CSCI 2270 – Data Structures

Instructor: Asa Ashraf

Assignment 6

Due Sunday, October 17, 2021 @ 11:59 PM

Binary Search Tree

OBJECTIVES

1. **Build a binary search tree (BST)**
2. **Traversal, Search and Deletion in a BST**

Background

A company wishes to build a directory for its employees. The primary operation will be to search a particular employee based on their unique employee ID number. The search operation should have fast lookup times.

Your team has decided to organize this directory as a binary search tree (BST), which will enable fast search for employee details and to extract their details efficiently. The employee details will be indexed by their unique employee ID numbers, but they will also store the following details:

- **empId**: Unique Employee ID (> 0).
- **empName**: Employee's name.
- **empLevel**: Employee's position grade in the company. Integers from [1-10].
- **empPhone**: Employee's office phone number.
- **empJoiningYear**: The year when the employee joined the company.

Your BST will utilize the following node struct:

```
struct EmployeeNode {
    int empId;
    string empName;
    int empLevel;
    string empPhone;
    int empJoiningYear;
    EmployeeNode* left  = NULL;
    EmployeeNode* right = NULL;
};
```

CSCI 2270 – Data Structures

Instructor: Asa Ashraf

Assignment 6

EmployeeTree Class

Your code should implement a binary search tree (BST) of EmployeeNode data type. A header file that lays out this tree can be found in *EmployeeTree.hpp* on Canvas. 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 EmployeeTree class*.

EmployeeTree()

➔ Constructor: Sets **`root`** data member to NULL.

~EmployeeTree()

➔ Destructor: Free all memory that was allocated and set **`root`** to NULL.

void printEmployeeDirectory()

➔ Print every node in the tree in ascending order of employee ID using the following format:

```
cout << node->empId << ", " << node->empName << endl;
```

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

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

void addEmployee(int empId, string empName, int empLevel, string empPhone, int empJoiningYear)

➔ Add a node to the BST in the correct place based on its Employee ID. In this case, a node X's left subtree will contain employees with **empId** < X.empId and the right subtree will contain **empId** > X.empId.

➔ *For example*, if the root node of the tree is the employee "ABC" with employee ID 1234, then the employee "DEF" with employee ID 2546 should be in the root's right subtree and "GHI" with employee ID 76 should be in its left subtree.

➔ It is guaranteed that no two employees have the same employee ID.

void findEmployee(int empId)

➔ Find the employee with the given employee ID, then print out their information:

```
cout << "Employee Information" << endl;
cout << "====================" << endl;
cout << "ID     : " << node->empId << endl;
```

```
cout << "Name   : " << node->empName << endl;
cout << "Level  : " << node->empLevel << endl;
cout << "Phone  : " << node->empPhone << endl;
cout << "Joined : " << node->empJoiningYear << endl;
```

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

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

void queryEmployees(int level, int year)

➔ Print all the employees whose joining year is greater than the input parameter **year** and whose level greater than or equal to the input parameter **level** in the **preorder** fashion using the following format:

```
cout << "Employees who joined after " << year << " and are at least
at level " << level << ":" << endl;

// each employee that satisfies the constraints should be printed
with
cout << "====================" << endl;
cout << "ID     : " << node->empId << endl;
cout << "Level  : " << node->empLevel << endl;
cout << "Joined : " << node->empJoiningYear << endl;
```

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

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

void printLevelNodes(int node_level)

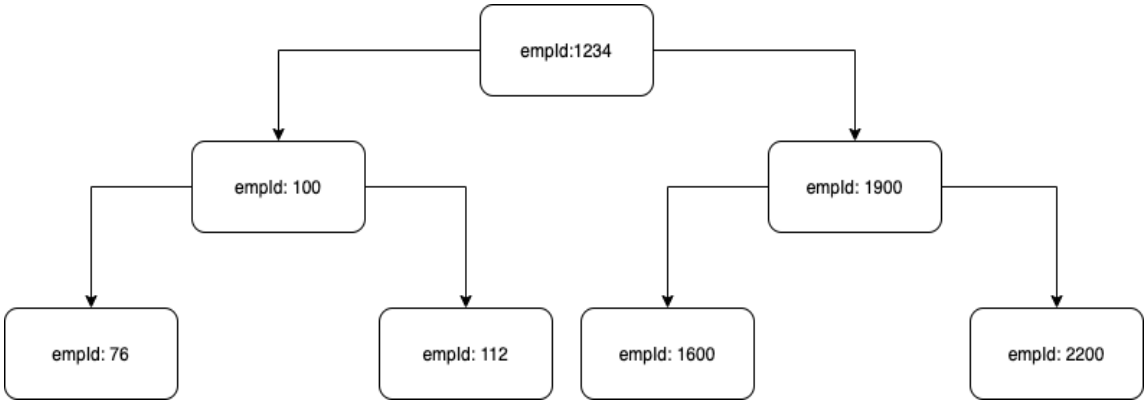
➔ Print all the nodes in the tree from left to right that are exactly at depth = **node_level** from the root of the tree. If the user passes a **node_level** value which is more than the maximum depth of the tree, then don't print anything and just exit the function.

➔ Use the following format to print the node's details:

```
cout << node->empId << ", " << node->empName << endl;
```

For example, for the tree (on next page), if **level** = 2, your function should print the

above statement for nodes corresponding to emp_id: 76, 112, 1600, 2200 in that order (nodes from left to right at depth=2 from the root node of the tree).



int getKthLowestEmployeeID(int K)

- ➔ Return the empId for Employee with the **K**th lowest empId in the BST.
- ➔ It is guaranteed that **K** >= 1 && **K** <= *Number of Nodes* in the tree.
- ➔ Example: K = 1 implies we want to find the Employee with the lowest empID. For the tree in the above figure, the result is ***76***.
- ➔ Example*:* K = 3 implies we want to find the Employee with the 3rd lowest empID. For the tree in the above figure, the result is ***112***.

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 employee from a CSV file and store that information in an EmployeeTree using your **addEmployee** function implementation.

The name of the CSV file with this information should be passed in as a command-line argument. An example file with fake data is provided to you. You can find *Employees.csv* on Canvas. It is organized in the following format:

```
<1:empId>,<1:empName>,<1:empLevel>,<1:empPhone>,<1:empJoiningYear>
<2:empId>,<2:empName>,<2:empLevel>,<2:empPhone>,<2:empJoiningYear>
...
```

CSCI 2270 – Data Structures

Instructor: Asa Ashraf

Assignment 6

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

```
====== Main Menu ======
1. Find an employee
2. Query employees with year and level
3. Print the directory
4. Get an employee with Kth lowest empId
5. Print employees at tree level K
6. Quit
```

The menu options have the following behavior:

- **Find an employee:** Calls your tree's **findEmployee** function on an employee specified by the user. The user is prompted for an employee ID.
- **Query employees with year and level:** Calls your tree's **queryEmployees** function for a year and level specified by the user. Prompts the user for a year and level.
- **Print the directory:** Call your tree's **printEmployeeDirectory** function
- **Get an employee with Kth lowest empId**: Calls your tree's **getKthLowestEmployeeID** function. User is prompted for k value.
- **Print employees at tree level k**: Call your tree's **printLevelNodes** function on the level specified by the user. The user is prompted for a node_level.
- **Quit:** Program exits.