Міністерство освіти і науки України

Національний університет «Львівська політехніка»

Кафедра систем штучного інтелекту

Звіт

про виконання лабораторних та практичних робіт блоку № 6

На тему: “Динамічні структури (Черга, Стек, Списки, Дерево). Алгоритми обробки динамічних структур.”.

З дисципліни: «Основи програмування»

до:

Практичних Робіт до блоку № 6

Виконав:

Студент групи ШІ-11

Голейчук Іван Миколайович

Львів 2024

Тема роботи:
"Основи динамічних структур даних та їх реалізація у програмуванні"

Мета роботи:
Дослідити принципи роботи з динамічними структурами даних, включаючи стек, чергу, зв’язні списки та дерева. Вивчити їхню реалізацію, основні операції, алгоритми обробки та практичне застосування. Ознайомитися з особливостями управління пам’яттю в контексті динамічних структур та їхнього впливу на ефективність програм.

Теоретичні відомості:

Основи Динамічних Структур Даних:
Вступ до динамічних структур даних: визначення та важливість
Виділення пам'яті для структур даних (stack і heap)
Приклади простих динамічних структур: динамічний масив
Стек:
Визначення та властивості стеку
Операції push, pop, top: реалізація та використання
Приклади використання стеку: обернений польський запис, перевірка балансу дужок
Переповнення стеку
Черга:
Визначення та властивості черги
Операції enqueue, dequeue, front: реалізація та застосування
Приклади використання черги: обробка подій, алгоритми планування
Розширення функціоналу черги: пріоритетні черги
Зв'язні Списки:
Визначення однозв'язного та двозв'язного списку
Принципи створення нових вузлів, вставка між існуючими, видалення, створення кільця(circular linked list)
Основні операції: обхід списку, пошук, доступ до елементів, об'єднання списків
Приклади використання списків: управління пам'яттю, FIFO та LIFO структури
Дерева:
Вступ до структури даних "дерево": визначення, типи
Бінарні дерева: вставка, пошук, видалення
Обхід дерева: в глибину (preorder, inorder, postorder), в ширину
Застосування дерев: дерева рішень, хеш-таблиці
Складніші приклади дерев: AVL, Червоно-чорне дерево
Алгоритми Обробки Динамічних Структур:
Основи алгоритмічних патернів: ітеративні, рекурсивні
Алгоритми пошуку, сортування даних, додавання та видалення елементів

Індивідуальний план опрацювання теорії:

Основи Динамічних Структур Даних

Вступ до динамічних структур даних

Що опрацьовано:
Визначено поняття динамічних структур даних та їх важливість у програмуванні. Розглянуто, як ці структури забезпечують гнучке використання пам’яті та адаптивність до змін розміру.
Джерела інформації:

- Лекції Олександра Пшеничного
- Практичні заняття
- Використання штучного інтелекту (чат GPT)

Виділення пам’яті для структур даних (stack і heap)

Що опрацьовано:
Розглянуто різницю між стеком (stack) та купою (heap) для зберігання динамічних структур. Вивчено основи роботи з динамічним виділенням та звільненням пам’яті (`new`, `delete`).
Джерела інформації:

- Лекції Олександра Пшеничного
- Youtube

Приклади простих динамічних структур: динамічний масив

Що опрацьовано:
Вивчено створення динамічного масиву, реалізацію його розширення, копіювання та управління пам’яттю.
Джерела інформації:

- Практичні заняття
- Використання штучного інтелекту

Стек

Визначення та властивості стеку

Що опрацьовано:
Розглянуто принципи роботи стеку за схемою LIFO (Last In, First Out).

Операції push, pop, top: реалізація та використання

Що опрацьовано:
Реалізовано основні операції стеку та їх використання у практичних задачах.

Приклади використання стеку

Що опрацьовано:
Розглянуто використання стеку для оберненого польського запису, перевірки балансу дужок у виразах.

Переповнення стеку

Що опрацьовано:
Вивчено причини та способи уникнення переповнення стеку.
Джерела інформації:

- Лекції Олександра Пшеничного
- Практичні заняття
- Youtube

Черга

Визначення та властивості черги

Що опрацьовано:
Ознайомлено з принципами роботи черги за схемою FIFO (First In, First Out).

