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-15 Шабанов Метін Шаміль огли* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Ахаладзе Ілля Елдарійович* _____
(прізвище, ім'я, по батькові)

Київ 2022

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**
27	B-дерево t=10, метод Шарра

3 Виконання

3.1 Псевдокод алгоритмів

```
t = 10
class BTreeNode:
        public procedure new(self, leaf=False):
                self.leaf = leaf
                self.keys = []
                self.children = []
        endprocedure
endclass
```

Додавання ключа:

```
function insert(k):
        root = self.root
        if len(root.keys) < (2 * t) – 1 then:
                insert_non_full(root, k)
        else:
                temp = new BTreeNode()
                root = temp
                temp.children.insert(0, root)
                split_children(temp, 0)
                insert_non_full(temp, k)
        endif
endfunction

function insert_non_full(node, k):
   i = len(node.keys) - 1
   if node.leaf then:
        while i >= 0 and k < node.keys[i]:
```

```
            i = i - 1
            node.keys.insert(i + 1, k)
        endwhile
    else:
        while len(node.children) < len(node.keys) + 1:
            node.children.append(BTreeNode(True))
        endwhile

        while i >= 0 and k < node.keys[i]:
            i = i – 1
        endwhile
        i = i + 1
        if len(node.children[i].keys) == (2 * t) - 1 then:
            split_children(node, i)
            if k > node.keys[i] then:
                i += 1
            endif
        endif
        insert_non_full(node.children[i], k)
end function

function split_children(x, i):
    y = x.children[i]
    z = BTreeNode(y.leaf)
    x.children.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:
```

```
            z.children = y.children[t: 2 * t]
            y.children = y.children[0: t]
        endif
        endfuction
```

Пошук ключа (Алгоритм Шарра):

```
function Sharr_algorithm(key, node):
    k = math.floor(math.log(len(node.keys), 2))
    i = 2 ** k
    if node.keys[i - 1] == key then:
        return node, i - 1, node.keys[i - 1]
    elif node.keys[i - 1] > key then:
        return binary_search(key, node)
    else:
        l = math.floor(math.log(len(node.keys) - 2 ** k + 1, 2))
        i = len(node.keys) + 1 - 2 ** l
        l = l - 1
        delta = math.floor(2 ** l)
        while delta != 0 and  0 < i <= len(node.keys):
            if node.keys[int(i) - 1] < key then:
                i = i + (delta // 2 + 1)
                l = l - 1
                delta = 2 ** l
            elif node.keys[int(i) - 1] > key then:
                i = i - (delta // 2 + 1)
                l = l - 1
                delta = 2 ** l
            else:
                return node, i - 1, node.keys[int(i) - 1]
            endif
```

```
        endwhile
        return null
endif
endfunction

function binary_search(key, node):
    temp_list = node.keys
    i = len(temp_list) // 2 + 1
    delta = len(temp_list) // 2
    temp_list.append(float('inf'))
    while delta != 0 and 1 <= i <= len(temp_list) then:
        if temp_list[i - 1] < key:
            i = i + (delta // 2 + 1)
            delta //= 2
        elif temp_list[i - 1] > key then:
            i = i - (delta // 2 + 1)
            delta //= 2
        else:
            temp_list.remove(float('inf'))
            return node, i - 1, temp_list[i - 1]
        endif
    endwhile
    temp_list.remove(float('inf'))
    return null

    Видалення ключа:
function delete(node, value):
    i = 0
    while i < len(node.keys) and value > node.keys[i]:
        i += 1
```

```
 if node.leaf then:
    if i < len(node.keys) and node.keys[i] == value then:
       node.keys.pop(i)
       return
    endif
    return
endif

 if i < len(node.keys) and node.keys[i] == value then:
    return delete_internal_node(node, value, i)
 elif len(node.children[i].keys) >= t then:
    delete(node.children[i], value)
 else:
    if i != 0 and i + 2 < len(node.children) then:
       if len(node.children[i - 1].keys) >= t then:
          delete_sibling(node, i, i - 1)
       elif len(node.children[i + 1].keys) >= t then:
          delete_sibling(node, i, i + 1)
       else:
          delete_merge(node, i, i + 1)
       endif
    elif i == 0 then:
       if len(node.children[i + 1].keys) >= t then:
          delete_sibling(node, i, i + 1)
       else:
          delete_merge(node, i, i + 1)
        endif
    elif i + 1 == len(node.children) then:
       if len(node.children[i - 1].keys) >= t then:
          delete_sibling(node, i, i - 1)
```

