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-14 Качмар Андрій* _____
(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 7

3.3 Програмна реалізація 10

3.3.1 Вихідний код 10

3.3.2 Приклади роботи 16

3.4 Тестування алгоритму 17

3.4.1 Часові характеристики оцінювання 17

ВИСНОВОК 18

КРИТЕРІЇ ОЦІНЮВАННЯ 19

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	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

3 ВИКОНАННЯ

3.1 Псевдокод алгоритмів

```
binarySearchNode(int key)
   firstPos = 0
   lastPos = numberOfkeys - 1
   resultPos = -1
   while (firstPos <= lastPos)
     middlePos = firstPos + (lastPos-firstPos)/2
     if (key at middlePos < key)
        firstPos = middlePos + 1
     else if (key at middlePos > key)
        lastPos = middlePos - 1
     else
        result = middlePos
     end if
   end while
return resultPos

insertKey(key, value)
   if(root.size = T*2-1)
      newNode = new Node(treeLevel)
      root = newNode
      root.setChild(0, currentRoot)
      splitNode(root at index 0)
      insertNode(newNode, key, value)
   else
      insertNode(root, key, value)
   end if
```

```
removeKey(treeNode,int key)
   if (treeNode.isLeaf)
       removeLeaf(treeNode, key)
   else
       if(binarySearchNode(key)!=-1)
          if(leftChild.length >= treeLevel)
             removePredecessor()
          else if(rightChild.length >= treeLevel)
             removeSuccessor()
          else
             joinChilds(leftChild, rightChild, treeNode)
             replaceMedian(treeNode)
          end if
       else
          childNode = treeNode at i
          if(leftSibling.length >= treeLevel)
             replaceWithLeftSibling()
          else if(rightChild.length >= treeLevel)
             replaceWithRightSibling ()
          else
             mergeSiblings(leftChild, rightChild)
             removeNode(childNode)
          end if
       end if
   end if
```

3.2 Часова складність пошуку

```
public TreeNodeEntry searchEntry(BTreeNode treeNode, int key) {  O(log n)
    int entryIndex = treeNode.getKeyWithBinarySearch(key);  O(log n)
    if (entryIndex != -1) {  O(1)
        return treeNode.getEntryAtIndex(entryIndex);
    }
    if (!treeNode.isLeafNode()) {  O(1)
        int childIndex = treeNode.getNextChildIndex(key);
        return searchEntry(treeNode.getChildAtIndex(childIndex), key);  O(log n)
    } else {
        return null;
    }
}
```

Часова складність пошуку в B дереві складає $O(\log n)$ де n визначається як кількість відсортованих ключів в дереві. Для пошуку ключа у вузлі дерева використовується бінарний пошук складність якого також $O(\log n)$.

