

**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-25 Карпов Л.В.  
(шифр, прізвище, ім'я, по батькові)

**Перевірив**

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

Київ 2022

## ЗМІСТ

<b>1</b>	<b>МЕТА ЛАБОРАТОРНОЇ РОБОТИ .....</b>	<b>3</b>
<b>2</b>	<b>ЗАВДАННЯ .....</b>	<b>4</b>
<b>3</b>	<b>ВИКОНАННЯ.....</b>	<b>7</b>
3.1	ПСЕВДОКОД АЛГОРИТМІВ.....	7
3.2	ЧАСОВА СКЛАДНІСТЬ ПОШУКУ .....	8
3.3	ПРОГРАМНА РЕАЛІЗАЦІЯ .....	9
3.3.1	<i>Вихідний код .....</i>	<i>9</i>
3.3.2	<i>Приклади роботи .....</i>	<i>16</i>
3.4	ТЕСТУВАННЯ АЛГОРИТМУ .....	19
3.4.1	<i>Часові характеристики оцінювання.....</i>	<i>19</i>
	<b>ВИСНОВОК .....</b>	<b>20</b>
	<b>КРИТЕРІЇ ОЦІНЮВАННЯ .....</b>	<b>21</b>

## 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	<b>В-дерево <math>t=50</math>, бінарний пошук</b>
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 ВИКОНАННЯ

#### 3.1 Псевдокод алгоритмів

if x is not None:

    i = 0

    while i < len(x.keys) and k > x.keys[i][0]:

        i += 1

    if i < len(x.keys) and k == x.keys[i][0]:

        return x, i

    elif x.leaf:

        return

    else:

        # Search its children

        return self.search(k, x.children[i])

else:

    # Search the entire tree

    return self.search(k, self.root)

i = 0

while i < len(x.keys) and k > x.keys[i][0]:

    i += 1

# Deleting the key if the node is a leaf

if x.leaf:

    if i < len(x.keys) and x.keys[i][0] == k:

        x.keys.pop(i)

        return

return

# Calling '\_deleteInternalNode' when x is an internal node and contains the key 'k'

if i < len(x.keys) and x.keys[i][0] == k:

```

        return self._deleteInternalNode(x, k, i)
# Recursively calling 'delete' on x's children
elif len(x.children[i].keys) >= T:
    self.delete(k, x.children[i])
# Ensuring that a child always has atleast 't' keys
else:
    if i != 0 and i + 2 < len(x.children):
        if len(x.children[i - 1].keys) >= T:
            self._deleteSibling(x, i, i - 1)
        elif len(x.children[i + 1].keys) >= T:
            self._deleteSibling(x, i, i + 1)
        else:
            self._deleteMerge(x, i, i + 1)
    elif i == 0:
        if len(x.children[i + 1].keys) >= T:
            self._deleteSibling(x, i, i + 1)
        else:
            self._deleteMerge(x, i, i + 1)
    elif i + 1 == len(x.children):
        if len(x.children[i - 1].keys) >= T:
            # self._deleteSibling(x, i, i - 1)
            pass
        else:
            self._deleteMerge(x, i, i - 1)
        i -= 1
    self.delete(k, x.children[i])

