Lab Practice Problem 10

Filename: `practice10_surname.c`

In this activity, you will complete a partially implemented C program that reads words from a file, counts the number of times each distinct word appears, and prints the results. This exercise is intended to reinforce your skills in string processing, file input, structures, and function implementation.

Try It Out

Write a program that reads up to 2,000 words from an input file, converts each word to lowercase, and counts how many times each distinct word appears. Afterward, sort the words by frequency (highest first) and alphabetically for any ties. Finally, print each word and its frequency on its own line.

Assumption

You may assume each word is at most 100 characters long and consists only of English letters. When determining matches, treat uppercase and lowercase letters as equivalent.

Your task is to complete the given C program. You can access the starter code here. The following function prototypes describe the functions you will implement:

`WordFreq create_word(const char *word);`
: Creates a new `WordFreq` struct by copying the string into the `word` field and initializing its frequency to 1. The struct is returned by value.

`WordList create_list(int capacity);`
: Initializes an empty `WordList` with the given capacity. The list is returned by value.

`WordFreq *get_word(WordList *list, const char *query);`
: Searches the list for a word matching `query`. If found, returns a pointer to the corresponding `WordFreq` inside the list. Otherwise, returns `NULL`.

`void increment_count(WordFreq *word);`
: Given a pointer to a `WordFreq`, increases its frequency by 1.

`void add_word(WordList *list, const char *word);`
: Creates a new `WordFreq` using the given string and appends it to the list, provided the logical size has not yet reached `capacity`. Updates the list's `size` accordingly.

`void process_word(WordList *list, char *line);`
: Converts the input word to lowercase, checks whether it already exists in the list, and either increments its frequency or inserts a new entry.

`void print_list(const WordList *list);`
: Prints each entry in the list on its own line using the format: `frequency word`

`int populate_list(WordList *list, const char *filename);`
: Opens the specified file, reads the number of words to process, and processes each one in order. Returns `1` on success or `0` if the file could not be opened.

`void selection_sort(WordList *list);`

Sorts the list using a multi-criteria selection sort. Words are ordered in descending frequency, with alphabetical order used to break ties.

Important You must not modify any other parts of the program. Your only task is to complete the definitions of the nine functions listed above. Do not add new helper functions; use only the helper functions already provided if needed. Failure to follow these restrictions will result in point deductions.

Input

The program reads input from a file named `words.txt`. The first line contains an integer $0 < \mathrm{N} \leq 2{,}000$, which specifies how many lines the program should process from the file. Each of these lines contains exactly one word. Words may appear in any mix of uppercase and lowercase English letters only. You may assume that no word exceeds 100 characters.

Output

The program prints each distinct word from the file, converted to lowercase, along with the total number of times it appears. The output is sorted by frequency in descending order, with alphabetical order used to break ties. Each line displays the frequency, followed by a single space, then the word.

words.txt

```
10
kiwi
KIWI
BANANA
banana
apple
Orange
apple
Cantaloupe
cantaloupe
kiwi
DoNotReadThisLine
```

Sample Output

```
3 kiwi
2 apple
2 banana
2 cantaloupe
1 orange
```

Guide Questions

1. What is the benefit of using a structure like `WordFreq` to store a word and its frequency together, rather than keeping two separate parallel arrays?
2. What is the purpose of the `size` field inside the `WordList` struct? How does it help distinguish between the logical number of stored elements and the physical capacity of the array?
3. In this program, `get_word()` returns a pointer to an element inside the `WordList`. What are the advantages of returning a pointer rather than returning a copy of the struct?
4. How would the program change if `get_word()` returned an index instead of a pointer? What additional work would be required in that case?
5. The function `get_word()` performs a case-sensitive comparison. This means it assumes that the `query` string has already been normalized (converted to lowercase). What limitations arise if a non-normalized word is passed to `get_word()`? How would this affect the correctness of the word frequencies? Should `get_word()` itself be responsible for normalizing the input, or is it better for the calling function (e.g., `process_word()`) to enforce this requirement?
6. In `selection_sort()`, what would happen if the algorithm compared the words alphabetically before comparing their frequencies? How would that alter the sorted output? When implementing a multi-criteria sort, how do you ensure that the primary criterion is evaluated first and that ties are correctly broken using the secondary criterion? How should the `if` conditions be structured to enforce this ordering?
7. Several function parameters in this program use the `const` keyword (e.g., `const char *word` or `const WordList *list`). What does `const` protect in each of these cases, and why is this protection useful? How does using `const` help prevent accidental modification of data?
8. **Optional.** The `const` keyword can appear in different positions when working with pointers (for example: `const T *p`, `T * const p`, and `const T * const p`). What does each pattern mean, and how do they differ?

Sample Solution

The solution will be released after the submission window has closed. You may review it on the activity page at that time.