

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-21 Сергієнко Сергій
(шифр, прізвище, ім'я, по батькові)

Перевірів

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

Київ 2023

ЗМІСТ

1	МЕТА ЛАБОРАТОРНОЇ РОБОТИ	3
2	ЗАВДАННЯ	4
3	ВИКОНАННЯ	7
3.1	ПСЕВДОКОД АЛГОРИТМІВ	7
3.2	ЧАСОВА СКЛАДНІСТЬ ПОШУКУ	12
3.3	ПРОГРАМНА РЕАЛІЗАЦІЯ	12
3.3.1	<i>Вихідний код</i>	12
3.3.2	<i>Приклади роботи</i>	19
3.4	ТЕСТУВАННЯ АЛГОРИТМУ	21
3.4.1	<i>Часові характеристики оцінювання</i>	21
	ВИСНОВОК	22
	КРИТЕРІЇ ОЦІНЮВАННЯ	23

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи – вивчити основні підходи проектування 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	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

binary_search(lines, i, arg, key):

max_i = (len(lines) // B_SIZE)

s_iterator = 1

s = 2 ** (arg - s_iterator)

while True: **if** lines[i * B_SIZE] <= key < lines[(i + 1) * B_SIZE]: **return** i **elif** lines[i * B_SIZE] > key:

i -= (s + 1)

else:

i += (s + 1)

i = max(i, 0)

i = min(i, max_i)

s_iterator += 1

if s == 0: **break**

s = 2 ** (arg - s_iterator)

if s < 1:

s = 0

return i**find_block**(lines, key): **if** key == lines[0] or lines[0] == [EMPTY]: **return** 0

max_i = (len(lines) // B_SIZE)

lines = lines + [float('inf')]

k = int(math.log(max_i, 2))

```

i = 2 ** k
if lines[i * B_SIZE] <= key < lines[(i + 1) * B_SIZE]:
    return i
elif lines[i * B_SIZE] > key:
    return binary_search(lines, i, k, key)
else:
    l = int(math.log(max_i - 2 ** k + 1, 2))
    return binary_search(lines, (max_i + 1 - 2 ** l), l, key)

search(key, lines, lines_overflow):
    lines_index = [int(line[0]) if line[0] != EMPTY else None for line in lines]
    block_index = find_block(lines_index, key)
    block = lines_index[block_index * B_SIZE: (block_index + 1) * B_SIZE]
    key_in_overflow = False
    if key in block:
        key_index = block.index(key) + block_index * B_SIZE
    elif lines[(block_index + 1) * B_SIZE - 1][1] != EMPTY:
        lines_index_overflow = [int(line[0]) if line[0] != EMPTY else None for line
in lines_overflow]
        block_index = int(lines[(block_index + 1) * B_SIZE - 1][1])
        block = lines_index_overflow[block_index * B_SIZE:(block_index + 1) *
B_SIZE]
    while True:
        if key in block:
            key_index = block.index(key) + block_index * B_SIZE
            key_in_overflow = True
            break
        elif lines_overflow[(block_index + 1) * B_SIZE - 1][1] == EMPTY:
            return None
    else:

```



```

        block_index = int(lines_overflow[(block_index + 1) * B_SIZE - 1][1])
        block = lines_index_overflow[block_index * B_SIZE: (block_index + 1)
* B_SIZE]
    else:
        return None
    main_index = int(lines_overflow[key_index][1]) if key_in_overflow else
int(lines[key_index][1])
    return key_index, main_index

```

modify(key, value, main_lines):

```

    indexes = search(key)
    if indexes is None:
        return None
    main_lines[indexes[2]] = value
    return True

