Technique: Linked Hash Table

Level: Hard

Smallest Subarray Covering All Values: Let's say you are given a large text document Doc. You are also given a set S of words. You want to find the smallest substring of Doc that contains all the words in S.

For example:

S: ["and", "of", "one"]

Doc: "a set of words that is complete in itself, typically containing a subject and predicate, conveying a statement, question, exclamation, or command, and consisting **of** a main clause **and** sometimes **one** or more subordinate clauses"

The underlined part above is the solution. Note that the order in which the words appear doesn't matter. Also, the length of the substring is in terms of number of characters.

Questions to Clarify:
Q. Can the substring contain a word multiple times?
A. Yes

Q. Can I assume that the input is all lowercase?
A. Yes, assume that the input has all lowercase.

Q. Can I assume that S has no duplicate words?
A. Yes, you can assume that.

Q. Can I ignore punctuation?
A. Yes, ignore punctuation.

Q. Can I assume that there are no punctuations inside words, e.g, "person's" or "under-write"
A. Yes, you can assume that.

Solution:
We can directly use a Linked Hash Table to solve this problem.

Let's take a simpler example:

```
S: ["and", "of", "one"]
Doc: "one of the car and bike and one of those"
```

We iterate through the words in Doc. We keep track of the last instance of each target word encountered. Let's say we have read 3 words in Doc already:

```
Doc: "one of the"
Words encountered: one (index 0), of (index 4)
```

Next, we read "car". But since "car" is not in S, we ignore it. We then come to "and", which is in S, so we add it to our Words encountered.

```
Doc: "one of the car and"
Words encountered: one (index 0), of (index 4), and (index 15)
```

Now, we see that we have 3 words in our Words encountered. We know that this can be a candidate for the result. We find the length of the candidate. How do we do that? We look at the first and last elements in the list - index 0 and 15.

So, the substring between index 0 and index `15 + length("and")` is a candidate for the smallest substring in Doc covering all words S. We record it and keep processing Doc. We reach "bike", which we ignore, and then we reach another "and". Since we want to keep track of the latest instance of each word encountered, we update the "and" in our Words encountered.

```
Doc: "one of the car and bike and"
Words encountered: one (index 0), of (index 4), and (index 24)
```

Again, we see that there are 3 words in the Words encountered, but we see that the length of the string `(0..24+2)` is longer than the one we had before `(0..15+2)`. So our earlier candidate was better.

We keep processing, and we encounter "one". We update the value of "one", and we move it forward, because we want to keep the words in order of their index.

```
Doc: "one of the car and bike and one"
Words encountered: of (index 4), and (index 24), one (index 28)
```

Again, since we have all 3 words of S, we check the length of the substring. It is `[4..28+2]`, which is again longer than our current result `(0..15+2)`. So we carry on. We encounter another "of". We update the list.

```
Doc: "one of the car and bike and one of"
Words encountered: and (index 24), one (index 28), of (index 32)
```

This time, the length of the substring is `[24..32+1]`, which is better than our earlier result `(0..15+2)`. So, we set this as our new result. We process "those", which doesn't belong to S, and we are done. The smallest substring covering the Doc is `[24..33]`.

Here are the things we did while maintaining the "Words encountered" list:
1. If we encounter a word that is not in S, ignore it.
2. If we encounter a word in S, add it to the front of the list along with its index. Delete any previous instance.
3. At any time, there is only one instance of a word in the list
4. After adding a word, if the list is full, i.e, if there are as many words as in S, we have a

candidate substring that contains all words in S. We check the length of the substring by comparing the indices of the first and last words in the list.

The core idea behind the algorithm is that we want to keep the last instance of each word encountered.

Linked Hash tables fits it well.We want to keep the words in order, while having the flexibility of deleting any word in O(1) time. The algorithm is given below.

Pseudocode:

(Note: Never write pseudocode in an actual interview. Unless you're writing a few lines quickly to plan out your solution. Your actual solution should be in a real language and use good syntax.)

```
class Node  // doubly linked list node
     string word
     int index
     Node next
     Node prev

     // getters and setters

function getShortestSubString(String doc, Set<String> S):

  hash table<word -> Node> = empty
  linked list<Node> = empty
  string result = empty

  for word, index in doc:
     if word not in S
         continue

     // delete any existing node with the word
     if word exists in hash table
         node = hash table.get(word)
         delete node from linked list

     // add new node with the word
     newNode = Node(word, index)
     linked list.addToFront(newNode)
     hash table.put(word, newNode) // update hash table

     // new node added, check if it makes a better result
     if (hash table.size() == S.size()) {
         stringEnd = linked list.tail.index + list.word.length - 1
         stringStart = linked list.head.index
         // check if current candidate is shorter than result
         if (stringEnd - stringStart + 1 < result.length) {
```

```
                result = doc.substring(stringStart, stingEnd)
            }
        }

    return result
```

Test Cases:
Edge Cases: Empty S, Empty Doc, No S in Doc
Base Cases: One word from S in Doc, Doc only has one instance of words in S
Regular Cases: Doc has multiple instances of words from S

Time Complexity: O(Doc.charCount)

We iterate through the entire doc, and we do O(1) work at each step.

