Балтийский государственный технический университет  
«ВОЕНМЕХ» им. Д. Ф. Устинова

Кафедра О7 «Информационные системы и программная инженерия»

**Практическая работа №3**по дисциплине «Структуры и организация данных»  
на тему «Оценка эффективности алгоритмов»  
часть 2 «Алгоритмы поиска»  
  
вариант 25

Выполнил:  
Студент Фокин В. К.  
Группа И924Б  
  
Преподаватель:  
Гладевич А. А.

Санкт-Петербург  
2023г.

Произвести сравнение указанных в вариативной части структур данных по пространственной сложности и вычислительной сложности поиска. Работа выполняется в четыре этапа.

На первом этапе требуется произвести теоретические расчеты сложности поиска и занимаемого структурой объема памяти (если в вариативной части не указано, что она должна вычисляться только в программе).

На втором этапе нужно сформировать тестовый набор из 100 ключей, включив в него как имеющиеся в файле, так и отсутствующие в нем ключи (меньшие 10000000, большие 100000000 и принадлежащие интервалу [10000000; 100000000) ), стараться обеспечить равномерный поиск по всей структуре.

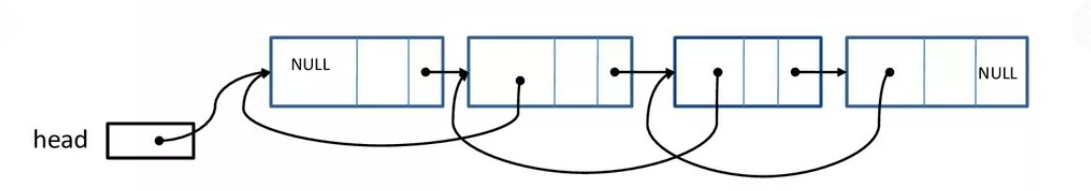
На третьем этапе нужно написать программу, в которой реализовать обе указанные структуры данных, заполнив их всеми значениями из приложенного файла test\_numbers.txt. Выполнить поиск 100 ключей в указанных структурах данных, для каждого ключа выводить сообщение о том, найден он или нет, и количество выполненных при поиске сравнений ключей, в конце программы вывести среднее количество сравнений, пришедшееся на один ключ. Подсчитать количество требуемой памяти для реализации каждой структуры и количество сравнений при поиске.

На заключительном четвертом этапе требуется произвести анализ полученных теоретических и практических результатов, сравнить поисковые структуры по вычислительной и пространственной сложности и выбрать наиболее эффективную из них.

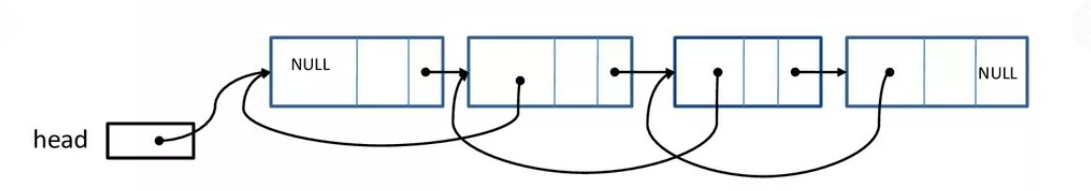
Вариант № 25

Упорядоченный двусвязный список, несбалансированное дерево бинарного поиска

Схематичное изображение упорядоченного двусвязного списка:



Схематичное изображение структуры хранения, использованной в программе для программирования упорядоченного двусвязного списка:

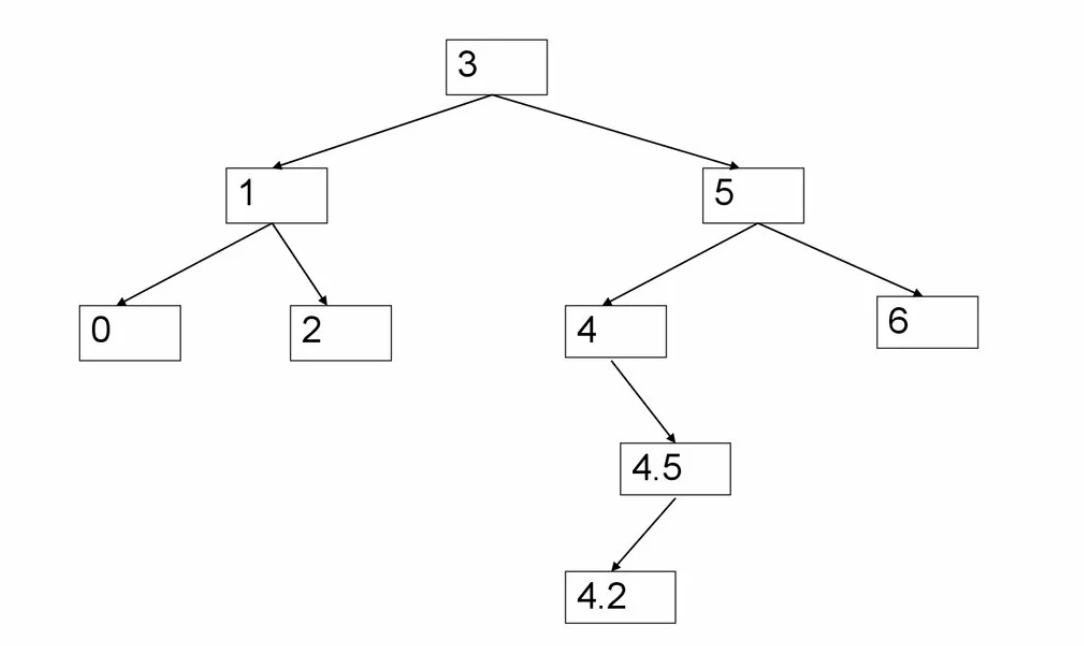


Трудоемкость поиска:

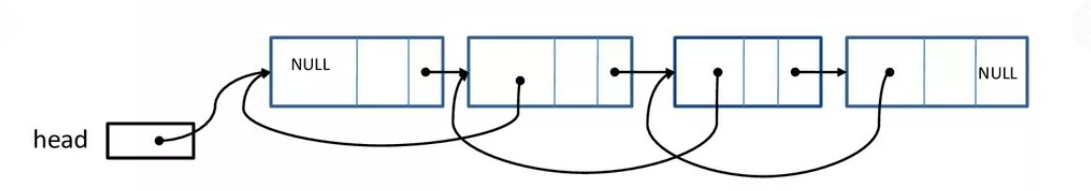
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Случай** | **Ситуация, соответствующая случаю** | **Обоснование** | **Ожидаемое число сравнений при поиске** | **Асимптотическая оценка сложности поиска** |
| наилучший | искомый элемент находится в самом начале списка | требуется выполнить одно сравнение |  |  |
| наихудший | искомый ключ отсутствует в списке | требуется осуществить перебор всех ключей | *200000* |  |
| наиболее вероятный | искомый ключ находится в середине списка | требуется осуществить перебор половины всех ключей | *100000* |  |

Требуемый объем памяти байт

Схематичное изображение несбалансированного дерева бинарного поиска:

**

Схематичное изображение структуры хранения, использованной в программе для программирования несбалансированного дерева бинарного поиска:



Трудоемкость поиска:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Случай** | **Ситуация, соответствующая случаю** | **Обоснование** | **Ожидаемое число сравнений при поиске** | **Асимптотическая оценка сложности поиска** |
| наилучший | искомый ключ находится в корне дерева | требуется выполнить одно сравнение |  |  |
| наихудший | дерево вырожденное, искомый ключ отсутствует в дереве | требуется осуществить перебор всех ключей, два сравнения на ключ умножить на количество узлов | *400000* |  |
| наиболее вероятный | дерево не вырожденное, искомый ключ присутствует в дереве | требуется осуществить перебор ключей на одной ветке дерева  два сравнения на ключ умножить на среднюю высоту дерева | *140* |  |

