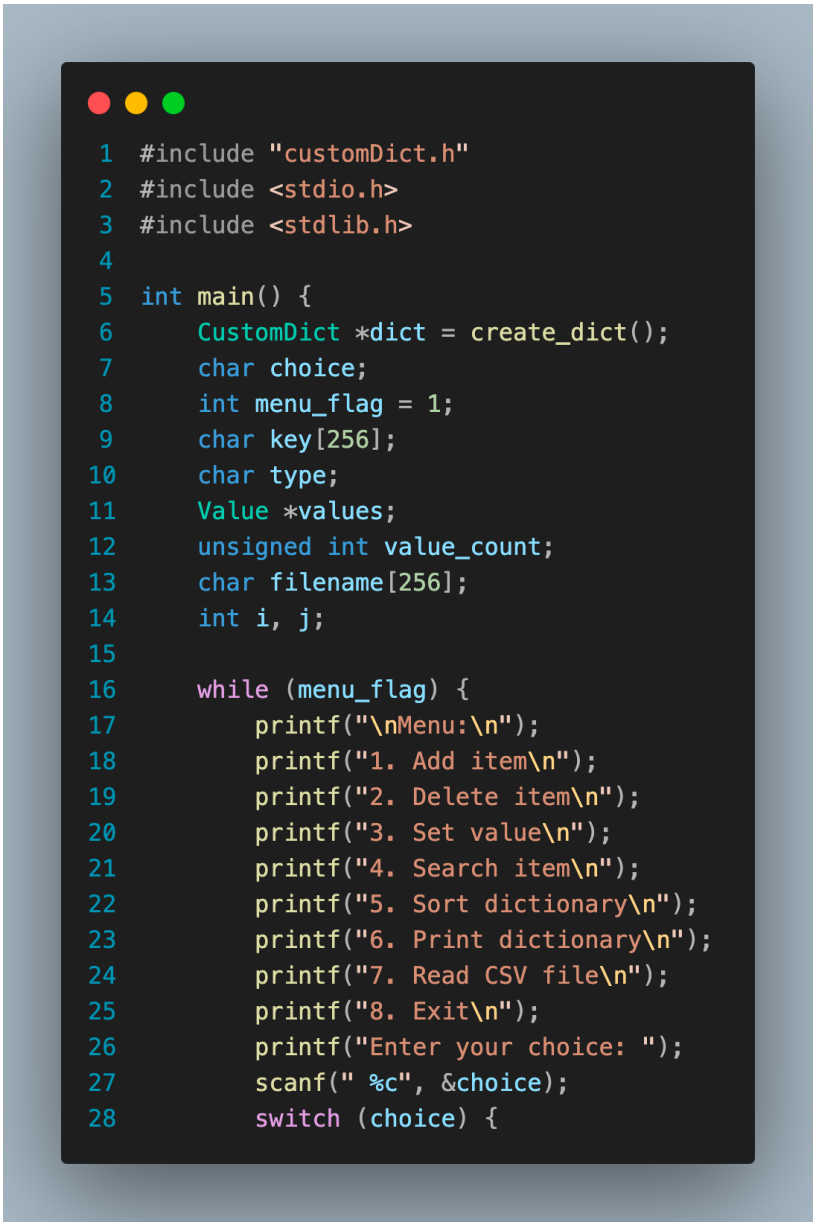
CSE102-Assignment 12

Yusuf Alperen Çelik
(220104004064)

https://www.youtube.com/watch?v=IhGxVAKH9YI

MAIN.C



```c
#include "customDict.h"
#include <stdio.h>
#include <stdlib.h>

int main() {
    CustomDict *dict = create_dict();
    char choice;
    int menu_flag = 1;
    char key[256];
    char type;
    Value *values;
    unsigned int value_count;
    char filename[256];
    int i, j;

    while (menu_flag) {
        printf("\nMenu:\n");
        printf("1. Add item\n");
        printf("2. Delete item\n");
        printf("3. Set value\n");
        printf("4. Search item\n");
        printf("5. Sort dictionary\n");
        printf("6. Print dictionary\n");
        printf("7. Read CSV file\n");
        printf("8. Exit\n");
        printf("Enter your choice: ");
        scanf(" %c", &choice);
        switch (choice) {
```

The provided C program implements a menu-driven interface for managing a custom dictionary. The main function initializes a dictionary and enters a loop to display a menu with options to add, delete, set values, search, sort, print the dictionary, read from a CSV file, or exit. Based on the user's choice, it performs the corresponding action, like adding or deleting

