

Отчет по лабораторной работе

по дисциплине «Структуры и алгоритмы обработки данных»

на тему:

«Методы поиска»

Выполнил:

Студент группы

БСТ1902

Игнатов В.С.

Вариант №5

Москва 2021

Оглавление

Задания на лабораторную работу	4
Ход работы.....	4
Задание 1	4
Задание 2	11
Задание 3	15

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

Задание 1

Создадим генератор случайных чисел и напомним 3 функции поиска:

```
/* Случайные данные */
let data = randomData(10);
function randomData(n = 10, minLim = 0, maxLim = 10) {
  let data = [];

  for (let j = 1; j <= n; j++) {
    let elem = Math.floor(minLim + Math.random() * (maxLim + 1 - minLim));
    data.push(elem);
  }

  return data;
}
```

```
/* Бинарный поиск */
function binarySearch(value, list) {
  let first = 0,
      last = list.length - 1,
      position = -1,
      found = false,
      middle;

  while (found === false && first <= last) {
    middle = Math.floor((first + last) / 2);

    if (list[middle] == value) {
      found = true;
    }
  }
}
```

```

        position = middle;
    } else if (List[middle] > value) {
        last = middle - 1;
    } else {
        first = middle + 1;
    }
}

return position;
}

/* Фиббоначиев поиск */
function fibMonaccianSearch(x, arr) {
    let n = arr.length;
    let fibMMm2 = 0;
    let fibMMm1 = 1;
    let fibM = fibMMm2 + fibMMm1;

    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] < x) {
            fibM = fibMMm1;
            fibMMm1 = fibMMm2;
            fibMMm2 = fibM - fibMMm1;
            offset = i;
        } else if (arr[i] > x) {
            fibM = fibMMm2;
            fibMMm1 = fibMMm1 - fibMMm2;
            fibMMm2 = fibM - fibMMm1;
        } else return i;
    }

    if (fibMMm1 && arr[n - 1] == x) {
        return n - 1;
    }

    return -1;
}

/* Интерполяционный поиск */

```

```

function interpolationSearch(key, arr) {
  let mid,
      left = 0,
      right = arr.length - 1;

  while (arr[left] < key && key < arr[right]) {
    mid =
      left +
      Math.floor(
        ((key - arr[left]) * (right - left)) / (arr[right] - arr[left])
      );

    if (arr[mid] < key) {
      left = mid + 1;
    } else if (arr[mid] > key) {
      right = mid - 1;
    } else {
      return mid;
    }
  }

  if (arr[left] == key) return left;
  else if (arr[right] == key) return right;
  else return -1;
}

/* Вызов функций поиска */
dataProcess(data, 3, binarySearch, "Бинарный поиск");
dataProcess(data, 3, fibMonaccianSearch, "Фиббоначиев поиск");
dataProcess(data, 3, interpolationSearch, "Интерполяционный поиск");

```

Результат работы:

Бинарный поиск: 0.04736328125 ms
Позиция: 2
► (10) [1, 2, 3, 4, 5, 6, 7, 7, 10, 10]
Фиббоначиев поиск: 0.12109375 ms
Позиция: 2
► (10) [1, 2, 3, 4, 5, 6, 7, 7, 10, 10]
Интерполяционный поиск: 0.14208984375 ms
Позиция: 2
► (10) [1, 2, 3, 4, 5, 6, 7, 7, 10, 10]

Код бинарного дерева

```

/* Бинарное дерево */
class Node {
  constructor(data) {

```

```

        this.data = data;
        this.left = null;
        this.right = null;
    }
}

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);
}
}

let tree = new BinaryTree();

data.forEach((item) => {
    tree.add(item);
});

```

Результат работы программы:

```
▼ BinaryTree {root: Node, size: 9} ⓘ  
  ▼ root: Node  
    data: 1  
    ▼ left: Node  
      data: 0  
      left: null  
      right: null  
      ▶ __proto__: Object  
    ▼ right: Node  
      data: 5  
      ▶ left: Node {data: 4, left: Node, right: null}  
      ▼ right: Node  
        data: 8  
        left: null  
        ▶ right: Node {data: 9, left: null, right: Node}  
        ▶ __proto__: Object  
      ▶ __proto__: Object  
    ▶ __proto__: Object  
  size: 9  
  ▶ __proto__: Object
```

Задание 2

```
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)];
}
}

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++;
    }
}
}

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]);
        }
    }
}
}

class HashTableRandom extends HashTableLinearP {
    constructor() {
        super();
    }

    hash(string) {
        let total = (string + ((625 * string + 6571) % 31104)) % this.table.length;

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

        return parseInt(total);
    }
}

let table = new HashTableLinearP();
data.forEach((item) => {
    table.put(Math.random(), item);
});
table.showTable();

let tableChains = new HashTableChains();
data.forEach((item) => {
    tableChains.put(Math.random(), item);
});
tableChains.showTable();

let tableRandom = new HashTableRandom();
data.forEach((item) => {
    tableRandom.put(Math.random(), item);
});
tableRandom.showTable();

```

Результат вывода:

0	:	1
1	:	5
2	:	8
3	:	0
4	:	9
5	:	4
6	:	4
7	:	10
8	:	2
9	:	3
0	:	► (10) [1, 5, 8, 0, 9, 4, 4, 10, 2, 3]
0	:	10
22	:	1
35	:	5
38	:	0
50	:	4
53	:	2
54	:	9
121	:	8
123	:	4
135	:	3
>		

Задание 3

```
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 теперь бью
тся
    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 работы:

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)

Вывод

Я изучил методы поиска, хеш-таблицы, методы хеширования и реализовал на практике.