Notes on Assignment 6

CSCI 2270: Data Structures

Section: 305 Section: 202

TA: Varad Deshmukh TA: Sanskar Katiyar

In this homework, you are asked to build an IMDB-like movie database using a binary search tree (BST) you learnt in the class. We hope that at this point, we have convinced you of the advantage of a binary search tree implementation — a BST has a hierarchical structure unlike a linked list, allowing you to do an average-case search with complexity *O(log(n))*. Each node in the tree stores the details of a single movie: it's title, IMDB ranking, year released and the IMDB rating.

In this homework, a driver file Driver.cpp (already provided to you) reads in records of movies one at a time. Your job will be to implement functions in MovieTree.cpp which generate a BST of these movies, and generates various queries on the tree. The driver will call your implementation of these functions, thus you should use the driver to test your functions. ***(NOTE: You do not have to implement the driver file for this homework.)***

MovieTree.cpp

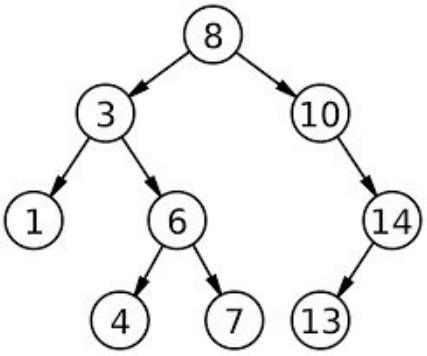
You will implement the following functions in **`MovieTree.cpp`**.

`MovieTree();`
`~MovieTree();`
`void printMovieInventory();`
`void addMovieNode(int ranking, string title, int year, float rating);`
`void findMovie(string title);`
`void queryMovies(float rating, int year);`
`void averageRating();`
`void printLevelNodes(int level);`

Some tips to approach the problem

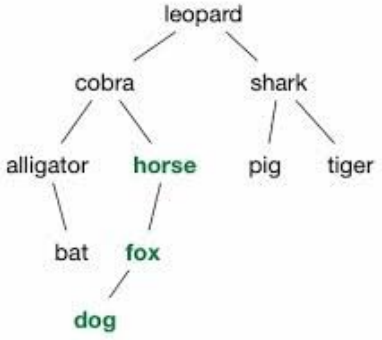
1. You're constructing a BST, where each node in the tree contains the movie record details, as well as a pointer to the left and right child (as every node in a BST should). The layout of the node is available to you as a structure — **`struct MovieNode`** — in the file **MovieTree.hpp**. ***Carefully read this file once before starting.***

2. In a BST, recall that the left subtree of a node contains values "less" than the node value, whereas the right subtree of a node contains values "greater" than the node value. This is easy to visualize when the node values are numbers (see figure below).



It is easy to say that the node with value 3 should be in the left subtree of 8, whereas node with value 10 should be in the right subtree. Here, we are building a BST **not over numbers, but over movie title names.** This means, we will be comparing strings to decide whether a node should be in the left subtree or the right.

Luckily, C++ allows you to use the usual operators (**<, >, ==**) to compare two strings. Thus, "cobra" < "leopard", "shark" > "leopard". This sort of comparison uses the ASCII values of the alphabets to decide which is smaller. Your BST will be loosely structured like the figure below (using movie titles):



3. Recursion is strongly recommended for implementing the functions ***which traverse through most or all of the tree***. Refer back to the preorder, inorder and postorder traversals we covered in the previous recitation. For example, the destructor **~MovieTree()** of the class should delete all the nodes in the class. This is exactly the silver problem from the recitation which used postorder traversal.

4. ***Implementing Recursion***: Notice that the root is a private data member of the MovieTree class. Thus the provided class methods cannot implement recursion directly. In order to perform recursion you will need to implement a helper function. Assuming that there was a class method for preorder traversal, here is how one may implement helper functions for recursion:

```
Node* preorderHelper(Node* node) {
    // not a class method
    if(!root) {
        cout << node->title;
        preorderHelper(node->left);
        preorderHelper(node->right);
    }
}

MovieTree::preorder() {
    // class method calls a helper recursive function
    preorderHelper(root)
}
```

Constructor `MovieTree()`, Destructor `~MovieTree()`

The constructor here is pretty simple. Similar to how we set the head to NULL (or nullptr) in a LL, we set root to NULL (or nullptr).

