**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)**

(шифр, прізвище, ім'я, по батькові)

*ІП-12 Орищенко Ярослав*

**Перевірив**

(прізвище, ім'я, по батькові)

*Сопов О.О.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc114359761)

[2 Завдання 4](#_Toc114359762)

[3 Виконання 7](#_Toc114359763)

[3.1 Псевдокод алгоритмів 7](#_Toc114359764)

[3.2 Часова складність пошуку 12](#_Toc114359765)

[3.3 Програмна реалізація 13](#_Toc114359766)

[3.3.1 Вихідний код 13](#_Toc114359767)

[3.3.2 Приклади роботи 23](#_Toc114359768)

[3.4 Тестування алгоритму 24](#_Toc114359769)

[3.4.1 Часові характеристики оцінювання 24](#_Toc114359770)

[Висновок 25](#_Toc114359771)

[Критерії оцінювання 26](#_Toc114359772)

# Мета лабораторної роботи

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

# 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 |

# Виконання

## Псевдокод алгоритмів

**function** uniform\_binary\_search(key, node, parent=None)

i = node.keys.length() / 2

delta = node.keys.length() / 2

**while** delta != 0 **do**:

delta = delta / 2

**if** i >= node.keys.length() or node.keys[i] > key **do**:

i = i – delta – 1

**else if** node.keys[i] < key **do**:

i = i + delta + 1

**else do**:

return i, node, parent

**if** node.keys[i] == key **do**:

return i, node, parent

**if** node.is\_leaf **do**:

return None

**else do**:

**if** node.keys[i] > key **do**:

return uniform\_binary\_search(key,node.children[i], node.parent)

**else do**:

return uniform\_binary\_search(key, node.children[i+1], node.parent)

**function** insert(key)

**if** root.keys.length() == max\_length **do**:

temp = BtreeNode()

temp.children.insert(0, root)

split\_child(temp, 0)

insert\_non\_full(temp, key)

**else do**:

insert\_non\_full(root, key)

**function** insert\_non\_full(node, key)

i = node.keys.length() – 1

**while** i >= 0 and key < node.keys[i] **do**:

i = i – 1

i = i + 1

**if** node.is\_leaf **do**:

node.keys.insert(i, key)

**else do**:

**if** node.children[i].length == self.max\_length **do**:

split\_child(node, i)

**if** key > node.keys[i] **do**:

i = i + 1

insert\_non\_full(node.children[i], key)

**function** split\_child(node, i)

t = Btree.t

y = node.children[i]

z = BtreeNode()

node.children.insert(i+1, z)

node.keys.insert(i, y.keys[t-1])

**if** y.is\_leaf == False **do**:

z.children = y.children[t:]

y.children = z.children[:t]

**function** replace(key, value)

index, node, parent = uniform\_bin\_search(key, root)

**if** node != None **do**:

node.keys.pop(index)

node.keys.insert(index, (key, value))

**function** delete\_key(key)

i, node, parent = uniform\_bin\_search(key, root)

**if** node == None **do**:

**return** False

**if** node.is\_leaf == True **do**:

**if** node == root or node.keys.length > min\_length **do**:

node.keys.pop(i)

**else do**:

pop\_element(key)

empty\_deletion(parent, node)

**else do**:

**if** node.keys.length > min\_length **do**:

**if** node.children[i].keys.length > min\_length **do**:

delete\_predecessor(node, i)

**else** if node.children[i+1].keys.length > min\_length **do**:

delete\_successor(node, i)

**else do**:

merge(node, i, i+1)

**else do**:

pop\_element(key)

empty\_deletion(parent, node)

**return** True

**function** pop\_element(key)

i, node, parent = uniform\_bin\_search(key, root)

node.keys.pop(i)

**function** empty\_delete(parent, node)

index = parent.children.index(node)

**if** index > 0 and parent.children[i-1].keys.length > min\_length **do**:

swap\_sibling(parent, index, index-1)

**else** **if** index < parent.children.length-1 and parent.children[i-1].keys.length > min\_length **do**:

swap\_sibling(parent, index, index+1)

**else do**:

**if** index > 0 **do**:

merge(parent, index, index-1)

**else do**:

merge(parent, index, index-1)

**if** node == root **do**:

return

**while** parent.keys.length < min\_length and parent != root **do**:

i, node, parent = uniform\_bin\_search(parent.keys[0], root)

empty\_delete(parent, node)

**function** swap\_sibling(node, i, j)

parent = node

node = parent.children[i]

sibling = parent.children[j]

