Министерство цифрового развития, связи и массовых коммуникаций

Ордена Трудового Красного Знамени федеральное государственное бюджетное

образовательное учреждение высшего образования

«Московский технический университет связи и информатики»

Отчёт по лабораторной работе № 2

«Методы поиска»

по дисциплине «Системы и алгоритмы обработки данных»

Выполнил: студент группы БВТ1905

Ахрамешин Алексей Сергеевич

Проверил:

Павликов Артем Евгеньевич

Москва

2021

Оглавление

[Цель работы 2](#_Toc72334923)

[Выполнение 3](#_Toc72334927)

[Снимки экрана выполнения программы 12](#_Toc72334931)

[Вывод 19](#_Toc72334933)

# Цель работы

В ходе проведения лабораторной работы №2 мне необходимо реализовать методы поиска в соответствии с заданием. Организовать генерацию начального набора случайных данных. Для всех вариантов добавить реализацию добавления, поиска и удаления элементов.

## Задание №1:

|  |  |  |  |
| --- | --- | --- | --- |
| **Бинарный поиск** | **Бинарное дерево** | **Фибоначчиев** | **Интерполяционный** |

## Задание №2:

|  |  |  |
| --- | --- | --- |
| **Простое рехэширование** | **Рехэширование с помощью**  **псевдослучайных чисел** | **Метод цепочек** |

## Задание № 3:

Расставить на стандартной 64-клеточной шахматной доске 8 ферзей так, чтобы ни один из них не находился под боем другого». Подразумевается, что ферзь бьёт все клетки, расположенные по вертикалям, горизонталям и обеим диагоналям

Написать программу, которая находит хотя бы один способ решения задач.

# Выполнение

# Задание №1

Код программы:

function generateArray(length) {

let array = [length],

minLimit = -25,

maxLimit = 10;

for (let i = 0; i < length; i++) {

array[i] = minLimit + Math.floor(Math.random() \* (maxLimit - minLimit + 1));

}

return array;

}

function binarySearch(value, array) {

let mass = array.sort((first, second) => first - second),

first = mass[0],

last = mass[mass.length - 1],

position = -1,

check = false,

middle;

while (check === false && first <= last) {

middle = Math.floor((last + first) / 2);

if (mass[middle] == value) {

position = middle;

check = true;

} else if (mass[middle] > value) {

last = mass[middle] - 1;

} else {

first = mass[middle] + 1;

}

}

return position;

}

function InterpolationSearch(value, array) {

let mass = array.sort((first, second) => first - second),

low = 0,

high = mass.length - 1,

zond;

while (mass[low] < value && mass[high] > value) {

zond = low + Math.floor(((value - mass[low]) \* (high - low)) / (mass[high] - mass[low]));

if (value < mass[zond]) {

high = zond - 1;

} else if (value > mass[zond]) {

low = zond + 1;

} else return zond;

}

if (mass[low] == value) return low;

else if (mass[high] == value) return high;

else return -1;

}

class Node {

constructor(data) {

this.data = data; // node value

this.left = null; // left node child reference

this.right = null; // right node child reference

}

}

class BinarySearchTree {

constructor() {

this.root = null; // корень bst

}

insert(data) {

let newNode = new Node(data);

if (this.root === null) {

this.root = newNode;

} else {

this.insertNode(this.root, newNode);

}

}

insertNode(node, newNode) {

if (newNode.data < node.data) {

if (node.left === null) {

node.left = newNode;

} else {

this.insertNode(node.left, newNode);

}

} else {

if (node.right === null) {

node.right = newNode;

} else {

this.insertNode(node.right, newNode);

}

}

}

findMinNode(node)

{

// if left of a node is null

// then it must be minimum node

if(node.left === null)

return node;

else

return this.findMinNode(node.left);

}

remove(data)

{

// root is re-initialized with

// root of a modified tree.

this.root = this.removeNode(this.root, data);

}

// Method to remove node with a

// given data

// it recur over the tree to find the

// data and removes it

removeNode(node, key)

{

// if the root is null then tree is

// empty

if(node === null)

return null;

// if data to be delete is less than

// roots data then move to left subtree

else if(key < node.data)

