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

**Катедра інформатики та програмної інженерії**

# Звіт

з лабораторної роботи № 3 з дисципліни
«Проєктування алгоритмів»

**„ Проєктування структур даних”**

**Виконав(ла)** *ІП-11 Лесів В.І.* _____
(шифр, прізвище, ім'я, по батькові)

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

Київ 2022

ЗМІСТ

# 1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

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

# 2 ЗАВДАННЯ

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 пошук |
| 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 | B-дерево t=10, бінарний пошук |
| 8 | B-дерево t=25, бінарний пошук |
| 9 | B-дерево t=50, бінарний пошук |
| 10 | B-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 | B-дерево t=10, однорідний бінарний пошук |
| 18 | B-дерево t=25, однорідний бінарний пошук |
| 19 | B-дерево t=50, однорідний бінарний пошук |
| 20 | B-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 | B-дерево t=10, метод Шарра |
| 28 | B-дерево t=25, метод Шарра |

| | |
|---|---|
| 29 | B-дерево t=50, метод Шарра |
| 30 | B-дерево t=100, метод Шарра |
| 31 | АВЛ-дерево |
| 32 | Червоно-чорне дерево |
| 33 | B-дерево t=250, бінарний пошук |
| 34 | B-дерево t=250, однорідний бінарний пошук |
| 35 | B-дерево t=250, метод Шарра |

# 3 ВИКОНАННЯ

## 3.1 Псевдокод алгоритмів

Вставлення запису.

```
function insert(k : BTreeNode)
      if length(root.keys) = (2 * t) - 1 then
    begin
          temp -> BTreeNode()
          self.root -> temp
          temp.child.insert(0, root)
          self.splitChild(temp, 0)
          self.insertNonFull(temp, k)
      else
          self.insertNonFull(root, k)
    end if
 end function
```

Вставлення запису, якщо вершина не заповнена.

```
function insertNonFull(x : BTreeNode, k : BTreeNode)
      i -> length(x.keys) - 1
      if x.leaf = True then
          begin
          x.keys.append((None, None))
          while i >= 0 and k[0] < x.keys[i][0] do
              begin
              x.keys[i + 1] -> x.keys[i]
              i -> i - 1
              end while
          x.keys[i + 1] -> k
      else
          while i >= 0 and k[0] < x.keys[i][0] do
              i -> i - 1
          i -> i + 1
          if length(x.child[i].keys) = (2 * t) - 1 then
              self.splitChild(x, i)
              if k[0] > x.keys[i][0] then
                  i -> i + 1
          self.insertNonFull(x.child[i], k)
          end if
end function
```

Розділення нащадків після додавання запису

```
  function splitChild(x : BTreeNode, i : int)
```

```
        y -> x.child[i]
        z -> BTreeNode(y.leaf)
        x.child.insert(i + 1, z)
        x.keys.insert(i, y.keys[t - 1])
        z.keys -> y.keys[t: (2 * t) - 1]
        y.keys -> y.keys[0: t - 1]
        if not y.leaf then
            begin
            z.child -> y.child[t: 2 * t]
            y.child -> y.child[0: t - 1]
            end if
    end function
```

Видалення запису

```
    function delete(x : BTreeNode, k : BTreeNode)
        i -> 0
        while i < length(x.keys) and k[0] > x.keys[i][0] do
            i -> i + 1
        if x.leaf then begin
            if i < len(x.keys) and x.keys[i][0] = k[0] then
                x.keys.pop(i)
                return
            return
            end if
        if i < length(x.keys) and x.keys[i][0] = k[0] then
            return self.deleteInternalNode(x, k, i)
        elif length(x.child[i].keys) >= t then
            self.delete(x.child[i], k)
        else
            if i != 0 and i + 2 < length(x.child) then begin
                if length(x.child[i - 1].keys) >= t then
                    self.deleteSibling(x, i, i - 1)
                elif length(x.child[i + 1].keys) >= t then
                    self.deleteSibling(x, i, i + 1)
                else
                    self.deleteMerge(x, i, i + 1)
            elif i = 0 then
                if length(x.child[i + 1].keys) >= t then
                    self.deleteSibling(x, i, i + 1)
                else
                    self.deleteMerge(x, i, i + 1)
            elif i + 1 = length(x.child) then
                if length(x.child[i - 1].keys) >= t then
```

```
                self.deleteSibling(x, i, i - 1)
            else
                self.deleteMerge(x, i, i - 1)
        self.delete(x.child[i], k)
        end if
```

