ASSIGNMENT 1 – HASH TABLES

Name: Vanaj Kamboj
Roll No: 21355100

Task 1
My Code:

```c
//  Created by Vanaj Kamboj on 8/10/2021.
//  Copyright © 2021 Vanaj Kamboj. All rights reserved.

#include <stdio.h>
#include <stdlib.h>
#include <curses.h>
#include <string.h>
#include <ctype.h>  //for isalnum

/*
By "size" of the hash table we mean how many slots or buckets it has
Choice of hash table size depends in part on choice of hash function, and collision resolution strategy
But a good general “rule of thumb” is:
The hash table should be an array with length about 1.3 times the maximum number of keys that will actually be in the table,
and Size of hash table array should be a prime number

So, let M = the next prime larger than 1.3 times the number of keys you will want to store in the table,
and create the table as an array of length M

(If you underestimate the number of keys, you may have to create a larger table and rehash the entries when it gets too full;
if you overestimate the number of keys, you will be wasting some space)
*/
#define ARRAY_SIZE 59
#define MAX_STRING_SIZE 20

int numElements = 0;
int collisions = 0;

typedef struct Element Element;

struct Element {
    unsigned int key;
    char* name;
    unsigned short int frequency;
};

// Creating a global hash table for ease of use
```

```c
Element* hashTable[ARRAY_SIZE];

int hash(char *s)
{
    int hash = 0;
    while (*s)
    {
        hash = (hash + *s) % ARRAY_SIZE;
        s++;
    }
    return hash;
}

int hash2(char *s) {

}

/*-----------------------------------------------------------------
https://www.geeksforgeeks.org/string-hashing-using-polynomial-rolling-hash-function/

Where P and M are some positive numbers.  And s[0], s[1], s[2] … s[n-1] are the values assigned to each character in English alphabet (a->1, b->2, … z->26).

Appropriate values for P and M

P: The value of P can be any prime number roughly equal to the number of different characters used.
For example: if the input string contains only lowercase letters of the English alphabet, then P = 31 is the appropriate value of P.
If the input string contains both uppercase and lowercase letters, then P = 53 is an appropriate option.
M: the probability of two random strings colliding is inversely proportional to m, Hence m should be a large prime number.
M = 10 ^9 + 9 is a good choice.

*/


Element* createNewElement(char* name){
    // TODO
    // you might want to use the function strcpy from the string package here!
    Element *newElement = (Element*) malloc(sizeof(Element));
    newElement->name = (char*) malloc(strlen(name) + 1);

    // Memory allocation completed, now add stuff to element
    strcpy(newElement->name, name);
    newElement->frequency = 1;
    newElement->key = hash(name);

    // printf("Key for %s is %d\n", name, newElement->key);
```

```
    return newElement;
}

// returns the index of element with name name or empty place where element can be
stored, or -1 if hash table is full!
int search(char* name, bool addCollisions) {
    //TODO
    int pos = hash(name);
    int startPos = pos;

    while (hashTable[pos] != NULL) {
        if (strcmp(hashTable[pos]->name, name) == 0)
            return pos;
        pos++;
        // Increment counter for collision metrics
        if (addCollisions)
            collisions++;
        pos = pos % ARRAY_SIZE;
        if (pos == startPos) {
            return -1;
        }
    }

    return pos;
}

// assuming that no element of key name is in the list (use search first!), add
element at the correct place in the list
// NB: it would be more efficient for search to return the index where it should be
stored directly, but aiming for simplicity here!
void insert(char* name) {
   // Check if element exists in table already
   int idx = search(name, true);

   if (idx == -1)
       printf("Hash Table full. Cannot Insert.\n");
   else if (hashTable[idx] == NULL) {
       // Search did not find name and returned empty bucket position
       hashTable[idx] = createNewElement(name);
    //    printf("Inserted %s\n", name);

       // Increment counter for metrics
       numElements++;
   } else {
       // Search has found element with name, so we increase frequency
       hashTable[idx]->frequency++;
    //    printf("Name %s already exists, increased frequency to %d\n", name,
hashTable[idx]->frequency);
   }
}
```

```
// void printElementDetails(Element* element){
//     printf("Name: %s, Frequency: %d \n", element->name, element->frequency);
// }

//  Reads the contents of a file and adds them to the hash table - returns 1 if file was successfully read and 0 if not.
int load_file ( char *fname ) {
    FILE* file = fopen(fname,"r");
    if(!file) {
        printf("Can not open the File");
        return 0;
    }
    else {
        printf("%s loaded!\n", fname);
        char buffer[1024];
        bool isFirst = true;
        bool foundQuote = false;
        char element[MAX_STRING_SIZE];

        while(fgets(buffer, 1024, file)) {
            memset(element, 0, MAX_STRING_SIZE);

            // Iterate over the buffer and print column elements
            buffer[strcspn(buffer, "\n")] = '\0';
            insert(buffer);

            // Print last element remaining inside the variable
            printf("%s", element);
        }
    }

    fclose(file);
    return 1;
}

void printMetrics() {
    printf("\tCapacity: %d\n", ARRAY_SIZE);
    printf("\tNum Terms: %d\n", numElements);
    printf("\tCollisions: %d\n", collisions);
    printf("\tLoad: %f\n", (double)numElements/ARRAY_SIZE);
}


int main() {

    load_file("names.csv");
    printMetrics();

    char input[MAX_STRING_SIZE];
```

```
    while (true)
    {
        printf("Enter term to get frequency or type \"quit\" to escape\n");
        fgets(input, MAX_STRING_SIZE, stdin);
        input[strcspn(input, "\n")] = '\0';

        if (strcmp(input, "quit") == 0)
            break;

        int index = search(input, false);
        if (index == -1) {
            printf("Name not found and hash table is full.\n");
        } else if (hashTable[index] == NULL) {
            printf("%s not in table.\n", input);
        } else {
            printf("%s %d\n", input, hashTable[index]->frequency);
        }
    }

    return 0;
}
```

