Отчёт по лабораторной работе

По дисциплине «Структуры и алгоритмы обработки данных»

На тему: «Методы поиска»

Выполнил:

Студент гр.БСТ1902

Бадмаев А.Д.

Москва 2021

Задания на лабораторную работу

1. Реализовать методы поиска (бинарный, интерполяционный, фиббоначиев) и измерить скорость работы.
2. Реализовать методы рехэширования (простое, с помощью псевдослучайных чисел, метод цепочек)
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.

Ход работы

Выполнение первого задания

```
// Создаём массив

function genArray(m, min_limit, max_limit) {
    var arr = new Array();
    for (var i = 0; i < m; i++) {
        arr[i] = Math.floor(Math.random() * (max_limit - min_limit + 1)) + min_li
mit;
    }
    return arr;
}
```

```
// Бинарный поиск

const binarySearch = (arr, target) => {
    let start = 0;
    let end = arr.length;
    let pivot = Math.floor((start + end) / 2);

    for (let i = 0; i < arr.length; i++) {
        if (arr[pivot] !== target) {
            if (target < arr[pivot]) end = pivot;
            else start = pivot;
            pivot = Math.floor((start + end) / 2);;
        };
```

```
        if (arr[pivot] === target)
            return pivot;
    };

    return -1;
};
```

```
// Интерполяционный поиск

function interSearch(arr, target) {
    let mid
    let low = 0
    let high = arr.length - 1;

    while (arr[low] < target && arr[high] > target) {
        mid = low + Math.floor(((target - arr[low]) * (high - low)) / (arr[high]
- arr[low]));
        if (arr[mid] < target) low = mid + 1;
        else if (arr[mid] > target) high = mid - 1;
        else return mid;
    }

    if (arr[low] === target) return low;
    else if (arr[high] === target) return high;
    else return -1;
}
```

```
// Фибоначчиев поиск

function fibonacSearch(arr, target) {
    let fibMMm2 = 0;
    let fibMMm1 = 1;
    let fibM = fibMMm2 + fibMMm1;
    let n = arr.length;
    while (fibM < n) {
        fibMMm2 = fibMMm1;
        fibMMm1 = fibM;
        fibM = fibMMm2 + fibMMm1;
    }
    let offset = -1;

    while (fibM > 1) {
        let i = Math.min(offset + fibMMm2, n - 1);
        if (arr[i] < target) {
            fibM = fibMMm1;
            fibMMm1 = fibMMm2;
            fibMMm2 = fibM - fibMMm1;
```

```
            offset = i;
        }
        else if (arr[i] > target) {
            fibM = fibMMm2;
            fibMMm1 = fibMMm1 - fibMMm2;
            fibMMm2 = fibM - fibMMm1;
        }
        else return i;
    }
    if (fibMMm1 && arr[n - 1] == target) {
        return n - 1
    }
    return -1;
}
```

```
//   Бинарное дерево

// Создаём класс узла

class Node {
    constructor(data) {
      this.data = data;
      this.left = null;
      this.right = null;
    }
  }

// Создаём класс дерева

class BinaryTree {
    constructor() {
        this.root = null;
        this.size = 0;
    }
    add(data) {
        const newNode = new Node(data);
        if (this.root === null) {
            this.root = newNode;
            this.size++;
        }
        let current = this.root;
        while (true) {
            if (data < current.data) {
                if (current.left === null) {
                    current.left = newNode;
                    this.size++;
                    break;
                } else {
                    current = current.left;
```

```
                }
            } else if (data > current.data) {
                if (current.right === null) {
                    current.right = newNode;
                    this.size++;
                    break;
                } else {
                    current = current.right;
                }
            } else {
                break;
            }
        }
    }
    /**
     * Максимальный узел в дереве
     * @memberof BinaryTree
     */
    getMax() {
        let current = this.root;
        while (current.right !== null) {
            current = current.right;
        }
        return current.data;
    }
    /**
     * Минимальный узел в дереве
     * @memberof BinaryTree
     */
    getMin() {
        let current = this.root;
        while (current.left !== null) {
            current = current.left;
        }
        return current.data;
    }
    /**
     * Количество узлов в дереве
     * @memberof BinaryTree
     */
    size() {
        return this.size;
    }
    find(data) {
        let current = this.root;
        while (current.data !== data) {
            if (data < current.data) {
                current = current.left;
            } else {
                current = current.right;
            }
```

```
            if (current === null) {
                return false;
            }
        }
        return true;
    }
    /**
     * Прямой обход дерева
     * @param {any} node
     * @memberof BinaryTree
     */
    preOrder(node) {
        if (node === null) {
            return;
        }
        console.log(node.data);
        this.preOrder(node.left);
        this.preOrder(node.right);
    }
    /**
     * Симметричный обход дерева
     * @param {any} node
     * @memberof BinaryTree
     */
    inOrder(node) {
        if (node === null) {
            return;
        }
        this.inOrder(node);
        console.log(node.data);
        this.inOrder(node);
    }
    /**
     * Обратный обход дерева
     * @param {any} node
     * @memberof BinaryTree
     */
    postOrder(node) {
        if (node === null) {
            return;
        }
        this.inOrder(node);
        this.inOrder(node);
        console.log(node.data);
    }
    /**
     * Обход дерева в ширину
     * @param {any} node
     * @memberof BinaryTree
     */
    bfs(node) {
```

```
        let queue = [];
        let values = [];
        queue.push(node);
        while (queue.length > 0) {
            let current = queue.shift();
            values.push(current.data);
            if (current.left) {
                queue.push(current.left);
            }
            if (current.right) {
                queue.push(current.right);
            }
        }
        return values;
    }
    remove(data) {
        const removeNode = function (node, data) {
            if (node == null) {
                return null;
            }
            if (data == node.data) {
                // У узла нет детей
                if (node.left === null && node.right === null) {
                    return null;
                }
                // У узла только правый ребенок
                if (node.left === null) {
                    return node.right;
                }
                // У узла только левый ребенок
                if (node.right === null) {
                    return node.left;
                }
                // У узла двое детей
                var current = node.right;
                while (current.left !== null) {
                    current = current.left;
                }
                node.data = current.data;
                node.right = removeNode(node.right, current.data);
                return node;
            } else if (data < node.data) {
                node.left = removeNode(node.left, data);
                return node;
            } else {
                node.right = removeNode(node.right, data);
                return node;
            }
        };
        this.root = removeNode(this.root, data);
    }
```

