Министерство цифрового развития, связи и массовых коммуникаций

Ордена Трудового Красного Знамени

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский технический университет связи и информатики»

Отчет по лабораторной работе №2

по дисциплине «Структура и алгоритмы обработки данных»

по теме «Методы поиска»

Выполнил: студент группы

БВТ1902

Подпоркин В.С.

Проверил:

Москва

2021 г.

Оглавление

Цель работы

Реализовать методы поиска, простое рехеширование и метод цепочек в соответствии с заданием. Организовать генерацию начального набора случайных данных и оценить время работы каждого алгоритма поиска.

Задание 1

Бинарный поиск	Бинарное дерево	Фибоначчиев	Интерполяционный

Код программы

```rust
// --------------
// Бинарный поиск
// --------------
pub fn binary_search<T: PartialOrd>(arr: &[T], el: &T) -> Option<usize> {
    let mut left = 0usize;
    let mut right = arr.len() - 1;
    loop {
        if left > right || right == usize::MAX { return None }
        let mid = left + (right - left) / 2;
        if &arr[mid] < el { left = mid + 1; }
        if &arr[mid] > el { right = mid.wrapping_sub(1); }
        if &arr[mid] == el { return Some(mid); }
    }
}

// ----------------
// Интерполяционный
// ----------------
pub fn interpolation_search(sorted_array: &[i32], to_find: i32) -> Option<usize>
{
    // Возвращает индекс элемента со значением to_find или None, если такого
элемента не существует
    let mut low = 0usize;
    let mut high = sorted_array.len() - 1;

    while sorted_array[low] < to_find && sorted_array[high] > to_find {
        if sorted_array[high] == sorted_array[low] {// Защита от деления на 0
            break;
        }

        let mid = low + (((to_find - sorted_array[low]) as usize) * (high -
low)) / ((sorted_array[high] - sorted_array[low]) as usize);

        match sorted_array[mid].cmp(&to_find) {
            Ordering::Less => low = mid + 1,
            Ordering::Greater => high = mid - 1,
            Ordering::Equal => return Some(mid),
```

```
        }
    }

    if sorted_array[low] == to_find {
        Some(low)
    } else if sorted_array[high] == to_find {
        Some(high)
    } else {
        None
    }
}

// -----------------
// Фибонначиев поиск
// -----------------
pub fn fibonacci_search(lys: &[i32], val: i32) -> Option<usize> {
    let mut fib_m_minus_2 = 0i32;
    let mut fib_m_minus_1 = 1i32;
    let mut fib_m = fib_m_minus_1 + fib_m_minus_2;
    while (fib_m as usize) < lys.len() {
        fib_m_minus_2 = fib_m_minus_1;
        fib_m_minus_1 = fib_m;
        fib_m = fib_m_minus_1 + fib_m_minus_2;
    }
    let mut index = -1;
    while fib_m > 1 {
        let i = usize::min((index + fib_m_minus_2) as usize, lys.len() - 1);
        if lys[i] < val {
            fib_m = fib_m_minus_1;
            fib_m_minus_1 = fib_m_minus_2;
            fib_m_minus_2 = fib_m - fib_m_minus_1;
            index = i as i32;
        }
        else if lys[i] > val {
            fib_m = fib_m_minus_2;
            fib_m_minus_1 = fib_m_minus_1 - fib_m_minus_2;
            fib_m_minus_2 = fib_m - fib_m_minus_1;
        }
        else {
            return Some(i);
        }
    }
    if (index as usize) < (lys.len()-1) && lys[(index as usize)+1] == val {
        return Some(index as usize + 1);
    }
    return None;
}

// -------------------------
// Поиск по бинарному дереву
// -------------------------
#[derive(Debug)]
pub struct Node<T> {
```

```
    pub left: Option<Box<Node<T>>>,
    pub value: T,
    pub right: Option<Box<Node<T>>>,
}

impl<T> Node<T> {
    pub fn new(value: T) -> Self {
        Node { left: None, value, right: None, }
    }

    fn add_to_node(&mut self, value: T) where T: Ord {
        #[inline]
        fn add_optionally<T: Ord>(node: &mut Option<Box<Node<T>>>, value: T) {
            match node {
                Some(boxed) => boxed.add_to_node(value),
                None => *node = Some(Box::new(Node::new(value))),
            }
        }
        match value.cmp(&self.value) {
            Ordering::Less => add_optionally(&mut self.left, value),
            Ordering::Greater => add_optionally(&mut self.right, value),
            Ordering::Equal => ()/* Значение уже существует */,
        }
    }

    pub fn has_value(&self, value: &T) -> bool where T: Ord {
        match value.cmp(&self.value) {
            Ordering::Less => match &self.left {
                Some(node) => node.has_value(value),
                None => false,
            },
            Ordering::Greater => match &self.right {
                Some(node) => node.has_value(value),
                None => false,
            },
            Ordering::Equal => true,
        }
    }

}


#[derive(Debug)]
pub struct MyBinaryTree<T> {
    root: Option<Node<T>>,
}

impl<T> MyBinaryTree<T> {
    pub fn new() -> Self {
        Self { root: None }
    }
```

```
    pub fn add(&mut self, value: T) where T: Ord {
        if let Some(root) = &mut self.root {
            root.add_to_node(value);
        } else {
            self.root = Some(Node::new(value));
        }
    }

    pub fn has_value(&self, value: &T) -> bool where T: Ord {
        if let Some(root) = &self.root {
            root.has_value(value)
        } else {
            false
        }
    }
}
```

