Задачі на списки

1. Написати функцію для впорядкування елементів зв'язного списку за зростанням

```
#include <iostream>
class LinkedList {
private:
    struct Node {
        int data;
        Node* next;

        Node(int value) : data(value), next(nullptr) {}
    };

    Node* head;

    Node* merge(Node* left, Node* right) {
        if (!left) return right;
        if (!right) return left;

        Node* result = nullptr;
        if (left->data <= right->data) {
            result = left;
            result->next = merge(left->next, right);
        }
        else {
            result = right;
            result->next = merge(left, right->next);
        }

        return result;
    }
    Node* getMiddle(Node* current) {
        if (!current) return current;

        Node* slowPtr = current;
        Node* fastPtr = current;

        while (fastPtr->next != nullptr && fastPtr->next->next != nullptr) {
            slowPtr = slowPtr->next;
            fastPtr = fastPtr->next->next;
        }

        return slowPtr;
    }
    Node* mergeSort(Node* current) {
        if (!current || !current->next) return current;

        Node* middle = getMiddle(current);
        Node* nextToMiddle = middle->next;
        middle->next = nullptr;

        Node* leftHalf = mergeSort(current);
        Node* rightHalf = mergeSort(nextToMiddle);

        return merge(leftHalf, rightHalf);
    }

public:
    LinkedList() : head(nullptr) {}

    void sort() {
        head = mergeSort(head);
}
};
```

2. Многочлен від однієї змінної з цілими коефіцієнтам можна подати у вигляді зв'язного списку, впорядкованим за зростанням степені змінної, без зберігання одночленів з нульовими коефіцієнтами. Написати функцію, яка реалізує обчислення похідної від многочлен.

```
#include <iostream>

struct Monomial {
    int coeff; // Коефіцієнт
    int exp; // Степінь
    Monomial* next; // Вказівник на наступний одночлен

    Monomial(int c, int e, Monomial* n = nullptr) : coeff(c), exp(e), next(n) {}
};

struct Polynomial {
    Monomial* head; // Вказівник на перший одночлен

    Polynomial() : head(nullptr) {} // Конструктор

    // Додавання одночлена до многочлена
    void addMonomial(int coeff, int exp) {
        if (coeff == 0) return;

        if (!head || head->exp > exp) {
            Monomial* newMonomial = new Monomial(coeff, exp, head);
            head = newMonomial;
        }
        else {
            Monomial* current = head;
            while (current->next && current->next->exp <= exp) {
                if (current->next->exp == exp) {
                    current->next->coeff += coeff;
                    if (current->next->coeff == 0) {
                        Monomial* temp = current->next;
                        current->next = current->next->next;
                        delete temp;
                    }
                    return;
                }
                current = current->next;
            }
            if (current->exp == exp) {
                current->coeff += coeff;
                if (current->coeff == 0) {
                    Monomial* temp = current;
                    head = current->next;
                    delete temp;
                }
            }
            else {
                Monomial* newMonomial = new Monomial(coeff, exp, current->next);
                current->next = newMonomial;
            }
        }
    }

    // Отримання похідної многочлена
    Polynomial derivative() const {
        Polynomial result;
        Monomial* current = head;
        while (current) {
            if (current->exp != 0) {
                result.addMonomial(current->coeff * current->exp, current->exp - 1);
```

```
            }
            current = current->next;
        }
        return result;
    }

    // Диференціювання многочлена на місці
    void differentiateInPlace() {
        Monomial* current = head;
        Monomial* prev = nullptr;

        while (current) {
            if (current->exp == 0) {
                Monomial* temp = current;
                if (prev) {
                    prev->next = current->next;
                }
                else {
                    head = current->next;
                }
                current = current->next;
                delete temp;
            }
            else {
                current->coeff *= current->exp;
                current->exp -= 1;
                prev = current;
                current = current->next;
            }
        }
    }
};
```

3. Многочлен від однієї змінної з цілими коефіцієнтам можна подати у вигляді зв'язного списку, впорядкованим за зростанням степені змінної, без зберігання одночленів з нульовими коефіцієнтами. Написати функцію, яка знаходить степінь многочлена.

```
    // Визначення степеня многочлена
    int degree() const {
        if (!head) return -1; // Якщо список порожній, повертаємо -1
        int maxExp = head->exp;
        Monomial* current = head->next;
        while (current) {
            if (current->exp > maxExp) {
                maxExp = current->exp;
            }
            current = current->next;
        }
        return maxExp;
    }
```

(В теорії можна повернути head → degree)

4. Поліном від однієї змінної з цілими коефіцієнтам можна подати у вигляді зв'язного списку, впорядкованим за зростанням степені змінної, без зберігання одночленів з нульовими коефіцієнтами. Написати функцію, яка знаходить суму двох поліномів.

```
// Додавання двох многочленів
Polynomial addPolynomials(const Polynomial& poly1, const Polynomial& poly2) {
    Polynomial result;
    Monomial* p1 = poly1.head;
    Monomial* p2 = poly2.head;

    while (p1 && p2) {
        if (p1->exp < p2->exp) {
            result.addMonomial(p1->coeff, p1->exp);
            p1 = p1->next;
        }
        else if (p1->exp > p2->exp) {
            result.addMonomial(p2->coeff, p2->exp);
            p2 = p2->next;
        }
        else {
            int sumCoeff = p1->coeff + p2->coeff;
            if (sumCoeff != 0) {
                result.addMonomial(sumCoeff, p1->exp);
            }
            p1 = p1->next;
            p2 = p2->next;
        }
    }

    while (p1) {
        result.addMonomial(p1->coeff, p1->exp);
        p1 = p1->next;
    }

    while (p2) {
        result.addMonomial(p2->coeff, p2->exp);
        p2 = p2->next;
    }

    return result;
}
```

5. Многочлен від трьох змінних з цілими коефіцієнтами можна подати зв’язним списком, впорядкованим за зростанням степенів змінних, без зберігання одночленів з нульовими коефіцієнтами. Написати функцію, яка обчислює значення многочлена при заданих значеннях змінних.

```
#include <iostream>

// Структура для одночлена
struct Monomial {
    int coeff; // Коефіцієнт
    char var; // Змінна ('x', 'y', або 'z')
    int exp; // Ступінь змінної
    Monomial* next; // Вказівник на наступний одночлен

    Monomial(int c, char v, int e) : coeff(c), var(v), exp(e), next(nullptr) {}
};
```

```
// Структура для многочлена
struct Polynomial {
    Monomial* head; // Вказівник на перший одночлен

    Polynomial() : head(nullptr) {} // Конструктор

    // Додати одночлен до многочлена
    void addMonomial(int coeff, char var, int exp) {
        if (coeff == 0) return; // Ігнорувати нульовий коефіцієнт

        Monomial* newMonomial = new Monomial(coeff, var, exp);

        // Вставка на початок або перед першим одночленом з більшим показником ступеня або
змінною
        if (!head || exp < head->exp || (exp == head->exp && var < head->var)) {
            newMonomial->next = head;
            head = newMonomial;
            return;
        }

        Monomial* current = head;
        Monomial* prev = nullptr;

        while (current && (current->exp < exp || (current->exp == exp && current->var <= var))) {
            if (current->exp == exp && current->var == var) {
                current->coeff += coeff;
                delete newMonomial;
                if (current->coeff == 0) { // Видалити одночлен, якщо коефіцієнт став нульовим
                    if (prev) {
                        prev->next = current->next;
                    }
                    else {
                        head = current->next;
                    }
                    delete current;
                }
                return;
            }
            prev = current;
            current = current->next;
        }

        newMonomial->next = current;
        if (prev) {
            prev->next = newMonomial;
        }
    }
};
```

6. До лінійного списку F з m цілих чисел, більшість з яких дорівнюють 0, застосовано стисле зв’язне зберігання. Написати

функцію для визначення i-го за порядком елемента списку F.

```
#include <iostream>
#include <stdexcept>

class CompressedLinkedList {
private:
    struct Node {
        int index; // Індекс елемента
        int value; // Значення елемента
        Node* next; // Вказівник на наступний вузол
        Node(int idx, int val) : index(idx), value(val), next(nullptr) {} // Конструктор вузла
    };

    Node* head; // Вказівник на голову списку
    int totalSize; // Загальний розмір списку

public:
    CompressedLinkedList(int size = 0) : head(nullptr), totalSize(size) {} // Конструктор списку

    ~CompressedLinkedList() { // Деструктор для видалення всіх вузлів
        Node* current = head;
        while (current != nullptr) {
            Node* next = current->next;
            delete current;
            current = next;
        }
    }

    void addElement(int value) { // Додати елемент у список
        if (value == 0) {
            totalSize++;
            return;
        }

        Node* newNode = new Node(totalSize, value);
        if (!head) {
            newNode->next = head;
            head = newNode;
        }
        else {
            Node* current = head;
            while (current->next != nullptr) {
                current = current->next;
            }
            newNode->next = current->next;
            current->next = newNode;
        }
        totalSize++;
    }

    int getElement(int index) const { // Отримати елемент за індексом
        if (index < 0 || index >= totalSize) {
            throw std::out_of_range("Index is out of bounds");
        }

        Node* current = head;
        while (current != nullptr) {
            if (current->index == index) {
                return current->value;
            }
            current = current->next;
        }

        return 0; // Якщо елемент не знайдений, його значення вважається нульовим
```

```
    }
};
```

7. До лінійного списку F з m цілих чисел, більшість з яких дорівнюють 0, застосовано стисле зв’язне зберігання. Написати функцію для визначення кількості елементів із значенням 0, номери яких належать інтервалу [i,j].

```
    int countZerosInRange(int start, int end) const { // Порахувати кількість нулів у заданому діапазоні
        if (start > end) return 0; // Неправильний інтервал

        Node* current = head;
        int currentIndex = 0;
        int zeroCount = 0;

        // Пропустити вузли до початку діапазону
        while (current && current->index < start) {
            currentIndex = current->index + 1;
            current = current->next;
        }

        // Якщо немає вузлів або перший вузол знаходиться за межами кінцевого індексу, всі елементи нулі
        if (!current || current->index > end) {
            return end - start + 1;
        }

        // Порахувати нулі від початку до першого ненульового елемента
        if (current->index > start) {
            zeroCount += current->index - start;
        }

        // Порахувати нулі між ненульовими елементами і до кінцевого індексу
        while (current && current->index <= end) {
            int nextIndex = (current->next) ? current->next->index : totalSize;

            // Якщо наступний індекс перевищує кінцевий, рахуємо нулі до кінцевого індексу
            if (nextIndex > end) {
                zeroCount += end - (current->index + 1) + 1;
            }
            else {
                zeroCount += nextIndex - (current->index + 1);
            }

            current = current->next;
        }

        return zeroCount;
    }
```

8. До лінійного списку F з m цілих чисел, більшість з яких дорівнюють 0, застосовано стисле зв’язне зберігання. Написати функцію для визначення номера першого за порядком елемента зі значенням 0.

```
int firstZeroIndex() const { // Знайти перший індекс з нульовим значенням
    if (head == nullptr) {
        return 0; // Якщо список порожній, перший елемент є нульовим.
    }

    Node* current = head;
    int expectedIndex = 0;

    while (current != nullptr) {
        if (current->index != expectedIndex) {
            // Проміжок перед індексом поточного вузла заповнений нулями.
            return expectedIndex;
        }
        expectedIndex++;
        current = current->next;
    }

    // Якщо проміжок не знайдено, перший нульовий індекс знаходиться в кінці списку.
    return expectedIndex;
}
```