items, setting values, searching for keys, sorting the dictionary, printing its contents, or reading data from a CSV file. The loop continues until the user chooses to exit.

```c
            case '1':
                printf("Enter key: ");
                scanf("%s", key);
                printf("Enter type (i: int, f: float, d: double, c: char): ");
                scanf(" %c", &type);
                value_count = 0;
                values = NULL;
                char value[256];
                printf("Enter values (enter 'e' to stop): ");
                while (1) {
                    scanf("%s", value);
                    if (strcmp(value, "e") == 0) break;
                    values = (Value *)realloc(values, (value_count + 1) * sizeof(Value));
                    switch (type) {
                        case 'i':
                            values[value_count].i = atoi(value);
                            break;
                        case 'f':
                            values[value_count].f = atof(value);
                            break;
                        case 'd':
                            values[value_count].d = atof(value);
                            break;
                        case 'c':
                            values[value_count].c = value[0];
                            break;
                    }
                    value_count++;
                }
                // Add a null terminator for char arrays
                if (type == 'c') {
                    values = (Value *)realloc(values, (value_count + 1) * sizeof(Value));
                    values[value_count].c = '\0';
                }
                add_item(dict, key, values, value_count, type);
                break;

            case '2':
                printf("Enter key to delete: ");
                scanf("%s", key);
                delete_item(dict, key);
                break;

            case '3':
                printf("Enter key: ");
                scanf("%s", key);
                printf("Enter type (i: int, f: float, d: double, c: char): ");
                scanf(" %c", &type);

                value_count = 0;
                values = NULL;

                printf("Enter values (enter 'e' to stop): ");
                while (1) {
                    scanf("%s", value);
                    if (strcmp(value, "e") == 0) break;

                    values = (Value *)realloc(values, (value_count + 1) * sizeof(Value));
                    switch (type) {
                        case 'i':
                            values[value_count].i = atoi(value);
                            break;
                        case 'f':
                            values[value_count].f = atof(value);
                            break;
                        case 'd':
                            values[value_count].d = atof(value);
                            break;
                        case 'c':
                            values[value_count].c = value[0];
                            break;
                    }
                    value_count++;
                }
                // Add a null terminator for char arrays
                if (type == 'c') {
                    values = (Value *)realloc(values, (value_count + 1) * sizeof(Value));
                    values[value_count].c = '\0';
                }
                set_value(dict, key, values, value_count, type);
                break;
            case '4':
                printf("Enter key to search: ");
                scanf("%s", key);
                i = Find_index(dict, key); /*Find the index of the key*/
                values = search_item(dict, key); /*Search for the item*/
                if (values != NULL) {
                    printf("Values: ");
                    for (j = 0; j < dict->items[i].value_count; ++j) { /*Print the values*/
                        switch (dict->items[i].type) {
                            case 'i':
                                printf("%d ", values[j].i);
                                break;
                            case 'f':
                                printf("%.2f ", values[j].f);
                                break;
                            case 'd':
                                printf("%.2lf ", values[j].d);
                                break;
                            case 'c':
                                printf("%c ", values[j].c);
                                break;
                        }
                    }
                    printf("\n");
                } else {
                    printf("Item not found.\n");
                }
                break;
            case '5':
                sort_dict(dict);
                printf("Dictionary sorted.\n");
                break;
            case '6':
                print_dict(dict);
                break;
            case '7':
                printf("Enter CSV filename: ");
                scanf("%s", filename);
                if (read_csv(dict, filename))
                {
                    printf("CSV file read.\n");
                } else {
                    printf("Failed to read file.\n");
                }
                break;
            case '8':
                free_dict(dict);
                printf("Exiting...");
                menu_flag = 0;
                break;
            default:
                printf("Invalid choice. Please try again.\n");
                break;
        }
    }
    return 0;
}
```

Case ‘1’ (Add item):

- Prompts the user to enter a key and the type of values (int, float, double, char).
- Continuously reads values from the user until ‘e’ is entered.
- Converts the input values to the specified type and stores them in a dynamically allocated array.
- Adds the item to the dictionary using the add_item function.

Case ‘2’ (Delete item):

- Prompts the user to enter the key of the item to be deleted.
- Deletes the item from the dictionary using the delete_item function.

Case ‘3’ (Set value):

- Similar to Case ‘1’, but updates the values of an existing item instead of adding a new one.
- Prompts the user for key, value type, and values, then sets the new values using the set_value function.

Case ‘4’ (Search item):

- Prompts the user to enter a key to search for.
- Searches for the item in the dictionary using the search_item function.
- If found, prints the values of the item; otherwise, indicates that the item was not found.

Case ‘5’ (Sort dictionary):

- Sorts the dictionary by keys using the sort_dict function.
- Informs the user that the dictionary has been sorted.

Case ‘6’ (Print dictionary):

- Prints the entire dictionary using the print_dict function.

Case ‘7’ (Read CSV file):

- Prompts the user to enter the filename of a CSV file.
- Reads the CSV file and populates the dictionary using the read_csv function.
- Informs the user whether the file was successfully read.

Case ‘8’ (Exit):

- Frees the memory allocated for the dictionary using the free_dict function.
- Sets the flag to exit the menu loop and informs the user that the program is exiting.

customDic.h

```h
#ifndef CUSTOMDICT_H
#define CUSTOMDICT_H

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

// Value union to store different data types
typedef union {
    int i;
    float f;
    double d;
    char c;
} Value;

// Item struct to represent a dictionary item
typedef struct {
    char *key;
    Value *values;
    unsigned int value_count;
    char type; // 'i' for int, 'f' for float, 'd' for double, 'c' for char
} Item;

// CustomDict struct to represent the dictionary
typedef struct {
    Item *items;
    unsigned int size;
    unsigned int capacity;
} CustomDict;

// Function prototypes
CustomDict* create_dict();
void add_item(CustomDict *dict, const char *key, Value *values, int value_count, char type);
void delete_item(CustomDict *dict, const char *key);
void set_value(CustomDict *dict, const char *key, Value *values, int value_count, char type);
Value* search_item(CustomDict *dict, const char *key);
void sort_dict(CustomDict *dict);
void print_dict(const CustomDict *dict);
void free_dict(CustomDict *dict);
int read_csv(CustomDict *dict, const char *filename);
char* copy_string(const char *source);
int Find_index(CustomDict *dict, const char *key);

#endif // CUSTOMDICT_H
```

