Implementing some succinct primitives

Kazi Tasnim Zinat, 116760904

November 7, 2019

T he objective of this assignment was to implement a succinct data structure to perform rank and select operations on. The language I used for this assignment is C++. While existing libraries like SDSL provide the underlying bit vector support, implementing the data structure from scratch made me understand the data structure properly and also drove me through a steep learning curve. For storing the bit-vector, I used the vector<bool>class of C++, which is unique in its type as it takes up only 1 bit of space for saving each value.

Task 1 — bit-vector rank

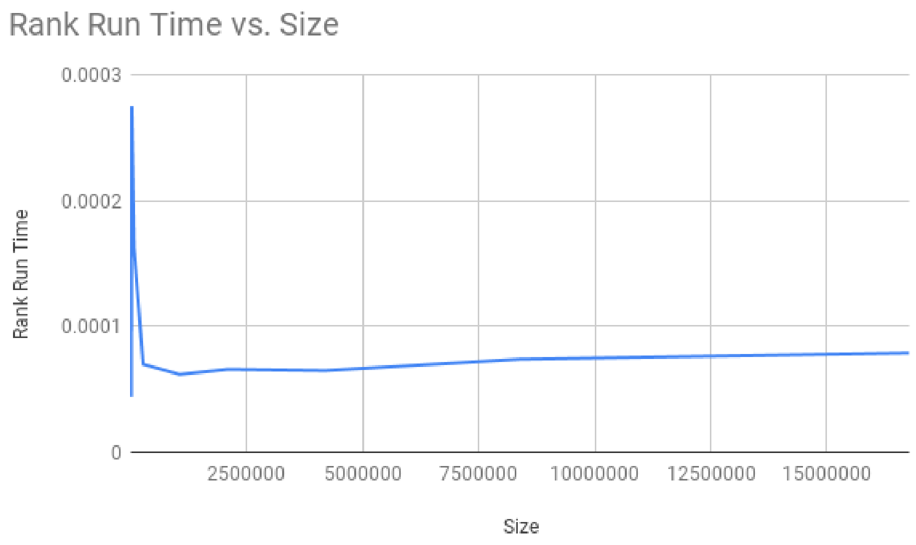
Implementation

For implementing the bit-vector rank I created a class Rank-support that takes a bit-vector B as parameter and then constructs block, super-block and bit pattern vectors over it. The rank1 function calculates the number of 1's upto and including given index by calculating the corresponding block, super-block indices. The rank0 function returns the number of zeros and overhead calculates the size of the structure in bits.

Most difficult part

As the bit-vector rank was the first task to be implemented, I initially ran into a lot of errors while implementing this one, like determining what data structure to use to store the blocks and super blocks, Whether to use ceiling or floor to calculate block and super-block sizes and so on. Implementing the R/_p that keeps track of the cumulative sum for bit pattern within the bloc was a specially challenging task. Also studying about C++ libraries and papers on how to actually make this rank operation constant time was very time consuming. But I had the most difficulty in performing the boundary checks on this code. Specially it gets tricky when the superblock size is not perfectly divisible by the block size.