9. До лінійного списку F з m цілих чисел, більшість з яких дорівнюють 0, застосовано стисле зв’язне зберігання. Написати функцію для визначення кількості невід’ємних елементів номери яких належать інтервалу [i,j].

```
int countNonNegativeInRange(int start, int end) const { // Порахувати кількість невід'ємних елементів у заданому діапазоні
    if (start > end) return 0; // Неправильний інтервал

    Node* current = head;
    int currentIndex = 0;
    int nonNegativeCount = 0;

    // Пропустити вузли до початку діапазону
    while (current && current->index < start) {
        currentIndex = current->index + 1;
        current = current->next;
    }

    // Якщо немає вузлів або перший вузол знаходиться за межами кінцевого індексу, всі елементи нулі
    if (!current || current->index > end) {
        return end - start + 1;
    }

    // Порахувати нулі від початку до першого ненульового елемента
    if (current->index > start) {
        nonNegativeCount += current->index - start;
    }

    // Порахувати нулі між ненульовими елементами і до кінцевого індексу
    while (current && current->index <= end) {
        int nextIndex = (current->next) ? current->next->index : totalSize;
        if (current->value > 0) nonNegativeCount++;
        // Якщо наступний індекс перевищує кінцевий, рахуємо нулі до кінцевого індексу
        if (nextIndex > end) {
            nonNegativeCount += end - (current->index + 1) + 1;
        }
        else {
            nonNegativeCount += nextIndex - (current->index + 1);
        }

        current = current->next;
```

```
    }

    return nonNegativeCount;
}
```

10. Список F з цілих чисел, більшість якого дорівнюють 0, представлений своїм зв’язним зберіганням. Написати функцію для представлення F стислим зв’язним зберіганням

```
void addElement(int value) { // Додати елемент у список
        if (value == 0) {
            totalSize++;
            return;
        }

        Node* newNode = new Node(totalSize, value);
        if (!head) {
            newNode->next = head;
            head = newNode;
        }
        else {
            Node* current = head;
            while (current->next != nullptr) {
                current = current->next;
            }
            newNode->next = current->next;
            current->next = newNode;
        }
        totalSize++;
    }

void convertToCompressedLinkedList(Node* head, CompressedLinkedList&
compressedList) {
    Node* current = head;
    while (current != nullptr) {
        compressedList.addElement(current->data);
        current = current->next;
    }
}
```

11. Лінійний список F цілих чисел зберігається як послідовно-зв’язний індексний список так, що числа, яка мають однакову останню цифру, розміщуються в один підсписок. Написати функцію,

яка додає до списку F елемент зі значенням k, якщо такого елемент у списку відсутній.

```
#include <iostream>

using namespace std;


//Вузол зв'язного списку
struct Node {
	int data;
	Node* next;
	Node(int val) : data(val), next(nullptr) {};
};

class LinkedList {
private:
	Node* head;
public:
	LinkedList() : head(nullptr) {};

	void add(int val) {
		Node* newNode = new Node(val);
		if (!head) {
			head = newNode;
		}
		else {
			Node* temp = head;
			while (temp->next) {
				temp = temp->next;
			}
			temp->next = newNode;
		}
	}

	bool numInList(int val) {
		Node* temp = head;
		while (temp) {
			if (temp->data == val) {
				return true;
			}
			temp = temp->next;
		}
		return false;
	}

	~LinkedList() {
		Node* temp = head;
		while (temp) {
			Node* nextNode = temp->next;
			delete temp;
			temp = nextNode;
		}
		head = nullptr;
	}
};

struct IndexedList {
	LinkedList arr[10];

	void add(int val) {
		int k = abs(val % 10);
		if (!arr[k].numInList(val)) arr[k].add(val);
	}
```

```
};
```

12. Лінійний список F цілих чисел зберігається як послідовно-зв’язний індексний список так, що числа, які мають дві однакові цифри, розміщуються в один підсписок. Написати функцію, що обчислює кількість елементів списку F із значенням k.
13. Лінійний список F цілих чисел зберігається як послідовно-зв’язний індексний список так, що числа, які мають дві однакові останні цифри, розміщуються в один підсписок. Написати функцію, яка вилучає з списку елемент зі значенням v, якщо він присутній.

```
#include <iostream>

using namespace std;

//Вузол зв'язного списку
struct Node {
		int data;
		Node* next;
		Node(int val) : data(val), next(nullptr) {};
};

class LinkedList {
public:
		Node* head;
		LinkedList() : head(nullptr) {};

		void add(int val) {
				Node* newNode = new Node(val);
				if (!head) {
						head = newNode;
				}
				else {
						Node* temp = head;
						while (temp->next) {
								temp = temp->next;
						}
						temp->next = newNode;
				}
		}

		bool remove(Node*& head, int val) {
				// Якщо список порожній, немає чого видаляти
				if (head == nullptr) {
						return false;
				}

				// Якщо перший вузол має задане значення
				if (head->data == val) {
						Node* temp = head;
						head = head->next;
						delete temp;
						return true;
				}

				// Знаходження вузла, який потрібно видалити
				Node* prev = head;
				Node* current = head->next;
				while (current != nullptr) {
						if (current->data == val) {
								prev->next = current->next;
```

```
                                        delete current;
                                        return true;
                                }
                                prev = current;
                                current = current->next;
                        }
                        return false;
                }

                bool numInList(int val) {
                        Node* temp = head;
                        while (temp) {
                                if (temp->data == val) {
                                        return true;
                                }
                                temp = temp->next;
                        }
                        return false;
                }
                int counting(int val){
                        Node* temp = head;
                        int count = 0;
                        while (temp) {
                                if (temp->data == val) {
                                        count++;
                                }
                                temp = temp->next;
                        }
                        return count;
                }
                ~LinkedList() {
                        Node* temp = head;
                        while (temp) {
                                Node* nextNode = temp->next;
                                delete temp;
                                temp = nextNode;
                        }
                        head = nullptr;
                }
};

struct IndexedList {
                LinkedList arr[10];

                void add(int val) {
                        int k = abs(val % 10);
                        if (!arr[k].numInList(val)) arr[k].add(val);
                }

                void remove(int val) {
                        int k = abs(val % 10);
                        bool deleted = arr[k].remove(arr[k].head, val);
                        deleted ? cout << "Element " << val << " was deleted!" : cout << "Element
" << val << " wasn`t deleted!";
                        std::cout << endl;
                }

                int countNum(int val) {
                        return arr[abs(val % 10)].counting(val);
                }
};
```

14. В елементах циклічного списку розміщені цілі числа. Написати функцію копіювання списку, помінявши порядок на

обернений

```
#include <iostream>

// Структура, що представляє вузол списку
struct Node {
    int data;
    Node* next;

    Node(int val) : data(val), next(nullptr) {}
};

// Клас для циклічного списку
class CircularList {
private:
    Node* head; // Початковий вузол списку
public:
    // Конструктор за замовчуванням
    CircularList() : head(nullptr) {}

    // Функція копіювання списку зі зміненим порядком на обернений
    CircularList reverseCopy() {
        CircularList reversedList;
        if (head == nullptr)
            return reversedList; // Повертаємо порожній список

        Node* current = head;
        // Проходимо по оригінальному списку
        do {
            // Вставляємо копію поточного вузла перед початковим вузлом нового списку
            reversedList.prepend(current->data);
            current = current->next;
        } while (current != head);

        return reversedList;
    }

    // Додавання елемента в початок списку
    void prepend(int val) {
        Node* newNode = new Node(val);
        if (head == nullptr) {
            head = newNode;
            newNode->next = head;
        }
        else {
            Node* temp = head;
            while (temp->next != head) {
                temp = temp->next;
            }
            temp->next = newNode;
            newNode->next = head;
            head = newNode;
        }
    }
};
```

15. В елементах циклічного списку розміщені цілі числа. Написати функцію, що розбиває множину поданих чисел на дві: з

парними значеннями та непарними, використовуючи для множин представлення циклічним списком.

```
#include <iostream>

// Структура, що представляє вузол списку
struct Node {
    int data;
    Node* next;

    Node(int val) : data(val), next(nullptr) {}
};

// Клас для циклічного списку
class CircularList {
private:
    Node* head; // Початковий вузол списку
public:
    // Конструктор за замовчуванням
    CircularList() : head(nullptr) {}

    // Додавання елемента в кінець списку
    void append(int val) {
        Node* newNode = new Node(val);
        if (head == nullptr) {
            head = newNode;
            newNode->next = head;
        }
        else {
            Node* temp = head;
            while (temp->next != head) {
                temp = temp->next;
            }
            temp->next = newNode;
            newNode->next = head;
        }
    }

    // Функція розділення чисел на парні та непарні
    void splitEvenOdd(CircularList& evenList, CircularList& oddList) {
        if (head == nullptr)
            return; // Нічого робити, якщо список порожній

        Node* current = head;
        do {
            if (current->data % 2 == 0) {
                // Додаємо парне число до списку парних чисел
                evenList.append(current->data);
            }
            else {
                // Додаємо непарне число до списку непарних чисел
                oddList.append(current->data);
            }
            current = current->next;
        } while (current != head);
    }
};
```

16. В елементах двохзв’язного списку розміщені цілі числа. Написати функцію, що вилучає всі мінімальні елементи.

```
#include <iostream>

struct Node {
    int data;
    Node* prev;
    Node* next;

    Node(int val) : data(val), prev(nullptr), next(nullptr) {}
};

class DoublyLinkedList {
private:
    Node* head;

public:
    DoublyLinkedList() : head(nullptr) {}

    void append(int val) {
        Node* newNode = new Node(val);
        if (head == nullptr) {
            head = newNode;
        }
        else {
            Node* current = head;
            while (current->next != nullptr) {
                current = current->next;
            }
            current->next = newNode;
            newNode->prev = current;
        }
    }

    void removeMinElements() {
        if (head == nullptr)
            return;

        int minVal = head->data;
        Node* current = head->next;
        while (current != nullptr) {
            if (current->data < minVal) {
                minVal = current->data;
            }
            current = current->next;
        }

        current = head;
        while (current != nullptr) {
            Node* nextNode = current->next;
            if (current->data == minVal) {
                if (current->prev != nullptr)
                    current->prev->next = current->next;
                if (current->next != nullptr)
                    current->next->prev = current->prev;
                if (current == head)
                    head = current->next;
                delete current;
            }
            current = nextNode;
        }
    }
};
```

17. Написати функцію для копіювання списку в “однозв’язному представлені” , зберігаючи взаємний порядок елементів, але