Output:

```
vanajkamboj@Vanajs-MacBook-Pro A2 % cd "/Users/vanajkamboj/My_files/Study/DSA/A2/" && gcc T1.c -o T1 && "/Users/vanajkamboj/My_files/Study/DSA
/A2/"T1
T1.c:53:1: warning: non-void function does not return a value [-Wreturn-type]
}
^
1 warning generated.
names.csv loaded!
        Capacity: 59
        Num Terms: 39
        Collisions: 23
        Load: 0.661017
Enter term to get frequency or type "quit" to escape
Synnot
Synnot 2
Enter term to get frequency or type "quit" to escape
bleb
bleb not in table.
Enter term to get frequency or type "quit" to escape
quit
vanajkamboj@Vanajs-MacBook-Pro A2 %
```

Task 2

Change in hash function:

```
int hash(char *s)
{
    int hash = 0;
    while (*s)
    {
        hash = (31*hash + *s) % ARRAY_SIZE;
        s++;
    }
    return hash;
}
```

Output:

Explanation:
We can see that the number of collisions have reduced from 23 to 14 using this function. I am multiplying my hash with a prime number because it gives the best chance of obtaining a unique key.

Task 3

My Code:

```
//  Created by Vanaj Kamboj on 10/10/2021.
//  Copyright © 2021 Vanaj Kamboj. All rights reserved.

#include <stdio.h>
#include <stdlib.h>
#include <curses.h>
#include <string.h>
#include <ctype.h>  //for isalnum

#define ARRAY_SIZE 59
#define MAX_STRING_SIZE 20

int numElements = 0;
int collisions = 0;

typedef struct Element Element;

struct Element {
    unsigned int key;
    char* name;
    unsigned short int frequency;
};

// Creating a global hash table for ease of use
Element* hashTable[ARRAY_SIZE];

int hash1(char *s)
{
    int hash = 0;
```

