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 3](#__RefHeading___Toc10834_3810339557)

[Задание 1 3](#__RefHeading___Toc10836_3810339557)

[Код программы 3](#__RefHeading___Toc10838_3810339557)

[Задание 2 6](#__RefHeading___Toc10840_3810339557)

[Код программы 6](#__RefHeading___Toc10842_3810339557)

[Задание 3 8](#__RefHeading___Toc10844_3810339557)

[Код программы 8](#__RefHeading___Toc10846_3810339557)

[Снимки экрана работы программ 11](#__RefHeading___Toc10848_3810339557)

[Вывод 12](#__RefHeading___Toc10850_3810339557)

## 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

*// --------------*  
*// 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

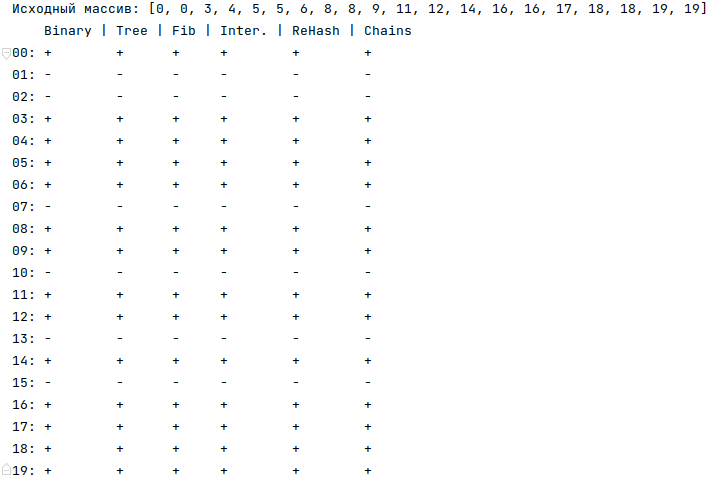
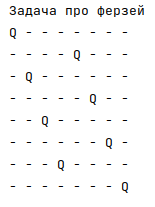


Рисунок 1 – Проверка корректности поисков (+ нашёл, - не нашёл)

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.