**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-25 Карпов Л.В.*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.Н.*

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 8](#_Toc114359765)

[3.3 Програмна реалізація 9](#_Toc114359766)

[3.3.1 Вихідний код 9](#_Toc114359767)

[3.3.2 Приклади роботи 16](#_Toc114359768)

[3.4 Тестування алгоритму 19](#_Toc114359769)

[3.4.1 Часові характеристики оцінювання 19](#_Toc114359770)

[Висновок 20](#_Toc114359771)

[Критерії оцінювання 21](#_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

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)

## Програмна реалізація

### 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)  
  
 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()

### Приклади роботи

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

Рисунок 3.1 –Додавання запису

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

Рисунок 3.2 – Пошук запису

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer error

Description automatically generated

Рисунок 3.3 – Редагування запису

A screenshot of a computer program

Description automatically generated

Рисунок 3.4 – Виделення запису

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer error

Description automatically generated

## 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 | 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 B-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.