

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 № 2 з дисципліни
«Проектування алгоритмів»

„Проектування структур даних”

Виконав(ла)

ІП-01 Пашковський Євгеній Сергійович
(шифр, прізвище, ім'я, по батькові)

Перевірів

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

Київ 2021

ЗМІСТ

1	МЕТА ЛАБОРАТОРНОЇ РОБОТИ	3
2	ЗАВДАННЯ	4
3	ВИКОНАННЯ.....	5
3.1	ПСЕВДОКОД АЛГОРИТМІВ.....	5
3.2	ЧАСОВА СКЛАДНІСТЬ ПОШУКУ	5
3.3	ПРОГРАМНА РЕАЛІЗАЦІЯ	6
3.3.1	<i>Вихідний код</i>	<i>6</i>
3.3.2	<i>Приклади роботи</i>	<i>14</i>
3.4	ТЕСТУВАННЯ АЛГОРИТМУ	16
3.4.1	<i>Часові характеристики оцінювання.....</i>	<i>16</i>
	ВИСНОВОК	17
	КРИТЕРІЇ ОЦІНЮВАННЯ	18

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 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
19	В-дерево $t=50$, бінарний пошук

3.1 Псевдокод алгоритмів

Пошук:

```
find(key) {
  if (this.root.length === 0) return false;
  let node = this.root;
  const result = BinarySearch(key, node)
  if result return result
  if (!isLeaf(node)) {
    node = choseInterval(node);
  } else {
    return false;
  }
}
```

Вставка:

```
insert(key, data, node = this.root, prevNode = null) {
  if (this.root.length === 0) {
    this.root.push(new BNode(key, data));
    return this;
  }

  if (!this.#isLeaf(node)) {
    this.insert(key, data, chooseInterval(node), node);
  } else {
    node.add(BNode(key, data));
  }

  if (node.length === 2 * this.t - 1) {
    split(node);
  }
  return this;
}
```

Видалення:

```
remove(key, node = this.root, prevNode = null) {
  bnode = binarySearch(key, node);

  const index = node.indexOf(bnode);
  let result = false;
  if (bnode && !(bnode.left || bnode.right)) {
    node.remove(key)
  } else if (bnode) {
    if (
      leftNode.length === this.t - 1 &&
      rightNode.length === leftNode.length
    ) {
      merge()
      result = this.remove(key, newNode, node);
    } else if (leftLeaf.length >= this.t) {
      leftSwap()
      result = this.remove(key, swapBNode.left, node);
    } else if (rightLeaf.length >= this.t) {
      rightSwap()
      result = this.remove(key, swapBNode.right, node);
    }
  } else {
    result = this.remove(key, chooseInterval(node), node);
    break;
  }
}

if (prevNode && node.length <= this.t - 1) {
  let parentBNode = null;
```

```

    for (const bnode of prevNode) {
        if (bnode.left === node || bnode.right === node) {
            parentBNode = bnode;
            break;
        }
    }
    const parentIndex = prevNode.indexOf(parentBNode);

    let swapBNode = null;
    if (parentBNode.left === node && parentBNode.right.length > this.t - 1) {
        swap(parentBNode.right);
    } else if (
        parentBNode.right === node &&
        parentBNode.left.length > this.t - 1
    ) {
        swap(parentBNode.left);
    } else if (
        (parentBNode.left === node &&
        parentBNode.right.length === this.t - 1) ||
        (parentBNode.right === node && parentBNode.left.length === this.t - 1)
    ) {
        merge()
        if (this.root.length === 0) this.root = newNode;
    } else {
        result = false;
    }
}
return result;
}

```

3.2 Часова складність пошуку

Бінарний пошук: $O(\log(n))$

Пошук у В-дереві: $O(\log(n))$

Програмна реалізація

3.2.1 Вихідний код

```

'use strict';

const DB = require('./DB.js');
const GUI = require('./GUI.js');
const stdinput = require('./stdinput.js');
const path = '/database.dat';

class Lab2 {
    async start() {
        const db = new DB();
        const gui = new GUI();

        db.loadFromFile(path);

        let flag = true;
        const loadGui = async () => {
            const drawCurrentDB = () => {
                gui.sendMessage(`Current db: ${db.path}`);
            };
            const drawMenu = () => {
                gui.sendMessage(