else:
delete_merge(node, i, i - 1)
endif
delete(node.children[i], value)
endif

3.2 Часова складність пошуку

Суть написаної реалізації полягає в тому, що алгоритм Шарра працює на обраній вершині, і у випадку, якщо не знайшов у вершині потрібне значення, то йде до нащадка з підходящого інтервалу, і шукає потрібне значення вже в ньому. Такий хід дій повторюється доти, доки занурення не дійде листової вершини. Враховуючи таку інтерпретацію порядку виконання, можемо стверджувати, що алгоритм працює за часовою складністю O(n * log(2*t - 1)), де n – висота дерева, t – заданий параметр B-дерева.

3.3 Програмна реалізація

3.3.1 Вихідний код

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

    def __getitem__(self, item):
        return self.keys[item]

class BTree:
    def __init__(self, t=10):
        self.root = BTreeNode(True)
        self.t = t

    def search(self, key, start_node='default'):
        if start_node == 'default':
            start_node = self.root

        data = self.__Sharr_algorithm(key, start_node)
        if data is not None:
            return data
        else:
            if not start_node.leaf:
                for i in range(len(start_node.keys)):
                    if key < start_node.keys[i]:
                        return self.search(key, start_node.children[i])
                if start_node.children:
```

```
                return self.search(key, start_node.children[-1])

    def insert(self, k):
        root = self.root
        if len(root.keys) < (2 * self.t) - 1:
            self.__insert_non_full(root, k)
        else:
            temp = BTreeNode()
            self.root = temp
            temp.children.insert(0, root)
            self.__split_children(temp, 0)
            self.__insert_non_full(temp, k)

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

        if i < len(node.keys) and node.keys[i] == value:
            return self.__delete_internal_node(node, value, i)
        elif len(node.children[i].keys) >= t:
            self.delete(node.children[i], value)
        else:
            if i != 0 and i + 2 < len(node.children):
                if len(node.children[i - 1].keys) >= t:
                    self.__delete_sibling(node, i, i - 1)
                elif len(node.children[i + 1].keys) >= t:
                    self.__delete_sibling(node, i, i + 1)
                else:
                    self.__delete_merge(node, i, i + 1)
            elif i == 0:
                if len(node.children[i + 1].keys) >= t:
                    self.__delete_sibling(node, i, i + 1)
                else:
                    self.__delete_merge(node, i, i + 1)
            elif i + 1 == len(node.children):
                if len(node.children[i - 1].keys) >= t:
                    self.__delete_sibling(node, i, i - 1)
                else:
                    self.__delete_merge(node, i, i - 1)
            self.delete(node.children[i], value)

    def edit(self, old_obj, new_value):
        self.delete(self.root, old_obj)
        new_obj = DatabaseObject(old_obj.index, new_value)
        self.insert(new_obj)

    def __insert_non_full(self, node, k):
        i = len(node.keys) - 1
        if node.leaf:
            while i >= 0 and k < node.keys[i]:
                i -= 1
            node.keys.insert(i + 1, k)
        else:
            while len(node.children) < len(node.keys) + 1:
                node.children.append(BTreeNode(True))
            while i >= 0 and k < node.keys[i]:
                i -= 1
```

