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:

ВНС Лабораторної Роботи № 10
Алготестер Лабораторної Роботи № 5
Алготестер Лабораторної Роботи № 7-8
Практичних Робіт до блоку № 6

Виконала:
Студентка групи ШІ-11
Андрусишин Соломія Володимирівна

Тема роботи: Вивчення основ динамічних структур даних у C++: стек, черга, зв'язні списки та дерева, а також алгоритмів обробки цих структур, включаючи операції додавання, видалення елементів та пошук. Розгляд основних принципів виділення пам'яті, переповнення стеку, обробки подій через чергу, а також обробки дерев різних типів (бінарних, AVL, червоно-чорних дерев) за допомогою ітеративних та рекурсивних алгоритмів.

Мета роботи: **:** Ознайомитися з основними динамічними структурами даних у C++, зокрема стеком, чергою, зв'язними списками та деревами, вивчити їх властивості та операції (push, pop, enqueue, dequeue тощо). Опанувати основні алгоритми пошуку, сортування, вставки та видалення елементів у цих структурах. Зрозуміти принципи виділення пам'яті для динамічних структур та їх обробку, включаючи випадки переповнення та особливості обробки складних дерев.

Теоретичні відомості:

1. ***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
2. ***Стек:***
   - Визначення та властивості стеку
   - Операції push, pop, top: реалізація та використання
   - Приклади використання стеку: обернений польський запис, перевірка балансу дужок
   - Переповнення стеку
3. ***Черга:***
   - Визначення та властивості черги
   - Операції enqueue, dequeue, front: реалізація та застосування
   - Приклади використання черги: обробка подій, алгоритми планування
   - Розширення функціоналу черги: пріоритетні черги
4. ***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

5. ***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
6. ***Алгоритми Обробки Динамічних Структур:***
   - Основи алгоритмічних патернів: ітеративні, рекурсивні
   - Алгоритми пошуку, сортування даних, додавання та видалення елементів

Індивідуальний план опрацьовування теорії:

1.Основи Динамічних Структур Даних:

- Джерела:
- https://acode.com.ua/urok-111-stek-i-kupa/#toc-1
- https://www.youtube.com/watch?v=NyOjKd5Qruk

- Висновок:

У пам’яті комп’ютера існують дві основні області для розміщення даних:

- **Stack (стек):**
  - Використовується для зберігання локальних змінних і викликів функцій.
  - Пам'ять виділяється та звільняється автоматично.
  - Обмежена за розміром, але швидша у доступі.
- **Heap (купа):**
  - Використовується для виділення динамічної пам’яті.
  - Виділення та звільнення пам’яті контролюється програмістом (наприклад, через `new`/`delete`)
  - Гнучкий і підходить для динамічних структур даних , а також для роботи з складними структурами.

Особливості динамічного масиву:

- Збільшення або зменшення розміру під час виконання програми.
- Ефективна робота зі змінними даними.
- Просте додавання та видалення елементів.

- ***2.Стек:***
  - Джерела:
  - https://dystosvita.org.ua/mod/page/view.php?id=888
  - https://disted.edu.vn.ua/courses/learn/13472
- Висновок:
  Стек — це динамічна структура даних, яка працює за принципом **LIFO** (Last In, First Out): останній елемент, який додається до стеку, буде першим, що видаляється.

Операції зі стеком

1. **Push:** Додає елемент у вершину стеку.
2. **Pop:** Видаляє елемент із вершини стеку.
3. **Top (або Peek):** Повертає елемент у вершині стеку без видалення.

Приклади використання стеку:

Обернений польський запис

3. Черга:

- Джерела:
- https://dystosvita.org.ua/mod/page/view.php?id=889

- Висновок:
  Черга (Queue) — працює за принципом **FIFO (First In, First Out)**: елементи додаються в кінець черги, а видаляються з початку. Це означає, що перший елемент, який був доданий, буде першим видаленим.
  -enqueue: додавання елемента в кінець черги.
  -dequeue: видалення елемента з початку черги.
  -front: перегляд елемента на початку черги.

4. Зв'язні Списки:

- Джерела:
- https://prometheus.org.ua/cs50/sections/section6.html

- Висновок:

Зв'язний список — це динамічна структура даних, яка складається з вузлів. Кожен вузол містить дані та вказівник на інший вузол.
Види списків:

- **Однозв'язний список:**
  Кожен вузол містить дані та вказівник на наступний вузол.

