Assignment 4
Trees and Heaps

Release Date	Due Date
3/15/2018	3/27/2018

Objectives

- To learn how to implement various types of trees, heaps, and traversals.

Introduction

You have just accepted a brand new software engineering internship on the Death Star. Grand Moff Tarkin wants a database of all living things in the universe. Luckily, you're just an intern, so you only have to code a small part of the prototype.

Your job will be to implement various forms of tree data structures and various related algorithms. You will create a binary search tree, balanced min and max heaps, and algorithms related to them.

Problem Specification

Note: In this assignment you are NOT allowed to use Java's (or other languages') default heap/tree/etc. utilities.

This assignment will have six main parts:
Part A: Reading data
Part B: Array-based min heap
Part C: Explicit height balanced max heap
Part D: Binary search tree
Part E: Generate test data
Part F: Extra credit challenges

Overview

1) Your program should be set up to receive two command line arguments. These two arguments will specify two input files for your program. Both input files will match the format of *starwars.txt*. The two arguments will represent the following:

    - A data file with names to store.
    - A data file with names to search for.

    (We will be testing your program with our own test files. Make sure your program can handle input files of any length.)

2) Your program will store the data from the first file into various data structures and then read the data from the second file and search for each name in it in your previously built structures. For duplicates, report the first hit.

Part A: Reading data

1) Your program should read two command line arguments specifying data files to read. The format of these data files will be that of *starwars.txt*.
2) Read the data from the first specified input file and store it in an array for later use. This array will be referred to as the "**first array**" in these instructions.
3) Read the data from the second specified input file and store it in another array for later use. This array will be referred to as the "**second array**" in these instructions.

Part B: Array-based min heap

3) Create an **implicit** (array-based) min heap.
4) Insert all of the names in the first array into your heap. Report the average time it takes to insert a name.
5) Print out the contents of the heap via a preorder traversal.
6) Print the minimum and maximum values stored in the heap and output the time taken to find each.
7) Search your heap for each name from the second array. If the name is found, report the following:
   - The index of the heap's internal array where the name was found.
   - The depth of the node represented by the index. (The root node has a depth of 0).
   - Whether the node is a leaf node or not.
   - The size of the subtree where the name was found. Size is defined as the number of nodes.
   - The time it took to find the name.
8) Make sure to also account for when names are not found.
9) For duplicates, report the first hit.
10) Report the average time spent searching for a name.

Part C: Explicit height balanced max heap

1) Create an **explicit** (node-based) max heap. You might need to create a **node** class to go with it.
2) The max heap should be **height balanced**.
   - In a height balanced tree, the following holds true for all nodes: $|H_L - H_R| \leq 1$. (where H_L and H_R are the heights of a node's left and right subtrees, respectively.)
3) Again, insert all of the names in the first array into your heap and report the average time it takes to insert a name.
4) Print out the contents of the heap with a postorder traversal.
5) Again, print the minimum and maximum values stored in the heap and output the time taken to find each.
6) Again, search for each name from the second array and report:
   - The depth of the node. (The root node has a depth of 0).
   - Whether the node is a leaf node or not.
   - The size of the subtree where the name was found.
   - The time it took to find the name.
7) Again, make sure to account for when names are not found.
8) For duplicates, report the first hit.
9) Again, report the average time spent searching for a name.

Part D: Binary search tree

1) Create a binary search tree class. Again, this will be of an **explicit** representation.
2) Again, insert all of the names in the first array into your search tree and report the average time it takes to do so.
3) Print out the contents of the tree with an inorder traversal.
4) Again, print the minimum and maximum values stored in the tree and output the time taken to find each.
5) Again, search for each name from the second array and report:
   - The depth of the node. (The root node has a depth of 0).
   - Whether the node is a leaf node or not.
   - The size of the subtree where the name was found.
   - The time it took to find the name.
6) Make sure to account for when names are not found. For duplicates, report the first hit.
7) Again, report the average time spent searching for a name.

Part E: Generate test data

1) After generating output for your first run, generate a random number *n*. Use this value to pick out *n* random strings from your **first array** (there can be duplicates) and create your all your trees using the selected strings.
2) Repeat this part for the following values of *n*. (You can generate for other values of *n* as well.)
    - `n = 10`
    - `n = 100`
    - `n = 500`

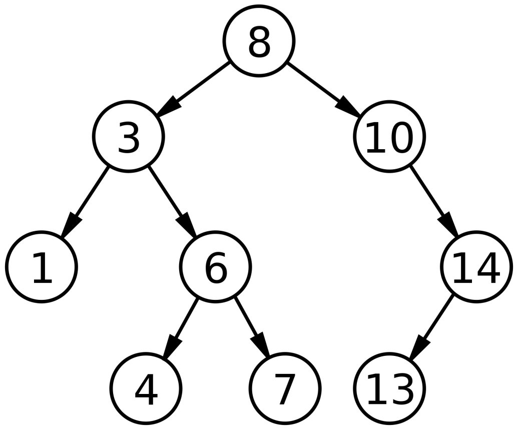
    `Print out your value of n before printing out the test case.`

Part F: Challenges (Extra credit)

The diameter of this tree is 7.
From node 4/7 to node 13.