Видалення запису, коли лежить не в листку

```
function deleteInternalNode(x : BTreeNode, k : BTreeNode, I : int)
    if x.leaf then
        if x.keys[i][0] = k[0] then
            x.keys.pop(i)
            return
        return

    if length(x.child[i].keys) >= t then
        x.keys[i] -> self.deletePredecessor(x.child[i])
        return
    elif length(x.child[i + 1].keys) >= t then
        x.keys[i] -> self.deleteSuccessor(x.child[i + 1])
        return
    else
        self.deleteMerge(x, i, i + 1)
        self.deleteInternalNode(x.child[i], k, self.t - 1)
end function
```

Видалення заміщенням попередником.

```
function deletePredecessor(x : BTreeNode)
    if x.leaf then
        return x.pop()
    n -> length(x.keys) - 1
    if length(x.child[n].keys) >= self.t then
        self.deleteSibling(x, n + 1, n)
    else
        self.deleteMerge(x, n, n + 1)
    self.deletePredecessor(x.child[n])
end function
```

Видалення заміщенням наступником.

```
function deleteSuccessor(x : BTreeNode)
    if x.leaf then
        return x.keys.pop(0)
    if length(x.child[1].keys) >= self.t then
        self.deleteSibling(x, 0,1)
```

```
        else
            self.deleteMerge(x, 0, 1)
        self.deletePredecessor(x.child[0])
    end function
```

Видалення з об’єднанням дітей.

```
    function deleteMerge(x : BTreeNode, I : int, j : int):
        cnode -> x.child[i]
        if j > I then
            begin
            rsnode -> x.child[j]
            cnode.keys.append(x.keys[i])
            for k -> 0 to length(rsnode.keys) do
                cnode.keys.append(rsnode.keys[k])
                if length(rsnode.child) > 0 then
                    cnode.child.append(rsnode.child[k])
            if length(rsnode.child) > 0 then
                cnode.child.append(rsnode.child.pop())
            new -> cnode
            x.keys.pop(i)
            x.child.pop(j)
        else
            lsnode -> x.child[j]
            lsnode.keys.append(x.keys[j])
            for i -> 0 to length(cnode.keys) do
                lsnode.keys.append(cnode.keys[i])
                if length(lsnode.child) > 0 then
                    lsnode.child.append(cnode.child[i])
            if length(lsnode.child) > 0 then
                lsnode.child.append(cnode.child.pop())
            new -> lsnode
            x.keys.pop(j)
            x.child.pop(i)
        end if

        if x = self.root and length(x.keys) = 0 then
            self.root -> new
    end function
```

Видалення сусідньої на одному рівні.

```
    function deleteSibling(x : BTreeNode, I : int, j : int):
        cnode -> x.child[i]
        if i < j then
            begin
```

```
                rsnode -> x.child[j]
                cnode.keys.append(x.keys[i])
                x.keys[i] -> rsnode.keys[0]
                if length(rsnode.child) > 0 then
                    cnode.child.append(rsnode.child[0])
                    rsnode.child.pop(0)
                rsnode.keys.pop(0)
            else
                lsnode -> x.child[j]
                cnode.keys.insert(0, x.keys[i - 1])
                x.keys[i - 1] -> lsnode.keys.pop()
                if len(lsnode.child) > 0 then
                    cnode.child.insert(0, lsnode.child.pop())
            end if
        end function
```

Пошук і зміна записів.

```
        function searchEdit(k : Tuple, x : BTreeNode = None, ed : int = None):
            if x is not None then
                begin
                lst -> x.keys[::]
                d -> trunc(len(lst) / 2)
                i -> d
                if length(lst) % 2 = 0 then
                    lst.append((lst[-1][0] ^ 2 + 1, 0))
                while d != 0 do
                    begin
                    if lst[i][0] = k then
                        self.window.lab.setText("Знайдено:        ("        +
str(x.keys[i][0]) + ", " + str(x.keys[i][1]) + ")")
                        if ed is not None then
                            x.keys[i] -> (x.keys[i][0], ed)
                        return x, (x.keys[i])
                    elif lst[i][0] < k then
                        i -> i + (trunc(d / 2) + 1)
                    else
                        i -> I - (trunc(d / 2) + 1)
                    d -> trunc(d / 2)
                end while
                if i < length(lst) and lst[i][0] = k then
                    self.window.lab.setText("Знайдено:          ("          +
str(x.keys[i][0]) + ", " + str(x.keys[i][1]) + ")")
                    if ed is not None then
```

```
                    x.keys[i] -> (x.keys[i][0], ed)
                return x, (x.keys[i])
            elif x.leaf:
                self.window.lab.setText("Не знайдено:(")
                return None
            else:
                if lst[i][0] > k then
                    return self.searchEdit(k, x.child[i], ed)
                else:
                    return self.searchEdit(k, x.child[i + 1], ed)

        else:
            return self.searchEdit(k, self.root, ed)
        end if
    end function
```

