

CS3310

Fourth Assignment (a4)

Report

Brandon Rodriguez

12-3-17

Preface

First off, a working, compiled jar of the assignment can be found in the root directory, called “a4.jar”. To run this, open up a terminal, navigate to the directory, and type “java -jar a4.jar”. The rest should be self-explanatory.

To find the source code of the assignment, navigate from the assignment’s root folder to “root/a3/src/edu/wmich/cs3310/a4”. A majority of the code is located within the controller.java class and the “DataStructures” subfolder.

Used libraries are included in the “root/a4/lib/” folder.

JavaDocs can be found at “root/Documents/JavaDocs/index.html”, as well as within the project dist folder.

Runtime output logs can be located at “root/Documents/RuntimeOutput/”.

Program was created using JetBrains’ “IntelliJ” IDE.

If not familiar with IntelliJ:

Should the jar file not execute, you should be able to load it into IntelliJ and get the same results. When first opening the project, use “Import Project” through IntelliJ’s splash screen. Then select “Create project from existing sources”.

If IntelliJ doesn’t automatically do it for you, make sure to set the “src” directory as the “Sources Root”, and the “tests” directory as the “Test Sources Root”. You may also need to load in junit, which should be provided within the lib folder.

You may also need to “Edit Configurations” and set Main as an application to launch from. But after that, it should work. Just run the program with the green arrow button and it should launch.

Problem Statement

This program focuses on implementation of B-Trees. Specifically, we are expected to implement 2-3 and 3-4 trees to hold ascii table data.

Program Description

Upon launch, this program generates 8 different trees. Four of them are 2-3 trees and four are 3-4 trees. For each tree type, there is a tree that uses a key of: Ascii character, Ascii decimal, Ascii hex, and Ascii octal.

The user is then given a menu of choices.

Option 1 will print data for all nodes in every tree, in breadth-first order.

Option 2 will print data for only leaf nodes, in traversal order.

Option 3 will let the user enter a value to search the trees with. This user input is first checked to see what key types it can possibly parse into. For each key that had a valid parse, the respective trees are searched.

Option 4 will exit program.

All options will print time comparisons of the action in 2-3 trees vs the action in 3-4 trees.

Implementation

In an attempt to ensure the trees were always the minimum possible height, my tree insertion logic always prioritizes filling available children spots, whenever possible.

For example, in a 2-3 tree, if a value's proper location is at a node whom already has 3 leaf children, then the following will happen. Instead of just instantly creating a new tree split and rearranging nodes, it will recursively check the parent of the current position for an open child spot. If there is still no open spot, it will recursively go up one more position and check again. This continues until either there are no more parents to recurse into, or it finally finds a parent with open spots. Only in an instance where there are no available parent/grandparent/etc spots does it attempt to expand the tree upward/outward.

The problem with this approach is that, due to prioritizing open spots, it does not prioritize “proper” placement past initially finding the insertion location. This can (and often does) result in node values inserted in locations that are technically out of order.

As result, each tree type has a large section of code for recursively checking values and double checking that the tree leaves are in the right order. This code is unfortunately extremely repetitive and I swear there has to be a better way to do it. This section of code is essentially called any time the tree is expanded.

I went this route in an attempt to keep the trees as short as possible. Thus, creation time is probably slower than it could be, but the tree is only meant to be created once. Searching, however, can theoretically be done many, many times over, and having the smallest height possible ensures that searching is as fast as possible.

I suppose I should also note that, while I’ve spent a ridiculous amount of time attempting to debug/improve my code for node insertion, there still appear to be some flaws. For example, all keytypes seem to have the same tree height (for 2-3 and 3-4, respectively) except for the ascii keys. For some reason, both ascii key trees have a larger height than any of the other keytypes. The trees still all appear to have the correct ordering at least.

Concluding Analysis

For initial generation, the 2-3 trees appeared to be quicker. Which makes sense, as there is likely less condition checking due to having less cases to account for.

For everything else though, 3-4 trees are quicker. This is likely due to the fact that both tree types should have an identical number of leaves. However, 3-4 trees will always have less internal nodes (and thus less overhead) to manage the same number of leaf nodes.

This is consistent with the general use of B-Trees, and is in line with the fact that some B-Trees have as many as 64 children per node. B-Trees are generally meant for optimized searching, even if it's at the trade off of slightly longer creation time. This fits with the fact that my 3-4 trees appear to be faster than my 2-3 trees in regards to everything except creation.