Vincent Yang

Write-up 1, Program 1

ECS 60, Sean Davis

4/10/2015

Time Results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| File | ADT# | Time#1 | Time#2 | Time#3 |
| File1 | 1 | 0.107086 | 0.108801 | 0.108422 |
| File2 | 1 | 57.5857 | 57.5804 | 57.5366 |
| File3 | 1 | 0.084307 | 0.083150 | 0.082953 |
| File4 | 1 | 32.8634 | 32.5877 | 32.631 |
| File1 | 2 | 0.092807 | 0.088998 | 0.089057 |
| File2 | 2 | 300.682 | 300.524 | 301.347 |
| File3 | 2 | 0.109212 | 0.105518 | 0.105533 |
| File4 | 2 | 149.947 | 155.752 | 170.386 |
| File1 | 3 | 0.068985 | 0.068959 | 0.069078 |
| File2 | 3 | 0.057618 | 0.057664 | 0.057597 |
| File3 | 3 | 0.057967 | 0.057752 | 0.057881 |
| File4 | 3 | 0.061039 | 0.060822 | 0.060731 |
| File1 | 4 | 0.083801 | 0.083849 | 0.083534 |
| File2 | 4 | 0.069156 | 0.07025 | 0.070618 |
| File3 | 4 | 0.069105 | 0.069781 | 0.069841 |
| File4 | 4 | 0.072085 | 0.072909 | 0.072931 |
| File1 | 5 | 0.074723 | 0.074396 | 0.074214 |
| File2 | 5 | 0.061192 | 0.060715 | 0.06091 |
| File3 | 5 | 0.061934 | 0.061755 | 0.061743 |
| File4 | 5 | 0.065572 | 0.065014 | 0.065093 |
| File1 | 6 | 0.304766 | 0.311911 | 0.309608 |
| File2 | 6 | 0.228541 | 0.229702 | 0.232232 |
| File3 | 6 | 0.265299 | 0.270016 | 0.274708 |
| File4 | 6 | 0.463072 | 0.490939 | 0.494689 |

Big O:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| File | ADT# | Insert | Delete | Entire series of Inserts | Series of Deletions | Entire File |
| 1(In) | 1(LinkedList) | O(1) | O(0) | N | 0 | N |
| 2(In/Del) | 1 | O(1) | O(n) | N | N2 | N2 |
| 3(In/Delrev) | 1 | O(1) | O(1) | N | N | N |
| 4(rand) | 1 | O(1) | O(n) | N | N2 | N2 |
| 1 | 2(CursorList) | O(1) | O(0) | N | 0 | N |
| 2 | 2 | O(1) | O(N) | N | N2 | N2 |
| 3 | 2 | O(1) | O(N) | N | N | N |
| 4 | 2 | O(1) | O(N) | N | N2 | N2 |
| 1 | 3(StackAr) | O(1) | O(0) | N | 0 | N |
| 2 | 3 | O(1) | O(1) | N | N | N |
| 3 | 3 | O(1) | O(1) | N | N | N |
| 4 | 3 | O(1) | O(1) | N | N | N |
| 1 | 4(StackLi) | O(1) | O(0) | N | 0 | N |
| 2 | 4 | O(1) | O(1) | N | N | N |
| 3 | 4 | O(1) | O(1) | N | N | N |
| 4 | 4 | O(1) | O(1) | N | N | N |
| 1 | 5(QueueAr) | O(1) | O(0) | N | 0 | N |
| 2 | 5 | O(1) | O(1) | N | N | N |
| 3 | 5 | O(1) | O(1) | N | N | N |
| 4 | 5 | O(1) | O(1) | N | N | N |
| 1 | 6(SkipList) | O(log n) | O(0) | N log n | 0 | N log n |
| 2 | 6 | O(log n) | O(log n) | N log n | N log n | N log n |
| 3 | 6 | O(log n) | O(log n) | N log n | N log n | N log n |
| 4 | 6 | O(log n) | O(log n) | N log n | N log n | N log n |

Note: Each paragraph has explanations for Big O for each file per data structure, then compared to others with similar structures / Big O’s. If an comparison is absent, then it is in the most similar data structure’s comparison. Ex: LinkedList vs. CursorList in CursorList’s paragraph; SkipList vs. CursorList/LinkedList in SkipList’s paragraph.