## 3.2 Часова складність пошуку

Середня часова складність пошуку становить:

$$\theta(\log n)$$

Найкраща часова складність становить:

$$O(1)$$

## 3.3 Програмна реалізація

### 3.3.1 Вихідний код

Main.py

```python
from visualisation import *
import sys


def main():
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    app.exec()


if __name__ == '__main__':
    main()
```

scrollabel.py

```python
from PyQt6.QtWidgets import *


class ScrollLabel(QScrollArea):
    def __init__(self):
```

```
        QScrollArea.__init__(self)

        self.setWidgetResizable(True)

        content = QWidget(self)
        self.setWidget(content)

        lay = QVBoxLayout(content)

        self.label = QLabel(content)
        self.label.setWordWrap(True)
        lay.addWidget(self.label)
```

visualization.py

```
from PyQt6.QtCore import *
from scrolllabel import *
import bTree1


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Lab 3: B-tree")

        self.tree = bTree1.BTree(25, self)

        self.treeLabel = ScrollLabel()
        self.box = QComboBox()
        self.box.addItems(["Оберіть дію:", "Заповнити", "Вставити",
"Видалити", "Знайти/Редагувати"])
        self.conButton = QPushButton("Виконати")

        self.values = QLineEdit()
        self.values.setPlaceholderText("Введіть значення")

        self.conButton.setCheckable(True)

        self.lab = QLabel()

        self.conButton.clicked.connect(self.tree.start)

        outerLayout = QHBoxLayout()
        leftLayout = QVBoxLayout()

        leftLayout.addWidget(self.box)
        leftLayout.addWidget(self.values)
        leftLayout.addWidget(self.conButton)
        leftLayout.addWidget(self.lab)

        self.setFixedSize(QSize(1000, 700))
        self.lab.setFixedSize(500, 200)
        self.treeLabel.setFixedSize(QSize(500, 700))
        outerLayout.addLayout(leftLayout)
        outerLayout.addWidget(self.treeLabel)

        container = QWidget()
        container.setLayout(outerLayout)

        self.setCentralWidget(container)
```

