to traverse the queue, but rather simply enter from the back and delete it.

The Skip List (SL) is a relatively efficient data structure since it searches, on average, half the size of the data structure, before any inserts or deletes. To insert, it traverses the levels of the skip list to see where it should belong, and for a large number of inserts, such as 250000 or greater, this is about half as many searches as any other ordered list giving it a time complexity of O(log n). The same applies to deletes, since, it has to re-traverse the list, but only on average half of how big the list is, which gives O(log n) again. Because of how well it is structured, it is able to handle all of the files in about the same time, taking the longest for File4.dat, since that is completely random, giving us the actual worst-case scenario.

The LL and CL are very similar in their time complexities because both of them are implemented as types of lists, but the CL simply takes the nodes and organizes them into an ‘linked array,’ storing the indexes of each node. This seems like it would be more efficient, but it creates trouble when it needs to traverse the whole array to find and delete (which is shown by its slow time in File4.dat and even slower time in File2.dat) and then after the deletion it needs to shift all the other indexes, which is quite tedious. The SL, on the other hand is much faster on average. This is because for an organized files like File1.dat (with just inserts) and File3.dat (deleting in the opposite order), the LL and CL are faster because it is already sorted and no comparisons need to be done to keep it in order. However, for unorganized files, like the unorganized File4.dat, the Skip List would be much faster since it can organize and access the data in, on average, log n time, which is faster than the average times for a CL or LL.

Between SL, SA, and QA, there is little difference in timings, because they are all simply just inserts and deletes, completely independent of how much data there is stored, inserting at the front, and deleting (for SA and SL) from the front or (for QA) from the back. However, as I mentioned above, the SL times are just slightly slower than that of SA and QA (which are about the same), because of the extra pointers that it has to create/manipulate/delete for a linked stack such as a SL.