

**Міністерство освіти і науки України**  
**Національний технічний університет України «Київський політехнічний**  
**інститут імені Ігоря Сікорського»**  
**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторної роботи № 3 з дисципліни  
«Проектування алгоритмів»

„Проектування структур даних”

**Виконав(ла)**

ІП-11, Книш Д.О.  
(шифр, прізвище, ім'я, по батькові)

**Перевірив**

Головченко М.Н.  
(прізвище, ім'я, по батькові)

Київ 2022

## ЗМІСТ

<b>1. МЕТА ЛАБОРАТОРНОЇ РОБОТИ.....</b>	<b>3</b>
<b>2. ЗАВДАННЯ.....</b>	<b>4</b>
<b>3. ВИКОНАННЯ.....</b>	<b>7</b>
3.1.ПСЕВДОКОД АЛГОРИТМІВ.....	7
3.2.ЧАСОВА СКЛАДНІСТЬ ПОШУКУ .....	9
3.3.ПРОГРАМНА РЕАЛІЗАЦІЯ .....	9
3.3.1.Вихідний код.....	9
3.3.2.Приклади роботи .....	18
3.4.ТЕСТУВАННЯ АЛГОРИТМУ.....	20
3.4.1.Часові характеристики оцінювання .....	20
<b>ВИСНОВОК .....</b>	<b>21</b>
<b>КРИТЕРІЇ ОЦІНЮВАННЯ.....</b>	<b>22</b>

1.

## МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи – вивчити основні підходи проектування та обробки складних структур даних.

Відповідно до варіанту (таблиця 2.1), записати алгоритми пошуку, додавання, видалення і редагування запису в структурі даних за допомогою псевдокоду (чи іншого способу по вибору).

Записати часову складність пошуку в структурі в асимптотичних оцінках.

Виконати програмну реалізацію невеликої СУБД з графічним (не консольним) інтерфейсом користувача (дані БД мають зберігатися на ПЗП), з функціями пошуку (алгоритм пошуку у вузлі структури згідно варіанту таблиця 2.1, за необхідності), додавання, видалення та редагування записів (запис складається із ключа і даних, ключі унікальні і цілочисельні, даних може бути декілька полів для одного ключа, але достатньо одного рядка фіксованої довжини). Для зберігання даних використовувати структуру даних згідно варіанту (таблиця 2.1).

Заповнити базу випадковими значеннями до 10000 і зафіксувати середнє (із 10-15 пошуків) число порівнянь для знаходження запису по ключу.

Зробити висновок з лабораторної роботи.

Таблиця 2.1 – Варіанти алгоритмів

№	Структура даних
1	Файли з щільним індексом з перебудовою індексної області, бінарний пошук
2	Файли з щільним індексом з областю переповнення, бінарний
3	Файли з не щільним індексом з перебудовою індексної області, бінарний пошук
4	Файли з не щільним індексом з областю переповнення, бінарний пошук
5	АВЛ-дерево
6	Червоно-чорне дерево

7	В-дерево $t=10$ , бінарний пошук
8	В-дерево $t=25$ , бінарний пошук
9	В-дерево $t=50$ , бінарний пошук
10	В-дерево $t=100$ , бінарний пошук
11	Файли з щільним індексом з перебудовою індексної області, однорідний бінарний пошук
12	Файли з щільним індексом з областю переповнення, однорідний бінарний пошук
13	Файли з не щільним індексом з перебудовою індексної області, однорідний бінарний пошук
14	Файли з не щільним індексом з областю переповнення, однорідний бінарний пошук
15	АВЛ-дерево
16	Червоно-чорне дерево
17	В-дерево $t=10$ , однорідний бінарний пошук
18	В-дерево $t=25$ , однорідний бінарний пошук
19	В-дерево $t=50$ , однорідний бінарний пошук
20	В-дерево $t=100$ , однорідний бінарний пошук
21	Файли з щільним індексом з перебудовою індексної області, метод Шарра
22	Файли з щільним індексом з областю переповнення, метод Шарра
23	Файли з не щільним індексом з перебудовою індексної області, метод Шарра
24	Файли з не щільним індексом з областю переповнення, метод Шарра
25	АВЛ-дерево
26	Червоно-чорне дерево
27	В-дерево $t=10$ , метод Шарра
28	В-дерево $t=25$ , метод Шарра
29	В-дерево $t=50$ , метод Шарра
30	В-дерево $t=100$ , метод Шарра