{

node.left = this.removeNode(node.left, key);

return node;

}

// if data to be delete is greater than

// roots data then move to right subtree

else if(key > node.data)

{

node.right = this.removeNode(node.right, key);

return node;

}

// if data is similar to the root's data

// then delete this node

else

{

// deleting node with no children

if(node.left === null && node.right === null)

{

node = null;

return node;

}

// deleting node with one children

if(node.left === null)

{

node = node.right;

return node;

}

else if(node.right === null)

{

node = node.left;

return node;

}

// Deleting node with two children

// minumum node of the rigt subtree

// is stored in aux

var aux = this.findMinNode(node.right);

node.data = aux.data;

node.right = this.removeNode(node.right, aux.data);

return node;

}

}

search(node, data) {

if (node === null) {

return 'Sorry, element is undefinded';

} else if (data < node.data) {

return this.search(node.left, data);

} else if (data > node.data) {

return this.search(node.right, data);

} else {

return node;

}

}

}

function fibonachchi(value) {

let f1 = 0,

f2 = 1,

cf = 1;

for (let i = 1; i <= value; i++) {

cf = f1 + f2;

f1 = f2;

f2 = cf;

}

return cf;

}

function fibonachchiSearch(value, start = 0, result = 0, array) {

let mass = array.sort((first, second) => first - second),

check = true,

index = 0,

f = 0;

console.log(mass);

while(check){

f = fibonachchi(index);

if(f > mass.length - 1){

f = mass.length-1;

if (mass[f] < value || mass.length == 0){return 'sorry'}

}

if(mass[f] == value){

console.log('success');

result+=f;

return result;

} else if (mass[f] > value){

start = fibonachchi(index - 1);

result+=start;

check = false;

} else { index++; }

}

if(check == false){

mass = mass.splice(start,f-1);

return fibonachchiSearch(value, start, result, mass);

}

}

let array = generateArray(100);

let test = [1,2,3,4,5,6,8,19,20,22,23];

console.log(array);

// const start = Date.now();

// console.log(binarySearch(6, [4,6,5,1,2,3,11]));

// console.log();

// const end = Date.now();

// console.log(`time is ${end-start}'ms`);

// const start = Date.now();

// console.log(InterpolationSearch(101, array))

// const end = Date.now();

// console.log(`time is ${end-start}'ms`)

let bTree = new BinarySearchTree()

array.forEach(data => bTree.insert(data))

const start = Date.now();

console.log(bTree.search(bTree.root, -20))

bTree.remove(-20)

console.log(bTree.search(bTree.root, -20))

const end = Date.now();

console.log(`time is ${end-start}'ms`)

// const startTime = Date.now();

// let start, index,result;

// console.log(fibonachchiSearch(101, start, result, array))

// const endTime = Date.now();

// console.log(`time is ${endTime-startTime}'ms`)

# Задание №2

Код программы:

Простое рехеширование:

const test = [8,19,14,12,10,5,7];

let hashTable = new Map(),

hashRefTable = [];

function hashValue(value){

return value%7;

}

function reHash(value){

const hash = hashValue(value);

for (let i = 0; i < 7; i++){

if(hashTable.has((hash+i) % 7) == false){

hashTable.set((hash+i) % 7, value);

break;

}

}

}

function simpleReHash(num, array){

array.forEach(value => reHash(value));

const hash = hashValue(num);

for (let [key, value] of hashTable){

console.log(`в ячейке ${key} cодержится ${value} `);

}

for (let i = 0; i<7; i++){

if(hashTable.get((hash + i)) == undefined){

return `Элемент ${num} не найден` ;

}

else if( num == hashTable.get((hash+i)%7)) {

return `Элемент ${num} найден в ячейке ${(hash+i)%7}`;

}

}

return ` Элемент не найден `;

}

console.log(simpleReHash(10, test));

Рехеширование методом цепочек:

const hashTable = new Map(),

test = [8,19,14,12,10,5,7];

class LinkedListNode {

constructor(value, next = null) {

this.value = value;

this.next = next;

}

append(value) {

if(this.next == null){

this.next = new LinkedListNode(value);