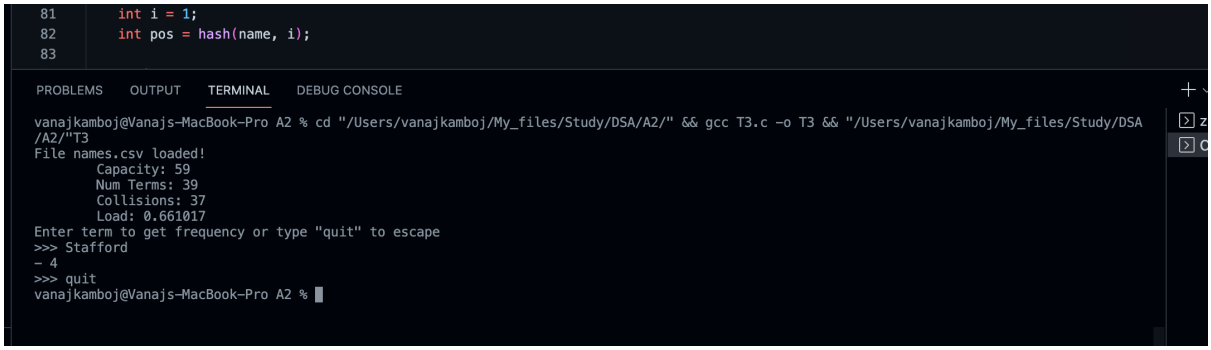
```
    while (*s)
    {
        hash = (hash + *s) % ARRAY_SIZE;
        s++;
    }
    return hash;
}
int hash2(char *s)
{
    int hash = 0;
    while (*s)
    {
        hash = (31*hash + *s) % ARRAY_SIZE;
        s++;
    }
    return hash;
}
int hash(char *s, int i){
    int hash = 0;
    int f, g, k;
    while(*s){
        k = hash + *s;
        f = k % ARRAY_SIZE;
        g = 1 + (k % (ARRAY_SIZE-1));
        hash = (f + i * g) % ARRAY_SIZE;
        s++;
    }
    return hash;
    printf("\n%d", hash);
}


Element* createNewElement(char* name, int i){
    // TODO
    // you might want to use the function strcpy from the string package here!
    Element *newElement = (Element*) malloc(sizeof(Element));
    newElement->name = (char*) malloc(strlen(name) + 1);

    // Memory allocation completed, now add stuff to element
    strcpy(newElement->name, name);
    newElement->frequency = 1;
    newElement->key = hash(name, i);

    // printf("Key for %s is %d\n", name, newElement->key);

    return newElement;
}

// returns the index of element with name name or empty place where element can be
stored, or -1 if hash table is full!
int search(char* name, bool addCollisions) {
```

```
    //TODO
    int i = 1;
    int pos = hash(name, i);

    // int startPos = pos;

    while (hashTable[pos] != NULL) {
        i++;
        if (strcmp(hashTable[pos]->name, name) == 0)
            return pos;

        pos = hash(name, i);
        // Increment counter for collision metrics
        if (addCollisions)
            collisions++;
        // pos = pos % ARRAY_SIZE;
    //     if (pos == startPos) {
    //         return -1;
    //     }
    }

    return pos;
}

// assuming that no element of key name is in the list (use search first!), add
element at the correct place in the list
// NB: it would be more efficient for search to return the index where it should be
stored directly, but aiming for simplicity here!
void insert(char* name) {
   // Check if element exists in table already
   int idx = search(name, true);

   if (idx == -1)
       printf("Hash Table full. Cannot Insert.\n");
   else if (hashTable[idx] == NULL ) {
       // Search did not find name and returned empty bucket position
       hashTable[idx] = createNewElement(name, idx);
    //    printf("Inserted %s\n", name);

       // Increment counter for metrics
       numElements++;
   } else {
       // Search has found element with name, so we increase frequency
       hashTable[idx]->frequency++;
    //    printf("Name %s already exists, increased frequency to %d\n", name,
hashTable[idx]->frequency);
   }
}


void printElementDetails(Element* element){
```

```
    printf("Name: %s, Frequency: %d \n", element->name, element->frequency);
}

//  Reads the contents of a file and adds them to the hash table - returns 1 if
file was successfully read and 0 if not.
int load_file ( char *fname ) {
    FILE* file = fopen(fname,"r");
    if(!file) {
        printf("Can not open the File");
        return 0;
    }
    else {
        printf("File %s loaded!\n", fname);
        char buffer[1024];
        // bool isFirst = true;
        // bool foundQuote = false;
        char element[MAX_STRING_SIZE];

        while(fgets(buffer, 1024, file)) {
            memset(element, 0, MAX_STRING_SIZE);

            // Iterate over the buffer and print column elements
            buffer[strcspn(buffer, "\n")] = '\0';
            insert(buffer);

            // Print last element remaining inside the variable
            printf("%s", element);
        }
    }

    fclose(file);
    return 1;
}

void printMetrics() {
    printf("\tCapacity: %d\n", ARRAY_SIZE);
    printf("\tNum Terms: %d\n", numElements);
    printf("\tCollisions: %d\n", collisions);
    printf("\tLoad: %f\n", (double)numElements/ARRAY_SIZE);
}


int main() {

    load_file("names.csv");
    printMetrics();
    printf("Enter term to get frequency or type \"quit\" to escape\n");

    char input[MAX_STRING_SIZE];
    while (true)
    {
```

```
        printf(">>> ");
        fgets(input, MAX_STRING_SIZE, stdin);
        // scanf("%s",&input);
        input[strcspn(input, "\n")] = '\0';

        if (strcmp(input, "quit") == 0)
            break;

        int index = search(input, false);
        if (index == -1) {
            printf("Name not found and hash table is full.\n");
        } else if (hashTable[index] == NULL) {
            printf("%s not in table.\n", input);
        } else {
            printf("- %d\n", hashTable[index]->frequency);
        }
    }

    return 0;
}
```