```

### 3.2 Часова складність пошуку

$O(\log n)$



## 3.3 Програмна реалізація

### 3.3.1 Вихідний код

```
import random

from pydantic import BaseModel, Field

iter_counter = 0

T = 50

class Node(BaseModel):
    keys: list = Field(default_factory=list)
    children: list['Node'] = Field(default_factory=list)

    @property
    def leaf(self):
        return len(self.children) == 0

class BTree(BaseModel):
    root: Node = Field(default_factory=Node)

    def printTree(self, x, lvl=0):
        """
        Prints the complete B-Tree
        :param x: Root node.
        :param lvl: Current level.
        """
        print("Level ", lvl, " --> ", len(x.keys), end=": ")
        for i in x.keys:
            print(i, end=" ")
        print()
        lvl += 1
        if len(x.children) > 0:
            for i in x.children:
                self.printTree(i, lvl)

    def search(self, k, x=None) -> tuple | None:
        """
        Search for key 'k' at position 'x'
        :param k: The key to search for.
        :param x: The position to search from. If not specified, then search
occurs from the root.
        :return: 'None' if 'k' is not found. Otherwise returns a tuple of
(node, index) at which the key was found.
        """
        global iter_counter
        iter_counter += 1
        if x is not None:
            i = 0
            while i < len(x.keys) and k > x.keys[i][0]:
                i += 1
            if i < len(x.keys) and k == x.keys[i][0]:
                return x, i
            elif x.leaf:
                return
            else:
                # Search its children
                return self.search(k, x.children[i])
```

```

    else:
        # Search the entire tree
        return self.search(k, self.root)

def insert(self, k):
    """
    Calls the respective helper functions for insertion into B-Tree
    :param k: The key to be inserted.
    """
    root = self.root
    # If a node is full, split the child
    if len(root.keys) == (2 * T) - 1:
        temp = Node()
        self.root = temp
        # Former root becomes 0'th child of new root 'temp'
        temp.children.insert(0, root)
        self._splitChild(temp, 0)
        self._insertNonFull(temp, k)
    else:
        self._insertNonFull(root, k)

def _insertNonFull(self, x, k):
    """
    Inserts a key in a non-full node
    :param x: The key to be inserted.
    :param k: The position of node.
    """
    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.children[i].keys) == (2 * T) - 1:
            self._splitChild(x, i)
            if k[0] > x.keys[i][0]:
                i += 1
            self._insertNonFull(x.children[i], k)
        else:
            self._insertNonFull(x.children[i], k)

def _splitChild(self, x, i):
    """
    Splits the child of node at 'x' from index 'i'
    :param x: Parent node of the node to be split.
    :param i: Index value of the child.
    """
    y = x.children[i]
    z = Node()
    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 delete(self, k: int, x: Node = None):
    """
    Calls the respective helper functions for deletion from B-Tree
    :param x: The node, according to whose relative position, helper

```

functions are called.

```
    :param k: The key to be deleted.
    """
    x = x or self.root
    i = 0
    while i < len(x.keys) and k > x.keys[i][0]:
        i += 1
    # Deleting the key if the node is a leaf
    if x.leaf:
        if i < len(x.keys) and x.keys[i][0] == k:
            x.keys.pop(i)
            return
        return

    # Calling '_deleteInternalNode' when x is an internal node and
contains the key 'k'
    if i < len(x.keys) and x.keys[i][0] == k:
        return self._deleteInternalNode(x, k, i)
    # Recursively calling 'delete' on x's children
    elif len(x.children[i].keys) >= T:
        self.delete(k, x.children[i])
    # Ensuring that a child always has atleast 't' keys
    else:
        if i != 0 and i + 2 < len(x.children):
            if len(x.children[i - 1].keys) >= T:
                self._deleteSibling(x, i, i - 1)
            elif len(x.children[i + 1].keys) >= T:
                self._deleteSibling(x, i, i + 1)
            else:
                self._deleteMerge(x, i, i + 1)
        elif i == 0:
            if len(x.children[i + 1].keys) >= T:
                self._deleteSibling(x, i, i + 1)
            else:
                self._deleteMerge(x, i, i + 1)
        elif i + 1 == len(x.children):
            if len(x.children[i - 1].keys) >= T:
                # self._deleteSibling(x, i, i - 1)
                pass
            else:
                self._deleteMerge(x, i, i - 1)
            i -= 1
        self.delete(k, x.children[i])

def _deleteInternalNode(self, x, k, i):
    """
    Deletes internal node
    :param x: The internal node in which key 'k' is present.
    :param k: The key to be deleted.
    :param i: The index position of key in the list
    """
    # Deleting the key if the node is a leaf
    if x.leaf:
        if x.keys[i][0] == k[0]:
            x.keys.pop(i)
            return
        return

    # Replacing the key with its predecessor and deleting predecessor
    if len(x.children[i].keys) >= T:
        x.keys[i] = self._deletePredecessor(x.children[i])
        return
    # Replacing the key with its successor and deleting successor
    elif len(x.children[i + 1].keys) >= T:
```

```

        x.keys[i] = self._deleteSuccessor(x.children[i + 1])
        return
# Merging the child, its left sibling and the key 'k'
else:
    self._deleteMerge(x, i, i + 1)
    self.delete(k, x.children[i])

def _deletePredecessor(self, x):
    """
    Deletes predecessor of key 'k' which is to be deleted
    :param x: Node
    :return: Predecessor of key 'k' which is to be deleted
    """
    if x.leaf:
        return x.keys.pop()
    n = len(x.keys) - 1
    if len(x.children[n].keys) >= T:
        self._deleteSibling(x, n + 1, n)
    else:
        self._deleteMerge(x, n, n + 1)
    return self._deletePredecessor(x.children[n])

def _deleteSuccessor(self, x):
    """
    Deletes successor of key 'k' which is to be deleted
    :param x: Node
    :return: Successor of key 'k' which is to be deleted
    """
    if x.leaf:
        return x.keys.pop(0)
    if len(x.children[1].keys) >= T:
        self._deleteSibling(x, 0, 1)
    else:
        self._deleteMerge(x, 0, 1)
    return self._deleteSuccessor(x.children[0])

def _deleteMerge(self, x, i, j):
    """
    Merges the children of x and one of its own keys
    :param x: Parent node
    :param i: The index of one of the children
    :param j: The index of one of the children
    """
    cNode = x.children[i]

    # Merging the x.children[i], x.children[j] and x.keys[i]
    if j > i:
        rsNode = x.children[j]
        cNode.keys.append(x.keys[i])
        # Assigning keys of right sibling node to child node
        for k in range(len(rsNode.keys)):
            cNode.keys.append(rsNode.keys[k])
            if not rsNode.leaf:
                cNode.children.append(rsNode.children[k])
        if not rsNode.leaf:
            cNode.children.append(rsNode.children.pop())
        new = cNode
        x.keys.pop(i)
        x.children.pop(j)
    # Merging the x.children[i], x.children[j] and x.keys[i]
    else:
        lsNode = x.children[j]
        lsNode.keys.append(x.keys[j])
        # Assigning keys of left sibling node to child node