bTree1.py

```
from math import trunc
```

```
class BTreeNode:
    def __init__(self, leaf=False):
        self.leaf = leaf
        self.keys = []
        self.child = []


class BTree:
    def __init__(self, t, visual):
        self.root = BTreeNode(True)
        self.t = t
        self.window = visual

    def start(self):
        enter = [int(i) for i in self.window.values.displayText().split()]
        ind = self.window.box.currentIndex()
        exis, y = [-1], -1
        if ind == 1:
                for i in range(enter[0]):
                        while y in exis:
                                y = randint(0, 10000)
                        exis.append(y)
                        self.insert((i,y))
        elif ind == 2:
                self.insert((enter[0], enter[1]))
        elif ind == 3:
                self.delete(self.root, (enter[0], ""))
        elif ind == 4:
                if len(enter) == 1:
                        self.searchEdit(enter[0])
        else:
            self.searchEdit(enter[0],None, enter[1])
        self.window.treeLabel.label.setText("")
        self.printTree(self.root)

    def insert(self, k):
        root = self.root
        if len(root.keys) == (2 * self.t) - 1:
            temp = BTreeNode()
            self.root = temp
            temp.child.insert(0, root)
            self.splitChild(temp, 0)
            self.insertNonFull(temp, k)
        else:
            self.insertNonFull(root, k)

    def insertNonFull(self, x, k):
        i = len(x.keys) - 1
        if x.leaf:
            x.keys.append((None, None))
            while i >= 0 and k[0] < x.keys[i][0]:
                x.keys[i + 1] = x.keys[i]
                i -= 1
            x.keys[i + 1] = k
        else:
            while i >= 0 and k[0] < x.keys[i][0]:
                i -= 1
            i += 1
            if len(x.child[i].keys) == (2 * self.t) - 1:
                self.splitChild(x, i)
                if k[0] > x.keys[i][0]:
                    i += 1
            self.insertNonFull(x.child[i], k)
```

```
    def splitChild(self, x, i):
        t = self.t
        y = x.child[i]
        z = BTreeNode(y.leaf)
        x.child.insert(i + 1, z)
        x.keys.insert(i, y.keys[t - 1])
        z.keys = y.keys[t: (2 * t) - 1]
        y.keys = y.keys[0: t - 1]
        if not y.leaf:
            z.child = y.child[t: 2 * t]
            y.child = y.child[0: t - 1]

    def delete(self, x, k):
        t = self.t
        i = 0
        while i < len(x.keys) and k[0] > x.keys[i][0]:
            i += 1
        if x.leaf:
            if i < len(x.keys) and x.keys[i][0] == k[0]:
                x.keys.pop(i)
                return
            return

        if i < len(x.keys) and x.keys[i][0] == k[0]:
            return self.deleteInternalNode(x, k, i)
        elif len(x.child[i].keys) >= t:
            self.delete(x.child[i], k)
        else:
            if i != 0 and i + 2 < len(x.child):
                if len(x.child[i - 1].keys) >= t:
                    self.deleteSibling(x, i, i - 1)
                elif len(x.child[i + 1].keys) >= t:
                    self.deleteSibling(x, i, i + 1)
                else:
                    self.deleteMerge(x, i, i + 1)
            elif i == 0:
                if len(x.child[i + 1].keys) >= t:
                    self.deleteSibling(x, i, i + 1)
                else:
                    self.deleteMerge(x, i, i + 1)
            elif i + 1 == len(x.child):
                if len(x.child[i - 1].keys) >= t:
                    self.deleteSibling(x, i, i - 1)
                else:
                    self.deleteMerge(x, i, i - 1)
            self.delete(x.child[i], k)

    def deleteInternalNode(self, x, k, i):
        t = self.t
        if x.leaf:
            if x.keys[i][0] == k[0]:
                x.keys.pop(i)
                return
            return

        if len(x.child[i].keys) >= t:
            x.keys[i] = self.deletePredecessor(x.child[i])
            return
        elif len(x.child[i + 1].keys) >= t:
            x.keys[i] = self.deleteSuccessor(x.child[i + 1])
            return
        else:
            self.deleteMerge(x, i, i + 1)
