Using Linked Lists

Step 1

The first Data Structure we will discuss for implementation of a lexicon is the **Linked List**. To refresh your memory, below are some important details regarding **Linked Lists**:

- We can choose to use a **Singly-Linked List**, in which nodes only have *forward* pointers and there is only one *head* pointer, or we can choose to a **Doubly-Linked List**, in which nodes have both *forward* and *reverse* pointers and there is both a *head* and a *tail* pointer
- To traverse through the elements of a **Linked List**, we start at one end and follow pointers until we reach our desired position in the list
- Whichever type of **Linked List** we choose to use, we will have a **O(*n*) worst-case** time complexity for "find" and "remove" operations (because we need to iterate through all *n* elements in the worst case to find our element of interest)
- Whichever type of **Linked List** we choose to use, we will have a **O(1) worst-case** time complexity to insert elements at either the front or the back of the list (assuming we have both a *head* and a *tail* pointer)
- If we want to keep our list in **alphabetical order**, whichever type of **Linked List** we choose to use, we will have a **O(*n*) worst-case** time complexity to insert elements into the list (because we need to iterate through all *n* elements in the worst case to find the right position to perform the actual insertion)

Now that we have reviewed the properties of the **Linked List**, we can begin to discuss how to actually use it to implement the three lexicon functions we previously described.

Step 2

Below is pseudocode to implement the three operations of a lexicon using a **Linked List**. In all three functions below, the backing **Linked List** is denoted as `linkedList`.

```
find(word):           // Lexicon's "find" function
    return linkedList.find(word)  // call the backing Linked List's "find" function
```

```
insertUnsorted(word): // Lexicon's "insert" function, assuming the list is unsorted
    linkedList.insertFront(word)  // call the backing Linked List's "insertFront" function
```

```
insertSorted(word):   // Lexicon's "insert" function, assuming the list is sorted
    linkedList.sortedInsert(word) // assuming the backing Linked List can do sorted insertions
```

```
remove(word):         // Lexicon's "remove" function
    linkedList.remove(word)       // call the backing Linked List's "remove" function
```

STOP and Think: In `insertUnsorted(word)`, we used the `insertFront` function of our **Linked List**. Could we have instead used the `insertBack` function?

Step 3

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

Step 4

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

To solve this problem please visit https://stepik.org/lesson/31305/step/4

Step 5

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

To solve this problem please visit https://stepik.org/lesson/31305/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/31305/step/6

Step 7

CODE CHALLENGE: Implementing a Lexicon Using a Doubly-Linked List

In C++, the `list` container is implemented as a **Doubly-Linked List**. In this code challenge, your task is to implement the three functions of the lexicon ADT described previously in C++ using the `list` container. Note that we mentioned two possible "insert" functions in the previous step (`insertSorted` and `insertUnsorted`). For this challenge, feel free to implement either one. Below is the C++ `Lexicon` class we have declared for you:

```
class Lexicon {
    public:
        list<string> linkedList;  // instance variable list 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++ `list`, 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/31305/step/7

Step 8

As you might have inferred, the **Linked List** might not be the best choice for implementing the lexicon ADT we described. To use a **Linked List**, we have two implementation choices.

We can choose to keep the elements **unsorted**. If we were to do so, the **worst-case time complexity** of **find** and **remove** operations would be **O(*n*)**, and that of **insertion** operations would be **O(1)**. However, if we were to iterate over the elements of the list, the elements would not be in any meaningful order.

Alternatively, we can choose to keep the elements **sorted**. If we were to do so, the **worst-case time complexity** of the **find** and **remove** operations would still be **O(*n*)**, but now, the **insert** operation would *also* be **O(*n*)**. However, if we were to iterate over the elements of the list, the elements *would* be 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 choosing from which end of the **Linked List**, *head* or *tail*, we wanted to begin our iteration.

In terms of memory efficiency, a **Linked List** 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.

In general, we're hoping that we have instilled enough intuition in you such that you shudder slightly every time you hear O(*n*) with regard to data structures, as we have explored so many different data structures that allow us to do better. Also, in this case specifically, recall that we explicitly mentioned that "find" was going to be our most frequently used operation by far. Therefore, even though we could potentially have O(1) "insert" operations, the fact that "find" operations are O(*n*) in this implementation approach should make you frustrated, or even angry, with us for wasting your precious time.

In the next section, we will discuss a slightly better approach for implementing our lexicon ADT: the **Array**.