This header file, `customDict.h`, defines the structure and function prototypes for a custom dictionary implementation in C. It includes necessary standard libraries and defines a `Value` union to store different data types (int, float, double, char). The `Item` struct represents a dictionary entry with a key, an array of values, the count of values, and the type of values. The `CustomDict` struct represents the dictionary itself, containing an array of `Item` structs, its current size, and its capacity. The

header also declares function prototypes for creating a dictionary, adding, deleting, setting, and searching for items, sorting and printing the dictionary, freeing allocated memory, reading dictionary data from a CSV file, copying strings, and finding the index of a key within the dictionary. These functions support the management and manipulation of the dictionary data structure.

customDict.c

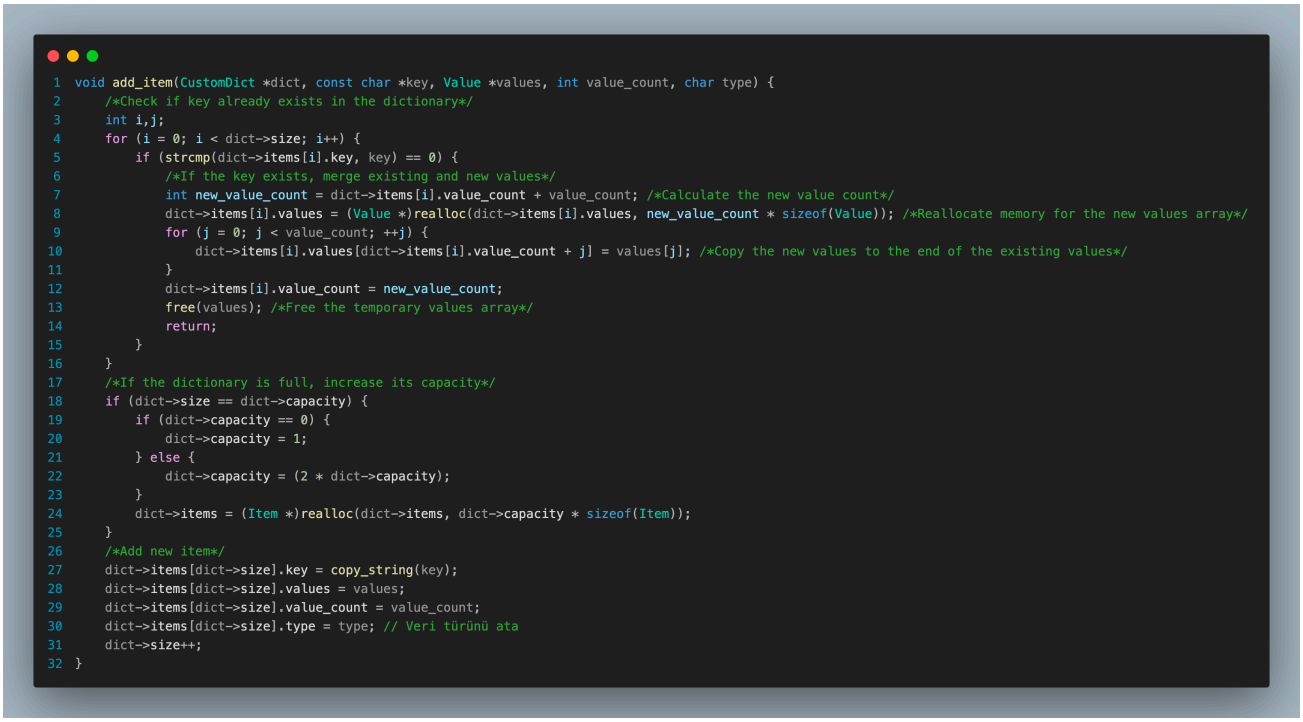
```c
#include "customDict.h"
/*Create a new custom dictionary*/
CustomDict* create_dict() {
    CustomDict *dict = (CustomDict *)malloc(sizeof(CustomDict));
    dict->items = NULL;
    dict->size = 0;
    dict->capacity = 0;
    return dict;
}
```

This function, create_dict, allocates memory for a new CustomDict structure, initializes its items pointer to NULL, and sets its size and capacity to 0. It then returns a pointer to the newly created dictionary.

```c
char* copy_string(const char *source) {
    /*Allocate memory for the new string including the null terminator*/
    char *dest = (char *)malloc(strlen(source) + 1); /*for \0 operator*/
    if (dest) {
        strcpy(dest, source);
    }
    return dest;
}
```

This function, copy_string, allocates memory for a new string that includes space for the null terminator. It then copies the content of the

source string to the newly allocated memory using strcpy. The function returns a pointer to the newly created copy of the string.