Требуемый объем памяти байт

Тестовый набор ключей:

присутствующие в файле:

97192338; 50910034; 20628925; 62588809; 88009629; 84215883; 27553155; 18185866; 68100269; 77963944; 52047144; 70705289; 26792643; 85415106; 99651495; 93469445; 89030389; 48355966; 80767881; 65654618; 55967837; 64339998; 99345171; 29929275; 57980340 11902190; 87551508; 48424307; 89502758; 70188037; 58505350; 23733327; 73515119; 60990510; 97721422; 86067699; 86659167; 79631823; 71737769; 45380349; 47662328; 85522949; 52043983; 94482354; 97038008; 73417676; 17399292; 33598899; 95110582; 91017354.

отсутствующие в файле:

Меньше присутствующих в файле:

4191; 11332; 1773; 728; 26264; 836; 10791; 183; 5517; 24298; 14055; 28585; 11369; 1412; 16075; 11547; 7150;

Больше присутствующих в файле:

100000363; 100025839; 100013721; 100030823; 100024140; 100031650; 100026812; 100031628; 100030836; 100022178; 100007828; 100015291; 100028677; 100012249; 100006774; 100023492;

В том же промежутке, что и присутствующие файлы:

10007120; 10030319; 10002573; 10024370; 10000125; 10002138; 10000947; 10005704; 10022622; 10019851; 10000229; 10007844; 10003252; 10024718; 10028431; 10014545;

Текст программы:

OrderedDoublyLinkedList.h:

#include <iostream>

using namespace std;

struct node {

int data;

node\* prev;

node\* next;

};

class ODLList {

private:

node\* begin;

node\* end;

public:

ODLList() { //конструктор

begin = nullptr;

end = nullptr;

}

ODLList(const ODLList& other) { //конструктор копирования

begin = nullptr;

end = nullptr;

//если список, который копируется, не пустой

if (other.begin != nullptr) {

//копируем первый элемент

node\* current = other.begin;

node\* newNode = new node;

newNode->data = current->data;

newNode->prev = nullptr;

newNode->next = nullptr;

begin = newNode;

end = newNode;

//копируем остальные элементы

current = current->next;

while (current != nullptr) {

node\* newNode = new node;

newNode->data = current->data;

newNode->prev = end;

newNode->next = nullptr;

end->next = newNode;

end = newNode;

current = current->next;

}

}

}

ODLList& operator=(const ODLList& other) { //перегруженный оператор присваивания

//проверяем на самоприсваивание

if (this == &other) {

return \*this;

}

//очищаем текущий список

node\* current = begin;

while (current != nullptr) {

node\* next = current->next;

delete current;

current = next;

}

begin = nullptr;

end = nullptr;

//если список, который копируется, не пустой

if (other.begin != nullptr) {

//копируем первый элемент

current = other.begin;

node\* newNode = new node;

newNode->data = current->data;

newNode->prev = nullptr;

newNode->next = nullptr;

begin = newNode;

end = newNode;

//копируем остальные элементы

current = current->next;

while (current != nullptr) {

newNode = new node;

newNode->data = current->data;

newNode->prev = end;

newNode->next = nullptr;

end->next = newNode;

end = newNode;

current = current->next;

}

}

return \*this;

}

void insert\_sorted(int value) {

node\* newNode = new node;

newNode->data = value;

newNode->prev = nullptr;

newNode->next = nullptr;

if (begin == nullptr) {

//если список пустой, устанавливаем новый элемент как начало и конец списка

begin = newNode;

end = newNode;

}

else if (value <= begin->data) {

//если значение нового элемента меньше или равно значению первого элемента,

//вставляем новый элемент в начало списка

newNode->next = begin;

begin->prev = newNode;

begin = newNode;