Програмна реалізація

3.2.1 Вихідний код

```
public void insertEntry(TreeNodeEntry entry) {
    BTreeNode currentRoot = this.rootNode;
    if (!isUpdatePosition(this.rootNode, entry)) {
        if (currentRoot.isKeysFull()) {
            BTreeNode newNode = new BTreeNode(treeLevel, isLeafNode: false);
            this.rootNode = newNode;
            this.rootNode.setChildAtIndex( index: 0, currentRoot);
            splitNode(newNode, index: 0, currentRoot);
            insertNode(newNode, entry);
        } else {
            insertNode(currentRoot, entry);
        }
    }
}

private void splitNode(BTreeNode parentNode, int index, BTreeNode treeNode) {
    BTreeNode newNode = new BTreeNode(treeLevel, treeNode.isLeafNode());
    newNode.setNumberOfKeys(treeLevel - 1);
    copyNodeKeys(newNode, treeNode);
    if (!newNode.isLeafNode()) {
        copyChildNodes(newNode, treeNode);
    }
    treeNode.clearKeyValue();
    treeNode.setNumberOfKeys(treeLevel - 1);

    for (int i = parentNode.getNumberOfKeys(); i >= index + 1; i--) {
        parentNode.setChildAtIndex( index: i + 1, parentNode.getChildAtIndex(i));
    }
    parentNode.setChildAtIndex( index: index + 1, newNode);

    for (int i = parentNode.getNumberOfKeys() - 1; i >= index; i--) {
        parentNode.setEntryAtIndex( index: i + 1, parentNode, i);
    }
    parentNode.setEntryAtIndex(index, treeNode, nodeIndex: treeLevel - 1);
    treeNode.clearEntryByIndex(treeLevel - 1);
    parentNode.increaseNumberOfKeys();
}
```

```java
private void copyNodeKeys(BTreeNode nodeTo, BTreeNode nodeFrom) {
    for (int i = 0; i < treeLevel - 1; i++) {
        nodeTo.setEntryAtIndex(i, nodeFrom,  nodeIndex: i + treeLevel);
    }
}

private void copyChildNodes(BTreeNode nodeTo, BTreeNode nodeFrom) {
    for (int i = 0; i < treeLevel; i++) {
        nodeTo.setChildAtIndex(i, nodeFrom.getChildAtIndex(i + treeLevel));
    }
    nodeFrom.clearChild();
}

private void insertNode(BTreeNode treeNode, TreeNodeEntry entry) {
    int writeEntryIndex = treeNode.getNumberOfKeys() - 1;
    if (treeNode.isLeafNode()) {
        insertEntryToLeaf(treeNode, entry);
    } else {
        writeEntryIndex = getWritePositionInCurrentNode(entry, treeNode, writeEntryIndex);
        if (treeNode.isChildFull(writeEntryIndex)) {
            splitNode(treeNode, writeEntryIndex, treeNode.getChildAtIndex(writeEntryIndex));
            if (entry.getKey() > treeNode.getKeyAtIndex(writeEntryIndex)) {
                writeEntryIndex++;
            }
        }
        insertNode(treeNode.getChildAtIndex(writeEntryIndex), entry);
    }
}
```

```java
private void insertEntryToLeaf(BTreeNode treeNode, TreeNodeEntry entry) {
    int writeEntryIndex = treeNode.getNumberOfKeys() - 1;
    while (writeEntryIndex >= 0 && entry.getKey() < treeNode.getKeyAtIndex(writeEntryIndex)) {
        treeNode.setEntryAtIndex( index: writeEntryIndex + 1, treeNode, writeEntryIndex);
        writeEntryIndex--;
    }
    writeEntryIndex++;
    treeNode.setEntryAtIndex(writeEntryIndex, treeNode, entry);
    treeNode.increaseNumberOfKeys();
}

private int getWritePositionInCurrentNode(TreeNodeEntry entry, BTreeNode treeNode, int writeEntryIndex) {
    while (writeEntryIndex >= 0 && entry.getKey() < treeNode.getKeyAtIndex(writeEntryIndex)) {
        writeEntryIndex--;
    }
    writeEntryIndex++;
    return writeEntryIndex;
}


public void removeEntry(int key) {
    removeNode(rootNode, key);
}

private void removeNode(BTreeNode treeNode, int key) {
    if (treeNode.isLeafNode()) {
        removeLeaf(treeNode, key);
