Data Structures & Algorithms LAB – Fall 2015

(BS-SE-F14 Morning & Afternoon)

Lab # 10 (Ungraded)

Instructions:

- **Attempt the following tasks exactly in the given order.**
- Make sure that there are no ***dangling pointers*** or ***memory leaks*** in your programs.
- Indent your code properly.
- Use meaningful variable and function names. Follow the naming conventions.
- Use meaningful prompt lines/labels for all input/output.
- **Although this is an Ungraded Lab (due to holiday on 24th December), you are required to complete all tasks of this lab, individually. You may need almost all the functions from this lab in your next lab. So, make sure that you have the working implementation of all the functions from this lab, when you come to the next lab.**

Task # 1.1 *(Max Time: 30 Minutes)*

In this lab you are going to start implementing a class for creating and storing **Binary Search Trees (BST)**. Each node of this BST will store the **roll number**, **name** and **CGPA** of a student. The class definitions will look like:

```
class StudentBST;

class StudentNode {
      friend class StudentBST;
  private:
      int rollNo;              // Student's roll number (must be unique)
      string name;             // Student's name
      double cgpa;             // Student's CGPA
      StudentNode *left;       // Pointer to the left subtree of a node
      StudentNode *right;      // Pointer to the right subtree of a node
};

class StudentBST {
  private:
      StudentNode *root;       // Pointer to the root node of the tree
   public:
      StudentBST();            // Default constructor
};
```

Implement the following two public member functions of the **`StudentBST`** class:

`bool insert (int rn, string n, double c)`
This function will insert a new student's record in the **`StudentBST`**. The 3 arguments of this function are the **roll number**, **name**, and **CGPA** of this new student, respectively. The insertion into the tree will be done based upon the roll number of the student. This function will check whether a student with the same roll number already exists in the tree. If it does not exist, then this function will make a new node for this new student, insert it into the tree at its appropriate location, and return **true**. If a student with the same roll number already exists, then this function should not make any changes in the tree and should return **false**. This function should not display anything on screen.

`bool search (int rn)`
This function will search the **`StudentBST`** for a student with the given **roll number** (see the parameter). If such a student is found, then this function should display the details (roll number, name, and CGPA) of this student and return **true**. If such a student is not found then this function should display an appropriate message and return **false**.

Task # 1.2 *(Max Time: 15 Minutes)*

`void inOrder ()`
This function will perform an **in-order** traversal of the **`StudentBST`** and display the details (roll number, name, and CGPA) of each student. The list of students displayed by this function should be **sorted in increasing order** of roll numbers of students. This function will be a public member function of the StudentBST class. This function will actually call the following helper function to achieve its objective.

`void inOrder (StudentNode* s)` ***// private member function of StudentBST class***
This will be a **recursive** function which will perform the in-order traversal on the subtree which is being pointed by **`s`**. This function will be a **private** member function of the **`StudentBST`** class.

Task # 1.3 *(Max Time: 15 Minutes)*

`~StudentBST ()`
This is the destructor for the **`StudentBST`** class. This function will call the following helper function to achieve its objective.

`void destroy (StudentNode* s)` ***// private member function of StudentBST class***
This will be a **recursive** function which will destroy (deallocate) the nodes of the subtree pointed by **`s`**. This function will be a **private** member function of the **`StudentBST`** class.

Task # 1.4 *(Max Time: 20 Minutes)*

Write a **menu-based** driver function to illustrate the working of all functions of the **`StudentBST`** class. The menu may look like as shown below:

```
1. Insert a new student
2. Search for a student
3. See the list of all students
4. Quit

Enter your choice:
```

Task # 1.5 *(Max Time: 20 Minutes)*

Now, implement the following public member function of the **`StudentBST`** class:

`bool remove (int rn)`

This function will search the BST for a student with the given **roll number** (see parameter). If such a student is found, then this function should remove the record of that student from the BST and should return **true**. If a student with the given roll number is not found, then this function should not make any changes in the tree and should return **false**. This function should not display anything on screen.

