Marko Neskovic

CMSC858D HW1 Writeup

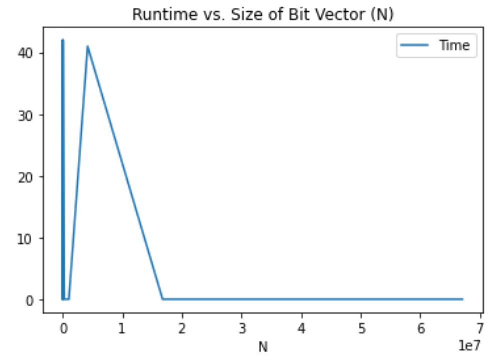
Github Link: https://github.com/mneskovic/CMSC858DHW1

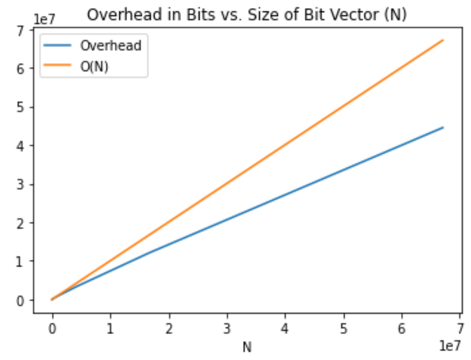
Rank Support

Description: In order to create the rank support structure itself, I first take a SDSL (Succinct Data Structure Library) bit vector, determine the block/superblock sizes using the log(n), initialize my Rs and Rb vectors using the number of superblocks/blocks that divide the bitvector (I also pad the bit vector so it divides evenly), and add the vectors as members. Then, I fill Rs and Rb by going through the original bit vector to calculate ranks for the blocks (using the formulas from class). In order to actually get the rank for a given i, I use Rs and Rb along with the in-block rank which I calculate by getting the block using SDSL's built-in get_int function and running popcount. Finally, to save and load, I save the Rs and Rb vectors by saving the size of each followed by each element and save the bit vector using SDSL's internal serialize function, and load by reading the vectors based on the size of each element and using the SDSL load function for the bit vector.

Hardest Part: The most difficult part of implementing the rank_support was understanding the underlying c++ functionalities necessary. First, installing sdsl and learning how the bitvector and its underlying functions worked took a good while. Then, when implementing the rank itself, doing an O(1) in-block rank query was also difficult because I initially couldn't wrap my head around how popcount was O(1), and also had difficulties getting the block itself using sdsl's functions. Finally, while I've coded in c++ before, the memory management aspect is always something I need to get used to so I ran into tons of seg faults and similar issues while doing my implementation.

Plots:





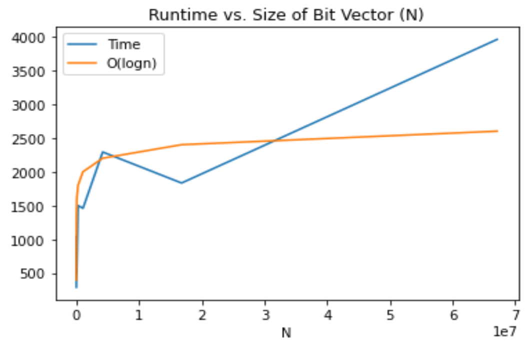
As we can see from the plots, our rank data structure matches the theoretical bounds. On the left, we see that increasing the size of the bit vector (even close to 70 million bits) resulted in roughly the same runtime of 0-40 nanoseconds. On the right, we can see that while the overhead increased with the bit vector size, it did so in o(n). In other words, as N quadrupled, the overhead consistently increased by less than 4x, and followed a trend away from *roughly* O(N).

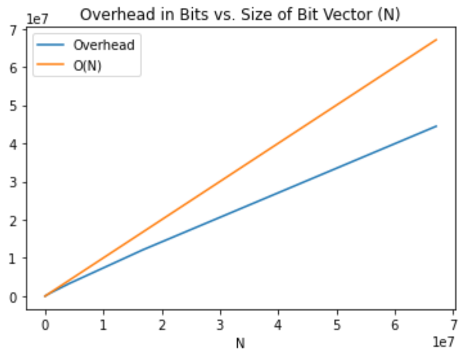
Select Support

Description: The implementation for my select data structure is very straightforward. I simply take the rank support for a given bit vector, and to find the select value I run binary search with the ranks as discussed in class. I simply divide the current subarray in half and if the rank of the middle element is higher, our i is in the left half, otherwise it's in the right half. The structure does not add any other elements, and saving/loading it is simply just saving and loading the rank data structure since that is all that is needed.