    } else {
        int i = treeNode.getKeyWithBinarySearch(key);
        findAndRemoveNode(treeNode, key, i);
    }
}
```

```java
private void removeLeaf(BTreeNode treeNode, int key) { //1
    int removeIndex = treeNode.getKeyWithBinarySearch(key);
    if (removeIndex != -1) {
        treeNode.removeChildIndex(removeIndex, LEFT_CHILD_DEFAULT_INDEX);
    }
}

private void findAndRemoveNode(BTreeNode treeNode, int key, int i) {
    if (i != -1) {
        findAndReplaceWithPredecessorOrSuccessor(treeNode, key, i); //2
    } else {
        i = treeNode.getSubtreeNodeIndex(key);
        BTreeNode childNode = treeNode.getChildAtIndex(i);
        findAndRemoveWithSiblings(treeNode, i, childNode);
        removeNode(childNode, key);
    }
}

private void findAndRemoveWithSiblings(BTreeNode treeNode, int i, BTreeNode childNode) {
    if (childNode.hasLessThanMax()) {
        BTreeNode leftSibling = treeNode.getLeftChildSibling(i);
        BTreeNode rightSibling = treeNode.getRightChildSibling(i);
        if (leftSibling != null && leftSibling.hasMoreThanMax()) {
            replaceWithLeftSibling(treeNode, i, childNode, leftSibling);
        } else if (rightSibling != null && rightSibling.hasMoreThanMax()) {
            replaceWithRightSibling(treeNode, i, childNode, rightSibling);
        } else {
            mergeSiblingsAndMoveRoot(treeNode, i, childNode, leftSibling, rightSibling);
        }
    }
}
```

```java
private void findAndReplaceWithPredecessorOrSuccessor(BTreeNode treeNode, int key, int i) { // 2
    BTreeNode leftChild = treeNode.getChildAtIndex(i);
    BTreeNode rightChild = treeNode.getChildAtIndex(i + 1);
    if (leftChild.hasMoreThanMax()) { //2a predecessor
        movePredecessorOrSuccessor(treeNode, i, isSuccessor: false, leftChild);
    } else if (rightChild.hasMoreThanMax()) { // 2b successor
        movePredecessorOrSuccessor(treeNode, i, isSuccessor: true, rightChild);
    } else {
        joinChildAndReplaceMedian(leftChild, rightChild, treeNode, i, key);
    }
}

private void joinChildAndReplaceMedian(BTreeNode leftChild, BTreeNode rightChild, BTreeNode treeNode, int index, int key) {
    int medianKeyIndex = mergeChildNodes(leftChild, rightChild);
    moveKeyAndInsertIntoMedian(treeNode, index, RIGHT_CHILD_DEFAULT_INDEX, leftChild, medianKeyIndex);
    removeNode(leftChild, key);
}

private void replaceWithRightSibling(BTreeNode treeNode, int i, BTreeNode childToNode, BTreeNode childFromNode) {
    childToNode.setEntryAtIndex(childToNode.getNumberOfKeys(), treeNode, i);
    if (!childToNode.isLeafNode()) {
        childToNode.setChildAtIndex( index: childToNode.getNumberOfKeys() + 1, childFromNode.getChildAtIndex(0));
    }
    childToNode.increaseNumberOfKeys();
    treeNode.setEntryAtIndex(i, childFromNode, nodeIndex: 0);
    childFromNode.removeChildIndex( index: 0, LEFT_CHILD_DEFAULT_INDEX);
}
```

```java
private void mergeSiblingsAndMoveRoot(BTreeNode treeNode, int i, BTreeNode childNode, BTreeNode leftSibling, BTreeNode rightSibling) {
    if (leftSibling != null) {
        int medianKeyIndex = mergeChildNodes(childNode, leftSibling);
        moveKeyAndInsertIntoMedian(treeNode, fromKeyIndex: i - 1, LEFT_CHILD_DEFAULT_INDEX, childNode, medianKeyIndex);
    } else if (rightSibling != null) {
        int medianKeyIndex = mergeChildNodes(childNode, rightSibling);
        moveKeyAndInsertIntoMedian(treeNode, i, RIGHT_CHILD_DEFAULT_INDEX, childNode, medianKeyIndex);
    }
}

private void movePredecessorOrSuccessor(BTreeNode treeNode, int insertIndex, boolean isSuccessor, BTreeNode child) {