```

remove(key, lines, main_lines, overflow_lines):

```

    indexes = search(key)
    if indexes is None:
        return
    block_index, main_index = indexes[0], indexes[2]
    key_in_block_index = indexes[1] - indexes[0] * B_SIZE
    lines = lines if not indexes[3] else overflow_lines
    block = lines[block_index * B_SIZE: (block_index + 1) * B_SIZE]
    block[key_in_block_index] = [EMPTY, EMPTY]
    for i in range(key_in_block_index + 1, len(block) - 1):
        if block[i] != [EMPTY, EMPTY]:
            block[i - 1], block[i] = block[i], block[i - 1]
    lines[block_index * B_SIZE:(block_index + 1) * B_SIZE] = block
    main_lines[main_index] = 'Removed'

```

return True

add(key, value, lines, main_lines, overflow_lines):

indexes = search(key)

if indexes is not None:

return None

block_index = indexes[0] // B_SIZE

block_lines = .lines[block_index * B_SIZE: (block_index + 1) * B_SIZE]

main_index = len(main_lines)

main_lines.append(value)

if block_index + 1 == len(.lines) // B_SIZE and (not [EMPTY, EMPTY] in
block_lines or block_lines.index([EMPTY, EMPTY]) == B_SIZE - 1):

lines += [[key, main_index]]

lines += [[EMPTY, EMPTY]] * (B_SIZE - 1)

return True

if [EMPTY, EMPTY] in block_lines:

block_pos = block_lines.index([EMPTY, EMPTY])

if block_pos != B_SIZE - 1:

block_lines[block_pos] = [key, main_index]

block_lines = sorted(block_lines, key=.custom_sort)

lines[block_index * B_SIZE:(block_index + 1) * B_SIZE] = block_lines

else:

values_block = block_lines[:-1] + [[key, main_index]]

values_block = sorted(values_block, key=.custom_sort)

overflow_value = values_block[-1]

values_block = values_block[:-1]

if len(overflow_lines) > 0:

values_block += [[EMPTY, len(overflow_lines) // B_SIZE]]

overflow_lines += [overflow_value]

else:

```

        values_block += [[EMPTY, '0000000000']]
        overflow_lines = [overflow_value]
        overflow_lines += [[EMPTY, EMPTY]] * (B_SIZE - 1)
        lines[block_index * B_SIZE:(block_index + 1) * B_SIZE] = values_block
    else:
        index_block_overflow = int(block_lines[-1][1])
        block_overflow = overflow_lines[index_block_overflow *
        B_SIZE:(index_block_overflow + 1) * B_SIZE]
        all_indexes = []
        all_blocks = block_lines[:-1] + [[key, main_index]]
        while True:
            if [EMPTY, EMPTY] in block_overflow:
                break
            else:
                all_indexes.append(index_block_overflow)
                all_blocks += block_overflow[:-1]
                index_block_overflow = int(block_overflow[-1][1])
                block_overflow = .overflow_lines[index_block_overflow *
                B_SIZE:(index_block_overflow + 1) * B_SIZE]
        empty_index = block_overflow.index([EMPTY, EMPTY])
        all_indexes.append(index_block_overflow)
        all_blocks += block_overflow
        all_blocks = sorted(all_blocks, key=.custom_sort)
        lines[block_index * B_SIZE:(block_index + 1) * B_SIZE - 1] =
all_blocks[:B_SIZE - 1]
        all_blocks = all_blocks[B_SIZE - 1:]
        if empty_index == B_SIZE - 1:
            overflow_lines[index_block_overflow * B_SIZE + (B_SIZE - 1)] =
[EMPTY, len(.overflow_lines) // B_SIZE]
            overflow_lines += [all_blocks[-2]]

```

```

        overflow_lines += [[EMPTY, EMPTY]] * (B_SIZE - 1)
    for i, index in enumerate(all_indexes):
        overflow_lines[index * B_SIZE:(index + 1) * B_SIZE - 1] =
            all_blocks[i * (B_SIZE - 1):(i + 1) * (B_SIZE - 1)]
    return True

