Essential Information

To compile the program, run the command “make” in the terminal. This will use the provided Makefile to compile all source files and create an executable named “myprogram”.

To run the program, type “./myprogram” in the terminal. No additional arguments are needed since the file path names are already included in the main.c file. To remove the executable files afterwards, type “make clean” into the command line.

The program loads UK property sales data from CSV files into memory. It builds a hash table with the indexing based on street names. Then, it compares searching for a street using a linear scan versus using the hash index.

Discussion

1. The fixed-size street array is usually faster to access than the dynamically allocated district string in terms of memory access speed. This is because the variable "street" is stored directly inside each record’s memory block, so the computer can quickly find it using a fixed offset. The variable "district" is a pointer to a separate spot in your computer's memory (the heap). This means the computer has to follow that pointer to get the actual data. Even though "district" doesn’t waste memory by being sized exactly to the string length, accessing it takes a little more time because it’s not stored contiguously with the rest of the record.

2. Although hash table search is expected to be very fast with O(1) time complexity, real-world measurements often show it’s not dramatically faster than linear search, which is O(n). This happens because hash tables rely on pointers and scattered memory locations, causing the computer’s cache to be less efficient. Linear search is slower theoretically, but it scans through data stored in one continuous block of memory. This makes better use of the CPU cache. So, memory access patterns and caching reduce the actual speed difference between the two methods, even if the hash search has fewer operations.

Appendix

myDSlib.c

```c
// CODE 1: Include necessary library(ies)
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <time.h>
#include <stddef.h>
#include <stdbool.h>
#include <stdlib.h>
#include "myDSlib.h"

// ------------------
Record *table = NULL;  // DO NOT CHANGE
size_t table_size = 0; // DO NOT CHANGE

// CODE 3: Implement all the functions here

void print_record(const Record *r)
{
    printf("Match found:\n");
    printf("Price          : %u\n", r->price);
    printf("Date           : %04d-%02d-%02d\n", r->date.year, r->date.month, r-
>date.day);
    printf("Postcode       : %s\n", r->postcode);
    printf("Street         : %s\n", r->street);
    printf("\n");
}

int count_unused_slots(IndexEntry **index_on_street)
{
    int count = 0;
    for (size_t i = 0; i < INDEX_SIZE; i++)
    {
        if (index_on_street[i] == NULL)
        {
            count++;
        }
    }
    return count;
}

unsigned int hash_string(const char *str)
{
    unsigned long hash = 5381;
    int c;
```

```
    while ((c = *str++))
        hash = ((hash << 5) + hash) + c;

    return (unsigned int)(hash % INDEX_SIZE);
}

void read_file(char *file_name)
{
    static size_t table_capacity = 0;
    FILE *my_file = fopen(file_name, "r");

    if (!my_file)
    {
        perror(file_name);
        return;
    }

    char line[512];

    while (fgets(line, sizeof(line), my_file))
    {
        line[strcspn(line, "\r\n")] = '\0';

        if (table_size == table_capacity)
        {
            size_t new_capacity = table_capacity ? table_capacity * 2 : 1024;
            Record *new_table = realloc(table, new_capacity * sizeof(Record));
            if (!new_table)
            {
                perror("realloc");
                fclose(my_file);
                return;
            }
            table = new_table;
            table_capacity = new_capacity;
        }

        Record *r = &table[table_size];
        char *field = strtok(line, ",");
        int col = 0;

        while (field)
        {
            switch (col)
            {
            case 0:
                strncpy(r->transaction_id, field, MAX_TRANSACTION_ID_LEN - 1);
```

```
                r->transaction_id[MAX_TRANSACTION_ID_LEN - 1] = '\0';
                break;
            case 1:
                r->price = (unsigned int)strtoul(field, NULL, 10);
                break;
            case 2:
                sscanf(field, "%d-%d-%d", &r->date.year, &r->date.month, &r-
>date.day);
                break;
            case 3:
                strncpy(r->postcode, field, MAX_POSTCODE_LEN - 1);
                r->postcode[MAX_POSTCODE_LEN - 1] = '\0';
                break;
            case 4:
                r->property_type = field[0];
                break;
            case 5:
                r->old_new = field[0];
                break;
            case 6:
                r->duration = field[0];
                break;
            case 7:
                strncpy(r->paon, field, MAX_FIELD_LEN - 1);
                r->paon[MAX_FIELD_LEN - 1] = '\0';
                break;
            case 8:
                strncpy(r->saon, field, MAX_FIELD_LEN - 1);
                r->saon[MAX_FIELD_LEN - 1] = '\0';
                break;
            case 9:
                strncpy(r->street, field, MAX_FIELD_LEN - 1);
                r->street[MAX_FIELD_LEN - 1] = '\0';
                break;
            case 10:
                strncpy(r->locality, field, MAX_FIELD_LEN - 1);
                r->locality[MAX_FIELD_LEN - 1] = '\0';
                break;
            case 11:
                strncpy(r->town, field, MAX_FIELD_LEN - 1);
                r->town[MAX_FIELD_LEN - 1] = '\0';
                break;
            case 12:
                r->district = malloc(strlen(field) + 1);
                if (r->district)
                    strcpy(r->district, field);
                break;
            case 13:
```

```
                strncpy(r->county, field, MAX_FIELD_LEN - 1);
                r->county[MAX_FIELD_LEN - 1] = '\0';
                break;
            case 14:
                r->record_status = field[0];
                break;
            case 15:
                r->blank_col = field[0];
                break;
            default:
                break;
            }

            field = strtok(NULL, ",");
            col++;
        }

        table_size++;
    }

    fclose(my_file);
}

IndexEntry **createIndexOnStreet(Record *table, size_t table_size)
{
    IndexEntry **index = calloc(INDEX_SIZE, sizeof(IndexEntry *));
    if (!index)
    {
        perror("calloc index");
        return NULL;
    }

    for (size_t i = 0; i < table_size; i++)
    {
        Record *rec = &table[i];
        unsigned int bucket = hash_string(rec->street);

        IndexEntry *newEntry = malloc(sizeof(IndexEntry));
        if (!newEntry)
        {
            perror("malloc newEntry");
            free_index(index);
            return NULL;
        }

        newEntry->key = strdup(rec->street);
        if (!newEntry->key)
        {
```

```
            perror("strdup");
            free(newEntry);
            free_index(index);
            return NULL;
        }
        newEntry->record_ptr = rec;
        newEntry->next = index[bucket];
        index[bucket] = newEntry;
    }

    return index;
}

void searchStreetLinear(Record *table, size_t table_size, const char *target_street,
bool printFlagLinearSearch)
{
    bool found = false;
    for (size_t i = 0; i < table_size; i++)
    {
        if (strcmp((table + i)->street, target_street) == 0)
        {
            found = true;
            if (printFlagLinearSearch)
            {
                print_record(table + i);
            }
        }
    }
    if (!found && printFlagLinearSearch)
    {
        printf("No records found for street: %s\n\n", target_street);
    }
}

void searchStreet(IndexEntry **index, const char *target_street, bool
printFlagHashIndexSearch)
{
    unsigned int bucket = hash_string(target_street);
    IndexEntry *curr = index[bucket];
    bool found = false;

    while (curr)
    {
        if (strcmp(curr->key, target_street) == 0)
        {
            found = true;
            if (printFlagHashIndexSearch)
            {
```

```
                print_record(curr->record_ptr);
            }
        }
        curr = curr->next;
    }

    if (!found && printFlagHashIndexSearch)
    {
        printf("No records found in hash index for street: %s\n\n", target_street);
    }
}

void free_index(IndexEntry **index_on_street)
{
    if (!index_on_street)
        return;

    for (size_t i = 0; i < INDEX_SIZE; i++)
    {
        IndexEntry *curr = index_on_street[i];
        while (curr)
        {
            IndexEntry *tmp = curr;
            curr = curr->next;
            free(tmp->key);
            free(tmp);
        }
    }

    free(index_on_street);
}

void free_table()
{
    if (!table)
        return;
    for (size_t i = 0; i < table_size; i++)
    {
        free(table[i].district);
        table[i].district = NULL;
    }
    free(table);
    table = NULL;
    table_size = 0;
}
```