    BTreeNode nodeForMove = child;
    BTreeNode nodeForDelete = nodeForMove;
    while (!nodeForMove.isLeafNode()) {
        nodeForDelete = nodeForMove;
        int childIndex = isSuccessor ? 0 : (treeNode.getNumberOfKeys() - 1);
        nodeForMove = nodeForMove.getChildAtIndex(childIndex);
    }
    int entryIndex = isSuccessor ? 0 : (nodeForMove.getNumberOfKeys() - 1);
    treeNode.setEntryAtIndex(insertIndex, nodeForMove, entryIndex);
    removeNode(nodeForDelete, treeNode.getKeyAtIndex(insertIndex));
}
```

```java
private void moveKeyAndInsertIntoMedian(BTreeNode fromNode, int fromKeyIndex, int childIndex, BTreeNode toNode, int medianIndex) {
    toNode.setEntryAtIndex(medianIndex, fromNode, fromKeyIndex);
    toNode.increaseNumberOfKeys();
    fromNode.removeChildIndex(fromKeyIndex, childIndex);
    if (fromNode == rootNode && fromNode.isEmpty()) {
        rootNode = toNode;
    }
}
```

```java
private int mergeChildNodes(BTreeNode mergeToNode, BTreeNode mergeFromNode) {
    int medianKeyIndex;
    if (mergeFromNode.getKeyAtIndex(0) < mergeToNode.getKeyAtIndex(mergeToNode.getNumberOfKeys() - 1)) {
        writeChildFromEnd(mergeToNode, mergeFromNode);
        medianKeyIndex = clearMedian(mergeToNode, mergeFromNode);
        writeChildFromStart(mergeToNode, mergeFromNode);
    } else {
        medianKeyIndex = clearMedian(mergeToNode, mergeToNode);
        writeChildFromStartByOne(mergeToNode, mergeFromNode, medianKeyIndex);
    }
    mergeToNode.addKeysLength(mergeFromNode.getNumberOfKeys());
    return medianKeyIndex;
}

private void writeChildFromStartByOne(BTreeNode mergeToNode, BTreeNode mergeFromNode, int medianKeyIndex) {
    int medianShift = medianKeyIndex + 1;
    int i;
    for (i = 0; i < mergeFromNode.getNumberOfKeys(); i++) {
        mergeToNode.setEntryAtIndex( index: medianShift + i, mergeFromNode, i);
        if (!mergeFromNode.isLeafNode()) {
            mergeToNode.setChildAtIndex( index: medianShift + i, mergeFromNode.getChildAtIndex(i));
        }
    }
    if (!mergeFromNode.isLeafNode()) {
        mergeToNode.setChildAtIndex( index: medianShift + i, mergeFromNode.getChildAtIndex(i));
    }
}
```

```java
private int clearMedian(BTreeNode mergeToNode, BTreeNode mergeFromNode) {
    int medianKeyIndex;
    medianKeyIndex = mergeFromNode.getNumberOfKeys();
    mergeToNode.clearEntryByIndex(medianKeyIndex);
    return medianKeyIndex;
}

private void writeChildFromStart(BTreeNode mergeToNode, BTreeNode mergeFromNode) {
    int i;
    for (i = 0; i < mergeFromNode.getNumberOfKeys(); i++) {
        mergeToNode.setEntryAtIndex(i, mergeFromNode);
        if (!mergeFromNode.isLeafNode()) {
            mergeToNode.setChildAtIndex(i, mergeFromNode.getChildAtIndex(i));
        }
    }
    if (!mergeFromNode.isLeafNode()) {
        mergeToNode.setChildAtIndex(i, mergeFromNode.getChildAtIndex(i));
    }
}

private void writeChildFromEnd(BTreeNode mergeToNode, BTreeNode mergeFromNode) {
    if (!mergeToNode.isLeafNode()) {
        int index = mergeFromNode.getNumberOfKeys() + mergeToNode.getNumberOfKeys() + 1;
        mergeToNode.setChildAtIndex(index, mergeToNode.getChildAtIndex(mergeToNode.getNumberOfKeys()));
    }
    for (int i = mergeToNode.getNumberOfKeys(); i > 0; i--) {
        mergeToNode.setEntryAtIndex( index: mergeFromNode.getNumberOfKeys() + i, mergeToNode,  nodeIndex: i - 1);
        if (!mergeToNode.isLeafNode()) {
            mergeToNode.setChildAtIndex( index: mergeFromNode.getNumberOfKeys() + i, mergeToNode.getChildAtIndex(i - 1));
        }
    }
}
```