```

```

        for k in range(len(cNode.keys)):
            lsNode.keys.append(cNode.keys[k])
            if not lsNode.leaf:
                lsNode.children.append(cNode.children[k])
        if not lsNode.leaf:
            lsNode.children.append(cNode.children.pop())
        new = lsNode
        x.keys.pop(j)
        x.children.pop(i)

    # If x is root and is empty, then re-assign root
    if x == self.root and len(x.keys) == 0:
        self.root = new

    @staticmethod
    def _deleteSibling(x, i, j):
        """
        Borrows a key from j'th child of x and appends it to i'th child of x
        :param x: Parent node
        :param i: The index of one of the children
        :param j: The index of one of the children
        """
        cNode = x.children[i]
        if i < j:
            # Borrowing key from right sibling of the child
            rsNode = x.children[j]
            cNode.keys.append(x.keys[i])
            x.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:
            # Borrowing key from left sibling of the child
            lsNode = x.children[j]
            cNode.keys.insert(0, x.keys[i - 1])
            x.keys[i - 1] = lsNode.keys.pop()
            if len(lsNode.children) > 0:
                cNode.children.insert(0, lsNode.children.pop())

def test():
    tree = BTree()

    # Insert
    customNo = 10000
    for i in random.sample(range(customNo), customNo):
        tree.insert((i, f'd{i}'))
    print()

    for _ in range(15):
        key = random.randint(0, customNo)
        global iter_counter
        iter_counter = 0
        print(tree.search(key))
        print(iter_counter)

if __name__ == '__main__':
    test()
import json
import tkinter as tk
from tkinter import messagebox
from tkinter.filedialog import asksaveasfilename, askopenfilename

```

```

from pydantic import ValidationError

import core

tree = core.BTree()

window = tk.Tk()
window.resizable(False, False)

label_key = tk.Label(window, text="Key")
label_value = tk.Label(window, text="Value")

entry_key = tk.Entry(window, width=10)
entry_value = tk.Entry(window, width=10)

def get_key():
    try:
        key = int(entry_key.get())
    except ValueError:
        messagebox.showerror("Value error", "Key should be number!")
        window.update()
        return
    return key

def add_event():
    key = get_key()

    if (res := tree.search(key)) is not None:
        node, index = res
        node.keys[index] = key, entry_value.get()
    else:
        tree.insert((key, entry_value.get()))

    # entry_key.delete(0, tk.END)
    entry_value.delete(0, tk.END)

def search_event():
    key = get_key()

    if (res := tree.search(key)) is not None:
        node, index = res
        entry_value.delete(0, tk.END)
        entry_value.insert(0, node.keys[index][1])
    else:
        messagebox.showerror("Value error", "Key has not found in tree!")
        window.update()
        return

def remove_event():
    key = get_key()

    if tree.search(key) is not None:
        tree.delete(key)
        entry_key.delete(0, tk.END)
        entry_value.delete(0, tk.END)
    else:
        messagebox.showerror("Value error", "Key has not found in tree!")
        window.update()
        return