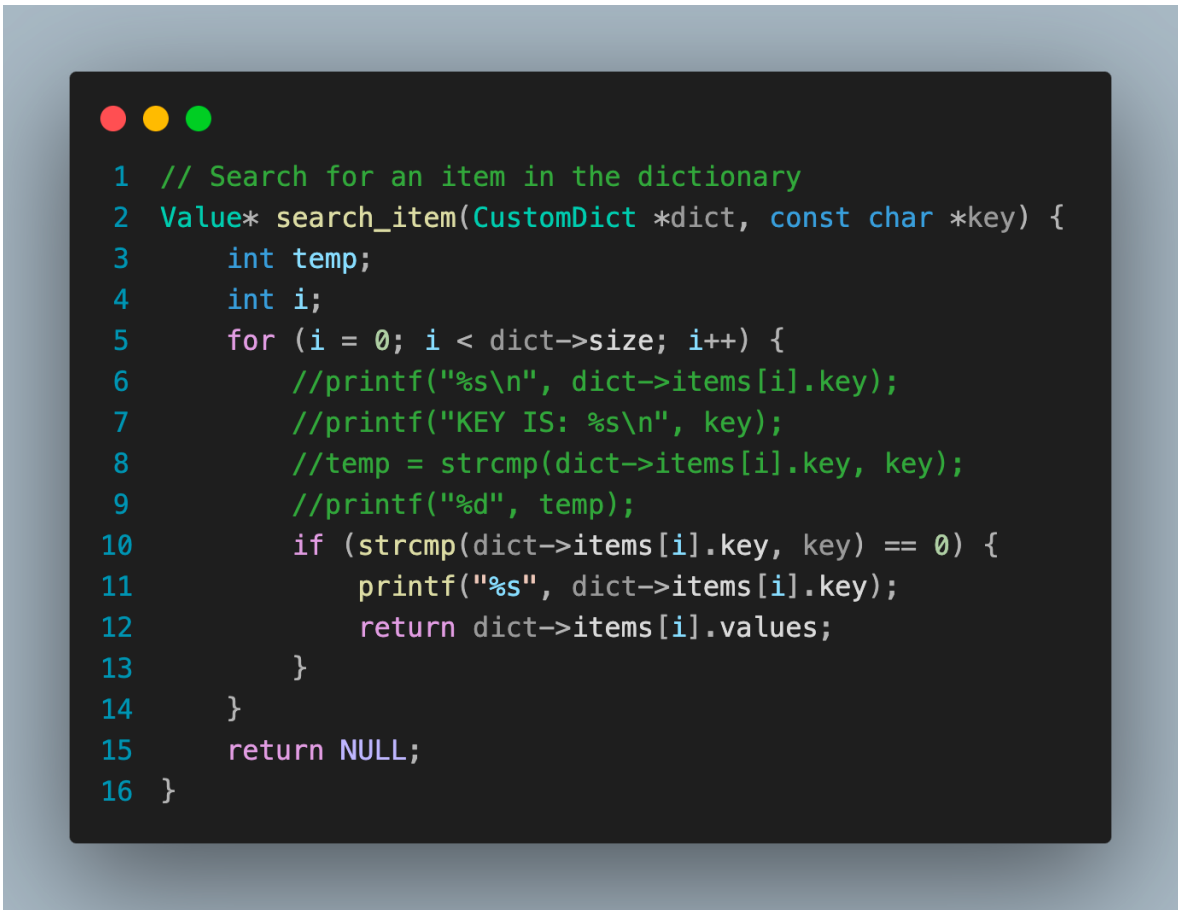
```c
void add_item(CustomDict *dict, const char *key, Value *values, int value_count, char type) {
    /*Check if key already exists in the dictionary*/
    int i,j;
    for (i = 0; i < dict->size; i++) {
        if (strcmp(dict->items[i].key, key) == 0) {
            /*If the key exists, merge existing and new values*/
            int new_value_count = dict->items[i].value_count + value_count; /*Calculate the new value count*/
            dict->items[i].values = (Value *)realloc(dict->items[i].values, new_value_count * sizeof(Value)); /*Reallocate memory for the new values array*/
            for (j = 0; j < value_count; ++j) {
                dict->items[i].values[dict->items[i].value_count + j] = values[j]; /*Copy the new values to the end of the existing values*/
            }
            dict->items[i].value_count = new_value_count;
            free(values); /*Free the temporary values array*/
            return;
        }
    }
    /*If the dictionary is full, increase its capacity*/
    if (dict->size == dict->capacity) {
        if (dict->capacity == 0) {
            dict->capacity = 1;
        } else {
            dict->capacity = (2 * dict->capacity);
        }
        dict->items = (Item *)realloc(dict->items, dict->capacity * sizeof(Item));
    }
    /*Add new item*/
    dict->items[dict->size].key = copy_string(key);
    dict->items[dict->size].values = values;
    dict->items[dict->size].value_count = value_count;
    dict->items[dict->size].type = type; // Veri türünü ata
    dict->size++;
}
```

The add_item function adds a new item to the CustomDict dictionary. It first checks if the key already exists in the dictionary. If the key exists, it merges the new values with the existing ones by reallocating memory for the values array and copying the new values to the end. It then updates the value count and frees the temporary values array. If the key does not exist and the dictionary is full, it increases the dictionary's capacity (doubling it) and reallocates memory for the items array. Finally, it adds the new item to the dictionary by copying the key, assigning the values, setting the value count, and assigning the type. The dictionary's size is incremented to reflect the addition of the new item.

```c
void delete_item(CustomDict *dict, const char *key)
{
    int i;
    for (i = 0; i < dict->size; ++i)
    {
        if (strcmp(dict->items[i].key, key) == 0)
        {
            free(dict->items[i].key);
            free(dict->items[i].values);
            dict->items[i] = dict->items[dict->size - 1];
            dict->size--;
            return;
        }
    }
}
```