}

else if (value >= end->data) {

//если значение нового элемента больше или равно значению последнего элемента,

//вставляем новый элемент в конец списка

newNode->prev = end;

end->next = newNode;

end = newNode;

}

else {

//ищем правильное место для вставки нового элемента

node\* current = begin;

while (current != nullptr && current->data < value) {

current = current->next;

}

//вставка нового элемента между двумя существующими элементами

newNode->next = current;

newNode->prev = current->prev;

current->prev->next = newNode;

current->prev = newNode;

}

}

void delete\_begin() {

if (begin == nullptr) {

//список пустой, ничего не делаем

return;

}

node\* toDelete = begin;

if (begin == end) {

//список содержит только один элемент

begin = nullptr;

end = nullptr;

}

else {

begin = begin->next;

begin->prev = nullptr;

}

delete toDelete;

}

void delete\_end() {

if (end == nullptr) {

//список пустой, ничего не делаем

return;

}

node\* toDelete = end;

if (begin == end) {

//список содержит только один элемент

begin = nullptr;

end = nullptr;

}

else {

end = end->prev;

end->next = nullptr;

}

delete toDelete;

}

void delete\_el(int key) {

node\* current = begin;

while (current != nullptr) {

if (current->data == key) {

//найден элемент с заданным ключом

if (current == begin) {

//удаляемый элемент является первым элементом списка

begin = current->next;

if (begin != nullptr) {

begin->prev = nullptr;

}

else {

end = nullptr;

}

}

else if (current == end) {

//удаляемый элемент является последним элементом списка

end = current->prev;

end->next = nullptr;

}

else {

//удаляемый элемент находится между двумя элементами списка

current->prev->next = current->next;

current->next->prev = current->prev;

}

delete current;

break;

}

current = current->next;

}

}

node\* getBegin() const {

return begin;

}

void print() const {

node\* current = begin;

while (current != nullptr) {

cout << current->data << " ";

current = current->next;

}

cout << endl;

}

void print\_back() const {

node\* current = end;

while (current != nullptr) {

cout << current->data << " ";

current = current->prev;

}

cout << endl;

}

};

BinarySearchTree.h:

#include <iostream>

unsigned long cntCmp2;

struct Node {

int data;

Node\* left;

Node\* right;

Node(int value) {

data = value;

left = nullptr;

right = nullptr;

}

};

class BSTree {

private:

Node\* root;

Node\* insertNode(Node\* root, int value) {

if (root == nullptr) {

return new Node(value);

}

if (value < root->data) {

root->left = insertNode(root->left, value);

}

else {

root->right = insertNode(root->right, value);

}

return root;

}

Node\* findMinNode(Node\* node) {

while (node->left != nullptr) {

node = node->left;

}

return node;

}

Node\* deleteNode(Node\* root, int value) {

if (root == nullptr) {

return root;

}

if (value < root->data) {

root->left = deleteNode(root->left, value);

}

else if (value > root->data) {

root->right = deleteNode(root->right, value);

}

else {

if (root->left == nullptr) {

Node\* temp = root->right;

delete root;

return temp;

}

else if (root->right == nullptr) {

Node\* temp = root->left;

delete root;

return temp;

}

Node\* minNode = findMinNode(root->right);

root->data = minNode->data;

root->right = deleteNode(root->right, minNode->data);

}

return root;

}

void clear(Node\* node) {

if (node != nullptr) {

clear(node->left);

clear(node->right);

delete node;

}

}

public:

BSTree() {

root = nullptr;

}

BSTree(const BSTree& other) {

root = copyNodes(other.root);

}

BSTree& operator=(const BSTree& other) {

if (this != &other) {

// Очистить существующее дерево

clear(root);

// Создать копию дерева из other

root = copyNodes(other.root);

}

return \*this;