31	АВЛ-дерево
32	Червоно-чорне дерево
33	В-дерево $t=250$ , бінарний пошук
34	В-дерево $t=250$ , однорідний бінарний пошук
35	В-дерево $t=250$ , метод Шарра

### 3.1.Псевдокод алгоритмів

```

function left_height(self) {
    self.left.height or 0
}

function right_height(self) {
    self.right.height or 0
}

function update_height(self) {
    self.height = 1 + max(self.left_height(), self.right_height());
}

function balance_factor(self) {
    left_height := self.left_height();
    right_height := self.right_height();
    if left_height >= right_height {
        return left_height - right_height;
    } else {
        return -(right_height - left_height);
    }
}

function rotate_left(self) {
    if not right {
        return false;
    }
    right_node := self.right;
    right_left_tree := take from right_node.left;
    right_right_tree := take from right_node.right;

    new_left_tree := replace right with right_right_tree;
    swap self.data, new_left_tree.data;
    left_tree := take from self.left;

    new_left_node := new_left_tree;
    new_left_node.right = right_left_tree;
    new_left_node.left = left_tree;
    left = new_left_node;

    if node := self.left {
        node.update_height();
    }
    self.update_height();
    return true;
}

function rotate_right(self) {
    if not left {
        return false;
    }
    left_node := self.left;
    left_right_tree := take from left_node.right;
    left_left_tree := take from left_node.left;

    new_right_tree := replace left with left_left_tree;

```

```

    swap self.data, new_right_tree.data;
    right_tree := take from right;

    new_right_node := new_right_tree;
    new_right_node.left = left_right_tree;
    new_right_node.right = right_tree;
    right = new_right_tree;

    if node := self.right {
        node.update_height();
    }
    self.update_height();
    return true;
}

function rebalance(self) {
    match balance_factor() {
        -2 => {
            right_node := self.right;
            if right_node.balance_factor() == 1 {
                right_node.rotate_right();
            }
            self.rotate_left();
            return true;
        }
        2 => {
            left_node := self.left;
            if left_node.balance_factor() == -1 {
                left_node.rotate_left();
            }
            self.rotate_right();
            return true;
        }
        _ => return false,
    }
}

function insert(self, data) {
    current_tree = self.root;
    while current_node := current_tree {
        if (data < current_node.data) {
            current_tree = current_node.right;
        } else {
            current_tree = current_node.left;
        }
    }
    current_tree = Node{
        data,
        left: None,
        right: None,
        height: 1,
    };
    update height for all previous nodes;
    rebalance all previous nodes;
    return true;
}

function delete(self, value) {
    current_tree := self.root;
    target_value := None;

    while current_node := current_tree {

```



```

        if (value < current_node.value) {
            current_tree = current_node.right;
        } else if (value > current_node.value) {
            current_tree = current_node.right;
        } else {
            target_value = current_node;
            break;
        }
    }
    if not target_value {
        return None;
    }
    target_node := target_value;
    delete target_node and return the value

    update height for all previous nodes;
    rebalance all previous nodes;
    return deleted node value;
}

function search(self, key) {
    current_tree := self.root;
    while current_node := current_tree {
        if (value < current_node.value) {
            current_tree = current_node.right;
        } else if (value > current_node.value) {
            current_tree = current_node.right;
        } else {
            return true;
        }
    }
    return false;
}

```

### 3.2. Часова складність пошуку

Пошук в AVL дереві в гіршому та середньому випадках здійснюється за  $O(\log(n))$ , а в найкращому — за  $O(1)$ .