- **Двозв'язний список:**
  Кожен вузол містить дані, вказівник на наступний вузол і вказівник на попередній вузол.

5. Дерева:

- Джерела:
- https://purecodecpp.com/uk/archives/2483
- https://javarush.com/ua/groups/posts/uk.4165.chervono-chorne-derevo-vlastivost-principi-organzac-mekhanzmi-vstavki

Висновок:
Дерево — це нелінійна структура даних, яка складається з вузлів, організованих у вигляді ієрархії.

- **Корінь дерева** (root): вузол, який є початком дерева.
- **Батьківські та дочірні вузли**: кожен вузол може мати дочірні вузли. Вузол, який має дочірній вузол, називається батьківським.
- **Листя** (leaf): вузли без дочірніх вузлів.

Типи дерев:

1. **Бінарне дерево:** Кожен вузол має максимум два дочірніх вузли.
2. **Повне бінарне дерево:** Усі рівні, крім останнього, заповнені, а вузли останнього рівня розташовані зліва.
3. **Дерево пошуку (Binary Search Tree, BST):** Бінарне дерево, де для кожного вузла:
   - Значення всіх вузлів лівого піддерева менше, ніж значення вузла.
   - Значення всіх вузлів правого піддерева більше, ніж значення вузла.
4. **AVL-дерево:** Бінарне дерево пошуку, збалансоване за висотою.
5. **Червоно-чорне дерево:** Самобалансуюче бінарне дерево пошуку.

Виконання роботи:

Завдання №1
(VNS Lab10 Variant 6) -
1. Написати функцію для створення списку. Функція може створювати порожній список, а потім додавати в нього елементи.
2. Написати функцію для друку списку. Функція повинна передбачати вивід повідомлення, якщо список порожній.

3. Написати функції для знищення й додавання елементів списку у відповідності зі своїм варіантом.
4. Виконати зміни в списку й друк списку після кожної зміни.
5. Написати функцію для запису списку у файл.
6. Написати функцію для знищення списку.
7. Записати список у файл, знищити його й виконати друк (при друці повинне бути видане повідомлення "Список порожній").
8. Написати функцію для відновлення списку з файлу.
9. Відновити список і роздрукувати його.
10.Знищити список.

-Записи в лінійному списку містять ключове поле типу int. Сформувати двонаправлений список. Знищити з нього елемент із заданим номером, додати елемент у початок списку.

Завдання №2
(Algotester Lab5 V.2) -

В пустелі існує незвичайна печера, яка є двохвимірною. Її висота це NN, ширина - MM.
Всередині печери є пустота, пісок та каміння. Пустота позначається буквою OO , пісок SS і каміння XX;
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.

Input

У першому рядку 2 цілих числа NN та MM - висота та ширина печери
У NN наступних рядках стрічка rowirowi яка складається з NN цифер - і-й рядок матриці, яка відображає стан печери до землетрусу.

Output

NN рядків, які складаються з стрічки розміром MM - стан печери після землетрусу.

Constraints

$1 \le N,M \le 1000$
$|row_i| = M$
$row_i \in \{X,S,O\}$

Завдання №3
(Algotester Lab5 V.2) –

Lab 78v1

Limits: 2 sec., 256 MiB

Ваше завдання - власноруч реалізувати структуру даних "Двозв’язний список".
Ви отримаєте QQ запитів, кожен запит буде починатися зі слова-ідентифікатора, після якого йдуть його аргументи.

Вам будуть поступати запити такого типу:

- **Вставка**:
  Ідентифікатор - insertinsert
  Ви отримуєте ціле число indexindex елемента, на місце якого робити вставку.
  Після цього в наступному рядку рядку написане число NN - розмір списку, який треба вставити.
  У третьому рядку NN цілих чисел - список, який треба вставити на позицію indexindex.
- **Видалення**:
  Ідентифікатор - eraseerase
  Ви отримуєте 2 цілих числа - indexindex, індекс елемента, з якого почати видалення та nn - кількість елементів, яку треба видалити.
- **Визначення розміру**:
  Ідентифікатор - sizesize
  Ви не отримуєте аргументів.
  Ви виводите кількість елементів у списку.
- **Отримання значення** ii-го елементу
  Ідентифікатор - getget
  Ви отримуєте ціле число - indexindex, індекс елемента.
  Ви виводите значення елемента за індексом.
- **Модифікація значення** ii-го елементу
  Ідентифікатор - setset
  Ви отримуєте 2 цілих числа - індекс елемента, який треба змінити, та його нове значення.
- **Вивід списку на екран**
  Ідентифікатор - printprint
  Ви не отримуєте аргументів.
  Ви виводите усі елементи списку через пробіл.
  Реалізувати використовуючи перегрузку оператора <<

