Fausitne Yiu

ECS 36C: Report 3

bt_succinct_dec.cc

In this program, I decided to read in the .dat file through fstream and store the first line into string variable structure, and the second line into string variable data. I checked for errors, like empty files, invalid inputs, etc. to return error messages through std::cerr. Since I chose to use string as my data type, there are a few for loops to loop through individual strings to check for its value. I would say if I were to do it again, I would have used sstream instead, for simplicity reasons. However, I wrote the error messages portion first, and then realize I needed to modify my checking of values even more in the EncodeSuccinctBinaryTree function.

For my EncodeSuccinctBinaryTree function, it takes in parameters unique_ptr Node called n, and my strings structure and data. With these parameters/inputs into the function, I was able to check whether structure has a 1 or 0 as the next line. If it has a 1, append the next data value into n and then erase that data value from data. Using my new n (unique_ptr Node), structure, and data, after seeing a 1 in structure, I feed the values back into my EncodeSuccinctBinaryTree, but this time either creating my root, as it is the first value into my n (unique_ptr Node) or adding the data value to the left child. If the next value of structure has a 0, simply erase it from structure and the function will recursively record the left child as nil, then if the left child is already nil, the right child will be recorded as nil and loops back up the tree to its "grandparents" or two parents up. This continues over and over again, with finally, we have a tree data structure, where values are stored either in the left of right child, based off of its alignment with 1.

bst.h

To verify my implementation, I modified the simple tester by changing the vector keys input into different types of tree structures. For example, 74 93, has a Null left child from the root, and vice versa, 51 43, has Null right child from the root. This allows the ceil, floor, and kth_small functions to test on two extremes of tree data structure. In addition, I tried different combinations of the vector keys input by deleting some children or child (either left or right) from their parent or adding children and child (either left or right) from their parent to check for any error in implementation. All and all, I checked all possible combinations of a tree structure vector key inputs could generate. As for the call to the functions themselves, I checked a variety of inputs that could be edge cases that lead to an error in implementation. For example, the left child and the right child of a parent are put in as parameters to check if the functions are capable of reaching there when they need to. I checked numbers greater than the greatest value for floor and numbers less than the smallest value for ceil.

For the exceptions, such as an empty tree, input that is too large or too small according to the vector keys, I made sure the edge cases are checked again. I checked with a large number compared to the vector keys for ceil and checked with a small number compared to the vector keys for floor, since that should trigger an error. In addition, I checked both a small and large number compared to the vector keys and 0, since these will all be invalid arguments for kth_small.