Операції enqueue, dequeue, front

Що опрацьовано:
Вивчено реалізацію та практичне застосування основних операцій черги.

Приклади використання черги

Що опрацьовано:
Досліджено використання черги для обробки подій та алгоритмів планування.

Розширення функціоналу черги

Що опрацьовано:
Ознайомлено з пріоритетними чергами для задач, що вимагають додаткових умов сортування.
Джерела інформації:

- Практичні заняття
- Використання штучного інтелекту

Зв’язні списки

Визначення однозв’язного та двозв’язного списку

Що опрацьовано:
Ознайомлено з відмінностями між однозв’язними та двозв’язними списками.

Принципи створення вузлів, вставка, видалення

Що опрацьовано:
Розглянуто базові операції для управління елементами списків.

Основні операції

Що опрацьовано:
Реалізовано обхід, пошук, об’єднання списків, створення кільцевих списків.

Приклади використання списків

Що опрацьовано:
Розглянуто використання списків у задачах управління пам’яттю та реалізації FIFO/LIFO структур.
Джерела інформації:

- Лекції Олександра Пшеничного
- Практичні заняття

Дерева

Вступ до структури "дерево"

Що опрацьовано:
Вивчено основи дерев, їх типи та властивості.

Бінарні дерева: вставка, пошук, видалення

Що опрацьовано:
Розглянуто базові операції для роботи з бінарними деревами.

Обхід дерева

Що опрацьовано:
Вивчено алгоритми обходу дерев: в глибину (preorder, inorder, postorder) та в ширину.

Застосування дерев

Що опрацьовано:
Розглянуто практичне використання дерев для хеш-таблиць та дерев рішень.

Складніші приклади дерев

Що опрацьовано:
Ознайомлено зі структурами AVL та червоно-чорних дерев.
Джерела інформації:

- Youtube
- Використання штучного інтелекту

Алгоритми обробки динамічних структур

Основи алгоритмічних патернів

Що опрацьовано:
Вивчено ітеративні та рекурсивні підходи.