### 3.3. Програмна реалізація

#### 3.3.1. Вихідний код

##### **data.rs**

```

use serde::{Deserialize, Serialize};
use std::cmp::Ordering;

#[derive(Debug, Serialize, Deserialize, Clone)]
pub struct AvlNodeData {
    pub key: u32,
    pub value: String,
}

impl Ord for AvlNodeData {
    fn cmp(&self, other: &Self) -> Ordering {
        self.key.cmp(&other.key)
    }
}

```

```

impl PartialOrd for AvlNodeData {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.key.cmp(&other.key))
    }
}

impl PartialEq for AvlNodeData {
    fn eq(&self, other: &Self) -> bool {
        self.key == other.key
    }
}

impl Eq for AvlNodeData {}

```

### **tree.rs**

```

use serde::{Deserialize, Serialize};
use std::cmp::max;
use std::mem::{replace, swap};

use super::data::AvlNodeData;

#[derive(Debug, Serialize, Deserialize, Clone)]
pub struct AvlNode {
    pub data: AvlNodeData,
    pub left: AvlTree,
    pub right: AvlTree,
    pub height: usize,
}

pub type AvlTree = Option<Box<AvlNode>>;

impl<'a> AvlNode {
    pub fn left_height(&self) -> usize {
        self.left.as_ref().map_or(0, |left| left.height)
    }

    pub fn right_height(&self) -> usize {
        self.right.as_ref().map_or(0, |right| right.height)
    }

    pub fn update_height(&mut self) {
        self.height = 1 + max(self.left_height(),
self.right_height());
    }

    pub fn balance_factor(&self) -> i8 {
        let left_height = self.left_height();
        let right_height = self.right_height();

        if left_height >= right_height {
            (left_height - right_height) as i8
        } else {
            -((right_height - left_height) as i8)
        }
    }

    pub fn rotate_left(&mut self) -> bool {
        if self.right.is_none() {

```

```

        return false;
    }

    let right_node = self.right.as_mut().unwrap();
    let right_left_tree = right_node.left.take();
    let right_right_tree = right_node.right.take();

    let mut new_left_tree = replace(&mut self.right,
right_right_tree);
    swap(&mut self.data, &mut
new_left_tree.as_mut().unwrap().data);
    let left_tree = self.left.take();

    let new_left_node = new_left_tree.as_mut().unwrap();
    new_left_node.right = right_left_tree;
    new_left_node.left = left_tree;
    self.left = new_left_tree;

    if let Some(node) = self.left.as_mut() {
        node.update_height();
    }

    self.update_height();

    true
}

pub fn rotate_right(&mut self) -> bool {
    if self.left.is_none() {
        return false;
    }

    let left_node = self.left.as_mut().unwrap();
    let left_right_tree = left_node.right.take();
    let left_left_tree = left_node.left.take();

    let mut new_right_tree = replace(&mut self.left,
left_left_tree);
    swap(&mut self.data, &mut
new_right_tree.as_mut().unwrap().data);
    let right_tree = self.right.take();

    let new_right_node = new_right_tree.as_mut().unwrap();
    new_right_node.left = left_right_tree;
    new_right_node.right = right_tree;
    self.right = new_right_tree;

    if let Some(node) = self.right.as_mut() {
        node.update_height();
    }

    self.update_height();

    true
}

pub fn rebalance(&mut self) -> bool {
    match self.balance_factor() {
        -2 => {
            let right_node = self.right.as_mut().unwrap();

```

```

        if right_node.balance_factor() == 1 {
            right_node.rotate_right();
        }

        self.rotate_left();

        true
    }

    2 => {
        let left_node = self.left.as_mut().unwrap();

        if left_node.balance_factor() == -1 {
            left_node.rotate_left();
        }

        self.rotate_right();

        true
    }
    _ => false,
}
}
}
}

