МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное бюджетное образовательное учреждение

высшего профессионально образования

“Московский авиационный институт

(национальный исследовательский университет)”

**ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И ПРИКЛАДНОЙ МАТЕМАТИКИ**

**Кафедра вычислительной математики и программирования**

Курсовой проект по курсу:

“Основы информатики”

II семестр

Задание 9

“Сортировка и поиск”

Работу выполнил:

студент 1 курса

Группы М8О-107Б-2018

Гамов Павел Антонович

Проверила:

асп. каф. 806 Ридли Александра Николаевна.

Научный руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Москва, 2019

Содержание

Введение…………………..……………..……………..……………..…………..3

Поставка задачи……………..……………..……………..……………..………..3

Алгоритмы……………..……………..……………..……………..……………..3

Разбор кода……………..……………..……………..……………..……………..4

1. makefile……………..……………..……………..……………..………..…4
2. table.h……………..……………..……………..……………..………….…4
3. table.c……………..……………..……………..……………..…………….5
4. main.c……………..……………..……………..……………..……………..8

Заключение……………..……………..……………..……………..……………..9

Список использованной литературы……………..……………..……………...10

**Введение**

Познакомившись с структурами хранения данных таких как списки или массивы возникает проблема быстрого доступа к нужному элементу. Если мы имеем массив, в котором мы знаем индекс нужного нам значения в нем, мы просто можем обратиться к нему по адресу и получить нужное нам значения, но чаще всего мы не знаем расположения нужного нам элемента, даже не знаем есть ли он там.

Обратимся к проблемам поиска и сопутствующим ему сортировкам.

**Постановка задачи**

Составить программу на языке Си с использованием процедур и функций для сортировки таблицы заданным методом, а также реализовать бинарный/двоичный поиск по таблице по указанному ключу.

**Алгоритмы и их разбор.**

Мой вариант сортировки требовал реализовать сортировку методом вставки с двоичным поиском. Идея этого алгоритма идет от простой сортировки методом вставки, когда массив сортируется от края до края, двигая выбранный элемент вглубь уже отсортированного массива, пока тот не встанет на свое место. Дело в том, что после захвата элемента, он линейно протаскивается по массиву, что при огромных размерах структуры будет крайне долго. Поэтому применяется принцип двоичного поиска. Так как часть массива уже отсортирована, на ней можно пустить алгоритм поиска, тем самым место куда нужно поставить элемент найдется не от О(n), а от O(log n). Все равно данная модификация не особо ускоряет процесс, ведь структура построена на массиве, и что бы освободить ячейку посреди него, правые или левые от него надо передвинуть на 1 ячейку, тем самым дав элементу встать.

Wolframalpha показывает теоретическую сложность алгоритма в O(log\_2(n!)), что быстрее чем О(n^2), но медленнее чем О(n log(n)), хотя ближе ко второму. Сравнивая с алгоритмом обычных вставок, сложность не покажет реального увеличения скорости, хотя на самом деле разница достаточно большая.

Отдельно касательно двоичного поиска, который используется в сортировке, да и просто при поиске по таблице. Его принцип строится на том, что ключ сравнивается с центральным ключом изначальной структуры, если первый больше, рекурсивно запускается дальнейший поиск по правой части массива, иначе по левой. Таким образом деля массив постоянно пополам ½ ¼ и так далее получаем логарифмическую сложность О(log(n)).

