Московский авиационный институт

(Национальный исследовательский университет)

Факультет прикладной математики и физики

Курсовой проект

по курсу

«Методы и языки программирования»

II семестр

Задание 9

Выполнила: Довженко А.

Группа: М8О-107Б-16, №5

Проверили: Ридли А.Н.

Ридли М.К.

Оценка:

Дата:

Москва

2017г.

Задача:

Метод сортировки: 12. Быстрая сортировка Хоара (нерекурсивный вариант)
Структура таблицы: 4. Тип ключа строковый, хранение ключа и значения вместе.

Составить и отладить программу на языке Си с использованием процедур и функций для сортировки таблицы заданным методом и двоичного поиска по ключу в таблице. Программа должна вводить значения элементов неупорядоченной таблицы и проверять работу процедуры сортировки в трех случаях: (1) элементы таблицы с самого начала упорядочены; (2) элементы таблицы расставлены в обратном порядке; (3) элементы таблицы не упорядочены. В последнем случае можно использовать встроенные процедуры генерации псевдослучайных чисел.
В качестве текста для записей взять изображение ASCII-графики.

Введение:

Задание курсового проекта очевидным образом распадается на две связанные задачи – задачу поиска по ключу в таблице и задачу сортировки элементов таблицы. Для их решения необходимо определить, что есть поиск по ключу и что есть сортировка. Так как нам предложен единственный вариант сортировки, то вопрос о ее выборе не стоит, но мы все же рассмотрим сложности других сортировок и их применимость.

Задача поиска по ключу в таблице отличается от задачи поиска в массиве тем, что в таблицах ключ обычно является составным и имеет регулярную структуру (в нашем случае он представляет собой строку), т.е. сам является массивом. Соответственно, при каждом сравнении ключей, проверка совпадения строк осуществляется их политерным сравнением до первого расхождения. То есть ищется неравенство двух букв с одинаковыми индексами. В случае неуспеха строки равны. В Си используется представление строк с терминирующим знаком в конце строки, поэтому их сравнение не представляет сложности.

Задачу поиска предложено решить с помощью алгоритма бинарного поиска, который выглядит следующим образом:

1. Определить значения элемента в середине структуры данных. Полученное значение сравнивается с ключом.
2. Если ключ меньше значения середины, то поиск осуществляется в первой половине элементов, иначе - во второй.

3. Поиск сводится к тому, что вновь определяется значение серединного элемента в выбранной половине и сравнивается с ключом.
4. Процесс продолжается до тех пор, пока не будет найден элемент со значением ключа или не станет пустым интервал для поиска.

Как можно понять из задания, сортируемые данные не являются изолированными значениями. Все данные входят в состав записей. В каждой записи содержится ключ, представляющий собой сортируемое значение, в то время как остальная часть записи состоит из сопутствующих данных, дополняющих ключ. Алгоритм сортировки должен быть реализован так, чтобы он вместе с ключами переставлял и сопутствующие данные. Преобразование алгоритма, предназначенного для сортировки чисел, в программу для сортировки записей не представляет концептуальных трудностей.

Нерекурсивная сортировка Хоара отличается от рекурсивной тем, что использует явный стек, заменяя рекурсивные вызовы помещением в стек параметров, а рекурсивные вызовы процедур и выходы из них – циклом, выталкивающим параметры из стека и обрабатывающим их, пока стек не пуст.

В нашей реализации функция принимает в качестве аргумента таблицу. Потом мы задаем левую и правую границу этой таблицы. В самом начале левая граница это 0, а правая – размер таблицы минус один. Нам понадобятся следующие переменные: size_t i, j указывают на левый и правый элементы. Под элементом будем понимать ключ. Key pivot хранит значение опорного элемента. Сохраняем в стек крайнее левое и правое значения. Можно сохранять сразу пару значений, но мы вместо этого сделаем два параллельных стека. В первый (lows) будем класть крайнее левое значение для следующего вызова, а во второй (highs) - крайнее правое.