1) In each of your heap/tree classes, include a method to determine the diameter of the heap/tree. The diameter of a heap/tree is equal to the number of nodes on the longest path between any two nodes without repeating a node. (+5)

2) In each of your heap/tree classes, include a method to find the Lowest Common Ancestor of two nodes. The Lowest Common Ancestor (LCA) of two nodes is the lowest (i.e. deepest) node that has both nodes as descendants, where we define each node to be a descendant of itself as well. (+5)



The LCA of 4 and 7 is 6.
The LCA of 7 and 13 is 8.
The LCA of 1 and 3 is 3.

If you do any of these challenges, please indicate that you've done so in your readme file.

Sample output

The following output is from using the starwars_tiny.txt file for both inputs

```
Min heap:
  Average insert time: xxx nanoseconds.
  Preorder traversal:
    C-3PO, OBI-WAN, R2-D2, LUKE
  Min: C-3PO Max: R2-D2
  Searching for "LUKE" …
    Found at index 2.
    Depth: 1.
    Leaf node.
    Subtree size: 1.
    Time to find: xxx nanoseconds.
  Searching for "OBI-WAN" …
    Found at index 1.
    Depth: 1.
    Subtree size: 2.
    Time to find: xxx nanoseconds.
  Searching for "C-3PO" …
    Found at index 0.
    Depth: 0.
    Subtree size: 4.
    Time to find: xxx nanoseconds.
  Searching for "R2-D2" …
    Found at index 3.
    Depth: 2.
    Leaf node.
    Subtree size: 1.
    Time to find: xxx nanoseconds.
  Max value search time: xxx nanoseconds.
  Min value search time: xxx nanoseconds.
  Average search time: xxx nanoseconds.

Max heap:
  Average insert time: xxx nanoseconds.
  Postorder traversal:
    LUKE, OBI-WAN, C-3PO, R2-D2
  Min: C-3PO Max: R2-D2
  Searching for "LUKE" …
    Found.
    Depth: 2.
    Leaf node.
    Subtree size: 1.
    Time to find: xxx nanoseconds.
  Searching for "OBI-WAN" …
    Found.
    Depth: 1.
    Subtree size: 2.
    Time to find: xxx nanoseconds.
  Searching for "C-3PO" …
    Found.
    Depth: 1.
    Leaf node.
    Subtree size: 1.
    Time to find: xxx nanoseconds.
  Searching for "R2-D2" …
    Found.
    Depth: 0.
    Subtree size: 4.
    Time to find: xxx nanoseconds.
  Max value search time: xxx nanoseconds.
  Min value search time: xxx nanoseconds.
  Average search time: xxx nanoseconds.
```

```
Binary search tree:
  Average insert time: xxx nanoseconds.
  Inorder traversal:
    C-3PO, LUKE, OBI-WAN, R2-D2
  Min: C-3PO Max: R2-D2
  Searching for "LUKE" …
    Found.
    Depth: 0.
    Subtree size: 4.
    Time to find: xxx nanoseconds.
  Searching for "OBI-WAN" …
    Found.
    Depth: 1.
    Subtree size: 2.
    Time to find: xxx nanoseconds.
  Searching for "C-3PO" …
    Found.
    Depth: 1.
    Leaf node.
    Subtree size: 1.
    Time to find: xxx nanoseconds.
  Searching for "R2-D2" …
    Found.
    Depth: 2.
    Leaf node.
    Subtree size: 1.
    Time to find: xxx nanoseconds.
  Max value search time: xxx nanoseconds.
  Min value search time: xxx nanoseconds.
  Average search time: xxx nanoseconds.
```

Submission Requirements

Output Requirements

All your output should be printed to the console as well as a text file.

Code Documentation

You must include documentation for your code. This means that you should include detailed information on the usage of your program as a whole, as well as each function/class within it. This can be accomplished via Javadoc or you can hand-type the documentation. *This includes how to compile and run your program.*

Analysis Report

Write a brief report (submitted as a .pdf file) on your observations of comparing theoretical vs empirically observed time complexities. This report should include:

- A brief description of problem statement(s).
- Algorithm descriptions (if these are standard, commonly known algorithms, then just mention their names along with customization to your specific solution(s), otherwise give the pseudo-code of your algorithms.
- Theoretically derived complexities of the algorithms used in your code
- Table(s) of the observed times.
- Plots comparing the measured times vs. bigger input files. (As n grows, how do your measured times grow?)
- Plots comparing theoretical vs. empirical along with your observations (e.g. do theoretical agree with your implementation, why? Why not?).

Coding Conventions and Programming Standards

You must adhere to all conventions in the CS 3310 Java coding standard. This includes the use of white spaces for readability and the use of comments to explain the meaning of various methods and attributes. Be sure to follow the conventions for naming files, classes, variables, method parameters and methods. Read the material linked from our class web-pages (in case you can’t recall programming styles and conventions from your CS1 and CS2 courses).

Assignment Submission

- Generate a .zip file that contains all your files, including:
  - Signed Plagiarism Declaration
  - Source code files
  - The original data input files
  - Your sample output files
  - Documentation of your code
  - Your analysis report
- **Don’t forget to follow the naming convention specified for submitting assignments.**