}

Node\* copyNodes(Node\* node) {

if (node == nullptr) {

return nullptr;

}

Node\* newNode = new Node(node->data);

newNode->left = copyNodes(node->left);

newNode->right = copyNodes(node->right);

return newNode;

}

void insert(int value) {

root = insertNode(root, value);

}

void del(int value) {

root = deleteNode(root, value);

}

bool search(int value) {

Node\* current = root;

cntCmp2 = 0;

while (current != nullptr) {

if (value == current->data) {

cntCmp2++;

return true;

}

else if (value < current->data) {

cntCmp2++;

current = current->left;

}

else {

cntCmp2++;

current = current->right;

}

}

return false;

}

};

programm3\_2.cpp:

#include <iostream>

#include <fstream>

#include <ctime>

#include <algorithm>

#include <Windows.h>

#include <iomanip>

#include "OrderedDoublyLinkedList.h"

#include "BinarySearchtree.h"

using namespace std;

const int N = 200000, H = 100;

bool isNumberUnique(int num, unsigned long arr[]) { //проверка на присутствие ключа в файле

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

if (arr[i] == num) {

return false;

}

}

return true;

}

int random(long min, long max) {

return min + rand() % (max - min + 1);

}

int main() {

SetConsoleOutputCP(1251);

srand(time(nullptr));

unsigned long arr[N];

int keys[H], totalCmp = 0;

char menu;

ODLList list;

BSTree tree;

ifstream file("C:\\Data Structures\\pr3\_2\\programm3\_2\\test.txt");

if (!file) {

cout << "Ошибка при открытии файла" << endl;

return 1;

}

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

file >> arr[i];

}

file.close();

for (int i = 0; i < 17; i++) { //числа, меньше имеющихся в файле

keys[i] = random(0, 10000000);

}

for (int i = 0, j = 17; i < N && j < 67; i += 3700, j++) { //числа, имеющиеся в файле

keys[j] = arr[i];

}

int a = 67;

while (a < 84) {

int rand = random(10000000, 100000000); //числа в промежутке с имеющимися в файле

if (isNumberUnique(rand, arr)) {

keys[a] = rand;

a++;

}

}

for (int i = 84; i < 100; i++) { //числа, больше имеющихся в файле

keys[i] = random(100000000, 1000000000);

}

do {

cout << "-----------------------------------------------" << endl;

cout << "1. Упорядоченный двусвязный линейный список |" << endl;

cout << "2. Несбалансированное дерево бинарного поиска |" << endl;

cout << "3. Посмотреть все ключи |" << endl;

cout << "0. Выход |" << endl;

cout << "-----------------------------------------------" << endl;

cin >> menu;

switch (menu) {

case '1':

system("cls");

totalCmp = 0;

for (int i = 0; i < N; i++) { //загрузка в список

list.insert\_sorted(arr[i]);

cout << i << endl;

}

cout << "----------------------------------------------------" << endl;

cout << "| Ключ | Имеется в файле? | Кол-во сравнений |" << endl;

cout << "----------------------------------------------------" << endl;

for (int i = 0; i < H; i++) { //поиск всех ключей

int cntCmp = 0;

int key = keys[i];

bool isFound = false;

node\* current = list.getBegin();

while (current != nullptr) {

if (current->data == key) {

isFound = true;

break;

}

current = current->next;

cntCmp++;

}

cout << "| " << left << setw(9) << key << " | ";

cout << left << setw(10) << (isFound ? "Да" : "Нет") << " | ";

cout << left << setw(12) << cntCmp << "|" << endl;

totalCmp += cntCmp;

}

cout << "----------------------------------------------------" << endl;

cout << "Всего совершено сравнений: " << totalCmp << endl;

cout << "Среднее количество сравнений: " << totalCmp / H << endl;

cout << "Количество затраченной памяти: " << N \* sizeof(node) << endl;

break;

case '2':

system("cls");

totalCmp = 0;

for (int i = 0; i < N; i++) { //загрузка в дерево

tree.insert(arr[i]);

