Noah Johnson

Intro to Discrete Structures Honors Contract

**Time Complexity Analyses for Searching:**

Search in a B+ Tree runs on O(logd n) time. (d = degree of the tree) This is because it doesn’t need to search every index for items and instead it moves from one node to the next based on the key and indexes. My search algorithm uses a nested while loop to compare the key to each value in a node and moving the pointer to the appropriate child node based on those comparisons. When it reaches a leaf it returns the pointer to that node. With n nodes in the tree searching would only take k = tree height units of complexity. With my test case using 18 numbers and a tree height of 4, finding the appropriate node takes 4 units of complexity. This is because my tree has a degree of three. Therefore searching has at most three options when choosing where to search and divide the indexes. [n/3x = 1] ‘x’ being the number of times it needs to traverse to a child node. With the one being the ultimate result of the node being found. Performing the following operations: 3x = n, log(3x) = log(18), x \* log(3) = log(n), x = log(n)/log(3), x = log3(n) Shows that my tree runs at O(log3 n) time when searching. Inserting and Removing take roughly the same time until the node needs to split or merge.

**Report on the Coding:**

I began writing the code for the B+ Tree with the Insert function. It quickly became apparent that the splitting, and later merging, of the nodes would be the most challenging portion of the project. I addressed the issue of splitting by creating separate cases for the root, internal nodes, and leaves. One or two pointers are assigned to new nodes accordingly, a separate arraylist called “OldChildren” is allocated to keep track of the children to be redistributed among the now split parent.

Traversal came next as it was necessary to define each node’s parent. The traversal makes its way through a straight path until it reaches a leaf node. Then it goes back to the leaf’s parent, and then back to the next leaf. This continues until all the children are covered. Then it recedes to the next internal node that hasn’t had all of its children covered. In order to make removal work, I needed to define the leaf node’s siblings. I used a different method of traversal that started at the leftmost leaf and bounced back and forth between internal nodes to find its immediately adjacent neighbor. These two nodes were matched to the appropriate left and right pointers, then the left would point to the right and the right would repeat the process to find the next neighbor. This repeats until the right pointer has nowhere to go but the root.

Removal was difficult to conceptualize at first, but after understanding the balancing nature of the B+ Tree I was able to create methods that would ensure the tree met the requirements after removal. Two methods handled the redistribution and merge processes. The merge method provided an extra rebalancing method to ensure all the nodes were on the proper levels.

I imagine my tree isn’t wholly efficient as this is my first time programming a B+ Tree. There may be better examples online, but I didn’t use them to model my code after. The B+ Tree Visualization available online was the most important tool for making my code. I made every decision to try and emulate what I saw happening in that example. Being able to actually see the operations happening was essential to my understanding of the project.

**Visual:**

This is a representation of how my B+ Tree stores the data, I use a specific means of traversal that prints each level of the tree in order to create this ‘toString’ method for the tree.