```

3.2 Часова складність пошуку

Максимальне число звернень до диска дорівнюватиме двійковому логарифму від заданого числа індексних блоків плюс одиниця. Одиниця потрібна оскільки, після пошуку номера запису в індексному області ми повинні ще звернутися до основної області файлу. Тому формула для обчислення максимального часу доступу в кількості звернень до диска виглядає наступним чином:

$$T_n = \log_2 N_{\text{бл.інд}} + 1.$$

3.3 Програмна реалізація

3.3.1 Вихідний код

```

import math
import os
import random

class IndexDirectFile:
    def __init__(self, num_indexes, index_filename, main_filename,
overflow_filename, fill_coefficient):
        self.num_indexes = num_indexes
        self.index_n = index_filename
        self.main_n = main_filename
        self.overflow_n = overflow_filename
        self.fill_coefficient = fill_coefficient
        self.b_size = 10
        self.max_index = 10
        self.empty = "000000None"
        self.lines = []
        self.main_lines = []

```

```

self.overflow_lines = []

def generate_file(self):
    indexes_per_block = int(self.b_size * self.fill_coefficient)
    empty_spaces_per_block = self.b_size - indexes_per_block
    num_blocks = (self.num_indexes + indexes_per_block - 1) //
indexes_per_block
    shuffled_list = list(range(1, self.num_indexes + 1))
    random.shuffle(shuffled_list)

    shuffled_iterator = 0
    with open(self.index_n, 'w') as f:
        for i in range(num_blocks):
            for j in range(indexes_per_block):
                index = i * indexes_per_block + j + 1
                if index <= self.num_indexes:
                    f.write(

f'{str(index).zfill(self.max_index)}, {str(shuffled_list[shuffled_iterator]).zfi
ll(self.max_index)}\n')

                    shuffled_iterator += 1
            else:
                f.write(f'{self.empty}, {self.empty}\n')
            for _ in range(empty_spaces_per_block):
                f.write(f'{self.empty}, {self.empty}\n')
    with open(self.main_n, 'w') as f:
        for i in range(len(shuffled_list)):
            f.write(f'value{str(shuffled_list.index(i + 1) +
1).zfill(self.max_index)}\n')
        with open(self.overflow_n, 'w') as _:
            pass

def upload_data(self):
    with open(self.main_n, 'r') as f:
        self.main_lines = [line.strip() for line in f]
    with open(self.index_n, 'r') as f:
        self.lines = [line.strip().split(',') for line in f]
    with open(self.overflow_n, 'r') as f:
        self.overflow_lines = [line.strip().split(',') for line in f]

def formatted_write(self, filename):
    if filename == self.main_n:
        with open(self.main_n, 'w') as f:

```

```

        for line in self.main_lines:
            f.write(f"{line}\n")
        return
data = self.lines if filename == self.index_n else self.overflow_lines
with open(filename, 'w') as f:
    for row in data:
        formatted_row = [str(item).zfill(self.max_index) for item in
row]

        f.write(','.join(formatted_row) + '\n')

def find_block(self, lines, key):
    if key == lines[0] or lines[0] == [self.empty]:
        return 0, 1
    lines = lines[:] + [float('inf')] * (self.b_size + 1)
    max_i = ((len(lines) - (self.b_size + 1)) // self.b_size)
    k = int(math.log(max_i, 2))
    i = 2 ** k
    while True:
        if lines[i * self.b_size] is None or lines[(i + 1) * self.b_size]
is None:

            i = i + 1 if i < max_i else i - 1
        else:
            break
    if lines[i * self.b_size] <= key < lines[(i + 1) * self.b_size]:
        return i, 1
    elif lines[i * self.b_size] > key:
        return self.binary_search(lines, i, k, key)
    else:
        l = int(math.log(max_i - 2 ** k + 1, 2))
        return self.binary_search(lines, (max_i + 1 - 2 ** l), l, key)

def binary_search(self, lines, i, arg, key):
    max_i = ((len(lines) - (self.b_size + 1)) // self.b_size)
    s_iterator = 1
    s = 2 ** (arg - s_iterator)
    comparisons_num = 1
    while True:
        comparisons_num += 1
        print("i: ", i, "s: ", s)
        while True:
            if lines[i * self.b_size] is None or lines[(i + 1) *
self.b_size] is None:

                i = i + 1 if i < max_i else i - 1