```

```

    '1 - Insert data\n2 - Remove data\n3 - Update data\n4 - Find data\n5 -
Load data from file\n6 - Save to file and exit'
    );
};
const drawInsertData = () => {
    gui.sendMessage('Input data: ');
};
const drawInsertKey = () => {
    gui.sendMessage('Input key: ');
};
const drawInsertPath = () => {
    gui.sendMessage('Input path: ');
};
gui.addElementCallback(drawCurrentDB);
let mainMenu = gui.addElementCallback(drawMenu);

const getKey = async () => {
    gui.removeElementCallback(mainMenu);
    const keyInput = gui.addElementCallback(drawInsertKey);
    const result = await stdinput();
    mainMenu = gui.addElementCallback(drawMenu);
    gui.removeElementCallback(keyInput);
    if (result === '') return undefined;
    if (result === '0') return 0;
    return Number(result);
};

const getData = async () => {
    gui.removeElementCallback(mainMenu);
    const dataInput = gui.addElementCallback(drawInsertData);
    const result = await stdinput();
    mainMenu = gui.addElementCallback(drawMenu);
    gui.removeElementCallback(dataInput);
    return result;
};

const getPath = async () => {
    gui.removeElementCallback(mainMenu);
    const pathInput = gui.addElementCallback(drawInsertPath);
    const result = await stdinput();
    mainMenu = gui.addElementCallback(drawMenu);
    gui.removeElementCallback(pathInput);
    return result;
};

const data = await stdinput();

switch (data) {
    case '1':
        db.insert(await getData(), await getKey());
        break;
    case '2':
        db.remove(await getKey());
        break;
    case '3':
        db.update(await getKey(), await getData());
        break;
    case '4':
        gui.sendMessage(db.find(await getKey()));
        gui.sendMessage('Press enter...');
        await stdinput();
        break;
    case '5':
        db.saveToFile();
        db.loadFromFile(await getPath());
        break;
    case '6':
        gui.sendMessage('Press enter...');
        await stdinput(true);
        flag = false;
        break;
}

```

```

        default:
            gui.sendError('Unexpected input');
            await stdin();
            break;
        }
        gui.clearElementCallbacks();
        db.saveToFile();
        return flag;
    };

    while (flag) {
        flag = await loadGui();
    }
    gui.clear();
}
}

module.exports = Lab2;

```

```

'use strict';
const BTree = require('./BTree.js');
const fs = require('fs');

const t = 50;

class DB {
    #btree = new BTree(t);
    path = null;
    autoKey = 0;
    constructor(...data) {
        for (const item of data) {
            this.insert(this.autoKey, item);
            this.autoKey++;
        }
    }

    find(key) {
        return this.#btree.find(key).data;
    }

    insert(data = null, key = this.autoKey) {
        // console.log(key);
        if (this.#btree.find(key)) {
            if (key === this.autoKey) {
                this.autoKey++;
                return this.insert(data);
            } else {
                throw new Error('This key already exists in the db');
            }
        }

        this.#btree.insert(key, data);
        if (key === this.autoKey) this.autoKey++;
        return this;
    }

    remove(key) {
        if (!this.#btree.find(key)) throw new Error('This key is not in the db');
        this.#btree.remove(key);
        return this;
    }

    update(key, data) {
        const bnode = this.#btree.find(key);
        if (!bnode) throw new Error('This key is not in the db');
        bnode.data = data;
        return this;
    }

    loadFromFile(path = this.path) {
        this.#btree = new BTree(t);
    }
}

```



```

    try {
      const file = fs.readFileSync(__dirname + path, { encoding: 'utf-8' });
      const lines = file.split('\n');

      for (const line of lines) {
        this.insert(line);
      }
    } catch {
      fs.writeFileSync(__dirname + path, '');
    }

    this.path = path;
  }

  saveToFile(path = this.path) {
    fs.writeFile(__dirname + path, this.string, err => {
      if (err) throw new Error('Error while saving to file: ' + err);
      console.log('File written successfully');
      this.path = path;
    });
  }

  get string() {
    const res = [];
    for (let i = 0; i < this.autoKey; i++) {
      const data = this.find(i);
      if (!data) continue;
      res.push(data);
    }
    return res.join('\n');
  }
}

module.exports = DB;