Сначала задаём начальные значения
i = low, j = high, pivot = table->notr_arr[(low + high/2)];

Здесь следует сразу же оговориться. Выбирать всегда средний элемент в качестве опорного – достаточно рискованно. Если известно, какой элемент будет выбран в качестве опорного, то можно подобрать такую последовательность, для которой сортировка будет происходить максимально медленно, за время порядка n^2. Поэтому в качестве элемента, относительно которого будет сортировка, берут либо случайный элемент, либо медиану из первого, последнего и среднего элементов.

Пока i будет меньше j (пока они не пересекутся) делаем следующее. Во-первых, нужно пропустить все уже отсортированные элементы. Далее, если границы ещё не пересеклись, то в случае, если порядок нарушен, то следует обменять местами переменные, после чего опять инкрементировать i и декрементировать j, чтобы не зациклиться. После этого цикла i и j пересекутся, i станет больше j, и мы получим ещё два массива, границы которых нужно поместить в стек: массив от левой границы до i, и массив от j до правой границы, если, конечно, мы не вышли за пределы границ.

Цикл заканчивается, когда стеки становятся пустыми.

Рассматривать алгоритмы других сортировок в данной работе мы не будем, только сравним их сложности и рассмотрим применимость.

Сортировка методом вставки в наихудшем случае выполняется за время $O(n^2)$. Несмотря на далеко не самое оптимальное асимптотическое поведение этого алгоритма, благодаря компактности его внутренних циклов он быстро справляется с сортировкой массивов с небольшим количеством элементов «на месте» (без дополнительной памяти, т.е. без выделения отдельного массива для работы и хранения выходных данных).

Время работы пирамидальной сортировки, как и время работы алгоритма сортировки слиянием (и в отличие от времени работы алгоритма сортировки вставкой), равно $O(nlgn)$. Как и сортировка методом вставок, и в отличие от сортировки слиянием, пирамидальная сортировка выполняется без привлечения дополнительной памяти: в любой момент времени требуется память для хранения вне массива только некоторого постоянного количества элементов. Таким образом, в пирамидальной сортировке сочетаются лучшие особенности двух вышеупомянутых сортировок.

Алгоритм быстрой сортировки, реализованный в данной работе, также сортирует n чисел «на месте», поэтому он хорошо работает даже в средах с виртуальной памятью, время его работы в наихудшем случае равно $O(n^2)$. Тем не менее, в среднем этот алгоритм выполняется за время $O(nlgn)$ и на практике по производительности превосходит алгоритм пирамидальной сортировки. Код алгоритма быстрой сортировки такой же компактный, как и код алгоритма сортировки вставкой. Алгоритм быстрой сортировки приобрел широкую популярность для сортировки больших массивов.

Алгоритмы сортировки, работающие по методы вставок и слияния, а также алгоритмы пирамидальной и быстрой сортировки имеют одну общую

особенность – все они работают по принципу попарного сравнения элементов входного массива.

Алгоритм сортировки выборкой имеет сложность $O(n^2)$. Как правило, алгоритм сортировки выборкой предпочтительнее простой вставки. Однако, если ключи в начале упорядочены или почти упорядочены, алгоритм вставки будет оставаться несколько более быстрым.

Сортировка Шелла является неустойчивой и может привести к резонансному явлению (трешингу) в системах с виртуальной памятью, когда члены последовательности попадают на разные страницы виртуальной памяти. Математический анализ сортировки Шелла исключительно сложен. Общая степенная сложностная оценка сортировок Шелла $O\left(n^{1+\delta}\right), 0 < \delta < 1$. Сортировка Шелла существенно быстрее простых сортировок с их квадратичной сложностью.

Решение:

Структура таблицы представляет собой массив, состоящий из записей, в которые входит ключ и значение. Ключ и значение реализованы в разных модулях программы и представляют собой строки фиксированной длины. Ввод таблицы осуществляется со стандартного потока ввода. Действия, доступные пользователю: ввод таблицы, печать таблицы, сортировка таблицы, поиск записи по ключу, удаление записи по ключу, реверс таблицы, перемешивание записей в таблице, удаление таблицы.