**if** i > j **do**:

key = sibling.keys.pop()

node.keys.insert(0, parent.keys.pop(i-1)

parent.keys.insert(i-1, key)

**if** node.is\_leaf == False **do**:

parent.children[j].children[-2].keys += parent.children[j].children[-1].keys

parent.children.children.pop()

node.children[0].keys.insert(0, node.keys.pop(0))

node.keys.insert(0, node.children.keys.pop())

**else do**:

key = sibling.keys.pop(0)

node.keys.append(parent.keys.pop(i))

parent.keys.insert(i, key)

**if** node.is\_leaf == False **do**:

parent.children[j].children[0].keys += parent.children[j].children[1].keys

parent.children[j].children.pop(1)

node.children[-1].keys.append(node.keys.pop())

node.keys.append(node.children[-1].keys.pop(0))

**function** delete\_predecessor(node, i)

parent = node

node = parent.children[i]

parent.keys.pop(i)

parent.keys.insert(i, node.keys.pop())

**function** delete\_successor(node, i)

parent = node

node = parent.children[i+1]

parent.keys.pop(i)

parent.keys.insert(i, node.keys.pop(0))

**function** merge(node, i, j)

parent = node

child = parent.children[i]

sibling = parent.children[j]

**if** i > j **do**:

sibling.keys += parent.keys.pop(j) + child.keys

**else do**:

child.keys += parent.keys.pop(i) + sibling.keys

**if** not child.is\_leaf **do**:

parent.children[i].children += parent.children[j].children

**if** i > j **do**:

parent.children.pop(i)

**else**:

parent.children.pop(j)

**if** parent == self.root **and** len(parent.keys) == 0 **do**:

**if** i > j **do**:

root = sibling

**else do**:

root = child

root.is\_leaf = True

## Часова складність пошуку

Часова складність однорідного бінарного пошуку становить O(log n). Висота B-дерева не перевищує , де t – степінь дерева, n – кількість елементів. У найгіршому випадку, шуканий ключ буде знаходитись у листі, тобто потрібно буде пройти вершин, та для кожної виконати однорідний бінарний пошук. Звідси, складність пошуку в B-дереві однорідним бінарним пошуком становить .

## Програмна реалізація

### Вихідний код

**main.py**

from program import BTreeProgram

if \_\_name\_\_ == "\_\_main\_\_":

program = BTreeProgram(50)

**b\_tree\_node.py**

class BTreeNode:

def \_\_init\_\_(self, is\_leaf=False):

self.keys = []

self.children = []

self.is\_leaf = is\_leaf

**b\_tree.py**

from b\_tree\_node import BTreeNode

class BTree:

def \_\_init\_\_(self, t):

self.root = BTreeNode(True)

self.t = t

self.print\_text = []

self.max\_length = 2 \* t - 1

self.min\_length = t - 1

self.compares = 0

def insert(self, key):

root = self.root

if len(root.keys) == self.max\_length:

temp = BTreeNode()

self.root = temp

temp.children.insert(0, root)

self.split\_child(temp, 0)

self.insert\_non\_full(temp, key)

else:

self.insert\_non\_full(root, key)

def insert\_non\_full(self, node, key):

i = len(node.keys) - 1

while i >= 0 and key[0] < node.keys[i][0]:

i -= 1

i += 1

if node.is\_leaf:

node.keys.insert(i, key)

else:

if len(node.children[i].keys) == self.max\_length:

self.split\_child(node, i)

if key[0] > node.keys[i][0]:

i += 1

self.insert\_non\_full(node.children[i], key)

def split\_child(self, node, i):

t = self.t

y = node.children[i]

z = BTreeNode(y.is\_leaf)

node.children.insert(i + 1, z)

node.keys.insert(i, y.keys[t-1])

z.keys = y.keys[t:]

y.keys = y.keys[:t-1]

if not y.is\_leaf:

z.children = y.children[t:]

y.children = y.children[:t]

def print\_tree(self, node, level=0, first\_run=True):

if first\_run:

self.print\_text = []

if len(self.print\_text) <= level:

self.print\_text.append("")

for key in node.keys:

self.print\_text[level] += f"{key[0]}|"

self.print\_text[level] = self.print\_text[level][:-1]

self.print\_text[level] += " "

if len(node.children) > 0:

for child in node.children:

self.print\_tree(child, level+1, False)

self.print\_text[level+1] += "\t"

self.print\_text[level] += "\t\t"