The destructor, however, requires a bit more work. As mentioned above, we performed deletion of a tree in Recitation 7 (Silver Exercise). There is no difference between a BT and a BST, when it comes to deleting the entire tree. Thus, feel free to port the same code over.

printMovieInventory

```
void printMovieInventory()
```

1. Check if the tree is empty and print the relevant message.

2. *Notice*: Nodes in the BST are stored on the basis of the title of the movies.

3. Given the above fact, ***which of the three traversals will result in a sorted order (here, by title) in a BST***?

4. Implement this traversal algorithm to solve this problem. You will need to implement a helper function for recursion.

addMovieNode

```
void addMovieNode(int ranking, string title, int year, float rating)
```

For most part this function is, essentially, equivalent to insertion in a BST (considering `MovieNode.title` is the key of the BST).

1. **Edge case:** Check if the tree is empty. If yes, dynamically create a new `MovieNode` using the provided arguments and NULL child links. Assign this new node to `root`. This would conclude the current function call.

2. **Base case:** Dynamically create and return a new `MovieNode` using the provided arguments and NULL child links. There is a `MovieNode` constructor to facilitate this creation for you *(check MovieTree.hpp)*.

3. **Recursive call**: While making recursive calls, make sure you are updating the links. The recursive call should also take care of the subtree you are exploring *(similar to search in a BST but update the links)*.

findMovie

```
void findMovie(string title)
```

This function is equivalent to searching in a BST with a few minor differences.

1. **Implement a helper function**: This function should return the node (if found). This is identical to the search/find function covered in the lecture/recitation. Write your function signature accordingly.

2. Use the helper function to identify which message you will print in the original function. This means you need the helper function to return the found node. ***What will your return if the node is not found?***

queryMovies

```
void queryMovies(float rating, int year)
```

In this function, you will be printing movies which meet certain criteria: their rating should be greater than or equal to the passed argument **rating** and were released after the passed argument **year.**

1. You will be traversing the tree and checking against each movie one at a time. The order of processing is given to you — **preorder.**

2. You can reuse the preorder function from the recitation code, and modify the printing function to use the format given to you in the function definition.

3. Make sure to test the edge case first to check if the tree is empty.

averageRating

```
void averageRating()
```

This function should compute the average rating across all the movies in the BST.

HINT: You have solved a similar problem before — the gold problem in the recitation — where you computed the sum of all the nodes in a tree.

1. Recursion is your friend again. Choose the appropriate traversal to scan the entire BST.

2. The traversal function should be called from inside the `averageRating()` function.

3. Keep track of two counters as you traverse the BST — the sum of all the ratings in the nodes traversed so far, and the total nodes traversed so far. These counters can be global variables, but a cleaner implementation would be to pass them as references in your traversal function.

4. Use the two counters — the sum of all ratings, and the total movies — to compute the average rating.

5. Make sure to test the edge case first to check if the tree is empty.

printLevelNodes

```
void printLevelNodes(int level);
```

This function will print, from left to right, all the nodes which exist at a certain depth in the tree.

1. You might have probably got the hang of it now — we are doing tree traversal again. We leave it to you to figure the correct order of traversal.

2. The trick is to pass to the traversal function a variable that keeps track of the current level in the tree — call it **`curLevel`**.

3. **Base case:** So far, you have been returning from the recursion once you hit the leaf node. In this function, the base case will change slightly. At each node of the tree traversal, you'll be creating an additional check --- if **`curLevel == level`** — and return if `true` (both variables will be passed to the traversal function). *NOTE: The original check for the leaf node is still required.*

4. **Recursive call:** You will be recursively calling the traversal function on the left and right children at each node.

5. Don't print anything if a node at the provided depth does not exist.