CS 7353: Analysis of Algorithms Project: Red-Black Tree

Noah Schrick

April 21, 2022

Contents

1 Problem Introduction **2**

2 Program Platform and Submission Files **2**

3 Programming Approach **3**

3.1 Node Class 3

3.2 Red-Black Tree Class 3

3.2.1 Insert 4

3.2.2 Delete 4

3.2.3 Tree Cleanup 4

3.2.4 Display 4

4 Results **4**

5 Part 2: Red-Black Discussion **5**

5.1 Question 2.1 5

5.2 Question 2.2 7

1 Problem Introduction

Red-Black trees are binary search trees that contain an additional one-bit field that denotes the color of a node as either "red" or "black". These trees have additional structural requirements that must be followed, as well as the traditional binary tree requirements. Red-Black trees are also balanced, so a height is guaranteed as a function of n. A full description of Red-Black trees were discussed in lecture, and the algorithms textbook by Cormen also gives a full work-through of the data structure. The main advantage of Red-Black trees is that the worst-case for searching, inserting, and deleting is $\mathcal{O}(log(n))$. For this project, an implementation of Red-Black is provided in C++, with results shown for given problems.

2 Program Platform and Submission Files

This problem was solved using C++ on a Linux system. Attached with the submission is a zip folder that contains:

- A CMakeLists.txt file for compiling
- An "images" folder that contains:
  1. Various images included in this report
- A "src" folder that contains:
  1. A Node.cpp and Node.h file for the Node class and associated functions
  2. A Red-Black.cpp and Red-Black.h file for the Red-Black Tree class and associated functions
  3. The main file
- A "build" folder that contains:
  1. A build.sh script to simplify the build process
  2. A run.sh script to simplify running the program
  3. Various CMake files
  4. The compiled binaries for the program and associated libraries

- Various LaTeX files used in the generation of this report.

This program offers no guarantee of functionality on other Operating Systems. Testing was only conducted on the local Linux machine.

3 Programming Approach

3.1 Node Class

This work approached the problem with a traditional method when working with tree structures by using a series of connected nodes. To achieve this, a "Node" class was implemented. The Node class contains 5 private members: a pointer to the parent node, a pointer to the left child, a pointer to the right child, an integer key, and an integer color. Public class functions include auxiliary functions to get and set the pointers for the parent and children nodes, get and set the color value, and get the key. The Node class is constructed with an integer key value, the color is initialized to red, and the parent and children node pointers are initialized to null pointers using C++'s "nullptr" keyword.

To minimize programmer error and to alleviate effort, an enum titled "colors" was created, with black being set to 0, and red set to 1. Likewise, a color name character array was created, with "black" in the first position, and red in the second. When getting and setting colors, rather than needing to remember (or look for) the value of red or black, the programmer can simply pass "red" or "black", which will then be converted to its proper integer value. Since Red-Black trees use bits to represent the color, this approach was preferred over the utilization of a string member for color.

For debugging purposes, a print function was also created. This function will print the key, color, parent key, left child key, and right child key of a given node.

3.2 Red-Black Tree Class

A red-black tree class was created to aid in the red-black tree functionality. It does not contain any data members other than a pointer to root node. This class primarily works to implement auxiliary functions needed to manage the tree, such as inserts, deletions, rotations, and re-colorings.

3.2.1 Insert

The insert function takes two arguments, both of which are pointers to nodes. The first argument is to aid in the recursion process, and begins as the pointer to the root node. The second argument is the pointer to the node that will be inserted. For the location of insertion, a recursion process is followed, where the key of the node to be inserted is compared to a current node. The tree is then traversed via recursion based on if the key is greater than or less than the current node. Respectively, the right child or left child of the current node is then passed. After the location is determined and the node is inserted, the cleanup function is called.

