Assignment 3: Spell Checker
Programming report

s4296850 & s4740556

Algorithms and Data Structures 2022

1 Problem description

General: The program checks if the words in a text are present in a certain dictionary. First the dictionary is inputted and then the text to be checked. The program all words which were not found in the dictionary as well as the number of such words.

Input-output behavior: Input: Words of the dictionary followed by a single exclamation mark. Then next input is text from which words are extracted and then checked whether they are in the dictionary.
Output: The words from the text which are not in the dictionary in the text followed by the number of such words.

Example input:

```
one
two
three
and
four
!
hello?
one,twwo; three!
And - "for".
```

Example output:

```
hello
twwo
for
3
```

2 Problem analysis

First, we have to same all the words from the first part of the input into the dictionary. Then we tackle the text. The words in the text are separated by non-alphabetical characters or spaces. The text has to be separated into words that have to be checked in the dictionary.

The main problem is the size of the dictionary and the time complexity needed to check whether a given word exists in the dictionary. The skeleton program implements a solution that stores all words in the dictionary individually as strings and checks every word from the text by comparing it to each of the stored words. This is, however, inefficient for large dictionaries. To make this more efficient in terms of memory size and search time we implemented the trie structure. In this structure, all words are saved character by character in something like tree structure, each letter branching to all possible successive letters given the previous letters, as shown in Diagram 2. This format allows for much more efficient storage and better performance. It solves the two subproblems we have to solve; storing the dictionary and searching through it.

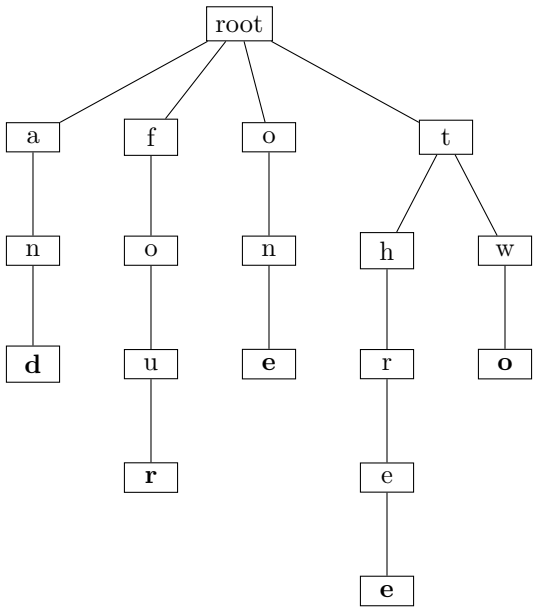

Diagram 1: trie structure for example input

3 Program design

Design choice. The program consists of three parts, the speller, dictionary and trie. This divides the structure of the process into three layers. The speller reads the input and inputs it into a structure called a dictionary. Dictionary includes the first node of trie inside of itself. The whole trie consists of nodes that correspond to letters of the input word.

Speller. Dictionary is created. Using a while loop, words are read as a string, set to all lowercase characters and added to the dictionary, until an exclamation mark is read. Then the input is read character by character until a non-alpha-numerical character is read. Then the presence of a word is checked within the dictionary. If the word is not present, it is printed to the output and the counter is increased. In the end, the counter is printed.

Dictionary. This structure includes a pointer to a trie. We can check whether we have a word in a dictionary by going through individual letters and going down the trie structure is the node with a given letter exists. If we come to the end of the word and the node which we are at is an end node, the word is in the dictionary, otherwise, it is not. We can add a word to the dictionary by going down the structure, in the same way, we checked it but if find that a certain node does exist we add it to the trie structure.

Trie. This structure includes has an indicator whether it is the end of the word, the array of pointers to possible next letters and an array of bools which are set to true when a corresponding pointer to a Trie is initialiseds. We can use the function getChild if we want to go check whether the trie has a certain node (used when checking for a word in a dictionary) and function addOrGetChild to look if we have a certain node and add it if we do not (when adding words to the dictionary).

Time Complexity. At first, the program stored all words as strings and had to check every new word against all stored words to determine whether a word is included. As for length d of dictionary and t words in the input text this had a complexity of $\mathcal{O}(t * d)$. But with the trie structure, each word is stored in constant time according to its size. Creating dictionary is d complexity. Checking each word is also a constant time, overall t complexity. Therefore the complexity is $\mathcal{O}(t + d)$.

4 Evaluation of the program

While developing the program, we were testing the performance on the example files given. Once the program was functional we used Valgrind to identify and resolve memory leaks, allocations of incorrect size and accesses to uninitialized memory.