Результат выполнения первого задания

```
[
  5, 9, 0, 10, 7,
  3, 3, 2,  4, 3
]
[
  0, 2, 3, 3,  3,
  4, 5, 7, 9, 10
]
Бинарный поиск: 0.167ms
Позиция: 2
[
  0, 2, 3, 3,  3,
  4, 5, 7, 9, 10
]
Интерполяционный поиск: 0.101ms
Позиция: 2
[
  0, 2, 3, 3,  3,
  4, 5, 7, 9, 10
]
Поиск Фибоначчи: 0.135ms
Позиция: 4
BinaryTree {
  root: Node {
    data: 0,
    left: null,
    right: Node { data: 2, left: null, right: [Node] }
  },
  size: 8
```

Выполнение второго задания

```
// Создание хеш таблицы

class HashTable {
    constructor() {
        this.table = new Array(137);
        this.values = [];
    }

    hash(string) {
        const H = 37;
        let total = 0;

        for (var i = 0; i < string.length; i++) {
            total += H * total + string.charCodeAt(i);
        }

        total %= this.table.length;

        if (total < 1) {
            this.table.length - 1;
```

```
        }

        return parseInt(total);
    }

    showTable() {
        for (const key in this.table) {
            if (this.table[key] !== undefined) {
                console.log(key, " : ", this.table[key]);
            }
        }
    }

    put(data) {
        const pos = this.hash(data);
        this.table[pos] = data;
    }

    get(key) {
        return this.table[this.hash(key)];
    }
}
// Хеш таблица с рехешированием методом цепочек
class HashTableChains extends HashTable {
    constructor() {
        super();
        this.buildChains();
    }

    buildChains() {
        for (var i = 0; i < this.table.length; i++) {
            this.table[i] = new Array();
        }
    }

    showTable() {
        for (const key in this.table) {
            if (this.table[key][0] !== undefined) {
                console.log(key, " : ", this.table[key]);
            }
        }
    }

    put(key, data) {
        const pos = this.hash(key);
        let index = 0;
        if (this.table[pos][index] === undefined) {
            this.table[pos][index] = data;
        } else {
            ++index;
            while (this.table[pos][index] !== undefined) {
```

```
                index++;
            }
            this.table[pos][index] = data;
        }
    }

    get(key) {
        const pos = this.hash(key);
        let index = 0;
        while (this.table[pos][index] != key) {
            if (this.table[pos][index] !== undefined) {
                return this.table[pos][index];
            } else {
                return undefined;
            }
            index++;
        }
    }
}
// хеш таблица с простым/линейным рехешированием
class HashTableLinearP extends HashTable {
    constructor() {
        super();
        this.values = new Array();
    }

    put(key, data) {
        let pos = this.hash(key);
        if (this.table[pos] === undefined) {
            this.table[pos] = key;
            this.values[pos] = data;
        } else {
            while (this.table[pos] !== undefined) {
                pos++;
            }
            this.table[pos] = key;
            this.values[pos] = data;
        }
    }

    get(key) {
        const hash = this.hash(key);
        if (hash > -1) {
            for (let i = hash; this.table[i] !== undefined; i++) {
                if (this.table[i] === key) {
                    return this.values[i];
                }
            }
        }
        return undefined;
    }
```

```
    remove(key) {
        const hashCode = this.hash(key);
        let list = this.table[hashCode];

        if (!list) {
            return;
        }

        list = undefined;
    }

    showTable() {
        for (const key in this.table) {
            if (this.table[key] !== undefined) {
                console.log(key, " : ", this.values[key]);
            }
        }
    }
}
// хеш таблица с псевдо-рандомных рехешированием
class HashTableRandom extends HashTableLinearP {
    constructor() {
        super();
    }

    hash(seed) {
        let total = (seed + ((125 * seed + 2784) % 5897)) % this.table.length;

        if (total < 1) {
            this.table.length - 1;
        }

        return parseInt(total);
    }
}
```