} else {

this.next.append(value);

}

}

find(value) {

let currentNode = this;

while (currentNode) {

if (value !== undefined && currentNode.value === value) {

return `Элемент ${value} найден в ячейке ${hashValue(value)}`;

}

currentNode = currentNode.next;

}

return null;

}

}

function hashValue(value){

return value%7;

}

function hashChain(array) {

for (let i = 0; i < array.length; i++) {

if(hashTable.has(hashValue(array[i])) == false){

let node = new LinkedListNode(array[i], null);

hashTable.set(hashValue(array[i]), node);

} else {

hashTable.get(hashValue(array[i])).append(array[i]);

}

}

return hashTable;

}

function foundChain(num) {

const hash = hashValue(num);

if (hashTable.has(hash)){

return hashTable.get(hash).find(num);

}

else {

return ` Элемент не найден `;

}

}

console.log(hashChain(test));

console.log(foundChain(7))

# Задание №3

Код программы:

class chessBoard {

constructor(a, b, c, d, e, f, g, h) {

this.a = a;

this.b = b;

this.c = c;

this.d = d;

this.e = e;

this.f = f;

this.g = g;

this.h = h;

}

}

function chessQuins() {

const board = {};

let map = new Map(),

diag1 = [],

diag2 = [];

let num = Math.floor(Math.random() \* (9 - 1) + 1);

for (let i = 0; i < 8; i++) {

const xArr = ['x','x','x','x','x','x','x','x'];

num = Math.floor(Math.random() \* (9 - 1) + 1);

while(map.get(num) == 1){

num = Math.floor(Math.random() \* (9 - 1) + 1);

}

if( diag1.indexOf(i + (num - 1)) == -1 || diag2.indexOf(Math.abs(i-(num-1))) == -1 ){

xArr[num-1] = 'Q';

map.set(num,1);

console.log(map);

diag1.push(i+(num-1));

diag2.push(Math.abs(i-(num-1)));

board[(i+1).toString()] = new chessBoard(...xArr);

} else {

i--;

}

}

return board;

}

console.table(chessQuins())

# Снимки экрана выполнения программы

Рис. 1 Работа алгоритма поиска

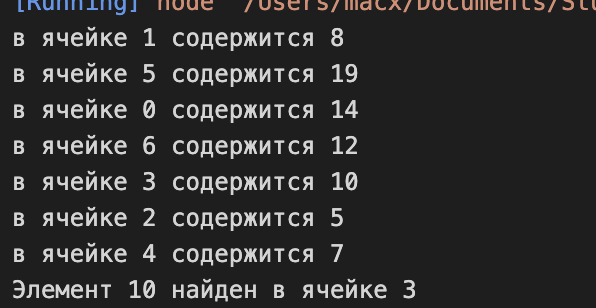


Рис. 2 Работа алгоритма хеширования

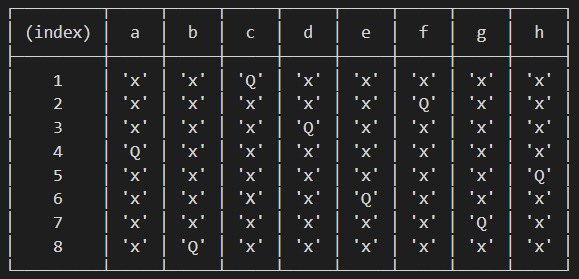


Рис. 3 Работа алгоритма расстановки ферзей

# Вывод

В ходе выполнения лабораторной работы я реализовал разные методы поиска, рехеширования, а также решил задачу на шахматную доску.

Бинарный поиск ищет при помощи сокращения области вдвое, интерполяционный метод берет пробу поиска не в середине рассматриваемой области. Фибоначчиев поиск ищет с помощью золотого сечения, а бинарное дерево представляет собой иерархическую структуру данных, в которой удобно искать, добавлять или удалять элементы.

Простое рехеширование нужно для ускоренного доступа к данным в таблицах. Рехеширование с помощью псевдослучайных чисел позволяет быстрее заполнить таблицу числом, место которого занято, а метод цепочек позволяет хранить все числа в удобном формате.