}

cout << "----------------------------------------------------" << endl;

cout << "| Ключ | Имеется в файле? | Кол-во сравнений |" << endl;

cout << "----------------------------------------------------" << endl;

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

int key = keys[i];

bool isFound = tree.search(key); //поиск всех ключей

cout << "| " << left << setw(9) << key << " | ";

cout << left << setw(10) << (isFound ? "Да" : "Нет") << " | ";

cout << left << setw(12) << cntCmp2 << "|" << endl;

totalCmp += cntCmp2;

}

cout << "----------------------------------------------------" << endl;

cout << "Всего совершено сравнений: " << totalCmp << endl;

cout << "Среднее количество сравнений: " << totalCmp / H << endl;

cout << "Количество затраченной памяти: " << N \* sizeof(Node) << endl;

break;

case '3':

system("cls");

cout << "-------------------------" << endl;

cout << "| # | Список ключей |" << endl;

cout << "-------------------------" << endl;

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

int key = keys[i];

cout << "| " << left << setw(4) << i + 1 << "| " << left << setw(12) << key << " |" << endl;

}

cout << "-------------------------" << endl;

break;

case '0':

break;

default:

cout << "Выбран несуществующий пункт меню" << endl;

}

} while (menu != '0');

}

Результаты работы программы:

При запуске программы появляется меню.

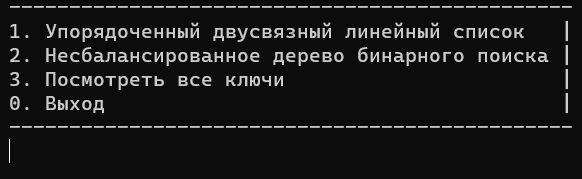


Рисунок 1 – Запуск программы

При нажатии на «1» программа добавит в упорядоченный двусвязный список элементы из файла, произведет поиск 100 ключей и выведет таблицу. В таблице выведен сам ключ, информация, находится ли он в файле, а также количество сравнений при его поиске, что показано на рисунках 2 и 3.