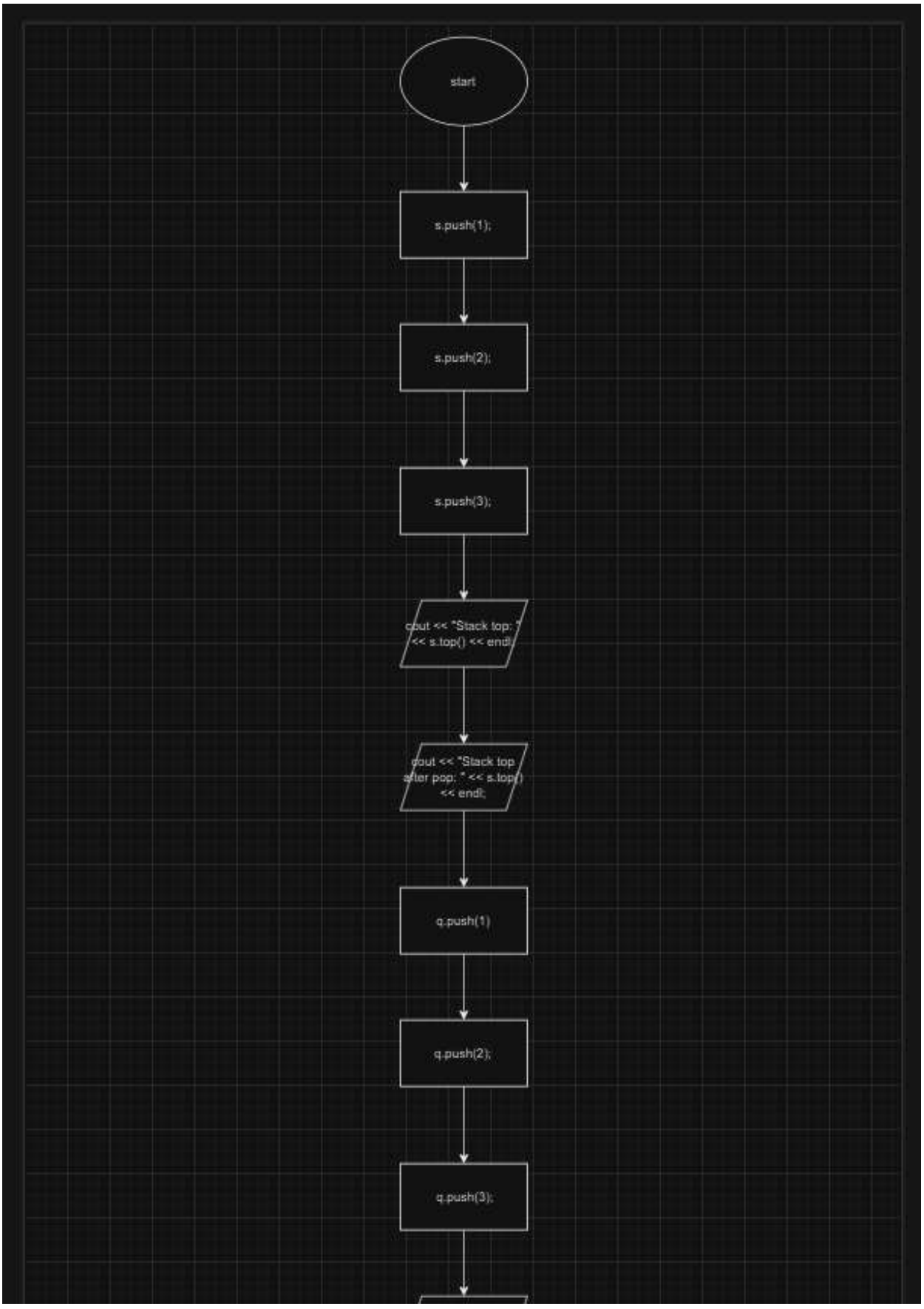
Алгоритми пошуку, сортування

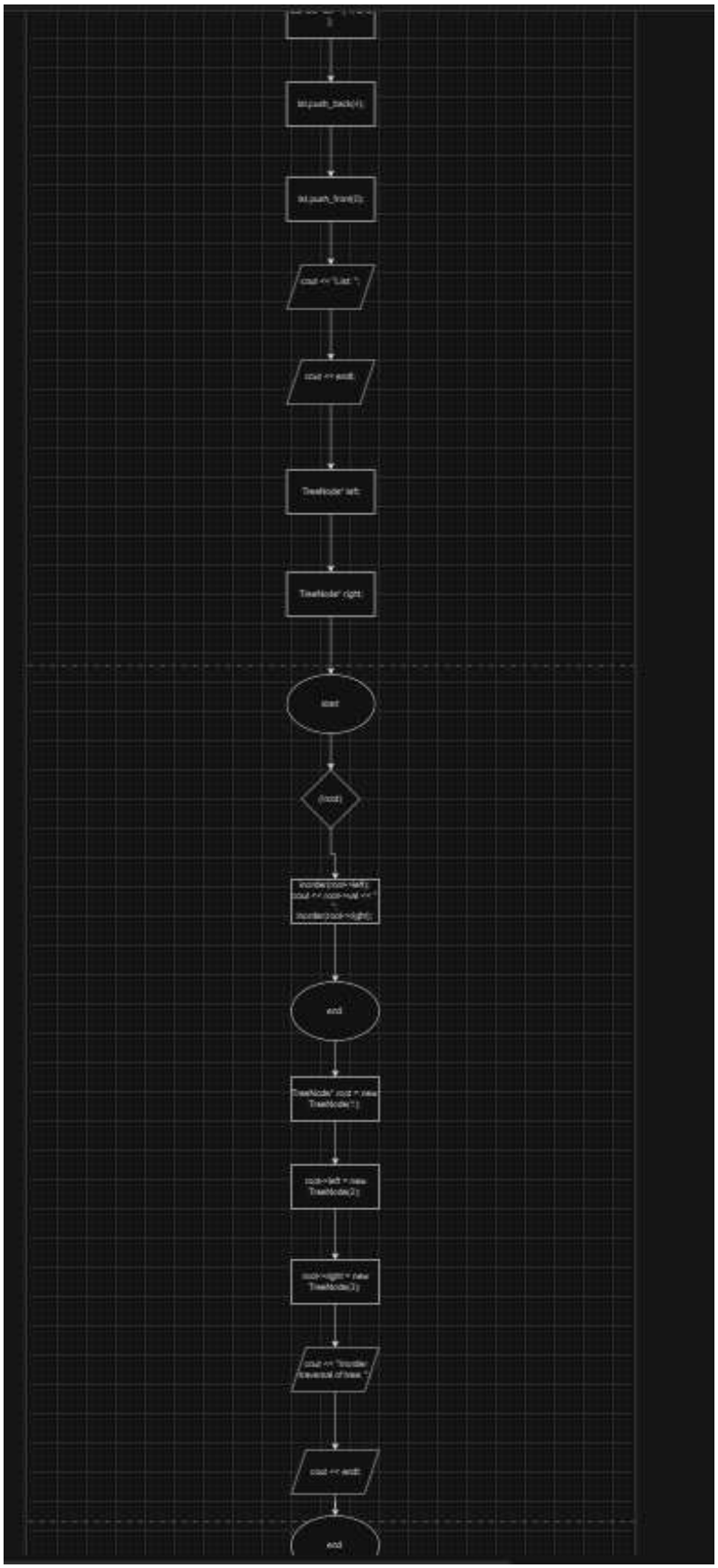
Що опрацьовано:
Ознайомлено з алгоритмами маніпуляцій даними у динамічних структурах.
Джерела інформації:

- Практичні заняття
- Використання штучного інтелекту

Виконання роботи:

Epic 6 Task 2 - Requirements management (understand tasks) and design activities (draw flow diagrams and estimate tasks 3-7




lst.push_back(4);
lst.push_front(3);
cout << "List: ";
cout << endl;
TreeNode* left;
TreeNode* right;
start
(root)
inorder(root->left);
cout << root->val << "
";
inorder(root->right);
end
TreeNode* root = new TreeNode(1);
root->left = new TreeNode(2);
root->right = new TreeNode(3);
cout << "Inorder traversal of tree: ";
cout << endl;
end

Epic 6 Task 3 - Lab# programming: VNS Lab 10

```cpp
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

struct Node {
    string key;
    Node* prev;
    Node* next;
};

class List {
private:
    Node* head;
    Node* tail;

public:
    List() : head(nullptr), tail(nullptr) {}

    void add(const string& key) {
        Node* newNode = new Node{ key, nullptr, nullptr };
        if (!head) {
            head = tail = newNode;
        }
        else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
    }

    void addAfter(const string& afterKey, const string& newKey) {
        Node* current = head;
        while (current) {
            if (current->key == afterKey) {
                Node* newNode = new Node{ newKey, current, current->next };
                if (current->next) {
                    current->next->prev = newNode;
                }
                current->next = newNode;
                if (current == tail) {
                    tail = newNode;
                }
                return;
            }
            current = current->next;
        }
    }
```

```
    void remove(const string& key) {
        Node* current = head;
        while (current) {
            if (current->key == key) {
                if (current->prev) {
                    current->prev->next = current->next;
                }
                else {
                    head = current->next;
                }
                if (current->next) {
                    current->next->prev = current->prev;
                }
                else {
                    tail = current->prev;
                }
                delete current;
                return;
            }
            current = current->next;
        }
    }

    void print() {
        Node* current = head;
        while (current) {
            cout << current->key << " ";
            current = current->next;
        }
        cout << endl;
    }

    void save(const string& filename) {
        ofstream file(filename);
        if (!file) return;
        Node* current = head;
        while (current) {
            file << current->key << endl;
            current = current->next;
        }
    }

    void load(const string& filename) {
        ifstream file(filename);
        if (!file) return;
        string key;
        while (getline(file, key)) {
            add(key);
        }
    }
```

```
    ~List() {
        Node* current = head;
        while (current) {
            Node* nextNode = current->next;
            delete current;
            current = nextNode;
        }
    }
};

int main() {
    List list;

    list.add("apple");
    list.add("banana");
    list.add("cherry");

    list.addAfter("banana", "orange");
    list.addAfter("cherry", "pear");

    list.print();

    list.remove("banana");

    list.print();

    list.save("list.txt");

    list.~List();

    List newList;
    newList.load("list.txt");

    newList.print();

    return 0;
}
```

23. Запису в лінійному списку містять ключове поле типу *char (рядок символів). Сформувати двонаправлений список. Знищити елемент із заданим ключем. Додати К елементів після елемента із заданим ключем.