```

#### **set.rs**

```

use super::tree::*;
use serde::{Deserialize, Serialize};
use std::cmp::Ordering;
use std::mem::replace;

use super::data::AvlNodeData;

#[derive(Debug, Serialize, Deserialize, Clone)]
pub struct AvlTreeSet {
    root: AvlTree,
}

impl<'a> Default for AvlTreeSet {
    fn default() -> Self {
        Self { root: None }
    }
}

impl<'a> AvlTreeSet {
    pub fn new() -> Self {
        Self { root: None }
    }

    pub fn insert(&mut self, data: AvlNodeData) -> bool {
        let mut prev_ptrs = Vec::<*mut AvlNode>::new();
        let mut current_tree = &mut self.root;

        while let Some(current_node) = current_tree {
            prev_ptrs.push(&mut **current_node);

            match current_node.data.cmp(&data) {

```

```

        Ordering::Less => current_tree = &mut
current_node.right,
        Ordering::Equal | Ordering::Greater =>
current_tree = &mut current_node.left,
    }
}

*current_tree = Some(Box::new(AvlNode {
    data,
    left: None,
    right: None,
    height: 1,
}));

for node_ptr in prev_ptrs.into_iter().rev() {
    let node = unsafe { &mut *node_ptr };
    node.update_height();
    node.rebalance();
}

true
}

pub fn take(&mut self, key: &u32) -> Option<AvlNodeData> {
    let mut prev_ptrs = Vec:::<*mut AvlNode>::new();
    let mut current_tree = &mut self.root;
    let mut target_value = None;

    while let Some(current_node) = current_tree {
        match current_node.data.key.cmp(&key) {
            Ordering::Less => {
                prev_ptrs.push(&mut **current_node);
                current_tree = &mut current_node.right;
            }
            Ordering::Equal => {
                target_value = Some(&mut **current_node);
                break;
            }
            Ordering::Greater => {
                prev_ptrs.push(&mut **current_node);
                current_tree = &mut current_node.left;
            }
        }
    };

    if target_value.is_none() {
        return None;
    }

    let target_node = target_value.unwrap();

    let taken_value = if target_node.left.is_none() ||
target_node.right.is_none() {
        if let Some(left_node) = target_node.left.take() {
            replace(target_node, *left_node).data
        } else if let Some(right_node) =
target_node.right.take() {
            replace(target_node, *right_node).data
        } else if let Some(prev_ptr) = prev_ptrs.pop() {
            let prev_node = unsafe { &mut *prev_ptr };

```

```

        let inner_value = if let Some(left_node) =
prev_node.left.as_ref() {
            // using of impl partial_eq
            if left_node.data == target_node.data {
                prev_node.left.take().unwrap().data
            } else {
                prev_node.right.take().unwrap().data
            }
        } else {
            prev_node.right.take().unwrap().data
        };

        prev_node.update_height();
        prev_node.rebalance();

        inner_value
    } else {
        self.root.take().unwrap().data
    }
} else {
    let right_tree = &mut target_node.right;

    if right_tree.as_ref().unwrap().left.is_none() {
        let mut right_node = right_tree.take().unwrap();

        let inner_value = replace(&mut target_node.data,
right_node.data);
        _ = replace(&mut target_node.right,
right_node.right.take());

        target_node.update_height();
        target_node.rebalance();

        inner_value
    } else {
        let mut next_tree = right_tree;
        let mut inner_ptrs = Vec:::<*mut AvlNode>::new();

        while let Some(next_left_node) = next_tree {
            if next_left_node.left.is_some() {
                inner_ptrs.push(&mut **next_left_node);
            }
            next_tree = &mut next_left_node.left;
        }

        let parent_left_node = unsafe { &mut
*inner_ptrs.pop().unwrap() };
        let mut leftmost_node =
parent_left_node.left.take().unwrap();

        let inner_value = replace(&mut target_node.data,
leftmost_node.data);
        _ = replace(&mut parent_left_node.left,
leftmost_node.right.take());

        parent_left_node.update_height();
        parent_left_node.rebalance();

        for node_ptr in inner_ptrs.into_iter().rev() {