зробивши у копії “двозв’язне циклічне представлення”.

```
#include <iostream>

// Клас для вузла двозв'язного циклічного списку
class DoublyCyclicNode {
public:
    int data;
    DoublyCyclicNode* prev;
    DoublyCyclicNode* next;

    DoublyCyclicNode(int val) : data(val), prev(nullptr), next(nullptr) {}
};

// Клас для двозв'язного циклічного списку
class DoublyCyclicLinkedList {
private:
    DoublyCyclicNode* head;

public:
    DoublyCyclicLinkedList() : head(nullptr) {}

    // Додавання елемента в кінець списку
    void append(int val) {
        DoublyCyclicNode* newNode = new DoublyCyclicNode(val);
        if (!head) {
            head = newNode;
            newNode->prev = newNode;
            newNode->next = newNode;
        }
        else {
            DoublyCyclicNode* last = head->prev;
            last->next = newNode;
            newNode->prev = last;
            newNode->next = head;
            head->prev = newNode;
        }
    }
};
// Клас для вузла однозв'язного списку
class SinglyNode {
public:
    int data;
    SinglyNode* next;

    SinglyNode(int val) : data(val), next(nullptr) {}
};

// Клас для однозв'язного списку
class SinglyLinkedList {
private:
    SinglyNode* head;

public:
    SinglyLinkedList() : head(nullptr) {}

    // Функція для копіювання списку у двозв'язний циклічний список
    void copyToDoublyCyclic(DoublyCyclicLinkedList& result) {
        if (head == nullptr)
            return;

        SinglyNode* current = head;
        do {
            result.append(current->data);
            current = current->next;
```

```
    } while (current);
  }
};

// Функція для копіювання однозв'язного списку у двозв'язний циклічний список
DoublyCyclicLinkedList copyToDoublyCyclic(SinglyLinkedList& list) {
  DoublyCyclicLinkedList result;
  list.copyToDoublyCyclic(result);
  return result;
}
```

18. Написати функцію, яка для двох скінченних множин цілих чисел, поданими впорядкованими списками, знаходить їх перетин

```
#include <iostream>

// Клас для вузла списку
class Node {
public:
  int data;
  Node* next;

  Node(int val) : data(val), next(nullptr) {}
};

// Клас для впорядкованого списку
class SortedLinkedList {
private:
  Node* head;

public:
  SortedLinkedList() : head(nullptr) {}

  // Додавання елемента впорядковано
  void append(int val) {
    Node* newNode = new Node(val);
    if (head == nullptr || val < head->data) {
      newNode->next = head;
      head = newNode;
    }
    else {
      Node* current = head;
      while (current->next != nullptr && val > current->next->data) {
        current = current->next;
      }
      newNode->next = current->next;
      current->next = newNode;
    }
  }

  // Функція для знаходження перетину з іншим впорядкованим списком
  SortedLinkedList intersection(SortedLinkedList& otherList) {
    SortedLinkedList result;
    Node* current1 = head;
    Node* current2 = otherList.head;

    while (current1 != nullptr && current2 != nullptr) {
      if (current1->data == current2->data) {
        result.append(current1->data);
        current1 = current1->next;
        current2 = current2->next;
      }
      else if (current1->data < current2->data) {
        current1 = current1->next;
      }
      else {
```

```
            current2 = current2->next;
        }
    }

    return result;
  }
};
```

19. В елементах списку розміщені цілі цифри. Написати функцію, що перевпорядковує список, збираючи спочатку всі додатні числа, а потім – інші.

```
#include <iostream>

// Клас для вузла списку
class Node {
public:
  int data;
  Node* next;

  Node(int val) : data(val), next(nullptr) {}
};

// Клас для списку цілих чисел
class LinkedList {
private:
  Node* head;

public:
  LinkedList() : head(nullptr) {}

  // Додавання елемента в кінець списку
  void append(int val) {
    Node* newNode = new Node(val);
    if (head == nullptr) {
      head = newNode;
    }
    else {
      Node* current = head;
      while (current->next != nullptr) {
        current = current->next;
      }
      current->next = newNode;
    }
  }

  // Функція для перевпорядкування списку
  void reorder() {
    if (head == nullptr || head->next == nullptr)
      return; // нічого робити, якщо список порожній або містить тільки один елемент

    Node* positiveHead = nullptr;
    Node* positiveTail = nullptr;
    Node* negativeHead = nullptr;
    Node* negativeTail = nullptr;

    Node* current = head;
    while (current != nullptr) {
      Node* nextNode = current->next;
      if (current->data > 0) {
        if (positiveHead == nullptr) {
          positiveHead = current;
          positiveTail = current;
          current->next = nullptr;
        }
```

```
            else {
                positiveTail->next = current;
                positiveTail = current;
                current->next = nullptr;
            }
        }
        else {
            if (negativeHead == nullptr) {
                negativeHead = current;
                negativeTail = current;
                current->next = nullptr;
            }
            else {
                negativeTail->next = current;
                negativeTail = current;
                current->next = nullptr;
            }
        }
        current = nextNode;
    }

    if (positiveHead != nullptr) {
        head = positiveHead;
        positiveTail->next = negativeHead;
    }
    else {
        head = negativeHead;
    }
  }
};
```

20. В елементах списку розміщені цілі цифри. Написати функцію, що перевпорядковує список, збираючи спочатку всі додатні числа, потім – 0, а потім – інші.

```
  // Функція для перевпорядкування списку
  void reorder() {
    if (head == nullptr || head->next == nullptr)
        return; // нічого робити, якщо список порожній або містить тільки один елемент

    Node* positiveHead = nullptr;
    Node* positiveTail = nullptr;
    Node* zeroHead = nullptr;
    Node* zeroTail = nullptr;
    Node* negativeHead = nullptr;
    Node* negativeTail = nullptr;

    Node* current = head;
    while (current != nullptr) {
        Node* nextNode = current->next;
        if (current->data > 0) {
            if (positiveHead == nullptr) {
                positiveHead = current;
                positiveTail = current;
                current->next = nullptr;
            }
            else {
                positiveTail->next = current;
                positiveTail = current;
                current->next = nullptr;
            }
        }
        else if (current->data == 0) {
            if (zeroHead == nullptr) {
```

```
                zeroHead = current;
                zeroTail = current;
                current->next = nullptr;
            }
            else {
                zeroTail->next = current;
                zeroTail = current;
                current->next = nullptr;
            }
        }
        else {
            if (negativeHead == nullptr) {
                negativeHead = current;
                negativeTail = current;
                current->next = nullptr;
            }
            else {
                negativeTail->next = current;
                negativeTail = current;
                current->next = nullptr;
            }
        }
        current = nextNode;
    }

    if (positiveHead != nullptr) {
        head = positiveHead;
        positiveTail->next = zeroHead;
        if (zeroHead != nullptr) {
            zeroTail->next = negativeHead;
        }
        else {
            positiveTail->next= negativeHead;
        }
    }
    else if (zeroHead != nullptr) {
        head = zeroHead;
        zeroTail->next = negativeHead;
    }
    else {
        head = negativeHead;
    }
}
```

21. В елементах списку розміщені цілі числа. Написати функцію, що перевпорядковує список, збираючи спочатку всі парні числа, а потім – непарні.

```
// Функція для перевпорядкування списку
void reorderList() {
    if (!head) return;  // Якщо список порожній, нічого не робимо

    Node* evenHead = nullptr;  // Голова списку парних чисел
    Node* evenTail = nullptr;  // Хвіст списку парних чисел
    Node* oddHead = nullptr;   // Голова списку непарних чисел
    Node* oddTail = nullptr;   // Хвіст списку непарних чисел

    Node* current = head;
    while (current) {
        if (current->data % 2 == 0) {  // Якщо число парне
            if (!evenHead) {
                evenHead = evenTail = current;
            }
            else {
                evenTail->next = current;
                evenTail = evenTail->next;
            }
        }
        else {  // Якщо число непарне
            if (!oddHead) {
                oddHead = oddTail = current;
            }
            else {
                oddTail->next = current;
                oddTail = oddTail->next;
            }
        }
        current = current->next;
    }

    if (evenTail) {
        evenTail->next = oddHead;  // З'єднання парного списку з непарним
    }
    if (oddTail) {
        oddTail->next = nullptr;  // Встановлення кінця списку
    }

    head = evenHead ? evenHead : oddHead;  // Новий head - голова парного або непарного
списку
}
```

22. Здійснити злиття двох впорядкованих списків в у зв’язному зберіганні в один, який також впорядкований й представлений новим списком.

```
// Функція для злиття двох впорядкованих списків
LinkedList mergeLists(const LinkedList& list1, const LinkedList& list2) {
    LinkedList mergedList;
    Node* current1 = list1.head;
    Node* current2 = list2.head;
    Node* mergedTail = nullptr;

    while (current1 != nullptr && current2 != nullptr) {
        if (current1->data <= current2->data) {
            if (mergedTail == nullptr) {
                mergedList.head = mergedTail = new Node(current1->data);
            }
            else {
                mergedTail->next = new Node(current1->data);
                mergedTail = mergedTail->next;
            }
            current1 = current1->next;
        }
        else {
            if (mergedTail == nullptr) {
                mergedList.head = mergedTail = new Node(current2->data);
            }
            else {
                mergedTail->next = new Node(current2->data);
                mergedTail = mergedTail->next;
            }
            current2 = current2->next;
        }
    }

    while (current1 != nullptr) {
        if (mergedTail == nullptr) {
            mergedList.head = mergedTail = new Node(current1->data);
        }
        else {
            mergedTail->next = new Node(current1->data);
            mergedTail = mergedTail->next;
        }
        current1 = current1->next;
    }

    while (current2 != nullptr) {
        if (mergedTail == nullptr) {
            mergedList.head = mergedTail = new Node(current2->data);
        }
        else {
            mergedTail->next = new Node(current2->data);
            mergedTail = mergedTail->next;
        }
        current2 = current2->next;
    }

    return mergedList;
}
```

23. Написати функцію, яка переміщує найбільший елемент списку у кінець. Список заданий однозв’язним представленням.

```
    // Функція для переміщення найбільшого елемента у кінець списку
    void moveMaxToEnd() {
```

```
    if (!head || !head->next) return; // Якщо список порожній або містить один елемент, нічого не робимо

    Node* maxNode = head;
    Node* maxPrev = nullptr;
    Node* current = head;
    Node* prev = nullptr;

    // Знаходимо найбільший елемент і його попередника
    while (current != nullptr) {
        if (current->data > maxNode->data) {
            maxNode = current;
            maxPrev = prev;
        }
        prev = current;
        current = current->next;
    }

    if (maxNode->next == nullptr) return; // Якщо найбільший елемент уже в кінці списку, нічого не робимо

    // Видаляємо maxNode з поточного місця
    if (maxPrev != nullptr) {
        maxPrev->next = maxNode->next;
    }
    else {
        head = maxNode->next; // Якщо maxNode був головою списку
    }

    // Знаходимо останній елемент списку
    Node* tail = head;
    while (tail->next != nullptr) {
        tail = tail->next;
    }

    // Додаємо maxNode в кінець списку
    tail->next = maxNode;
    maxNode->next = nullptr;
}
```

24. Написати функцію, яка вилучає з списку у зв’язному представлені всі вузли розташовані на позиціях кратних 5.

```
// Функція для видалення вузлів на позиціях кратних 5
void removeNodesAtMultiplesOfFive() {
```

```
    if (!head) return;  // Якщо список порожній, нічого не робимо

    Node* current = head;
    Node* prev = nullptr;
    int position = 0;  // Починаємо з позиції 0

    while (current != nullptr) {
        if (position % 5 == 0) {
            Node* nodeToDelete = current;
            if (prev != nullptr) {
                prev->next = current->next;
            }
            else {
                head = current->next;  // Якщо видаляємо голову списку
            }
            current = current->next;
            delete nodeToDelete;
        }
        else {
            prev = current;
            current = current->next;
        }
        position++;
    }
```

