Hacettepe University Computer Science Department

BBM 203 Programming Assignment 4 – REPORT

Name Surname: Mert Çökelek

ID: 21727082

Email: b21727082@cs.hacettepe.edu.tr - 100mert101@gmail.com

Subject: Implementing a login system with the Abstract Data Type "Trie" in C Programming Language

Data Due: 06.01.2019

Advisor: Pelin Canbay

1. **What is the main problem and what needs to be understood?**
    a. The main goal in this assignment is to gain experience about Trie Data Structure.
    b. Using recursion and dynamic memory, we are expected to write a C program that allows the user to add new names to the database with their passwords, search for given string in the database, login to system with the passwords, delete the given name and lastly list all the names in the database. This database is constructed in Trie Data Type, because in this way, we can avoid wasting memory for the unneccessary strings, if there are common parts of different names, we can hold their common parts with 1 allocated memory, and also traverse on this tree is more efficient, every time looking for a new character, the alternatives to look for are decreasing.

2. **My approach to the problem is:**
    a. *Determine how to use input.*
        i. There is 1 input file named input.txt. It contains the commands such as add, search, login, delete and list.
        ii. Take input file from as command line argument.
        iii. Read it line by line by "fgets" command and split every line into parts seperated by commas.
        iv. If the first element of the splitted string is 'a', we need to add the given name to the tree.
        v. If it is 's', we need to search for the given string,if it exists output it and its password to the file. Else, print an error message to the output file.

vi. If it is 'q', search for the given name and if it is found, check the given password with its password. If the passwords match, print 'login successful', else, print an error message to the output file.

vii. If it is 'd', search for the given string in the tree. If it is found, delete the name from the trie and 'print deletion is successful'. Else, print an error message to the file.

viii. Finally, if it is 'l', list all the elements in the tree. If they have common parts, Use indentation to make the readibility better.

b. *Determine the data structures to be used.*

3. **The Data Structures I used:**

We are supposed to use Trie data structure. My Trie consists of nodes.

Here is my Node structure.

```
struct n{
    char letter, *password;
    struct n* children, *next;
};
typedef struct n node;
```

Letter corresponds to the node's main info. Letters(nodes) come together and create names.

The node which holds the last letter of the name also has a *password* which is a char pointer.

**children* is a list pointer. It can be expanded and shrinked dynamically, in cases addition and deletion. **next* holds the next element of the node.

To create a new node, I used init() function.

```
node* init(char letter, char* password){                /* for creating a new node. like a constructor. */
    node* newNode = (node*)malloc(sizeof(node));
    newNode->next = NULL;
    newNode->letter = letter;
    newNode->children = NULL;

    newNode->password = (char*)malloc(sizeof(char) * strlen(password));

    if(password[strlen(password)-1] == '\n')        /* if there is a \n at the end of the string, delete it. */
        password[strlen(password)-1] = '\0';
    strcpy(newNode->password, password);

    return newNode;
}
```

This function takes parameters for the new node's letter, and if there is, its password and returns that new node.

My detailed Algorithm is:

1. While reading input, if command is add:
    1.1. Hold the name.

1.2. Add that name to the tree.

1.3. To do this, I defined a function:

```
node* add_to_tree(node* root, node* parent, char* name, char* password, int i, int* success){

    if(password[strlen(password)-1] == '\n')    /* if password contaions newline char, remove it. */
        password[strlen(password)-1] = '\0';    /* because it causes problem in login function, passwords don't match.*/

    /* Checking if the parent has a child with the same letter. */
    if(childExists(parent, name[i]) == NULL){               /* If not, add a new child to the parent. */
        if(i < strlen(name)-1){                             /* If not end of the name, keep going deeper. */
            node* newChild = init(name[i], "");             /* The new node to be added */
            parent->children = append(parent->children, newChild);      /* Realloc the children array of the parent, and
            node* tempParent = childExists(parent, name[i]);        /* This holds the parent of the new added node. */
            return add_to_tree(root, tempParent, name, password, i+1, success); /* Since not end of name, recursion.  */
        }

            /* Last letter and password is adding, Base Case. */
        else {
            node* newChild = init(name[i], password);
            parent->children = append(parent->children, newChild); /* Add the new node to the parent's children array. *
            *success = 1; /* If success = 1, printing name added successfully. */
            return root;
        }
    }
    else if(i == strlen(name)-1){                           /* If the last letter is to be added */
        if(childExists(parent, name[i])->password[0] != ""[0]){     /* If the name is completely registered before*/
            *success = 0;       /* Since trying to add the same name twice, succession is failed. */
        } else {
            /* If the new name is a sub string of the registered names. e.g Cemile exists, new element is Cemil.*/
            strcpy(childExists(parent, name[i])->password, password);
            *success = 1;
        }
        return root;
    }
    else {        /* If the parent has a child with the given letter, don't add it, go next letter recursively. */
        node* tempParent = childExists(parent, name[i]);
        return add_to_tree(root, tempParent, name, password, i+1, success);
    }
}
```

1.4. This function takes 5 parameters: For the root, current parent, name and password to be added, i as a counter which holds the current index of the name, and success variable.