Организация программы:

Модуль	Действие
main.c	Организация ввода и вывода
table.h, table.c	Реализация таблицы и функций над ней
data.h, data.c	Реализация записи и функций над ней
key.h, key.c	Реализация ключа и функций над ним
stack.h, stack.c	Вспомогательный модуль, реализующий стек, для нерекурсивной сортировки Хоара

main.c:

Программа управляется через этот файл. Команды insert и ins добавляют запись в таблицу. Команды print и p печатают таблицу. Команды run и r сортируют таблицу, используя нерекурсивную сортировку Хоара. Команды find и f находят элемент по ключу. Команды delete и del удаляют

запись из таблицы. Команды reverse и rev реверсят таблицу. Команды scramble и scr перемешивают записи таблицы. Команды destroy и des удаляют таблицу. Команды quit и q организуют выход из программы.

key.h, key.c:

```
typedef struct _key {
   char value[KEY_LEN];
} *Key;
```

Как видно из объявления структуры, ключ – это массив литер. KEY_LEN, согласно заданию варианта, равен 6 (байтам).

Key key_create(void);
Создание ключа key.

void key_destroy(Key *key);
Освобождение памяти, выделенной под ключ *key.

void key_insert(Key key);
Вставка введенного значения в ключ.

void key_print(Key key);
Печать ключа.

int32_t key_cmp(Key key1, Key key2);
Сравнение ключей на основе лексикографической упорядоченности. Если ключи равны, возвращается 0, если key1 > key2 возвращается 1, если key1 < key2 возвращается -1.

uint32_t key_len(void);
Вспомогательная функция, возвращающая KEY_LEN.

data.h, data.c:

```
typedef struct _data {
   char value[DATA_LEN];
} *Data;
```

Структура значения записи аналогична структуре ключа и представляет собой массив литер.

Data data_create(void);
Создание значения записи data.

void data_destroy(Data *data);
Освобождение памяти, выделенной под значение *data.

void data_insert(Data data);
Вставка введенного значения в значение записи.

void data_print(Data data);
Печать значения.

uint32_t data_len(void);
Вспомогательная функция, возвращающая DATA_LEN.

table.h, table.c:

```
typedef struct _note Note;
typedef struct _table *Table;

struct _note {
    Key key;
    Data data;
};

struct _table {
    Note *note_arr;
    uint32_t size;
    bool sorted;
};
```

В структуре _note организуем совместное хранение ключа и значения. Структура таблицы представляет собой массив записей _note, размер этого массива и булево значение, отсортирован массив или нет. Последнее сделано для удобства, в общем случае можно реализовать функцию проверки, отсортирована ли таблица.

Table table_create(void);
Создание таблицы.

void table_destroy(Table *table);
Освобождение памяти, выделенной под таблицу *table.

void table_push(Table table, uint32_t rows);
Вставка в таблицу rows записей.

void table_note_delete(Table table, Key key);
Удаление записи из таблицы.

void table_qsort(Table table);
Реализация нерекурсивной сортировки Хоара.

Note *table_binary_search(Table table, Key key, size_t *pos);
Бинарный поиск записи в таблице по ключу.

void table_print(Table table);
Печать таблицы table.

void note_print(Note *note);
Печать записи *note.

bool table_is_sorted(Table table);
Проверка, отсортирована ли таблица.

void table_reverse(Table table);
Расстановка записей таблицы в обратном порядке.

void table_scramble(Table table);
Перемешивание записей таблицы.

stack.h, stack.c:

```
typedef int Item;
typedef struct _stack Stack;
struct _stack {
   int ptr;
   Item *data;
};
```

Реализация стека на массиве.

Stack *stack_create(size_t stack_size);
Создание стека. Аргумент функции stack_size нужен для ортогональности, на самом деле память под стек выделяется динамически.

void stack_push(Stack *s, Item value);
Добавление элемента в стек.

Item stack_pop(Stack *s);
Изъятие элемента из стека.

bool stack_is_empty(Stack *s);
Проверка, пуст ли стек.

void stack_destroy(Stack **s);
Удаление стека.