**Разбор кода**

**makefile**

MacBook-Pro-Pavel:9\_kur pavelgamov$ cat makefile

CC = gcc

CCFLAGS = -Wall -pedantic -std=c99

###\_\_\_\_###

a.out: main.o table.o ; $(CC) $(CCFLAGS) -g main.o table.o

main.o: main.c table.h ; $(CC) $(CCFLAGS) -c -g main.c

table.o: table.c table.h ; $(CC) $(CCFLAGS) -c -g table.c

clean: ; rm \*.o

###\_\_\_###

**table.h**

MacBook-Pro-Pavel:9\_kur pavelgamov$ cat table.h

#ifndef TABLE\_H

#define TABLE\_H

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#include <time.h>

typedef struct Node {

float key;

char \* value;

} Node;

typedef struct Table {

Node \* data;

int size, sorted;

} Table;

Структура таблица состоит из массива nod’ов, а также размера таблицы – целого значения и флага sorted. В свою очередь сам node состоит из числа типа float, а также строки.

Table \* createTable();

void addToTable(Table \* table);

void printTable(Table \* table);

void deleteTable(Table \*\* table);

int binarySearchByKey(Table \* table, float value);

void insertionSort(Table \* table);

void deleteFromTable(Table \* table, float key);

void randomizer(Table \* table);

#endif

**table.c**

MacBook-Pro-Pavel:9\_kur pavelgamov$ cat table.c

#include "table.h"

float epsilon() {

float e = 1;

while (e + 1 != 1) {

e/=2;

}

return e;

}

Нужна для сравнения ключей, так как обычная операция равно не работает правильно для чисел с плавающей запятой.

Table \* createTable()

{

int size = 0;

Table \* table = (Table \*)malloc(sizeof(Table));

Node \* array = (Node \*)malloc(sizeof(Node) \* size);

table->data = array;

table->size = size;

table->sorted = 0;

return table;

}

Возвращает указатель на структуру типа table размера 0.

void addToTable(Table \* table)

{

Node \* array = table->data;

array = (Node \*)realloc(array, sizeof(Node) \* (table->size + 1));

table->data = array;

char \* arr = (char \*)malloc(sizeof(char) \* 120);

table->data[table->size].value = arr;

scanf(" %g %s\n", &(table->data[table->size].key), table->data[table->size].value);

table->size++;

table->sorted = 0;

}

Считывает ключ и значение, выделяет память, расширяет массив, изменяет размер на 1 больше.

void printTable(Table \* table)

{

for (int i = 0; i < table->size; i++)

printf("key:\t%g\t\tvalue:\t%s\n", table->data[i].key, table->data[i].value);

}

void deleteTable(Table \*\* table)

{

if (\*table != NULL) {

if ((\*table)->size != 0) {

for (int i = 0; i < (\*table)->size; i++)

free((\*table)->data[i].value);

free((\*table)->data);

}

free(\*table);

\*table = NULL;

}

}

Удаление таблицы и ее чистка в памяти программы.

int binarySearchByKey(Table \* table, float value)

{

if (!(table->sorted))

insertionSort(table);

float e = epsilon();

int low = 0, top = table->size - 1, mid;

while (low <= top) {

mid = (low + top) / 2;

if (fabs(table->data[mid].key - value) < e)

return mid;

else if (value > table->data[mid].key)

low = mid + 1;

else

top = mid - 1;

}

return -1;

}

Принимает таблицу и ключ, который нужно найти, выполняет половинное деление пока не наткнется на нужный элемент, возвращает его индекс в массиве, если не нашел, возвратит -1, при существовании двух или больше одинаковых ключей, вернет индекс первого попавшегося.

int binarySearch(Table \* table, float item, int low, int high, float e)

{

if (high <= low) {

if (item > table->data[low].key)

return low + 1;

else

return low;

}

int mid = (low + high) / 2;

if (fabs(table->data[mid].key - item) < e)

return mid + 1;

if (item > table->data[mid].key)

return binarySearch(table, item, mid + 1, high, e);

return binarySearch(table, item, low, mid - 1, e);

}

Похожая функция только используется для поиска по массиву учитывая границы, рекурсивный поиск элемента, так как теперь мы ищем не ключ, а место, куда можно его пристроить.

void insertionSort(Table \* table)