```

```

        else:
            break
    if lines[i * self.b_size] <= key < lines[(i + 1) * self.b_size]:
        return i, comparisons_num
    elif lines[i * self.b_size] > key:
        i -= (s + 1)
    else:
        i += (s + 1)
    i = max(i, 0)
    i = min(i, max_i)
    s_iterator += 1
    if s == 0:
        break
    s = 2 ** (arg - s_iterator)
    if s < 1:
        s = 0
    return i, comparisons_num

def search(self, key, check=True):
    lines_index = [int(line[0]) if line[0] != self.empty else None for line
in self.lines]
    block_index, comparisons_num = self.find_block(lines_index, key)

    block = lines_index[block_index * self.b_size: (block_index + 1) *
self.b_size]
    if not check:
        return block_index

    key_in_overflow = False
    if key in block: # ключ є у індексному файлі
        key_index = block.index(key) + block_index * self.b_size
    elif block.index(None) == self.b_size - 1 and self.lines[(block_index +
1) * self.b_size - 1][1] != self.empty:
        # ключ може бути у області переповнення
        lines_index_overflow = [int(line[0]) if line[0] != self.empty else
None for line in self.overflow_lines]
        block_index = int(self.lines[(block_index + 1) * self.b_size -
1][1])
        block = lines_index_overflow[block_index * self.b_size: (block_index
+ 1) * self.b_size]
    while True:
        if key in block:
            key_index = block.index(key) + block_index * self.b_size

```

```

        key_in_overflow = True
        break
        elif self.overflow_lines[(block_index + 1) * self.b_size -
1][1] == self.empty:
            return None
        else:
            block_index = int(self.overflow_lines[(block_index + 1) *
self.b_size - 1][1])
            block = lines_index_overflow[block_index * self.b_size:
(block_index + 1) * self.b_size]
            else:
                return None

            main_index = int(self.overflow_lines[key_index][1]) - 1 if
key_in_overflow else int(
                self.lines[key_index][1]) - 1
            return block_index, key_index, main_index, key_in_overflow,
comparisons_num, self.main_lines[main_index]

def modify(self, key, updated_value):
    indexes = self.search(key)
    if indexes is None:
        return None
    self.main_lines[indexes[2]] = updated_value
    return True

def remove(self, key):
    indexes = self.search(key)
    if indexes is None:
        return
    block_index, main_index = indexes[0], indexes[2]
    key_in_block_index = indexes[1] - indexes[0] * self.b_size
    lines = self.lines if not indexes[3] else self.overflow_lines
    block = lines[block_index * self.b_size: (block_index + 1) *
self.b_size]
    block[key_in_block_index] = [self.empty, self.empty]
    for i in range(key_in_block_index + 1, len(block) - 1):
        if block[i] != [self.empty, self.empty]:
            block[i - 1], block[i] = block[i], block[i - 1]
    lines[block_index * self.b_size:(block_index + 1) * self.b_size] =
block

    self.main_lines[main_index] = '00000000Removed'
    return True

```



```

def custom_sort(self, item):
    if item[0] == self.empty:
        return float('inf')
    else:
        return int(item[0])

def add(self, key, value):
    indexes = self.search(key)
    if indexes is not None:
        return None
    block_index = self.search(key, False)
    block_lines = self.lines[block_index * self.b_size: (block_index + 1) *
self.b_size]

    key = str(key).zfill(10)
    main_index = str(len(self.main_lines) + 1).zfill(10)
    self.main_lines.append(value)

    if block_index + 1 == len(self.lines) // self.b_size and (not
[self.empty, self.empty] in block_lines or

block_lines.index(

[self.empty, self.empty]) == self.b_size - 1):
        self.lines += [[key, main_index]]
        self.lines += [[self.empty, self.empty]] * (self.b_size - 1)
        return True
        if (len(self.lines) // self.b_size > block_index + 1 and
self.lines[(block_index + 1) * self.b_size][
0] == self.empty
        and (not [self.empty, self.empty] in block_lines
            or block_lines.index([self.empty, self.empty]) ==
self.b_size - 1)):
            self.lines[(block_index + 1) * self.b_size] = [key, main_index]
            return True

    if [self.empty, self.empty] in block_lines: # є вільне місце у
основному блоці
        block_pos = block_lines.index([self.empty, self.empty])
        if block_pos != self.b_size - 1: # є більш ніж 2 місця у основному
блоці
            block_lines[block_pos] = [key, main_index]