3.2.2 Delete

The delete function takes two arguments. The first is a pointer to a node, and similar to the insert process, this pointer is used to aid in a recursion process. The second argument is an integer that corresponds to the key of the node to delete. The recursion process is identical to that of the insertion process, with an additional check to see if the key is equal to the key of the current node. If the keys are equal, the deletion process is performed. For the deletion process itself, the approach taken was the successor method as discussed during lecture. The cleanup function was called after the node deletion was completed.

3.2.3 Tree Cleanup

The cleanup function was implemented by following the pseudocode of the three cases discussed during lecture, and the function takes a sole argument, which is a pointer to the node currently being examined. The function makes use of a while loop that checks if the current node and its parent are both red. Each case respectively calls the left and right rotate as necessary.

3.2.4 Display

4 Results

As a note, the "." represents the right child, and the ‘ represents the left child. The terminal-based print does appear to merge together, so in some subtrees it may appear that there are two red nodes next to each other. However,

despite this first-glance look, it is verifiable that there are no two red nodes next to each other.

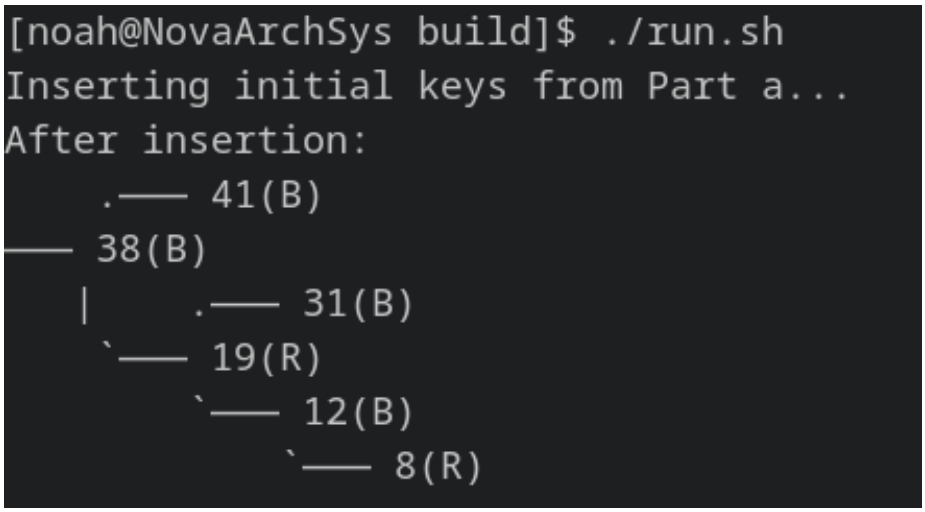

Figure 1: Part 1.B: Initial Tree After Key Insertions

5 Part 2: Red-Black Discussion

5.1 Question 2.1

When inserting a node into a red-black tree and then immediately deleting the same node, the red-black tree before the first insertion is not necessarily the same as the tree after the deletion. An example of this can be seen in Figure 8. When inserting node 100 into the tree, it gets inserted as the right child of node 28. Since both node 28 and node 100 are red, it causes a rotation in the tree. After the node is deleted, the tree does not get rotated back, so the tree now has node 15 as the left child of node 28, instead of node 28 being the right child of node 15.