Notes

Гарантується, що усі дані коректні. Виходу за межі списку або розмір, більший ніж розмір списку недопустимі.
Індекси починаються з нуля.

Для того щоб отримати 50%50% **балів за лабораторну достатньо написати свою структуру.**
Для отримання 100%100% **балів ця структура має бути написана як шаблон класу, у якості параметру використати** intint**.**
Використовувати STL заборонено.

Завдання №4
(Class Practice Work)

Задача №1 - Реверс списку (Reverse list)

Реалізувати метод реверсу списку: Node* reverse(Node *head);
Умови задачі:

- використовувати цілочисельні значення в списку;
- реалізувати метод реверсу;
- реалізувати допоміжний метод виведення вхідного і обернутого списків;

Мета задачі

Розуміння структур даних: Реалізація методу реверсу для зв'язаних списків є чудовим способом для поглиблення розуміння зв'язаних списків як фундаментальної структури даних. Він заохочує практичний підхід до вивчення того, як структуруються пов'язані списки та як ними маніпулювати.

Розвиток алгоритмічне мислення: Це завдання розвиває алгоритмічне мислення. Перевертання пов'язаного списку вимагає логічного підходу до маніпулювання покажчиками, що є ключовим навиком у інформатиці.

Засвоїти механізми маніпуляції з покажчиками: пов'язані списки значною мірою залежать від покажчиків. Це завдання покращить навички маніпулювання вказівниками, що є ключовим аспектом у таких мовах, як C++.

Розвинути навички розв'язувати задачі: перевернути пов'язаний список непросто й вимагає творчого й логічного мислення, таким чином покращуючи свої навички розв'язування поставлених задач.

Пояснення прикладу

Спочатку ми визначаємо просту структуру ***Node*** для нашого пов'язаного списку.
Потім функція ***reverse*** ітеративно змінює список, маніпулюючи наступними покажчиками кожного вузла.
printList — допоміжна функція для відображення списку.
Основна функція створює зразок списку, демонструє реверсування та друкує вихідний і обернений списки.

Задача №2 - Порівняння списків

bool compare(Node *h1, Node *h2);
Умови задачі:

- використовувати цілочисельні значення в списку;
- реалізувати функцію, яка ітеративно проходиться по обох списках і порівнює дані в кожному вузлі;
- якщо виявлено невідповідність даних або якщо довжина списків різна (один список закінчується раніше іншого), функція повертає ***false***.

Мета задачі

Розуміння рівності в структурах даних: це завдання допомагає зрозуміти, як визначається рівність у складних структурах даних, таких як зв'язані списки. На відміну від примітивних типів даних, рівність пов'язаного списку передбачає порівняння кожного елемента та їх порядку.

Поглиблення розуміння зв'язаних списків: Порівнюючи зв'язані списки, дозволяють покращити своє розуміння обходу, фундаментальної операції в обробці зв'язаних списків.

Розуміння ефективність алгоритму: це завдання також вводить поняття ефективності алгоритму. Студенти вчаться ефективно порівнювати елементи, що є навичкою, важливою для оптимізації та зменшення складності обчислень.

Розвинути базові навики роботи з реальними програми: функції порівняння мають вирішальне значення в багатьох реальних програмах, таких як виявлення змін у даних, синхронізація структур даних або навіть у таких алгоритмах, як сортування та пошук.

Розвинути навик вирішення проблем і увага до деталей: це завдання заохочує скрупульозний підхід до програмування, оскільки навіть найменша неуважність може призвести до неправильних результатів порівняння. Це покращує навички вирішення проблем і увагу до деталей.

Пояснення прикладу

- Для пов'язаного списку визначено структуру ***Node***.
- Функція ***compare*** ітеративно проходить обидва списки одночасно, порівнюючи дані в кожному вузлі.
- Якщо виявлено невідповідність даних або якщо довжина списків різна (один список закінчується раніше іншого), функція повертає ***false***.
- Основна функція ***main*** створює два списки та демонструє порівняння.

