CMSC 858D Spring 2022 HW 1

MG Hirsch

See instructions here: https://rob-p.github.io/CMSC858D_S22/assignments/01_assignment_01

Part 1: Rank

Description

The rank support data structure stores an int_vector of blocks and an int_vector of superblocks that are populated during a single pass over the data. To calculate rank, it sums the values of the related block and superblock along with the number of extra bits.

Experiments

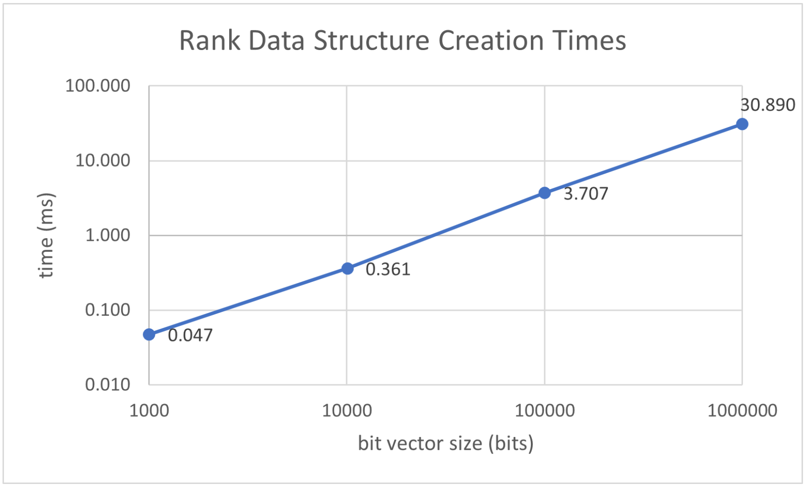
Experimental script can be found in `part1/part1.cpp`.

For varying sizes of the bit vector, 1000, 10,000, 100,000, and 1,000,000 bits, I:

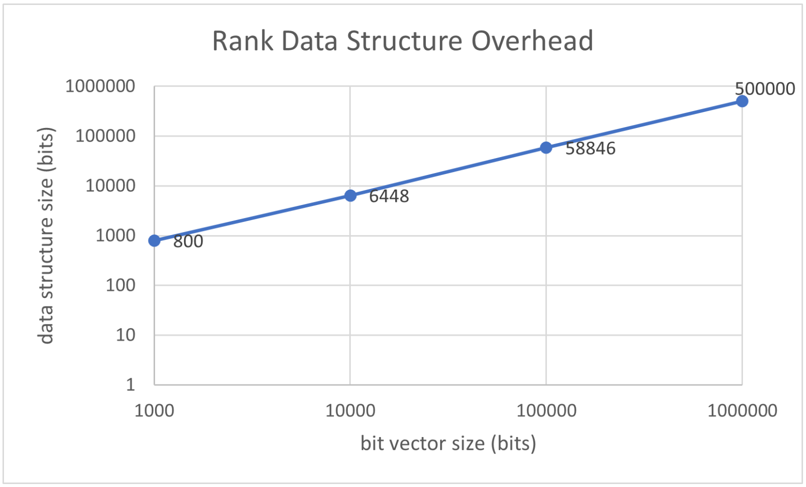
1. Timed the creation of the rank support structure
2. Calculated the overhead of the structure
3. Timed calling `rank1` on a random bit 16 times

Results

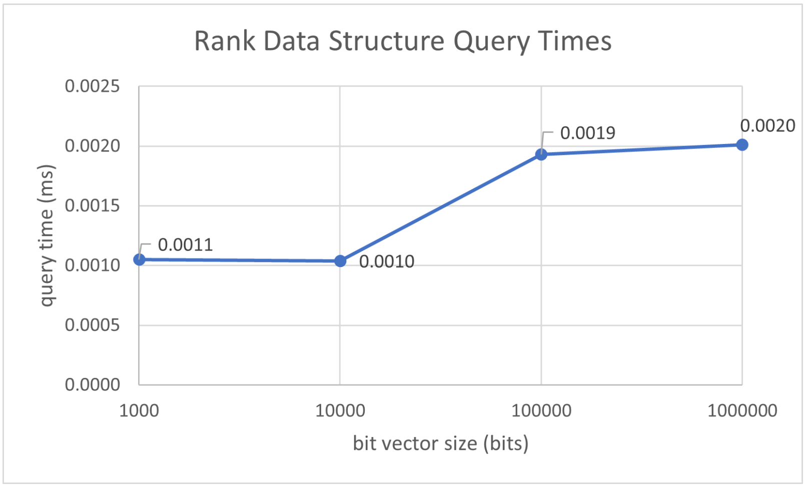
Results are averaged over 10 runs.



