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-12 Мельник М.О.* ____________
(шифр, прізвище, ім'я, по батькові)

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

Київ 2022

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ ... 3

2 ЗАВДАННЯ ... 4

3 ВИКОНАННЯ ... 7

3.1 Псевдокод алгоритмів ... 7

3.2 Часова складність пошуку ... 9

3.3 Програмна реалізація ... 10

3.3.1 Вихідний код ... 10

3.3.2 Приклади роботи ... 13

3.4 Тестування алгоритму ... 14

3.4.1 Часові характеристики оцінювання ... 14

ВИСНОВОК ... 15

КРИТЕРІЇ ОЦІНЮВАННЯ ... 16

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	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	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

Варіант 18

3.1 Псевдокод алгоритмів

Кожен ключ вершини є кортежем, де перший елемент є власне ключем, а всі наступні — значення, які відповідають цьому ключеві. Під key варто розуміти власне цей кортеж, під k — лише перший елемент.

```
CLASS Node:
   keys: array
   child: array
   leaf: boolean=child NOT EMPTY

t = 25

max_keys = t*2-1
min_keys = t-1

PROCEDURE Search(k, node, parent)
   n = length(node.keys)
    IF n mod 2 == 0 append INF to node.keys
    i = ceildiv(n, 2)
    d = floordiv(n, 2)
   WHILE d >0:
      IF node.keys[i-1][0] == k:
         RETURN node, parent, i-1, node.keys[i-1]
      ELSE IF node.keys[i-1][0] < k:
         i += ceildiv(d, 2)
      ELSE:
         i -= ceildiv(d, 2)
      d = floordiv(d, 2)
      IF node.keys[i-1][0] == k:
         return node, parent, i-1, node.keys[i-1]

      IF node.leaf: return None
      ELSE:
         IF keys[i-1][0] > k:
            RETURN Search(k, node.child[i-1], node)
         ELSE
            RETURN Search(k, node.child[i], node)

PROCEDURE Edit(key):
   result = Search(key[0], root)
   if result:
      node = result[0]
      i = result[2]
      node.keys[i] = key

PROCEDURE Insert(key):
   IF length(root.keys) != max_keys:
      InsertNode(root, key)
   ELSE:
      --Split root--
      INIT new_root type Node
      new_root.child.append(root)
      SplitChild(new_root, 0)
      root = new_root
      Insert(key)

PROCEDURE InsertNode(node, key):
   i = length(node.keys) - 1
```

```
  WHILE i >= 0 AND node.keys[i][0] >= key[0]
    i -= 1

  IF node.leaf: node.keys.insert(i+1, key)
  ELSE:
    --Split child i+1 if exceed max n of keys--
    IF length(node.child[i+1].keys) == max_keys:
      SplitChild(node, i+1)
      IF node.keys[i+1][0] < key[0]:
        i += 1
    InsertNode(node.child[i+1], key)

PROCEDURE SplitChild(parent, i):
  --Split child i in node parent in two children--
  INIT new_child TYPE Node
  half_max = floordiv(max_keys, 2)
  child = parent.child[i]
  middle = child.keys[half_max]

  new_child.keys = child.keys[half_max+1]
  child.keys = child.keys[:half_max]

  IF NOT child.leaf:
    new_child.child = child.child[half_max+1:]
    child.child = child.child[:half_max+1]

  parent.keys.insert(i, middle)
  parent.child.insert(i+1, new_child)


PROCEDURE Delete(k)
  result = Search(k, root)
  IF result:
    node = result[0]
    parent = result[0]
  ELSE:
    RETURN

  i = DeleteInNode(node, k)
  IF node.leaf:
    IF length(node.keys) < min_keys:
      i = parent.child.index(node)
      IF left sibling exists and has enough keys ( > min_keys):
        borrow key to node from parent and to parent from left sibling:
          node.keys.insert(0, parent.keys.pop(i-1))
              parent.keys.insert(i-1, parent.child[i-1].keys.pop())
      ELSE:
        IF right sibling exists and has enough keys:
          borrow key to node from parent and to parent from right sibling:
            node.keys.append(parent.keys.pop(i))
                parent.keys.insert(i, parent.child[i+1].keys.pop(0))
        ELSE IF right sibling doesn't exist:
          merge node with left sibling and one key in parent:
            node.keys = parent.child[i-1].keys + [parent.keys.pop(i-1)] + node.keys
                parent.child.pop(i-1)
        ELSE:
          merge node with rigth sibling and one key in parent:
            node.keys = node.keys + [parent.keys.pop(i)] + parent.child[i+1].keys
                parent.child.pop(i+1)
  ELSE:
    find the righteestmost descendant of left sibling:
      sibling = node.child[i]
      WHILE NOT sibling.leaf:
```

```
        sibling = sibling.child[-1]
    -- if we can borrow a key, borrow --
    IF length(sibling.keys) > min_keys:
      node.keys.insert(i, sibling.keys.pop())
    ELSE:
      find the leftesetmost descendant of right sibling and its parent:
        parent = node
        sibling = node.child[i+1]
        while not sibling.leaf:
          parent = sibling
          sibling = parent.child[0]
      -- if we can borrow a key, borrow --
      IF length(sibling.keys) > min_keys:
        node.keys.insert(i, sibling.keys.pop(0))
      ELSE:
        -- if it turns out siblings are all leafs, merge them
        IF parent == node:
          node.child[i].keys += node.child[i+1].keys
          node.child[i].child += node.child[i+1].child
          node.child.pop(i+1)
        ELSE:
          node.keys.insert(i, sibling.keys.pop(0))
          -- if we can't borrow a key from leftestmost descendant of right sibling, but there is a spare key in the node after leftestmost descendant, borrow this spare key to its parent, borrow a key from parent to leftestmost descendant, borrow a key from leftestmost descendant to node, else merge those two descendants after borrowing a key to node
          IF length(parent.child[1].keys) > min_keys:
            sibling.keys.append(parent.keys.pop(0))
            parent.keys.insert(i, parent.child[1].keys.pop(0))
          ELSE:
            sibling.keys = sibling.keys + [parent.keys.pop(0)] + parent.child[1].keys
            parent.child.pop(1)

PROCEDURE DeleteInNode(node, k):
  i = 0
  WHILE i < length(node.keys):
    IF node.keys[i][0] == k:
      node.keys.pop(i)
      RETURN i
    i += 1
```

