Continuous Assessment 2

Update 12/11/2017 12:05 in moviedictionary.py the methods findmin() and findmax() should be interpreted as: find the minimum (maximum) value in the subtree rooted at this node.
Update 10/11/2017 12:05 moviedictionary.py file updated to correct the _isthisapropertree() debugging method.
Update 03/11/2017 12:10 moviedictionary.py file updated to remove the date comparisons from the Movie class.
This lab is part of the formal continuous assessment for CS2515. It focuses on the implementation and use of Binary Search Trees. It will count for up to 10% of the total marks available for this module.

1. Implementing a Movie Catalogue using Binary Search Trees

FlixNet wants to implement library of movie files that users can stream. The library is dynamic, and keeps changing, with movies being added and removed frequently (because the production companies keep offering FlixNet incentives to push movies to the users). Users will want to search the catalogue to see if particular movies are available. This assignment is to write a Python class to represent the catalogue using recursively-defined Binary Search Trees.

We will start with a class, Movie, representing a movie's title, date of release, running time, production status (i.e. released, in production, etc), and some user rating statistics, including a popularity rating, an average vote, and the number of user reviews for the movie. This class is given in moviedictionary.py and you don't need to change it. You should inspect it closely, though, to see what methods are offered. Note: for the required part of this assignment, we will not be able to represent two different movies with the same title -- the first one added wil be the one that is stored, until it is explicitly removed.

An outline of a BinarySearchTree class is given in moviedictionary.py. The class specifies a node in a BST, and since the node contains references to the left and right children, this gives a recursive definition of the (sub)tree which has this node as the root. The initialisation method and the instance variables are given. You do not need to modify this initialisation method. The other methods are given in terms of their signature, with some comments on their use. Some of the methods are public, and some are private (the private methods have names that start with an underscore). You must provide working code for the public methods, without changing the method signatures (i.e. name and input arguments). You are advised to provide working code for the private methods, since these are intended to be helper methods for implementing the public methods, but you are not required to do so -- if you prefer a different way of implementing the class, you are welcome to use it. But when we test your code, the test routines will issue calls to the public methods, so you must provide working definitions for these as specified.

The moviedictionary.py file also includes two extra helper methods which you can use while testing and debugging your code:
_print_structure(self) will print out to the screen a representation of the structure of the tree rooted at this node, one node per line. Each line contains the element at the node, the height, the elements of the two children (or * if no child), and the parent element (or *).
_isthisapropertree(self) will return True if the tree rooted at this node is a properly implemented tree - that is, all parent and child references match up.

i. In order to develop and test your code, you will need to create some trees, so I advise you to start by implementing the add(self, movie) method. The file includes a simple class method _testadd(), which you invoke by typing BSTNode._testadd() on the IDLE command line. Make sure you understand what tree should be created, and then test your code by running this method.

ii. Implement the __str__(self) method, which should create a string representing the inorder traversal of the tree.

iii. Implement the search(self, title) method, which will return the node (i.e. a subtree) containing a movie called exactly title (or None if there is no such movie in the tree). Test your code on the tree returned by the _testadd() method.

iv. Implement the accessor methods -- findmin(self), findmax(self), height(self), size(self), leaf(self), semileaf(self), full(self), internal(self) -- and test your code.

v. Now implement the remove(self, title) method. You are advised to implement _remove_node(self)first, using the pseudocode supplied in moviedictionary.py. Test your code using the class method _test().

vi. Test your code by creating a tree and then adding or removing elements at the command line. Print the tree statistics and print the ordered sequence of elements.

2. Using the Binary Search Tree

Three testfiles are supplied:

1. small_movies_metadata.txt
2. small_repeated_movies_metadata.txt
3. movies_metadata.txt

These files are extracted from the Movies metadata dataset provided by the Data Mining website Kaggle, and the larger file contains over 44000 movies issued since 1900. Note that the department has not inspected the names of these movies, nor inspected the content of these movies. They are taken from public data, and only minimal changes have been made to the dataset, to remove some unprintable characters, and to remove some fields from all data entries. The department is not responsible for the titles.

Using the supplied method build_tree(filename), create binary search trees of all the movies for each of the testfiles. For each file, state the number of unique movies in your binary search tree, the height of the tree, and the minimum height that would be required if the tree was perfectly balanced. For each file, search for movies with the title "Four Lions", "Wonder Woman", "Touch of Evil" and "Delicatessen". For each movie you find in the tree, state the statistics for that node and the ordered sequence of elements in the subtree for that node.