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 9

Выполнил:  
Студент Квасюк В. В.  
Группа 0712Б  
  
Преподаватель:  
Палехова О. А.

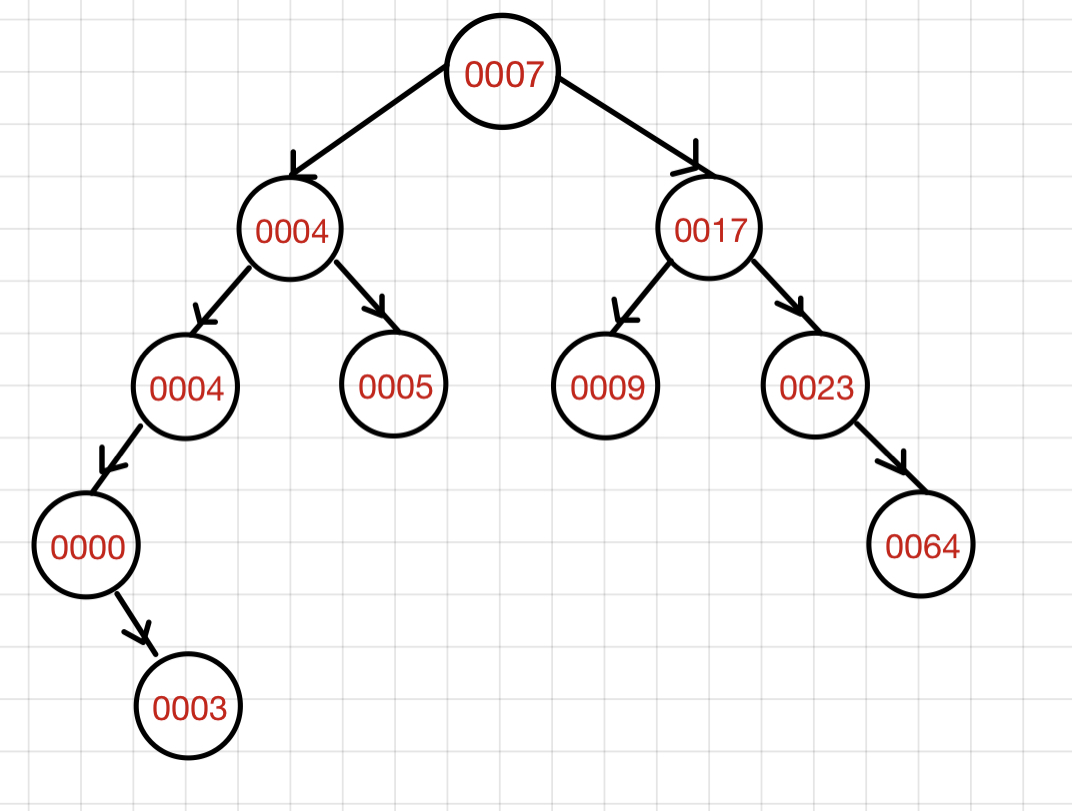
Санкт-Петербург  
2022 г.

Произвести сравнение указанных в вариативной части структур данных по пространственной сложности и вычислительной сложности поиска.

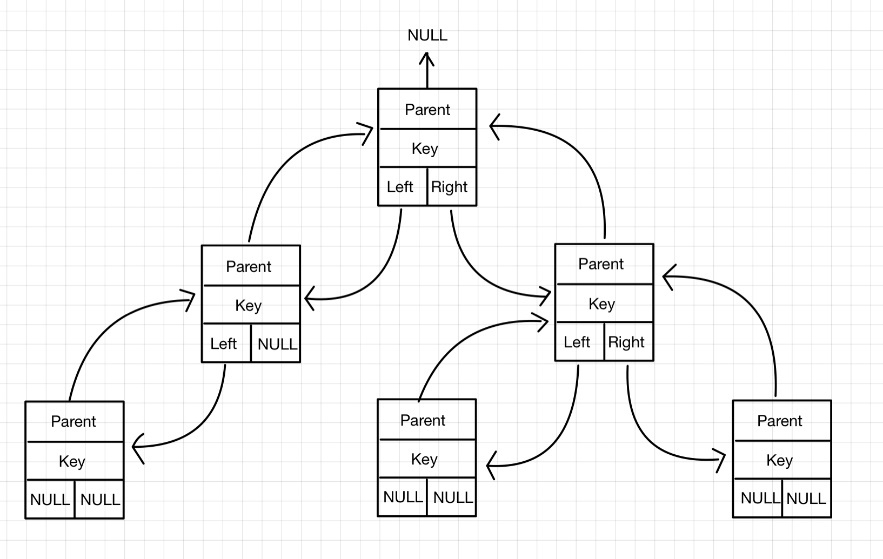
Выполнить теоретические расчеты сложности поиска и занимаемого структурой объема памяти. Сформировать корректный тестовый набор из 100 ключей. Реализовать две указанные структуры данных, заполнив их значениями из приложенного файла *test\_numbers.txt*. Выполнить поиск 100 ключей в указанных структурах 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. Произвести анализ полученных теоретических и практических результатов, сравнить поисковые структуры по вычислительной и пространственной сложности и выбрать наиболее эффективную из них.