```

```

'use strict';

const BNode = require('./BNode.js');

class BTree {
  constructor(t, key, rootData = null) {
    this.t = t;
    this.root = key ? [new BNode(key, rootData)] : [];
  }

  find(key) {
    if (this.root.length === 0) return false;
    let node = this.root;
    while (true) {
      let leftP = 0;
      let rightP = node.length - 1;

      while (true) {
        if (Math.abs(leftP - rightP) === 1 || leftP === rightP) {
          if (node[leftP].key === key) return node[leftP];
          if (node[rightP].key === key) return node[rightP];

          break;
        }
        const middleP = Math.floor((rightP + leftP) / 2);
        const middle = node[middleP];

        if (key === middle.key) {
          return middle;
        } else if (key < middle.key) {
          rightP = middleP;
        } else {
          leftP = middleP;
        }
      }
    }
  }
}

```

```

    if (!this.#isLeaf(node)) {
      for (let i = 0; i < node.length; i++) {
        const currBNode = node[i];
        const nextBNode = node[i + 1];

        if (i === 0 && key <= currBNode.key) {
          node = currBNode.left;
          break;
        }
        if (!nextBNode || (key >= currBNode.key && key <= nextBNode.key)) {
          node = currBNode.right;
          break;
        }
      }
    } else {
      return false;
    }
  }
}

insert(key, data, node = this.root, prevNode = null) {
  if (this.root.length === 0) {
    this.root.push(new BNode(key, data));
    return this;
  }

  if (!this.#isLeaf(node)) {
    for (let i = 0; i < node.length; i++) {
      const currBNode = node[i];
      const nextBNode = node[i + 1];

      if (i === 0 && key <= currBNode.key) {
        this.insert(key, data, currBNode.left, node);
        break;
      }
      if (!nextBNode || (key >= currBNode.key && key <= nextBNode.key)) {
        this.insert(key, data, currBNode.right, node);
        break;
      }
    }
  } else {
    let i = 0;
    while (i < node.length && key > node[i].key) {
      i++;
    }
    node.splice(i, 0, new BNode(key, data));
  }

  if (node.length === 2 * this.t - 1) {
    const nodeL = node.slice(0, this.t - 1);
    const middleNode = node.slice(this.t - 1, this.t);
    const nodeR = node.slice(this.t);

    const middleItem = middleNode[0];
    middleItem.left = nodeL;
    middleItem.right = nodeR;
    if (!prevNode) {
      this.root = middleNode;
    } else {
      let i = 0;
      while (i < prevNode.length && key > prevNode[i].key) {
        i++;
      }
      prevNode.splice(i, 0, middleItem);
      for (const bnode of prevNode) {
        if (bnode.left.includes(middleItem)) {
          bnode.left = nodeR;
        }

        if (bnode.right.includes(middleItem)) {

```

```

        bnode.right = nodeL;
    }
}
}
return this;
}

remove(key, node = this.root, prevNode = null) {
    let leftP = 0;
    let rightP = node.length - 1;

    let bnode = null;
    while (true) {
        if (Math.abs(leftP - rightP) === 1 || leftP === rightP) {
            if (node[leftP].key === key) {
                bnode = node[leftP];
            }
            if (node[rightP].key === key) {
                bnode = node[rightP];
            }

            break;
        }
        const middleP = Math.floor((rightP + leftP) / 2);
        const middle = node[middleP];

        if (key === middle.key) {
            bnode = middle;
            break;
        } else if (key < middle.key) {
            rightP = middleP;
        } else {
            leftP = middleP;
        }
    }

    const index = node.indexOf(bnode);
    let result = false;
    if (bnode && !(bnode.left || bnode.right)) {
        if (index >= 0) {
            node.splice(index, 1);
        }
    } else if (bnode) {
        const leftNode = bnode.left;
        const rightNode = bnode.right;

        let rightLeaf = bnode.right;
        while (true) {
            if (rightLeaf[0].left === null) break;
            rightLeaf = rightLeaf[0].left;
        }

        let leftLeaf = bnode.left;
        while (true) {
            if (leftLeaf[leftLeaf.length - 1].right === null) break;
            leftLeaf = leftLeaf[leftLeaf.length - 1].right;
        }

        if (
            leftNode.length === this.t - 1 &&
            rightNode.length === leftNode.length
        ) {
            const prevBNode = node[index - 1];
            const nextBNode = node[index + 1];

            node.splice(index, 1);

            let newNode = bnode.left.concat([bnode]).concat(bnode.right);
            bnode.left = newNode[newNode.indexOf(bnode) - 1].right || null;
            bnode.right = newNode[newNode.indexOf(bnode) + 1].left || null;
        }
    }
}

