In this lab, you will work with a *left-leaning red-black binary search tree* (LLRB-BST) to implement two functions: one to count the number of keys within a specified range and another to find the key closest to a given target value. You will implement each function in two different ways: one with a time complexity of $O(n)$ and another with a time complexity of $O(\log n)$, where n is the number of nodes in the tree. The LLRB-BST implementation is provided in the `red-black-bst.hpp` file.
As an example, consider the following sequence of insertions into an initially empty Red-Black BST:

```
RedBlackBST<int, string> tree;
tree.put(30, "thirty");    tree.put(10, "ten");
tree.put(40, "forty");     tree.put(20, "twenty");
tree.put(50, "fifty");     tree.put(35, "thirty-five");
```

Exercise 1: *Count keys in a given range* ....

Write a function to count how many keys in the tree lie within a given range `[low, high]`. Implement the following function in the provided `RedBlackBST<int,string>` class:

```
int countInRange(RedBlackBST<int, string>& tree, int low, int high);
```

(a) Implement the `countInRange1` function in $O(n)$ time: Use an in-order traversal to collect keys in sorted order, then count how many fall within the range.

(b) Implement the `countInRange2` function in $O(\log n)$ time: Use the properties of the BST to skip subtrees that cannot contain keys in the range.

Test your implementation in the `main()` function by inserting several integer–string pairs into the Red-Black BST and calling `countInRange` with different ranges. For example:

```
int count = countInRange(tree, 15, 35); // should return 3 (20, 30, and 35)
```

Exercise 2: *Closest key to a given target* ....

Write a function to find the key that is numerically closest to a given target number. Implement the following function in the provided `RedBlackBST<int,string>` class (from `red-black-bst.hpp`):

```
int findClosestKey(RedBlackBST<int, string>& tree, int target);
```

(a) Implement the `findClosestKey1` function in $O(n)$ time: Use an in-order traversal to collect keys in sorted order, then iterate through the list to find the closest key.

(b) Implement the `findClosestKey2` function in $O(\log n)$ time: Use the properties of the BST to traverse the tree and keep track of the closest key found so far.

Test your implementation in the `main()` function by inserting several integer–string pairs into the Red-Black BST and calling `findClosestKey` with different target values. For example:

```
int closest = findClosestKey(tree, 25); // should return 20 or 30
int closest2 = findClosestKey(tree, 44); // should return 40
```