Computer Science I Program 6: Wordle Candidate Ranking (Heaps)

Please Check Webcourses for the Due Date

Please read the whole assignment before you start coding

Objective

Give practice with implementing heaps
Give practice with structs, functions, and dynamic memory in C

Wordle Candidate Ranking

Every day, millions of people play the New York Times game Wordle. Behind the scenes, Wordle relies on a large collection of five-letter words and a clever system that evaluates guesses, filters possibilities, and ranks the most promising answers. The developers now want to build a smarter internal tool that can automatically sort through all valid words and suggest the strongest candidates based on previously played guesses.

That's where you come in!

Now that you've learned about strings, arrays, and heaps, you have the tools to design a system that can intelligently process Wordle clues. Your program will be responsible for analyzing a dictionary of words, interpreting the feedback from multiple guesses (green, yellow, gray), and identifying which words could still be the hidden answer. Once these possible words are found, you will assign each one a score based on how frequently its letters appear across the entire dictionary. The idea is that words containing more common letters are generally stronger guesses.

Your final task is to store these `{word, score}` pairs in a max-heap, which always keeps the best-scoring candidate at the top. Using heap operations, your program will repeatedly extract words in descending order of likelihood, producing a neatly sorted list of suggested solutions.

Your Wordle-ranking tool will follow these major steps:

1. Filter the dictionary to keep only words that match every clue from the user's guesses.
2. Compute letter frequencies across the full dictionary and give each candidate word a score based on those frequencies.
3. Build a max-heap of `{word, score}` structures using `heapify`.
4. Repeatedly remove the maximum (heapsort) to print all candidates from highest score to lowest.

By completing this assignment, you will create a mini Wordle-solver engine that mirrors the logic used by real puzzle designers.

To build your Wordle ranking tool, you will rely on two main structures: one to represent each potential Wordle candidate, and one to manage the max-heap your program will use to prioritize those candidates.

Each filtered dictionary word will be packaged into an `Entry` object that stores both the word itself and its score as the sum of frequencies of each of the letters in the word

```
typedef struct Entry{
    char word[6];
    int score;
} Entry;


typedef struct HeapStruct {
    Entry* heaparray;
    int capacity;
    int size;
} HeapStruct;
```

Implementation Requirements/Run Time Requirements

In this program you will build a tool that analyzes a list of Wordle guesses and their feedback (G = green, Y = yellow, B = gray) and uses this information to determine which dictionary words are still possible answers. You will then score these remaining words and rank them using a max-heap, producing a prioritized list of candidate solutions.

In this assignment, the Wordle logic is slightly simplified so that you can focus on filtering and heap operations without dealing with the full complexity of the real game. Use the following rules:

1. **Green (G)**: The letter is correct and must appear in that exact position.
2. **Yellow (Y):** The letter must appear somewhere in the word, but not at that position.
3. **Gray (B)**: The letter does not appear anywhere in the word, unless the same letter in a different position received **Y** as feedback.

The way in which Wordle provides its feedback (and the algorithm you should implement) is as follows:

1. First go through and mark all correct letters in each slot with **G**. Now, take these letters and positions out of consideration for the rest of the algorithm.
2. For the remaining letters in the guess, go through these letters, from left to right. See if this letter is in the remaining letters in the correct answer. If it is, mark this slot at **Y**, then mark that letter in the correct answer as "used", meaning that it can't be matched again with a different letter in the guess.

Notice that the effect of the algorithm given in step 2 is that if a letter appears in the guess incorrectly in more places than it appears in the correct word, then leftmost slots with the letter in the guess will be marked with **Y** while the others will be marked with **B**.

Here is an example that should fully clarify how Wordle gives its feedback:

```
Correct:  BABBA
Guess:    AAABB
Feedback: YGBGY
```

In particular, this guess would never get the feedback BGYGY, because the A in index 0 of the guess gets processed before the A in index 2.

Letter Frequency Table and Scoring

To compute the score of each candidate word, your program will use a provided letter-frequency table**.** This table represents how often each letter (a–z) appears within a large Wordle-style corpus. The array `freq[26]` stores these values, where `freq[0]` corresponds to `'a'`, `freq[1]` corresponds to `'b'`, and so on. You must use this table exactly as provided and must not modify its values. Letters with higher frequencies contribute more to a word's score.

```
const int freq[26] = {
    /* a  b  c  d  e  f  g  h  i  j  k  l  m */
       8, 2, 3, 4, 13,3, 2, 6, 7, 0, 1, 4, 2,
    /* n  o  p  q  r  s  t  u  v  w  x  y  z */
       7, 7, 2, 0, 6, 6, 9, 3, 1, 2, 0, 2, 0
};
```

Each candidate word receives a score equal to the sum of the frequencies of its letters, counting repeated letters multiple times. Formally, for a 5-letter word `word`, the score is defined as:

$$\text{score(word)} = \sum_{i=0}^{4} \text{freq}[\, word[i] - 'a' \,]$$

For example, the word `"serve"` receives the score:

```
score = freq['s'] + freq['e'] + freq['r'] + freq['v'] + freq['e']
```

These computed scores determine the ordering inside the max-heap: words with higher scores are considered stronger candidates and must appear earlier in the output.

Memory Requirements

Your program must manage memory responsibly throughout its execution. Specifically:

- The dictionary must be allocated dynamically based on the number of words provided in the input.
- The heap array must also be allocated dynamically, sized according to the number of candidate entries that pass the filtering stage.
- All dynamically allocated memory must be freed before the program terminates, including the dictionary, heap array, and any auxiliary structures you create.
- Do not create or store unnecessary extra copies of the dictionary or candidate words; use only the memory required for the assignment

Run time Requirement:

Your program should comfortably handle dictionaries with tens of thousands of words**.**
The intended complexity is:

- Filtering: $O(n)$
- Scoring: $O(k)$
- Heapify: $O(k)$
- Heap Sort: $O(k\log\ k)$

where ***n*** is the total number of words and ***k*** is the number of words that passed filtering.

Input

The first line of input contains a single positive integer, ***n*** ($1 \leq n \leq 10^6$), representing the number of words in the dictionary.

Each of the next ***n*** lines represents words in the dictionary where each one of them has exactly 5 lowercase letters.

The next input section begins with an integer ***m*** ($1 \leq m \leq 5$), representing the number of guess/feedback pairs. For each of the ***m*** guesses, two lines follow: the guessed word (five lowercase letters) and a feedback string of length five made up of the characters G (green), Y (yellow), and B (gray)**.**

Output

Once the heap has been constructed and all candidate entries have been inserted, your program will repeatedly remove the maximum entry and print it. If two or more words receive the same numeric score, ties must be broken alphabetically (i.e., pick the one that comes first in alphabetical order).

The output will list all valid candidate words in descending score order, using this format:

```
<score1> <word1>
<score2> <word2>
...
```
$<score_n>$ $<word_n>$

If no dictionary words satisfy all constraints, the program must print

```
No candidates found.
```

Sample Input	Sample Output
10 cigar serve evade focal blush crane sassy about delta smile 2 caffe BBBBG sunny GBBBB	39 serve 32 smile

Sample Input	Sample Output
5 stare rates tears smile crane 1 xxxxx BBBBB	42 rates 42 stare 42 tears 37 crane 32 smile

Deliverables

Please submit a source file `wordleheap.c`, via Webcourses, for your solution to the problem.