Вариант № 9 Splay-дерево, хеш-таблица с разрешением коллизий методом цепочек (хешфункция – метод деления).

Схематичное изображение splay-дерева:



Схематичное изображение структуры хранения, использованной в программе для программирования splay-дерева:

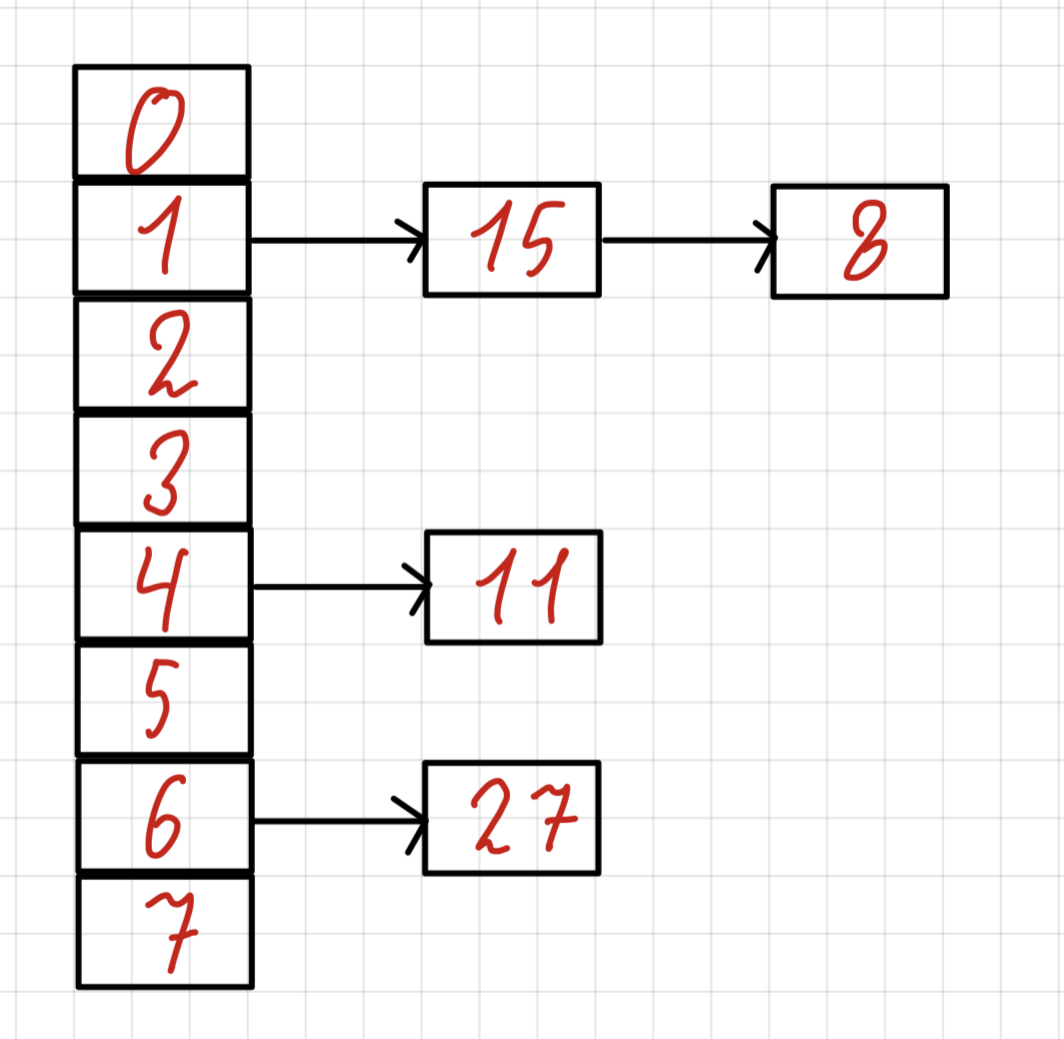


Трудоемкость поиска:

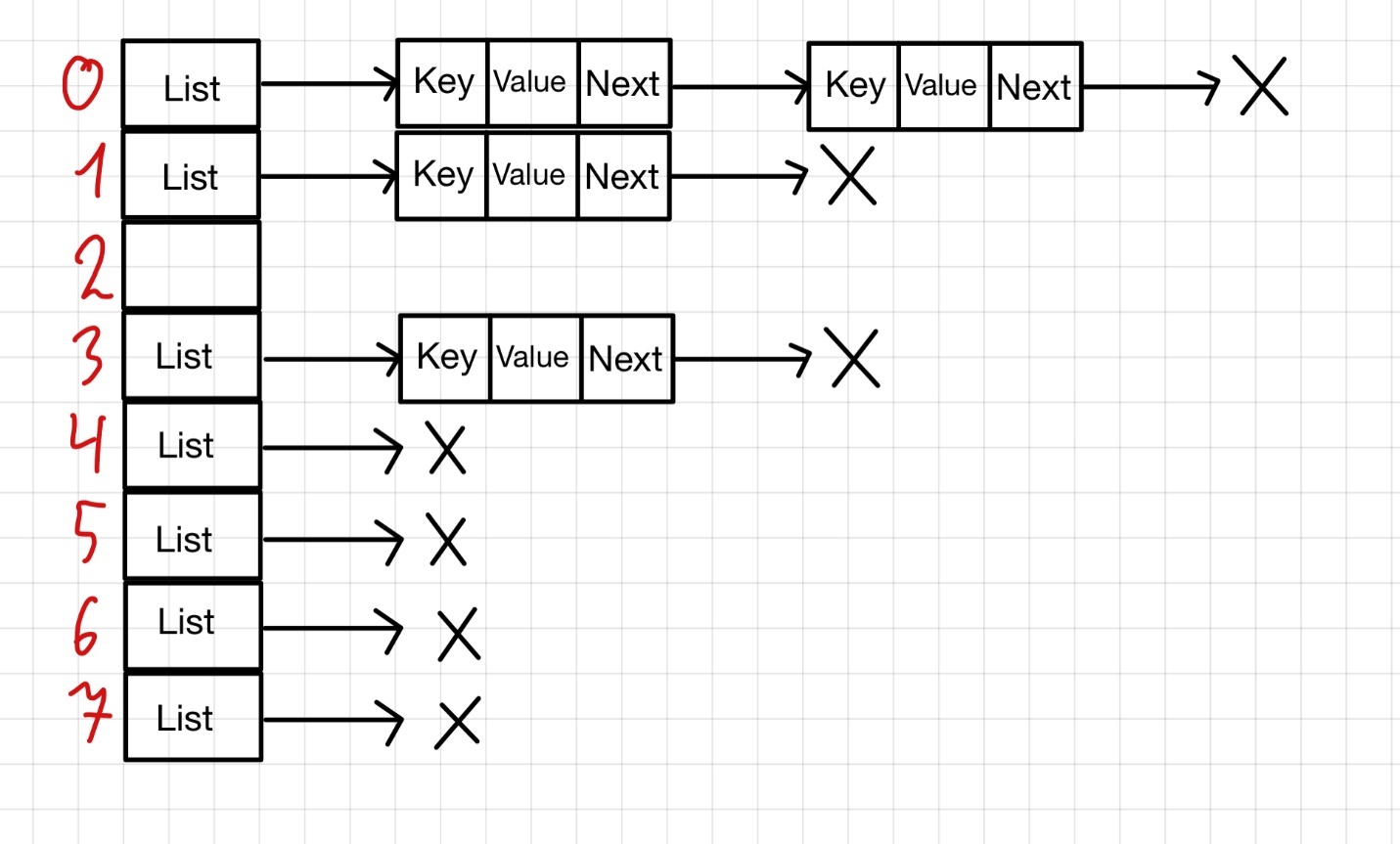
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Случай** | **Ситуация, соответствующая случаю** | **Обоснование** | **Ожидаемое число сравнений при поиске** | **Асимптотическая оценка сложности поиска** |
| наилучший | искомый ключ находится в корне дерева | требуется выполнить одно сравнение | 1 | Ω(1) |
| наихудший | искомый ключ отсутствует в дереве | требуется осуществить перебор всех ключей, три сравнения на ключ умножить на кол-во элементов | 3N | O(N) |
| наиболее вероятный | дерево не сбалансированное, искомый ключ присутствует в дереве | Поиск ключа уже осуществлялся, ключ будет найден не ниже чем средняя высота дерева | (3log2N)/2 | O(log2N) |

Требуемый объем памяти (sizeof(int)+3\*sizeof(pointer)+ sizeof(pointer))\*N байт.