25. В елементах зв’язного списку розміщені різні цілі числа. Написати функцію, що роздруковує значенням елементів розташованих між найменшим і найбільшим елементами списку.

```
// Функція для друку значень між найменшим і найбільшим елементами
void printValuesBetweenMinMax() const {
    if (!head) return; // Якщо список порожній, нічого не робимо
    // Знаходимо найменший і найбільший елементи
    Node* minNode = head;
    Node* maxNode = head;
    Node* current = head;

    while (current != nullptr) {
        if (current->data < minNode->data) {
            minNode = current;
        }
        if (current->data > maxNode->data) {
            maxNode = current;
        }
        current = current->next;
    }

    if (maxNode < minNode) {
        std::swap(maxNode, minNode);
    }

    // Друкуємо значення між minNode і maxNode
    current = minNode->next;
    while (current != maxNode) {
        std::cout << current->data << " ";
        current = current->next;
    }
    std::cout << std::endl;
}
```

26. Написати функцію, яка у списку, що зберігається у зв’язному представленні, з кожної групи сусідніх однакових елементів залишає лише один.

```
// Метод для видалення сусідніх дублікатів
```

```
void removeDuplicates() {
    if (head == nullptr) {
        return;
    }

    Node* current = head;
    while (current != nullptr && current->next != nullptr) {
        if (current->data == current->next->data) {
            Node* duplicate = current->next;
            current->next = current->next->next;
            delete duplicate;
        }
        else {
            current = current->next;
        }
    }
}
```

27. Написати функцію для впорядкування за зростаючи порядком елементів списку у в’язному зберіганні, використовуючи “сортування вибором”.

```
// Метод для сортування списку за зростанням (сортування вибором)
void selectionSort() {
    if (head == nullptr) {
        return;
    }

    Node* current = head;

    while (current != nullptr) {
        Node* minNode = current;
        Node* nextNode = current->next;

        while (nextNode != nullptr) {
            if (nextNode->data < minNode->data) {
                minNode = nextNode;
            }
            nextNode = nextNode->next;
        }

        // Обмін значень
        if (minNode != current) {
            int temp = current->data;
            current->data = minNode->data;
            minNode->data = temp;
        }

        current = current->next;
    }
}
```

Задачі на дерева

1. Використовуючи відповідний механізм черг або стеків, написати функцію, яка виводить елементи бінарного дерева, поданого в стандартній формі, по рівнях (починаючи з кореня дерева, далі з

синів кореня й далі)

```
#include <iostream>
#include <queue>

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class BinaryTree {
public:
    TreeNode* root;

    BinaryTree() : root(nullptr) {}

    void printByLevels() {
        if (!root) {
            return;
        }

        std::queue<TreeNode*> q;
        q.push(root);

        int level = 0; // Змінна для зберігання номера поточного рівня

        while (!q.empty()) {
            int size = q.size(); // Кількість елементів у черзі - це кількість елементів на поточному
рівні

            std::cout << "Level " << level << ": "; // Виводимо номер рівня

            while (size--) {
                TreeNode* node = q.front();
                q.pop();
                std::cout << node->val << " ";

                if (node->left != nullptr) {
                    q.push(node->left);
                }
                if (node->right != nullptr) {
                    q.push(node->right);
                }
            }

            std::cout << std::endl;
            level++; // Переходимо до наступного рівня
        }
    }
};
```

2. Написати не рекурсивну функцію (з використанням стека) для друкування відміток вузлів бінарного дерева, поданого в стандартній формі, при його проходженні в оберненому порядку.

```
#include <iostream>
#include <queue>
```

```
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class BinaryTree {
public:
    TreeNode* root;

    BinaryTree() : root(nullptr) {}

    void printPostorder() {
     if (root == nullptr)
        return;

     std::stack<TreeNode*> stack1;
     std::stack<TreeNode*> stack2;

     stack1.push(root);

     while (!stack1.empty()) {
        TreeNode* node = stack1.top();
        stack1.pop();
        stack2.push(node);

        if (node->left != nullptr)
            stack1.push(node->left);

        if (node->right != nullptr)
            stack1.push(node->right);
     }

     // Виведення відміток в оберненому порядку з використанням другого стеку
     while (!stack2.empty()) {
        TreeNode* node = stack2.top();
        stack2.pop();
        std::cout << node->val << " ";
     }
    }
};
```

3. Написати функцію для визначення кількості входження елементів більших за k до невпорядкованого бінарного дерева, що зберігається у стандартній формі.
4. Написати функцію для визначення кількості входження елементів менших за k до невпорядкованого бінарного дерева,

що зберігається у стандартній формі.
5. Написати функцію для перевірки входження значення k до елементів невпорядкованого бінарного дерева, що зберігається у стандартній формі.

```
#include <iostream>

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class BinaryTree {
public:
    TreeNode* root;

    BinaryTree() : root(nullptr) {}

    // Рекурсивна функція для підрахунку кількості входжень елементів більших за k
    int countGreater(TreeNode* node, int k) {
        if (node == nullptr) {
            return 0;
        }

        int count = 0;
        if (node->val > k) {
            //node->val < k - для менших за k, node->val == k - для входжень k,
            count = 1;
        }

        return count + countGreater(node->left, k) + countGreater(node->right, k);
    }

    // Зовнішня функція для визначення кількості входжень елементів більших за k
    int countGreaterThan(int k) {
        return countGreater(root, k);
    }
};
```

6. Написати функцію для знаходження у невпорядкованому бінарному дереві, що зберігається у “стандартній формі”, вершини зі значенням v та рівня, де розташована ця вершина.

```
    std::pair<int, int> findNodeLevel(int target) {
```

```
    if (root == nullptr)
        return std::make_pair(-1, -1);

    std::queue<std::pair<TreeNode*, int>> q;
    q.push(std::make_pair(root, 0));

    while (!q.empty()) {
        TreeNode* node = q.front().first;
        int level = q.front().second;
        q.pop();

        if (node->val == target)
            return std::make_pair(target, level);

        if (node->left != nullptr)
            q.push(std::make_pair(node->left, level + 1));
        if (node->right != nullptr)
            q.push(std::make_pair(node->right, level + 1));
    }

    return std::make_pair(-1, -1);
}
```

7. Написати функцію для визначення кількості входжень елементів із значенням з інтервалу [u, v] до невпорядкованого бінарного дерева, що зберігається у стандартній формі.

```
int countInRange(TreeNode* node, int u, int v) {
    if (node == nullptr) {
        return 0;
    }

    int count = 0;

    // Якщо значення в поточному вузлі знаходиться в межах інтервалу [u, v],
    // збільшуємо лічильник
    if (node->val >= u && node->val <= v) {
        count++;
    }

    // Рекурсивно обходимо ліве піддерево, якщо інтервал [u, v] перетинається з лівим піддеревом
    count += countInRange(node->left, u, v);
    // Рекурсивно обходимо праве піддерево, якщо інтервал [u, v] перетинається з правим піддеревом
    count += countInRange(node->right, u, v);

    return count;
}

int countInRange(int u, int v) {
    return countInRange(root, u, v);
}
```

8. Написати функцію для визначення кількості листів з відмітками, що належать інтервалу [u, v], у невпорядкованого бінарного дерева, що зберігається у стандартній формі.

```
int countLeavesInRange(TreeNode* node, int u, int v) {
    if (node == nullptr) {
```

```
        return 0;
    }

    // Якщо це листок, перевіряємо, чи його значення належить інтервалу [u, v]
    if (node->left == nullptr && node->right == nullptr) {
        if (node->val >= u && node->val <= v) {
            return 1;
        }
        else {
            return 0;
        }
    }

    // Рекурсивно обходимо ліве та праве піддерева
    int leftLeaves = countLeavesInRange(node->left, u, v);
    int rightLeaves = countLeavesInRange(node->right, u, v);

    // Повертаємо суму листків обох піддерев
    return leftLeaves + rightLeaves;
}

int countLeavesInRange(int u, int v) {
    return countLeavesInRange(root, u, v);
}
```

9. Написати функцію, яка визначає кількість внутрішніх вершин бінарного дерева, що представлене у стандартній формі

```
int countInternalNodes(TreeNode* node) {
    if (node == nullptr) {
        return 0;
    }

    // Якщо це листок, вертаємо 0
    if (node->left == nullptr && node->right == nullptr) {
        return 0;
    }

    // Рекурсивно обходимо ліве та праве піддерева
    // та додаємо 1 за поточну вершину
    return 1 + countInternalNodes(node->left) + countInternalNodes(node->right);
}

int countInternalNodes() {
    return countInternalNodes(root);
}
```

10. Написати функцію для визначення висоти невпорядкованого бінарного дерева, що зберігається у стандартній формі.

```
int height_help(TreeNode* node) {
    if (node == nullptr) {
        return -1; // Висота пустого дерева -1
```

```
        }

        // Рекурсивно знаходимо висоту лівого та правого піддерева
        int leftHeight = height_help(node->left);
        int rightHeight = height_help(node->right);

        // Повертаємо більшу з висот лівого та правого піддерева, плюс 1 за поточний вузол
        return std::max(leftHeight, rightHeight) + 1;
    }

    int height() {
        return height_help(root);
    }
```

11. Написати функію, що визначає кількість вузлів на шляху від корення дерева степеня 3, яке зберігається у стандартній формі, до вузла з заданим значенням v. Якщо таких вузлів декілька, обрати будь-яке з них.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <queue>

using namespace std;

class TreeNode {
public:
    int val;
    TreeNode* children[3];

    TreeNode(int value) : val(value) {
        for (int i = 0; i < 3; ++i) {
            children[i] = nullptr;
        }
    }
};

class Tree {
private:
    void add(TreeNode* node, int value) {
        if (node == nullptr) {
            cout << "Помилка: Спроба додати до нульового вузла." << endl;
            return;
        }

        TreeNode* newNode = new TreeNode(value);

        int randomChild = rand() % 3;

        if (node->children[randomChild]) {
            add(node->children[randomChild], value);
        }
        else {
            node->children[randomChild] = newNode;
        }
    }

    int countNodesOnPath(TreeNode* node, int target, bool& founded) {
        if (node == nullptr)
            return 0;

        if (node->val == target) {
            founded = true;
```

```
            return 0;
        }

        int subCount = 0;
        for (int i = 0; i < 3; ++i) {
            if (!founded)
                subCount = countNodesOnPath(node->children[i], target, founded);
        }

        if (founded)
            return 1 + subCount;
        else
            return 0;
    }

public:
    TreeNode* root;
    Tree() : root(nullptr) {
        srand(time(nullptr));
    }

    void add(int value) {
        if (root == nullptr) {
            root = new TreeNode(value);
        }
        else {
            add(root, value);
        }
    }

    int countNodesOnPathToValue(int value) {
        bool found = false;
        int count = countNodesOnPath(root, value, found);
        if (found)
            return count;
        else
            return -1;
    }
};
```

12. Написати функцію для визначення висоти “дерева степеня 3”, що зберігається у “розширеній стандартній формі”.