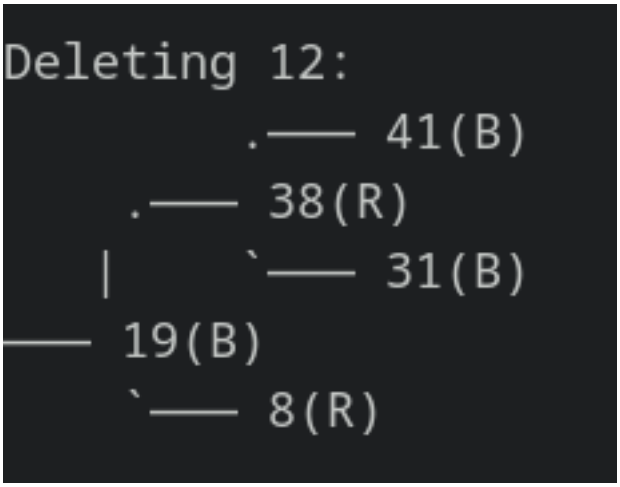

Figure 2: Part 1.B: Tree After Deleting Key 12

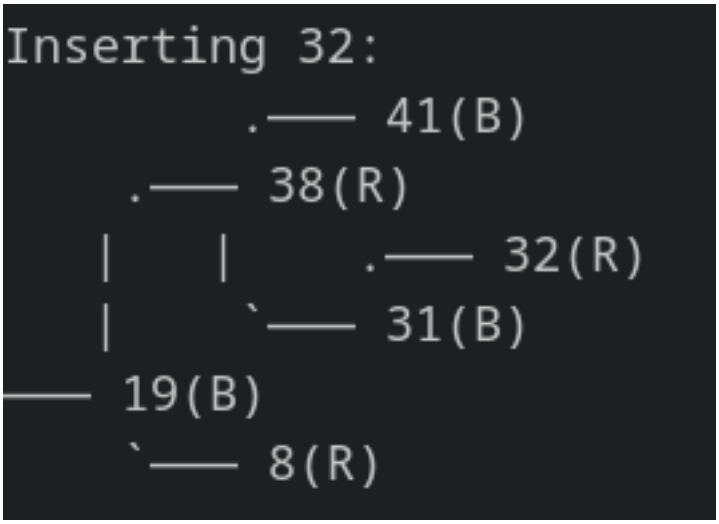

Figure 3: Part 1.B: Tree After Inserting Key 32

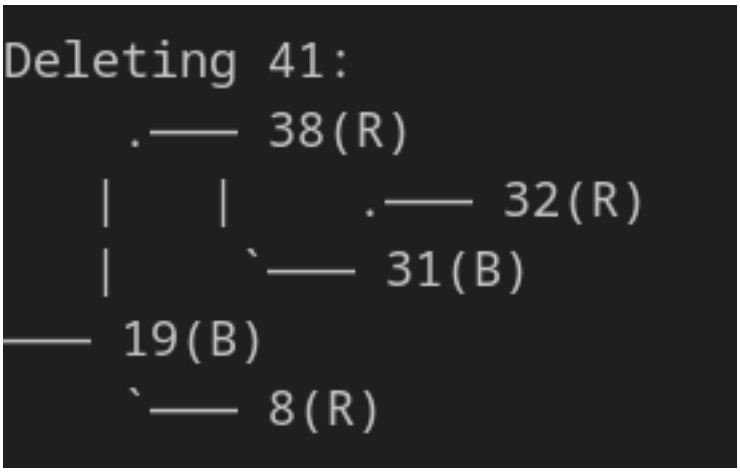

Figure 4: Part 1.B: Tree After Deleting Key 41

5.2 Question 2.2

Maintaining the black height of nodes during the insertion process does not adversely affect the asymptotic performance of the process. Both Case 1 and Case 2 do not change the black height, so no work must be performed in these cases. In Case 1, black height is also not directly changed. In Case 1, the black node may shift down, but the height itself does not change. However, as a result of Case 1, we may color root as red, which then gets corrected to black. This part does change the black height. However, recoloring is constant time, and the black height only needs to be altered for the nearest neighbors. Similar to the rotation process, there is no need to walk down the entirety of the tree; only the nearest children may need an adjustment, which is constant time.