Item stack_top(Stack *s);
Получение последнего добавленного элемента в стеке.

Вывод:

Метод бинарного поиска широко распространен в задачах поиска в структурах данных. Кроме структур данных, построенных на связных списках, потому что бинарный поиск должен иметь возможность обратиться к любому элементу данных по индексу. Реализация алгоритма достаточно простая, и он быстро работает (за $O(lgn)$).

Задача сортировки является наиболее фундаментальной задачей при изучении алгоритмов. Зачастую в приложении не обойтись без сортировки информации. В данной работе мы убедились, что даже для решения такой тривиальной задачи, как бинарный поиск, сортировка необходима. Часто в алгоритмах сортировка используется в качестве основной подпрограммы.

Имеется большой выбор алгоритмов сортировки, в которых применяются самые разные технологии. Фактически в алгоритмах сортировки используются многие важные методы (зачастую разработанные еще на заре компьютерной эры), применяемые при разработке различных классов алгоритмов. В процессе реализации алгоритмов сортировки на передний план выходят многие прикладные проблемы. Выбор наиболее производительной программы сортировки в той или иной ситуации может зависеть от многих факторов, таких как предварительные знания о ключах и сопутствующих данных, об иерархической организации памяти компьютера (наличии кэша и виртуальной памяти) и программной среды. Многие из этих вопросов можно рассматривать на уровне алгоритмов, а не кода.

Код программы:

main.c

```
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <inttypes.h>

#include "data.h"
#include "key.h"
#include "table.h"

uint32_t main(void)
{
```

```
uint32_t rows = 0;
size_t pos;
Table table = table_create();
Key key = key_create();

char s[9];

printf("\nДля получения помощи в использовании программы напишите 'help' или 'h'.\n\n");
while (1) {
   scanf("%8s", s);
   if (!strcmp(s, "insert") || !strcmp(s, "ins")) {
      if(table == NULL)
         table = table_create();
      printf("Введите количество строк:\n");
      scanf("%u ", &rows);
      table_push(table, rows);
   } else if (!strcmp(s, "delete") || !strcmp(s, "del")) {
      if(table == NULL) printf("Таблицы не существует, воспользуйтесь командами 'help' или 'h'.\n");
      else {
         key_insert(key);
         table_note_delete(table, key);
      }
   } else if (!strcmp(s, "quit") || !strcmp(s, "exit") || !strcmp(s, "q")) {
      if (table != NULL) table_destroy(&table);
      if (key != NULL) key_destroy(&key);
      break;
   } else if (!strcmp(s, "run") || !strcmp(s, "r")) {
      if(table == NULL)
         printf("Таблицы не существует, воспользуйтесь командами 'help' или 'h'.\n");
      else
         table_qsort(table);
   } else if (!strcmp(s, "print") || !strcmp(s, "p")) {
      if (table == NULL)
         printf("Таблицы не существует, воспользуйтесь командами 'help' или 'h'.\n");
      else
         table_print(table);
   } else if (!strcmp(s, "find") || !strcmp(s, "f")) {
      if (table == NULL) printf("Таблицы не существует, воспользуйтесь командами 'help' или 'h'.\n");
      else {
         key_insert(key);
         Note *note_found = table_binary_search(table, key, &pos);
         if (note_found == NULL) {
            printf("Записи с таким ключом нет в таблице.\n");
         } else {
            note_print(note_found);
         }
      }
   } else if (!strcmp(s, "destroy") || !strcmp(s, "des")) {
      if (table == NULL) printf("Таблицы не существует, воспользуйтесь командами 'help' или 'h'.\n");
      else {
         table_destroy(&table);
      }
   } else if (!strcmp(s, "reverse") || !strcmp(s, "rev")) {
      if (table == NULL) printf("Таблицы не существует, воспользуйтесь командами 'help' или 'h'.\n");
      else {
         table_reverse(table);
      }
   } else if (!strcmp(s, "scramble") || !strcmp(s, "scr")) {
      if (table == NULL) printf("Таблицы не существует, воспользуйтесь командами 'help' или 'h'.\n");
      else {
         table_scramble(table);
      }
   } else if (!strcmp(s, "help") || !strcmp(s, "h")) {
      printf("\n\nКоманды 'insert' и 'ins' добавляют запись в таблицу.\n");
      printf("\nКоманды 'print' и 'p' печатают таблицу.\n");
      printf("\nКоманды 'run' и 'r' сортируют таблицу, используя нерекурсивную сортировку Хоара.\n");
      printf("\nКоманды 'find <key>' и 'f <key>' находят элемент по ключу.\n");
      printf("\nКоманды 'delete <key>' и 'del <key>' удаляют запись из таблицы.\n");
      printf("\nКоманды 'reverse' и 'rev' реверсят таблицу.\n");
```

```
            printf("\nКоманды 'scramble' и 'scr' перемешивают записи таблицы.\n");
            printf("\nКоманды 'destroy' и 'des' удаляют таблицу.\n");
            printf("\nКоманды 'quit' и 'q' выходят из программы.\n");
        }
    }
    return 0;
}
```