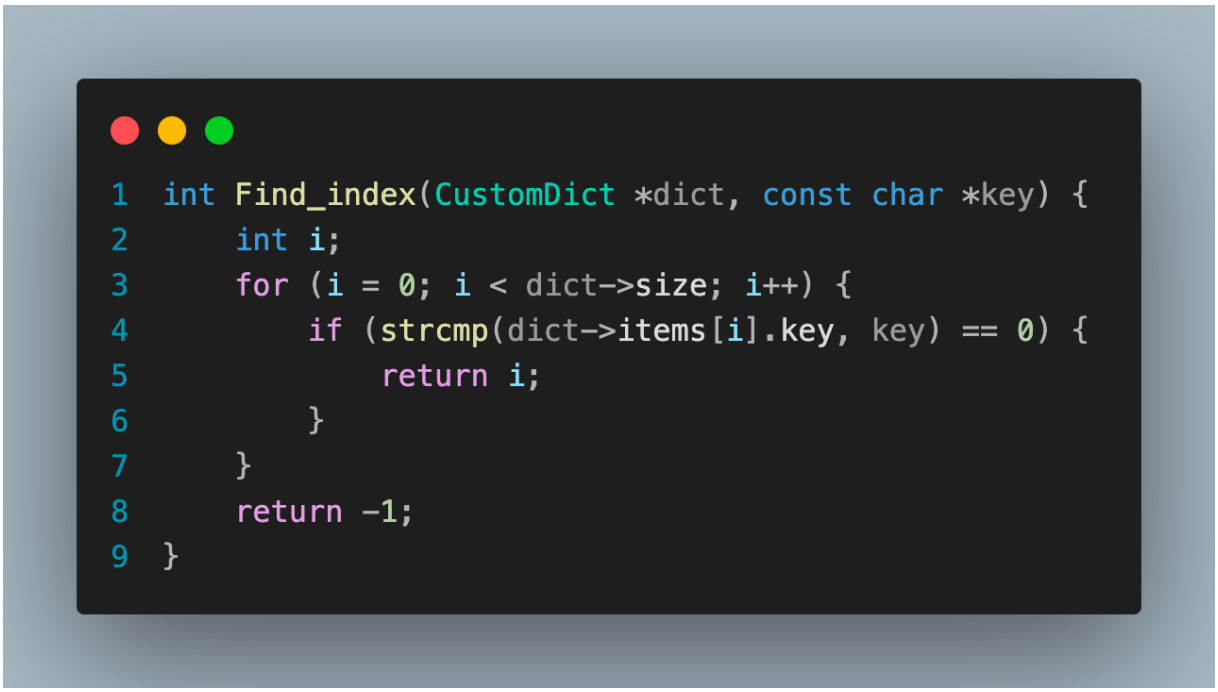
The `delete_item` function removes an item from the `CustomDict` dictionary based on the provided key. It iterates through the dictionary to find the item with the matching key. Once the key is found, it frees the memory allocated for the key and the values of that item. It then replaces the item with the last item in the dictionary to fill the gap and decrements the size of the dictionary by one. This effectively removes the specified item from the dictionary. If the key is not found, the function does nothing.

```c
// Set the value for an existing item
void set_value(CustomDict *dict, const char *key, Value *values, int value_count, char type) {
    delete_item(dict, key);
    add_item(dict, key, values, value_count, type);
}
```



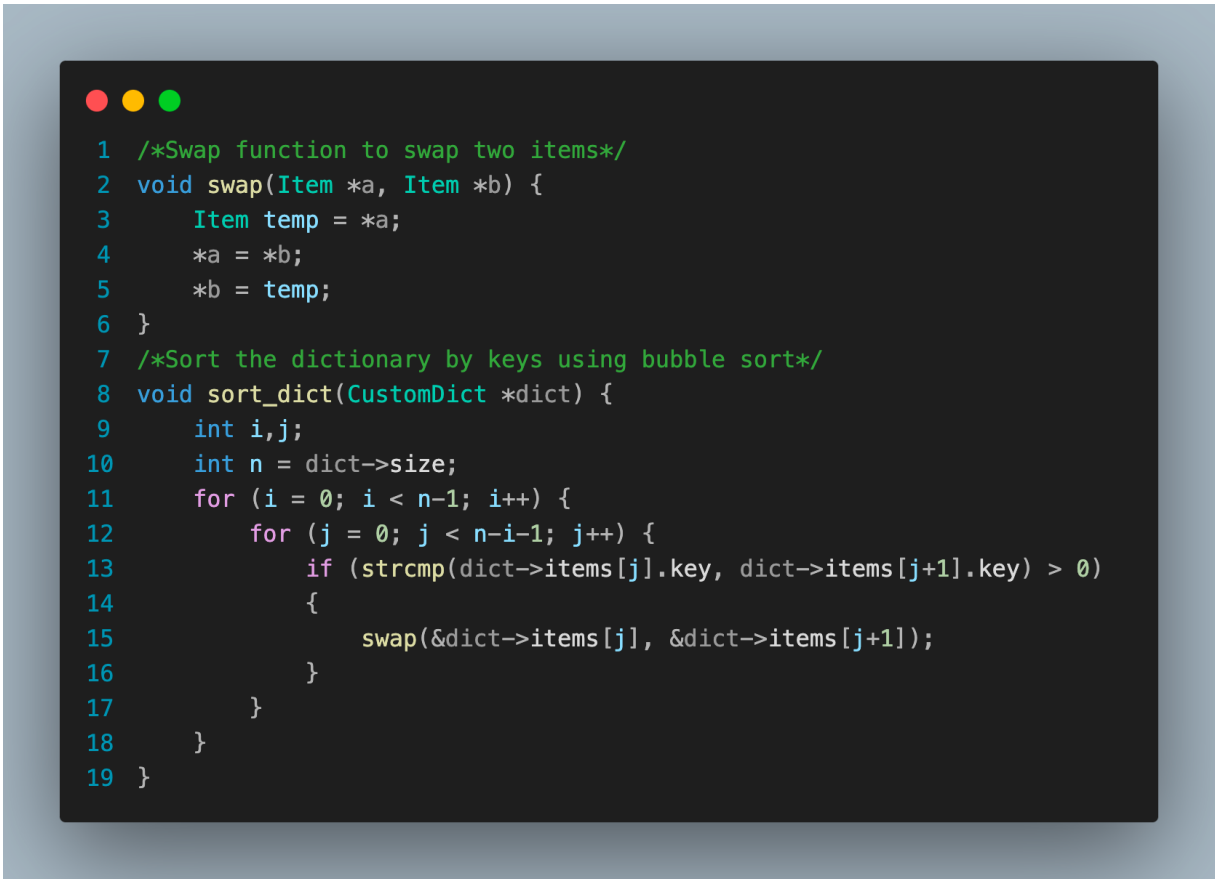
```c
// Search for an item in the dictionary
Value* search_item(CustomDict *dict, const char *key) {
    int temp;
    int i;
    for (i = 0; i < dict->size; i++) {
        //printf("%s\n", dict->items[i].key);
        //printf("KEY IS: %s\n", key);
        //temp = strcmp(dict->items[i].key, key);
        //printf("%d", temp);
        if (strcmp(dict->items[i].key, key) == 0) {
            printf("%s", dict->items[i].key);
            return dict->items[i].values;
        }
    }
    return NULL;
}
```