The final program was accepted by Themis, it passed all the test cases. It was tested for memory leaks by Valgrind and there are none. On the example file, Valgrind generated the following output:

```
...$ valgrind ./speller <  exampleInput.txt
==3896== Memcheck, a memory error detector
```

```
==3896== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3896== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==3896== Command: ./speller
==3896==
hello
twwo
for
3
==3896==
==3896== HEAP SUMMARY:
==3896==     in use at exit: 0 bytes in 0 blocks
==3896==   total heap usage: 57 allocs, 57 frees, 11,176 bytes allocated
==3896==
==3896== All heap blocks were freed -- no leaks are possible
==3896==
==3896== For lists of detected and suppressed errors, rerun with: -s
==3896== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5 Process description

The part of the program for separating the individual words from the input was partially provided and the solution was straightforward. This part, concerning speller.c, was not complicated. We were able to finish this part during the lab. In this version of the program, the dictionary was based on an array of strings. Then we started implementing tries and all our troubles and misery began.

We decided to implement a trie separate from the dictionary. Knowing the general structure from lectures we worked together by implementing various functions of the trie. Then we implemented it into the dictionary. By the end of the second hour our the lab session, the program looked as if it could work. The speller was calling the dictionary which in turn was repeatedly calling the trie to add a character to the structure or check whether it is already there. We were nearly ready to celebrate but we encountered an unfortunate problem; a segmentation fault.

The problem or multiple problems were somewhere in the code but because of lack of testing, we could not pinpoint their origin. As the Lab ended we stopped working on the program but weren't able to stop thinking about it. This very evening Michal, after hours of coding, achieved a significant advancement. Our code was not only compiling but also passed the first 5 test cases on Themis. What stopped us from success was Valgrind which in its ultimate wisdom found hundreds of errors. Although everything was freed as it should be, the program was still far from perfect.

What followed were hours and hours of debugging and trying to find the culprit of the errors. Unused to reading Valgrind error messages, we were left hopeless. The code looked like a crime scene. Every section was separated and analysed bit by bit. Commented parts of code and print statements were found all over the place.

As we found out, our program relied on accessing uninitialized memory to eliminate whether a certain trie node exists. We introduced a new array of hasChildren to the trie, which indicated whether a certain part of memory is allocated and can be safely accessed. We desperately stitched it to our existing code. After a long fight, the monstrosity we created was finally alive. It finally passed the last three cases on Themis. To our surprise, though, in the process, the program lost its ability to pass the cases it passed previously. Fortunately, after some more time, we managed to solve that problem as well. Changing the memory allocation of Tries from 'struct TrieNode' to 'Trie' solved the problem. We popped a bottle of champagne. It was a great victory for the team.

Then we just had to clean the mess left after the debugging and then we started on this beautiful report. We worked in parallel using a real-time collaborative environment available on `www.replit.com`. It is therefore hard to pinpoint who did what. The general structure was written by both of us. Then, although we worked separately at home, we exchanged ideas and frustrations. Together, we achieved this amazing goal.

6 Conclusions

The program solved the problem in an efficient manner. The amount of data stored in the program using a trie it little memory and processing power, as compared to the provided solution which used an array. The time complexity was previously described. Due to the use of tries checking whether a word is in a dictionary related

only to the length of the word and not the number of words in the program. With the use of tries, each new word can take as little as a single node on the trie structure (assuming a similar word already exists in the dictionary).

7 Appendix: program text

Listing 1: speller.c

```c
/* name: speller.c
 *
 * author: Michal Tesnar (m.tesnar@student.rug.nl, s4740556)
 * author: Wojciech Trejter (w.trejter@student.rug.nl, s4296850)
 *
 * date: 02.03.2022
 *
 * description: This program reads in a dictionary terminated by '!'.
 * Then it reads input and counts how many words are not in the
 * dictionary and prints them.
 */

#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include "dictionary.h"


// remove non-alphabetic characters and convert to lower case
void trimWord(char *word) {
  int k = 0;
  for (int i = 0; i < (int) strlen(word); i++) {
    if (isalpha(word[i])) {
      word[k] = tolower(word[i]);
      k++;
    }
  }
  word[k] = '\0';
}

int main(int argc, char *argv[]) {
  char word[LENGTH + 1] = "";
  // read in the dictionary
  dict *dictionary = newEmptyDict();
  while (scanf("%45s", word) && word[0] != '!') {
    trimWord(word);
    addWord(word, dictionary);
  }
  getchar(); // removing the '!' at the end of dictionary
  int counter = 0; // number of unknown words
  int index = 0; // index in the string
  int c = EOF;
  while ((c = getchar()) && c != EOF) { // while there are characters on the
      line

    if (isalpha(c)) { //is c a letter of alphabet? Add it to the word
      word[index] = tolower(c); // convert and add to the word
      word[index + 1] = '\0'; // always add '\0' at the end so that each
         word ends with it
      index++;
    } else {
      if (word[0] != '\0' && !check(word, dictionary)) { // check if the
         word is in the dictionary if it is not empty
        counter++; // count unknown words
```

```
52          printf("%s\n", word);
53        }
54        // reset reading of the word
55        word[0] = '\0';
56        index = 0;
57      }
58    }
59    // print number of unknown words
60    printf("%d\n", counter);
61    freeDict(dictionary);
62    return 0;
63  }
```

Listing 2: dictionary.c

