# 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)	

## 3MICT

1	МЕТА ЛАБОРАТОРНОІ РОБОТИ	3
2	ЗАВДАННЯ	4
3	ВИКОНАННЯ	7
	3.1 ПСЕВДОКОД АЛГОРИТМІВ	7
	3.2 ЧАСОВА СКЛАДНІСТЬ ПОШУКУ	8
	3.3 ПРОГРАМНА РЕАЛІЗАЦІЯ	9
	3.3.1 Вихідний код	9
	3.3.2 Приклади роботи	16
	3.4 ТЕСТУВАННЯ АЛГОРИТМУ	19
	3.4.1 Часові характеристики оцінювання	19
ВИ	<b>ІСНОВОК</b>	20
КР	РИТЕРІЇ ОПІНЮВАННЯ	21

# 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, бінарн	
	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 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])
      Часова складність пошуку
O(\log n)
```

3.2

#### 3.3 Програмна реалізація

#### 3.3.1 Вихідний код

```
import random
from pydantic import BaseModel, Field
T = 50
class Node(BaseModel):
    @property
class BTree(BaseModel):
        print("Level ", lvl, " --> ", len(x.keys), end=": ")
                self.printTree(i, lvl)
            while i < len(x.keys) and k > x.keys[i][0]:
            if i < len(x.keys) and k == x.keys[i][0]:
```

```
temp = Node()
    self.root = temp
    temp.children.insert(0, root)
    self._splitChild(temp, 0)
    self. insertNonFull(temp, k)
    self. insertNonFull(root, k)
i = len(x.keys) - 1
if x.leaf:
    x.keys.append((None, None))
    x.keys[i + 1] = k
    while i \ge 0 and k[0] < x.keys[i][0]:
    if len(x.children[i].keys) == (2 * T) - 1:
        self._splitChild(x, i)
        if k[0] > x.keys[i][0]:
    self. insertNonFull(x.children[i], k)
x.keys.insert(i, y.keys[T - 1])
z.keys = y.keys[T: (2 * T) - 1]
```

```
while i < len(x.keys) and k > x.keys[i][0]:
    if i < len(x.keys) and x.keys[i][0] == k:
        x.keys.pop(i)
if i < len(x.keys) and x.keys[i][0] == k:</pre>
        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)
            self. deleteMerge(x, i, i + 1)
            self. deleteSibling(x, i, i + 1)
            self. deleteMerge(x, i, i + 1)
        if len(x.children[i - 1].keys) >= T:
            self. deleteMerge(x, i, i - 1)
    if x.keys[i][0] == k[0]:
       x.keys.pop(i)
if len(x.children[i].keys) >= T:
    x.keys[i] = self. deletePredecessor(x.children[i])
```

```
x.keys[i] = self. deleteSuccessor(x.children[i + 1])
    self. deleteMerge(x, i, i + 1)
   return x.keys.pop()
n = len(x.keys) - 1
if len(x.children[n].keys) >= T:
   self. deleteSibling(x, n + 1, n)
   self. deleteMerge(x, n, n + 1)
if x.leaf:
if len(x.children[1].keys) >= T:
   self. deleteSibling(x, 0, 1)
    self. deleteMerge(x, 0, 1)
cNode = x.children[i]
   cNode.keys.append(x.keys[i])
    for k in range(len(rsNode.keys)):
        cNode.keys.append(rsNode.keys[k])
           cNode.children.append(rsNode.children[k])
       cNode.children.append(rsNode.children.pop())
   x.keys.pop(i)
   x.children.pop(j)
   lsNode = x.children[j]
   lsNode.keys.append(x.keys[j])
```

```
for k in range(len(cNode.keys)):
                lsNode.keys.append(cNode.keys[k])
                    lsNode.children.append(cNode.children[k])
                lsNode.children.append(cNode.children.pop())
            x.keys.pop(j)
            x.children.pop(i)
            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)
            lsNode = x.children[j]
            cNode.keys.insert(0, x.keys[i - 1])
            x.keys[i - 1] = lsNode.keys.pop()
                cNode.children.insert(0, lsNode.children.pop())
    tree = BTree()
   test()
from tkinter.filedialog import asksaveasfilename, askopenfilename
```

```
from pydantic import ValidationError
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)
        key = int(entry key.get())
       window.update()
    key = get key()
        node.keys[index] = key, entry value.get()
    key = get key()
        entry value.insert(0, node.keys[index][1])
        window.update()
    key = get_key()
        tree.delete(key)
       messagebox.showerror("Value error", "Key has not found in tree!")
        window.update()
```

```
tree = core.BTree()
     filepath = askopenfilename(
         raw data = input file.read()
     filepath = asksaveasfilename(
     if not filepath:
         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)
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 i 3.2 показані приклади роботи програми для додавання і пошуку запису.

Рисунок 3.1 –Додавання запису

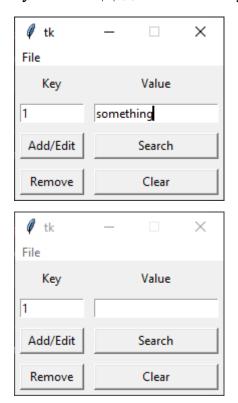


Рисунок 3.2 – Пошук запису



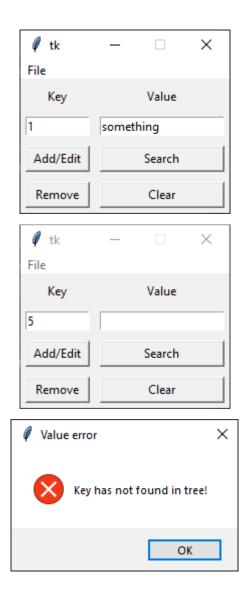


Рисунок 3.3 – Редагування запису

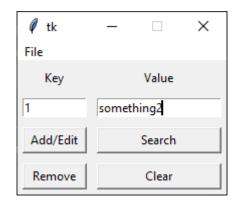
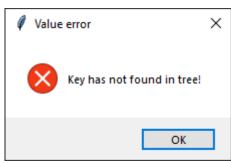


Рисунок 3.4 – Виделення запису







## 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	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 додаванн, 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.