The above figure shows how long creation for the rank data structure took for each bit vector size. There is an apparent linear relationship between the size of the bit vector and the creation time (note the log scale on both axis). This is consistent with the single pass over the bit vector for creation.



The above figure shows the overhead size of the structure with respect to the bit vector size. There is a linear relationship between the two (note the log scale on both axes). This is consistent between the theoretical $o(n)$ relationship.



The above figure shows the time taken to perform 16 rank operations for random indices with respect to the bit vector size. There is a near constant relationship, with a jump between 10,000 and 100,000 bits.

Hardest part

What I found the most difficult part of this task was getting all the index calculations correct. I began trying to implement it exactly as was presented in class and ran across problems when blocks and superblocks did not align. After simplifying the implementation to have the length of the superblocks equal to the square of the length of the blocks, things worked out much easier. Then determining the relevant block and superblock to increment as well as rules for when to copy or keep 0 was straightforward (after staring at it for a while).

Part 2: Select

Description

The select support data structure stores a rank support data structure. To answer rank queries, it performs a binary search over the bit vector calculating the ranks of the indices.

Experiments

Experimental script can be found in `part2.cpp`.

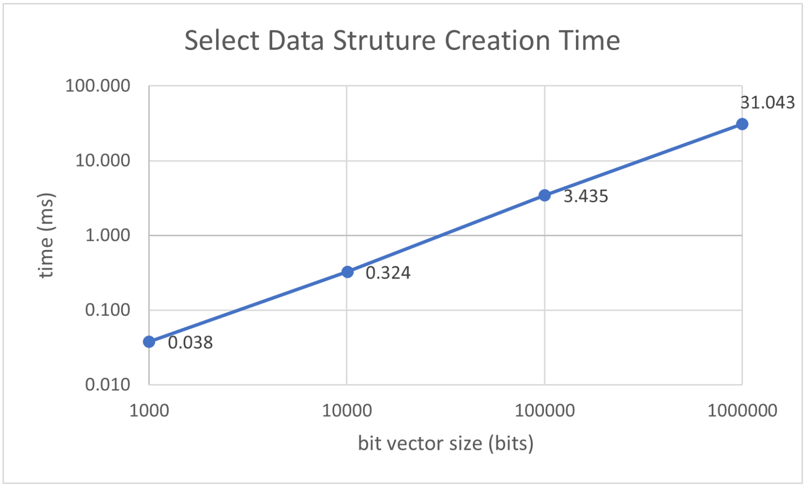
For varying sizes of the bit vector, 1000, 10,000, 100,000, and 1,000,000 bits, I:

1. Timed the creations of the select support structure

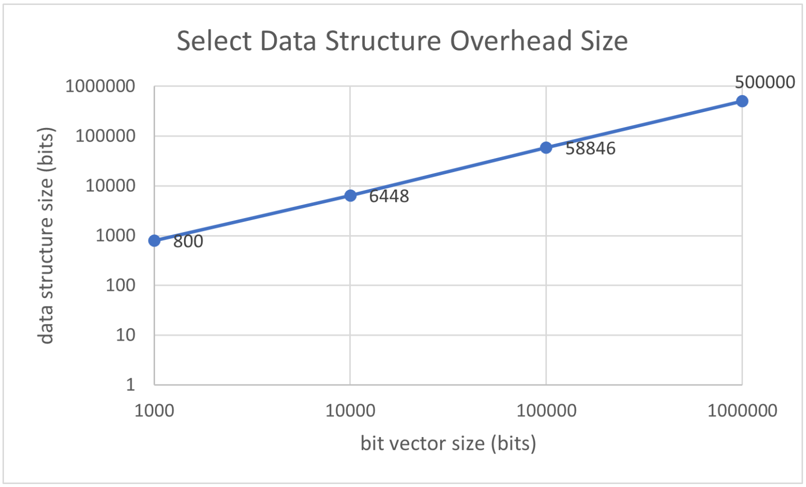
2. Calculated the overhead of the structure
3. Timed calling `select1` on a random bit 16 times