Epic 6 Task 4 - Lab# programming: Algotester Lab 5

Lab 5v2

Обмеження: 1 сек., 256 МіБ

В пустелі існує незвичайна печера, яка є двохвимірною. Її висота це N, ширина - M.

Всередині печери є пустота, пісок та каміння. Пустота позначається буквою O , пісок S і каміння X;

Одного дня стався землетрус і весь пісок посипався вниз. Він падає на найнижчу клітинку з пустотою, але він не може пролетіти через каміння.

Ваше завдання сказати як буде виглядати печера після землетрусу.

Вхідні дані

У першому рядку 2 цілих числа N та M - висота та ширина печери

У N наступних рядках стрічка row_i яка складається з N цифер - i-й рядок матриці, яка відображає стан печери до землетрусу.

Вихідні дані

N рядків, які складаються з стрічки розміром M - стан печери після землетрусу.

Обмеження

$1 \leq N, M \leq 1000$

$|row_i| = M$

$row_i \in \{X, S, O\}$

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int n, m;
    cin >> n >> m;
    vector<string> cave(n);
    for (int i = 0; i < n; ++i) cin >> cave[i];

    for (int col = 0; col < m; ++col) {
        int empty_row = n - 1;
        for (int row = n - 1; row >= 0; --row) {
            if (cave[row][col] == 'S') {
                cave[row][col] = 'O';
                cave[empty_row--][col] = 'S';
            }
            else if (cave[row][col] == 'X') {
                empty_row = row - 1;
            }
        }
    }
```

```
    for (const auto& row : cave) cout << row << "\n";

    return 0;
}
```

Epic 6 Task 5 - Lab# programming: Algotester Lab 7-8

```
#include <iostream>
#include <vector>
using namespace std;

template<typename T>
class List
{
public:
    List();
    ~List();

    void append(T data);
    void insert_at(int index, const std::vector<T>& values);
    void remove(int index, int count);
    T get_at(int index) const;
    void set_at(int index, T value);
    int length() const;
    void display() const;

    template <typename U>
    friend ostream& operator<<(ostream& os, const List<U>& list);

private:
    struct Node {
        T data;
        Node* next;
        Node* prev;

        Node(T data = T(), Node* next = nullptr, Node* prev = nullptr)
            : data(data), next(next), prev(prev) {}
    };
    int Size;
    Node  *head;
```

```
    Node *tail;
};

template<typename T>
List<T>::List()
{
    Size = 0;
    head = nullptr;
    tail = nullptr;
}

template<typename T>
List<T>::~List()
{
    while (head != nullptr) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }
}

template <typename T>
void List<T>::append(T data)
{
    Node* temp = new Node(data);
    if (head == nullptr) {
        head = tail = temp;
    } else {
        temp->prev = tail;
        tail->next = temp;
        tail = temp;
    }
    Size++;
}

template <typename T>
void List<T>::insert_at(int index, const std::vector<T>& values)
{
    if (index < 0 || index > Size) return;

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

    for (int i = 0; i < index; i++) {
        prev = current;
        current = current->next;
    }

    for (const T& value : values) {
        Node* newNode = new Node(value);
        if (prev) {
```

```
            prev->next = newNode;
            newNode->prev = prev;
        }
        else {
            head = newNode;
        }
        newNode->next = current;
        if (current) {
            current->prev = newNode;
        }
        else {
            tail = newNode;
        }
        prev = newNode;
        Size++;
    }
}

template <typename T>
void List<T>::remove(int index, int count)
{
    if (index < 0 || index >= Size || count <= 0) return;

    Node* current = head;

    for (int i = 0; i < index; i++) {
        current = current->next;
    }

    for (int i = 0; i < count; i++) {
        Node* toDelete = current;
        current = current->next;

        if (toDelete->prev != nullptr) {
            toDelete->prev->next = toDelete->next;
        }
        else {
            head = toDelete->next;
        }

        if (toDelete->next != nullptr) {
            toDelete->next->prev = toDelete->prev;
        }
        else {
            tail = toDelete->prev;
        }

        delete toDelete;
        Size--;
    }
}
```

```cpp
template <typename T>
T List<T>::get_at(int index) const {
    Node* current = head;
    for (int i = 0; i < index; i++) {
        current = current->next;
    }
    return current->data;
}

template <typename T>
void List<T>::set_at(int index, T value) {
    if (index < 0 || index >= Size) throw out_of_range("Index out of range");