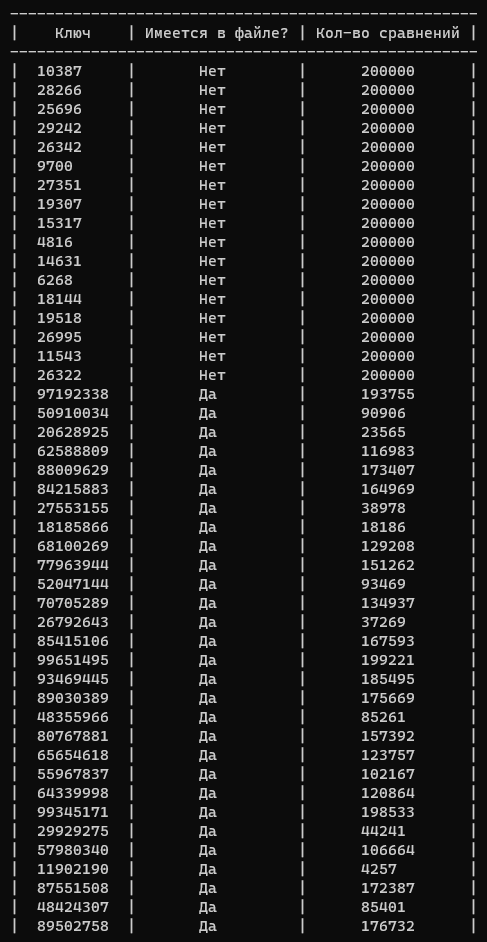


Рисунок 2 – Таблица с информацией

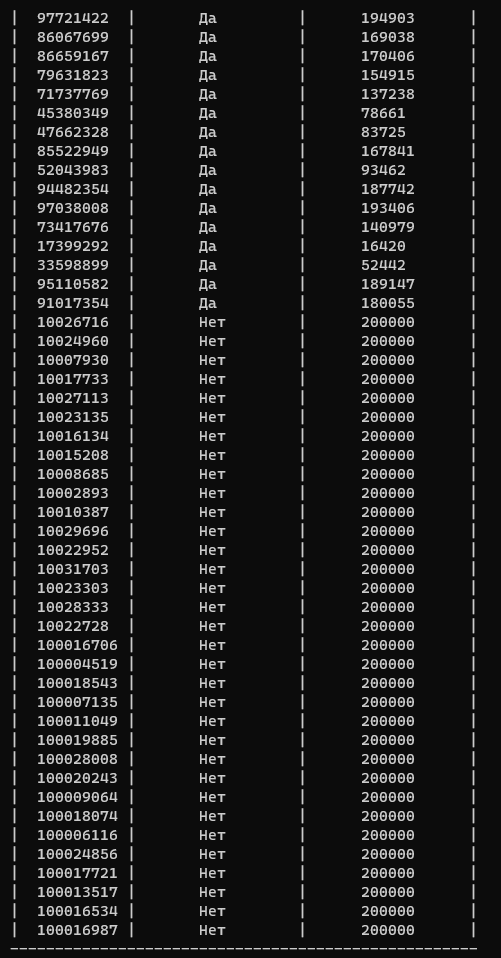


Рисунок 3 – Таблица с информацией

Так же будет выведено общее количество сравнений, среднее количество сравнений и количество памяти, что показано на рисунке 4.

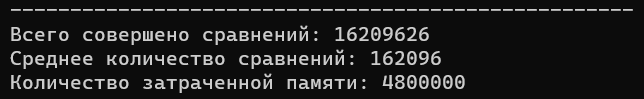


Рисунок 4 – Информация после таблицы

При нажатии на «2» программа добавит в несбалансированное дерево бинарного поиска элементы из файла, произведет поиск 100 ключей и выведет таблицу. В таблице выведен сам ключ, информация, находится ли он в файле, а также количество сравнений при его поиске, что показано на рисунках 5 и 6.