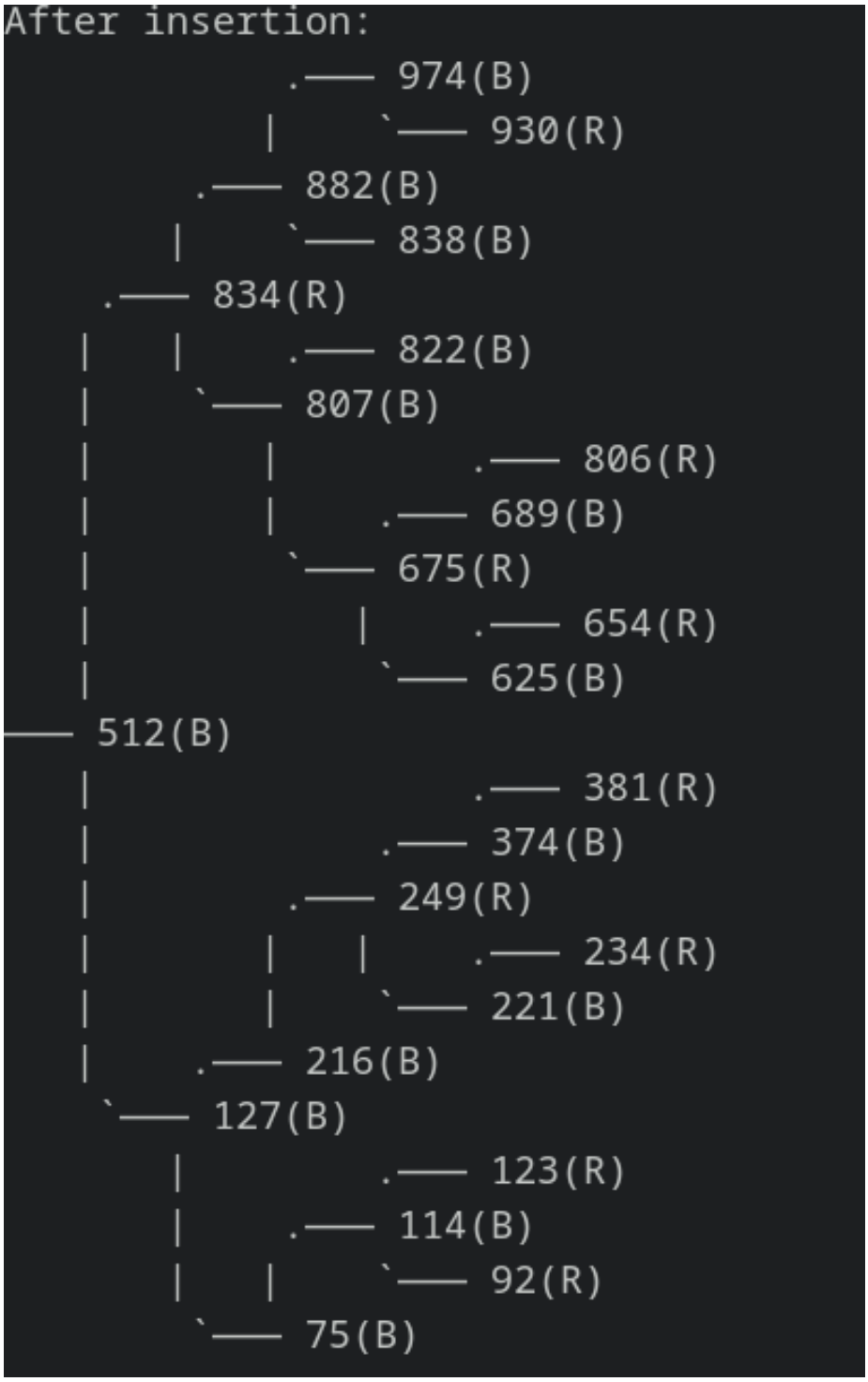

Figure 5: Part 1.C: Initial Tree After Key Insertions

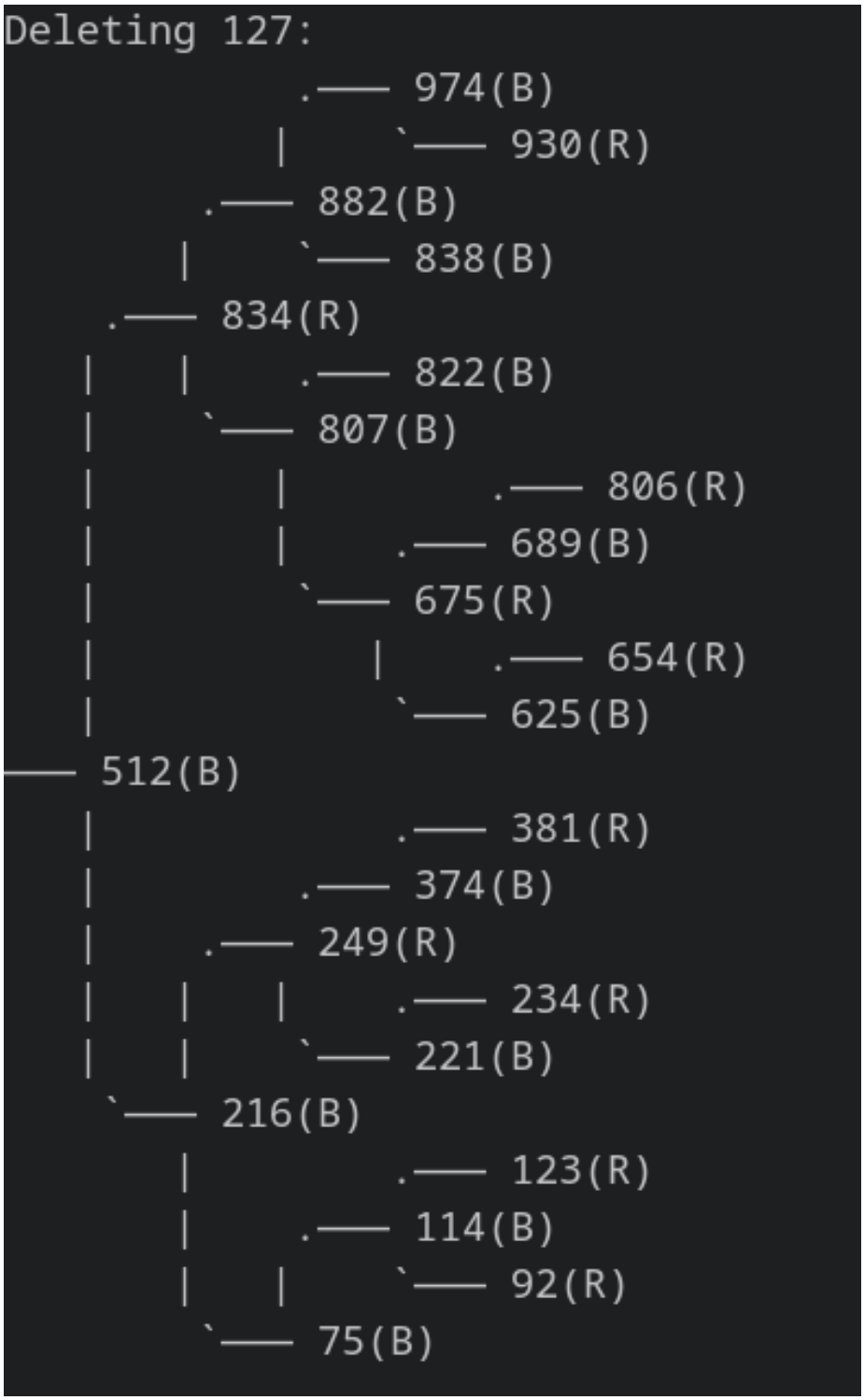

Figure 6: Part 1.C: Tree After Deleting Key 127