table.h

```
#ifndef _TABLE_H_
#define _TABLE_H_

#include <stdbool.h>
#include <string.h>
#include <inttypes.h>

#include "key.h"
#include "data.h"

typedef struct _note Note;
typedef struct _table *Table;

Table table_create(void);
void table_destroy(Table *table);
void table_push(Table table, uint32_t rows);
void table_note_delete(Table table, Key key);
void table_qsort(Table table);
Note *table_binary_search(Table table, Key key, size_t *pos);
void table_print(Table table);
void note_print(Note *note);
bool table_is_sorted(Table table);
void table_reverse(Table table);
void table_scramble(Table table);

#endif
```

table.c

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdbool.h>
#include <string.h>
#include <inttypes.h>
#include <time.h>

#include "table.h"
#include "key.h"
#include "data.h"
#include "stack.h"

#define stack_size 1
#define SWAP(A,B) {Note T = A; A = B; B = T; };

struct _note {
    Key key;
    Data data;
};
struct _table {
    Note *note_arr;
    uint32_t size;
    bool sorted;
};

uint32_t randomAB(int a, int b);

Table table_create(void)
{
    Table table = (Table) calloc(1, sizeof(*table));
```

```
    if(!table) {
        fprintf(stderr, "Error:no memory\n");
        exit(1);
    }
    table->note_arr = NULL;
    table->size = 0;
    table->sorted = false;
    return table;
}

void table_destroy(Table *table)
{
    for (uint32_t i = 0; i < (*table)->size - 1; ++i) {
        key_destroy(&((*table)->note_arr[i].key));
        data_destroy(&((*table)->note_arr[i].data));
    }
    free(*table);
    *table = NULL;
}

void table_push(Table table, uint32_t rows)
{
    for(uint32_t i = 0; i < rows; ++i) {
        Key tmp_key = key_create();
        Data tmp_data = data_create();
        fscanf(stdin, "%s", tmp_key->value);
        fscanf(stdin, "%s", tmp_data->value);
        size_t pos;
        Note *note_found = table_binary_search(table, tmp_key, &pos);
        if (note_found != NULL) {
            key_destroy(&tmp_key);
            data_destroy(&tmp_data);
            printf("Элемент с таким ключом уже существует.\n");
            return;
        }
        table->size++;
        table->note_arr = (Note *) realloc(table->note_arr, table->size * sizeof(Note));
        table->note_arr[table->size - 1].key = tmp_key;
        table->note_arr[table->size - 1].data = tmp_data;
        table->sorted = false;
    }
}

void table_note_delete(Table table, Key key)
{
    size_t pos;
    Note *note_found = table_binary_search(table, key, &pos);
    if(note_found == NULL) {
        return;
    }
    key_destroy(&(table->note_arr[pos].key));
    data_destroy(&(table->note_arr[pos].data));
    for (uint32_t i = pos; i < table->size - 1; ++i) {
        table->note_arr[i].key = table->note_arr[i + 1].key;
        table->note_arr[i].data = table->note_arr[i + 1].data;
    }
    table->size--;
    table->note_arr = (Note *) realloc(table->note_arr, table->size * sizeof(Note));
}

void table_qsort(Table table)
{
    size_t i, j;
    Key pivot;
    Stack *lows  = stack_create(stack_size);
    Stack *highs = stack_create(stack_size);
    int32_t low, high;
```