```
    int getHeight() {
        return height(root);
    }
```

```
int height(TreeNode* node) {
    if (node == nullptr) {
        return 0;
    }

    int maxChildHeight = 0;
    for (int i = 0; i < 3; ++i) {
        int childHeight = height(node->children[i]);
        if (childHeight > maxChildHeight) {
            maxChildHeight = childHeight;
        }
    }

    return 1 + maxChildHeight;
}
```

Задачі на дерева пошуку

1. Побудувати дерево “двійкового пошуку” за заданою множиною цілих чисел й занумерувати його вершини згідно з обходом в “симетричному порядку”.

```
#include <iostream>
```

```
using namespace std;

// Вершина дерева
struct TreeNode {
    int value;
    TreeNode* left;
    TreeNode* right;
    int inOrderNumber;

    TreeNode(int val) : value(val), left(nullptr), right(nullptr), inOrderNumber(0) {}
};

// Клас дерева двійкового пошуку
class BST {
private:
    TreeNode* root;
    int currentNumber;

    // Вставка нового елемента в дерево
    TreeNode* insert(TreeNode* node, int value) {
        if (node == nullptr) {
            return new TreeNode(value);
        }
        if (value < node->value) {
            node->left = insert(node->left, value);
        }
        else {
            node->right = insert(node->right, value);
        }
        return node;
    }

    // Обхід в симетричному порядку (in-order traversal)
    void inOrderTraversal(TreeNode* node) {
        if (node == nullptr) return;
        inOrderTraversal(node->left);
        node->inOrderNumber = ++currentNumber;
        cout << "Node value: " << node->value << " -> InOrder Number: " << node->inOrderNumber
<< endl;
        inOrderTraversal(node->right);
    }

public:
    BST() : root(nullptr), currentNumber(0) {}

    // Додавання нового елемента в дерево
    void insert(int value) {
        root = insert(root, value);
    }

    // Запуск обходу в симетричному порядку
    void inOrderTraversal() {
        currentNumber = 0;
        inOrderTraversal(root);
    }

    // Функція для створення дерева з масиву
    void buildFromArray(int values[], int size) {
        for (int i = 0; i < size; ++i) {
            insert(values[i]);
        }
    }
};
```

2 Написати функцію для вставки вузла зі значенням v у бінарне дерево пошуку, якщо таке значення у ньому відсутнє

3. Написати функцію, для вилучення вузла зі значенням v з дерева бінарного пошуку, якщо таке значення у ньому присутнє.

```
#include <iostream>

using namespace std;

// Вершина дерева
struct TreeNode {
    int value;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}
};

// Клас дерева двійкового пошуку
class BST {
private:
    TreeNode* root;

    // Вставка нового елемента в дерево
    TreeNode* insert(TreeNode* node, int value) {
        if (node == nullptr) {
            return new TreeNode(value);
        }
        if (value < node->value) {
            node->left = insert(node->left, value);
        }
        else if (value > node->value) {
            node->right = insert(node->right, value);
        }
        return node;
    }

    // Пошук мінімального значення в піддереві
    TreeNode* findMin(TreeNode* node) {
        while (node->left != nullptr) {
            node = node->left;
        }
        return node;
    }

    // Видалення елемента з дерева
    TreeNode* remove(TreeNode* node, int value) {
        if (node == nullptr) return node;

        if (value < node->value) {
            node->left = remove(node->left, value);
        }
        else if (value > node->value) {
            node->right = remove(node->right, value);
        }
        else {
            // Вузол знайдений
            if (node->left == nullptr) {
                TreeNode* temp = node->right;
                delete node;
                return temp;
            }
            else if (node->right == nullptr) {
                TreeNode* temp = node->left;
                delete node;
                return temp;
            }
```

```
            // Вузол з двома дітьми
            TreeNode* temp = findMin(node->right);
            node->value = temp->value;
            node->right = remove(node->right, temp->value);
        }
        return node;
    }
public:
    BST() : root(nullptr) {}

    // Додавання нового елемента в дерево
    void insert(int value) {
        root = insert(root, value);
    }

    // Видалення елемента з дерева
    void remove(int value) {
        root = remove(root, value);
    }
};
```

4. Написати функцію для об’єднання двох дерев двійкового пошуку зі значенням у деревах відповідно менше K і більше дорівнює K.

```
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

TreeNode* find_max_node(TreeNode* root) {
    while (root->right != nullptr) {
        root = root->right;
    }
    return root;
}

TreeNode* merge_separated(TreeNode* less_root, TreeNode* greater_equal_root, int K) {
    if (less_root == nullptr) {
        return greater_equal_root;
    }
    TreeNode* max_less = find_max_node(less_root);
    max_less->right = greater_equal_root;
    return less_root;
}
```

5. Написати функцію для визначенням найбільшого значення у вузлах непорожнього дерева бінарного пошуку, що менше за k.

```
    int findMaxLessThanK(int k) {
        int maxVal = INT_MIN; // Початкове значення максимального значення

        // Пошук у дереві
        while (root) {
```

```
        // Якщо значення в поточному вузлі менше за k, зберегти його та перейти до правого піддерева
        if (root->value < k) {
            maxVal = root->value;
            root = root->right;
        }
        // Якщо значення в поточному вузлі більше або рівне k, перейти до лівого піддерева
        else {
            root = root->left;
        }
    }

    return maxVal;
}
```

6. Написати функцію для визначенням найменшого значення у вузлах непорожнього дерева бінарного пошуку, що більше за k.

```
int findMinGreaterThanK(int k) {
    int minVal = INT_MAX; // Початкове значення мінімального значення

    // Пошук у дереві
    while (root) {
        // Якщо значення в поточному вузлі більше або рівне k, зберегти його та перейти до лівого піддерева
        if (root->value >= k) {
            minVal = root->value;
            root = root->left;
        }
        // Якщо значення в поточному вузлі менше за k, перейти до правого піддерева
        else {
            root = root->right;
        }
    }

    return minVal;
}
```

7. Написати функцію, що обчислює суму числових значень у вершинах дерева двійкового пошуку більших за k.

```
int sumOfNodesGreaterThanK(TreeNode* node, int k) {
    if (node == nullptr) {
        return 0; // Базовий випадок: якщо дерево порожнє або дійшли до кінця гілки, повернути 0
    }

    // Ініціалізуємо суму
```

```
    int sum = 0;

    // Якщо значення поточного вузла більше за k, додаємо його до суми
    if (node->value > k) {
        sum += node->value;
    }

    // Рекурсивно обчислюємо суму для лівого та правого піддерева
    sum += sumOfNodesGreaterThanK(node->left, k);
    sum += sumOfNodesGreaterThanK(node->right, k);

    return sum;
}


int sum(int k) {
    return sumOfNodesGreaterThanK(root, k);
}
```

8.Написати функцію для визначення найменшого значення у вузлах непорожнього дерева бінарного пошуку

```
int findMinValue() {
    // Якщо дерево порожнє, повертаємо -1 або можемо обробити цей випадок по-іншому в залежності від потреб
    if (root == nullptr) {
        // Вибираємо -1 як позначку відсутності значень у порожньому дереві
        return -1;
    }

    // Проходимо до останнього листка лівого піддерева
    while (root->left != nullptr) {
        root = root->left;
    }

    // Повертаємо значення останнього листка
    return root->value;
}
```

9. Написати функцію для визначення кількості входжень додатних елементів до дерева бінарного пошуку

```
int countPositiveValues(TreeNode* Node) {
    if (Node == nullptr) {
        return 0; // Базовий випадок: якщо дерево порожнє, повернути 0
    }

    // Ініціалізуємо лічильник кількості додатних значень
    int count = 0;

    // Рекурсивно обходимо дерево та підраховуємо кількість додатних значень
    count += countPositiveValues(Node->left);
    if (Node->value > 0) {
        count++;
    }
    count += countPositiveValues(Node->right);

    return count;
}
int count() {
    return countPositiveValues(root);
}
```

10. Множина цілих чисел представлена деревом двійкового пошуку. Написати нерекурсивну функцію, що записує у масив цю множину чисел впорядковану за зростанням.

```
#include <stack>

vector<int> inorderTraversal() {
    vector<int> result; // Масив для зберігання впорядкованої множини чисел
    stack<TreeNode*> s; // Стек для ітеративного обходу дерева

    TreeNode* current = root;

    // Ітеративний обхід дерева
    while (current != nullptr || !s.empty()) {
        // Додаємо всі ліві дочірні вузли в стек
        while (current != nullptr) {
            s.push(current);
            current = current->left;
        }

        // Піднімаємося до кореня зі стеку і додаємо його значення до результату
        current = s.top();
        s.pop();
        result.push_back(current->value);

        // Переходимо до правого піддерева
        current = current->right;
    }

    return result;
}
```

11. Написати функцію для копіювання AVL-дерева (+проста реалізація)

```
#include <iostream>
#include <algorithm>
// Визначення вузла AVL-дерева
struct TreeNode {
    int key, height;
    TreeNode* left; TreeNode* right;

    TreeNode(int k) : key(k), height(1), left(nullptr), right(nullptr) {}
};

// Клас для AVL-дерева
class AVLTree {
private:
    TreeNode* root;

    // Отримання висоти вузла
    int getHeight(TreeNode* node) {
        return node ? node->height : 0;
    }

    // Оновлення висоти вузла
    void updateHeight(TreeNode* node) {
        if (node) {
            node->height = 1 + std::max(getHeight(node->left), getHeight(node->right));
        }
    }

    // Отримання балансу вузла
    int getBalance(TreeNode* node) {
        return node ? getHeight(node->left) - getHeight(node->right) : 0;
    }
```

```
// Правий поворот
TreeNode* rightRotate(TreeNode* y) {
    TreeNode* x = y->left;
    TreeNode* T2 = x->right;

    x->right = y;
    y->left = T2;

    updateHeight(y);
    updateHeight(x);

    return x;
}

// Лівий поворот
TreeNode* leftRotate(TreeNode* x) {
    TreeNode* y = x->right;
    TreeNode* T2 = y->left;

    y->left = x;
    x->right = T2;

    updateHeight(x);
    updateHeight(y);

    return y;
}

// Вставка ключа у AVL-дерево
TreeNode* insert(TreeNode* node, int key) {
    if (!node) return new TreeNode(key);

    if (key < node->key) {
        node->left = insert(node->left, key);
    }
    else if (key > node->key) {
        node->right = insert(node->right, key);
    }
    else {
        return node; // однакові ключі не допускаються
    }

    updateHeight(node);

    int balance = getBalance(node);

    if (balance > 1 && key < node->left->key) {
        return rightRotate(node);
    }

    if (balance < -1 && key > node->right->key) {
        return leftRotate(node);
    }

    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
```

```
        return node;
    }

    // Функція копіювання вузла
    TreeNode* copyTree(TreeNode* root) {
        if (!root) return nullptr;

        TreeNode* newNode = new TreeNode(root->key);
        newNode->height = root->height;
        newNode->left = copyTree(root->left);
        newNode->right = copyTree(root->right);

        return newNode;
    }

    // Видалення дерева (рекурсивна допоміжна функція)
    void deleteTree(TreeNode* node) {
        if (node) {
            deleteTree(node->left);
            deleteTree(node->right);
            delete node;
        }
    }

public:
    AVLTree() : root(nullptr) {}

    ~AVLTree() {
        deleteTree(root);
    }

    void insert(int key) {
        root = insert(root, key);
    }

    AVLTree copy() {
        AVLTree newTree;
        newTree.root = copyTree(root);
        return newTree;
    }

    void printInOrder(TreeNode* node) const {
        if (node != nullptr) {
            printInOrder(node->left);
            std::cout << node->key << " ";
            printInOrder(node->right);
        }
    }

    void printInOrder() const {
        printInOrder(root);
    }
};
```

Задачі на графи
Структура суміжності

```
#include <iostream>
#include <list>
#include <vector>

using namespace std;

class Graph_s {
private:
```

```
    int numOfVertex;
    vector<list<int>> adjList;

public:
    Graph_s(int V) {
        this->numOfVertex = V;
        adjList.resize(V);
    }

    void addEdge(int v, int w) {
        adjList[v].push_back(w);
        adjList[w].push_back(v);
    }
};
```

1. Написати функцію для побудови кістякового дерева графа пошуком в глибину. Граф представлений структурою суміжності.