```

```

    if (prevBNode) prevBNode.right = newNode;
    if (nextBNode) nextBNode.left = newNode;
    if (node.length === 0) {
        node.push(...newNode);
        newNode = node;
    }
    result = this.remove(key, newNode, node);
} else if (leftLeaf.length >= this.t) {
    const swapBNode = leftLeaf.pop();

    node.splice(index, 1);
    leftLeaf.push(bnode);
    node.splice(index, 0, swapBNode);

    const buf = { left: bnode.left, right: bnode.right };
    bnode.left = swapBNode.left;
    bnode.right = swapBNode.right;
    swapBNode.left = buf.left;
    swapBNode.right = buf.right;
    result = this.remove(key, swapBNode.left, node);
} else if (rightLeaf.length >= this.t) {
    const swapBNode = rightLeaf.shift();

    node.splice(index, 1);

    rightLeaf.unshift(bnode);
    node.splice(index, 0, swapBNode);

    const buf = { left: bnode.left, right: bnode.right };
    bnode.left = swapBNode.left;
    bnode.right = swapBNode.right;
    swapBNode.left = buf.left;
    swapBNode.right = buf.right;
    result = this.remove(key, swapBNode.right, node);
}
} else {
    for (let i = 0; i < node.length; i++) {
        const currBNode = node[i];
        const nextBNode = node[i + 1];

        if (i === 0 && key <= currBNode.key) {
            result = this.remove(key, currBNode.left, node);
            break;
        }
        if (!nextBNode || (key >= currBNode.key && key <= nextBNode.key)) {
            result = this.remove(key, currBNode.right, node);
            break;
        }
    }
}

if (prevNode && node.length <= this.t - 1) {
    let parentBNode = null;
    for (const bnode of prevNode) {
        if (bnode.left === node || bnode.right === node) {
            parentBNode = bnode;
            break;
        }
    }
    const parentIndex = prevNode.indexOf(parentBNode);

    let swapBNode = null;
    if (parentBNode.left === node && parentBNode.right.length > this.t - 1) {
        swapBNode = parentBNode.right.shift();

        prevNode.splice(parentIndex, 1);
        const buf = { left: parentBNode.left, right: parentBNode.right };

        parentBNode.left = node[node.length - 1]?.right || null;
        parentBNode.right = swapBNode.left || null;
    }
}

```

```

        swapBNode.left = buf.left;
        swapBNode.right = buf.right;

        node.push(parentBNode);
        prevNode.splice(parentIndex, 0, swapBNode);
    } else if (
        parentBNode.right === node &&
        parentBNode.left.length > this.t - 1
    ) {
        swapBNode = parentBNode.left.pop();

        prevNode.splice(parentIndex, 1);
        node.unshift(parentBNode);
        const buf = { left: parentBNode.left, right: parentBNode.right };

        parentBNode.left = swapBNode.right;
        parentBNode.right = node[0].left;

        swapBNode.left = buf.left;
        swapBNode.right = buf.right;

        prevNode.splice(parentIndex, 0, swapBNode);
    } else if (
        (parentBNode.left === node &&
            parentBNode.right.length === this.t - 1) ||
        (parentBNode.right === node && parentBNode.left.length === this.t - 1)
    ) {
        const prevBNode = prevNode[parentIndex - 1];
        const nextBNode = prevNode[parentIndex + 1];

        prevNode.splice(parentIndex, 1);

        const newNode = parentBNode.left
            .concat([parentBNode])
            .concat(parentBNode.right);
        parentBNode.left =
            parentBNode.left[parentBNode.left.length - 1]?.right || null;
        parentBNode.right = parentBNode.right[0]?.left || null;
        if (prevBNode) prevBNode.right = newNode;
        if (nextBNode) nextBNode.left = newNode;
        if (prevNode.length === 0) prevNode.push(...newNode);
        if (this.root.length === 0) this.root = newNode;
    } else {
        result = false;
    }
    }
    return result;
}

#isLeaf(arr) {
    let flag = true;
    for (const el of arr) {
        if (el.left || el.right) {
            flag = false;
        }
    }
    return flag;
}

static isBTree(obj) {
    return obj instanceof BTree;
}
}

module.exports = BTree;