3.2 Часова складність пошуку

Під час пошуку у B-дереві ми проходимо від кореня до вершини, яка містить або не містить шуканий ключ. У найгіршому випадку ми проходимо $h = 0(\log_{\mathrm{t}} n)$ вершин, у середньому випадку ми також проходимо $h = 0(\log_{\mathrm{t}} n)$ вершин, адже у B-дереві, зрозуміло, більшість вершин є листковими. Для знаходження наступної вершини для пошуку, ми використовуюємо однорідний бінарний пошук, часова складність якого $O(n) = O(\log_2 n), n \in [t, 2t] \Rightarrow O(t) = O(\log_2 t)$. Таким чином, часова складність пошуку у B-дереві $O(n) = O(h \cdot \log_2 t) = O(\log_t n \cdot \log_2 t) = O(\ln t \ln n)$.

3.3 Програмна реалізація

3.3.1 Вихідний код

```
import random

class Node:
  def __init__(self):
   self.keys = []
   self.child = []

  @property
  def leaf(self):
   return not self.child


class BTree:
  def __init__(self, t):
   self.t = t
   self.min_keys = t - 1
   self.max_keys = 2 * t - 1

   self.root = Node()

   self.comps = 0

  def insert(self, key):
   if len(self.root.keys) != self.max_keys:
     self._insert_in_node(self.root, key)
   else:
     new_root = Node()
     new_root.child.append(self.root)
     self._split_child(new_root, 0)
     self.root = new_root
     self.insert(key)

  def _insert_in_node(self, node, key):
   # find index to be inserted
   i = len(node.keys) - 1
   while i >= 0 and node.keys[i][0] >= key[0]:
     i -= 1

   if node.leaf:
     node.keys.insert(i + 1, key)
   else:
     if len(node.child[i+1].keys) == self.max_keys:
      self._split_child(node, i + 1)
      if node.keys[i+1][0] < key[0]:
        i += 1
     self._insert_in_node(node.child[i+1], key)

  def _split_child(self, parent, i):
   new_child = Node()
   half_max = self.max_keys // 2
   child = parent.child[i]
   middle = child.keys[half_max]

   new_child.keys = child.keys[half_max+1:]
   child.keys = child.keys[:half_max]

   if not child.leaf:
     new_child.child = child.child[half_max+1:]
```

```
   child.child = child.child[:half_max+1]

  parent.keys.insert(i, middle)
  parent.child.insert(i+1, new_child)

 def search(self, k):
  r = self._search_in_node(k, self.root)
  return r[-1][1:] if r else r

 def _search_in_node(self, k, node, parent=None):
  keys = list(node.keys)
  n = len(keys)
  if not n % 2:
   keys.append((float('inf'), 0))
   n += 1
  i = n // 2 + int(n % 2)  # ceil
  d = n // 2  # floor
  while d:
   self.comps += 1
   if keys[i - 1][0] == k:
    return node, parent, i-1, node.keys[i - 1]
   elif keys[i - 1][0] < k:
    i = i + d // 2 + int(d % 2)
   else:
    i = i - d // 2 - int(d % 2)
   d = d // 2
  if keys[i - 1][0] == k:
   self.comps += 1
   return node, parent, i-1, node.keys[i - 1]
  if node.leaf:
   return None
  else:
   if keys[i-1][0] > k:
    self.comps += 1
    return self._search_in_node(k, node.child[i-1], node)
   else:
    return self._search_in_node(k, node.child[i], node)

 def edit(self, key):
  r = self._search_in_node(key[0], self.root)
  if r:
   node, _, i, _ = r
   node.keys[i] = key
   return True
  else:
   return None

 def delete(self, k):
  r = self._search_in_node(k, self.root)
  if r:
   node, parent, _, _ = r
  else:
   return False

  i = self._delete_in_node(node, k)