```
    void spanning_tree_dfs(int vertex, vector<bool>& visited, Graph_s& spanningTree) {
        visited[vertex] = true;

        for (int neighbor : this->adjList[vertex]) {
            if (!visited[neighbor]) {
                spanningTree.addEdge(vertex, neighbor);
                spanning_tree_dfs(neighbor, visited, spanningTree);
            }
        }
    }

    Graph_s build_spanning_tree() {
        Graph_s spanningTree(this->numOfVertex);
        vector<bool> visited(this->numOfVertex, false);

        for (int i = 0; i < this->numOfVertex; i++) {
            if (!visited[i]) {
                spanning_tree_dfs(i, visited, spanningTree);
            }
        }

        return spanningTree;
    }
```

2.Написати функцію, яка перевіряє зв’язність неорієнтованого графа, поданого структурою суміжності.

```
    void is_connected_dfs(int vertex, vector<bool>& visited) {
        visited[vertex] = true;

        for (int neighbor : this->adjList[vertex]) {
            if (!visited[neighbor]) {
                is_connected_dfs(neighbor, visited);
            }
        }
```

```
}

bool is_connected() {
    vector<bool> visited(this->numOfVertex, false);

    int start_vertex = 0;
    is_connected_dfs(start_vertex, visited);

    for (int i = 0; i < numOfVertex; i++) {
        if (!visited[i]) return false;
    }

    return true;
}
```

3. Написати функцію, яка визначає кількість ізольованих вершин неорієнтованого графа, поданого структурою суміжності.

```
int countIsolatedVertices() {
    int isolatedVertices = 0;
    for (const auto& adj : adjList) {
        if (adj.empty()) // Перевіряємо, чи список суміжних вершин порожній
            isolatedVertices++;
    }
    return isolatedVertices;
}
```

4. Написати функцію, яка визначає кількість вершин степеня 3 неорієнтованого графа, поданого структурою суміжності

```
int countDegreeThreeVertices() {
    int degreeThreeVertices = 0;
    for (int v = 0; v < numOfVertex; ++v) {
        if (adjList[v].size() == 3) // Перевіряємо, чи вершина має степінь 3
            degreeThreeVertices++;
    }
    return degreeThreeVertices;
}
```

5. Написати функцію, яка визначає кількість висячих вершин неорієнтованого графа, поданого структурою суміжності

```
int countEndVertices() {
    int degreeThreeVertices = 0;
    for (int v = 0; v < numOfVertex; ++v) {
        if (adjList[v].size() == 1) // Перевіряємо, чи вершина має степінь 1
            degreeThreeVertices++;
    }
    return degreeThreeVertices;
}
```

6. Написати функцію, яка визначає кількість ребер неорієнтованого графа, поданого структурою суміжності

```
int countEdges() {
    int edgeCount = 0;
    for (int v = 0; v < numOfVertex; ++v) {
        edgeCount += adjList[v].size();
    }
    return edgeCount / 2; // Ділимо на 2, бо граф неорієнтований, і кожне ребро враховується двічі
}
```

7. Написати функцію, яка перевіряє зв’язний неорієнтованого графа, поданий структурою суміжності на ейлеровість.

```
bool is_eulerian() {
    //Перевірити зв'язність графа. Якщо не зв'язний поe=вернути false;

    int oddDegreeVertices = 0;
    for (int v = 0; v < numOfVertex; ++v) {
        if (adjList[v].size() % 2 != 0) // Перевіряємо, чи вершина має степінь 2
            oddDegreeVertices++;
    }

    // Граф є ейлеровим, якщо всі вершини мають парний степінь (Eulerian cycle)
    // або рівно дві вершини мають непарний степінь (Eulerian path)
    if (oddDegreeVertices == 0 || oddDegreeVertices == 2) {
        return true;
    }

    return false;
}
```

8. Знайти всі вершини графа, що поданий структурою суміжності, які досяжні від заданої вершини.

```
void getReachableVerticesDFS(int currVertex, vector<bool>& visited, vector<int>& reachableVertices) {
    visited[currVertex] = true;
    reachableVertices.push_back(currVertex);

    for (int neighbor : adjList[currVertex]) {
        if (!visited[neighbor]) {
            getReachableVerticesDFS(neighbor, visited, reachableVertices);
```

```
        }
    }
}

vector<int> getReachableVertices(int startVertex) {
    vector<bool> visited(numOfVertex, false);
    vector<int> reachableVertices;
    getReachableVerticesDFS(startVertex, visited, reachableVertices);
    return reachableVertices;
}
```

9. Написати функцію, яка перевіряє чи є заданий граф транзитивним (для довільних вершин u, v, w, якщо u, v – суміжні, а також v, w – суміжні, суміжним є u та w)

```
bool isTransitive() {
    for (int i = 0; i < numOfVertex; ++i) {
        for (int j : adjList[i]) {
            for (int k : adjList[j]) {
                if (i != k && !hasEdge(i, k)) {
                    return false;
                }
            }
        }
    }
    return true;
}

bool hasEdge(int u, int v) {
    for (int x : adjList[u]) {
        if (x == v) {
            return true;
        }
    }
    return false;
}
```

10. Написати функцію, яка визначає кількість ребер у доповнені неорієнтованого графа, поданого структурою суміжності.

```
int countComplementEdges() {
    int totalEdges = numOfVertex * (numOfVertex - 1) / 2; // Загальна кількість можливих ребер у повному графі
    int originalEdges = countEdges(); // Кількість ребер у вихідному графі
    return totalEdges - originalEdges; // Повертаємо різницю між загальною кількістю ребер і кількістю ребер у вихідному графі
}
```

11. Написати функцію, яка перевіряє суміжність двох заданих вершин у неорієнтованому графі, що поданий структурою суміжності.

```
bool isAdjacent(int v, int w) {
    for (int x : adjList[v]) {
        if (x == w) {
            return true;
        }
    }
    return false;
```

```
    }
```

12. Написати функцію, яка за структурою суміжності графа будує його матрицю суміжності.

```
    int getNumOfVertex() const {
        return numOfVertex;
    }

    const vector<list<int>>& getAdjList() const {
        return adjList;
    }

Graph_m convertToAdjMatrix(Graph_s& g) {
    int numOfVertex = g.getNumOfVertex();
    const vector<list<int>>& adjList = g.getAdjList();
    Graph_m newGraph(numOfVertex);
    for (int v = 0; v < numOfVertex; ++v) {
        for (auto x : adjList[v])
            newGraph.addEdge(v, x);
    }
    return newGraph;
}
```

13. Знайти всі вершини графа, що поданий “структурою суміжності”, до яких існує шлях заданої довжини d від заданої вершини.

```
class Graph_s {
private:
    int numOfVertex;
    vector<list<pair<int, int>>> adjList;

public:
    Graph_s(int V) {
        this->numOfVertex = V;
        adjList.resize(V);
```

```
    }

    void addEdge(int v, int w, int weight) {
        adjList[v].push_back(make_pair(w, weight));
        adjList[w].push_back(make_pair(v, weight));
    }

    vector<int> findNodesWithGivenDistance(int src, int d) {
        vector<int> nodes;

        vector<bool> visited(numOfVertex, false);
        vector<int> distance(numOfVertex, 0);

        queue<int> q;
        q.push(src);
        visited[src] = true;

        while (!q.empty()) {
            int u = q.front();
            q.pop();

            for (auto neighbor : adjList[u]) {
                int v = neighbor.first;
                int weight = neighbor.second;

                if (!visited[v]) {
                    visited[v] = true;
                    distance[v] = distance[u] + weight;
                    q.push(v);
                    if (distance[v] == d)
                        nodes.push_back(v);
                }
            }
        }

        return nodes;
    }
};
```

14. Написати функцію, яка вилучає всі петлі “псевдо графа”, поданого “структурою суміжності”.’

```
#include <iostream>
#include <list>
#include <vector>
#include <set>

using namespace std;

class Graph_s {
private:
    int numOfVertex;
```

```
    vector<list<pair<int, int>>> adjList;

public:
    Graph_s(int V) {
        this->numOfVertex = V;
        adjList.resize(V);
    }

    void addEdge(int v, int w, int weight) {
        adjList[v].push_back(make_pair(w, weight));
        adjList[w].push_back(make_pair(v, weight));
    }

    void removeDuplicateEdges() {
        for (int i = 0; i < numOfVertex; ++i) {
            set<pair<int, int>> seen;
            for (auto it = adjList[i].begin(); it != adjList[i].end(); ) {
                pair<int, int> edge = make_pair(min(i, it->first), max(i, it->first));
                if (seen.find(edge) != seen.end()) {
                    it = adjList[i].erase(it);
                }
                else {
                    seen.insert(edge);
                    ++it;
                }
            }
        }
    }
};
```

15. Написати функцію, яка перевіряє ациклічність орієнтованого графа, поданого структрою суміжності.

```
    bool isCyclicUtil(int v, vector<int>& color) {
        if (color[v] == 1) return true; // цикл знайдено
        if (color[v] == 2) return false; // вузол вже оброблений

        color[v] = 1; // позначити вузол як оброблюваний

        for (int neighbor : adjList[v]) {
            if (isCyclicUtil(neighbor, color)) {
                return true;
            }
```

```
        }

        color[v] = 2; // позначити вузол як оброблений
        return false;
    }

    bool isCyclic() {
        vector<int> color(numOfVertex, 0); // 0 = білий, 1 = сірий, 2 = чорний

        for (int i = 0; i < numOfVertex; i++) {
            if (color[i] == 0) { // якщо вузол не відвіданий
                if (isCyclicUtil(i, color)) {
                    return true;
                }
            }
        }
        return false;
    }
```

16. Написати функцію, яка для орієнтованого графа будує орієнтований граф з протилежною орієнтацією дуг. Графи представлені “структурами суміжності”.

```
    Graph_s getTranspose() {
        Graph_s g_t(numOfVertex);

        for (int v = 0; v < numOfVertex; ++v) {
            for (int neighbor : adjList[v]) {
                g_t.addEdge(neighbor, v);
            }
        }

        return g_t;
    }
```

Структура суміжності

```
#include <iostream>
#include <vector>

using namespace std;

class Graph_m {
private:
    int numOfVertex;
    vector<vector<int>> adjMatrix;

public:
    Graph_m(int V) {
```

```
        this->numOfVertex = V;
        adjMatrix.resize(V, vector<int>(V, 0));
    }

    void addEdge(int v, int w) {
        adjMatrix[v][w] = 1;
        adjMatrix[w][v] = 1;
    }
};
```

1. Написати функцію для побудови кістякового дерева графа пошуком в глибину. Граф представлений матрицею суміжності.