The comments are explaining my algorithm for this function.

1.5. The functions used in this add_to_tree function are:

1.5.1. node* childExists(node* parent, char letter): The parameters are: Parent node and a letter.

1.5.2. Checks if the parent has a child which holds the given letter. If yes, returns that child node. Else, returns NULL.

1.5.3. node* init(char letter, char* pass): Creates a new node.

1.5.4. node* append(node* Node, node* newNode): Adds the new node to the previous nodes' list.

2. If command is search:

2.1. I defined a function for this purpose.

```
int search(node* head, char* name, int i, char** pass) {
    /* The node does not exist at root's children. */
    if (childExists(head, name[i]) == NULL) {
        if (i == 0)
            return 0;
        else
            return 1;
    }
    node* iter = childExists(head, name[i]);    /* iter = iterator node */

    /* End of searching name, there are alternatives */
    if(i == strlen(name)-1){
        if(strcmp(iter->password, "") != 0){
            *pass = iter->password;
            return 3;
        }else
            return 2;
    }
    else
        return search(iter, name, i+1, pass);
}
```

2.2.

2.3. The parameters are:

2.3.1. *Head:* the root node, *Name:* searching name, *pass:* password pointer,

2.4. The return values are for: *No record, incorrect username, not enough username, and successful outputs, respectively.*

2.5. If the root does not contain the first letter of the name, return 0.

2.6. If the n'th letter exists in tree, but the rest not, return 1.

2.7. If the n'th letter does not exists, return 2;

2.8. If name fully exists in tree, print successful and the password(return 3)

```
/* splitted[1] = searching name */
switch(search(trie, searchingName, 0, &pass)){
    case 0: printf("\"%s\" no record\n", splitted[1]); break;
    case 1: printf("\"%s\" incorrect username\n", splitted[1]); break;
    case 2: printf("\"%s\" not enough username\n", splitted[1]); break;
    case 3: printf("\"%s\" password: \"%s\"\n", splitted[1], pass); break;
    default: break;
}
```

3. If command is login:

3.1. The operations are very similar to search command. The only difference is if the name is found, we need to check if the passwords are matching.

4. If command is delete, I defined the function Remove():

```
void Remove(node** root_ref, char name[], char fullName[], int* success){
    node* iter = *root_ref;
    node* parent = *root_ref;
    int i = 0; // index counter for the name
    while(childExists(iter, name[i])){
        parent = iter;
        iter = childExists(iter, name[i]);
        i++;
    }
    if(strlen(name) == 1 && numberOfChildrenOfNode(iter) > 0){
        if(parent == iter){
            *success = 2;           return;
        }
        if(strcmp(name, fullName) == 0) {
            *success = 4;           return;
        }
        *success = 1;       return;
    }

    /************************  BASE CASES  *********************************/
    if(parent == iter){                         /* None of them have changed, no record */
        *success = 2; return;
    }
    if(parent->letter == (*root_ref)->letter){      /* incorrect username */
        *success = 3; return;
    }
    if(i == strlen(name) && numberOfChildrenOfNode(iter) > 1 && iter->password[0] == '\0'){      /* There are alternatives,
        *success = 4; return;
    }
```

```
    /*************************    RECURSIVE CASES ******************************/

    if(i == strlen(name)){
        if(strcmp(iter->password, "") != 0) {        // iter has a password, last letter
            if(strcmp(name, fullName) == 0){        // Cemile, Cemil -> Cemile is deleted.
                iter->password = "";
                if (numberOfChildrenOfNode(iter) == 0) {                // iter is  a leaf, we can delete iter from tre
                    deleteNode(&(parent->children), iter->letter);
                }

                name[i-1] = '\0';
                return Remove(root_ref, name, fullName, success);

            }else{                                  // Cemile, Cemil, e is deleted, recursion finished.
                *success = 1; return;
            }


        }else{                                      // Correct letter, Middle element of the name.
            if(numberOfChildrenOfNode(iter) == 0){      // iter is a leaf, remove it.
                deleteNode(&(parent->children), name[i-1]);
                name[i-1] = '\0';
                return Remove(root_ref, name, fullName, success);
            }else if(strcmp(name, fullName) == 0) {
                *success = 4;
                return;
            }else{                                      // Selim, Selin, m deleted, Selin should stay
                    name[i-1] = '\0';
                    return Remove(root_ref, name, fullName, success);
            }
        }
    }else{
        *success = 3; return;
    }
```

The Parameters:

Root_ref: Since we need to make changes on root, I passed its reference.