3.2.2 Приклади роботи

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

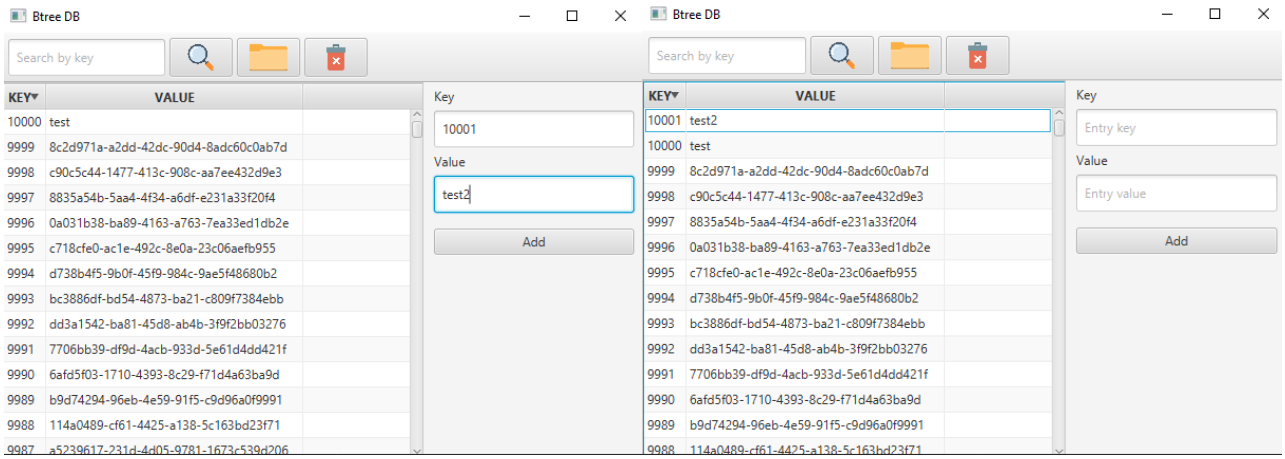

Рисунок 3.1 –Додавання запису

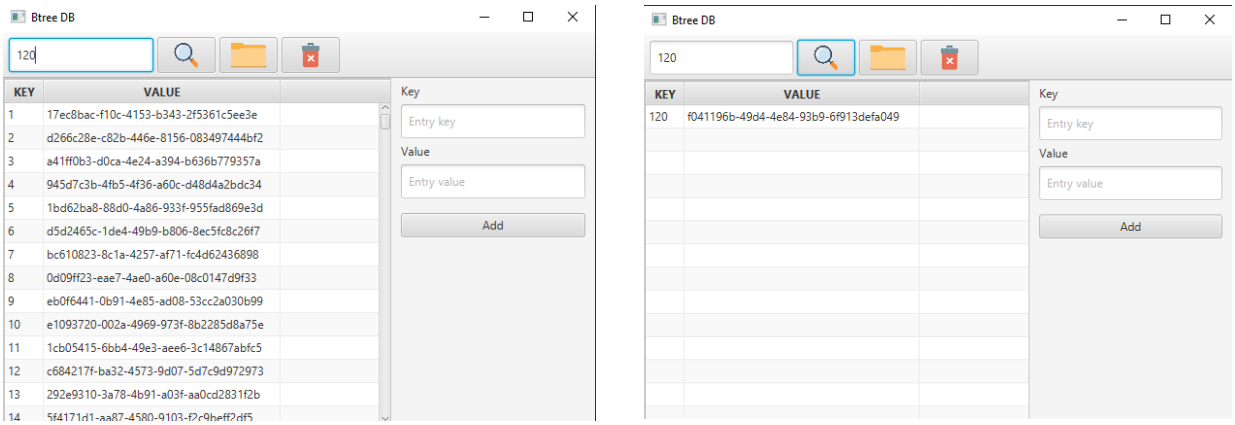

Рисунок 3.2 – Пошук запису

Тестування алгоритму

3.2.3 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	29
2	26
3	18
4	7
5	8
6	34
7	31
8	25
9	11
10	19
11	27
12	25
13	17
14	22
15	23

ВИСНОВОК

В рамках лабораторної роботи було виконано програмну реалізацію СУБД з графічним інтерфейсом користувача. Дані БД збережено на ПЗП. В створеній СУБД реалізовано функції пошуку, додавання, видалення та редагування записів. За основу БД використано B-дерево з рівнем 10. Для функції пошуку у вузлі дерева було використано бінарний пошук. Отже було вивчено основні підходи проектування та обробки складних структур даних.

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.