CSCI 115 Final Project -Luis Espinoza

Part 1:

Time Complexities for Add, Removed, Search/Update, Insert, and Append

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Add | Remove | Search/Update | Insert | Append |
| Linked List | O(1) | O(n) | O(n) | O(n) | O(n) |
| Double Linked List | O(1) | O(n) | O(n) | O(n) | O(n) |
| Skip List | O(1) | O(logn) | O(logn) | O(logn) | O(logn) |

It is with no surprise that Skip List is the most efficient for every operation in the table. Appending can be improved by tracking of the tail as well as the head in any of these structures allowing immediate access to the last element. In addition, a circular list of any of these but a single direction linked list would also cut travel in any direction by half.

The issue with any linked list is the cost of traversal time required to find any node for the removal, update, or insertion. All of which rely on some kind of search operation. Update is a simple assignment, which is trivial in cost and amounts to nothing more than a search in the first place.

Ultimately, a skip list is the clear winner in any setting where large amounts of nodes are present. The memory overhead and complexity of design may not be worth the cost if the data set isn’t sufficiently large. Skip lists really shine in their ability to traverse large sections of our list all at once, severely cutting down on travel cost; which; really, is the trouble with any operation in a linked list.

Part 2:

Time Complexities for Bubble Sort, Selection Sort, Insertion Sort, and Shell Sort

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Bubble Sort | Selection Sort | Insertion Sort | Shell Sort |
| Best Case | O(n) | O(n2) | O(n) | O(nlogn) |
| Worse Case | O(n2) | O(n2) | O(n2) | O(n2) |
| Average | O(n2) | O(n2) | O(n2) | O(n1.5-2) |
| Avg Run-time  N=1000 | 0.0397003 | 0.03962 | 0.0001869 | 0.0012465 |
| Avg Run-time N=10000 | 3.82438 | 3.77171 | 0.0017536 | 0.0191263 |

Methodology:

An array/vector was created with 10,000 random entries. This array was copied so that every algorithm would sort the **same** unsorted set separately. In total, there were 8 copies of the random unsorted set. Running time was recorded for every algorithm sorting the first 1000 entries on one copy and then the entire 10,000 entries on a separate copy. This was conducted a total of ten times running each algorithm in turn on an entirely new unordered set. After recording all runtimes for each algorithm, the averages were calculated for N=1000 and N=10000. This approach sought to control as many variables as possible to isolate the performances of the algorithm while giving a way of comparing directly each of their performances. I concede it is possible any oversight in this approach is a consequence of my limited experience.

Bubble Sort has the trivial feature that if an array is already sorted it only needs to make one pass after the initial pass to be sure it’s sorted. Not efficient in any real world applications where other more efficient options can be implemented. A fun toy algorithm nonetheless. It should be noted, findings for N=10,0000 suggest it is only marginally outperformed by Selection Sort. For such a low value, it might be tempting to say it is a good candidate for optimality but seeing this difference in performance for such a low value of N is actually very significant. These small differences add up at very large values of N.

Selection Sort has the advantage of requiring less memory than more complicated (but efficient) sorting algorithms because it sorts in place. However, it requires O(n2) in all cases. If the array is already sorted, O(n) is all that it takes since it will only compare each element once. Results suggest it is more performant than Bubble Sort which coincides with the conclusions of the community at large.

Insertion Sort is known to be efficient for partially sorted sets. The time complexity of the algorithm suggests it would perform similarly to Bubble Sort and Selection Sort but that is a quickly discarded notion when seeing the results of average running times. Insertion Sort sees a significant jump in performance. This is not surprising for the uninitiated and agrees with known results.

Shell Sort seeks to leverage the feature of Insertion Sort’s best case scenario, the partially sorted set. It sorts in large gaps and progressively sorting by smaller and smaller gaps until it decomposes to an insertion sort. This approach is inherently unstable but if that doesn’t matter, this approach is very efficient. It should be noted that the selection of gap sequence can have a significant effect on the performance. The results gathered here suggest it is outperformed by Insertion Sort. This is likely, because the data set is very small and/or the gap sequence is suboptimal. It is likely provable, although very difficult, leveraging some number theory, why Shell Sort performs so poorly when the size of the set and the gap sequence share some deep number theoretic property creating some inefficiencies.

Part 3:

Time Complexities for Insertion, Search, and Deletion

Binary Search Trees (BST) have an average case of O(logn) when the tree exhibits some level of balancing. Worse case for the BST is if data members are inserted in sorted order. Operations perform in linear time because the tree becomes nothing more than a linked list. Essentially a tree with only right branches. As a consequence, the height is equal to N.

AVL Trees always perform in O(logn) because their self-balancing nature aims to fix the glaring issue in BSTs; that order of insertion can severely unbalance the tree and create issues in performance later.

A balanced tree can preserve the O(logn) complexity by keeping the height of the tree within a reasonable boundary. A data set of n=63 with a height of 50 means the search will traverse down 50 levels before it finds the node in the worse case. However, a height of 6 significantly cuts down traversal. It’s not hard to see where the logarithm comes from. Every level of height doubles the amount of nodes in the previous level. This has the affect of dividing the data by two every descent which speeds up searches. If the tree is balanced, the upper bound of possible nodes will only be n <= 2m or written another way log(n) <= m where m is the height. Log(63) < 6 [63 <= 26] in other words the number of nodes is bounded by the next power of two for optimal performance. If it exceeds this bound, performance begins to see losses.

Part 4: