Program 3: A Heaping Treespoon*

Dr. William Kreahling

November 22, 2021

Overview

In this assignment, you will construct a program for building, heapifying, and printing trees. Your program should be constructed and executed as described below.

1. Input a list of paths from a text file.
2. Store each path in a `PathNode` object.
3. Build a complete tree of `PathNode` objects. **Do not order the tree during creation** but instead *recursively* build the tree according to the order in which they are read from the text file. *You must keep the tree in a linked list.*
4. Heapify the tree, to create a min heap. You may not just swap data, you must move the nodes.
5. *Print the path lengths and paths in the tree level-by-level starting with the root.*
6. **You must store your heap in a linked list with the `PathNode` class shown below. You are not allowed to use any predefined lists or sorting methods.**

Program Structure

Your program should consist of a `Driver` class, a `PathNode` class, and a `Heap` class. Additionally, your program should follow the structure below but you can change the return/data types, parameters and thrown exceptions (as long as your program still follows good coding practices. I.e., If using an objected-oriented language, follow object-oriented guidelines.)

public, private, and return types are not depicted, but should be used accordingly [language dependent]

*Based on an assignment by Dr. Scott Barlowe

```
public class PathNode implements Comparable {
    /** An ArrayList of vertex IDs ordered by appearance in the path. */
    private ArrayList<Integer> path;

    /** Reference to the left child. */
    private PathNode left;

    /** Reference to the right child. */
    private PathNode right;

    /** Reference to the parent. */
    private PathNode parent;

    /** Reference to the node directly to the left on the same tree level. */
    /** Alternatively you could do generationRight instead going to the right */
    private PathNode generationLeft; // left sibling or cousin

    /** True if the node is last in the level. */
    private boolean isLevelEnd;

    /** True if the node is the right-most node in the last level. */
    private boolean isLastNode;

    <SNIP>
}

public class Heap {

    /** Temporary storage for the paths starting at tempPath[1]. */
    private ArrayList<PathNode> tempPath;

    /**
     * Reads inputFile given at the command line and places the contents of each line into the
     * path field found in each PathNode object. The order is the same as found in the text file.
     * Adds the PathNode object to tempPath starting at tempPath[1].
     *
     * @param inputFile Name of the input file to be read.
     * @throws FileNotFoundException if the input file cannot be found.
     */
    readPaths(String inputFile) throws FileNotFoundException

    /**
     * Recursively builds a complete binary tree. Places PathNode objects in tempPath ArrayList into a
     * complete binary tree in order of appearance in the text file.
     *
     * @param index     Index of the current node in tempPath.
     * @param parent    Parent of the current node.
     * @return A reference to the node just placed in the tree.
     */
    PathNode buildCompleteTree(int index, int parent)
```

```
    /**
     * Recursive method that sets isLevelEnd.
     * @param root Root of the subtree.
     */
    setLevelEnd(PathNode root)
```

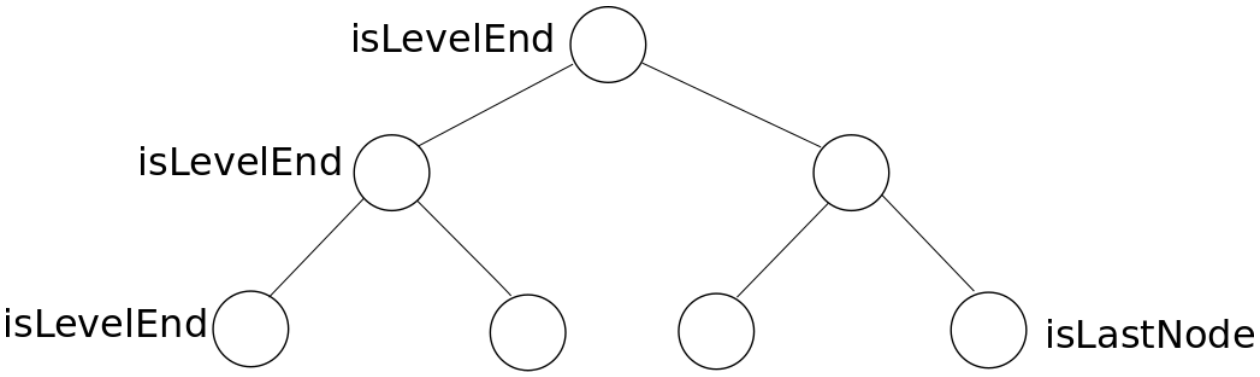

Figure 1: Set the level end fields of the PathNodes

```
    /**
     * Recursive method that sets the "generation" link of PathNode objects from right-to-left.
     * generation is a term I use to indicate nodes on the same level (these may be siblings or
     * cousins)
     * @param root Root of the subtree.
     */
    setGenerationLinks(PathNode root)
```

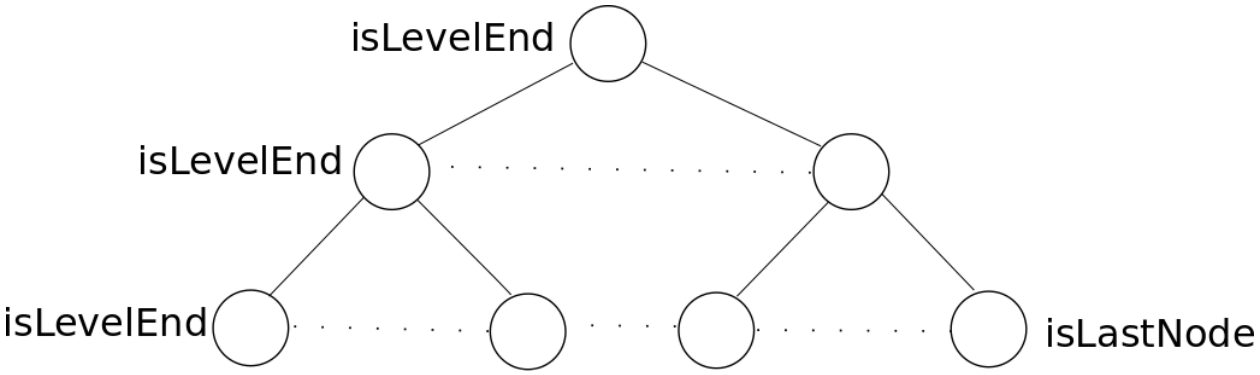

Figure 2: Create Generation Links

```
    /**
     * Prints the path lengths from left-to-right at each level in the tree in the form specified
     * by the instructions.
     * @param root Root of the whole tree to begin printing from.
     */
    printTreeLevels(PathNode root)

}
```

The Driver class should create a Heap object and call its go method and handle exceptions.

Input

Your program should take the path of an input file as a *command line parameter*. It should also take a label as a *command line parameter* to be used as the basename for output files that will be produced. Each line in the input represents a path in a graph (that graph is not depicted). Sample input is shown in Listing 1.

```
0 4 3
0 1
0 4 1 2 3
0 1 2
0 2
0 4 1 2
0 2 3
0 4
0 1 2 3
0 4 1
```

Listing 1: Sample input

The graph resulting from the above input is partially shown in Figure 3. Each vertex contains the length of the path in the represented subtree.

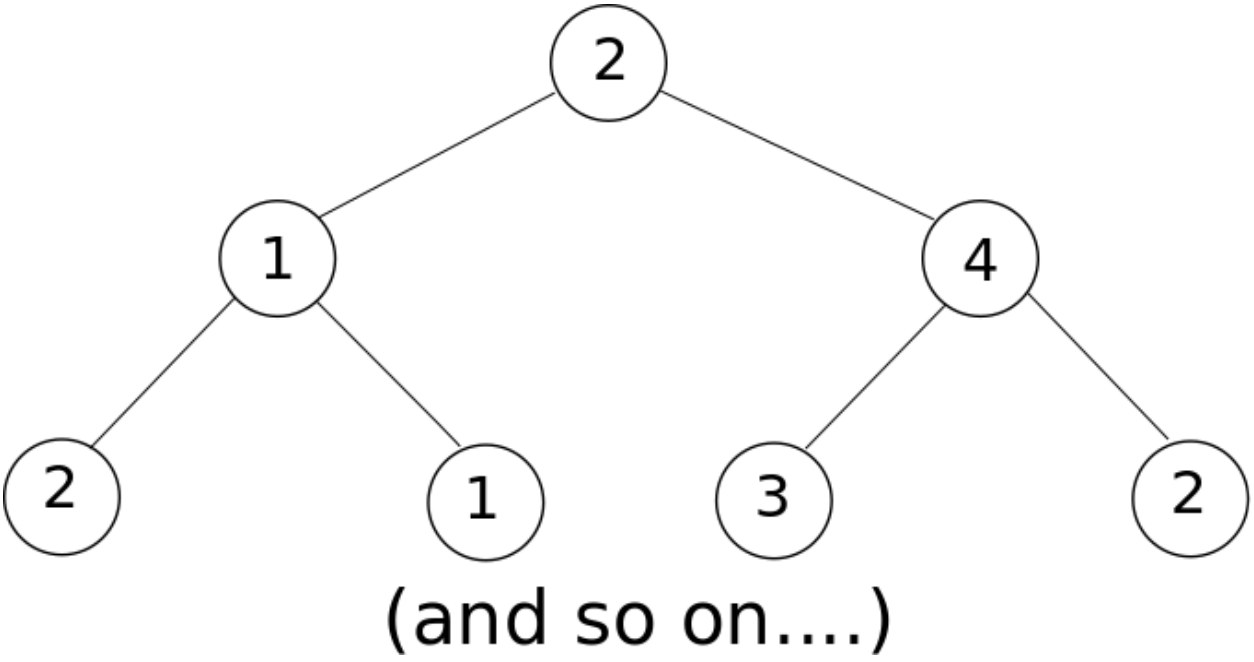

Figure 3: Partial tree from Listing 1.

Output

You program should printout the graph before and after heapifying to files named *<label>Before.dot* and *<label>After.dot*

Sample output for the graph from the sample input in Listing 1, before heapification, would be:

```
digraph inputBefore{
        0[label="2(0, 4, 3)"];
        1[label="1(0, 1)"];
        2[label="4(0, 4, 1, 2, 3)"];
        3[label="2(0, 1, 2)"];
        4[label="1(0, 2)"];
        5[label="3(0, 4, 1, 2)"];
        6[label="2(0, 2, 3)"];
        7[label="1(0, 4)"];
        8[label="3(0, 1, 2, 3)"];
        9[label="2(0, 4, 1)"];
        0 -> 1;
        0 -> 2;
        1 -> 3;
        1 -> 4;
        2 -> 5;
        2 -> 6;
        3 -> 7;
        3 -> 8;
        4 -> 9;
}
```

and

Sample output for the graph from the sample input in Listing 1, after heapification, would be:

```
digraph inputAfter{
        0[label="1(0, 1)"];
        1[label="1(0, 4)"];
        2[label="2(0, 2, 3)"];
        3[label="2(0, 4, 3)"];
        4[label="1(0, 2)"];
        5[label="3(0, 4, 1, 2)"];
        6[label="4(0, 4, 1, 2, 3)"];
        7[label="2(0, 1, 2)"];
        8[label="3(0, 1, 2, 3)"];
        9[label="2(0, 4, 1)"];
        0 -> 1;
        0 -> 2;
        1 -> 3;
        1 -> 4;
        2 -> 5;
        2 -> 6;
        3 -> 7;
        3 -> 8;
        4 -> 9;
}
```

You must adhere to the heapification algorithm covered in class.

When you run these output files through the graphviz program (on agora) using the command:

```
dot -Tpng -ofilenameB.png inputBefore.dot
```

and

```
dot -Tpng -ofilenameA.png inputAfter.dot
```

The graphviz program will create the following PNG files:

