Using Arrays

Step 1

The second Data Structure we will discuss for implementation of a lexicon is the **Array**. To refresh your memory, below are some important details regarding **Arrays**:

- Because all of the slots of an **Array** are allocated adjacent to one another in memory, and because every slot of an **Array** is exactly the same size, we can find the exact memory address of any slot of the **Array** in O(1) time if we know the index (i.e., we have **random access**)
- **Arrays** cannot simply be "resized," so if we were to want to insert new elements into an **Array** that is full, we would need to allocate a new **Array** of large enough size, and we would then need to copy over all elements from the old **Array** into the new one, which is a O(*n*) operation
- If an **Array** is **unsorted**, to find an element, we potentially need to iterate over all *n* elements, making this a **O(*n*)** operation in the **worst case**
- If an **Array** is **sorted**, to find an element, because of **random access**, we can perform a **Binary Search** algorithm to find the element in **O(log *n*)** in the **worst case**

Recall that we mentioned that we will very rarely be modifying our list of words. Thus, even though the time complexity to insert or remove elements in the worst case is O(*n*), because this occurs so infrequently, this slowness is effectively negligible for our purposes.

Now that we have reviewed the properties of the **Array**, 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 an **Array**. Recall that, because "find" is our most used operation by far, we want to be able to use **Binary Search** to find elements, meaning we definitely want to keep our **Array sorted** and keep all elements in the **Array adjacent** to one another (i.e., no gaps between filled cells). In all three functions below, the backing **Array** is denoted as `array`.

```
find(word):   // Lexicon's "find" function
    return array.binarySearch(word) // call the backing Array's "find" function
```

```
insert(word): // Lexicon's "insert" function, keeping the lexicon sorted
    array.sortedInsert(word)        // assuming the backing Array can do sorted insertions
```

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

STOP and Think: By making sure we do sorted insertions, we are making our insertion algorithm slower. Why would we do this instead of just allocating extra space and always adding new elements to the end of the **Array**?

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 **`insert`** function defined in the previous pseudocode?

To solve this problem please visit https://stepik.org/lesson/31306/step/4

Step 5

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/31306/step/5

Step 6

CODE CHALLENGE: Implementing a Lexicon Using an Array

In C++, the `vector` container is implemented as a **Dynamic Array**. In this code challenge, your task is to implement the three functions of the lexicon ADT described previously in C++ using the `vector` container. Note that insertions *must* keep the elements in increasing order. Also, you should use C++'s `binary_search` function in your find function. Below is the C++ `Lexicon` class we have declared for you:

```
class Lexicon {
    public:
        vector<string> array;     // instance variable array 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++ `vector`, 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/31306/step/6

Step 7

As you can see, we improved our performance by using an **Array** to implement our lexicon instead of a **Linked List**. We wanted to be able to exploit the **Binary Search** algorithm, so we forced ourselves to keep our **Array sorted**.

By keeping the elements in our **Array** sorted, we are able to use Binary Search to **find** elements, which, as you should hopefully recall, has a **worst-case time complexity** of **O(log *n*)**. However, because we need to keep our elements sorted to be able to do Binary Search, the **time complexity** of **inserting** and **removing** elements is **O(*n*)** in the **worst case**. This is because, even if we do a Binary Search to *find*

the insertion point, we might have to move over O(*n*) elements to make room for our new element in the worst case.

Also, by keeping the elements in our **Array** sorted, 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 **Array** we wanted to begin our iteration.

In terms of memory efficiency, with **Dynamic Arrays**, as we grow the **Array**, we typically double its size, meaning at any point in time, we will have allocated at most 2*n* slots, giving this approach a **space complexity** of **O(*n*)**, which is as good as we can get if we want to store all *n* elements.

We started off by saying that insert and remove operations are pretty uncommon in this specific application, so even though both operations are O(*n*) in this implementation, we find them acceptable. However, our motivation in this entire text is speed, so like always, we want to ask ourselves: can we go even *faster*? In the next section, we will discuss an even better approach for implementing our lexicon ADT: the **Binary Search Tree**.