```

```

        let node = unsafe { &mut *node_ptr };
        node.update_height();
        node.rebalance();
    }

    target_node.update_height();
    target_node.rebalance();

    inner_value
}
};

for node_ptr in prev_ptrs.into_iter().rev() {
    let node = unsafe { &mut *node_ptr };
    node.update_height();
    node.rebalance();
}

Some(taken_value)
}

pub fn contains(&self, key: &u32) -> bool {
    let mut current_tree = &self.root;

    while let Some(current_node) = current_tree {
        match current_node.data.key.cmp(&key) {
            Ordering::Less => {
                current_tree = &current_node.right;
            }
            Ordering::Equal => {
                return true;
            }
            Ordering::Greater => {
                current_tree = &current_node.left;
            }
        }
    };

    false
}

pub fn get(&self, key: &u32) -> Option<&AvlNodeData> {
    let mut current_tree = &self.root;

    println!("Searching for {}...", key);
    let mut copms: u32 = 0;

    while let Some(current_node) = current_tree {
        copms += 1;

        match current_node.data.key.cmp(&key) {
            Ordering::Less => {
                current_tree = &current_node.right;
            }
            Ordering::Equal => {
                println!(" - Found with total comparisons:
{}.", copms);

                return Some(&current_node.data);
            }
            Ordering::Greater => {

```

```

        current_tree = &current_node.left;
    }
};

}

println!(" - Not found.");

None
}

pub fn modify(&mut self, key: &u32, value: &String) -> bool {
    let mut current_tree = &mut self.root;

    while let Some(current_node) = current_tree.as_mut() {
        match current_node.data.key.cmp(&key) {
            Ordering::Less => {
                current_tree = &mut current_node.right;
            }
            Ordering::Equal => {
                let cl = current_node.clone();
                let bx = Box::new(AvlNode {
                    data: AvlNodeData {
                        key: key.to_owned(),
                        value: value.to_owned(),
                    },
                    left: cl.left,
                    right: cl.right,
                    height: cl.height,
                });

                _ = replace(current_node, bx);

                return true;
            }
            Ordering::Greater => {
                current_tree = &mut current_node.left;
            }
        }
    }

    false
}

pub fn clear(&mut self) {
    self.root.take();
}

pub fn is_empty(&self) -> bool {
    self.root.is_none()
}

pub fn len(&self) -> usize {
    self.iter().count()
}

pub fn iter(&'a self) -> impl Iterator<Item = &'a
AvlNodeData> + 'a {
    self.node_iter().map(|node| &node.data)
}