    Node* current = head;
    for (int i = 0; i < index; i++) {
        current = current->next;
    }
    current->data = value;
}

template <typename T>
int List<T>::length() const {
    return Size;
}

template <typename T>
void List<T>::display() const {
    Node* current = head;
    while (current != nullptr) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << endl;
}

template <typename T>
ostream& operator<<(ostream& os, const List<T>& list) {
    typename List<T>::Node* current = list.head;
    while (current != nullptr) {
        os << current->data << " ";
        current = current->next;
    }
    return os;
}

int main() {
    List<int> list;

    int Q;
    cin >> Q;
```

```
    for (int i = 0; i < Q; ++i) {
        string command;
        cin >> command;

        if (command == "insert") {
            int index, N;
            cin >> index >> N;
            vector<int> elements(N);
            for (int j = 0; j < N; ++j) cin >> elements[j];
            list.insert_at(index, elements);
        } else if (command == "erase") {
            int index, n;
            cin >> index >> n;
            list.remove(index, n);
        } else if (command == "size") {
            cout << list.length() << endl;
        } else if (command == "get") {
            int index;
            cin >> index;
            cout << list.get_at(index) << endl;
        } else if (command == "set") {
            int index, value;
            cin >> index >> value;
            list.set_at(index, value);
        } else if (command == "print") {
            cout << list << endl;
        }
    }

    return 0;
}
```

Epic 6 Task 6 - Practice# programming: Practice Task

```
#include <iostream>
using namespace std;

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

Node* reverse(Node* head) {
    Node* prev = nullptr;
    Node* current = head;
```

```
    Node* next = nullptr;

    while (current != nullptr) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }

    return prev;
}

void printList(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << endl;
}

int main() {
    Node* head = new Node(1);
    head->next = new Node(2);
    head->next->next = new Node(3);
    head->next->next->next = new Node(4);
    head->next->next->next->next = new Node(5);

    cout << "Original list: ";
    printList(head);

    head = reverse(head);

    cout << "Reversed list: ";
    printList(head);

    return 0;
}

#include <iostream>
using namespace std;

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

bool compare(Node* h1, Node* h2) {
    while (h1 != nullptr && h2 != nullptr) {
        if (h1->data != h2->data) {
```

```
            return false;
        }
        h1 = h1->next;
        h2 = h2->next;
    }
    return h1 == nullptr && h2 == nullptr;
}

void printList(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << endl;
}

int main() {
    Node* head1 = new Node(1);
    head1->next = new Node(2);
    head1->next->next = new Node(3);

    Node* head2 = new Node(1);
    head2->next = new Node(2);
    head2->next->next = new Node(3);

    Node* head3 = new Node(1);
    head3->next = new Node(4);
    head3->next->next = new Node(3);

    cout << "List 1: ";
    printList(head1);

    cout << "List 2: ";
    printList(head2);

    cout << "List 3: ";
    printList(head3);

    cout << "List 1 and List 2 are equal: " << (compare(head1, head2) ? "Yes" :
"No") << endl;
    cout << "List 1 and List 3 are equal: " << (compare(head1, head3) ? "Yes" :
"No") << endl;

    return 0;
}

#include <iostream>
using namespace std;