The search_item function searches for an item in the CustomDict dictionary by its key. It iterates through the dictionary, comparing each item's key with the provided key using strcmp. If a matching key is found, it prints the key (using a printf statement) and returns a pointer to the values associated with that key. If no matching key is found by the end of the loop, the function returns NULL, indicating that the item with the specified key does not exist in the dictionary.



```c
int Find_index(CustomDict *dict, const char *key) {
    int i;
    for (i = 0; i < dict->size; i++) {
        if (strcmp(dict->items[i].key, key) == 0) {
            return i;
        }
    }
    return -1;
}
```

The Find_index function searches for an item in the CustomDict dictionary by its key. It iterates through the dictionary, comparing each item's key with the provided key using strcmp. If a matching key is found, it returns the index of the item. If no matching key is found by the end of the loop, the function returns -1, indicating that the item with the specified key does not exist in the dictionary.

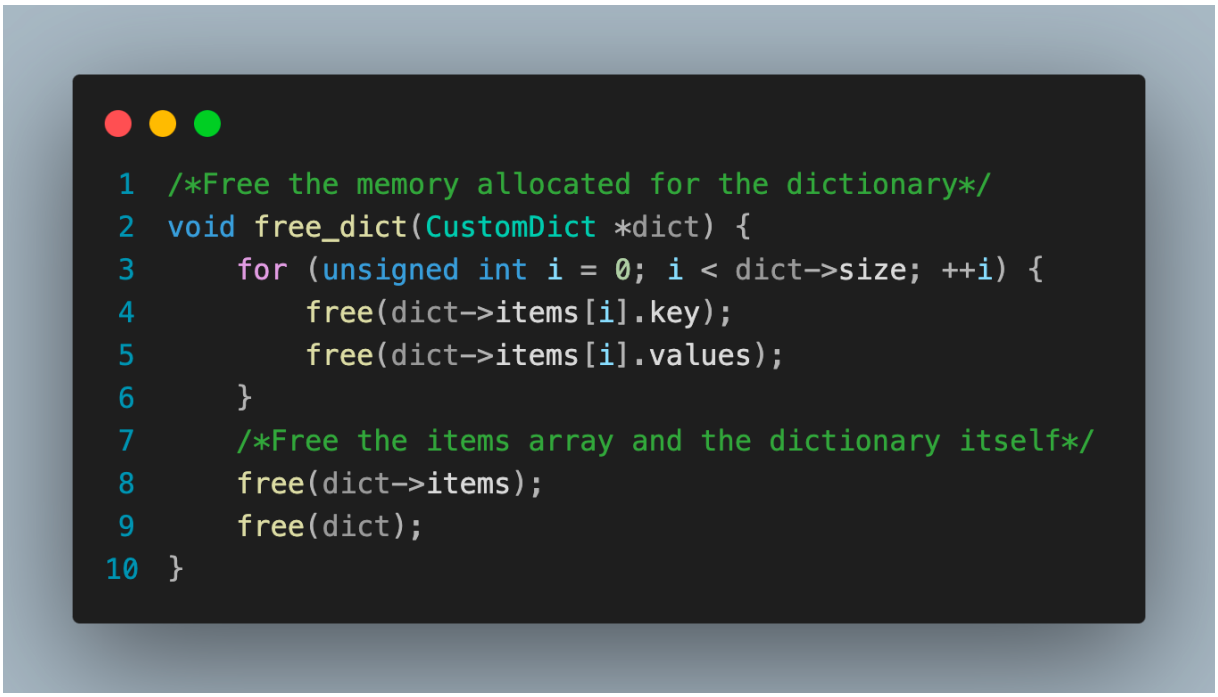


```c
/*Swap function to swap two items*/
void swap(Item *a, Item *b) {
    Item temp = *a;
    *a = *b;
    *b = temp;
}
/*Sort the dictionary by keys using bubble sort*/
void sort_dict(CustomDict *dict) {
    int i,j;
    int n = dict->size;
    for (i = 0; i < n-1; i++) {
        for (j = 0; j < n-i-1; j++) {
            if (strcmp(dict->items[j].key, dict->items[j+1].key) > 0)
            {
                swap(&dict->items[j], &dict->items[j+1]);
            }
        }
    }
}
```

