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.

Challenges:

Skip List presented the same issues of complexity when first presented in the semester. It was difficult to hold the complexity of design in mind when dealing with such level of abstraction. Ultimately the approach that made the most sense was to create a vector in the Skip List class that held pointers to Linked Lists that served to represent each level of the Skip List. This powerful abstraction made it easier to think about such a complex structure. It was not a lightweight solution but drastically improved readability.

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.

Challenges:

Refactoring the sorting algorithms was not complicated and leveraged many helper functions to parse the data for direct comparison. The change from integers to a more complex data structure created requirements unwrapping the data for the algorithm.

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.

Challenges:

It was not a lightweight solution to use inheritance to maximize code reuse, but it served as a meaningful abstraction. AVL inheriting from BST and overriding only the operations that diverged from the BST was very useful way to think about the problem. For example, BST and AVL insertion were essentially the same operation. Where they differ is that AVL then checks balance of the tree and applies necessary rotations. It does this recursively and results in a balanced tree. If the only difference is that one balances the tree after insertion and the other doesn’t, it makes the most sense to diverge definitions of insertion for the two classes at exactly this moment.

Conversion from a classless implementation of these trees to one where a class gives it a meaningful name and abstraction was a challenge when dealing with recursion but ultimately the only factor was time. Time to think about it. Time to implement it. Time to fix inevitable bugs.

Part 4:

Graphs

Graphs come up in mathematics to graphically represent relationships. They powerfully abstract many different systems. The components can be listed as Vertices, and Edges. Vertices can contain values. Edges can be weighted to represent different relationships. Orientation can be represented within Edges to represented directed or undirected relationships. Undirected naturally representing symmetric relationships.

BFS vs DFS

BFS:

Time taken for BFS A to J : 5.5e-05 seconds

A==>B==>I==>J

Time taken for BFS A to D : 1.6e-05 seconds

A==>B==>D

Time taken for BFS B to A : 6.3e-05 seconds

B==>I==>J==>A

Time taken for BFS C to B : 6.8e-05 seconds

C==>E==>F==>G==>H==>I==>J==>A==>B

DFS:

Time taken for DFS A to J : 2.6e-05 seconds

A==>C==>E==>G==>H==>I==>J

Time taken for DFS A to D : 2.2e-05 seconds

A==>B==>D

Time taken for DFS B to A : 1.4e-05 seconds

B==>I==>J==>A

Time taken for DFS C to B : 2.1e-05 seconds

C==>E==>G==>H==>I==>J==>A==>B

While the objective of each version of the search is different. It was surprising to see their performance differ so much. It also wasn’t overwhelmingly in one direction. DFS won in some cases while BFS won in other cases. Admittedly, I’m not sure I understand these algorithms deeply enough to understand why.

Dijkstra’s algorithm was run from a starting vector to all vectors calculating the shortest paths. Run time was calculated after calculating all paths. The average run time per path can be taken as total/10. Average run time for a path from A was 4.05x10-5 which lands between the BFS and DFS. Admittedly, a student’s implementation for Dijkstra’s is very inefficient and a better engineer (or more experienced student) could optimize the algorithm instead of spending days fixing bugs in the code. The results are encouraging once the bugs were dealt with the algorithm could calculate all shortest paths as designed.

Challenges:

Dijkstra’s Algorithm. Simple to some. However, it is incredibly easy to lose track of what’s happening in the code. Use of named lambdas helped provide a way to separate the logic into meaningful names. It also had the effect of documenting, without comments, the purpose of particular blocks code. It did not necessarily make the problem easier to code but it did make the code easier to work on by abstracting entire blocks of code so they can be disregarded while working on smaller parts. I learned a lot about design, and given more time, I could probably design a much more readable version in the future.

Final Thoughts:

There is an obsession, in a subsample of Software Engineers, with aggressive optimization. Often the more optimal code gets the less readable and, usually, less maintainable it becomes. Compilers have gotten so incredible at optimizing code that often the changes one could make to code for optimization barely make any difference to final result. For example we can compare the following code snippet:



It’s clear the first block is more concise but it’s very unclear without looking at the larger context what is exactly is happening. What is k for? What is func and what is exactly is it being fed 42 for?

The second block names every bit of its context. It becomes clear at a glance what our if block is supposed to do without looking at the larger context. The mental strain of keeping all these contexts in your head are not required because they’ve been abstracted away. They can be unpacked at any moment by looking at their assignment statements.

Admittedly, this example is trivial because the cost of assignment is negligible, but a compiler will optimize away this cost, regardless. The point is that most decisions a compiler makes will do a lot more for optimization than an engineer would. The difference between a readable insertion sort vs a much less readable insertion sort is almost practically non-existent once a compiler processes them. Of course, a compiler will not replace a bubble sort with a quick sort but those are the decisions that an engineer can make and they will actually matter. What doesn’t matter to the compiler is when an engineer decides to “optimize” his quick sort by making it as concise and terse as possible when he doesn’t realize a much more readable version will still be optimal because the compiler is better at it than he/she is.

The biggest lesson in any project of this scale or larger is the realization that maintaining and reading one’s own code base (or anyone’s for that matter) is much easier when steps are taken to make it readable and understandable. The cost in mental energy just to maintain complex abstractions in one’s head is not worth perceived gains in performance. Debugging became much easier once pieces of code were named meaningfully and organized into abstractions that could be disregarded when necessary. It also helped isolate problems when necessary. I’m a much better problem solver not because of the practice this project gave me but more because it has informed my decision making when designing a solution.