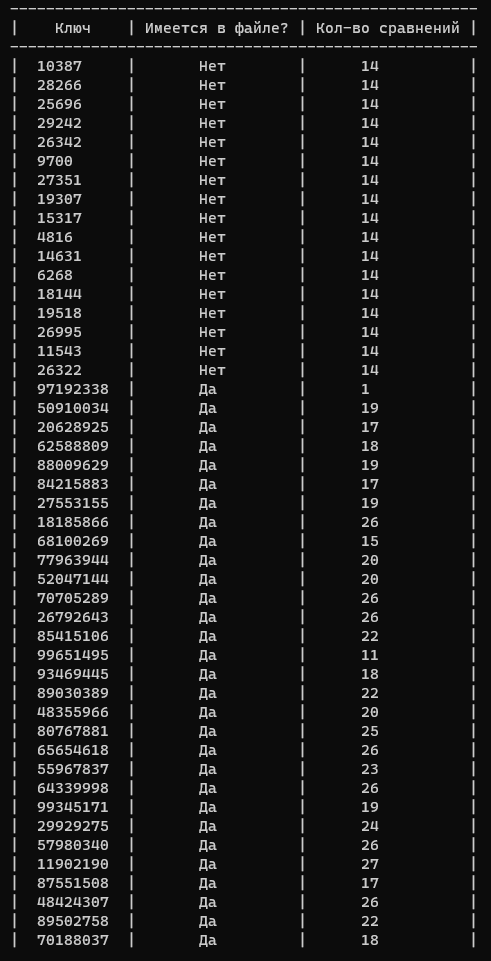


Рисунок 5 – Таблица с информацией

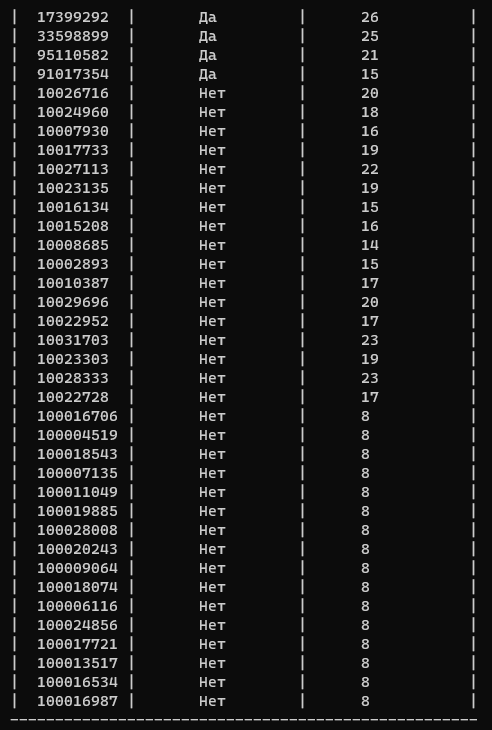


Рисунок 6 – Таблица с информацией

При нажатии на «3» будут выведены все ключи, что показано на рисунках 7 и 8.

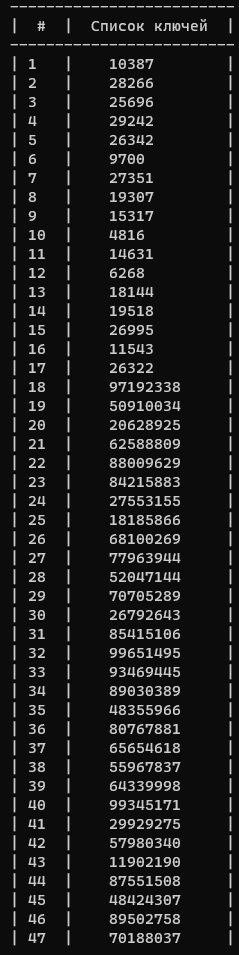


Рисунок 7 – Вывод всех ключей

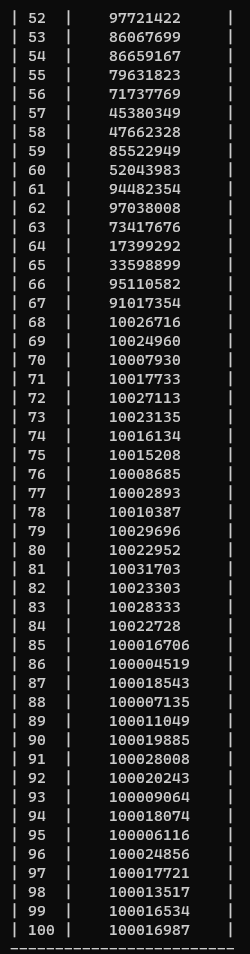


Рисунок 8 – Вывод всех ключей

При выборе несуществующего пункта меню появится сообщение об ошибке, что показано на рисунке 9.



Рисунок 9 – Выбор несуществующего пункта меню

При нажатии на «0» работа программы прекратиться.

Выводы:

Данные, полученные при поиске в упорядоченном двусвязном списке, полностью соответствуют расчетным.

Данные, полученные при поиске в несбалансированном дереве бинарного поиска, отличаются от расчетных с небольшой погрешностью из-за неучтенного коэффициента.

По трудоемкости:

В худшем случае оптимальным будет двусвязный список, так как произведет меньше сравнений.

В среднем случае оптимальным будет несбалансированное дерево бинарного поиска, так как произведет меньше сравнений.

В лучшем случае алгоритмы поиска работают одинаково.

Пространственная сложность обоих алгоритмов одинакова, так как они реализованы с помощью одной и той же структуры хранения.