Plots



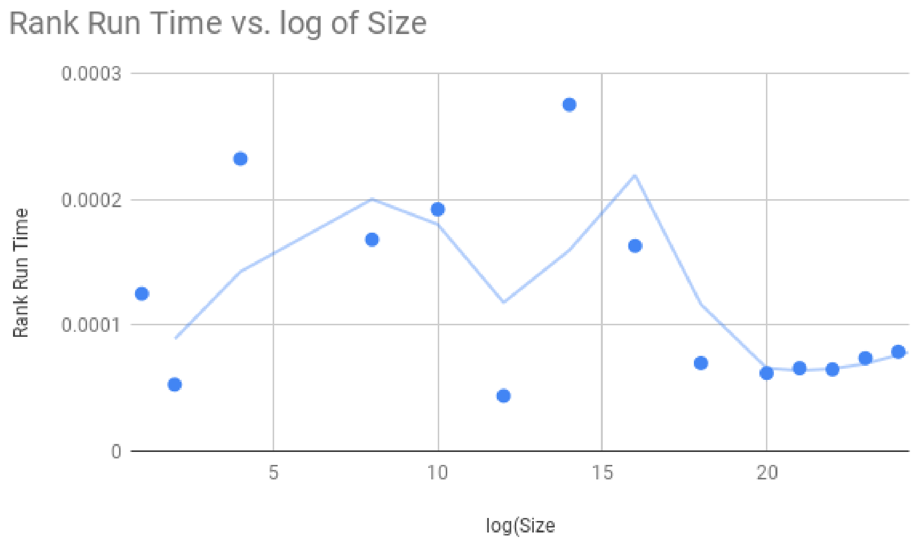


Figure 1: Run time for Rank operation vs the log value of input size. In the first figure the after the initial spike the result run time stabilizes. Then it is almost constant, within 0.1 miliseconds. We can see that no deterministic answer can be reached from the second plot as the run time varies un-correlated to size, that might be due to the processor being busy in other tasks. But we can say it shows a somewhat constant result if we observe the last few values trend line.

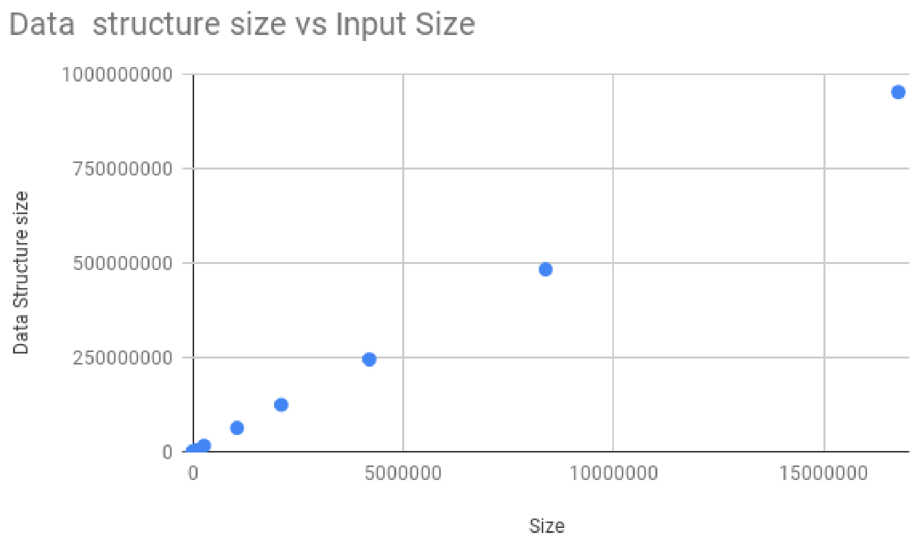


Figure 2: Input size vs the data structure size (bit-vector inclusive). As we have included the bit vector here, the data structure size seems to be increasing in $O(Z)$

Task 2 — bit-vector select

Implementation

For implementing the bit-vector select I created a class Select-support that takes a rank-support class as parameter and uses its block, super-block and bit pattern vectors structure to calculate select. The select1 function returns the index of the i^th 1 in the bit-vector where i is the parameter. The rank0 function returns the index of i^th zeros and overhead returns the size of the underlying rank structure in bits.