Задание 2

Простое рехеширование	Метод цепочек

Код программы

```
const INIT_SIZE: usize = 16;

pub struct ReHashMap<T> {
    inner: Vec<Option<T>>,
}

impl<T> ReHashMap<T> {
    pub fn new(size: usize) -> Self { Self { inner: (0..size).map(|_|
None).collect() } }

    pub fn add(&mut self, value: T) where T: Hash + Eq {
        let mut hasher = DefaultHasher::new();
        value.hash(&mut hasher);
        let hash = hasher.finish();

        for i in 0..self.inner.len() {
            let size = self.inner.len();
            let idx = ((hash as usize) % size + i) % size;
            if self.inner[idx].is_none() {
                self.inner[idx] = Some(value);
                return;
            }
            if self.inner[idx].is_some() && *self.inner[idx].as_ref().unwrap()
== value {
                return;
            }
```

```
        }
        panic!("Хеш-карта заполнена.")
    }

    pub fn has_value(&self, value: &T) -> bool where T: Hash + Eq {
        let mut hasher = DefaultHasher::new();
        value.hash(&mut hasher);
        let hash = hasher.finish();

        for i in 0..self.inner.len() {
            let size = self.inner.len();
            let idx = ((hash as usize) % size + i) % size;
            match &self.inner[idx] {
                Some(x) => if x.eq(value) { return true },
                _ => (),
            }
        }
        false
    }
}


struct Chain<T> {
    value: T,
    next: Option<Box<Chain<T>>>,
}

impl<T> Chain<T> {
    pub fn find(&self, value: &T) -> bool where T: Eq {
        if self.value == *value {
            true
        } else if let Some(next) = &self.next {
            next.find(value)
        } else {
            false
        }
    }

    pub fn put(&mut self, value: T) {
        match &mut self.next {
            Some(next) => next.put(value),
            None => self.next = Some(Box::new(Chain { value, next: None })),
        }
    }
}

pub struct ChainHashMap<T> {
    inner: Vec<Option<Chain<T>>>,
```

```
}

impl<T> ChainHashMap<T> {
    pub fn new(size: usize) -> Self { Self { inner: (0..size).map(|_|
None).collect() } }

    pub fn add(&mut self, value: T) where T: Hash + Eq {
        let mut hasher = DefaultHasher::new();
        value.hash(&mut hasher);
        let hash = hasher.finish();
        let idx = (hash as usize) % self.inner.len();

        match &mut self.inner[idx] {
            Some(chain) => chain.put(value),
            None => self.inner[idx] = Some(Chain { value, next: None }),
        }
    }

    pub fn has_value(&self, value: &T) -> bool where T: Hash + Eq {
        let mut hasher = DefaultHasher::new();
        value.hash(&mut hasher);
        let hash = hasher.finish();
        let idx = (hash as usize) % self.inner.len();

        match &self.inner[idx] {
            Some(chain) => chain.find(value),
            None => false,
        }
    }
}
```