Output:



I would not suggest using double hashing method for this particular problem as we can see that the number of collisions have significantly increased.

Task 4

My Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <curses.h>
#include <string.h>
#include <ctype.h>  //for isalnum

#define ARRAY_SIZE 99991
#define MAX_STRING_SIZE 100
```

```
int numElements = 0;
int collisions = 0;

typedef struct Person Person;
struct Person{
    int id;
    char* depositionId;
    char* surname;
    char* forename;
    char* personType;
    char* gender;
    char* nationality;
    char* religion;
    char* occupation;
    int age;
    Person* next;
};

typedef struct Element Element;
struct Element{
    int key;
    Person* person;
};

// Creating a global hash table for ease of use
Element* hashTable[ARRAY_SIZE];

int hash(char *s)
{
    int hash = 0;
    while (*s)
    {
        hash = (31*hash + *s) % ARRAY_SIZE;
        s++;
    }
    return hash;
}

Person *createNewPerson(char buffer[])
{
    int column = 0;
    bool foundQuote = false;
    char element[MAX_STRING_SIZE];
    memset(element, 0, MAX_STRING_SIZE);    // Clear variable to build next field

    // Result Person struct
    Person *person = malloc(sizeof(Person));

    // Iterate over the buffer and print column elements
    for (int i = 0; i < strlen(buffer); i++) {
        switch (buffer[i])
```

```
        {
            case ',':
                if (!foundQuote) {
                    switch (column)
                    {
                    case 0:
                        person->id = atoi(element);
                        break;
                    case 1:
                        person->depositionId = strdup(element);
                        break;
                    case 2:
                        person->surname = strdup(element);
                        break;
                    case 3:
                        person->forename = strdup(element);
                        break;
                    case 4:
                        person->age = atoi(element);
                        break;
                    case 5:
                        person->personType = strdup(element);
                        break;
                    case 6:
                        person->gender = strdup(element);
                        break;
                    case 7:
                        person->nationality = strdup(element);
                        break;
                    case 8:
                        person->religion = strdup(element);
                        break;
                    // case 9:
                    //     person->occupation = strdup(element);
                    //     break;
                    default:
                        break;
                    }

                    // Store this in our struct.
                    memset(element, 0, MAX_STRING_SIZE);    // Clear variable to
build next field
                    column++;
                }
                break;
            case '\"':
                foundQuote = !foundQuote;
                continue;
            default:
                strcat(element, (char[2]) { buffer[i], '\0' });
                break;
```

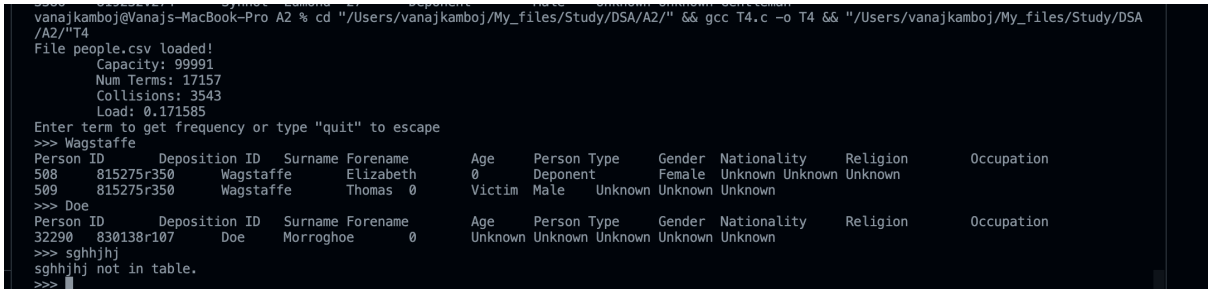
```
        }
    }