```
    stack_push(lows,  0);
    stack_push(highs, table->size - 1);


    while (!stack_is_empty(lows)) {
        low  = stack_pop(lows);
        high = stack_pop(highs);
        i = low;
        j = high;
        pivot = table->note_arr[(low + high)/2].key;
        do {
            while (key_cmp(table->note_arr[i].key, pivot) == -1) {
                ++i;
            }
            while (key_cmp(table->note_arr[j].key, pivot) == 1) {
                --j;
            }
            if (i <= j) {
                if (key_cmp(table->note_arr[i].key, table->note_arr[j].key) == 1) {
                    SWAP(table->note_arr[i], table->note_arr[j]);
                }
                ++i;
                if (j > 0) {
                    --j;
                }
            }
        } while (i <= j);

        if (i < high) {
            stack_push(lows, i);
            stack_push(highs, high);
        }
        if (j > low) {
            stack_push(lows, low);
            stack_push(highs, j);
        }
    }


    stack_destroy(&lows);
    stack_destroy(&highs);
    table->sorted = true;
}

Note *table_binary_search(Table table, Key key, size_t *pos)
{
    if (table->size == 0)
        return NULL;

    if (!table_is_sorted(table)) {
        table_qsort(table);
    }
    size_t start = 0;
    size_t end = table->size;
    size_t mid = (start + end) / 2;

    int lg;
    size_t mid_old = -1;

    while ((lg = key_cmp(key, table->note_arr[mid].key)) != 0) {
        if (lg > 0)
            start = mid;
        else
            end = mid;
        mid_old = mid;
        mid = (start + end) / 2;
        if (mid == mid_old)
            return NULL;
    }
```

```
    *pos = mid;
    return table->note_arr + mid;
}

void table_print(Table table)
{
    if (table->size == 0) {
        printf("Таблица пуста.\n");
        return;
    }
    for (uint32_t i = 0; i < table->size; ++i) {
        note_print(table->note_arr + i);
    }
    printf("\n");
}

void note_print(Note *note)
{
    key_print(note->key);
    printf(" ");
    data_print(note->data);
    printf("\n");
}

bool table_is_sorted(Table table)
{
    return table->sorted;
}

void table_reverse(Table table)
{
    Note *tmp;
    for(uint32_t i = 0; i <= (table->size - 1) / 2; ++i) {
        *tmp = table->note_arr[i];
        table->note_arr[i] = table->note_arr[table->size - 1 - i];
        table->note_arr[table->size - 1 - i] = *tmp;
    }
    table->sorted = false;
}

void table_scramble(Table table)
{
    srand((uint32_t)time(0));
    for(uint32_t i = 0; i < table->size - 1; ++i) {
        uint32_t j = randomAB(0, table->size - 1);
        uint32_t k = randomAB(0, table->size - 1);
        SWAP(table->note_arr[j], table->note_arr[k]);
    }
}

uint32_t randomAB(int a, int b)
{
    return a + rand() % (b - a + 1);
}
```

data.h

```
#ifndef _DATA_H_
#define _DATA_H_


#include <stdbool.h>
#include <string.h>
#include <inttypes.h>

#define DATA_LEN 63


typedef struct _data {
```

```
    char value[DATA_LEN];
} *Data;

Data data_create(void);
void data_destroy(Data *data);
void data_insert(Data data);
void data_print(Data data);
uint32_t data_len(void);