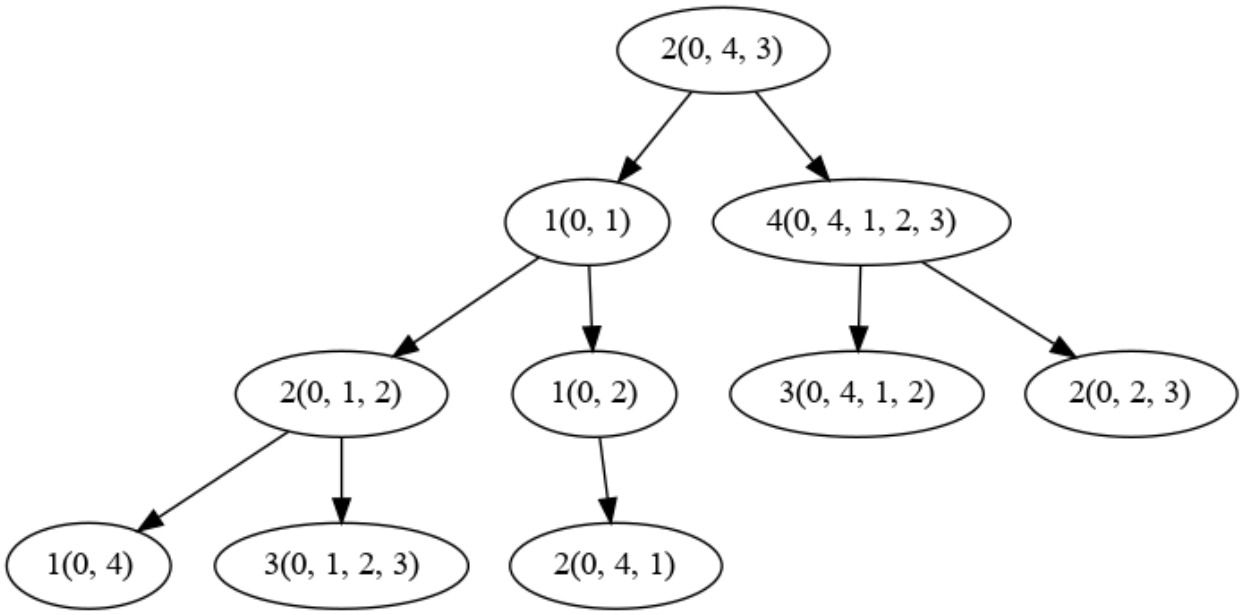

Figure 4: Graph of the input file before heapification

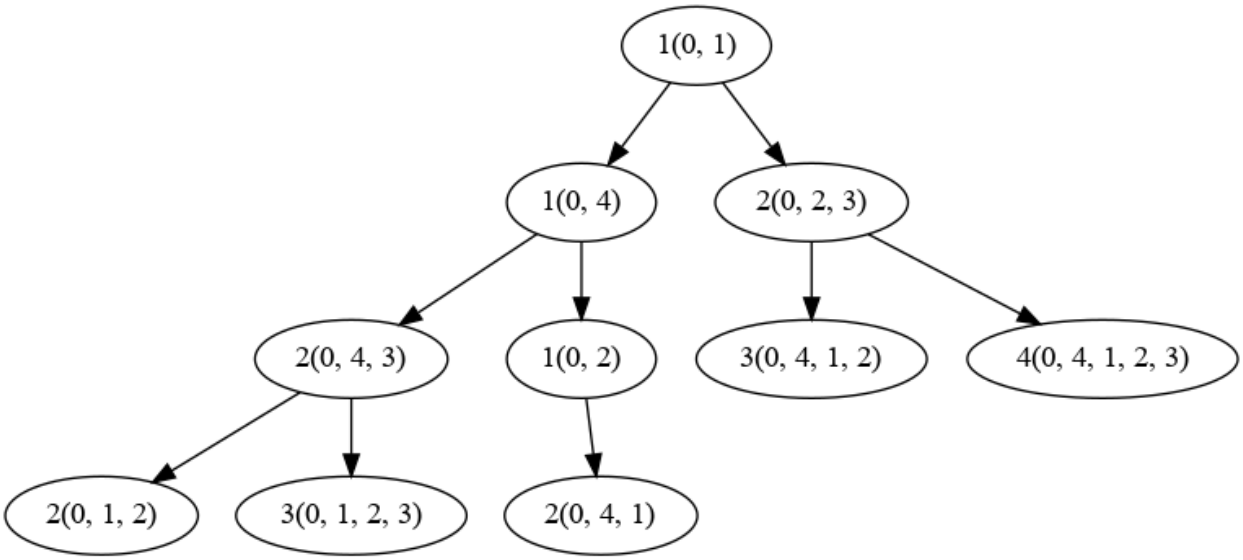

Figure 5: Graph of the input file after heapification

Submit

You may complete the assignment in any language you wish that runs on agora. For any language other than Java, you must supply a README file describing how to compile and/or execute your program and follow the structure above as closely as possible.

```
>javac *.java

>java Driver <filename> <label>
```

- The project is due at midnight on Friday December 10th.
- ***The score is 0 if the program does not compile on agora.***
- You may work in teams of one or two students from the class. You cannot work with students you worked with on program 1 or program 2.
- You may talk with other students in the class about the concepts involved in doing the project, but anything involving actual code needs to just involve your team. In other words, you cannot show your team's code to students outside of your team and you cannot look at the code of another team.
- Your solution must follow the course coding standards.
- Your program will be tested with multiple input graphs.

Submit your files as a tarred and compressed file via **handin** on agora. If your tar file is called program3.tbz, then the command will be:

```
> handin.351.1 3 program3.tbz
```