Alex Elkman

4/9/2015

ECS 60: Sean Davis

Programming Assignment #1 Write-Up

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| File | ADT# | Time 1 | Time 2 | Time 3 | Average |
| File1 | 1 | 0.108406 | 0.109753 | 0.109856 | 0.109338 |
| File2 | 1 | 56.9734 | 56.9484 | 56.9753 | 56.9657 |
| File3 | 1 | 0.084574 | 0.083424 | 0.08328 | 0.083759333 |
| File4 | 1 | 31.8969 | 31.9224 | 31.9666 | 31.92863333 |
| File1 | 2 | 0.092294 | 0.083913 | 0.083923 | 0.08671 |
| File2 | 2 | 302.089 | 301.878 | 302.163 | 302.0433333 |
| File3 | 2 | 0.111962 | 0.100491 | 0.100555 | 0.104336 |
| File4 | 2 | 150.128 | 158.758 | 158.737 | 155.8743333 |
| File1 | 3 | 0.072851 | 0.072739 | 0.072739 | 0.072776333 |
| File2 | 3 | 0.06994 | 0.068416 | 0.068062 | 0.068806 |
| File3 | 3 | 0.067921 | 0.067772 | 0.067839 | 0.067844 |
| File4 | 3 | 0.070903 | 0.07101 | 0.071093 | 0.071002 |
| File1 | 4 | 0.096239 | 0.097263 | 0.097119 | 0.096873667 |
| File2 | 4 | 0.080379 | 0.078577 | 0.078449 | 0.079135 |
| File3 | 4 | 0.081074 | 0.079508 | 0.079431 | 0.080004333 |
| File4 | 4 | 0.081718 | 0.082855 | 0.082555 | 0.082376 |
| File1 | 5 | 0.076422 | 0.073305 | 0.073338 | 0.074355 |
| File2 | 5 | 0.070758 | 0.070867 | 0.070755 | 0.070793333 |
| File3 | 5 | 0.071627 | 0.071529 | 0.071512 | 0.071556 |
| File4 | 5 | 0.074867 | 0.073252 | 0.073031 | 0.073716667 |
| File1 | 6 | 0.325823 | 0.329811 | 0.328024 | 0.327886 |
| File2 | 6 | 0.227165 | 0.22616 | 0.234468 | 0.229264333 |
| File3 | 6 | 0.267528 | 0.265438 | 0.278026 | 0.270330667 |
| File4 | 6 | 0.414608 | 0.438984 | 0.429868 | 0.42782 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | File1 | File2 | File3 | File4 |
| ADT 1  Linked List | 1 Insert | O(1) | O(1) | O(1) | O(1) |
| 1 Delete |  | O(N) | O(1) | O(N) |
| Series Insert | O(N) | O(N) | O(N) | O(N) |
| Series Delete |  | O(N2) | O(N) | O(N2) |
| Whole File | O(N) | O(N2) | O(N) | O(N2) |
| ADT 2  Cursor List | 1 Insert | O(1) | O(1) | O(1) | O(1) |
| 1 Delete |  | O(N) | O(1) | O(N) |
| Series Insert | O(N) | O(N) | O(N) | O(N) |
| Series Delete |  | O(N2) | O(N) | O(N2) |
| Whole File | O(N) | O(N2) | O(N) | O(N2) |
| ADT 3  Stack Array | 1 Insert | O(1) | O(1) | O(1) | O(1) |
| 1 Delete |  | O(1) | O(1) | O(1) |
| Series Insert | O(N) | O(N) | O(N) | O(N) |
| Series Delete |  | O(N) | O(N) | O(N) |
| Whole File | O(N) | O(N) | O(N) | O(N) |
| ADT 4  Stack List | 1 Insert | O(1) | O(1) | O(1) | O(1) |
| 1 Delete |  | O(1) | O(1) | O(1) |
| Series Insert | O(N) | O(N) | O(N) | O(N) |
| Series Delete |  | O(N) | O(N) | O(N) |
| Whole File | O(N) | O(N) | O(N) | O(N) |
| ADT 5  Queue Array | 1 Insert | O(1) | O(1) | O(1) | O(1) |
| 1 Delete |  | O(1) | O(1) | O(1) |
| Series Insert | O(N) | O(N) | O(N) | O(N) |
| Series Delete |  | O(N) | O(N) | O(N) |
| Whole File | O(N) | O(N) | O(N) | O(N) |
| ADT 6  Skip List | 1 Insert | O(logN) | O(logN) | O(logN) | O(logN) |
| 1 Delete |  | O(logN) | O(logN) | O(logN) |
| Series Insert | O(NlogN) | O(NlogN) | O(NlogN) | O(NlogN) |
| Series Delete |  | O(NlogN) | O(NlogN) | O(NlogN) |
| Whole File | O(NlogN) | O(NlogN) | O(NlogN) | O(NlogN) |