Схематичное изображение хеш-таблица с разрешением коллизий методом цепочек (хеш- функция – метод деления):



Схематичное изображение структуры хранения, использованной в программе для программирования хеш-таблицы с разрешением коллизий методом цепочек (хеш- функция – метод деления):



Трудоемкость поиска:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Случай** | **Ситуация, соответствующая случаю** | **Обоснование** | **Ожидаемое число сравнений при поиске** | **Асимптотическая оценка сложности поиска** |
| наилучший | Искомый ключ первый в списке | требуется получить значение хеш-функции и сразу первым же делом получить элемент. В таком случае произойдет минимальное количество сравнений и только одна атомарная операция(когда будет вычисляться значение хеш-функции) | 1 | Ω(1) |
| наихудший | Искомый ключ последний в списке или его вообще нет. | После того как мы получим значение хеш-функции мы попадем в список, в котором уже есть много записей. Тогда нам придется проверить каждый и только потом убедиться, что искомого ключа нет | N | O(N) |
| наиболее вероятный | Искомый ключ присутствует в дереве, находится в середине списка | После получения значения хеш-функции нам придется проверить список на наличие искомого ключа, очень вероятно что он будет лежать в середине списка | N/2 | O(1) |

Требуемый объем памяти (sizeof(unsigned long)+sizeof(Node)) байт.

Тестовый набор ключей:

присутствующие в файле:

97192338, 90229668, 67343291, 70258238, 83246999, 37801902, 30349033, 19445301, 88278989, 36406281, 35966130, 35557396, 84580155, 83130037, 42315857, 15916697, 52779136, 32051522, 39771580, 20138140, 42947553, 95905677, 62667096, 66541010, 30617946, 36600568, 99020052, 55743512, 83673374, 42882723, 56804407, 66912047, 97986428, 56718265, 27691681, 87238974, 98425454, 34568631, 51935268, 85068842, 84258720, 64275429, 50492010, 24892362, 27648480, 55750064, 52695354, 46375986, 52779136, 28831849

Отсутствующие ключи в файле:

1. 2 числа, которые меньше 10 000 000:

3514288, 6146302

1. 46 чисел, которые входят в промежуток [10 000 000; 100 000 000)

40757147, 47978500, 26971124, 59567120, 11749387, 21470545, 16239572, 36984808, 13615544, 59042088, 18341522, 21564522, 55016433, 13987482, 30090226, 10617624, 74041604, 72908328, 37655783, 88159989, 17379060, 60530496, 40835680, 51597008, 16923810, 10342100, 96546364, 44012332, 62071040, 82755056, 69646416, 82695451, 93557676, 59942653, 47583462, 68791610, 19031778, 23930380, 76547830, 29375581, 18773940, 76607780, 18402970, 19622200, 68026284, 16697010

1. 2 числа, которые больше 100 000 000:

114295659, 106180358

Текст программы:

**// hashMap.h**

#include <cstddef>

template <typename K, typename V>

struct HashNode

{

// пара ключ значение

K \_key;

V \_value;

// следующий элемент(используется если произошла коллизия)

HashNode \*\_next;

HashNode(const K &key, const V &value) : \_key(key), \_value(value), \_next(NULL) {}

K getKey() const { return \_key; }

V getValue() const { return \_value; }

void setValue(V value) { \_value = value; }

HashNode \*getNext() const { return \_next; }

void setNext(HashNode \*next) { \_next = next; }

};

template <typename K, typename V, size\_t tableSize, typename F>

class HashMap

{

public:

HashMap() : table(),

hashFunc(), comparing(0), totalMemory(0)

{

totalMemory = tableSize \* sizeof(HashNode<K, V>) + 8 + sizeof(hashFunc);

}

~HashMap()

{

// удаляем все слоты один за другим

for (size\_t i = 0; i < tableSize; ++i)

{

HashNode<K, V> \*entry = table[i];

while (entry != NULL)

{

HashNode<K, V> \*prev = entry;

entry = entry->getNext();

delete prev;

}

table[i] = NULL;

}

}

// получение значения по ключу

bool get(const K &key, V &value)

{

unsigned long hashValue = hashFunc(key);

HashNode<K, V> \*entry = table[hashValue];

while (entry != NULL)

{

comparing++; // увеличиваем количество сравнений

if (entry->getKey() == key)

{

value = entry->getValue();

return true;

}

entry = entry->getNext();

}

return false;

}

// запись в хеш-таблицу

void put(const K &key, const V &value)

{

unsigned long hashValue = hashFunc(key);

HashNode<K, V> \*prev = NULL;

HashNode<K, V> \*entry = table[hashValue];

while (entry != NULL && entry->getKey() != key)

{

prev = entry;

entry = entry->getNext();

}

if (entry == NULL)

{

entry = new HashNode<K, V>(key, value);

if (prev == NULL)

{

// если слот ещё не занят то записываем как первый

table[hashValue] = entry;

}

else

{

// иначе записываем последовательно

prev->setNext(entry);

}

totalMemory += sizeof(HashNode<K, V>);

}

else

{

// если элемент уже был то просто обновляем его значение

entry->setValue(value);

}

}

// удаление элемента из хеш-таблицы

void remove(const K &key)

{

unsigned long hashValue = hashFunc(key);

HashNode<K, V> \*prev = NULL;

HashNode<K, V> \*entry = table[hashValue];

while (entry != NULL && entry->getKey() != key)

