Question:

The following paragraph excerpts from Thinking in Java, 2nd edition, Revision 12, 2000 by Bruce Eckel.

But computers are not so much machines as they are mind amplification tools ("bicycles for the mind," as Steve Jobs is fond of saying) and a different kind of expressive medium. As a result, the tools are beginning to look less like machines and more like parts of our minds, and also like other forms of expression such as writing, painting, sculpture, animation, and filmmaking.

(a). First, save that paragraph into a 1-D character array, **tmp**.

```
Char tmp[1000] = "But computers are …"
```

Then, use the structure definition to define a new type, **keyWord**. There are two members in a variable of type **keyWord**. One is a character array with 50 elements, and the other is a variable of type **int**. The code segment defining the new type, **keyWord**, is the following:

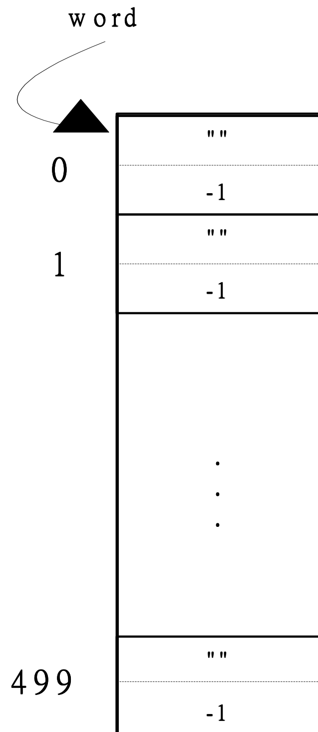
```
struct keyword{
    char str[50];
    int index;
};

typedef struct keyword keyWord;
```

After you define the **keyWord** type, allocate an array, **word**, to hold the whole information of the paragraph saved in 1-D array, **tmp**.

```
keyWord word[500];
```

Next, initialize each character array of **keyWord** elements as an empty string and each integer member, index, of them as –1. The result **word** array is shown in the following figure:



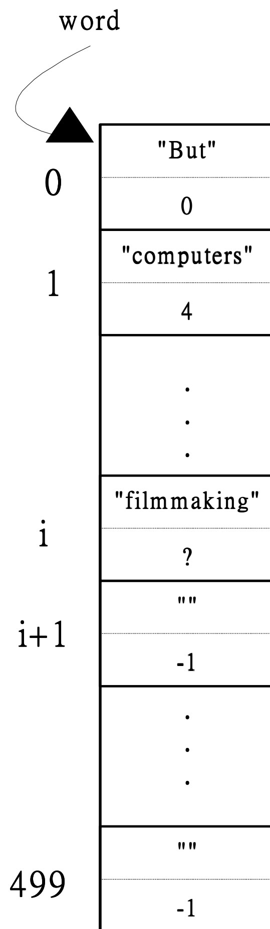
Please save the given 1-D array, **tmp**, into that data structure, **word**. Here are the rules to tell you how to do:

(1). For the k-th word, **s1**, in that 1-D array, **tmp**. (The occurrence of a word in the **tmp** may be multiple. For each occurrence, you save the word into the **word** regardless of how many times you save it into the **wrod**.)

```
word[0].str = s1
word[0].index = starting_index_of(s1)
```

(2). All punctuation marks and blank characters you run into are neglecting in the saving process.

After assign all the words in the **tmp** into the **word**, the value contained in the **word** is like the following:



In the above figure, the last word saved into the **`word`** array is **`filmmaking`**, whose index in the **`word`** array is **`i`**.

The final program must print all the elements in the **`word`** array. The format is like the following:

```
Before sorting:
                str                 index
-------------------------------------
word[0]:        But                   0
word[1]:        computers             4
           .
           .
           .
word[i]:        filmmaking            *
```

In the above sample output, the last word saved into the **word** array is **filmmaking**, whose index in the **word** array is **i**. And "*****" means the index of the starting character of the word, **filmmaking**, in the **tmp** array.

(b) Using the **bubble sort** (in p220 of textbook)to sort the **word** array in Part a. In other words, after the program finishes the sorting process:

`word[0]) < word[1] < … < word[i]`

First, we define how to compare two given **keyWord** elements:

(1). When you get two **keyWord** elements (ex: **word[j], word[k]**), compare the **str** member of them (i.e.: **word[j].str, word[l].str**). View all uppercase characters in a word as the corresponding lowercase characters. In other words, the word comparison is case-insensitive.

Ex:

Strings, such as "IS", "Is" or "iS", are all the same as "is".

Next, use the function, **strcmp**, to determine the order of these two character arrays (i.e.: strings). There are three possible scenarios. We list them along with the actions we take.

(1.1) if word[j].str > word[k].str, then word[j] > word[k]

(1.2) if word[j].str < word[k].str, then word[j] < word[k]

(1.3) if word[j].str = word[k].str, then repeat step (2)

(2). If **word[j].str = word[k].str**, then extract each **index** member (i.e.: word[j].index, word[k].index) of those two **keyWord** elements. Compare these two integers to determine the order of **word[j]** and **word[k]**. There are two possible scenarios.

(2.1) if word[j].index > word[k].index, then word[j] > word[k]

(2.2) if word[j].index < word[k].index, then word[j] < word[k]

Finally, modify the function, print the sorted **word** array on the screen.

```
After sorting:
           str                 index
-----------------------------------
word[0]:    a                    ?
```

```
Word[1]:    a                  ?
word[2]:    amplification       ?
             .
             .
             .
word[i]:    writing            ?
```

(c). Please write a program. Let a user inputs an English word, and the program will use binary search (in p224 of textbook) to find out possible **keyWord** element, which contains the character array the same as input word.

The final output on the screen is the following:

```
Enter string search key: tool
Value not found.

Enter string search key: Are

            str              index
------------------------------------
word[?]    are              ?
word[?]    are              ?
                 .
                 .
                 .
word[?]    are              ?
```