```
    void spanning_tree_dfs(int vertex, vector<bool>& visited, Graph_m& spanningTree) {
        visited[vertex] = true;

        for (int neighbor = 0; neighbor < numOfVertex; neighbor++) {
            if (adjMatrix[vertex][neighbor] == 1 && !visited[neighbor]) {
                spanningTree.addEdge(vertex, neighbor);
                spanning_tree_dfs(neighbor, visited, spanningTree);
            }
        }
    }

    Graph_m build_spanning_tree() {
        Graph_m spanningTree(this->numOfVertex);
        vector<bool> visited(this->numOfVertex, false);

        for (int i = 0; i < this->numOfVertex; i++) {
            if (!visited[i]) {
                spanning_tree_dfs(i, visited, spanningTree);
            }
        }

        return spanningTree;
    }
```

2.Написати функцію, яка перевіряє зв’язність неорієнтованого графа, поданого матрицею суміжності.

```
    void is_connected_dfs(int vertex, vector<bool>& visited) {
        visited[vertex] = true;

        for (int i = 0; i < numOfVertex; ++i) {
            if (adjMatrix[vertex][i] && !visited[i]) {
                is_connected_dfs(i, visited);
            }
        }
    }
```

```
bool is_connected() {
    vector<bool> visited(numOfVertex, false);

    int start_vertex = 0;
    is_connected_dfs(start_vertex, visited);

    for (int i = 0; i < numOfVertex; ++i) {
        if (!visited[i]) return false;
    }

    return true;
}
```

3. Написати функцію, яка визначає кількість ізольованих вершин неорієнтованого графа, поданого матрицею суміжності.

```
int countIsolatedVertices() {
    int isolatedVertices = 0;
    for (int i = 0; i < numOfVertex; ++i) {
        bool isIsolated = true;
        for (int j = 0; j < numOfVertex; ++j) {
            if (adjMatrix[i][j] == 1) {
                isIsolated = false;
                break;
            }
        }
        if (isIsolated)
            isolatedVertices++;
    }
    return isolatedVertices;
}
```

4. Написати функцію, яка визначає кількість вершин степеня 3 неорієнтованого графа, поданого матрицею суміжності

```
int countDegreeThreeVertices() {
    int degreeThreeVertices = 0;
    for (int v = 0; v < numOfVertex; ++v) {
        int degree = 0;
        for (int w = 0; w < numOfVertex; ++w) {
            degree += adjMatrix[v][w];
        }
        if (degree == 3) // Перевіряємо, чи вершина має степінь 3
            degreeThreeVertices++;
    }
    return degreeThreeVertices;
}
```

5. Написати функцію, яка визначає кількість висячих вершин неорієнтованого графа, поданого матрицею суміжності

```
int countDegreeThreeVertices() {
    int degreeThreeVertices = 0;
    for (int v = 0; v < numOfVertex; ++v) {
        int degree = 0;
        for (int w = 0; w < numOfVertex; ++w) {
            degree += adjMatrix[v][w];
        }
        if (degree == 1) // Перевіряємо, чи вершина має степінь 1
            degreeThreeVertices++;
    }
```

```
    return degreeThreeVertices;
}
```

6. Написати функцію, яка визначає кількість ребер неорієнтованого графа, поданого матрицею суміжності

```
int countEdges() {
    int edgeCount = 0;
    for (int v = 0; v < numOfVertex; ++v) {
        for (int w = v; w < numOfVertex; ++w) {
            if (adjMatrix[v][w] == 1) edgeCount++;
        }
    }
    return edgeCount; // Ділимо на 2, бо граф неорієнтований, і кожне ребро враховується двічі
}
```

7. Написати функцію, яка перевіряє зв’язний неорієнтованого графа, поданий матрицею суміжності на ейлеровість.

```
bool is_eulerian() {
    //Перевірити зв'язність графа. Якщо не зв'язний пое=вернути false;

    int oddDegreeVertices = 0;
    for (int v = 0; v < numOfVertex; ++v) {
        int degree = 0;
        for (int w = 0; w < numOfVertex; ++w) {
            degree += adjMatrix[v][w];
        }
        if (degree % 2 != 0) { // Перевіряємо, чи вершина має непарний степінь
            oddDegreeVertices++;
        }
    }

    // Граф є ейлеровим, якщо всі вершини мають парний степінь (Eulerian cycle)
    // або рівно дві вершини мають непарний степінь (Eulerian path)
    if (oddDegreeVertices == 0 || oddDegreeVertices == 2) {
        return true;
    }

    return false;
}
```

8. Знайти всі вершини графа, що поданий матрицею суміжності, які досяжні від заданої вершини.

```
void getReachableVerticesDFS(int currVertex, vector<bool>& visited, vector<int>& reachableVertices) {
    visited[currVertex] = true;
    reachableVertices.push_back(currVertex);

    for (int neighbor = 0; neighbor < numOfVertex; ++neighbor) {
        if (adjMatrix[currVertex][neighbor] == 1 && !visited[neighbor]) {
            getReachableVerticesDFS(neighbor, visited, reachableVertices);
        }
    }
```

```
    }

    vector<int> getReachableVertices(int startVertex) {
        vector<bool> visited(numOfVertex, false);
        vector<int> reachableVertices;
        getReachableVerticesDFS(startVertex, visited, reachableVertices);
        return reachableVertices;

    }
```

9. Написати функцію, яка перевіряє чи є заданий граф транзитивним (для довільних вершин u, v, w, якщо u, v – суміжні, а також v, w – суміжні, суміжним є u та w)

```
    void getReachableVerticesDFS(int currVertex, vector<bool>& visited, vector<int>&
reachableVertices) {
        visited[currVertex] = true;
        reachableVertices.push_back(currVertex);

        for (int neighbor : adjList[currVertex]) {
            if (!visited[neighbor]) {
                getReachableVerticesDFS(neighbor, visited, reachableVertices);
            }
        }
    }

    vector<int> getReachableVertices(int startVertex) {
        vector<bool> visited(numOfVertex, false);
        vector<int> reachableVertices;
        getReachableVerticesDFS(startVertex, visited, reachableVertices);
        return reachableVertices;
    }
```

10. Написати функцію, яка визначає кількість ребер у доповнені неорієнтованого графа, поданого матрицею суміжності.

```
    int countComplementEdges() {
        int totalEdges = numOfVertex * (numOfVertex - 1) / 2; // Загальна кількість можливих ребер
у повному графі
        int originalEdges = countEdges(); // Кількість ребер у вихідному графі
        return totalEdges - originalEdges; // Повертаємо різницю між загальною кількістю ребер і
кількістю ребер у вихідному графі
    }
```

11. Написати функцію, яка перевіряє суміжність двох заданих вершин у неорієнтованому графі, що поданий матрицею суміжності.

```
    bool isAdjacent(int v, int w) {
        return adjMatrix[v][w] == 1;
    }
```

12. Написати функцію, яка за матрицею суміжності графа будує його структуру суміжності.

```
    int getNumOfVertex() const {
        return numOfVertex;
```

```
    }

    const vector<vector<int>>& getAdjMatrix() const {
        return adjMatrix;
    }

Graph_s convertToAdjList(Graph_m& g) {
    int numOfVertex = g.getNumOfVertex();
    const vector<vector<int>>& adjMatrix = g.getAdjMatrix();
    Graph_s newGraph(numOfVertex);
    for (int v = 0; v < numOfVertex; ++v) {
        for (int w = 0; w < numOfVertex; ++w) {
            if (adjMatrix[v][w] == 1) {
                newGraph.addEdge(v, w);
            }
        }
    }
    return newGraph;
}
```

13. Написати функцію, яка для орієнтованого графа будує орієнтований граф з протилежною орієнтацією дуг. Графи представлені “матрицею суміжності”.

```
    Graph_m getTranspose() {
        Graph_m g_t(numOfVertex);

        for (int v = 0; v < numOfVertex; ++v) {
            for (int w = 0; w < numOfVertex; ++w) {
                if (adjMatrix[v][w] == 1) {
                    g_t.addEdge(w, v);
                }
            }
        }

        return g_t;
    }
```

14. Знайти всі вершини графа, що поданий “матрицею суміжності”, до яких існує шлях заданої довжини d від заданої вершини.

```
class Graph_m {
private:
    int numOfVertex;
    std::vector<std::vector<int>> adjMatrix;

public:
    Graph_m(int V) {
        this->numOfVertex = V;
        adjMatrix.resize(V, std::vector<int>(V, 0)); // Initialize adjacency matrix with all zeroes
    }
```

```
    void addEdge(int v, int w, int weight) {
        adjMatrix[v][w] = weight; // Since it's an undirected graph, set both directions
        adjMatrix[w][v] = weight;
    }

    std::vector<int> findNodesWithGivenDistance(int src, int d) {
        std::vector<int> nodes;

        std::vector<bool> visited(numOfVertex, false);
        std::vector<int> distance(numOfVertex, 0);

        std::queue<int> q;
        q.push(src);
        visited[src] = true;

        while (!q.empty()) {
            int u = q.front();
            q.pop();

            for (int v = 0; v < numOfVertex; ++v) {
                if (adjMatrix[u][v] != 0 && !visited[v]) { // If there's a connection and not visited
                    visited[v] = true;
                    distance[v] = distance[u] + adjMatrix[u][v];
                    q.push(v);
                    if (distance[v] == d)
                        nodes.push_back(v);
                }
            }
        }

        return nodes;
    }
};
```

15. Написати функцію, яка вилучає всі петлі “псевдо графа”, поданого “структурою суміжності”.’ - Складно :(

16. Написати функцію, яка перевіряє ациклічність орієнтованого графа, поданого матрицею суміжності.

```
class Graph_m {
private:
    int numOfVertex;
    vector<vector<int>> adjMatrix;

public:
    Graph_m(int V) {
        this->numOfVertex = V;
        adjMatrix.resize(V, vector<int>(V, 0));
    }

    void addEdge(int v, int w) {
        adjMatrix[v][w] = 1;
    }
```

```
    bool isCyclicUtil(int v, vector<int>& color) {
        if (color[v] == 1) return true; // цикл знайдено
        if (color[v] == 2) return false; // вузол вже оброблений

        color[v] = 1; // позначити вузол як оброблюваний

        for (int i = 0; i < numOfVertex; i++) {
            if (adjMatrix[v][i] != 0) { // існує ребро
                if (isCyclicUtil(i, color)) {
                    return true;
                }
            }
        }

        color[v] = 2; // позначити вузол як оброблений
        return false;
    }

    bool isCyclic() {
        vector<int> color(numOfVertex, 0); // 0 = білий, 1 = сірий, 2 = чорний

        for (int i = 0; i < numOfVertex; i++) {
            if (color[i] == 0) { // якщо вузол не відвіданий
                if (isCyclicUtil(i, color)) {
                    return true;
                }
            }
        }
        return false;
    }
};
```

Задачі на матриці

1.1 Написати функцію для побудови послідовно-зв’язного індексного зберігання розрідженої матриці B[10,40] за звичайним представленням двовимірним масивом.