  if node.leaf:
   if len(node.keys) < self.min_keys:  # if now number of keys violates properties
    i = parent.child.index(node)
    # if the left sibling exists and has enough keys, swap: left sibling -> parent -> node
    if i != 0 and len(parent.child[i-1].keys) > self.min_keys:
     node.keys.insert(0, parent.keys.pop(i - 1))
     parent.keys.insert(i - 1, parent.child[i - 1].keys.pop())
```

```
   else:
     # if the right sibling exists and has enough keys, swap: right sibling -> parent -> node
     if i != len(parent.child) - 1 and len(parent.child[i + 1].keys) > self.min_keys:
      node.keys.append(parent.keys.pop(i))
      parent.keys.insert(i, parent.child[i + 1].keys.pop(0))
     # elif right sibling doesnt exist (and left sibling has not enough keys),
     # merge node with left sibling
     elif i == len(parent.child) - 1:
      node.keys = parent.child[i - 1].keys + [parent.keys.pop(i - 1)] + node.keys
      parent.child.pop(i - 1)

     # else right sibling exists, but has not enough keys, merge with right sibling
     else:
      node.keys = node.keys + [parent.keys.pop(i)] + parent.child[i + 1].keys
      parent.child.pop(i + 1)
 else:
  sibling = node.child[i]  # work with left sibling
  while not sibling.leaf:  # find the rightest descendant of the left sibling
   sibling = sibling.child[-1]

  if len(sibling.keys) > self.min_keys:  # if we can borrow a key, do it
   node.keys.insert(i, sibling.keys.pop())
  else:
   parent = node  # keep track of a parent
   sibling = node.child[i+1]  # work with right sibling
   while not sibling.leaf:  # find the leftest descendant of the right sibling
     parent = sibling
     sibling = parent.child[0]

   if len(sibling.keys) > self.min_keys:  # if can borrow, do it
     node.keys.insert(i, sibling.keys.pop(0))
   else:
     if parent == node:  # if the immediate right sibling of node is leaf, merge the left and right
      node.child[i].keys += node.child[i+1].keys
      node.child[i].child += node.child[i+1].child
      node.child.pop(i+1)
     else:
      node.keys.insert(i, sibling.keys.pop(0))
      # if the leftest descendant of right sibling has not enough keys and the descendant which comes
      # after him has enough keys, swap: child[1] -> parent -> child[0]
      if len(parent.child[1].keys) > self.min_keys:
        sibling.keys.append(parent.keys.pop(0))
        parent.keys.insert(i, parent.child[1].keys.pop(0))
      else:  # else merge them two
        sibling.keys = sibling.keys + [parent.keys.pop(0)] + parent.child[1].keys
        parent.child.pop(1)
 return True

def _delete_in_node(self, node, k):
 for i, key in enumerate(node.keys):
  if key[0] == k:
   node.keys.pop(i)
   return i

def __repr__(self):
 def _print(x, l):
  r = "  " * l + str([a[0] for a in x.keys])[1:-1] + "\n"
  for child in x.child:
   r += _print(child, l+1)
  return r
 return _print(self.root, 0)

def insert_random_values(self):
```

```
    values = list(range(10000))
    random.shuffle(values)
    for value in values:
      self.insert((value, float('inf')))

  def start_testing(self):
    for i in range(1, 11):
      self.search(100*i)
      print(f"Search number: {i}\nComparisons: {self.comps}")
      self.comps = 0
```