            self.deleteInternalNode(x.child[i], k, self.t - 1)
```

```
    def deletePredecessor(self, x):
        if x.leaf:
            return x.pop()
        n = len(x.keys) - 1
        if len(x.child[n].keys) >= self.t:
            self.deleteSibling(x, n + 1, n)
        else:
            self.deleteMerge(x, n, n + 1)
        self.deletePredecessor(x.child[n])

    def deleteSuccessor(self, x):
        if x.leaf:
            return x.keys.pop(0)
        if len(x.child[1].keys) >= self.t:
            self.deleteSibling(x, 0, 1)
        else:
            self.deleteMerge(x, 0, 1)
        self.deleteSuccessor(x.child[0])

    def deleteMerge(self, x, i, j):
        cnode = x.child[i]

        if j > i:
            rsnode = x.child[j]
            cnode.keys.append(x.keys[i])
            for k in range(len(rsnode.keys)):
                cnode.keys.append(rsnode.keys[k])
                if len(rsnode.child) > 0:
                    cnode.child.append(rsnode.child[k])
            if len(rsnode.child) > 0:
                cnode.child.append(rsnode.child.pop())
            new = cnode
            x.keys.pop(i)
            x.child.pop(j)
        else:
            lsnode = x.child[j]
            lsnode.keys.append(x.keys[j])
            for i in range(len(cnode.keys)):
                lsnode.keys.append(cnode.keys[i])
                if len(lsnode.child) > 0:
                    lsnode.child.append(cnode.child[i])
            if len(lsnode.child) > 0:
                lsnode.child.append(cnode.child.pop())
            new = lsnode
            x.keys.pop(j)
            x.child.pop(i)

        if x == self.root and len(x.keys) == 0:
            self.root = new

    def deleteSibling(self, x, i, j):
        cnode = x.child[i]
        if i < j:
            rsnode = x.child[j]
            cnode.keys.append(x.keys[i])
            x.keys[i] = rsnode.keys[0]
            if len(rsnode.child) > 0:
                cnode.child.append(rsnode.child[0])
                rsnode.child.pop(0)
            rsnode.keys.pop(0)
        else:
            lsnode = x.child[j]
            cnode.keys.insert(0, x.keys[i - 1])
```

```
            x.keys[i - 1] = lsnode.keys.pop()
            if len(lsnode.child) > 0:
                cnode.child.insert(0, lsnode.child.pop())

    def printTree(self, x, l=0):

self.window.treeLabel.label.setText(self.window.treeLabel.label.text()+"Level
"+str(l)+" "+str(len(x.keys))+": ")
        for i in x.keys:

self.window.treeLabel.label.setText(self.window.treeLabel.label.text()+"("+st
r(i[0])+", "+str(i[1])+") ")

self.window.treeLabel.label.setText(self.window.treeLabel.label.text()+"\n")
        l += 1
        if len(x.child) > 0:
            for i in x.child:
                self.printTree(i, l)

    def searchEdit(self, k, x=None, ed=None):
        if x is not None:
            lst = x.keys[::]
            d = trunc(len(lst) / 2)
            i = d
            if len(lst) % 2 == 0:
                lst.append((lst[-1][0] ** 2 + 1, 0))
            while d != 0:
                if lst[i][0] == k:
                    self.window.lab.setText("Знайдено: (" + str(x.keys[i][0])
+ ", " + str(x.keys[i][1]) + ")")
                    if ed is not None:
                        x.keys[i] = (x.keys[i][0], ed)
                    return x, (x.keys[i])
                elif lst[i][0] < k:
                    i += (trunc(d / 2) + 1)
                else:
                    i -= (trunc(d / 2) + 1)
                d = trunc(d / 2)
            if i < len(lst) and lst[i][0] == k:
                self.window.lab.setText("Знайдено: (" + str(x.keys[i][0]) +
", " + str(x.keys[i][1]) + ")")
                if ed is not None:
                    x.keys[i] = (x.keys[i][0], ed)
                return x, (x.keys[i])
            elif x.leaf:
                self.window.lab.setText("Не знайдено:(")
                return None
            else:
                if lst[i][0] > k:
                    return self.searchEdit(k, x.child[i], ed)
                else:
                    return self.searchEdit(k, x.child[i + 1], ed)

        else:
            return self.searchEdit(k, self.root, ed)
```