return self.print\_text

def uniform\_bin\_search(self, key, node: BTreeNode, parent=None):

i = len(node.keys) // 2

delta = len(node.keys) // 2

while delta != 0:

self.compares += 1

delta = delta // 2

if i >= len(node.keys) or node.keys[i][0] > key:

i -= (delta + 1)

elif node.keys[i][0] < key:

i += delta + 1

else:

return i, node, parent

if i == len(node.keys):

i -= 1

if i < 0:

i = 0

if node.keys[i][0] == key:

return i, node, parent

if node.is\_leaf:

return None, None, None

else:

if node.keys[i][0] > key:

return self.uniform\_bin\_search(key, node.children[i], parent=node)

else:

return self.uniform\_bin\_search(key, node.children[i+1], parent=node)

def replace(self, key, value):

index, node, parent = self.uniform\_bin\_search(key, self.root)

if node is not None:

node.keys.pop(index)

node.keys.insert(index, (key, value))

def console\_print(self, x, level=0):

print("Level ", level, " ", len(x.keys), end=":")

for i in x.keys:

print(i, end=" ")

print()

level += 1

if len(x.children) > 0:

for i in x.children:

self.console\_print(i, level)

def delete\_key(self, key):

i, node, parent = self.uniform\_bin\_search(key, self.root)

if not node:

return False

if node.is\_leaf:

# if root is the only node in tree - delete key

# or node is a leaf with enough elements

if node == self.root or len(node.keys) > self.min\_length:

node.keys.pop(i) # simple deletion from the leaf

else:

self.pop\_element(key)

self.empty\_delete(parent, node) # deal with siblings if node has minimum of keys

else:

if len(node.keys) > self.min\_length:

if len(node.children[i].keys) > self.min\_length:

self.delete\_predecessor(node, i)

elif len(node.children[i+1].keys) > self.min\_length:

self.delete\_successor(node, i)

else:

self.merge(node, i, i+1) # merge left and right children if both have minimum of keys

else:

self.pop\_element(key)

self.empty\_delete(parent, node) # deal with siblings if node has minimum of keys

return True

def empty\_delete(self, parent, node):

index = parent.children.index(node)

if index > 0 and len(parent.children[index - 1].keys) > self.min\_length:

self.swap\_sibling(parent, index, index-1) # swapping with left sibling

elif index < len(parent.children) - 1 and len(parent.children[index + 1].keys) > self.min\_length:

self.swap\_sibling(parent, index, index+1) # swapping with right sibling

else:

if index > 0:

self.merge(parent, index, index-1) # merging with left sibling

else:

self.merge(parent, index, index+1) # merging with right sibling

if node == self.root:

return

# make sure the structure of B-Tree is not violated

while len(parent.keys) < self.min\_length and parent != self.root:

i, node, parent = self.uniform\_bin\_search(parent.keys[0][0], self.root)

self.empty\_delete(parent, node)

def pop\_element(self, key):

index, node, parent = self.uniform\_bin\_search(key, self.root)

node.keys.pop(index)

@staticmethod

def swap\_sibling(node, i, j):

parent = node

node = parent.children[i]

sibling = parent.children[j]

if i > j:

key = sibling.keys.pop()

node.keys.insert(0, parent.keys.pop(i - 1))

parent.keys.insert(i - 1, key)

if not node.is\_leaf:

parent.children[j].children[-2].keys += parent.children[j].children[-1].keys

parent.children[j].children.pop()

node.children[0].keys.insert(0, node.keys.pop(0))

node.keys.insert(0, node.children[0].keys.pop())

else:

key = sibling.keys.pop(0)

node.keys.append(parent.keys.pop(i))

parent.keys.insert(i, key)

if not node.is\_leaf:

parent.children[j].children[0].keys += parent.children[j].children[1].keys

parent.children[j].children.pop(1)

node.children[-1].keys.append(node.keys.pop())

node.keys.append(node.children[-1].keys.pop(0))

@staticmethod

def delete\_predecessor(node, i):

parent = node

node = parent.children[i]

parent.keys.pop(i)

parent.keys.insert(i, node.keys.pop())

@staticmethod

def delete\_successor(node, i):

parent = node

node = parent.children[i+1]

parent.keys.pop(i)

parent.keys.insert(i, node.keys.pop(0))

def merge(self, node, i, j):

parent = node

child = parent.children[i]

sibling = parent.children[j]

if i > j:

sibling.keys += [parent.keys.pop(j)] + child.keys

else:

child.keys += [parent.keys.pop(i)] + sibling.keys

if not child.is\_leaf:

parent.children[i].children += parent.children[j].children

if i > j:

parent.children.pop(i)

else:

parent.children.pop(j)

