Project 3 **Scott Barlowe**
CS 351

Order...Order in the Tree!

OVERVIEW

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

Minimum Requirements

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 insertion 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. Print the path lengths and paths in the tree level-by-level starting with the root.

Extra Credit (10 points will be added to any assignment/exam/project that has been graded so far.)

5. Heapify the tree.
6. Print the path lengths and paths in the tree level-by-level starting with the root.

NOTE: You must store your heap in a linked list with the PathNode class below. You also are not allowed to use any predefined sorting methods.

PROGRAM STRUCTURE

Your program should consist of a **heapDriver** class, a **PathNode** class, and a **Heap** class. Additionally, your program should follow the structure below but you can change the return types and parameters where not specified. Additionally, you may change the data type of fields, add fields, and add helper methods.

PathNode Class

```
ArrayList<Integer> path     //  List of vertex ID ordered by appearance in the path.  The list items are ordered
                            //  in the text file from source to destination as they appear in the path.

PathNode left               //  Reference to left child

PathNode right              //  Reference to right child

PathNode parent             //  Reference to the parent

PathNode sibling            // Reference to the node directly to the left.  Here a sibling is
                            // defined as nodes appearing on the same tree level

Boolean isLevelEnd          // True if the node is last in the level going from right to left

Boolean isLastNode          // True if the node is the right-most node in the last level
```

Heap Class

```
ArrayList<PathNode> tempPath        // Temporary storage for the paths starting at tempPath[1]
```

/***

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 of **PathNode** objects is the same as found in the text file. Adds the new **PathNode** object to **tempPath** starting at **tempPath** [1].

@param: String inputFile → Name of the input file to be read
***/

void readPaths(String inputFile)

/***

Recursively builds a complete binary tree. Places **PathNode** objects in **tempPath** into a complete binary tree in order of appearance in the text file. The left child of a parent located at **tempPaths**[index] is found at **tempPath**[2*index] and the right child is found at **tempPath**[2*index + 1].

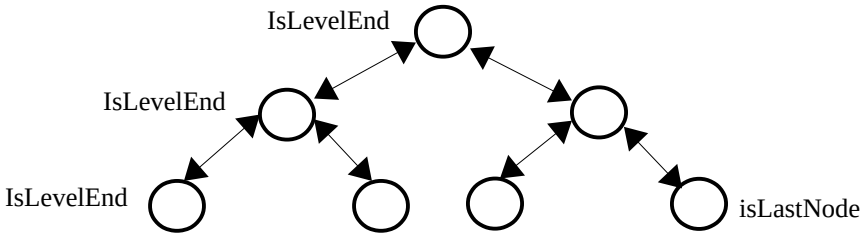
@param: tempPath
@param: index → index of the current node in **tempPath**
@param: newParent → parent of the current node

@returns: Returns a reference to the node just placed in the tree
***/

PathNode buildCompleteTree(ArrayList<PathNode> tempPaths, int index, PathNode newParent)

/***

Recursive method that sets **isLevelEnd.**

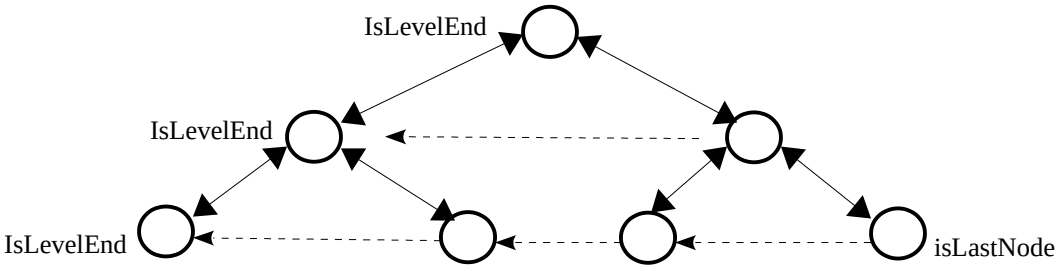


@param: root → root of the subtree
***/

setLevelEnd(PathNode root)

(continued on next page)

/**
Recursive method that sets the **sibling** link of **PathNode** objects as shown below.



@param: root → root of the subtree
**/
setSiblingLinks(PathNode root)

/**
Prints the path lengths from left to right at each level in the tree in the form:

Root: Root's Path Length
Level 1: Path Len (vertex 0, vertex 1, vertex 2,...) Path Len (vertex 0, vertex 1, vertex 2,...)
Level 2: Path Len (vertex 0, vertex 1, vertex 2,...) Path Len (vertex 0, vertex 1, vertex 2,...) Path Len(...)
Level n:

**/
printTreeLevels

/**
Creates a min-heap structure based on path length.
**/
heapify

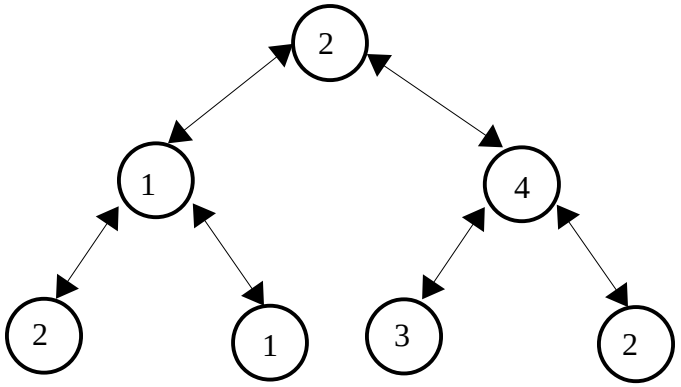
INPUT

Your program should take the name of an input file as a command line parameter. Each line in the argument represents a path in a graph. For example the input

```
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
```

would result in a tree starting out as follows where the number represents the path length:



(...and so on...)

Output for the **printTreeLevels** method before heapifying would look like:

Root: 2 (0, 4, 3)
Level 1: 1 (0, 1) → 4 (0, 4, 1, 2, 3)
Level 2: 2 (0, 1, 2) → 1 (0, 2) → 3 (0, 4, 1, 2) → 2 (0, 2, 3)

...and so on...

Output for the **printTreeLevels** method after heapifying would follow the same format but adhere to the heapifying algorithm covered in class.

You may complete the assignment in any language you wish. If you complete the project in a 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. You may work individually or in groups. You may talk to other teams about the concepts in the project but you are not allowed to share any code.

Your program should compile and execute with

>>>javac *.java

>>>java heapDriver input.txt

The assignment is due at midnight on December 7. No late assignments will be accepted. Submit your assignment using the following command:

>>>handin.351.1 3 project3.jar