**Linked Lists** are typically unsorted because they can only be traversed through in order. Also, with their design it is easy to insert at the beginning of a linked list, especially since there is no need of pushing every other element back. Therefore, it takes constant (O(1)) time to insert at the beginning of the list. This makes linked lists very efficient for file 1, since it is O(1) inserts \* N inserts. For file 2, deletes are done for specific numbers, that all happen to come from the end of the list. Therefore, one must traverse through the entire list to reach it, and then delete it. Each delete takes O(n) time. File 3 has inserts then deletes from the front of the list, which is O(1) for each insert then O(1) for each delete. File 4 has randomized deletes and this takes on average n/2. This is displayed in the chart as its runtimes are about half of file 2’s runtimes. Finally, file 3 has a faster runtime than file 1 because deletes are faster than inserts, and both files have the same number of operations. Deletes are faster because inserts have to find space to allocate memory while insertions do not.

**CursorList** is a hybrid between an array and a linked list in that many nodes (spaces) are allocated, then filled. For inserts, this is faster (even though they have the same Big O) than linked list because it allocates a block of memory then fills it as opposed to finding spots of memory then filling those, then finding more spot. However, it is slower than deletes in Linked List because it must delete and reopen spaces in the CursorSpace list (a list that holds the available array indexes), as well as change data in the actual main CursorList. Also, these array indexes are spread all over in memory so it must grab different parts. Even though CursorList’s deletes have the same Big O as LinkedList, the differences are explained later in this paper (Question 1). This has the same runtime patterns (same structural reasoning) as LinkedList for files 1, 2, 3, and 4.

**StackAr** (Stacks) have O(1) inserts and deletes since they may disregard the value of the element they are pulling. Therefore, these are the fastest data structure. The array implementation of Stacks is faster than the LinkedList implementation because it does not have to allocate memory multiple times. These have about the same runtime for all files because inserts and deletes are all extremely fast (O(1)).

**StackLi** is the same data structure as above but slower because it must allocate space for each insert, which explains discrepancies between this and StackAr even though they have the same Big O.

**QueueAr** has O(1) inserts and deletes, but is a bit slower than StackAr because they must keep track of two indexes – the front and back.

The idea of a **Skip List** is to give a linked list log n search times. In the worst case, (with pseudo randomness) it takes n time to find an element (or appropriate spot for an element), at which point the program may insert or delete. Assuming that there is a static pattern for the ½ distribution from list to list (moving up the “tower”), the Big O for inserts and deletes is log n. Since it takes about the same time to find all of the data (front to back are all distributed in order, but search times for all are about log n), Skip List has about the same runtimes for all files. This is faster than LinkedList and CursorList for files 2 and 4 (faster deletes), but slower for files 1 and 3 (slower inserts). This is the fastest of the list ADT’s because it does not need to look through every element to find the proper spot for inserts/deletes.

**Question 1: Why is CursorList slower than the normal list?**

CursorList is slower than the normal list because it must modify the main data as well as CursorSpace. Every modification of CursorSpace changes two elements. Therefore, it has twice as many things to change (the main CursorList as well as CursorSpace) as opposed to LinkedList that only has to care about the main list’s nodes.

**Question 2: Why does CursorList take longer on the 2nd and 3rd runs of file 4?**

CursorList takes longer on the 2nd and 3rd runs of file 4 (the random inserts/deletes) because CursorSpace (the list holding the available array indexes) is global. When CursorSpace is created, each index’s data points to the next array index. Each node is about 32 bits while the cache is 64 bits and therefore every time the cache is filled, it can be used twice – once for the first half (first node) and again for the second half (second node). However, after the first run is completed the CursorSpace’s pointers are all over the place and then, when data is accessed from the CursorSpace, each data access must move 64 bits over to the cache, but only the first 32 are used. This makes every access of the CursorSpace after the first run a little bit slower, since more disc accesses are needed.