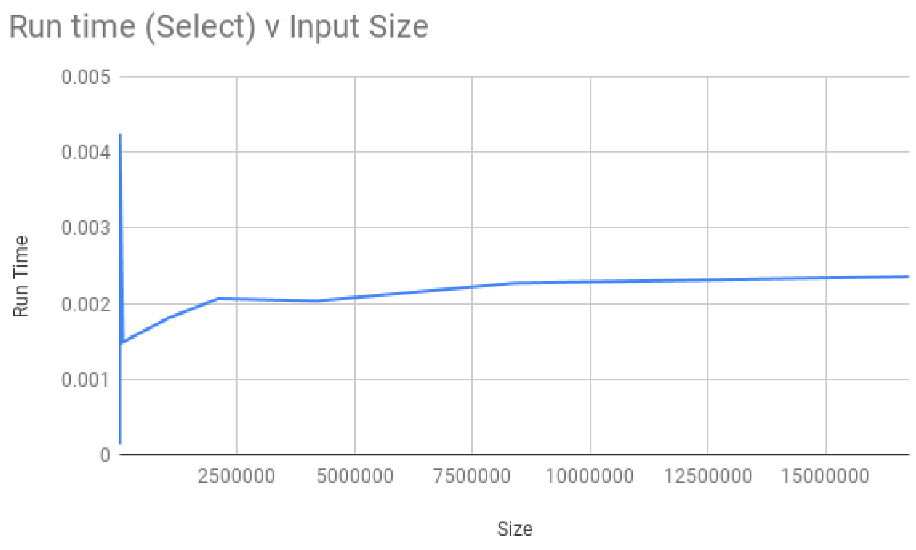
Most difficult part

I think the bit-select was the easiest to implement among the three given tasks. Once the binary search is done for select1 it can be implemented in a similar way for $select0$.

Plots

Plotting overhead for select

Plotting overhead for select is not necessary as the underlying data structure is same as rank. So the values are same as rank for all input sizes.



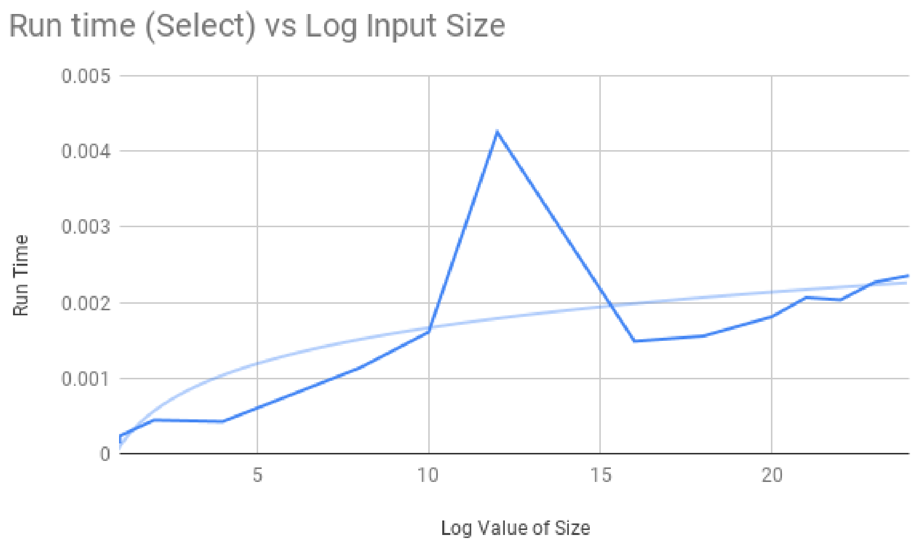


Figure 3: Run time for Select operation vs the input size. We can see that this plot follows a logarithmic trend after wobbling for some initial values. The picture with logarithmic sizes shows a linear relationship. This is expected for a log time select operation

Task 3 — wavelet tree construction and query

Implementation

The wavelet tree is constructed in a recursive way for a given string. First, the unique characters in the strings are extracted and they create the set of alphabet for the current node. Then a separate bit vector is created to depict whether the character at some position is lexicographically lesser than the middle character calculated over the alphabet or not. Then rank and select support are built over the bit vector corresponding to the node. Then the left and right children of the tree are constructed over the subsets of alphabet that mapped to zero or one respectively. The paper titled wavelet tree for competitive programming [1] really helped to me understand the basics of implementing a a wavelet tree.

Most difficult part

For wavelet tree, we had to write two constructors- One that takes only a string and builds a wavelet tree over it, and for the other we needed to read a file where the tree had been already saved and reconstruct the tree from reading the file. As C++ does not provide any default serialization, the dumping to and loading the tree from file had been the most difficult part with wavelet tree implementation.