```

```

def clear_event():
    global tree
    tree = core.BTree()

def open_from_file_event():
    filepath = askopenfilename(
        filetypes=[("B-tree files", "*.btree"), ("All files", "*.*")],
        initialdir='~/Desktop',
    )
    if not filepath:
        return
    with open(filepath, "r") as input_file:
        raw_data = input_file.read()
        try:
            data = json.loads(raw_data)
            global tree
            tree = core.BTree(**data)
        except ValidationError:
            messagebox.showerror("File error", "File is broken!")

def save_to_file_event():
    filepath = asksaveasfilename(
        defaultextension="*.btree",
        filetypes=[("B-tree files", "*.btree"), ("All files", "*.*")],
        initialdir='~/Desktop',
    )
    if not filepath:
        return
    with open(filepath, "w") as output_file:
        output_file.write(tree.model_dump_json())

button_add = tk.Button(window, text='Add/Edit', command=add_event)
button_search = tk.Button(window, text='Search', command=search_event)
button_remove = tk.Button(window, text='Remove', command=remove_event)
button_clear = tk.Button(window, text='Clear', command=clear_event)

menu = tk.Menu(window)
window.config(menu=menu)

menu_file = tk.Menu(menu, tearoff=0)
menu_file.add_command(label='Open', command=open_from_file_event)
menu_file.add_command(label='Save', command=save_to_file_event)

menu.add_cascade(label='File', menu=menu_file)

label_key.grid(row=0, column=0, sticky="ew", padx=5, pady=5)
label_value.grid(row=0, column=1, sticky="ew", padx=5, pady=5)

entry_key.grid(row=1, column=0, sticky="ew", padx=5, pady=5)
entry_value.grid(row=1, column=1, sticky="ew", padx=5, pady=5)

button_add.grid(row=2, column=0, sticky="ew", padx=5, pady=5)
button_search.grid(row=2, column=1, sticky="ew", padx=5, pady=5)
button_remove.grid(row=3, column=0, sticky="ew", padx=5, pady=5)
button_clear.grid(row=3, column=1, sticky="ew", padx=5, pady=5)

window.mainloop()

```

### 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

The image shows two screenshots of a Tkinter window titled 'File'. The window has a title bar with a feather icon, the text 'tk', and standard window controls (minimize, maximize, close). The main content area is divided into two columns: 'Key' and 'Value'. In the first screenshot, the 'Key' field contains '1' and the 'Value' field contains 'something'. Below the fields are four buttons: 'Add/Edit', 'Search', 'Remove', and 'Clear'. In the second screenshot, the 'Key' field still contains '1', but the 'Value' field is empty. The buttons remain the same.

Рисунок 3.2 – Пошук запису

The image shows a single screenshot of a Tkinter window titled 'File'. The window has a title bar with a feather icon, the text 'tk', and standard window controls (minimize, maximize, close). The main content area is divided into two columns: 'Key' and 'Value'. The 'Key' field contains '1' and the 'Value' field is empty. Below the fields are four buttons: 'Add/Edit', 'Search', 'Remove', and 'Clear'.



tk

File

Key	Value
1	something

Add/Edit Search

Remove Clear

tk


File

Key	Value
5	

Add/Edit Search

Remove Clear

Value error

 Key has not found in tree!

OK

Рисунок 3.3 – Редагування запису

tk

File

Key	Value
1	something2

Add/Edit Search

Remove Clear

Рисунок 3.4 – Виделення запису

tk

File

Key	Value
1	something2

Add/Edit Search

Remove Clear

tk


File

Key	Value
1	

Add/Edit Search

Remove Clear

Value error

 Key has not found in tree!

OK

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

## ВИСНОВОК

В рамках лабораторної роботи була вивчена структура та метод побудови В-дерева та розроблені алгоритми додавання, редагування, пошуку та видалення елементів В-дерева за ключем.

Застосовується В-дерево в основному в проектуванні баз даних оскільки воно значно підвищує швидкість пошуку індексованої інформації

## 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.