Задача №3 – Додавання великих чисел

Node* add(Node *n1, Node *n2);
Умови задачі:

- використовувати цифри від 0 до 9 для значень у списку;
- реалізувати функцію, яка обчислює суму двох чисел, які збережено в списку; молодший розряд числа записано в голові списка (напр. 379 $\Longrightarrow$ 9→7→3);
- функція повертає новий список, передані в функцію списки не модифікуються.

Мета задачі

Розуміння операцій зі структурами даних: це завдання унаочнює практичне використання списка для обчислювальних потреб. Арифметичні операції з великими числами це окремий клас задач, для якого використання списків допомагає обійти обмеження у представленні цілого числа сучасними комп'ютерами.

Поглиблення розуміння зв'язаних списків: Застосовування зв'язаних списків для арифметичних операції з великими числами дозволяє покращити розуміння операцій з обробки зв'язаних списків.

Розуміння ефективність алгоритму: це завдання дозволяє порівняти швидкість алгоритму додавання з використанням списків зі швидкістю вбудованих арифметичних операцій. Студенти вчаться розрізняти позитивні та негативні ефекти при виборі структур даних для реалізації практичних програм.

Розвинути базові навики роботи з реальними програми: арифметичні операції з великими числами використовуються у криптографії, теорії чисел, астрономії, та ін.

Розвинути навик вирішення проблем і увага до деталей: завдання покращує розуміння обмежень у представленні цілого числа сучасними комп'ютерами та пропонує спосіб його вирішення.

Бінарні дерева

Задача №4 - Віддзеркалення дерева

TreeNode *create_mirror_flip(TreeNode *root);
Умови задачі:

- використовувати цілі числа для значень у вузлах дерева
- реалізувати функцію, що проходить по всіх вузлах дерева і міняє місцями праву і ліву вітки дерева
- функція повертає нове дерево, передане в функцію дерево не модифікується

Мета задачі

Розуміння структур даних: Реалізація методу віддзеркалення бінарного дерева покращує розуміння структури бінарного дерева, виділення пам'яті для вузлів та зв'язування їх у єдине ціле. Це один з багатьох методів роботи з бінарними деревами.

Розвиток алгоритмічне мислення: Це завдання розвиває алгоритмічне мислення. Прохід всіх вузлів дерева продемонструє розгортання рекурсивного виклику.

Задача №5 - Записати кожному батьківському вузлу суму підвузлів

void tree_sum(TreeNode *root);
Умови задачі:

- використовувати цілочисельні значення у вузлах дерева;
- реалізувати функцію, яка ітеративно проходить по бінарному дереві і записує у батьківський вузол суму значень підвузлів
- вузол-листок не змінює значення
- значення змінюються від листків до кореня дерева

Мета задачі

Розуміння структур даних: Реалізація методу підрахунку сум підвузлів бінарного дерева покращує розуміння структури бінарного дерева. Це один з багатьох методів роботи з бінарними деревами.

Розвиток алгоритмічне мислення: Це завдання розвиває алгоритмічне мислення. Прохід всіх вузлів дерева демонструє розгортання рекурсивного виклику.

Завдання №5
(Self Practice Work)

Реалізуй клас Library для управління динамічною колекцією книг із функціями додавання, видалення, пошуку, оновлення, виведення списку книг, зміни розміру масиву та підтримкою сортування і фільтрації за автором чи роком видання.

Завдання №1
(VNS Lab10 Variant 6)

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

using namespace std;

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

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

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

    void add(int value) {
        Node* newNode = new Node{value, nullptr, nullptr};
        if (head == nullptr) {
            head = newNode;
            tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
    }

    void print_list() {
        if (head == nullptr) {
            cout << "Список порожній" << endl;
            return;
        }
        Node* current = head;
        while (current != nullptr) {
            cout << current->data << " ";
            current = current->next;
        }
        cout << endl;
    }
```

```
void delete_by_index(int index) {
    if (head == nullptr) {
        cout << "Список порожній, немає чого видаляти" << endl;
        return;
    }