```



```

fn node_iter(&'a self) -> impl Iterator<Item = &'a AvlNode> +
'a {
    AvlTreeSetNodeIter {
        prev_nodes: Vec::new(),
        current_tree: &self.root,
    }
}

#[derive(Debug)]
pub struct AvlTreeSetNodeIter<'a> {
    prev_nodes: Vec<&'a AvlNode>,
    current_tree: &'a AvlTree,
}

impl<'a> Iterator for AvlTreeSetNodeIter<'a> {
    type Item = &'a AvlNode;

    fn next(&mut self) -> Option<Self::Item> {
        loop {
            match *self.current_tree {
                None => match self.prev_nodes.pop() {
                    None => {
                        return None;
                    }

                    Some(ref prev_node) => {
                        self.current_tree = &prev_node.right;

                        return Some(prev_node);
                    }
                },

                Some(ref current_node) => {
                    if current_node.left.is_some() {
                        self.prev_nodes.push(&current_node);
                        self.current_tree = &current_node.left;

                        continue;
                    }

                    if current_node.right.is_some() {
                        self.current_tree = &current_node.right;

                        return Some(current_node);
                    }

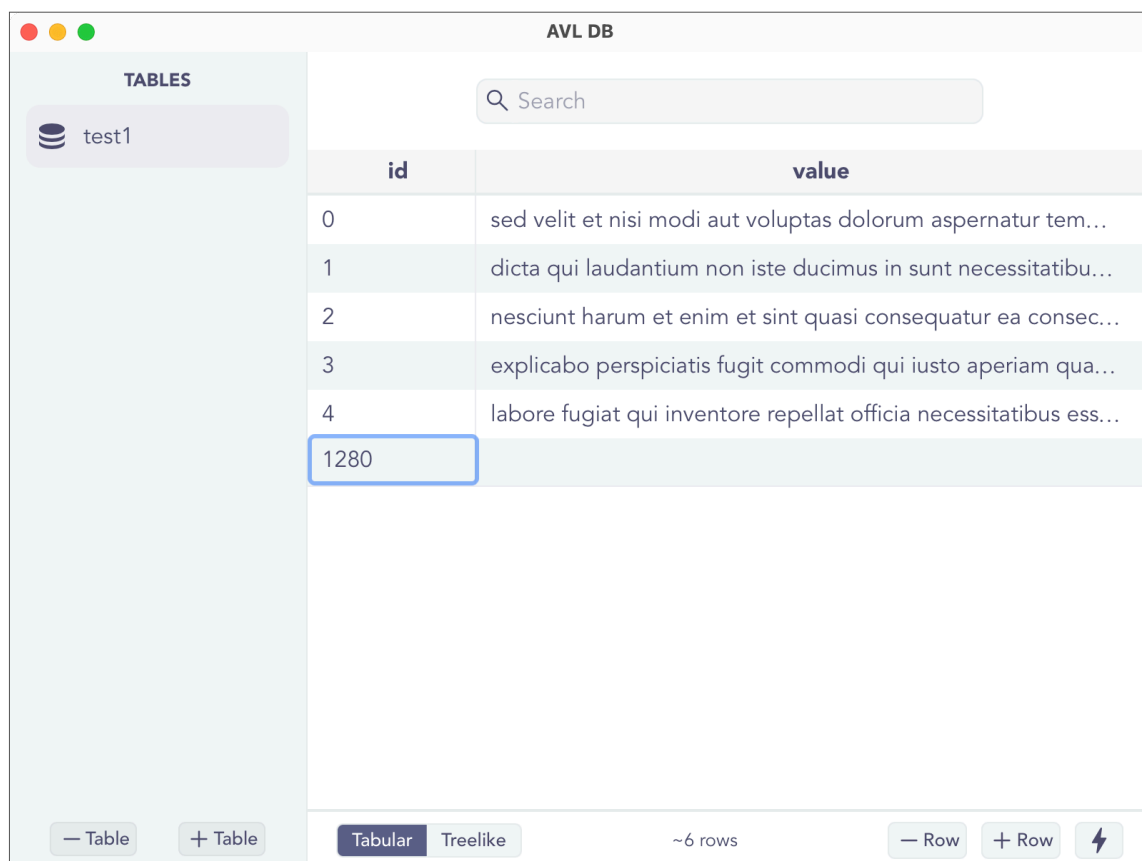
                    self.current_tree = &None;

                    return Some(current_node);
                }
            }
        }
    }
}