3.3.2 Приклади роботи

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

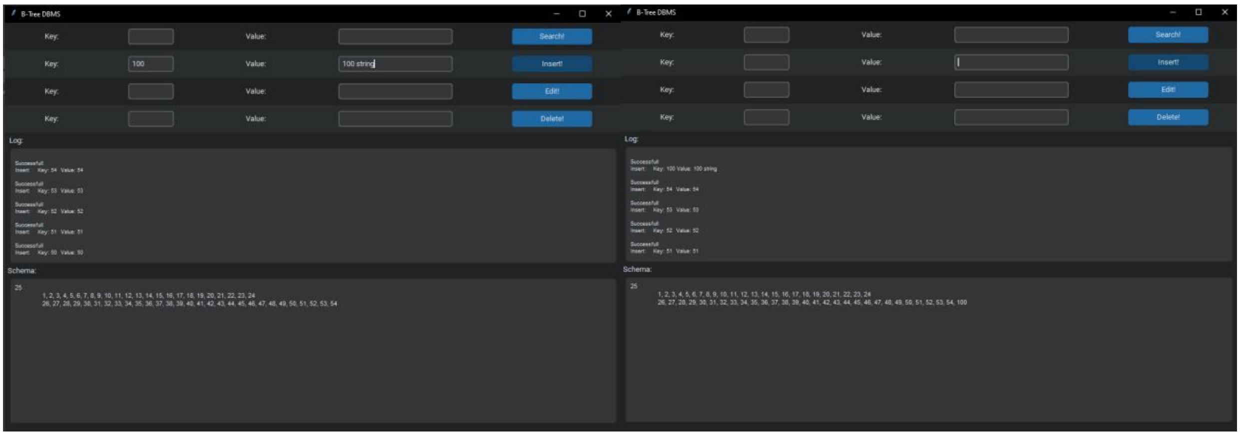

Рисунок 3.1 –Додавання запису

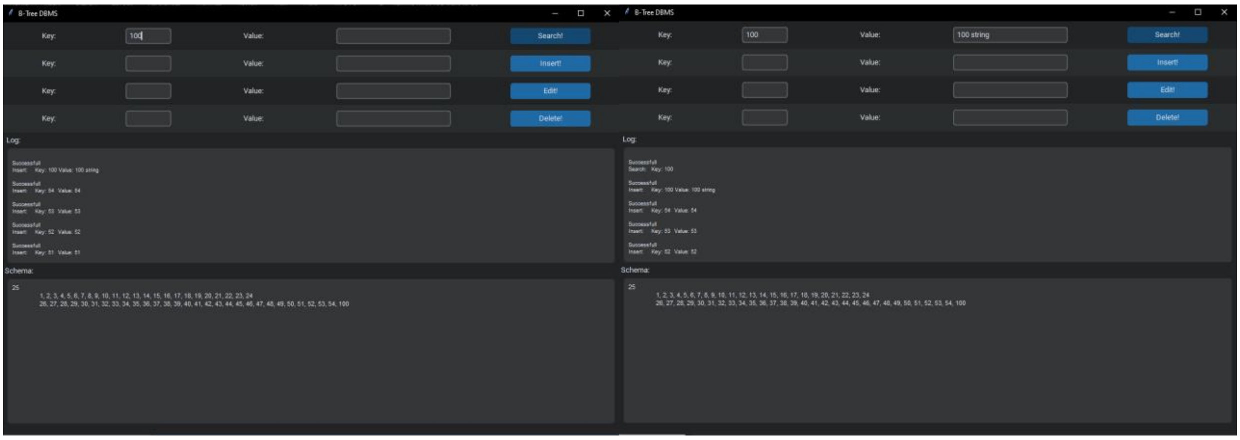

Рисунок 3.2 – Пошук запису

3.4 Тестування алгоритму

3.4.1 Часові характеристики оцінювання

В таблиці 3.1 наведено кількість порівнянь для 15 спроб пошуку запису по ключу.

Таблиця 3.1 – Число порівнянь при спробі пошуку запису по ключу

Номер спроби пошуку	Число порівнянь
1	15
2	13
3	14
4	15
5	13
6	10
7	13
8	12
9	14
10	13
Середнє	13,2

ВИСНОВОК

В рамках лабораторної роботи було записано алгоритми пошуку, додавання, видалення і редагування запису у B-дереві із t=25 за допомогою псевдокоду. Була записана часова складність пошуку в асимптотичних оцінках, яка склала $O(n) = O(\ln t \ln n)$.

Було виконано програмну реалізацію невеликої СУБД з графічним інетрфейсом користувача з функціями пошуку, додавання, видалення та редагування записів використовучи B-дерево із t=25.

Було заповнено базу випадковими значеннями до 10000 і зафіксовано середнє число порівнянь, яке склало 13,2.

Було зроблено висновок, що B-дерево є доволі ефективним способом зберігання даних, особливо якщо доступ до них здійснюється фізичними блоками, завдяки своїй малій висоті і збалансованості, які підтримуються під час всіх операцій зі зміни даних в дереві.

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.