Starting for the first two ADT, linked list and cursor list, we notice that the trends are very similar between the two. We inserted the data in the front of the list, and thusly the results for inserting are O(1). Deletion works similarly, except we must search for the item to delete, which in worst case will be at the end of the list. Thusly, it must be O(N) for both. Notice that the series deletion for both ADT 1 and 2 are quite slow in file 2 and 4 versus that of 3. It’s obviously because in 3 we start at the beginning for deletions, while in 2 and 4, we start at the end. In order to traverse a linked list to the end to delete an item, we must iterate through it each time. This causes huge delays in the processing time and it would be a terrible idea to pick such an ADT for doing this type of work. File 4 is on average quicker than file 2 because the items are randomly inserted, meaning we may get lucky and get rid of some items early on that aren’t near the end, which will drastically speed up the search time. We also see that the cursor list is on average slower than the linked list, this is because it needs to move the cursor for both insertions and deletions. This can be verified by stepping through in GDB. As a result, the average big O tends to be higher for the cursor list, increasing processing times a bit.

The stack array, stack list, and queue array all have quite similar code, and thusly produce very similar results. Since arrays have random access as opposed to linked lists, they are great for searching for items. We see that these three all outperform ADTs 1 and 2 especially noted on files 2 and 4. This is because of the point just mentioned, that when we need to find the last item, we can simply pop it off the top. Thusly, insertions and deletions are O(1) for both the stack array and stack list. For the stacks, we simply pop or push the item onto the last slot of the array, and for the queue we pop off the first item, and push on the last slot as well. Thus, the queue array also has O(1) for insertions and deletions. Thusly, all three have O(N) operations for series insertions/deletions, because they must perform O(1) \* N operations.

Skip list is the most unique of all the ADTs covered so far, utilizing searching for both insertions and deletions. It deals with searches by using a randomly assigned number of heads per entry that have progressively lower percentage chance of being assigned based on how many there are. Then, it starts its search at the highest order head items, and compares the item it’s searching for to the item there, and moves down to the next head based on this result. It’s actually somewhat similar to the algorithm a binary search tree uses, by whittling down its search in between certain parameters it is able to find it in a log(N) fashion. The fact that it needs to search for both deletions and insertions may seem to be a drawback at a first glance, however since they are only O(logN), the average of these two comes out to be somewhat close to the results of ADTs 3-5, and much better than ADTs 1 and 2 for files 2 and 4. So, while it is slower for all files than ADTs 3-5, it is still a very viable ADT to use depending on the format of the data you are using, and is great for finding data with random access very quickly.

Familiarizing myself with these ADTs was a very useful process, and learning how to use them, especially with part 3 of the project, really helped me to understand what is going on in their implementation. They each have good uses for different situations, depending on the type of data you are analyzing. For example, if we need to sort data, then probably using a linked list would be more useful, but if we need to do a lot of random deletions to data, then linked lists are not a great idea, and using the skip list or perhaps stacks/queues would be the way to go. In my opinion, the skip list is probably the most versatile of all the ADTs and I would probably use it if I didn’t know what type of data to expect in the results, or didn’t know what I was going to be doing with the data once I got it.