```
#include <iostream>

struct Node {
    int row;
    int col;
    int value;
    Node* next;

    Node(int r, int c, int v) : row(r), col(c), value(v), next(nullptr) {}
};

class SparseMatrix {
```

```
private:
    Node** matrix;
    int rows, cols;

public:
    SparseMatrix(int rows, int cols) : rows(rows), cols(cols) {
        matrix = new Node * [cols]();
    }

    ~SparseMatrix() {
        for (int i = 0; i < cols; ++i) {
            Node* current = matrix[i];
            while (current != nullptr) {
                Node* toDelete = current;
                current = current->next;
                delete toDelete;
            }
        }
        delete[] matrix;
    }

    void insert(int row, int col, int value) {
        if (row >= rows || col >= cols) {
            std::cerr << "Invalid indices!" << std::endl;
            return;
        }

        if (matrix[col] == nullptr || matrix[col]->row > row) {
            Node* newNode = new Node(row, col, value);
            newNode->next = matrix[col];
            matrix[col] = newNode;
        }
        else {
            Node* current = matrix[col];
            while (current->next != nullptr && current->next->row < row) {
                current = current->next;
            }

            if (current->row == row) {
                current->value = value;
            }
            else {
                Node* newNode = new Node(row, col, value);
                newNode->next = current->next;
                current->next = newNode;
            }
        }
    }
};

SparseMatrix buildSparseMatrix(int B[10][40], int rows, int cols) {
    SparseMatrix sm(rows, cols);
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            if (B[i][j] != 0) {
                sm.insert(i, j, B[i][j]);
            }
        }
    }
    return sm;
}
```

2. Написати функцію для знаходження середнього арифметичного елементів розрідженої матриці B[20, 50] при послідовно-зв’язному

індексному зберіганні

```
double average() {
    int sum = 0;
    /*int count = 0;*/
    for (int i = 0; i < cols; ++i) {
        Node* current = matrix[i];
        while (current != nullptr) {
            sum += current->value;
           /* ++count;*/
            current = current->next;
        }
    }
    //if (count == 0) {
    //    std::cerr << "Matrix is empty!" << std::endl;
    //    return 0.0;
    //}

    return static_cast<double>(sum) / (rows*cols);
}
```

3. Написати функцію для знаходження максимального елемента розрідженої матриці B[20, 50] при послідовно-зв’язному індексному зберіганні

```
int maximum() {
    int max = std::numeric_limits<int>::min(); // Встановлюємо початкове значення max на найменшу можливу інтову величину
    for (int i = 0; i < cols; ++i) {
        Node* current = matrix[i];
        while (current != nullptr) {
            if (current->value > max) {
                max = current->value;
            }
            current = current->next;
        }
    }
    return max;
}
```

4. Написати функцію для знаходження мінімального елемента розрідженої матриці B[20, 50] при послідовно-зв’язному індексному зберіганні

```
int minimum() {
    int min = std::numeric_limits<int>::max(); // Встановлюємо початкове значення min на найбільшу можливу інтову величину
    for (int i = 0; i < cols; ++i) {
        Node* current = matrix[i];
        while (current != nullptr) {
            if (current->value < min) {
                min = current->value;
            }
            current = current->next;
        }
    }
    return min;
```

```
    }
```

5. Написати функцію для реалізації поелементного додавання двох розріджених матриць при послідовно-зв’язному індексному зберігані.

```
    Node**& getMatrix() {
        return matrix;
    }

    int getCol() {
        return cols;
    }

    int getRow() {
        return rows;
    }

SparseMatrix addTwoMatrix(SparseMatrix& a, SparseMatrix& b) {
    if (a.getRow() != b.getRow() || a.getCol() != b.getCol()) {
        std::cerr << "Matrix dimensions mismatch!" << std::endl;
        return SparseMatrix(0, 0);
    }

    SparseMatrix result(a.getRow(), a.getCol());

    for (int i = 0; i < a.getCol(); ++i) {
        Node* currentA = a.getMatrix()[i];
        Node* currentB = b.getMatrix()[i];

        while (currentA != nullptr && currentB != nullptr) {
            if (currentA->row == currentB->row) {
                result.insert(currentA->row, i, currentA->value + currentB->value);
                currentA = currentA->next;
                currentB = currentB->next;
            }
            else if (currentA->row < currentB->row) {
                result.insert(currentA->row, i, currentA->value);
                currentA = currentA->next;
            }
            else {
                result.insert(currentB->row, i, currentB->value);
                currentB = currentB->next;
            }
        }

        while (currentA != nullptr) {
            result.insert(currentA->row, i, currentA->value);
            currentA = currentA->next;
        }

        while (currentB != nullptr) {
            result.insert(currentB->row, i, currentB->value);
            currentB = currentB->next;
        }
    }

    return result;
}
```

7. Написати функцію для знаходження номера рядка матриці з максимальною кількістю 0 елементів у розрідженій матриці B[20, 50] при зв’язному стислому

зберіганні.

```
int findRowWithMaxZeros() {
    int zero_count[ROWS] = { 0 };

    // Initialize zero_count with the number of columns
    for (int i = 0; i < rows; ++i) {
        zero_count[i] = cols;
    }

    // Traverse the matrix to adjust zero counts
    for (int i = 0; i < cols; ++i) {
        Node* current = matrix[i];
        while (current != nullptr) {
            zero_count[current->row]--; // Decrement for each non-zero value
            current = current->next;
        }
    }

    // Find the row with the maximum number of zeros
    int max_zeros = zero_count[0];
    int max_row = 0;
    for (int i = 1; i < rows; ++i) {
        if (zero_count[i] > max_zeros) {
            max_zeros = zero_count[i];
            max_row = i;
        }
    }

    return max_row;
}
```

Задачі на стек

1. Стек S у зв’язному зберіганні містить цілі числа. Створити два стеки S1 і S2 перемістивши до першого всі парні числа, а до другого - непарні, зберігаючи попередній взаємний порядок.

```
#include <iostream>

#define MAX_SIZE 500

class Stack {
private:
    int top;
    int data[MAX_SIZE];

public:
    Stack() {
```

```
        top = -1;
    }

    Stack& operator=(const Stack& other) {
        // Перевірка на самокопіювання
        if (this == &other)
            return *this;

        // Очищаємо стек
        top = -1;

        // Копіюємо значення з other.data до data, зберігаючи порядок
        for (int i = 0; i <= other.top; ++i) {
            data[i] = other.data[i];
            top++;
        }

        return *this;
    }

    bool isEmpty() {
        return top == -1;
    }

    bool isFull() {
        return top == MAX_SIZE - 1;
    }

    void push(int value) {
        if (!isFull()) {
            top++;
            data[top] = value;
        }
        else {
            std::cout << "Stack Overflow!" << std::endl;
        }
    }

    int pop() {
        if (!isEmpty()) {
            int temp = data[top];
            top--;
            return temp;
        }
        else {
            std::cout << "Stack Underflow!" << std::endl;
            return -1; // Значення -1, що вказує про помилку, можна змінити на власний вибір.
        }
    }

    int front() {
        if (!isEmpty()) {
            return data[top];
        }
        else {
            std::cout << "Stack is Empty!" << std::endl;
            return -1; // Аналогічно, можна змінити на власний вибір.
        }
    }
};

void splitStack(Stack& S, Stack& S1, Stack& S2) {
    Stack tempStack;

    while (!S.isEmpty()) {
```

```
        tempStack.push(S.pop()); // Копіюємо значення в тимчасовий стек
    }

    while (!tempStack.isEmpty()) {
        int current = tempStack.pop();

        if (current % 2 == 0) {
            S1.push(current); // Додаємо парне число до S1
        }
        else {
            S2.push(current); // Додаємо непарне число до S2
        }

        S.push(current);
    }
}
```

2. Написати функцію для перетворення виразу з цілих чисел та операцій “-”, “/”, записаного у звичайній інфіксній формі у представлення у формі ПОЛІЗ. Потрібний стек реалізувати власноруч.

```
int prec(char c) {
    if (c == '^')
        return 3;
    else if (c == '/' || c == '*')
        return 2;
    else if (c == '+' || c == '-')
        return 1;
    else
        return -1;
}

char* infixToPostfix(const char s[]) {
    static char result[1000];
    int resultIndex = 0;
    int len = strlen(s);
    Stack stack;

    bool prevWasOperator = true; // Позначаємо, що перед поточним символом був оператор або початок рядка

    for (int i = 0; i < len; i++) {
        char c = s[i];

        if (c == ' ')
            continue;

        if (isdigit(c) || (c == '-' && prevWasOperator)) {
            // Якщо це цифра або унарний мінус
            if (c == '-') {
                result[resultIndex++] = c;
                i++;
                while (i < len && s[i] == ' ') {
                    i++;
                }
            }
            while (i < len && (isdigit(s[i]) || (s[i] == '.' && isdigit(s[i + 1])))) {
                result[resultIndex++] = s[i++];
            }
            result[resultIndex++] = ' ';
            i--; // Adjust for the next iteration of the loop
            prevWasOperator = false;
        }
        else if (isalpha(c)) {
```

```
            result[resultIndex++] = c;
            result[resultIndex++] = ' '; // Add a space to separate operands
            prevWasOperator = false;
        }
        else if (c == '(') {
            stack.push(c);
            prevWasOperator = true;
        }
        else if (c == ')') {
            while (!stack.isEmpty() && stack.front() != '(') {
                result[resultIndex++] = stack.pop();
                result[resultIndex++] = ' ';
            }
            stack.pop();
            prevWasOperator = false;
        }
        else {
            while (!stack.isEmpty() &&
                ((prec(c) < prec(stack.front())) ||
                    (prec(c) == prec(stack.front()) && c != '^'))) {
                result[resultIndex++] = stack.pop();
                result[resultIndex++] = ' ';
            }
            stack.push(c);
            prevWasOperator = true;
        }
    }

    while (!stack.isEmpty()) {
        result[resultIndex++] = stack.pop();
        result[resultIndex++] = ' ';
    }

    // Remove the trailing space
    if (resultIndex > 0 && result[resultIndex - 1] == ' ') {
        result[--resultIndex] = '\0';
    }
    else {
        result[resultIndex] = '\0';
    }

    return result;
}
```

3. Написати функцію для обчислення значення виразу з цілих чисел та операцій “+”, “*”, записаного у формі ПОЛІЗ. Потрібний стек реалізувати власноруч.

```
int evaluatePostfix(const char* expression) {
    Stack stack;
    int len = strlen(expression);

    for (int i = 0; i < len; i++) {
        char c = expression[i];

        if (c == ' ')
            continue;

        if (isdigit(c) || (c == '-' && isdigit(expression[i + 1]))) {
            // Якщо це число або унарний мінус
            int num = 0;
            int sign = 1;
```

```
            if (c == '-') {
                sign = -1;
                i++;
            }
            while (i < len && isdigit(expression[i])) {
                num = num * 10 + (expression[i++] - '0');
            }
            i--; // Коригуємо для наступної ітерації
            stack.push(num * sign);
        }
        else if (c == '+') {
            int val1 = stack.pop();
            int val2 = stack.pop();
            stack.push(val2 + val1);
        }
        else if (c == '*') {
            int val1 = stack.pop();
            int val2 = stack.pop();
            stack.push(val2 * val1);
        }
    }

    return stack.pop();
}
```