```

```
'use strict';
```

```

class BNode {
    constructor(key, data = null) {
        this.key = key;
    }
}

```

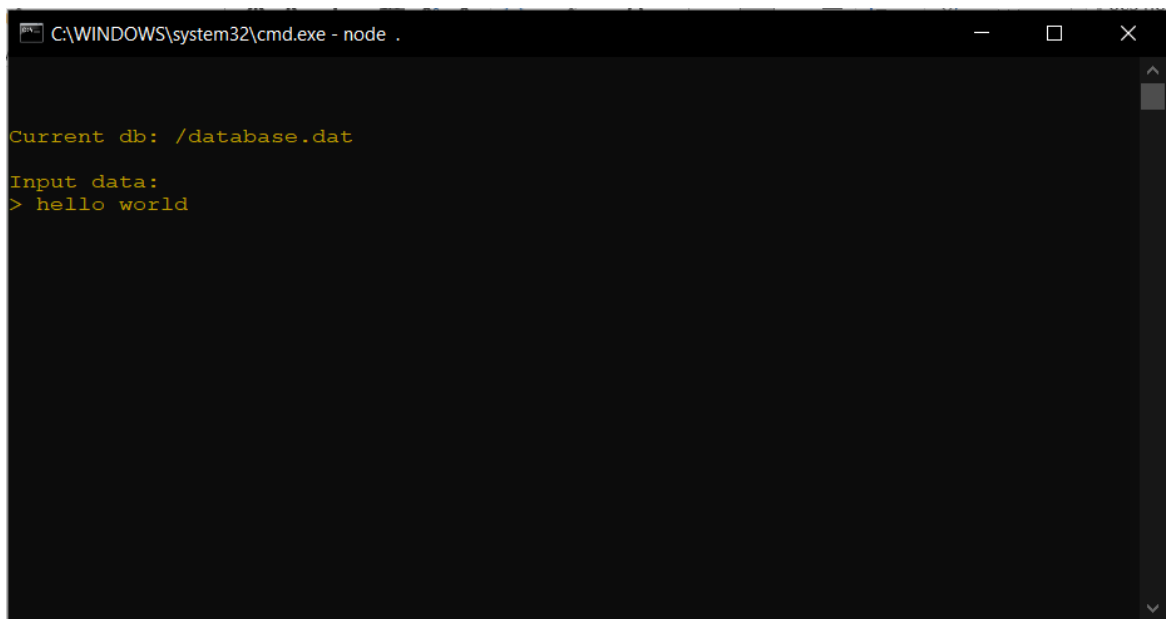
```
    this.data = data;
    this.left = null;
    this.right = null;
  }

  static isBNode(obj) {
    return obj instanceof BNode;
  }
}

module.exports = BNode;
```

