Using Binary Search Trees

Step 1

EXERCISE BREAK: The third Data Structure we will discuss for implementation of a lexicon is the **Binary Search Tree**. Given that we will be doing significantly more "find" operations than "insert" or "remove" operations, which type of a **Binary Search Tree** would be the optimal choice for us *in practice*?

To solve this problem please visit https://stepik.org/lesson/31307/step/1

Step 2

In general, if we want to choose a **Binary Search Tree** from the ones we discussed in this text, it should be clear that the only viable contenders are the **AVL Tree** and the **Red-Black Tree** because they are guaranteed to have a **O(log *n*) worst-case** time complexity for all three operations (whereas the **Regular Binary Search Tree** and **Randomized Search Tree** are O(*n*) in the worst case). Note that we would prefer to use an **AVL Tree** since they have a stricter balancing requirement, which translates into faster "find" operations *in practice*.

Now that we have refreshed our memory regarding the time complexity of self-balancing **Binary Search Trees**, we can begin to discuss how to actually use them to implement the three lexicon functions we previously described.

Step 3

Below is pseudocode to implement the three operations of a lexicon using a **Binary Search Tree**. Again, we would choose to use an **AVL Tree** because of their self-balancing properties. In all three functions below, the backing **Binary Search Tree** is denoted as `tree`.

```
find(word):   // Lexicon's "find" function
    return tree.find(word) // call the backing BST's "find" function
```

```
insert(word): // Lexicon's "insert" function
    tree.insert(word)      // call the backing BST's "insert" function
```

```
remove(word): // Lexicon's "remove" function
    tree.remove(word)      // call the backing BST's "remove" function
```

STOP and Think: By storing our words in a **Binary Search Tree**, is there some efficient way for us to iterate through the words in alphabetical order?

Step 4

EXERCISE BREAK: What is the **worst-case** time complexity of the **`find`** function defined in the previous pseudocode?

To solve this problem please visit https://stepik.org/lesson/31307/step/4

Step 5

EXERCISE BREAK: What is the **worst-case** time complexity of the **`insert`** function defined in the previous pseudocode?

To solve this problem please visit https://stepik.org/lesson/31307/step/5

Step 6

EXERCISE BREAK: What is the **worst-case** time complexity of the **`remove`** function defined in the previous pseudocode?

To solve this problem please visit https://stepik.org/lesson/31307/step/6

Step 7

CODE CHALLENGE: Implementing a Lexicon Using a Binary Search Tree

In C++, the `set` container is (typically) implemented as a **Red-Black Tree**. We say "typically" because the implementation of the `set` container depends on the specific compiler/library, but most (almost all) use **Red-Black Trees**. In this code challenge, your task is to implement the three functions of the lexicon ADT described previously in C++ using the `set` container. Below is the C++ `Lexicon` class we have declared for you:

```
class Lexicon {
    public:
        set<string> tree;          // instance variable BST object
        bool find(string word);    // "find" function of Lexicon class
        void insert(string word); // "insert" function of Lexicon class
        void remove(string word); // "remove" function of Lexicon class
};
```

If you need help using the C++ `set`, be sure to look at the C++ Reference.

Sample Input:

```
N/A
```

Sample Output:

```
N/A
```

To solve this problem please visit https://stepik.org/lesson/31307/step/7

Step 8

As you can see, we improved our performance even further by using a **self-balancing Binary Search Tree** to implement our lexicon. By doing so, the **worst-case time complexity** of **finding**, **inserting**, and **removing** elements is **O(log *n*)**.

Also, because we are using a **Binary Search Tree** to store our words, if we were to perform an **in-order traversal** on the tree, we would iterate over the elements in a meaningful order: they would be in **alphabetical order**. Also, we could choose if we wanted to iterate in *ascending* alphabetical order or in *descending* alphabetical order by simply changing the order in which we recurse during the in-order traversal:

```
ascendingInOrder(node):  // Recursively iterate over the words in ascending order
    ascendingInOrder(node.leftChild)   // Recurse on left child
    output node.word                   // Visit current node
    ascendingInOrder(node.rightChild)  // Recurse on right child
```

```
descendingInOrder(node): // Recursively iterate over the words in descending order
    descendingInOrder(node.rightChild) // Recurse on right child
    output node.word                   // Visit current node
    descendingInOrder(node.leftChild)  // Recurse on left child
```

In terms of memory efficiency, a **Binary Search Tree** has exactly one node for each word, meaning the **space complexity** is **O(*n*)**, which is as good as we can get if we want to store all *n* elements.

Of course, yet again, it is impossible to satisfy a computer scientist. So, like always, we want to ask ourselves: can we go even *faster*? Note that, up until now, every approach we described had a time complexity that depended on *n*, the number of elements in the data structure. In other words, as we insert more and more words into our data structure, the three operations we described take longer and longer. In the next section, we will discuss another good approach for implementing our lexicon: the **Hash Table** (as well as the **Hash Map** to implement a dictionary.