{

float e = epsilon();

char \* arr;

int i, location, j;

float selected;

for (i = 1; i < table->size; ++i) {

j = i - 1;

selected = table->data[i].key;

arr = table->data[i].value;

location = binarySearch(table, selected, 0, j, e);

while (j >= location) {

table->data[j+1] = table->data[j];

j--;

}

table->data[j+1].key = selected;

table->data[j+1].value = arr;

}

table->sorted = 1;

}

Заводит эпсилон для сравнения чисел, в цикле берет ключ объекта, вызывает двоичный поиск для нахождения места, потом в цикле сдвигает все что левее него вправо, производит вставку элемента, так, пока весь массив не будет отсортирован по ключам. После сортировки ставит флаг того, что тот сортирован.

void deleteFromTable(Table \* table, float key)

{

int pos = binarySearchByKey(table, key);

if (pos != -1) {

free(table->data[pos].value);

for (int i = pos; i < table->size - 1; i++)

table->data[i] = table->data[i+1];

Node \* arr = table->data;

arr = (Node \*)realloc(arr, sizeof(Node) \* (table->size - 1));

table->data = arr;

table->size--;

}

}

Удаляет элемент по ключу из таблицы, включает двоичный поиск по ключу, находит индекс если такой есть, удаляет нод, сдвигает элементы на его место, уменьшает размер таблицы.

void randomizer(Table \* table)

{

srand(time(NULL));

int iteration = (table->size + (rand() % table->size)), l, r;

float key;

char \* value;

for (int i = 0; i < iteration; i++) {

l = rand() % table->size;

r = rand() % table->size;

if (l != r) {

key = table->data[l].key;

value = table->data[l].value;

table->data[l] = table->data[r];

table->data[r].key = key;

table->data[r].value = value;

}

}

}

Функция принимает на вход таблицу и случайно еe перемешивает, создавая запутанный массив, требуется для демонстрации сортировки.

**main.c**

MacBook-Pro-Pavel:9\_kur pavelgamov$ cat main.c

#include "table.h"

char q = ' ';

int pass = 1, res;

float req;

int main(int argc, char const \*argv[])

{

Table \* table = createTable();

while (pass) {

q = getchar();

switch (q) {

case '+': addToTable(table); break;

case '-': scanf(" %g", &req); deleteFromTable(table, req); break;

case 'f': scanf(" %g", &req);

res = binarySearchByKey(table, req);

if (res != -1)

printf("in table number : %d value : %s\n", res, table->data[res].value);

else

puts("not in table");

break;

case 'p': printTable(table); break;

case 's': insertionSort(table); break;

case 'r': randomizer(table); break;

case 'q': deleteTable(&table); pass = 0; break;

}

}

return 0;

}

**Заключение**

Сортировки являются важной частью программ на структуры и данные. В зависимости от вида структур используются различные сортировки различной сложности и эффективности. Придумано уже множество различных алгоритмов, каждый из которых может быть эффективным в том или ином случае. В целом, человечеству еще придется подумать над новыми типами и методами сортировок, ведь раньше алгоритмы упирались в количество памяти в машинах, теперь же, имея дешевую память, можно задействовать огромное количество дополнительной памяти для достижения наилучшего среднего результата для любой ситуации. Нынешние возможности много поточности позволяют уже сейчас разбивать сортировки на потоки, что дает ощутимый прирост в скорости выполнения тех или иных алгоритмов.

**Список использованной литературы**

Сортировка и поиск - http://ermak.cs.nstu.ru/cprog/html/046.htm

Сортировка и сложность двоичной вставки - http://qaru.site/questions/6413194/binary-insertion-sort-and-complexity

Структуры на Си - https://spravochnick.ru/bazy\_dannyh/predstavlenie\_dannyh\_v\_pamyati\_kompyutera/tipy\_i\_struktury\_dannyh/

Динамические структуры данных на языке Си - https://younglinux.info/c/stack

Массив структур на Си - https://metanit.com/cpp/c/6.4.php