    // Add last element remaining inside the variable
    // We know last column is pokedex
    person->occupation = strdup(element);

    return person;
}

Element *createNewElement(Person *person)
{
    Element *newElement = (Element*) malloc(sizeof(Element));

    // Use hash function and get key
    newElement->key = hash(person->surname);
    newElement->person = person;

    // printf("Key for %s is %d\n", name, newElement->key);
    return newElement;
}

// returns the index of element with name name or empty place where element can be
stored, or -1 if hash table is full!
int search(char* surname, bool addCollisions) {
    //TODO
    int pos = hash(surname);
    int startPos = pos;

    while (hashTable[pos] != NULL) {
        if (strcmp(hashTable[pos]->person->surname, surname) == 0)
            return pos;
        pos++;
        // Increment counter for collision metrics
        if (addCollisions)
            collisions++;
        pos = pos % ARRAY_SIZE;
        if (pos == startPos) {
            return -1;
        }
    }

    return pos;
}

// assuming that no element of key name is in the list (use search first!), add
element at the correct place in the list
// NB: it would be more efficient for search to return the index where it should be
stored directly, but aiming for simplicity here!
void insert(Person* person)
{
```

```
    // Check if element exists in table already
    int idx = search(person->surname, true);

    if (idx == -1)
        printf("Hash Table full. Cannot Insert.\n");
    else if (hashTable[idx] == NULL)
    {
        // Search did not find name and returned empty bucket position
        hashTable[idx] = createNewElement(person);

        // Increment counter for metrics
        numElements++;
    }
    else
    {
        // Search has found element with name, so we increase frequency
        // Traverse our linked list to the last node
        Person *current = hashTable[idx]->person;
        while (current->next != NULL) {
            current = current->next;
        }

        // Insert person to end of LL
        current->next = person;
    }
}

//  Reads the contents of a file and adds them to the hash table - returns 1 if
file was successfully read and 0 if not.
int load_file ( char *fname ) {
    FILE* file = fopen(fname,"r");
    if(!file) {
        printf("Can not open the File");
        return 0;
    }
    else {
        printf("File %s loaded!\n", fname);
        char buffer[1024];
        // bool isFirst = true;
        // bool foundQuote = false;
        char element[MAX_STRING_SIZE];
        while(fgets(buffer, 1024, file)) {
            memset(element, 0, MAX_STRING_SIZE);

            // Iterate over the buffer and print column elements
            buffer[strcspn(buffer, "\n")] = '\0';
            Person* person = createNewPerson(buffer);
            // printf("Person for insert -> %d -> %s %s\n", person->id, person-
>forename, person->surname);
            insert(person);
        }
```

```
    }

    fclose(file);
    return 1;
}

void printMetrics() {
    printf("\tCapacity: %d\n", ARRAY_SIZE);
    printf("\tNum Terms: %d\n", numElements);
    printf("\tCollisions: %d\n", collisions);
    printf("\tLoad: %f\n", (double)numElements/ARRAY_SIZE);
}

void printPersonList(Person *head) {
    printf("Person ID\tDeposition ID\tSurname\tForename\tAge\tPerson
Type\tGender\tNationality\tReligion\tOccupation\n");
    Person *current = head;
    do {
        printf("%d\t%s\t%s\t%s\t%d\t%s\t%s\t%s\t%s\t%s\n", current->id, current-
>depositionId, current->surname, current->forename, current->age, current-
>personType, current->gender, current->nationality, current->religion, current-
>occupation);
        current = current->next;
    } while (current != NULL);
}


int main() {

    load_file("people.csv");
    printMetrics();
    printf("Enter term to get frequency or type \"quit\" to escape\n");

    char input[MAX_STRING_SIZE];
    while (true)
    {
        printf(">>> ");
        fgets(input, MAX_STRING_SIZE, stdin);
        input[strcspn(input, "\n")] = '\0';

        if (strcmp(input, "quit") == 0)
            break;

        int index = search(input, false);
        if (index == -1) {
            printf("Name not found and hash table is full.\n");
        } else if (hashTable[index] == NULL) {
            printf("%s not in table.\n", input);
        } else {
            printPersonList(hashTable[index]->person);
        }
```

```
    }

    return 0;
}
```

Output:



In task four I have created a hash table that stores linked lists of same surnames under one bucket. Each bucket holds the Head of the linked list and points to the next person with same surname.