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

Код программы

```
#[derive(Copy, Clone)]
pub struct Figure
{
    pub x: i32,
    pub y: i32,
}

impl Figure {
    pub fn new(x: i32, y: i32) -> Self {
        Self { x, y }
```

```
    }
}

pub struct Chess {
    board: Box<[[bool; 8]; 8]>,
}

impl Chess {
    pub fn new() -> Self {
        Self {
            board: Box::new([[true; 8]; 8]),
        }
    }

    pub fn reset(&mut self) {
        for line in &mut self.board.iter_mut() {
            for x in line {
                *x = true;
            }
        }
    }

    pub fn remove_position(&mut self, mut i: usize, mut j: usize) {
        for k in 0..8 {
            self.board[i][k] = false;
            self.board[k][j] = false;
        }

        let mut temp_i = i;
        let mut temp_j = j;

        temp_i -= usize::min(temp_i, temp_j);
        temp_j -= usize::min(temp_i, temp_j);

        while temp_i =≠ 7 && temp_j =≠ 7 {
            self.board[temp_i][temp_j] = false;
            temp_i += 1;
            temp_j += 1;
        }

        while i=≠7 && j=≠0 {
            i += 1;
            j -= 1;
        }

        while i=≠0 && j=≠7
        {
            self.board[i][j]=false;
```

```
        i -= 1;
        j += 1;
    }
}

pub fn check(temp: Figure, table: &[Figure]) -> bool {
    for i in table {
        if i.x == temp.x && i.y == temp.y {
            return true;
        }
    }
    false
}

pub fn solve(&mut self) -> Vec<Figure> {
    let mut places = Vec::new();
    let mut i = 0i32;
    let mut j = 0;
    while places.len()≠8 {
        if self.board[i as usize][j as usize] {
            let place = Figure::new(j, i);
            places.push(place);
            self.remove_position(i as usize, j as usize);
            i += 2;
            j += 1;
        }
        if i>7
        {
            i = 0;
        }

        while j < 8 && !self.board[i as usize][j as usize]
        {
            i = (i + 1) % 8;
        }
    }
    return places;
}

pub fn printBoard(&self, table: &[Figure]) {
    let mut quieen = Figure::new(0, 0);

    for i in 0..8 {
        for j in 0..8 {
            quieen.x = j;
            quieen.y = i;

            if Self::check(quieen, table) {
                print!("Q ");
```

```
                } else if self.board[i as usize][j as usize] {
                    print!("+ ");
                } else {
                    print!("- ");
                }
            }
            println!();
        }
    }
}
```

Снимки экрана работы программ

```
Исходный массив: [0, 0, 3, 4, 5, 5, 6, 8, 8, 9, 11, 12, 14, 16, 16, 17, 18, 18, 19, 19]
    Binary | Tree | Fib | Inter. | ReHash | Chains
00: +        +      +     +        +        +
01: -        -      -     -        -        -
02: -        -      -     -        -        -
03: +        +      +     +        +        +
04: +        +      +     +        +        +
05: +        +      +     +        +        +
06: +        +      +     +        +        +
07: -        -      -     -        -        -
08: +        +      +     +        +        +
09: +        +      +     +        +        +
10: -        -      -     -        -        -
11: +        +      +     +        +        +
12: +        +      +     +        +        +
13: -        -      -     -        -        -
14: +        +      +     +        +        +
15: -        -      -     -        -        -
16: +        +      +     +        +        +
17: +        +      +     +        +        +
18: +        +      +     +        +        +
19: +        +      +     +        +        +
```

Рисунок 1 – Проверка корректности поисков (+ нашёл, - не нашёл)

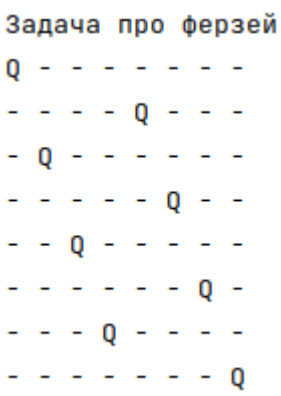

Рисунок 2 – Результат поиска позиций для 8 ферзей

Вывод

Я реализовал методы поиска: бинарный, фиббоначиев, интерполяционный. Создал структуры данных: бинарноре дерево поиска, хеш карта на основе рехеширования и хеш карту на методе цепочек. Организовал генерации начальных значений. А так же написал алгоритм решения задачи про 8 ферзей.