```

```

        block_lines = sorted(block_lines, key=self.custom_sort)
        self.lines[block_index * self.b_size:(block_index + 1) *
self.b_size] = block_lines
    else: # є лише 1 місце у основному блоці, треба створювати блок
для переповнення
        values_block = block_lines[:-1] + [[key, main_index]]
        values_block = sorted(values_block, key=self.custom_sort)
        overflow_value = values_block[-1]
        values_block = values_block[:-1]
        if len(self.overflow_lines) > 0: # є область переповнення
            values_block += [[self.empty, str(len(self.overflow_lines)
// self.b_size).zfill(self.max_index)]]
            self.overflow_lines += [overflow_value]
        else: # немає області переповнення
            values_block += [[self.empty, '0000000000']]
            self.overflow_lines = [overflow_value]

            self.overflow_lines += [[self.empty, self.empty]] *
(self.b_size - 1)

            self.lines[block_index * self.b_size:(block_index + 1) *
self.b_size] = values_block

    else: # немає вільного місця в блоці -> вже існує відповідний блок у
області переповнення
        index_block_overflow = int(block_lines[-1][1])
        block_overflow = self.overflow_lines[index_block_overflow *

self.b_size:(index_block_overflow + 1) * self.b_size]
        all_indexes = []
        all_blocks = block_lines[:-1] + [[key, main_index]]
        while True:
            if [self.empty, self.empty] in block_overflow: # є пусте місце
в блоці
                break
            else:
                all_indexes.append(index_block_overflow)
                all_blocks += block_overflow[:-1]
                index_block_overflow = int(block_overflow[-1][1])
                block_overflow = self.overflow_lines[index_block_overflow *

self.b_size:(index_block_overflow + 1) * self.b_size]
                empty_index = block_overflow.index([self.empty, self.empty])
                all_indexes.append(index_block_overflow)

```

```

all_blocks += block_overflow
all_blocks = sorted(all_blocks, key=self.custom_sort)
self.lines[block_index * self.b_size:(block_index + 1) *
self.b_size - 1] = all_blocks[:self.b_size - 1]
all_blocks = all_blocks[self.b_size - 1:]
if empty_index == self.b_size - 1: # якщо є тільки одне вільне
місце
self.overflow_lines[index_block_overflow * self.b_size +
(self.b_size - 1)] = \
[self.empty, str(len(self.overflow_lines) //
self.b_size).zfill(10)]
self.overflow_lines += [all_blocks[-2]] # all_blocks[-1] =
None
self.overflow_lines += [[self.empty, self.empty]] *
(self.b_size - 1)
for i, index in enumerate(all_indexes):
self.overflow_lines[index * self.b_size:(index + 1) *
self.b_size - 1] = \
all_blocks[i * (self.b_size - 1):(i + 1) * (self.b_size -
1)]
return True

```

3.3.2 Приклади роботи

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

The screenshot shows the Lab3 application interface. It has three main sections: 'Create', 'Select', and 'Generate', each with an 'Enter DB name' field and a corresponding button. Below these is a 'Selected Database:' section with a text field containing 'main.txt'. Further down, there are fields for 'Key:', 'Value:', 'Found value:' (containing 'value0000005500'), and 'Comparisons num:' (containing '11'). At the bottom are buttons for 'Find', 'Add', 'Modify', 'Remove', and 'Update'. On the right side, there is a table with 10 columns (labeled 1 to 10) and 10 rows of data. The data appears to be hexadecimal or binary representations of values.