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

self.root = sibling if i > j else child

self.root.is\_leaf = True

**program.py**

from b\_tree import BTree

from random import randint

import customtkinter

from customtkinter import \*

from tkinter import messagebox

from tkinter import \*

customtkinter.set\_appearance\_mode("dark")

customtkinter.set\_default\_color\_theme("dark-blue")

class BTreeProgram:

def \_\_init\_\_(self, t):

self.tree = BTree(t)

self.elements\_amount = 0

self.window = CTk()

self.window.title("B-Tree")

self.window.geometry("1030x600")

self.commands = CTkFrame(self.window)

self.commands.grid(row=0, column=0, sticky=(N, W))

self.visual\_tree = []

self.autofill\_button = CTkButton(self.commands, text="Auto Fill", width=240, command=self.autofill)

self.autofill\_button.grid(row=0, column=0, padx=10, pady=5)

self.elem\_quantity = CTkEntry(self.commands, placeholder\_text="Quantity", width=140)

self.elem\_quantity.grid(row=0, column=1)

self.insert\_button = CTkButton(self.commands, text="Insert Data", width=240, command=self.insert)

self.insert\_button.grid(row=1, column=0, padx=10, pady=5)

self.insertion\_key = CTkEntry(self.commands, placeholder\_text="Key", width=140)

self.insertion\_key.grid(row=1, column=1)

self.insertion\_data = CTkEntry(self.commands, placeholder\_text="Data", width=140)

self.insertion\_data.grid(row=1, column=2)

self.search\_button = CTkButton(self.commands, text="Search", width=240, command=self.search)

self.search\_button.grid(row=2, column=0, padx=10, pady=5)

self.search\_key = CTkEntry(self.commands, placeholder\_text="Key", width=140)

self.search\_key.grid(row=2, column=1)

self.edit\_button = CTkButton(self.commands, text="Edit element", width=240, command=self.replace)

self.edit\_button.grid(row=3, column=0, padx=10, pady=5)

self.edit\_key = CTkEntry(self.commands, placeholder\_text="Key", width=140)

self.edit\_key.grid(row=3, column=1, pady=5)

self.edit\_value = CTkEntry(self.commands, placeholder\_text="Data", width=140)

self.edit\_value.grid(row=3, column=2, pady=5, padx=10)

self.delete\_button = CTkButton(self.commands, text="Delete element", width=240, command=self.delete)

self.delete\_button.grid(row=4, column=0, padx=10, pady=5)

self.delete\_value = CTkEntry(self.commands, placeholder\_text="Key", width=140)

self.delete\_value.grid(row=4, column=1)

self.print\_button = CTkButton(self.commands, text="Print Tree", width=240, command=self.pprint)

self.print\_button.grid(row=6, column=0, padx=10, pady=5)

self.window.mainloop()

def autofill(self):

try:

amount = int(self.elem\_quantity.get())

except ValueError:

messagebox.showerror(title="Error", message="Fill the fields correctly")

return

if amount != 0:

for i in range(amount):

self.tree.insert((self.elements\_amount, randint(0, 1000)))

self.elements\_amount += 1

self.pprint()

def insert(self):

key = int(self.insertion\_key.get())

data = self.insertion\_data.get()

if data != "":

self.tree.insert((key, data))

self.pprint()

def search(self):

try:

key = int(self.search\_key.get())

except ValueError:

messagebox.showerror(title="Error", message="Fill the fields correctly")

return

else:

i, node, parent = self.tree.uniform\_bin\_search(key, self.tree.root)

if node:

print(f"node: {[elem[0] for elem in node.keys]}\n"

f"index: {i}\n"

f"key: {key}\n"

f"data: {node.keys[i][1]}\n")

else:

messagebox.showerror(title="Error", message="Key does not exist")

def replace(self):

try:

key = int(self.edit\_key.get())

value = self.edit\_value.get()

except ValueError:

messagebox.showerror(title="Error", message="Fill the fields correctly")

return

else:

self.tree.replace(key, value)

self.pprint()

def delete(self):

try:

key = int(self.delete\_value.get())

except ValueError:

messagebox.showerror(title="Error", message="Fill the fields correctly")

return

else:

if not self.tree.delete\_key(key):