myDSlib.h

```c
#ifndef MYDSLIB_H
#define MYDSLIB_H
// CODE 1: Include necessary library(ies)
#include <stddef.h>
#include <stdbool.h>


#define MAX_FIELD_LEN 100          // DO NOT CHANGE: for more info check Record
struct (we know it is always less than 100 chars)
#define MAX_TRANSACTION_ID_LEN 39   // DO NOT CHANGE: for more info check Record
struct (we know this column is always 38 character plus one null terminator)
#define MAX_POSTCODE_LEN 9          // DO NOT CHANGE: for more info check Record
struct (we know that this column has max 8+1 chars)


#define INDEX_SIZE 180001  // CODE 2: you must find a proper value that balances
size/performance
/*
I experimented and tried to pick a prime number that would give me a load factor of
around 0.7. (Prime nums I believe work better for modulo)
*/

// CODE 3: You can add more constants with #define here if needed

// DO NOT CHANGE: Date
typedef struct {
    int year, month, day;
} Date;

// DO NOT CHANGE: Record
typedef struct {
    char transaction_id[MAX_TRANSACTION_ID_LEN];
    unsigned int price;
    Date date;
    char postcode[MAX_POSTCODE_LEN];
    char property_type;      // One character
    char old_new;            // One character
    char duration;           // One character
    char paon[MAX_FIELD_LEN];
    char saon[MAX_FIELD_LEN];
    char street[MAX_FIELD_LEN];
    char locality[MAX_FIELD_LEN];
    char town[MAX_FIELD_LEN];
    char *district;          // Allocated dynamically
    char county[MAX_FIELD_LEN];
    char record_status;      // One character
```

```
    char blank_col;          // One character
} Record;

// DO NOT CHANGE: IndexEntry
typedef struct IndexEntry {
    char* key;           // The street name (copied from table[i].street)
    Record* record_ptr;  // Pointer back to original record
    struct IndexEntry* next;
} IndexEntry;


// Global table pointer
extern Record* table;    // DO NOT CHANGE: A global pointer to save the read data from
csv files (a pointer to array of Records)
extern size_t table_size;// DO NOT CHANGE: A global variable showing the number of
rows (number of Records) read from csv files (rows in table)
// CODE 4: ADD more global variables if you need

// CODE 5: Declare necessary functions here (only the functions being used files other
than myDSlib.c)
void read_file(char* file_name);
IndexEntry** createIndexOnStreet(Record* table, size_t table_size);
void searchStreetLinear(Record *table, size_t table_size, const char *target_street,
bool printFlagLinearSearch);
void searchStreet(IndexEntry **index, const char *target_street, bool
printFlagHashIndexSearch);
void free_index(IndexEntry **index_on_street);
void free_table();
int count_unused_slots(IndexEntry **index_on_street);

#endif
```