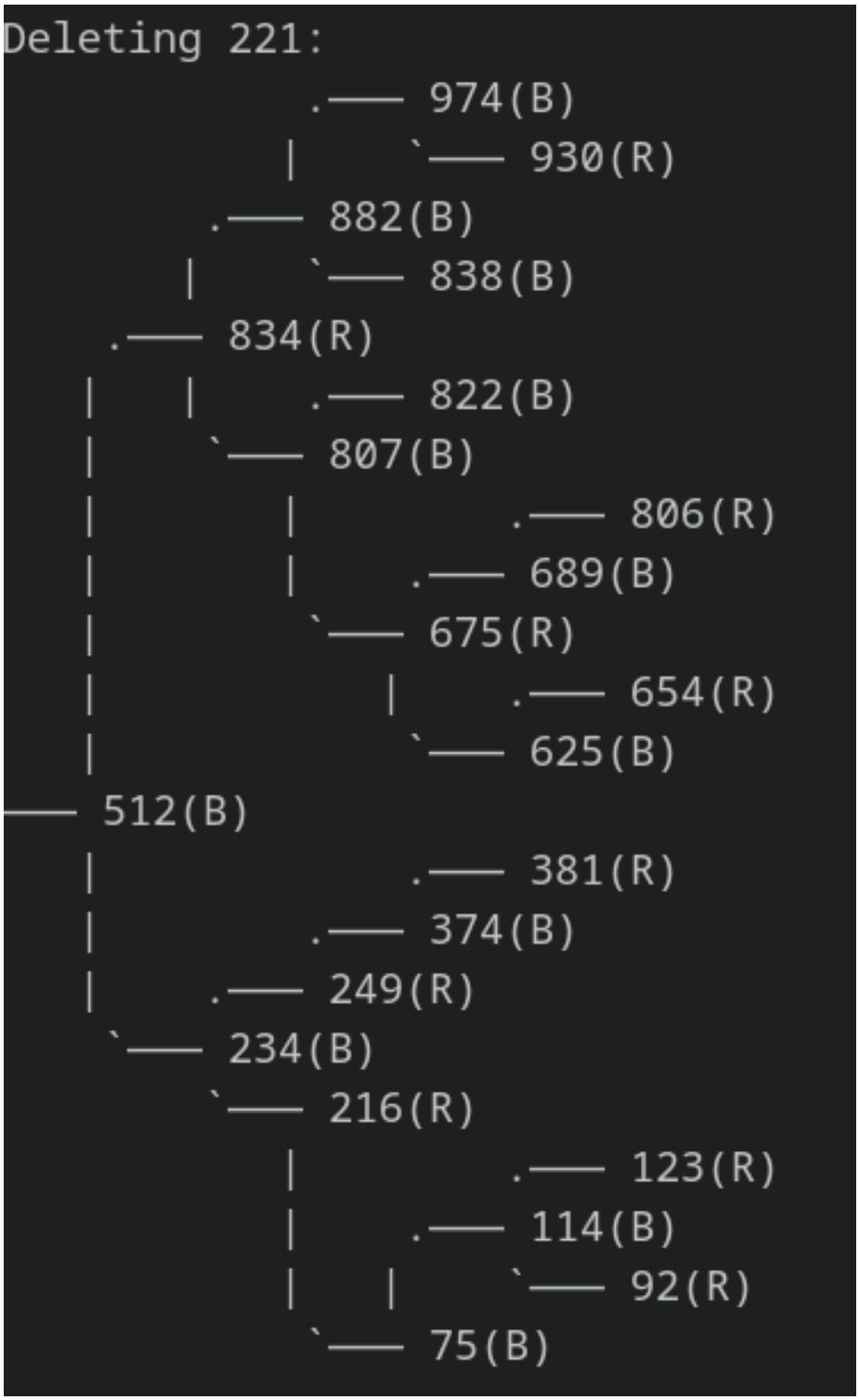

Figure 7: Part 1.C: Tree After