Results

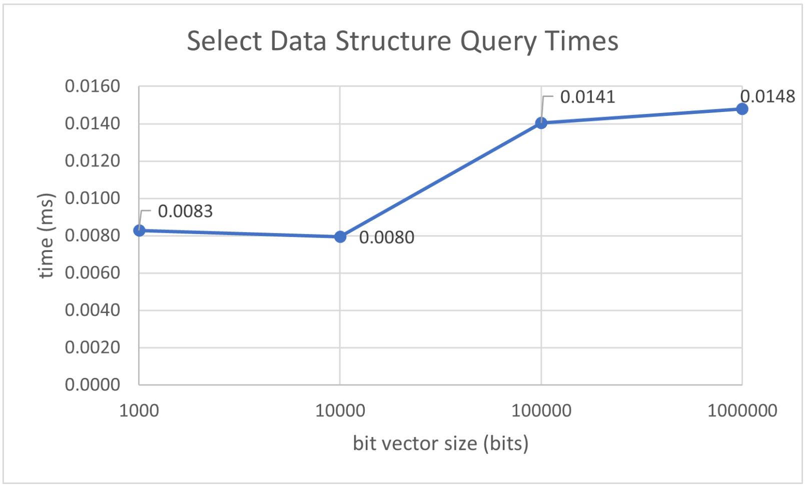
Results are averaged over 10 runs.



The above figure shows the time for creation of the select data structure with respect to the bit vector size. The creation time scales approximately linearly with respect to the bit vector size (note the log scale on both axis). This is consistent with the single pass over the bit vector elements.



The above figure shows the overhead size of the select data structure with respect to the bit vector size. The overhead scales linearly with respect to the bit vector size (note the log scale on both axis). This is consistent with the $o(n)$ theoretical bound.



The above figure shows the time taken to perform 16 select operations on random indices. It shows a near constant relationship between bit vector size and query time, with a jump between 10,000 and 100,000 bits.

Hardest part

I found this part very straight forward. The hardest part was probably implementing the binary search, just because it took a few more lines of code.

Part 3: Sparse Vector

Description

The SparseVector class stores a bit vector, a related rank support data structure, and a vector of elements. It uses the rank support data structure to answer queries about the elements' locations in the underlying bit vector.

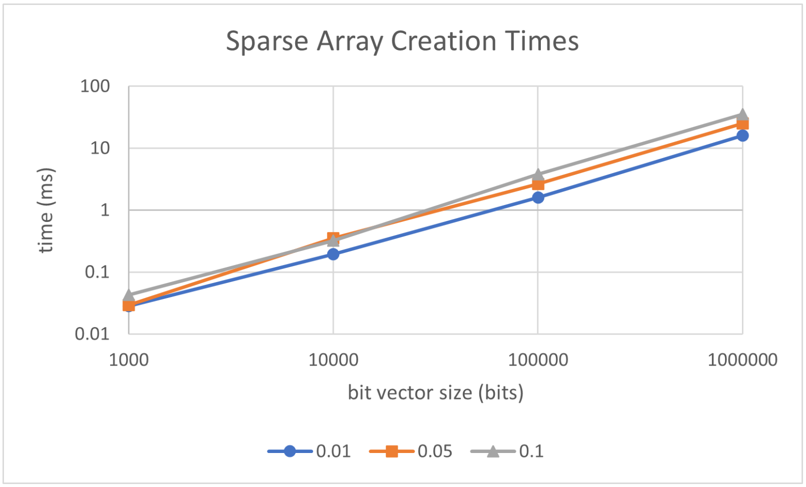
Experiments

I time the creation and filling of the sparse array.

