CS50 Spell Checker PSet WalkThrough

Welcome to the Spell Checker Pset

CS50: Implementing a Spell Checker using Hash Tables

👋 Welcome, CS50 learners!\ Today we'll walk through the full solution to one of your trickiest psets — `speller`.

We'll cover:

1. What each function in `dictionary.c` does
2. How the hash table is set up
3. How to optimize for time and memory

What's the Goal?

Build a spell checker using a hash table

You need to implement these 5 functions in `dictionary.c`:

1. `load` – Load dictionary words into memory
2. `check` – Check if a word exists
3. `hash` – Map each word to a bucket
4. `size` – Return number of words loaded
5. `unload` – Free all used memory

All for speed and no memory leaks

Understanding the Data Structure

🔗Hash Table Structure

We use:

```
typedef struct node {
    char word[LENGTH + 1];
```

```
    struct node *next;
} node;
```

- An **array of 20,000 linked lists is used instead of 26 to improve speed (faster lookups) although it sacrifices memory.**\ Feel free to tweak it

```
const unsigned int N = 20000;
node *table[N];
```

Each word is hashed to an index `i`, and inserted into `table[i]`\ Multiple words in the same bucket → Linked list!

Hash Function Explained

`hash(const char *word)`

```
unsigned int hash(const char *word) {
    unsigned int roll_sum = 0;
    for (int i = 0; i < strlen(word); i++) {
        unsigned int squared = pow(toupper(word[i]), 2);
        if (i == round(strlen(word) / 2))
            roll_sum += round(sqrt(roll_sum)) + 17;

        roll_sum += squared + 47;
    }
    return roll_sum % N;
}
```

1. Spreads words more evenly
2. Prevents collisions
3. Works for all cases (upper/lower)

Loading the Dictionary

`bool load(const char *dictionary)`

```
FILE *dict_open = fopen(dictionary, "r");
if (dict_open == NULL) return false;

char buffer[LENGTH + 1];
while (fscanf(dict_open, "%s", buffer) != EOF) {
```

```
    node *n = malloc(sizeof(node));
    if (n == NULL) return false;

    strcpy(n->word, buffer);
    int hash_index = hash(buffer);

    n->next = table[hash_index];
    table[hash_index] = n;
    count_words++;
}
fclose(dict_open);
return true;
```

Key Notes:

1. Memory allocated for each word
2. Insert at head of list for efficiency
3. Count each word for `size()`

Checking for a Word

`bool check(const char *word)`

```
int word_index = hash(word);
node *cursor = table[word_index];

while (cursor != NULL) {
    if (strcasecmp(word, cursor->word) == 0)
        return true;
    cursor = cursor->next;
}
return false;
```

1. Case-insensitive check
2. Traverse linked list
3. Return `false` if not found

Counting Words

`unsigned int size(void)`

```
return count_words;
```

1. Simply return the global counter
2. Updated during load
3. Fast O(1) time complexity!

Freeing Memory

`bool unload(void)`

```
for (int i = 0; i < N; i++) {
    node *temp = table[i];
    node *cursor = table[i];

    while (temp != NULL) {
        cursor = cursor->next;
        free(temp);
        temp = cursor;
    }
}
return true;
```

1. Free every node in every bucket
2. Traverse all linked lists
3. Prevent memory leaks

Summary

What we've done

1. Efficiently stored words using a hash table
2. Implemented key dictionary operations
3. Avoided memory leaks with careful `malloc` / `free`
4. Leveraged C pointers and structs

Now you're ready to:

1. Solve `speller`

2. Build real-time spell-checkers!

Happy Coding Projectstakers!