// Keaton Spiller

// CS302

// 03/05/2024 Winter 2024

// Program # 3

Efficiency Writeup

I think If I were to attempt to solve the 2-3 tree again in the future I would spend more time drawing out the black box schematics of where the tree could break. The biggest issue I ran into while debugging each step was in mirroring the balance of the left subtree with the balance of the right subtree of the tree.

As I was programming I would get the left subtree of the tree to work well, but would run into issues balancing the right subtree that would break the left subtree. When to push up the tree to rebalance and add a data member below, or when to add a data member into the same node was a constant challenge in this assignment.

It may have been a bad idea to choose a 2-3 tree instead of a red-black tree. Originally, I thought that a 2-3 tree would be a simpler and cleaner design to implement because there would be less cases, however after spending time solving each edge case at a time I realized that the 2-3 tree would be a very difficult endeavor.

The only place I used getters was in the Node, to get the pointers, and data members, so we can integrate the implementation of multiple derived data types in the same Tree. The only way I could figure out how to access the data of the derived classes within each node was using a setter in the Node function. This is where I did the down casting, to access the data of the class being pointed to by the base pointer.

I order to get pieces of the 2-3 tree to work I tried to modularize as much as I could in the insert function, this helped me debug what each function was doing, and help reduce the complexity of having too much in one function. I had a separate function for pushing up, pushing down, and heavily commented when I would be recursively calling the left subtree to insert, the right subtree to insert, or handling the middle node cases which were handled differently depending on when the current node was full with two data members filled, one data member filled, or when the children are empty or not.

I created separate functions strictly for visuals to understand what my data structure was doing as I was inserting each Node. The height function was tremendously useful when the tree was balanced correctly to help see where each node’s level was located, and to manually draw out what my tree looked like before and after each insert, on paper and on a whiteboard. When the tree was incorrectly balanced the height function would print the incorrect level of node, showing a larger level than was true, and showing up only during gdb debugging.

The efficiency of the tree is semi great, as it attempts to fill the left and right nodes as well as possible but doesn’t always fill both of the middle nodes before dropping down to the next level. I think this is an edge case that would need more debugging to find how we can fill the middle right data value before continuing.

In order to handle duplicate data I had the comparison function check the equivalence of the values that were being compared and return a 2 instead of a 1 or 0, when the obj1 == obj2. This allowed me to do a comparison after each comparison of two objects, and validate when duplicates were found. To account for these detected duplicates, I used a vector and smart pointer.

When these duplicates were found in the insert function, and to avoid adding a duplicate value to the tree I would add this value to a separate vector and immediately return. This was to keep track of duplicate values if needed, and to avoid breaking the push-up / push-down logic when trying to rebalance the tree. Normally in a BST the duplicate values are placed to the right of the subtree. Yet in all the visualizers I explored online everyone avoided adding duplicates.