messagebox.showerror(title="Error", message="Key does not exist")

else:

self.elements\_amount -= 1

self.pprint()

def pprint(self):

if self.visual\_tree:

for row in self.visual\_tree:

row.configure(text="")

text = self.tree.print\_tree(self.tree.root)

if self.elements\_amount <= 130:

for i, level in enumerate(text):

x = CTkLabel(self.window, text=level, font=("Courier", 12, "normal"))

x.grid(row=i+1, column=0)

self.visual\_tree.append(x)

else:

self.tree.console\_print(self.tree.root)

print("\n")

def test(self):

for \_ in range(15):

i, node, parent = self.tree.uniform\_bin\_search(randint(0, self.elements\_amount), self.tree.root)

if node:

print(f"\nkey: {node.keys[i][0]}")

print(self.tree.compares)

self.tree.compares = 0

### Приклади роботи

На рисунках 3.2 і 3.3 показані приклади роботи програми для додавання і пошуку запису.

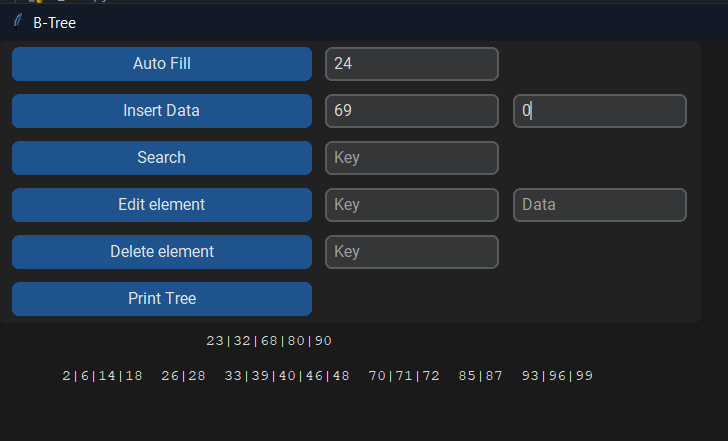


Рисунок 3.1 - Попередній вигляд B-дерева (t = 3)

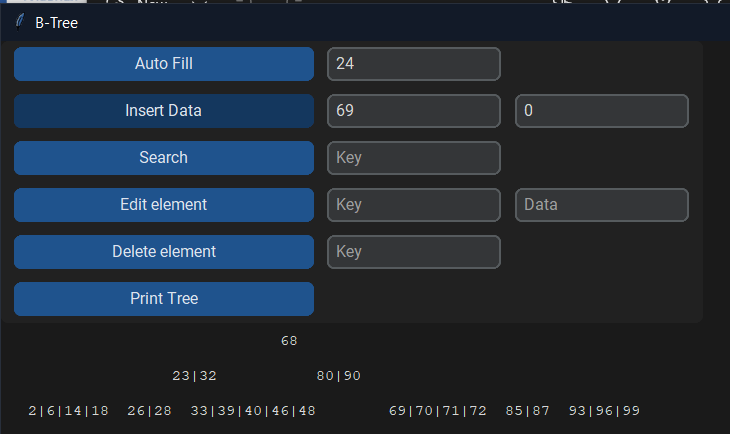


Рисунок 3.2 –Результат додавання запису 69

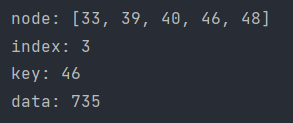


Рисунок 3.3 – Результат пошуку запису 46 у дереві

## Тестування алгоритму

### 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 | Шуканий ключ | Число порівнянь |
| 1 | 711 | 8 |
| 2 | 6906 | 12 |
| 3 | 8232 | 12 |
| 4 | 9300 | 11 |
| 5 | 411 | 8 |
| 6 | 6186 | 12 |
| 7 | 8412 | 12 |
| 8 | 159 | 11 |
| 9 | 7423 | 12 |
| 10 | 1644 | 9 |
| 11 | 1602 | 11 |
| 12 | 5091 | 12 |
| 13 | 8437 | 9 |
| 14 | 4632 | 11 |
| 15 | 7401 | 12 |
| Середня кількість порівнянь для знаходження елементу: 10.8 | | |

Висновок

В рамках лабораторної роботи було вивчено основні підходи проектування та обробки складних структур даних. На практиці було розроблено СУБД із графічним інтерфейсом, на основі структури даних B-дерево. У даній СУБД було реалізовано такі функції, як додавання, пошук, заміна та видалення даних.

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.