Name[]: The name to be deleted from the tree. At end of every function calls, name's last letter is being removed.

FullName[]: It's used for checking if name is deleted. Comparison.

Success:* In the base cases, there are different combinations for this variable.

In Remove() function:

In base cases, There are 7 important combinations.

1. If name length is 1, this means input is 1 letter or we successfully deleted all the letters except the first one of the letter. And if the node which holds the first letter of the name has children, there are 3 possibilities:
    a. Parent = iter: This means in tree, no iterations are made. So, decided that the root does not have a child like this letter, “No Record”, success =2;
    b. If name = fullName: “name”: the string to be operated, every deletion, one letter is deleted from “name”, but “fullname” remains the same. This comparison helps us know whether the name is ever deleted or not. If so, success = 4, “not enough username”.
    c. Else, means we can successfully remove the first letter(last node to be checked) from the tree, success = 1, “deletion is successful”.
2. If parent = root and the previous if statements are passed, this means there is no such name on this tree.
3. If i = len(name) and iter has children more than 1 and iter does not have a password, this means there are alternative nodes which can be deleted, so success: 4, not enough username.

In recursive Cases:

We need to iterate on the tree to find which elements need to be deleted.

- We are starting deleting name letters from last index to first. E.g if name is “Cemil”, deletion order should be ‘l, i, m, e, c’.
- If the node which contains the letter is found on the tree, we need to check if it is the correct node. For this purpose, I declared a counter and it increments every time the node is iterating. If it is equal to the length of the name, than we are on the correct place.
- But this does not mean we can delete that element, we need to check if it has a password, children, sibling. If it has children, we can’t remove that node from tree, we just reset its password, and return.
- If it has siblings, than the parent node’s children pointer should point the next element of this node.

```
Remove(&trie, name, fullname, &success);

switch (success){
    case 1: fprintf(output,"\"%s\" deletion is successful\n", fullname); break;
    case 2: fprintf(output,"\"%s\" no record\n", fullname); break;
    case 3: fprintf(output,"\"%s\" incorrect username\n", fullname); break;
    case 4: fprintf(output,"\"%s\" not enough username\n", fullname); break;
    default: break;
}
```

5. If command is list, We need to print all the names in the tree.

```c
void preOrderTraversal(node* tree, char* str, FILE* output){
    int i;
    if(tree != NULL){
        if(strcmp(tree->password, "") != 0){    /*  Node has a password*/
            str = concat(str, tree->letter);    /* str is the output, add the node's letter to str. */
            fprintf(output,"-%s\n", str);
            node* iter = tree->children;
            /* For all the children of the parent node, call the function until reaching last letter */
            for(i =0; i < numberOfChildrenOfNode(tree); i++){
                preOrderTraversal(iter, str, output);
                iter = iter->next;

            }
            /* Node has a password and also children, Like Cemil, Cemile */

        } else{    /* Node does not have a password, so keep going until finding a complete name. */
            str = concat(str, tree->letter);
            node* iter = tree->children;

            /* If a node has more than 2 children, print all of them. */
            for(i = 0; i < numberOfChildrenOfNode(tree); i++){

                preOrderTraversal(iter, str, output);
                iter = iter->next;
            }
        }
        strcpy(str, "");       /* Reset the output string to empty string for the next subtree. */
    }
}
```

Str keeps the iterated nodes' letters as a complete string.

The other functions in my code and details about them:

Void deleteNode(node parent, char* letter)**:

This function searches for the given letter in the given parent's children list.

If there is a match, removes that node from the parent's children.

Used in "Remove" function.

Int numberIfChildrenOfNode(node* n):

Returns the number of children of the given node

Used in add_to_tree, preOrderPrint, Remove functions.

Char split(char* line, char* token, int amount):**

Splits a given string into parts by the given token. Amount is number of splitted elements.

Used in reading command file.

Char* concat(char* str, char c):

Appends the given char to the given string and returns that string.

Used in preOrderTraversal function to create the output name.

Node* readInput(char* input, node* trie):

This is the function where every operation on the tree is being done.

Reading the input file line by line, splitting it into pieces,

Add, search, login, delete, list operations.

This function calls the other functions(add_to_tree, search, login, delete, preOrderTraversal, split.).

Called in main function and process is finished.