    Node* current = head;
    int currentIndex = 0;

    while (current != nullptr && currentIndex != index) {
        current = current->next;
        currentIndex++;
    }

    if (current == nullptr) {
        cout << "Елемент з таким номером не знайдено" << endl;
        return;
    }

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

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

    delete current;
    cout << "Елемент видалено" << endl;
}

void push_front(int value) {
    Node* newNode = new Node{value, head, nullptr};
    if (head != nullptr) {
        head->prev = newNode;
    } else {
        tail = newNode;
    }
```

```
87          head = newNode;
88      }
89
90      void save_to_file(const string& filename) {
91          ofstream file(filename);
92          if (!file) {
93              cout << "Помилка відкриття файлу" << endl;
94              return;
95          }
96
97          Node* current = head;
98          while (current != nullptr) {
99              file << current->data << " ";
100             current = current->next;
101         }
102         file.close();
103         cout << "Список збережено у файл" << endl;
104     }
105
106     void clear_list() {
107         while (head != nullptr) {
108             Node* temp = head;
109             head = head->next;
110             delete temp;
111         }
112         tail = nullptr;
113         cout << "Список очищено" << endl;
114     }
115
116     void list_restore(const string& filename) {
117         ifstream file(filename);
118         if (!file) {
119             cout << "Помилка відкриття файлу" << endl;
120             return;
121         }
122
123         clear_list();
124         int value;
125         while (file >> value) {
126             add(value);
127         }
```

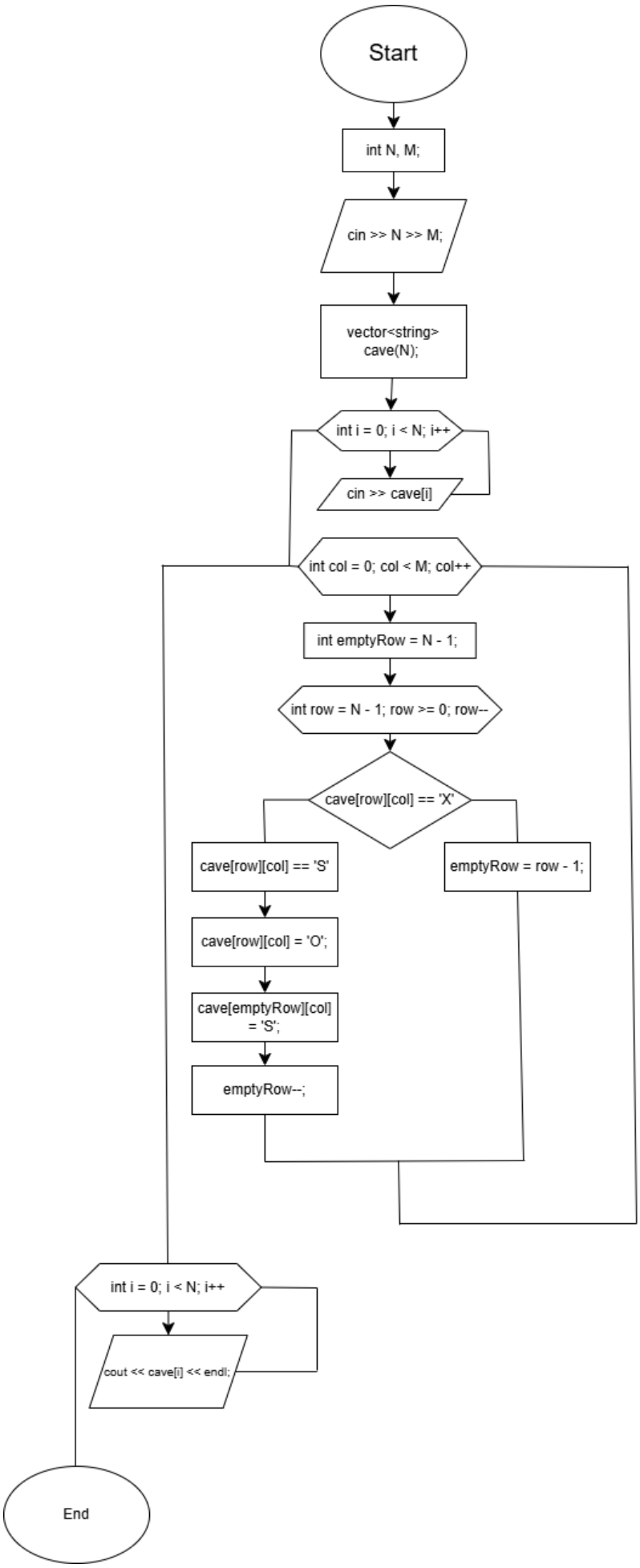
```
        }
        file.close();
        cout << "Список відновлено з файлу" << endl;
    }

