CSCI 2270 – Data Structures

Instructors: Ashraf, Godley

Assignment 4

Due Wednesday, July 8 2020 @ 11:59 PM

Binary Search Tree

OBJECTIVES

1. **Insert nodes to build a binary search tree (BST)**
2. **Search and Traverse a BST**
3. **Delete nodes in a BST**
4. **Perform a Rotate on your BST**

Background

In 2009, Netflix held a competition to see who could best predict user ratings for films based on previous ratings without any other information about the users or films. The grand prize of US$1,000,000 was given to the BellKor's Pragmatic Chaos team which bested Netflix's own algorithm for predicting ratings by 10.06%. This kind of data science is facilitated with the application of good data structures. In fact, cleaning and arranging data in a conducive manner is half the battle in making successful predictions. Imagine you are attempting to predict user ratings given a dataset of IMDB's top 100 movies. Building a binary search tree will enable you to search for movies and extract their features very efficiently. The movies will be accessed by their titles, but they will also store the following features:

- IMDB ranking (1-100)
- Title
- Year released
- IMDB average rating

Your binary search tree will utilize the following struct with default and overloaded constructors:

CSCI 2270 – Data Structures

Instructors: Ashraf, Godley

```
struct MovieItem{
    int ranking;
    string title;
    int year;
    float rating;
    MovieItem* left = NULL;
    MovieItem* right = NULL;

    MovieItem(int rank, string t, int y, float r) {
        title = t;
        ranking = rank;
        year = y;
        rating = r;
    }
};
```

MovieInventory Class

Your code should implement a binary search tree of MovieItem data type. A header file that lays out this tree can be found in *MovieInventory.hpp* on Moodle. As usual, **DO NOT** modify the header file. *You may implement helper functions in your .cpp file to facilitate recursion if you want as long as you don't add those functions to the MovieInventory class*.

MovieInventory()

➔ Constructor: Initialize any member variables of the class to default

~MovieInventory()

➔ Destructor: Free all memory that was allocated

void printMovieInventory()

➔ Print every node in the tree in alphabetical order of titles using the following format:

```
// for every Movie node (m) in the tree
cout << "Movie: " << m->title << " " << m->rating << endl;
```

If there is no movie entry in the tree, print the following message instead:

```
cout << "Tree is Empty. Cannot print" << endl;
```

void addMovieItem(int ranking, std::string title, int year, float rating)

➔ Add a node to the tree in the correct place based on its **title**. Every node's left children should come before it alphabetically, and every node's right children should come after it alphabetically. *Hint: you can compare strings with <, >, ==, string::compare() function.*

➔ *For example*, if the root node of the tree is the movie "Forrest Gump", then the movie

"Iron Man" should be in the root's right subtree and "Aladdin" should be in its left subtree.

➔ You can assume that no two movies have the same title

void deleteMovieItem(std::string title)

➔ Delete the node that contains the title. **When deleting a tree node with both children take the replacement from its in-order successor** (min of the right sub-tree). If the movie does not exist in the data structure, print the following message

```
cout << "Movie: " << title << " not found, cannot delete." << endl;
```

void getMovie(string title)

➔ Find the movie with the given **title**, then print out its information:

```
cout << "Movie Info:" << endl;
cout << "==================" << endl;
cout << "Ranking:" << node->ranking << endl;
cout << "Title  :" << node->title << endl;
cout << "Year   :" << node->year << endl;
cout << "rating :" << node->rating << endl;
```

If the movie isn't found, print the following message instead:

```
cout << "Movie not found." << endl;
```

void searchMovies(float rating, int year)

➔ Print all the movies whose rating is at least as good as the input parameter **rating** and that are newer than **year** in the **in-order** fashion using the following format:

```
cout << "Movies that came out after " << year << " with rating at
least " << rating << ":" << endl;
// each movie that satisfies the constraints should be printed with
cout << m->title << "(" << m->year << ") " << m->rating << endl;
```

If there is no movie entry in the tree, print the following message instead:

```
cout << "Tree is Empty. Cannot query Movies" << endl;
```

void averageRating()

➔ Print the average rating for all movies in the tree. If the tree is empty, print 0.0. Use the following format:

```
cout << "Average rating:" << average << endl;
```