The `swap` function exchanges the contents of two `Item` structs. The `sort_dict` function sorts the items in a `CustomDict` dictionary by their keys using the bubble sort algorithm. It iterates through the dictionary multiple times, comparing adjacent items' keys with `strcmp`. If an item's key is greater than the next item's key, the `swap` function is called to exchange them. This process continues until the entire dictionary is sorted in ascending order by key.

```c
// Print the dictionary
void print_dict(const CustomDict *dict) {
    int i,j;
    for (i = 0; i < dict->size; ++i) {
        printf("Key: %s, Values: ", dict->items[i].key);
        Value *values = dict->items[i].values;
        for (j = 0; j < dict->items[i].value_count; ++j) {
            switch (dict->items[i].type) {
                case 'i':
                    printf("%d ", values[j].i);
                    break;
                case 'f':
                    printf("%.2f ", values[j].f);
                    break;
                case 'd':
                    printf("%.2lf ", values[j].d);
                    break;
                case 'c':
                    printf("%c ", values[j].c);
                    break;
            }
        }
        printf("\n");
    }
}
```

The print_dict function prints all the items in a CustomDict dictionary. It iterates through each item in the dictionary, printing the key and its associated values. For each item's values, it checks the type ('i' for int, 'f' for float, 'd' for double, 'c' for char) and prints each value accordingly. After printing all the values for an item, it moves to the next item and continues until all items in the dictionary have been printed.


/*Free the memory allocated for the dictionary*/
void free_dict(CustomDict *dict) {
for (unsigned int i = 0; i < dict->size; ++i) {
free(dict->items[i].key);
free(dict->items[i].values);
}
/*Free the items array and the dictionary itself*/
free(dict->items);
free(dict);
}

```c
int read_csv(CustomDict *dict, const char *filename) {
    FILE *file = fopen(filename, "r");
    if (file == 0) {
        return 0; /*file cannot be opened*/
    }
    char line[1024];
    while (fgets(line, sizeof(line), file)) {
        char junkchar;
        char *token = strtok(line, ", ");
        char type = token[0]; /*Read the data type*/
        //fscanf(file, "%c", &junkchar);
        char *key = strtok(NULL, ", "); /*Read the key*/
        int value_count = 0;
        Value *values = NULL;
        while ((token = strtok(NULL, ",")) != NULL) {
            /*Reallocate memory for values array and store the new value*/
            values = (Value *)realloc(values, (value_count + 1) * sizeof(Value));
            switch (type) {
                case 'i':
                    values[value_count].i = atoi(token);
                    break;
                case 'f':
                    values[value_count].f = atof(token);
                    break;
                case 'd':
                    values[value_count].d = atof(token);
                    break;
                case 'c':
                    values[value_count].c = token[1];
                    break;
            }
            value_count++;
        }
        /*Add a null terminator for char arrays*/
        if (type == 'c') {
            values = (Value *)realloc(values, (value_count + 1) * sizeof(Value));
            values[value_count].c = '\0';
            value_count++;
        }
        add_item(dict, key, values, value_count, type);
    }
    fclose(file);
    return 1;
}
```

The `read_csv` function reads data from a CSV file and populates the `CustomDict` dictionary. It opens the specified file and reads each line into a buffer. For each line, it uses `strtok` to tokenize the line by commas, first extracting the data type and key. It then iterates through the remaining tokens, reallocating memory for the values array and converting the tokens to the appropriate data type (int, float, double, or char). If the type is `char`, it adds a null terminator. Once all values for a line are processed, it calls `add_item` to add the key and

values to the dictionary. The function returns 1 if the file is successfully read, or 0 if the file cannot be opened.

INPUT-OUTPUT

```
alperen@Alperen-MacBook-Pro-2 Assignment_12 % gcc --ansi -o program main.c customDict.c
alperen@Alperen-MacBook-Pro-2 Assignment_12 % ./program

Menu:
1. Add item
2. Delete item
3. Set value
4. Search item
5. Sort dictionary
6. Print dictionary
7. Read CSV file
8. Exit
Enter your choice:
```

Menu

INPUT-OUTPUT

```
Menu:
1. Add item
2. Delete item
3. Set value
4. Search item
5. Sort dictionary
6. Print dictionary
7. Read CSV file
8. Exit
Enter your choice: 7
Enter CSV filename: data.csv
CSV file read.

Menu:
1. Add item
2. Delete item
3. Set value
4. Search item
5. Sort dictionary
6. Print dictionary
7. Read CSV file
8. Exit
Enter your choice: 6
Key: age, Values: 25 12 556 1 192561
Key: weight, Values: 12.56 664.20 5.50
Key: blood_type, Values: A B C D E F  X Y Z
Key: height, Values: 10.50 20.25 30.75 40.20
Key: score, Values: 30 45 120
Key: price, Values: 9.80 7.60
Key: value, Values: 15.30 25.80 35.60 48.90 56.20
Key: quantity, Values: 5 10 15 20
Key: amount, Values: 18.90 24.70 33.60
Key: letter, Values: M N O P
Key: temperature, Values: 22.50 30.80 38.10 42.60
Key: count, Values: 8 16 32
Key: distance, Values: 55.40 67.90
Key: speed, Values: 40.50 55.20 60.90 75.30
Key: size, Values: 100 200 300 400
Key: balance, Values: 500.25 1000.50
```