#endif
```

data.c

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdbool.h>
#include <string.h>
#include <inttypes.h>

#include "data.h"

Data data_create(void)
{
    Data data = (Data) calloc(1, sizeof(*data));
    if(!data) {
        fprintf(stderr, "Error:no memory\n");
        exit(1);
    }
    return data;
}


void data_destroy(Data *data)
{
    free(*data);
    *data = NULL;
}

void data_insert(Data data)
{
    fscanf(stdin, "%s", data->value);
}

void data_print(Data data)
{
    printf("%s", data->value);
}

uint32_t data_len(void)
{
    return DATA_LEN;
}
```

key.h

```
#ifndef _KEY_H_
#define _KEY_H_

#include <stdbool.h>
#include <string.h>
#include <inttypes.h>

#define KEY_LEN 6

typedef struct _key {
    char value[KEY_LEN];
} *Key;
```

```
Key key_create(void);
void key_destroy(Key *key);
void key_insert(Key key);
void key_print(Key key);
int32_t key_cmp(Key key1, Key key2);
uint32_t key_len(void);

#endif
```

key.c

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdbool.h>
#include <string.h>
#include <inttypes.h>

#include "key.h"

uint32_t key_len(void)
{
   return KEY_LEN;
}

Key key_create(void)
{
   Key key = (Key) calloc(1, sizeof(*key));
   if(!key) {
      fprintf(stderr, "Error:no memory\n");
      exit(1);
   }
   return key;
}

void key_destroy(Key *key)
{
   free(*key);
   *key = NULL;
}

void key_insert(Key key)
{
   fscanf(stdin, "%s", key->value);
}

void key_print(Key key)
{
   printf("%s", key->value);
}

int32_t key_cmp(Key key1, Key key2)
{
   for (uint32_t i = 0; i < KEY_LEN; ++i) {
      if (key1->value[i] > key2->value[i])
         return 1;
      if (key1->value[i] < key2->value[i])
         return -1;
   }
   return 0;
}
```

stack.h

```
#ifndef STACK_H
#define STACK_H

#include <stdlib.h>
#include <stdbool.h>
```

```
#define MEMORY_LIMIT 1
#define STACK_UNDERFLOW 2
#define STACK_OVERFLOW 3

typedef int Item;
typedef struct _stack Stack;

Stack *stack_create(size_t stack_size);
void stack_push(Stack *s, Item value);
Item stack_pop(Stack *s);
bool stack_is_empty(Stack *s);
void stack_destroy(Stack **s);
Item stack_top(Stack *s);

#endif // __STACK_H__
```

stack.c

```
#include <stdlib.h>
#include <stdbool.h>
#include <stdio.h>

#include "stack.h"

struct _stack {
   int ptr;
   Item *data;
}; //Stack

Stack *stack_create(size_t stack_size)
{
   Stack *s = (Stack *) malloc(sizeof(Stack));
   if(!s) {
      fprintf(stderr, "%s\n", "Error: no memory");
      exit(MEMORY_LIMIT);
   }
   s->data = NULL;
   s->ptr  = 0;
   return s;
}

void stack_push(Stack *s, Item value)
{
   s->data = (Item *) realloc(s->data, s->ptr * sizeof(Item) + sizeof(Item));
   if(!s->data) {
      fprintf(stderr, "%s\n", "Error: no memory\n");
      exit(MEMORY_LIMIT);
   }
   s->data[s->ptr++] = value;
}

Item stack_pop(Stack *s)
{
   if(stack_is_empty(s)) {
      fprintf(stderr, "%s\n", "Error: stack underflow\n");
      exit(STACK_UNDERFLOW);
   } else {
      return s->data[--s->ptr];
   }
}

bool stack_is_empty(Stack *s)
{
   return (s->ptr <= 0);
}

void stack_destroy(Stack **s)
{
   free((*s)->data);
```

```
    (*s)->data = NULL;
    (*s)->ptr = 0;
    free(*s);
    *s == NULL;
}

Item stack_top(Stack *s)
{
    if(s->ptr <= 0 || s->data == NULL) {
        return 0;
    } else {
        return s->data[(s->ptr) - 1];
    }
}
```