Modify the menu (**Task # 1.4**) so that user can also remove a student by specifying the roll number of the student.

Task # 1.6 *(Max Time: 20 Minutes)*

Modify the menu and implement the required member functions of **`StudentBST`** class, so that user can also see the output of **pre-order** and **post-order** traversals of the **`StudentBST`**.

❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄

Take a break (of not more than 20 minutes ☺).

Have some tea/coffee (after all, these are your winter vacations).

And then start Task # 2 ☺

❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄❄

Task # 2.1 *(Max Time: 40 Minutes)*

Implement a **BST** class to store integers (as we have already done in lectures). Class declarations will look like as shown below:

```
class BSTNode {
      friend class BST;
  private:
      int data;
      BSTNode *left, *right;
};
```

```
class BST {
  private:
      BSTNode *root;
  public:
      ...
};
```

Apart from the **default constructor** and **destructor**, the **BST** class should provide functions to **insert**, **search**, and **remove** elements (integers) from the BST. Also, write a **menu-based** driver function to illustrate the working of all functions of the **BST** class.

Task # 2.2 *(Max Time: 20 Minutes)*

Modify the menu and implement the required member functions of **BST** class, so that user can also see the output of **pre-order**, **in-order**, and **post-order** traversals of the **BST**.

Task # 2.3 *(Max Time: 15 Minutes)*

Implement the following functions of the **BST** class to **recursively** search for a particular value in the BST:

```
bool recSearch (int key);                  // public (driver function)
bool recSearch (BSTNode* b, int key);      // private (workhorse function)
```

Task # 2.4 *(Max Time: 15 Minutes)*

Implement the following functions of the **BST** class to **recursively** determine (and return) the count of the nodes present in the BST:

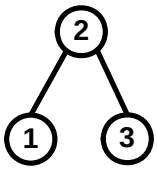
```
int countNodes ();                         // public (driver function)
int countNodes (BSTNode* b);               // private (workhorse function)
```

Task # 2.5 *(Max Time: 20 Minutes)*

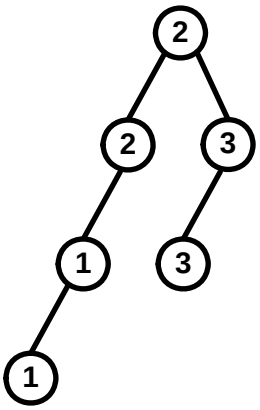
Write a function **doubleTree** (determine the exact function prototype yourself) of the **BST** class, which for each node in the BST, creates a new duplicate node, and inserts the duplicate as the left child of the original node. See the following example.

Example:

BST before calling **`doubleTree`** function:



BST after calling **`doubleTree`** function:

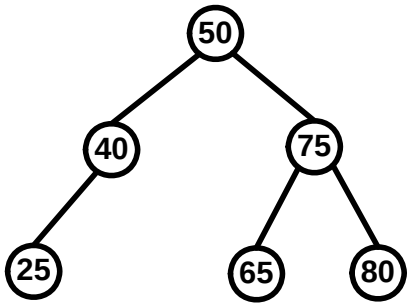


Task # 2.6 *(Max Time: 40 Minutes)*

Implement a member function of the **`BST`** class which prints **all the root-to-leaf paths** of a given BST. The prototype of your function will be:

```
void printAllPaths ();
```

For example, if we call this function on the following BST:



It should print the following paths:

```
50 -> 40 -> 25
50 -> 75 -> 65
50 -> 75 -> 80
```

Hint 1: You will need to implement a recursive helper function. Think about the prototype of this recursive helper function.

Hint 2: You need some way to communicate the list of paths from one function call to another.

Hint 3: You can assume that the maximum length of a root-to-leaf path will be 100.

Note: Do NOT use any global or static variables in your implementation.