```
            i += 1
            if len(node.children[i].keys) == (2 * self.t) - 1:
                self.__split_children(node, i)
                if k > node.keys[i]:
                    i += 1
            self.__insert_non_full(node.children[i], k)

    def __split_children(self, x, i):
        t = self.t
        y = x.children[i]
        z = BTreeNode(y.leaf)
        x.children.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.children = y.children[t: 2 * t]
            y.children = y.children[0: t]

    def __Sharr_algorithm(self, key, node):
        k = math.floor(math.log(len(node.keys), 2))
        i = 2 ** k
        if node.keys[i - 1] == key:
            return node, i - 1, node.keys[i - 1]
        elif node.keys[i - 1] > key:
            return self.__binary_search(key, node)
        else:
            l = math.floor(math.log(len(node.keys) - 2 ** k + 1, 2))
            i = len(node.keys) + 1 - 2 ** l
            l -= 1
            delta = math.floor(2 ** l)
            while delta != 0 and  0 < i <= len(node.keys):
                if node.keys[int(i) - 1] < key:
                    i = i + (delta // 2 + 1)
                    l -= 1
                    delta = 2 ** l
                elif node.keys[int(i) - 1] > key:
                    i = i - (delta // 2 + 1)
                    l -= 1
                    delta = 2 ** l
                else:
                    return node, i - 1, node.keys[int(i) - 1]
            return None

    def __binary_search(self, key, node):
        temp_list = node.keys
        i = len(temp_list) // 2 + 1
        delta = len(temp_list) // 2
        temp_list.append(float('inf'))
        while delta != 0 and 2 <= i <= len(temp_list):
            if temp_list[i - 2] < key:
                i = i + (delta // 2 + 1)
                delta //= 2
            elif temp_list[i - 2] > key:
                i = i - (delta // 2 + 1)
                delta //= 2
            else:
                temp_list.remove(float('inf'))
                return node, i - 2, temp_list[i - 2]
        temp_list.remove(float('inf'))
        return None

    def __delete_internal_node(self, node, value, i):
        t = self.t
```

```
        if node.leaf:
            if node.keys[i] == value:
                node.keys.pop(i)
                return
            return

        if len(node.children[i].keys) >= t:
            node.keys[i] = self.__delete_predecessor(node.children[i])
            return
        elif len(node.children[i + 1].keys) >= t:
            node.keys[i] = self.__delete_successor(node.children[i + 1])
            return
        else:
            self.__delete_merge(node, i, i + 1)
            self.__delete_internal_node(node.children[i], value, self.t - 1)

    def __delete_predecessor(self, node):
        if node.leaf:
            return node.keys.pop()
        n = len(node.keys) - 1
        if len(node.children[n].keys) >= self.t:
            self.__delete_sibling(node, n + 1, n)
        else:
            self.__delete_merge(node, n, n + 1)
        self.__delete_predecessor(node.children[n])

    def __delete_successor(self, node):
        if node.leaf:
            return node.keys.pop(0)
        if len(node.children[1].keys) >= self.t:
            self.__delete_sibling(node, 0, 1)
        else:
            self.__delete_merge(node, 0, 1)
        self.__delete_successor(node.children[0])

    def __delete_merge(self, node, i, j):
        cnode = node.children[i]
        if j > i:
            rsnode = node.children[j]
            cnode.keys.append(node.keys[i])
            for k in range(len(rsnode.keys)):
                cnode.keys.append(rsnode.keys[k])
                if len(rsnode.children) > 0:
                    cnode.children.append(rsnode.child[k])
            if len(rsnode.children) > 0:
                cnode.children.append(rsnode.children.pop())
            new = cnode
            node.keys.pop(i)
            node.children.pop(j)
        else:
            lsnode = node.child[j]
            lsnode.keys.append(node.keys[j])
            for i in range(len(cnode.keys)):
                lsnode.keys.append(cnode.keys[i])
                if len(lsnode.children) > 0:
                    lsnode.children.append(cnode.children[i])
            if len(lsnode.children) > 0:
                lsnode.children.append(cnode.children.pop())
            new = lsnode
            node.keys.pop(j)
            node.children.pop(i)