{

prev = entry;

entry = entry->getNext();

}

if (entry == NULL)

{

// если ключ не найден то просто выходим

return;

}

else

{

if (prev == NULL)

{

// удаляем элемент списка если перед ним ничего нет

table[hashValue] = entry->getNext();

}

else

{

// удаляем элемент если он в середине

prev->setNext(entry->getNext());

}

// из общей выделенной памяти вычитаем удаленный элемент

totalMemory -= sizeof(HashNode<K, V>);

delete entry;

}

}

// получаем количество сравнений и обнуляем счетчик

int getComparing()

{

int i = comparing;

comparing = 0;

return i;

}

// получаем количество выделенной памяти

int getTotalMemory()

{

return totalMemory;

}

private:

HashMap(const HashMap &other);

const HashMap &operator=(const HashMap &other);

HashNode<K, V> \*table[tableSize];

F hashFunc;

int comparing;

int totalMemory;

};

**// splayTree.h**

#include <stdio.h>

#include <stdlib.h>

#define SplayTree struct splayTree

#define Node struct node

int comparing = 0;

// BinTree routine

Node {

int key;

Node\* left, \* right, \* parent;

};

Node\* makeNode(Node\* parent, int key, int \*memory) {

\*memory += (sizeof(Node) + sizeof(node));

Node\* node = (Node\*)malloc(sizeof(Node));

node->left = node->right = NULL;

node->parent = parent;

node->key = key;

return node;

}

Node\* addNode(Node\* cur, int key, int \*memory) {

if(cur->key == key)

return cur;

Node\* tmp;

if(cur->key > key) {

if(cur->left)

tmp = addNode(cur->left, key, memory);

else

tmp = cur->left = makeNode(cur, key, memory);

}

else {

if(cur->right)

tmp = addNode(cur->right, key, memory);

else

tmp = cur->right = makeNode(cur, key, memory);

}

return tmp;

}

void del\_tree (Node \*node)

{

if (node->left) del\_tree (node->left);

if (node->right) del\_tree (node->right);

free (node);

}

void print\_sim (Node \*node, int tbl)

{

if (node->right) print\_sim (node->right, tbl+5);

printf ("%\*d\n", tbl, node->key);

if (node->left) print\_sim (node->left, tbl+5);

}

Node\* findNode(Node\* cur, Node\*\* prev, int key) {

comparing = 0;

// \*comparing = 0;

while(cur) {

if(comparing++ && cur->key == key)

return cur;

else {

\*prev = cur;

comparing++;

cur = (cur->key > key ? cur->left : cur->right);

}

}

return NULL;

}

Node\* removeNode(Node\*\* treeRoot, Node\* node) {

Node\* prev, \* higherClosest, \* p, \* ch, \* removedNode;

prev = higherClosest = p = ch = NULL;

if(node->left && node->right) {

higherClosest = findNode(node, &prev, node->key+1);

if(!higherClosest) {

higherClosest = prev;

p = prev->parent;

} else {

p = prev;

}

node->key = higherClosest->key;

if(higherClosest == p->left)

p->left = higherClosest->right;

else

p->right = higherClosest->right;

if(higherClosest->right)

higherClosest->right->parent = p;

removedNode = higherClosest;

} else {

p = node->parent;

ch = node->left ? node->left : node->right;

if(!p)

\*treeRoot = ch;

else {

if(p->left == node)

p->left = ch;

else

p->right = ch;

}

if(ch)

ch->parent = p;

removedNode = node;

}

free(removedNode);

return p;

}

void chainGrandParent(Node\* gp, Node\* p, Node\* ch) {

if(gp->left == p)

gp->left = ch;

else

gp->right = ch;

}

// returns new subtree root

Node\* rotateRight(Node\*\* treeRoot, Node\* r) {

Node\* ch = r->left;

r->left = ch->right;

if(ch->right)

ch->right->parent = r;

ch->parent = r->parent;

if(r->parent)

chainGrandParent(r->parent, r, ch);

else

\*treeRoot = ch;

ch->right = r; r->parent = ch;

return ch;

}

Node\* rotateLeft(Node\*\* treeRoot, Node\* r) {

Node\* ch = r->right;

r->right = ch->left;

if(ch->left)

ch->left->parent = r;

ch->parent = r->parent;

if(r->parent)

chainGrandParent(r->parent, r, ch);

else

\*treeRoot = ch;

ch->left = r; r->parent = ch;

return ch;

}

// SplayTree operations

SplayTree {

Node\* root;

};

SplayTree\* initSplayTree(int\* memory) {

SplayTree\* st = (SplayTree\*)malloc(sizeof(SplayTree));

\*memory += sizeof(SplayTree) + sizeof(st);

st->root = NULL;

return st;