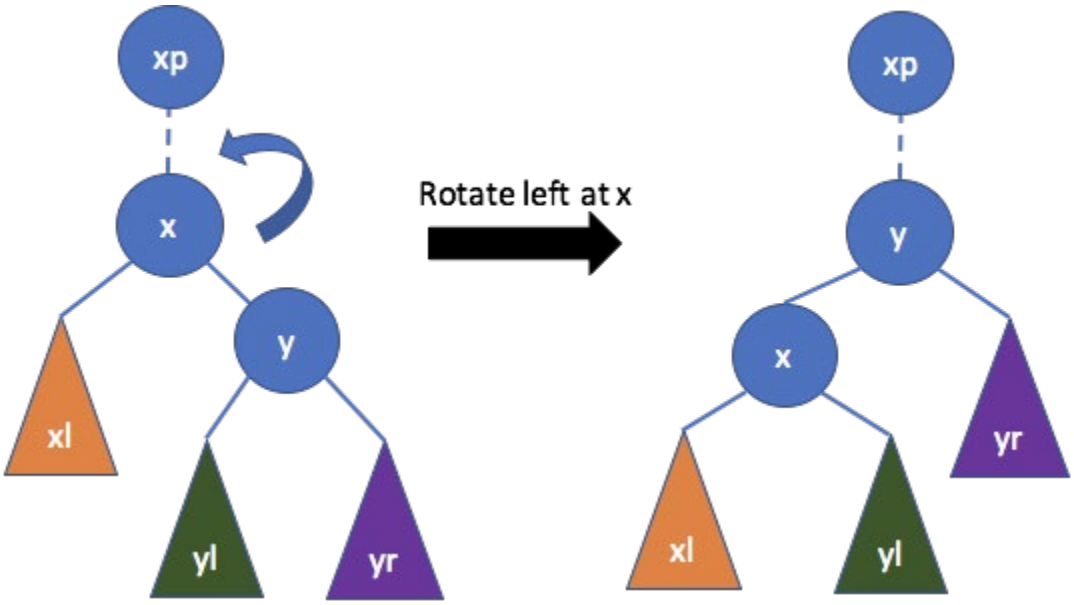
If there is no movie entry in the tree, print the following message instead:

```
cout << "Average rating:0.0" << endl;
```

void leftRotate(std::string title)

➔ Rotate the node with the given **title** towards the left. Refer to the following illustration. A left rotation is performed at node **x.**

- After the rotation, the parent of **x** now becomes the parent of **y**. **y** becomes parent of **x**.
- Set the children pointers: Three children pointers are modified. i) The left subtree of **y** becomes the right subtree of **x**. ii) **x** and it's left descendants become the left subtree of **y**, with **x** as the left child. iii) If **x** was the left (or right) child of **xp** before the rotation, make **y** the left (or right) child of **xp** after rotation accordingly. This can be implemented by comparing the titles of **y** and **xp**: if **y.title** < **xp.title**, make **y** the left child of **xp**, else make **y** the right child of **xp**. A less riskier approach is to keep track of whether **x** was the left child or the right child before the rotation, and make **y** the left or the right child accordingly.
- Take care of the boundary cases: i) If **x** was the root before rotation, make **y** the new root. ii) If **x** does not have a right child, don't perform any rotation.



Driver

For this assignment, the driver code has been provided to you in the *Driver.cpp* file. The main function will read information about each movie from a CSV file and store that information in a MovieInventory using your **addMovieItem** function implementation.

CSCI 2270 – Data Structures

Instructors: Ashraf, Godley

The name of the CSV file with this information should be passed in as a command-line argument. Example files are *Movies.csv*, *Documentaries.csv* on Moodle in the format:

```
<Movie 1 ranking>,<Movie 1 title>,<Movie 1 year>,<Movie 1 rating>
<Movie 2 ranking>,<Movie 2 title>,<Movie 2 year>,<Movie 2 rating>
Etc…
```

Note: For autograding's sake, insert the nodes to the tree in the order they are read in.

After reading the information on each movie from the file and building the tree, the user is displayed the following menu:

```
======Main Menu======
1. Find a movie
2. Search movies
3. Print the inventory
4. Average Rating of movies
5. Delete movie
6. Rotate movies around
7. Quit
```

The menu options have the following behavior:

- **Find a movie:** Calls your tree's **getMovie** function on a movie specified by the user. The user is prompted for a movie title.

```
cout << "Enter a movie title:" << endl;
```

- **Search movies:** Calls your tree's **searchMovies** function on a rating and year specified by the user. Prompts the user for a rating and year.

```
cout << "Enter a minimum rating:" << endl;
// get user input
cout << "Enter a minimum year:" << endl;
```

- **Print the inventory:** Call your tree's **printMovieInventory** function
- **Average Rating of movies:** Calls your tree's **averageRating** function
- **Delete movie**: Calls your **deleteMovieItem** function on a movie specified by the user. The user is prompted for a movie title.

```
cout << "Enter a movie title:" << endl;
```

- **Rotate movies around:** Calls your **leftRotate** on a movie specified by the user.

The user is prompted for a movie title, which is the title of the node the rotation is centered on.

```
cout << "Enter a movie title:" << endl;
```

- **Quit:** Program exits after printing a friendly message to the user.

```
cout << "Goodbye!" << endl;
```

"Please note that once you are done with your assignment on Coderunner you need to click on 'finish attempt' and the 'submit all and finish'. If you don't do this, your submission will not get graded."