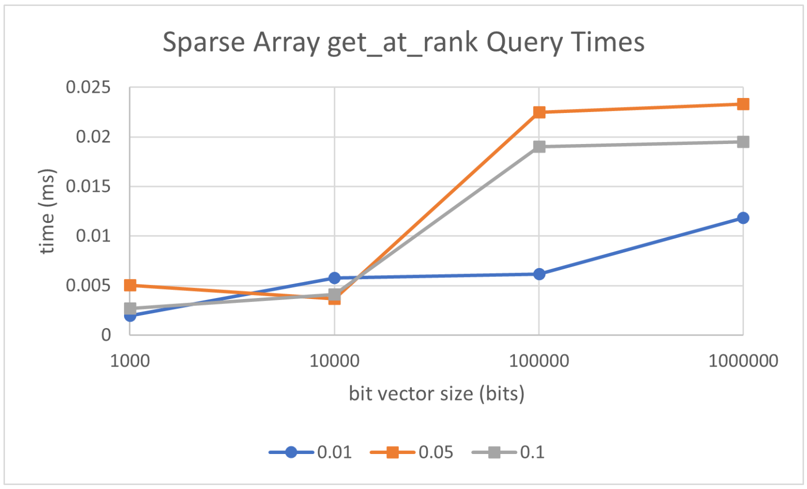
For each of the methods `get_at_rank`, `get_at_index`, and `num_elem_at`, I timed calling the method 100 times with a random index (indicing the bit vector for `get_at_index` and `num_elem_at` and indicing the elements for `get_at_rank`.)

Results

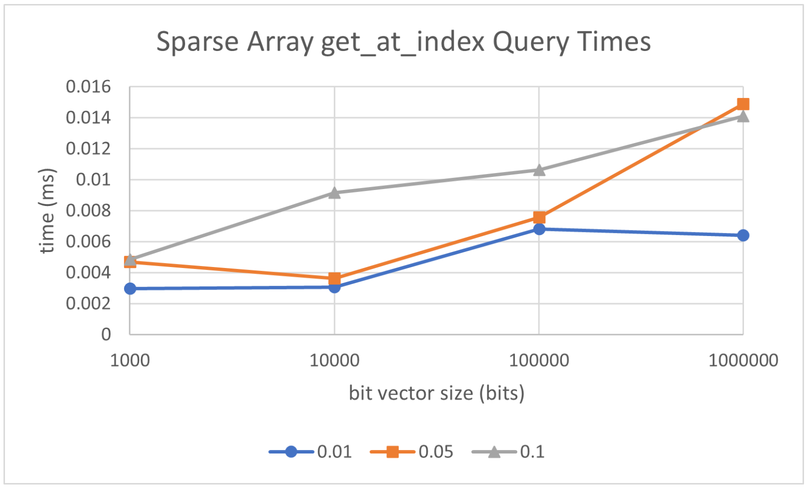
All results are averaged over 10 runs.



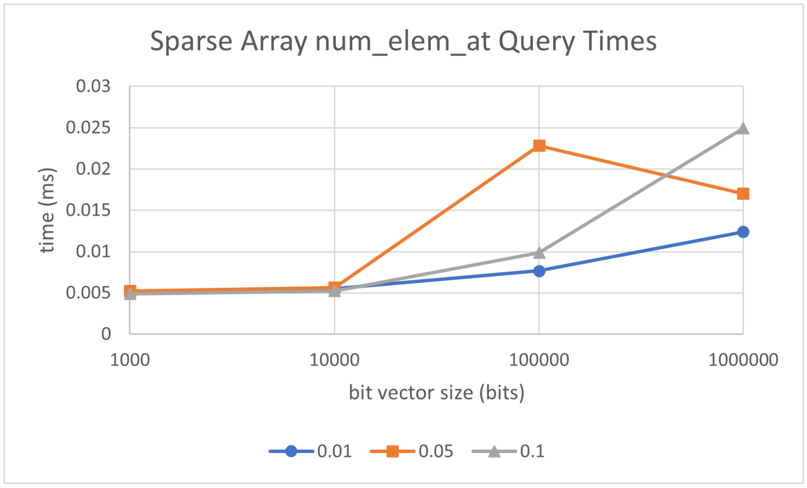
The above figure plots the sparse array creation times against the bit vector size (both axes in log scale) with different levels of sparsity. The creation time included the creation of the bit vector, filling in the elements, and creating the rank support data structure. At all sparsity levels, there appears to be an approximate linear relationship.



The above figure shows the time taken to run the `get_at_rank` query for a random index 100 times with different sparsity levels with respect to the bit vector size.



The figure above shows the time taken to run the `get_at_index` query for a random index 100 times with different sparsity levels with respect to the bit vector size.



The figure above shows the time taken to run the `num_elem_at` query for a random index 100 times with different sparsity levels with respect to the bit vector size.

Hardest part

The hardest part of this part was deciding where to build the rank support data structure. At first I considered adding it at te end of the eppend method or at the beginning of the method calls that used it, but instead I decided to make it it's own method and make the user call it. That way it's only being built as often as it needs to be.