1. 04/12/2020: movieLibStub.py file edited at 18:00 to change the line in build\_library() to file = open(filename, 'r', encoding="utf8") to correct a problem with character encoding in the large movies.txt file. Some people are reporting another error when loading that file with line 23. I haven't seen the error, but I am told that just deleting the line from movies.txt fixes it.
2. 27/11/2020: movieLibStub.py file edited at 13:00 to replace calls to movieBST() in \_testadd() and \_test() with calls to MovieLib()
3. 20/11/2020: bstStub.py file edited at 12:30 to replace 'title' with 'searchitem' in the search(...) method.

This assignment is part of the formal continuous assessment for CS2515. It focuses on the implementation and use of a Binary Search Tree. It will count for up to 10% of the total marks available for CS2515.

Since this is part of the formal assessment, the University's rules on plagiarism apply. You should not copy anyone else's code, you should not download code obtained from any website, you should not get anybody else to write the code for you, and you should not submit code that may have been submitted for similar assignments in previous years for this module. The work you submit must be your own. However, you are free to reuse any code that has been delivered this year in CS2515 lectures or labs by lecturing staff or demonstrators.

# The PyFlix Library

The online Movie Library PyFlix must maintain a library of movie files that are available for users to stream. The library is dynamic, and keeps changing, with movies being added and removed frequently. Users will want to search the catalogue to see if particular movies are available. This assignment is to write Python classes to represent the catalogue using a recursively defined Binary Search Tree. It will involve writing two classes - one for a generic BinarySearchTree, and one for a MovieLib. A MovieLib object will then contain a BST as one of its fields and will issues appropriate method calls to that BST. The end user will never see the BST but will only interact with the MovieLib. The MovieLib object should only have methods that make sense for managing and searching a movie library. We will also need a class to specify Movie objects (and a similar class so that we can test the underlying BST implementation).

# Implementing the Binary Search Tree

An outline of a BinarySearchTree class is given in the bstStub.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. You must provide working code for every method, without changing the default method signatures (i.e., name and input arguments). When we test your code, we will be issuing calls to each of the specified methods. You can add extra methods (public or private) if it helps you to structure your code, but we will not test them with explicit calls. The file also contains a class TestClass, which is used in the test routines to test the implementation of the BST. Note that the BST will accept objects of any class.

The file also includes 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 properly. While you are developing and debugging, you are advised to call this method after every addition or removal of a node, as it will quickly help to identify errors. But do not leave these calls in your final submitted code, as they will slow down the execution significantly on the largest test case.

•\_properBST(self) will return True if the tree rooted at this node is a proper Binary Search Tree. First, it calls self.\_isthisapropertree(), and if that returns true, it then checks that the BST property is satisfied (i.e., that all left descendants have a lower value, and all right descendants have a higher value).

In order to develop and test your code, you will need to create some trees, so I advise you develop the code in the following sequence, testing as you go along.

•Start by implementing the add(self, obj) method. The file includes a simple class method \_testadd(), which you invoke by typing BSTNode.\_testadd() on the python command line. Make sure you understand what tree should be created, and then test your code by running this method.

•Then implement the \_\_str\_\_(self) method, which should create a string representing the in-order traversal of the tree. This will allow you to print out the contents of the trees as you create them, so that you can check whether or not elements have been successfully added or removed.

•Implement the search\_node(search, item)method, which will return the BSTNode which matches searchitem (or None if there is no match). Then implement search(self, searchitem) which returns the element maintained at that BSTNode -- you can do this by simply calling search\_node(search, item), and then reading the appropriate field from the BSTNode that is returned. Test your code on the tree returned by the \_testadd() method.

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

•Now implement the remove(self, searchitem) method. You are advised to implement remove\_node(self) first, using the pseudocode supplied in bstStub.py, and then use that in your implementation of remove(self, searchitem). Test your code using the class method \_test().

•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.

Implementing the movie library

There is much less work to do in this section. An outline of a MovieLib class is given in the movieLibStub.py downloadfile. Again, you need to complete the body of the specified methods. Do not change the method signature of any of these methods, as they will be called during testing. The file also contains a Movie class, and again you should not edit this in any way. - instead, you should use it to create Movie objects. The Movie class represents a movie's title, date of release, and running time. You should inspect it closely 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 will be the one that is stored, until it is explicitly removed.

The idea for the MovieLib class is to maintain a reference to the root of a binary search tree, and when you are asked to add a movie to the library, create the correct Movie object and then call the add(self, obj) method of the referenced BSTNode. Similarly, each of the remaining methods should call the appropriate methods on that BSTNode. In some cases, you will have to check that the BSTNode exists (i.e., that the movie library is not empty). As before, some test methods have been supplied so you can check the basics of your implementation.

Evaluating the movie library

Three test files are supplied:

•smallmovies.txt

•small\_repeated\_movies.txt

•movies.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 School of Computer Science 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 School is not responsible for the titles or for the presence of any movie in that list…

Using the supplied method build\_library(filename), create a library of all the movies for each of the test files. For each file, state the number of unique movies in your binary search tree. For each file, search for movies with the title "Wonder Woman", "Touch of Evil" and "Delicatessen" and print out the movie details if the movie is found in the library.

# Extra

If you have your implementation of the library completed, and you have tested it and made sure it is working, and you want a challenge, then adapt your BST implementation so that it maintains an AVL tree (i.e., a balanced Binary Search Tree), and compare the performance. There are no extra marks for this.

# Submission

•You must submit two files, which must be named bst.py and movieLib.py. These files must contain your implementation of the BSTNode class, the MovieLib class, the supplied Movie class, and any other classes that you implemented that you need for your implementation run

•Make sure that you compile and test your code before submitting, and that you submit exactly the version that you tested (i.e., don't edit the files in any way after your last successful test, not even to improve formatting or add comments).