Deleting Key 221

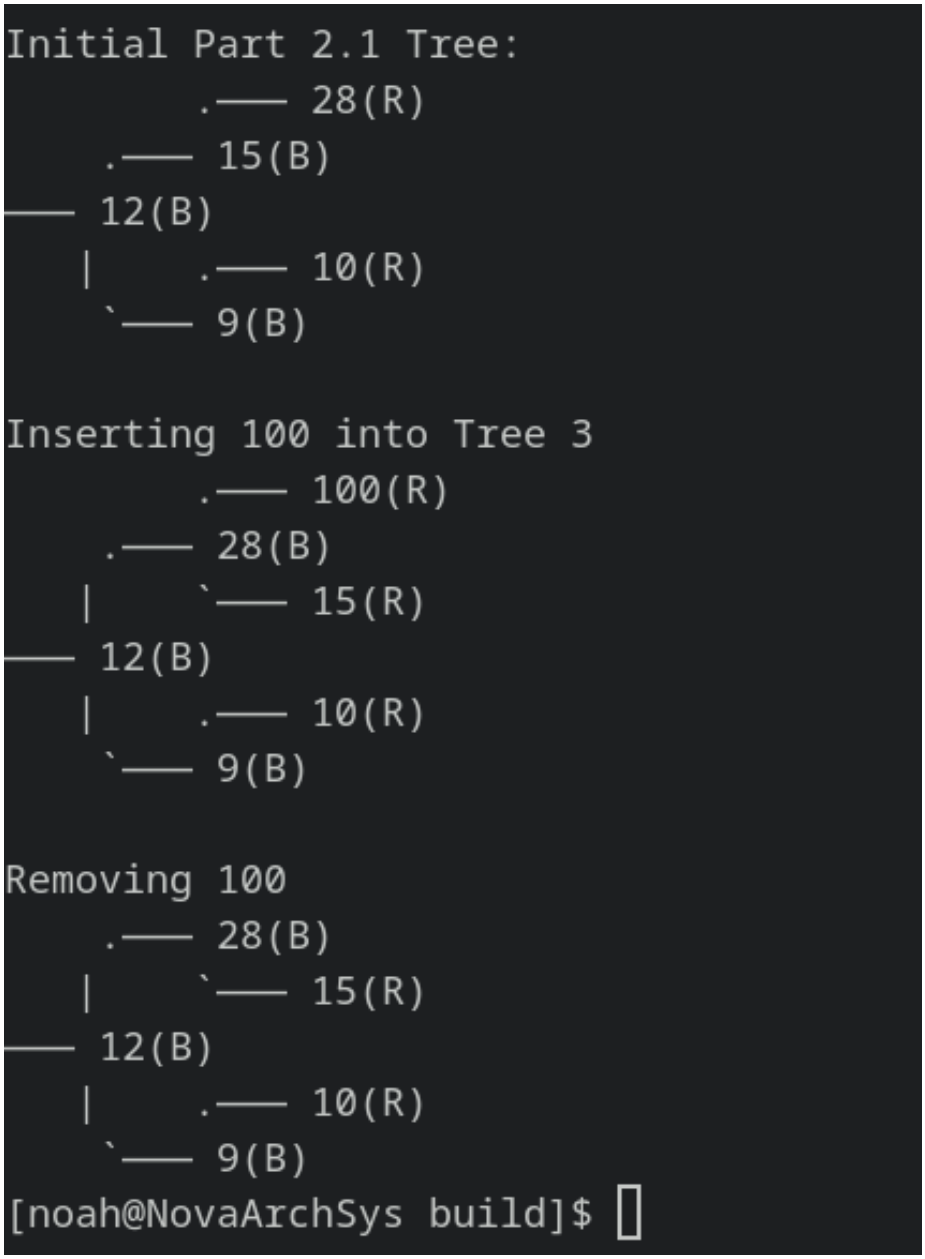

Figure 8: Part 2.1: Example Tree