INPUT-OUTPUT

```
Key: size, Values: 100 200 300 400
Key: balance, Values: 500.25 1000.50

Menu:
1. Add item
2. Delete item
3. Set value
4. Search item
5. Sort dictionary
6. Print dictionary
7. Read CSV file
8. Exit
Enter your choice: 2
Enter key to delete: balance

Menu:
1. Add item
2. Delete item
3. Set value
4. Search item
5. Sort dictionary
6. Print dictionary
7. Read CSV file
8. Exit
Enter your choice: 6
Key: age, Values: 25 12 556 1 192561
Key: weight, Values: 12.56 664.20 5.50
Key: blood_type, Values: A B C D E F  X Y Z
Key: height, Values: 10.50 20.25 30.75 40.20
Key: score, Values: 30 45 120
Key: price, Values: 9.80 7.60
Key: value, Values: 15.30 25.80 35.60 48.90 56.20
Key: quantity, Values: 5 10 15 20
Key: amount, Values: 18.90 24.70 33.60
Key: letter, Values: M N O P
Key: temperature, Values: 22.50 30.80 38.10 42.60
Key: count, Values: 8 16 32
Key: distance, Values: 55.40 67.90
Key: speed, Values: 40.50 55.20 60.90 75.30
Key: size, Values: 100 200 300 400

Menu:
1. Add item
2. Delete item
3. Set value
4. Search item
5. Sort dictionary
6. Print dictionary
7. Read CSV file
8. Exit
Enter your choice:
```

INPUT-OUTPUT

```
Menu:
1. Add item
2. Delete item
3. Set value
4. Search item
5. Sort dictionary
6. Print dictionary
7. Read CSV file
8. Exit
Enter your choice: 3
Enter key: alperen
Enter type (i: int, f: float, d: double, c: char): i
Enter values (enter 'e' to stop): 12 12 14 14 e

Menu:
1. Add item
2. Delete item
3. Set value
4. Search item
5. Sort dictionary
6. Print dictionary
7. Read CSV file
8. Exit
Enter your choice: 6
Key: age, Values: 25 12 556 1 192561
Key: weight, Values: 12.56 664.20 5.50
Key: blood_type, Values: A B C D E F  X Y Z
Key: height, Values: 10.50 20.25 30.75 40.20
Key: score, Values: 30 45 120
Key: price, Values: 9.80 7.60
Key: value, Values: 15.30 25.80 35.60 48.90 56.20
Key: quantity, Values: 5 10 15 20
Key: amount, Values: 18.90 24.70 33.60
Key: letter, Values: M N O P
Key: temperature, Values: 22.50 30.80 38.10 42.60
Key: count, Values: 8 16 32
Key: distance, Values: 55.40 67.90
Key: speed, Values: 40.50 55.20 60.90 75.30
Key: size, Values: 100 200 300 400
Key: alperen, Values: 12 12 14 14
```

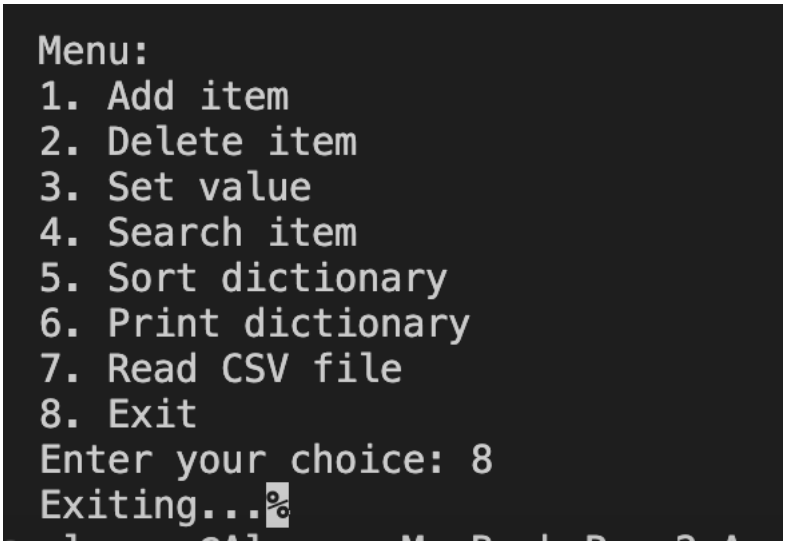
INPUT-OUTPUT

```
Menu:
1. Add item
2. Delete item
3. Set value
4. Search item
5. Sort dictionary
6. Print dictionary
7. Read CSV file
8. Exit
Enter your choice: 4
Enter key to search: amount
amount
Values: 18.90 24.70 33.60
```

INPUT-OUTPUT

```
Menu:
1. Add item
2. Delete item
3. Set value
4. Search item
5. Sort dictionary
6. Print dictionary
7. Read CSV file
8. Exit
Enter your choice: 6
Key: age, Values: 25 12 556 1 192561
Key: alperen, Values: 12 12 14 14
Key: amount, Values: 18.90 24.70 33.60
Key: blood_type, Values: A B C D E F  X Y Z
Key: count, Values: 8 16 32
Key: distance, Values: 55.40 67.90
Key: height, Values: 10.50 20.25 30.75 40.20
Key: letter, Values: M N O P
Key: price, Values: 9.80 7.60
Key: quantity, Values: 5 10 15 20
Key: score, Values: 30 45 120
Key: size, Values: 100 200 300 400
Key: speed, Values: 40.50 55.20 60.90 75.30
Key: temperature, Values: 22.50 30.80 38.10 42.60
Key: value, Values: 15.30 25.80 35.60 48.90 56.20
Key: weight, Values: 12.56 664.20 5.50
```

INPUT-OUTPUT



YOUTUBE LINK:

https://www.youtube.com/watch?v=IhGxVAKH9YI