struct Node {
```

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

Node* add(Node* n1, Node* n2) {
    Node* result = nullptr;
    Node* tail = nullptr;
    int carry = 0;

    while (n1 != nullptr || n2 != nullptr || carry) {
        int sum = carry;

        if (n1 != nullptr) {
            sum += n1->data;
            n1 = n1->next;
        }

        if (n2 != nullptr) {
            sum += n2->data;
            n2 = n2->next;
        }

        carry = sum / 10;
        Node* newNode = new Node(sum % 10);

        if (result == nullptr) {
            result = newNode;
            tail = result;
        }
        else {
            tail->next = newNode;
            tail = tail->next;
        }
    }

    return result;
}

void printList(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        cout << current->data;
        current = current->next;
    }
    cout << endl;
}

int main() {
    Node* n1 = new Node(9);
    n1->next = new Node(7);
```

```
    n1->next->next = new Node(3);  // Number: 379

    Node* n2 = new Node(1);
    n2->next = new Node(5);
    n2->next->next = new Node(2);  // Number: 251

    cout << "Number 1: ";
    printList(n1);

    cout << "Number 2: ";
    printList(n2);

    Node* sum = add(n1, n2);

    cout << "Sum: ";
    printList(sum);

    return 0;
}

#include <iostream>
using namespace std;

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

TreeNode* create_mirror_flip(TreeNode* root) {
    if (root == nullptr) {
        return nullptr;
    }

    TreeNode* newRoot = new TreeNode(root->data);
    newRoot->left = create_mirror_flip(root->right);
    newRoot->right = create_mirror_flip(root->left);

    return newRoot;
}

void printTree(TreeNode* root) {
    if (root == nullptr) return;
    cout << root->data << " ";
    printTree(root->left);
    printTree(root->right);
}

int main() {
    TreeNode* root = new TreeNode(1);
```

```
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->right->left = new TreeNode(6);
    root->right->right = new TreeNode(7);

    cout << "Original Tree: ";
    printTree(root);
    cout << endl;

    TreeNode* mirroredRoot = create_mirror_flip(root);

    cout << "Mirrored Tree: ";
    printTree(mirroredRoot);
    cout << endl;

    return 0;
}

#include <iostream>
using namespace std;

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

int sumSubtree(TreeNode* root) {
    if (root == nullptr) {
        return 0;
    }

    int leftSum = sumSubtree(root->left);
    int rightSum = sumSubtree(root->right);

    if (root->left != nullptr || root->right != nullptr) {
        root->data = leftSum + rightSum;
    }
    return root->data + leftSum + rightSum;
}

void tree_sum(TreeNode* root) {
    sumSubtree(root);
}

void printTree(TreeNode* root) {
    if (root == nullptr) return;
    cout << root->data << " ";
```

```
    printTree(root->left);
    printTree(root->right);
}

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->right->left = new TreeNode(6);
    root->right->right = new TreeNode(7);

    cout << "Original Tree: ";
    printTree(root);
    cout << endl;

    tree_sum(root);

    cout << "Tree with sums: ";
    printTree(root);
    cout << endl;

    return 0;
}
```

Epic 6 Task 7 - Practice# programming: Self Practice Task

```
#include <iostream>
#include <queue>
#include <stack>
#include <list>

using namespace std;


void stack_example() {
```

```
    stack<int> s;
    s.push(1);
    s.push(2);
    s.push(3);
    cout << "Stack top: " << s.top() << endl;
    s.pop();
    cout << "Stack top after pop: " << s.top() << endl;
}


void queue_example() {
    queue<int> q;
    q.push(1);
    q.push(2);
    q.push(3);
    cout << "Queue front: " << q.front() << endl;
    q.pop();
    cout << "Queue front after pop: " << q.front() << endl;
}


void list_example() {
    list<int> lst = { 1, 2, 3 };
    lst.push_back(4);
    lst.push_front(0);
    cout << "List: ";
    for (int val : lst) cout << val << " ";
    cout << endl;
}


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

void inorder(TreeNode* root) {
    if (!root) return;
    inorder(root->left);
    cout << root->val << " ";
    inorder(root->right);
}

void tree_example() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    cout << "Inorder traversal of tree: ";
    inorder(root);
```

```
    cout << endl;
}

int main() {

    stack_example();


    queue_example();


    list_example();


    tree_example();

    return 0;
}
```

Робота у команді:

Ми зібрались з командою обсудили деталі 6 епіку і домовились зустрітись якщо в когось виникнуть питання.

Висновок:

У ході роботи було досліджено принципи роботи з основними динамічними структурами даних: стеком, чергою, зв’язними списками та деревами. Розглянуто їхню реалізацію, основні операції (вставка, видалення, пошук) та алгоритми обробки. Проаналізовано вплив управління пам’яттю на ефективність цих структур. Здобуті знання підтвердили, що динамічні структури даних забезпечують гнучкість у використанні пам’яті та ефективне розв’язання завдань, що потребують швидкого доступу та модифікації даних.