Hardest Part: The most difficult part of implementing this was accounting for the base/stopping cases. I originally thought that by slowly inching the left and right boundaries together, we would eventually stop when the two have crossed. However because of the way rank works, this is not always the case since multiple indices can have the same rank. The challenge was coming up with stop cases for the edge cases where we land on the rank we're looking for vs. when we've landed on the actual index we're looking for.

Plots:





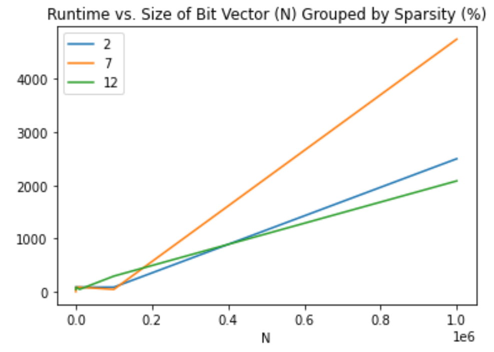
As we can see from our plots, our select data structure also matches the theoretical bounds. On the right, we see that the overhead is the same as with the rank support, which makes sense since we are only using the rank data structure as overhead. On the left, we see that as our N drastically increases, our runtime does not increase proportionally to n, but rather more closely follows the theoretical bound of *roughly* O(logn).

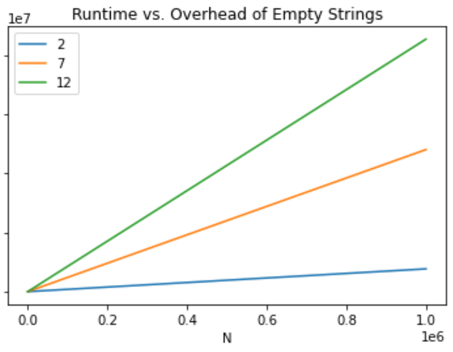
Sparse Array

Description: My implementation of the sparse array consists of a bit vector and string vector combination, in addition to an underlying rank support. To create the array, a bit vector is defined with the fixed size, and elements are appended by inserting them (push_back) into the string vector while also assigning that bit to a position in the bit vector. In order to get_at_rank, we simply look at the rth element in the list of values. For get_at_index, we use the rank at the index to get which element it is in the sequence of values. For num_elem_at, it is simply the rank at the index. Finally, size and num_elem are simply the sizes of the bit vector and string vector. To save and load, I save the string vector by writing the strings to a file line by line and the bit vector using serialize, and then I load the vectors by loading the strings line by line and using the bit vector load. I also added a function called add_rank_support which needs to be called before doing any operations. This is done to avoid building the rank support during the operations themselves or after every change to the bitvector, which changes the complexity.

Hardest Part: The hardest part of my implementation was coming up with a good way to ensure that the complexity of the lookup operations remains O(1). Since they used all used rank support it was natural they should be O(1), however the construction of the rank support itself is not O(1), and depends on the underlying bit vector which changes. Initially, I had the rank support constructed as needed (when a lookup function is called), however this changed the complexity of the functions to O(n). I eventually settled on just creating a new void function which does this when the user needs it, but it requires documentation and is a little tedious. This is still far better than doing it at the function call or every time the bit vector changes (since if we append n 1's, the complexity becomes O(n^2) for that operation.

Plots:





As we can see in our plots, while the runtime of our functions seems to increase a bit over time (as a result of outliers), it is generally not affected by the size of our bit vector nor the sparsity and runs in O(1). On the right we can see that the size of the bit_vector and its sparsity seemed to directly affect our overhead. This makes sense because our overhead is determined by the number of empty strings (0's in bit vector) that we have to store. The more 0's in the bit vector, the more we need to store.