Space Complexity: O(Doc.charCount)
In the worst case, the result might contain the entire Doc.

```
public String getShortestSubString(String doc, HashSet<String> wordSet) {
    String result = null;

    HashMap<String, Node> nodeMap = new HashMap<>();
    LinkedList linkedList = new LinkedList();

    /*
     * We move any complicated string processing in a WordIterator.
     * This lets us focus on the core algorithm instead of spending time on
     * handling string corner cases.
     * If the interviewer asks you to implement it, you can do that later.
     * In most cases, the interviewer will not ask you to implement it.
     */
    WordIterator iter = new WordIterator(doc);
    while (iter.hasNext()) {
        WordIndex wordIndex = iter.next();
        String word = wordIndex.getWord();
        if (!wordSet.contains(word))
            continue;

        if (nodeMap.containsKey(word)) {
            Node toDelete = nodeMap.get(word);
            linkedList.delete(toDelete);
        }

        Node newNode = new Node(word, wordIndex.getIndex());
        linkedList.append(newNode);
        nodeMap.put(word, newNode);

        if (nodeMap.size() == wordSet.size()) {
```

```
            int startIndex = linkedList.head.getIndex();
            int endIndex = linkedList.tail.getIndex()
                                    + linkedList.tail.getWord().length() - 1;
            if (result == null || (endIndex - startIndex + 1) < result.length()) {
                result = doc.substring(startIndex, endIndex + 1);
            }
        }
    }

    return result;
}


/*
 * Helper Code: Ask the interviewer if they want you to implement these. Most
 * interviewers won't ask you to implement it.
 */

private class WordIndex {
    String word;
    int index;

    public WordIndex(String word, int index) {
        super();
        this.word = word;
        this.index = index;
    }

    public String getWord() {
        return word;
    }

    public int getIndex() {
        return index;
    }
}


/*
 * In this iterator, the position variable will always be at the start of the next
 * word in the string.
 */
public class WordIterator implements Iterator<WordIndex>{
    String str;
    int position;

    public WordIterator(String str) {
        this.str = str.trim(); // eliminate trailing whitespaces
        this.position = 0;
```

```
        // advance position to next alphabet in str
        advanceToNextAlphabet();
    }

    /*
     * Advances the position variable to the next alphabet
     */
    private void advanceToNextAlphabet() {
        while(position < str.length()
                    && !Character.isAlphabetic(str.charAt(position))) {
            position++;
        }
    }

    @Override
    public boolean hasNext() {
        return position < str.length();
    }

    @Override
    public WordIndex next() {
        if (!hasNext())
            return null;

        int wordStartIndex = position;
        // advance position to next non-alphabet
        while(position < str.length()
                        && Character.isAlphabetic(str.charAt(position))) {
            position++;
        }
        int wordEndIndex = position - 1;

        advanceToNextAlphabet();

        return new WordIndex(
                 str.substring(wordStartIndex, wordEndIndex + 1), wordStartIndex);
    }
}

public class LinkedList {
    Node head;
    Node tail;

    public LinkedList() {
        head = null;
        tail = null;
    }

    /*
     * This is the Append Function technique that we learned earlier.
```

```
     * In this case, we add a line - "toAdd.setPrev(tail);" - since this is a doubly
     * linked list.
     */
    public void append(Node toAdd) {
        if (head == null) {
            head = toAdd;
        } else {
            tail.setNext(toAdd);
            toAdd.setPrev(tail);
        }
        tail = toAdd;
    }

    /*
     * Deletes a Node from a doubly linked list.
     */
    public void delete(Node toDelete) {
        if (toDelete == null)
            return;

        if (toDelete.getPrev() != null)
            toDelete.getPrev().setNext(toDelete.getNext());

        if (toDelete.getNext() != null)
            toDelete.getNext().setPrev(toDelete.getPrev());

        if (toDelete == tail)
            tail = toDelete.getPrev();

        if (toDelete == head)
            head = toDelete.getNext();
    }
}

// Node that stores Key and Value. It is a doubly linked list node,
// so it stores next and previous node.
public class Node {
    Node next;
    Node prev;
    String word;
    int index;

    public Node(String word, int index) {
        super();
        this.word = word;
        this.index = index;
    }

    public Node getNext() {
        return next;
```

```
    }

    public void setNext(Node next) {
        this.next = next;
    }

    public Node getPrev() {
        return prev;
    }

    public void setPrev(Node prev) {
        this.prev = prev;
    }

    public String getWord() {
        return word;
    }

    public int getIndex() {
        return index;
    }

}
```