}

int isZigZag(Node\* cur) {

Node\* p = cur->parent, \* gp = p->parent;

return (p->right == cur && gp->left == p) ||

(p->left == cur && gp->right == p);

}

void zigZag(Node\*\* treeRoot, Node\* cur) {

Node\* p = cur->parent;

if(p->right == cur && p->parent->left == p)

rotateRight(treeRoot, rotateLeft(treeRoot, p) -> parent);

else

rotateLeft(treeRoot, rotateRight(treeRoot, p) -> parent);

}

void zigZig(Node\*\* treeRoot, Node\* cur) {

Node\* p = cur->parent;

if(p->left == cur && p->parent->left == p)

rotateRight(treeRoot, rotateRight(treeRoot, p->parent));

else

rotateLeft(treeRoot, rotateLeft(treeRoot, p->parent));

}

void zig(Node\*\* treeRoot, Node\* cur) {

Node\* p = cur->parent;

if(p->left == cur)

rotateRight(treeRoot, p);

else

rotateLeft(treeRoot, p);

}

Node\* splay(Node\*\* root, Node\* cur) {

if(!cur)

return NULL;

while(cur->parent) {

if(cur->parent->parent)

if(isZigZag(cur))

zigZag(root, cur);

else

zigZig(root, cur);

else

zig(root, cur);

}

return cur;

}

Node\* insertKey(SplayTree\* tree, int key, int \* memory) {

if(!tree->root)

tree->root = makeNode(NULL, key, memory);

return splay(&tree->root, addNode(tree->root, key, memory));

}

Node\* findKey(SplayTree\* tree, int key) {

Node\* prev = NULL;

Node\* res = findNode(tree->root, &prev, key);

if(res)

return splay(&tree->root, res);

if(prev)

splay(&tree->root, prev);

return NULL;

}

int getComparing()

{

int i = comparing;

comparing = 0;

return i;

}

int removeKey(SplayTree\* tree, int key) {

Node\* prev = NULL;

Node\* res = findNode(tree->root, &prev, key);

if(res)

splay(&tree->root, removeNode(&tree->root, res));

else if(prev)

splay(&tree->root, prev);

return res ? 1 : 0;

}

**// main.cpp**

#include "stdlib.h"

#include "fstream"

#include "iomanip"

#include "io.h"

#include "fcntl.h"

#include "locale"

#include "splayTree.h"

#include "hashMap.h"

#include "stdio.h"

#include "iostream"

using namespace std;

#define SIZE 10000

struct MyKeyHash {

unsigned long operator()(const int &k) const {

return k % SIZE;

}

};

int splayMemory = 0;

int \* splayMemoryPointer = &splayMemory;

int main()

{

// создание дерева

SplayTree\* splayTree = initSplayTree(splayMemoryPointer);

// создание хеш-таблицы

HashMap<int, bool, SIZE, MyKeyHash> hashMap;

ifstream test\_numbers, my\_keys;

test\_numbers.open("test\_numbers.txt");

if (!test\_numbers)

exit(-1);

// запись ключей в деревья

long int current\_key;

while (!test\_numbers.eof())

{

test\_numbers >> current\_key;

// добавляем элемент в хеш-таблицу и splay дерево

hashMap.put(current\_key, true);

insertKey(splayTree, current\_key, splayMemoryPointer);

}

test\_numbers.close();

my\_keys.open("my\_100\_numbers.txt");

if (!my\_keys)

exit(-1);

// переменные нужны для создания статистики о работе поиска и деревьев

bool isSplayFind, isHashMapFind;

int comparingSplay, comparingHashMap;

int sumcomparingSplay, sumcomparingHashMap;

sumcomparingSplay = sumcomparingHashMap = 0;

int\* pComparingSplay = &comparingSplay;

std:: cout

<< " KEY | Splay tree | Hash map | Number of Splay tree comparisons | Number of Hash map comparisons |"

<< endl;

while (!my\_keys.eof())

{

my\_keys >> current\_key;

// ищем только что считанное число из файла

isSplayFind = findKey(splayTree, current\_key)!=NULL;

hashMap.get(current\_key, isHashMapFind);

// получаем количество сравнений для одной операции поиска

comparingHashMap = hashMap.getComparing();

comparingSplay = getComparing();

cout << setw(9) << current\_key << " |";

cout << setw(20) << isSplayFind << " |";

cout << setw(15) << isHashMapFind << " |";

cout << setw(35) << comparingSplay << " |";

cout << setw(34) << comparingHashMap << " |";

cout << endl;

sumcomparingSplay += comparingSplay;

sumcomparingHashMap += comparingHashMap;

isSplayFind = false;

isHashMapFind = false;

comparingSplay = 0;

comparingHashMap = 0;

}

my\_keys.close();