Plots

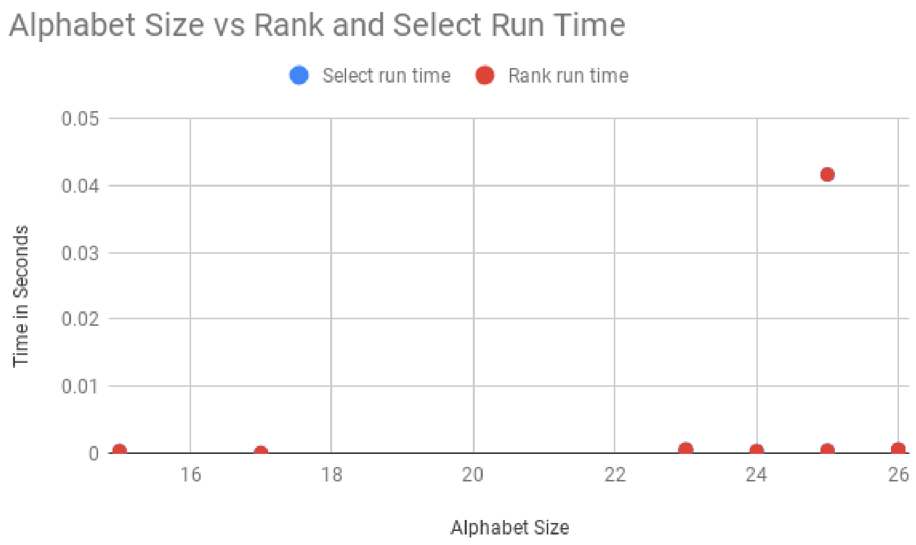
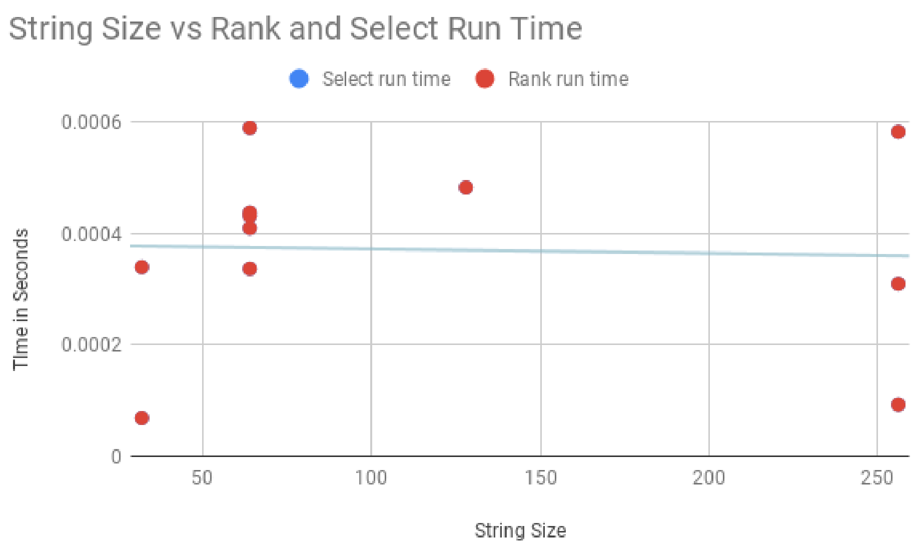


Figure 4: One in inexplicable things with my assignment is that in all cases of the Wavelet Tree rank and select both take the exact same time. This can be clearly seen from the first figure where the blue mark for select is hidden by the red mark of Rank for every value. Also one of the test cases with alphabet size 25 and string size 128 took 40 milseconds to run which makes it very difficult to observe the changes in other values. So that value will be omitted from the following figures.