3.2.2 Приклади роботи

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

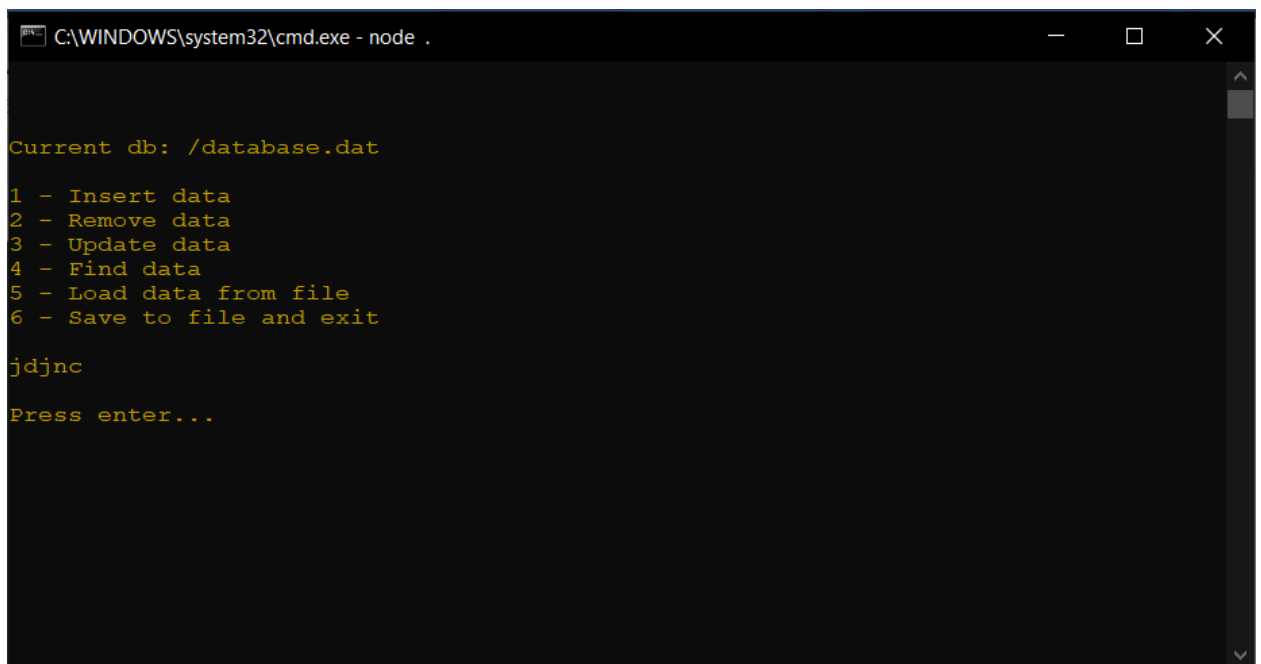
A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe - node .". The window has a dark background with yellow text. It shows the command "Current db: /database.dat" followed by "Input data:" and a prompt "> hello world".

```
C:\WINDOWS\system32\cmd.exe - node .

Current db: /database.dat

Input data:
> hello world
```

Рисунок 3.1 –Додавання запису

A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe - node .". The window has a dark background with yellow text. It shows a menu of database operations: "Current db: /database.dat", "1 - Insert data", "2 - Remove data", "3 - Update data", "4 - Find data", "5 - Load data from file", "6 - Save to file and exit". Below the menu, it shows "j djnc" and "Press enter...".

```
C:\WINDOWS\system32\cmd.exe - node .

Current db: /database.dat

1 - Insert data
2 - Remove data
3 - Update data
4 - Find data
5 - Load data from file
6 - Save to file and exit

j djnc

Press enter...
```

Рисунок 3.2 – Пошук запису

3.3 Тестування алгоритму

3.3.1 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 (ключ)	Число порівнянь
1 (21003)	7
2 (500)	5
3 (203)	9
4 (1102)	7
5 (30004)	9
6 (100233)	8
7 (83)	8
8 (8356)	7
9 (2314)	9
10 (123)	10
11 (8653)	7
12 (79595)	8
13 (2355)	8
14 (97776)	7
15 (123193)	9
Середнє	~7,86667

ВИСНОВОК

В рамках лабораторної роботи була створена невелика СУБД з використанням В-дерева і бінарного пошуку в її основі, досліджена швидкодія такого методу доступу до даних (через структуру даних В-дерево). Усі операції виконуються за логарифмічний час, що забезпечує дуже ефективну та швидку роботу з даними. Детально розглянув алгоритми пошуку, вставки і видалення, реалізовані в В-дереві. Дуже складним є алгоритм видалення, адже, на відміну від вставки, видалення може бути здійснено в будь-якій вершині (не тільки в листовій), що породжує декілька випадків, які потрібно відслідковувати для збереження цілісності самого В-дерева та його властивостей (збалансованість, кількість вузлів в кожній вершині дерева, тощо).

КРИТЕРІЇ ОЦІНЮВАННЯ

За умови здачі лабораторної роботи до 22.10.2021 включно максимальний бал дорівнює – 5. Після 22.10.2021 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

- псевдокод алгоритму – 20%;
- аналіз часової складності – 5%;
- програмна реалізація алгоритму – 60%;
- тестування алгоритму – 10%;
- висновок – 5%.

+1 додатковий бал можна отримати за реалізацію графічного інтерфейсу.

+1 додатковий бал можна отримати за реалізацію графічного зображення структури ключів.