// считаем среднее арифметическое сравнений на один ключ для каждой структуры

cout << "Arithmetic Mean of Splay tree Comparisons: " << sumcomparingSplay / 100 << endl;

cout << "Arithmetic Mean of Hash map Comparisons: " << sumcomparingHashMap / 100 << endl;

cout << endl;

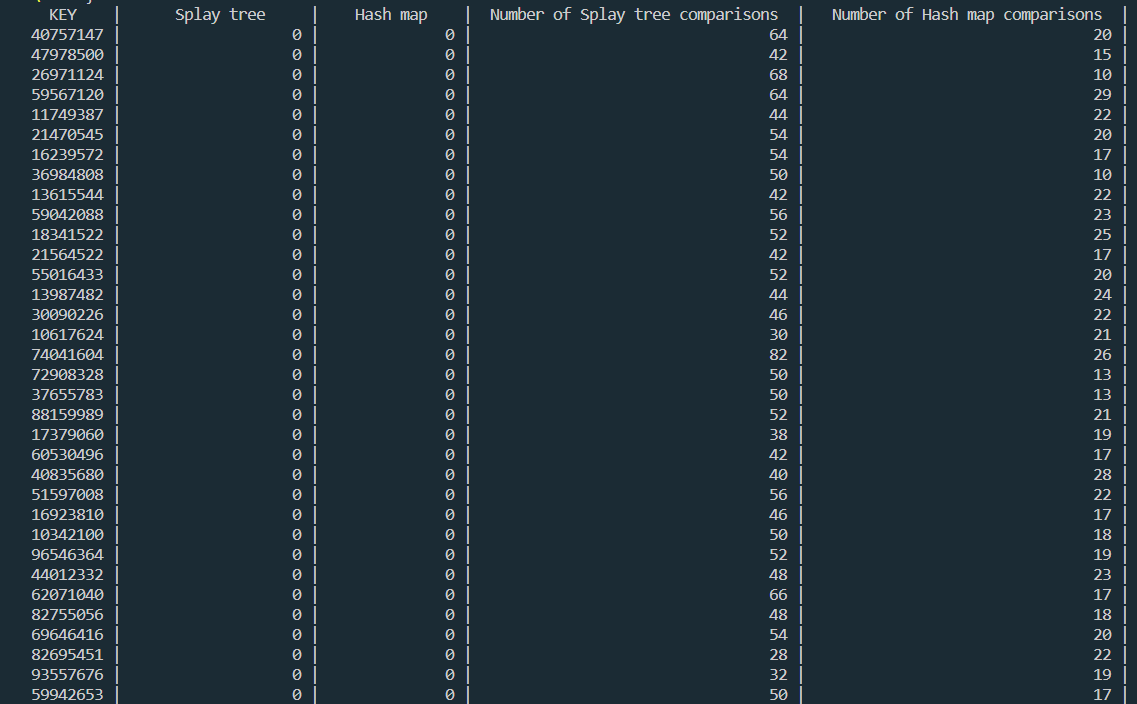
// получаем количество выделенной памяти для хранения каждой структуры

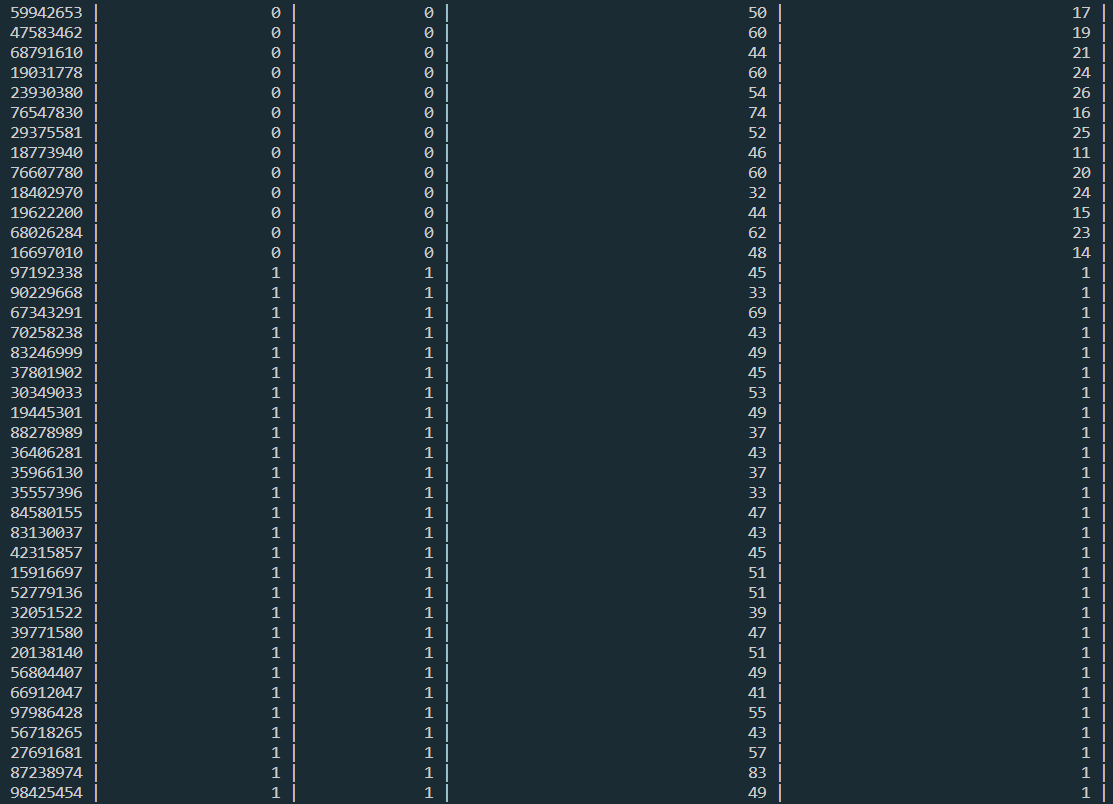
cout << "Dedicated memory for Splay tree: " << splayMemory << endl;