```
1  /* name: dictionary.c
2   *
3   * author: Michal Tesnar (m.tesnar@student.rug.nl, s4740556)
4   * author: Wojciech Trejter (w.trejter@student.rug.nl, s4296850)
5   *
6   * date: 06.03.2022
7   *
8   * description: The dictionary is a functional implementation for the tries.
        It can add words or check whether words are already in the dictionary
9   */
10
11 // implements a dictionary
12
13 #include <stdbool.h>
14 #include <stdlib.h>
15 #include <string.h>
16 #include "dictionary.h"
17
18 dict *newEmptyDict() {
19   dict *d = malloc(sizeof(dict));
20   if (d == NULL) {
21     return NULL;
22   }
23   d->root = newTrie();
24   return d;
25 }
26
27 // add word to the dictionary if it is is not already known
28 void addWord(char word[LENGTH + 1], dict *d) {
29   int index = 0;
30   Trie dest = d->root; //the root of the trie
31   while (word[index] != '\0') {
32     dest = addOrGetChild(dest, word[index]); //add / go through the node in
          the trie
33     index++;
34   }
35   dest->endNode = 1;
36 }
37
38 // check whether word is in dictionary
39 bool check(const char *word, dict *d) {
40   int index = 0;
41   Trie dest = NULL;
42   dest = getChild(d->root, word[index]); //first letter
43   int isEndNode = 1;
44   while (word[index] != '\0' && dest != NULL) { //end at the end of the word
```

```
      or when the letter is not found in the trie
    isEndNode = dest->endNode; //update isEndNode
    index++;
    dest = getChild(dest, word[index]); //check the next letter of the word
  }
  // we went to the end of the word or dest == NULL
  if (word[index] == '\0' && isEndNode == 1) {
    return 1;
  }
  return 0;
}

void freeDict(dict *d) {
  freeTrie(d->root);
  free(d);
}
```

Listing 3: dictionary.h

```
/* name: dictionary.h
 *
 * author: Michal Tesnar (m.tesnar@student.rug.nl, s4740556)
 * author: Wojciech Trejter (w.trejter@student.rug.nl, s4296850)
 *
 * date: 06.03.2022
 *
 * description: Declaration of dictonary; header.
 */

#ifndef DICTIONARY_H
#define DICTIONARY_H

#include <stdbool.h>
#include "trie.h"

// maximum length for a word
#define LENGTH 45

// a dictionary is an array
typedef struct dict {
  Trie root;
} dict;

dict *newEmptyDict();

void addWord(char word[LENGTH + 1], dict *d);

bool check(const char *word, dict *d);

void freeDict(dict *d);

#endif // DICTIONARY_H
```

Listing 4: trie.c

```
/* name: trie.c
 *
 * author: Michal Tesnar (m.tesnar@student.rug.nl, s4740556)
 * author: Wojciech Trejter (w.trejter@student.rug.nl, s4296850)
 *
 * date: 06.03.2022
```

```c
 *
 * description: Implementation of trie for a dictionary.
 * It is possible to add a node with a letter and search
 * whether a node with letter exists.
 */

#include "trie.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

Trie newTrie() {
  Trie output = malloc(sizeof(struct TrieNode));
  if (output == NULL) {
    return NULL;
  }
  //initialise the elements of the trie
  output->endNode = 0;
  output->children = malloc(sizeof(Trie) * 26);
  output->hasChild = malloc(sizeof(int) * 26);
  for (int i = 0; i < 26; i++) { //initialising hasChild to 0
    output->hasChild[i] = 0;
  }
  return output;
}

// return the child or NULL if the desired child does not exist
Trie getChild(Trie t, char ch) {
  if (ch == '\0') { //make sure we are looking for a real solution
    return NULL;
  }
  if ((t->hasChild[ch - 'a']) != 0) { //return a child if it exists
    return (t->children[ch - 'a']);
  }
  // if not found return NULL
  return NULL;
}

//this function returns the node of the new child, or one which already
   exists with the char ch
Trie addOrGetChild(Trie t, char ch) {
  if (t->hasChild[ch - 'a'] != 0) { //if the node exists return it
    return (t->children[ch - 'a']);
  }
  t->children[ch - 'a'] = newTrie(); //otherwise create a new child
  t->hasChild[ch - 'a'] = 1; //and indicate it exist
  return t->children[ch - 'a'];
}

//recursively frees every Trie from the very bottom
void freeTrie(Trie t) {
  for (int i = 0; i < 26; i++) {
    if (t->hasChild[i] != 0) {
      freeTrie(t->children[i]);
    }
  }
  free(t->children);
  free(t->hasChild);
  free(t);
}
```

Listing 5: trie.c

```c
/* name: trie.h
 *
 * author: Michal Tesnar (m.tesnar@student.rug.nl, s4740556)
 * author: Wojciech Trejter (w.trejter@student.rug.nl, s4296850)
 *
 * date: 06.03.2022
 *
 * description: Declaration of trie; header.
 */

#include <stdbool.h>

typedef struct TrieNode *Trie;

struct TrieNode {
  bool endNode;
  int *hasChild;
  Trie *children;
};

Trie newTrie();

Trie getChild(Trie t, char ch);

Trie addOrGetChild(Trie t, char ch);

void freeTrie(Trie t);
```