```

### 3.3.2. Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми для додавання і пошуку запису.



AVL DB

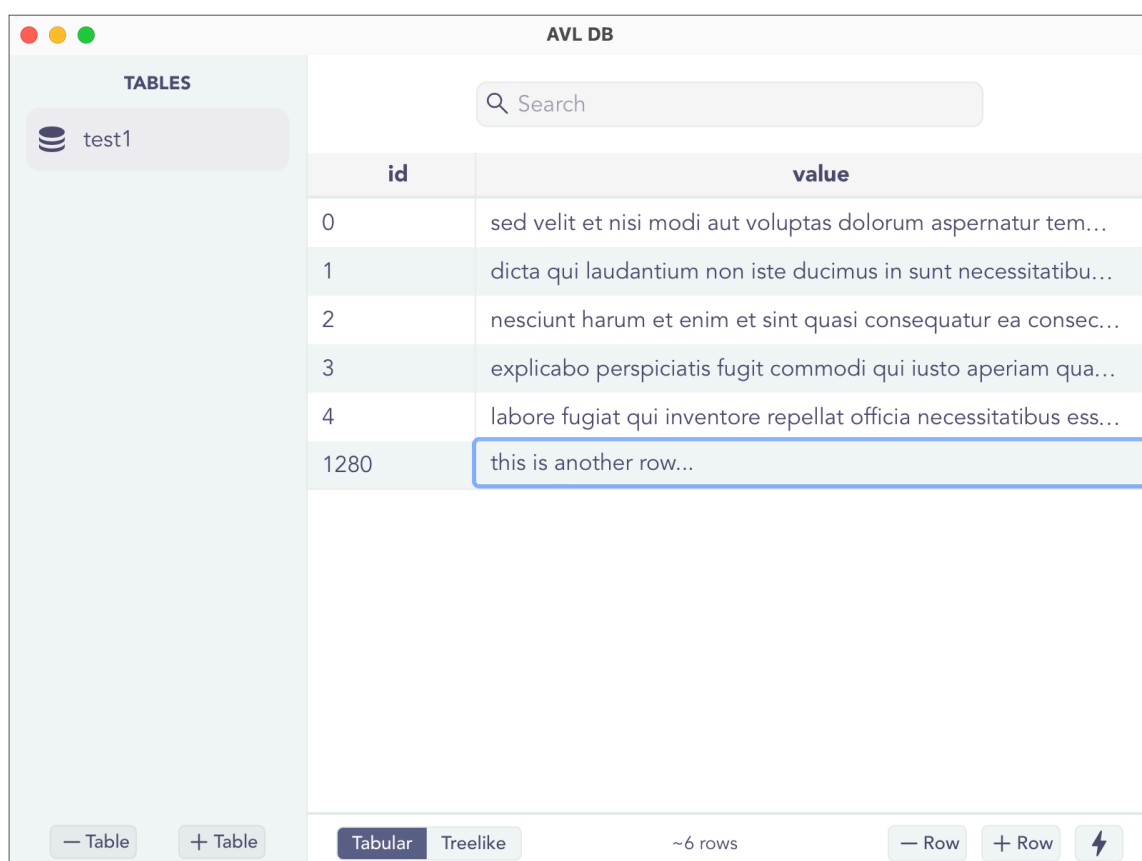
TABLES

test1

Search

id	value
0	sed velit et nisi modi aut voluptas dolorum aspernatur tem...
1	dicta qui laudantium non iste ducimus in sunt necessitatibu...
2	nesciunt harum et enim et sint quasi consequatur ea consec...
3	explicabo perspiciatis fugit commodi qui iusto aperiam qua...
4	labore fugiat qui inventore repellat officia necessitatibus ess...
1280	

— Table + Table Tabular Treelike ~6 rows — Row + Row ⚡



AVL DB

TABLES

test1

Search

id	value
0	sed velit et nisi modi aut voluptas dolorum aspernatur tem...
1	dicta qui laudantium non iste ducimus in sunt necessitatibu...
2	nesciunt harum et enim et sint quasi consequatur ea consec...
3	explicabo perspiciatis fugit commodi qui iusto aperiam qua...
4	labore fugiat qui inventore repellat officia necessitatibus ess...
1280	this is another row...