### 3.3.2 Приклади роботи

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

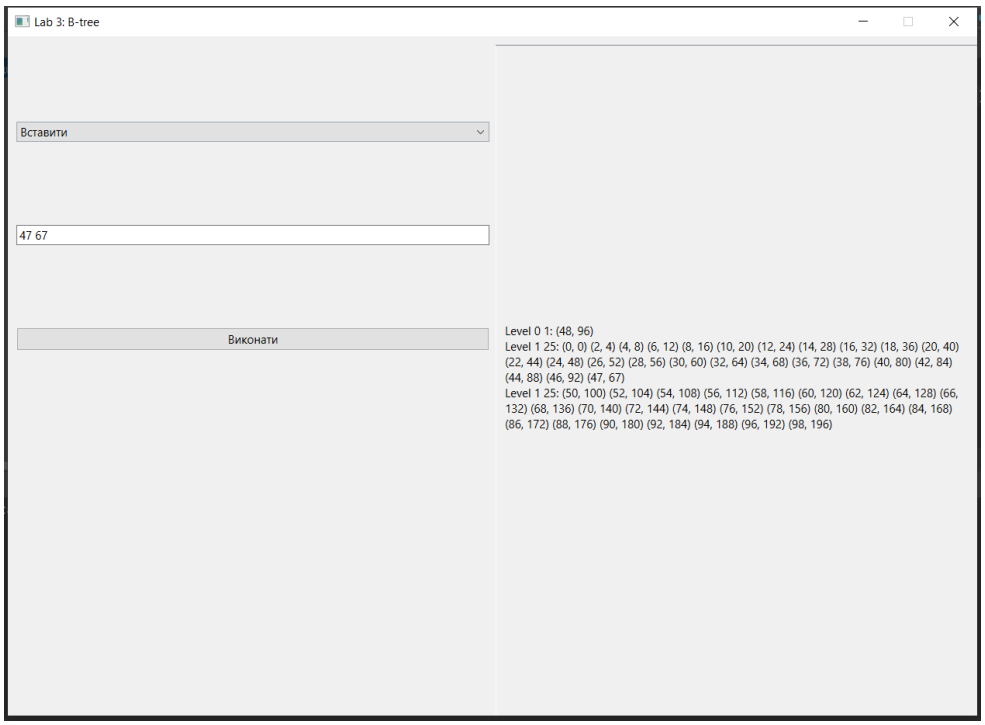


Рисунок 3.1 –Додавання запису

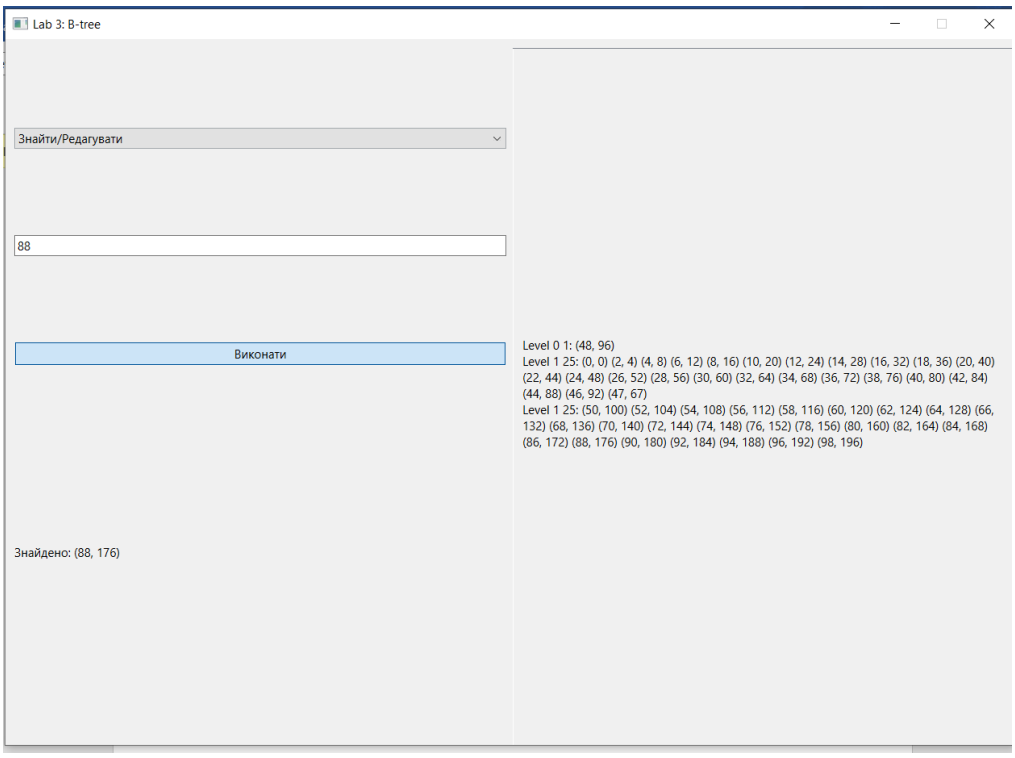


Рисунок 3.2 – Пошук запису

## 3.4 Тестування алгоритму

### 3.4.1 Часові характеристики оцінювання

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

Таблиця 3.1 – Число порівнянь при спробі пошуку запису по ключу

| Номер спроби пошуку | Число порівнянь |
|---|---|
| 1 | 17 |
| 2 | 17 |
| 3 | 17 |
| 4 | 16 |
| 5 | 18 |
| 6 | 18 |
| 7 | 16 |
| 8 | 17 |
| 9 | 17 |
| 10 | 16 |
| 11 | 19 |
| 12 | 17 |
| 13 | 17 |
| 14 | 18 |
| 15 | 17 |
| Середня | 17 |

# ВИСНОВОК

В рамках лабораторної роботи було вивчено основні підходи проєктування та обробки складних структур даних, реалізовано програмне забезпечення, що дає здійснювати змогу здійснювати додавання, видалення, пошук та зміну даних, в основі яких лежить B-дерево з t=25 та однорідний бінарний пошук. У ході роботи було досліджено алгоритм пошуку і ми дійшли висновку, що для того, щоб знайти запис нам потрібно не більше, ніж log n. Середня кількість порівнянь при пошуку становить 17.

# 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.