Assignment 2

z5019338
z5131048

March 3, 2019

1 Introduction

2 A Data Type

Consider a syntactic dictionary data type *Dict*

$$
\begin{aligned}
&\textbf{proc } \mathit{addWord}^{D}(\textbf{value } w)\cdot\\
&\quad D : \left[\ \mathit{True}, D := D_0 \cup \{w\}\ \right]\\
&\textbf{proc } \mathit{checkWord}^{D}(\textbf{value } w)\cdot\\
&\quad \textbf{var } b \cdot b : \left[\ \mathit{True}, b := (w \in D)\ \right]\\
&\textbf{proc } \mathit{delWord}^{D}(\textbf{value } w)\cdot\\
&\quad D : \left[\ D \neq \langle\rangle \wedge w \in D, D := D_0 \backslash \{w\}\ \right]
\end{aligned}
$$

3 A Refinement

4 A More Realistic Data Refinement

5 The C Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include ”dict.h”

void newdict(Dict *dp) {
    //Memory allocation
    Dict newTrie = (struct __tnode__ *) malloc(sizeof(struct __tnode__));

    //Initialises end of word to false.
    newTrie->eow = FALSE;

    //Sets all children to NULL.
    for (int i = 0; i < VECSIZE; i++) {
        newTrie->cvec[i] = NULL;
    }

    *dp = newTrie;
}

//works iterative.
//void addword (const Dict r, const word w) {
// TNode *curr = r;
// int level;
// int length = strlen(w);
// int i;
//
// for (level = 0; level < length; level++) {
// //this locates the letter index based off the ASCII Table
// i = w[level] - 97;
//// printf("Character: %c, %d\n", w[level], w[level]);
//
// //if there is no node for that letter, make a new one
// if (curr->cvec[i] == NULL) {
// //make a new node and add.
// Dict newNode;
// newdict(&newNode);
// curr->cvec[i] = newNode;
// }
// // and then/or else, just traverse to it.
// printf("current letter added: %d\n", i);
// curr = curr->cvec[i];
// }
// //once loop is finished, this current node should be an end of word.
// curr->eow = TRUE;
//}

//recursive
void addword (const Dict r, const word w) {
    //w points at first letter of word (and then current letter as we recursively call).
    //if the letter its pointing at is null, then we have reached the end of the word.

    if (*w == '\0') {
            r->eow = TRUE;
            return;
        } else {
            //This selects which index it is between 0 and 26.
            int i = *w - 97;
            //if the next index does not exist, need to create one!
            if (r->cvec[i] == NULL) {
                Dict newNode;
                newdict(&newNode);
                r->cvec[i] = newNode;
            }
            //try again for next index in word.
            addword(r->cvec[i], w+1);
        }
}

bool checkword (const Dict r, const word w) {
        if (*w == '\0' && r->eow == TRUE) {
            return TRUE;
        } else {
            int i = *w - 97;
            if (r->cvec[i] == NULL) {

                return FALSE;
            } else {
                return checkword(r->cvec[i], w+1);
            }
        }
}

//bool checkword (const Dict r, const word w) {
// TNode *curr = r;
// int level;
// int length = strlen(w);
// int i;
//
// for (level = 0; level < length; level++) {
// i = w[level] - 97;
// //case if nextnode is null for that letter, then word not in dictionary
// printf("level: %d\n", level);
// if (curr->cvec[i] == NULL) {
// return FALSE;
// }
// curr = curr->cvec[i];
// }
// //case if word length is complete AND the current node is not a eow then TRUE.
// if (level == length && curr->eow == FALSE) {
// return FALSE;
// } else {
// return TRUE;
// }
//}

//eow = false;
void delword (const Dict r, const word w) {
    if (*w == '\0') {
        r->eow = FALSE;
    } else {
        int i = *w - 97;
        delword(r->cvec[i], w+1);
    }
}

void barf(char *s) {
    printf("%s\n", s);
}
```