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

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

3.3.2 Приклади роботи

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

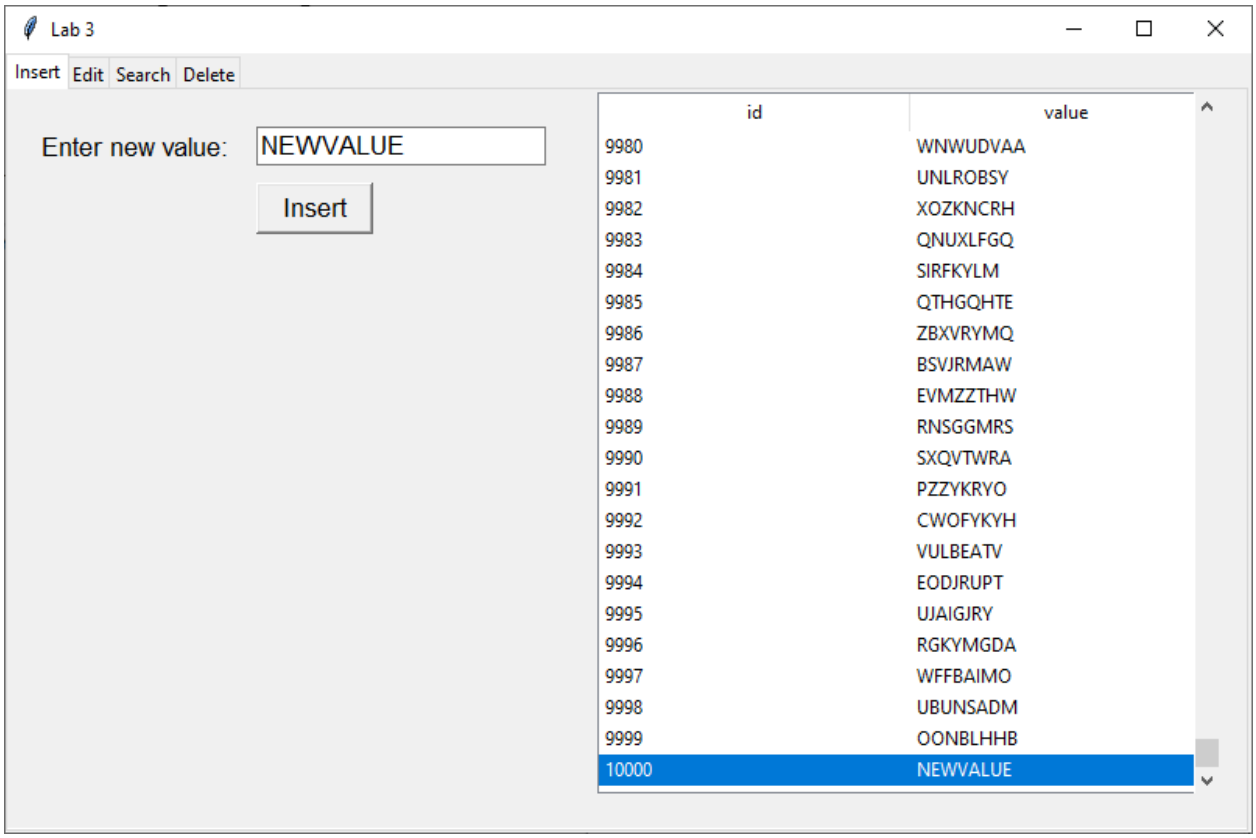

Рисунок 3.1 –Додавання запису

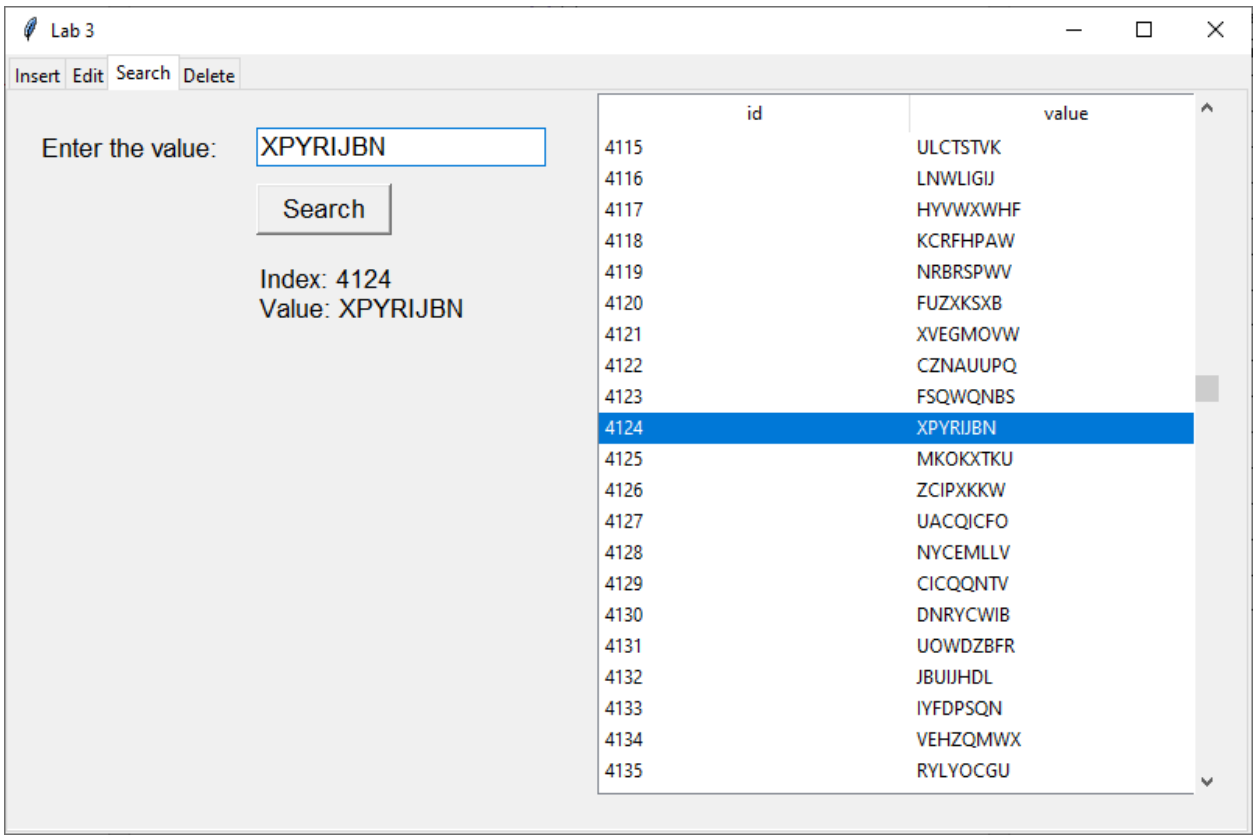

Рисунок 3.2 – Пошук запису

3.4 Тестування алгоритму

3.4.1 Часові характеристики оцінювання

В таблиці 3.1 наведено кількість порівнянь для 15 спроб пошуку запису по ключу. В середньому, число порівнянь дорівнює 3179 операціям.

Таблиця 3.1 – Число порівнянь при спробі пошуку запису по ключу

Номер спроби пошуку	Число порівнянь
1	6473
2	2171
3	2175
4	6473
5	4327
6	4325
7	2173
8	4324
9	2176
10	2178
11	2171
12	2181
13	30
14	2176
15	4327

Висновок

В рамках лабораторної роботи вивчено основні підходи проектування та обробки складних структур даних на прикладі структури B-дерева та алгоритму Шарра. Виконано програмну реалізацію невеликої СУБД з графічним інтерфейсом користувача, з функціями пошуку, додавання, видалення та редагування записів. Під час розробки програмної реалізації методу пошуку також було враховано особливості використаної структури даних та модифіковано алгоритм відповідним чином.

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.