— Table + Table Tabular Treelike ~6 rows — Row + Row ⚡

Рис. 3.1 — Додавання запису

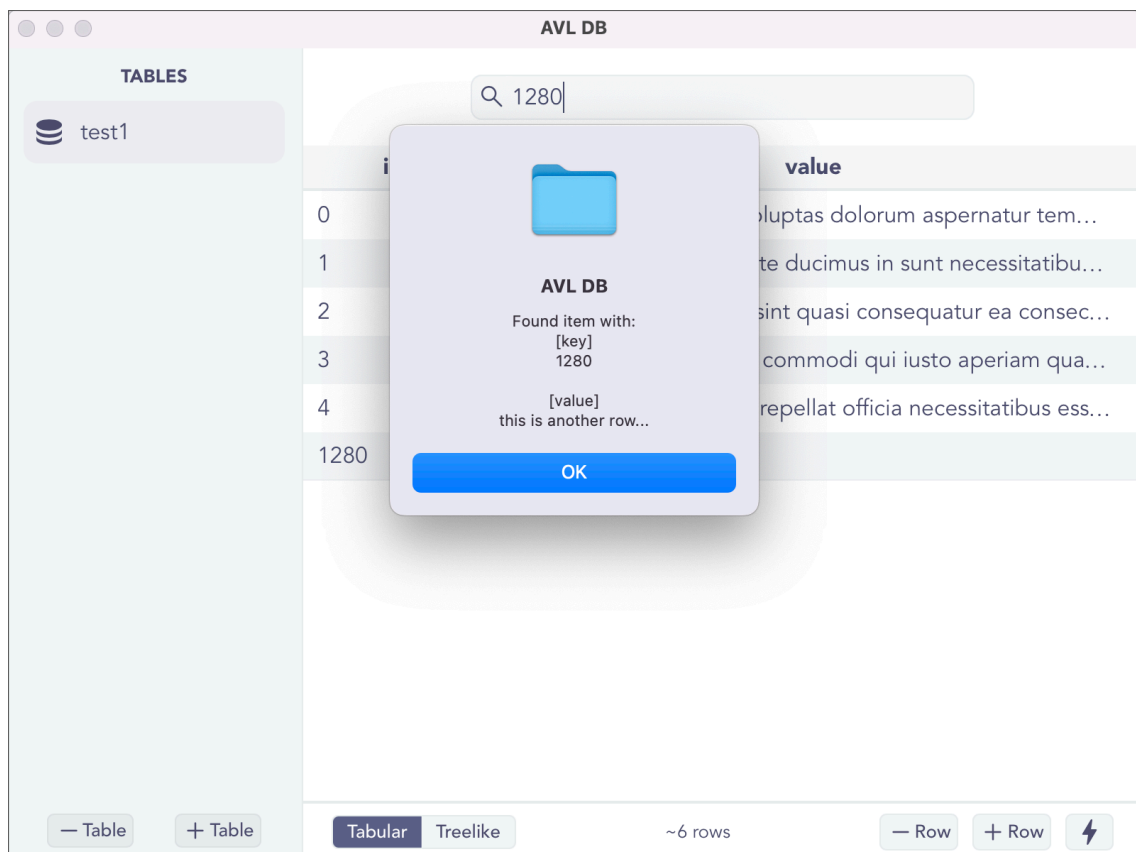


Рис. 3.2 — Поиск запису

### 3.4.Тестування алгоритму

#### 3.4.1.Часові характеристики оцінювання

В таблиці 3.1 наведено кількість порівнянь для 15 спроб пошуку запису по ключу для 10000 значень.

Таблиця 3.1 – Число порівнянь при спробі пошуку запису по ключу

Номер спроби пошуку	Число порівнянь
1	13
2	14
3	11
4	9
5	11
6	12
7	14
8	13
9	14
10	13
11	3
12	11
13	12
14	13
15	12

В середньому алгоритму необхідно:

$$\bar{x} = \frac{13 + 14 + 11 + 9 + 11 + 12 + 14 + 13 + 14 + 13 + 3 + 11 + 12 + 13 + 12}{15} = 11,6$$

порівнянь.

## ВИСНОВОК

В рамках лабораторної роботи було розроблено програму зі зручним повноцінним графічним інтерфейсом яка надає можливість керувати базами даних, збережених на ПЗП та використовує структуру AVL дерева для виконання операцій додавання, видалення, редагування та пошуку запису у базі по ключу. Після проведення серії експериментів з пошуку по ключу запису в базі даних, яка має 10000 елементів, згенерованих випадковим чином, отримано результат 11,6 порівнянь в середньому, що відповідає асимптотичній оцінці пошуку в AVL дереві —  $O(\log(n))$ :  $\log(10000) \approx 13,28$ .

## КРИТЕРІЇ ОЦІНЮВАННЯ

За умови здачі лабораторної роботи до 13.11.2022 включно максимальний бал дорівнює – 5. Після 13.11.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

- псевдокод алгоритму – 15%;
- аналіз часової складності – 5%;
- програмна реалізація алгоритму – 65%;
- тестування алгоритму – 10%;
- висновок – 5%.

+1 додатковий бал можна отримати за реалізацію графічного зображення структури ключів.