Результат выполнения второго задания

Простое/линейное рехеширование

```
0  :  0
1  :  2
2  :  3
3  :  3
4  :  3
5  :  4
6  :  5
7  :  7
8  :  9
9  :  10
```

Псевдо-рандомное рехеширование

```
28  :  0
47  :  4
50  :  3
59  :  3
64  :  3
69  :  5
70  :  9
98  :  7
103  :  2
120  :  10
```

Методом цепочек

```
0  :  [
  0, 2, 3, 3,  3,
  4, 5, 7, 9, 10
]
```

Выполнение третьего задания

```
/* 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 к
летки,
расположенные по вертикалям, горизонталям и обеим диагоналям
Написать программу, которая находит хотя бы один способ решения задач. */

const OCCUPIED = 1, //метка "поле бьётся"
    FREE = 0, //метка "поле не бьётся"
    ISHERE = -1; //метка "ферзь тут"

class Queen {
    constructor(N) {
        this.N = N;

        for (let i = 0; i < 2 * this.N - 1; i++) {
            if (i < this.N) this.columns[i] = ISHERE;

            this.diagonals1[i] = FREE;
            this.diagonals2[i] = FREE;
        }
    }

    columns = [];
    solutions = [];
    diagonals1 = [];
    diagonals2 = [];

    run(row = 0) {
        for (let column = 0; column < this.N; ++column) {
            if (this.columns[column] >= 0) {
```

```
                //текущий столбец бьётся, продолжить
                continue;
            }

            let thisDiag1 = row + column;

            if (this.diagonals1[thisDiag1] == OCCUPIED) {
                //диагональ '\' для текущих строки и столбца бьётся, продолжить
                continue;
            }
            let thisDiag2 = this.N - 1 - row + column;

            if (this.diagonals2[thisDiag2] == OCCUPIED) {
                //диагональ '/' для текущих строки и столбца бьётся, продолжить
                continue;
            }

            this.columns[column] = row;
            this.diagonals1[thisDiag1] = OCCUPIED; //занять диагонали, которые те
перь бьются
            this.diagonals2[thisDiag2] = OCCUPIED;

            if (row == this.N - 1) {
                //найдена последняя строка - есть решение
                this.solutions.push(this.columns.slice());
            } else {
                //иначе рекурсия
                this.run(row + 1);
            }

            this.columns[column] = ISHERE;
            this.diagonals1[thisDiag1] = FREE;
            this.diagonals2[thisDiag2] = FREE;
        }
    }
}

function getLine(solution) {
    return solution.reduce((previous, current, currentIndex) => {
        return previous + `(${currentIndex + 1},${current + 1})`;
    }, "");
}

function queenPositions(N = 8) {
    let table = new Queen(N);

    console.log(`Размер доски: ${table.N}x${table.N}`);
    console.time("Время вычисления");
    table.run();

    console.timeEnd("Время вычисления");
```

```
    console.log(`Количество решений: ${table.solutions.length}`);

    // table.solutions.forEach((solution) => console.log(getLine(solution)));
}

queenPositions();
```

Результат выполнения третьего задания

```
Размер доски: 8x8
Время вычисления: 3.268ms
Количество решений: 92
(1,1)(2,7)(3,5)(4,8)(5,2)(6,4)(7,6)(8,3)
(1,1)(2,7)(3,4)(4,6)(5,8)(6,2)(7,5)(8,3)
(1,1)(2,6)(3,8)(4,3)(5,7)(6,4)(7,2)(8,5)
(1,1)(2,5)(3,8)(4,6)(5,3)(6,7)(7,2)(8,4)
(1,6)(2,1)(3,5)(4,2)(5,8)(6,3)(7,7)(8,4)
(1,4)(2,1)(3,5)(4,8)(5,2)(6,7)(7,3)(8,6)
(1,5)(2,1)(3,8)(4,4)(5,2)(6,7)(7,3)(8,6)
(1,3)(2,1)(3,7)(4,5)(5,8)(6,2)(7,4)(8,6)
(1,5)(2,1)(3,4)(4,6)(5,8)(6,2)(7,7)(8,3)
(1,7)(2,1)(3,3)(4,8)(5,6)(6,4)(7,2)(8,5)
(1,5)(2,1)(3,8)(4,6)(5,3)(6,7)(7,2)(8,4)
(1,4)(2,1)(3,5)(4,8)(5,6)(6,3)(7,7)(8,2)
```