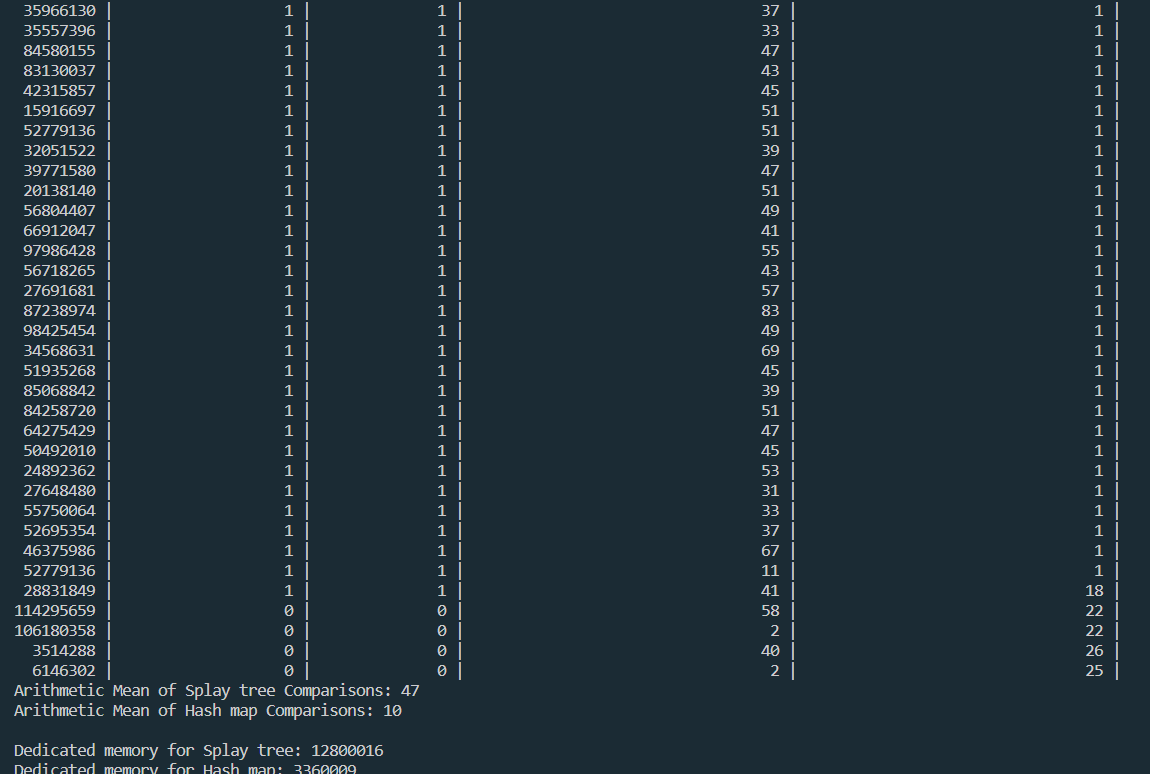
cout << "Dedicated memory for Hash map: " << hashMap.getTotalMemory() << endl;

return 0;

}

Результаты работы программы: 





Выводы:

По асимптотике наилучший случай у хеш-таблицы и splay дерева одинаковый.

По асимптотике средний вариант лучше у хеш-таблицы, что позволяет сделать очевидный выбор в пользу хеш-таблицы.

По результатам работы программы можно сделать очевидный выбор, что для поиска лучше использовать хеш-таблицу.

Обе структуры хорошо себя показали в поиске ключей, но у каждой есть свои плюсы и минусы, поэтому объективно выделить лучшую структуру сложно, каждая хорошо себя показывает во время определенных задач. И выбирать из стоит так-же для определённых задач, так что надо смотреть по обстоятельствам.