Row	1	2	3	4	5	6	7	8	9	10
165	000	000	000	000	00000009953	00	00	00	0000	0000
166	000	000	000	000	00000009959	00	00	00	0000	0000
1661	000	000	000	000	00000009965	00	00	00	0000	0000
1662	000	000	000	000	00000009971	00	00	00	0000	0000
1663	000	000	000	000	00000009977	00	00	00	0000	0000
1664	000	000	000	000	00000009983	00	00	00	0000	0000
1665	000	000	000	000	00000009989	00	00	00	0000	0000
1666	000	000	000	000	00000009995	00	00	00	0000	0000
1667	000	000	000	000	000000None	00	00	00	0000	0000
ov1	000	000	000	000	000000None	00	00	00	0000	0000

Lab3

Create
Enter DB name
Create

Select
Select

Generate
Enter DB name
Generate

Selected Database: main.txt

Key: 10001 Value: value

Found value: value0000005500 Comparisons num: 11

Find Add Modify Remove Update

Row	1	2	3	4	5	6	7	8	9	10
1655	000	000	000	000	0000009953	00	00	00	0000	0000
1660	000	000	000	000	0000009959	00	00	00	0000	0000
1661	000	000	000	000	0000009965	00	00	00	0000	0000
1662	000	000	000	000	0000009971	00	00	00	0000	0000
1663	000	000	000	000	0000009977	00	00	00	0000	0000
1664	000	000	000	000	0000009983	00	00	00	0000	0000
1665	000	000	000	000	0000009989	00	00	00	0000	0000
1666	000	000	000	000	0000009995	00	00	00	0000	0000
1667	000	000	000	000	0000010001	00	00	00	0000	0000
ov1	000	000	000	000	000000None	00	00	00	0000	0000

Рисунок 3.1 – Додавання запису

Lab3

Create
Enter DB name
Create

Select
Select

Generate
Enter DB name
Generate

Selected Database: main.txt

Key: 5500 Value:

Found value: value0000005500 Comparisons num: 11

Find Add Modify Remove Update

Row	1	2	3	4	5	6	7	8	9	10
1	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
2	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
3	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
4	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
5	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
6	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
7	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
8	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
9	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
10	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

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	11
2	12
3	12
4	9
5	12
6	8
7	12
8	12
9	11
10	9
11	11
12	10
13	9
14	8
15	12

Середнє число порівнянь: 10.6.

Висновок

В рамках лабораторної роботи я виконав програмну реалізацію СУБД з графічним інтерфейсом користувача з функціями пошуку, додавання, видалення та редагування записів, що дозволило мені вивчити основні підходи проектування та обробки складних структур даних. Для зберігання даних я запрограмував структуру даних файл з щільним індексом. Переповнення індексної області у моїй програмі вирішується створенням області переповнення. Алгоритмом пошуку у структурі даних є метод Шарра.

Заповнивши базу даних випадковими значеннями до 10000, я зафіксував середнє із 15 пошуків число порівнянь для знаходження запису по ключу, що становить 10.6 порівнянь. Часові характеристики пошуку складають двійковий логарифм з кількості блоків плюс одиниця. Всього індексних блоків у досліджуваній базі даних вийшло 1667, отже максимальний час доступу в зверненнях до диска становить 12 звернень. Оскільки отримане число порівнянь є меншим за максимальний час доступу, то алгоритм пошуку в структурі даних реалізовано правильно.

КРИТЕРІЇ ОЦІНЮВАННЯ

За умови здачі лабораторної роботи до 26.11.2023 включно максимальний бал дорівнює – 5. Після 26.11.2023 максимальний бал дорівнює – 4,5.

Критерії оцінювання у відсотках від максимального балу:

- псевдокод алгоритму – 10%;
- аналіз часової складності – 5%;
- програмна реалізація алгоритму – 50%;
- робота з гіт – 20%
- тестування алгоритму – 10%;
- висновок – 5%.

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

+1 додатковий бал можна отримати за виконання та захист роботи до 19.11.2023.