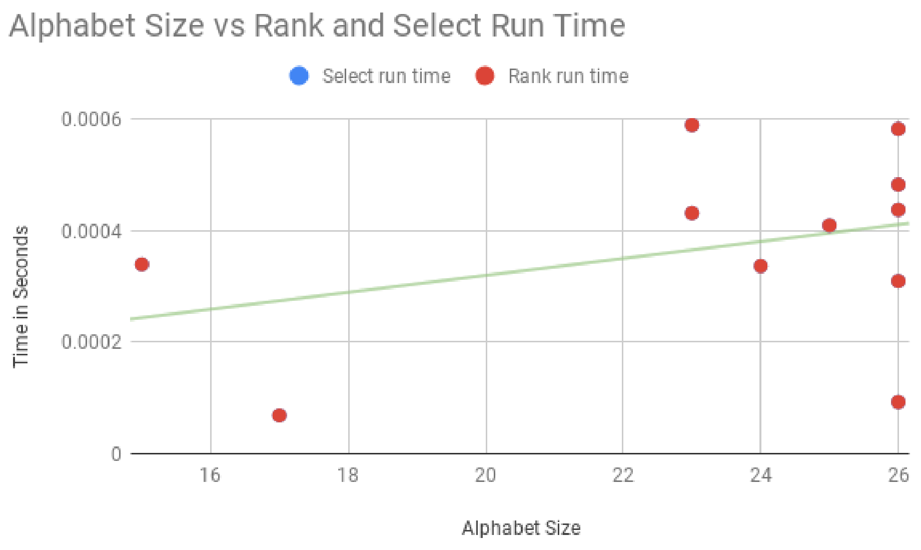


Figure 5: The Rank and Select operations for various String sizes provided non-deterministic result- as we can see from the first Figure that for the same string size the run time varies over the range. Also if we look at the second figure, we will see that any specific relation between alphabet size and time can not be determined as well

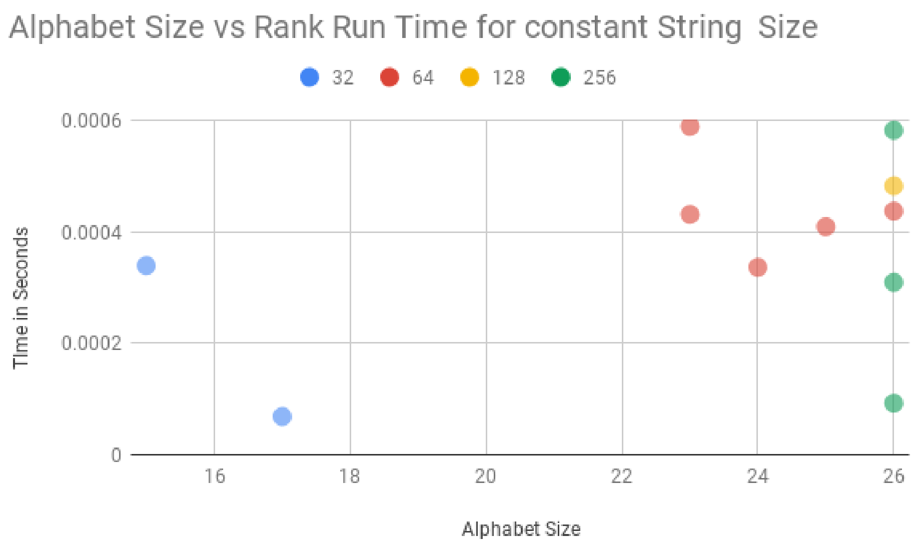


Figure 6: Here each color represents a fixed string Size whereas we plotted Alphabet horizontally and Time vertically. We can see even for a fixed length string size we can not really say whether increasing or decreasing alphabet size has any effects ir not. Maybe using more data points could have been useful is this case.

Take Away

Implementing the succinct data structure has been both challenging and time consuming. This helped me truly understand the succinct data structures. But I guess I learned a lot while doing this. Specially this has been helpful to rush up my C++ knowledge. Unfortunately, I think there are many corner cases that remain unhandled and the results were not impressive as well. Maybe that's why using a library for this kind of operations is a better idea.

References

[1] Robinson CASTRO et al. “Wavelet Trees for Competitive Programming”. In: *Olympiads in Informatics* 10 (2016), pp. 19–37.