    ~DoubleLinkedList() {
        clear_list();
    }
};

int main() {
    DoubleLinkedList list;

    for (int i = 1; i <= 5; i++) {
        list.add(i);
    }

    cout << "Список після додавання елементів:" << endl;
    list.print_list();

    list.push_front(0);
    cout << "Список після додавання елемента в початок:" << endl;
    list.print_list();

    list.delete_by_index(2);
    cout << "Список після видалення другого елемента:" << endl;
    list.print_list();

    list.save_to_file("list.txt");

    list.clear_list();
    cout << "Список після очищення:" << endl;
    list.print_list();

    list.list_restore("list.txt");
    cout << "Список після відновлення з файлу:" << endl;
    list.print_list();

    return 0;
}
```

```
Список після додавання елементів:
1 2 3 4 5
Список після додавання елемента в початок:
0 1 2 3 4 5
Елемент видалено
Список після видалення другого елемента:
0 1 3 4 5
Список збережено у файл
Список очищено
Список після очищення:
Список порожній
Список очищено
Список відновлено з файлу
Список після відновлення з файлу:
0 1 3 4 5
Список очищено
```

Завдання №2
(Algotester Lab5 V.2)

```cpp
#include <iostream>
#include <vector>
#include <string>
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 emptyRow = N - 1;
        for (int row = N - 1; row >= 0; row--) {
            if (cave[row][col] == 'X') {
                emptyRow = row - 1;
            } else if (cave[row][col] == 'S') {
                cave[row][col] = '0';
                cave[emptyRow][col] = 'S';
                emptyRow--;
            }
        }
    }

    for (int i = 0; i < N; i++) {
        cout << cave[i] << endl;
    }

    return 0;
}
```

```
5 5
SS0SS
00000
S00XX
0000X
00S00
00000
000SS
000XX
S000X
SSS00
```


Start
int N, M;
cin >> N >> M;
vector<string> cave(N);
int i = 0; i < N; i++
cin >> cave[i]
int col = 0; col < M; col++
int emptyRow = N - 1;
int row = N - 1; row >= 0; row--
cave[row][col] == 'X'
cave[row][col] == 'S'
emptyRow = row - 1;
cave[row][col] = 'O';
cave[emptyRow][col] = 'S';
emptyRow--;
int i = 0; i < N; i++
cout << cave[i] << endl;
End

Завдання №3
(VNS Lab7 Task1)

Варіант з написанням своєї структури:

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

using namespace std;

enum Operation
{
    insert_arr,
    erase_arr,
    size_arr,
    capacity_arr,
    get_arr,
    set_arr,
    print_arr,
    invalid
};

Operation get_operation(const string &command)
{
    if (command == "insert") return insert_arr;
    if (command == "erase") return erase_arr;
    if (command == "size") return size_arr;
    if (command == "capacity") return capacity_arr;
    if (command == "get") return get_arr;
    if (command == "set") return set_arr;
    if (command == "print") return print_arr;
    return invalid;
}

class dynamic_array
{
private:
    int *data;
    int size_;
    int capacity_;

    void resize(int new_capacity)
    {
        int *new_data = new int[new_capacity];
        for (int i = 0; i < size_; ++i)
        {
            new_data[i] = data[i];
        }
        delete[] data;
```

```
        data = new_data;
        capacity_ = new_capacity;
    }

public:
    dynamic_array() : size_(0), capacity_(1)
    {
        data = new T[capacity_];
    }

    ~dynamic_array()
    {
        delete[] data;
    }

    void insert(int index, int N, T *arr)
    {
        while (size_ + N > capacity_)
        {
            resize(capacity_ * 2);
        }

        for (int i = size_ - 1; i >= index; --i)
        {
            data[i + N] = data[i];
        }

        for (int i = 0; i < N; ++i)
        {
            data[index + i] = arr[i];
        }

        size_ += N;
    }

    void erase(int index, int n)
    {
        for (int i = index; i < size_ - n; ++i)
        {
            data[i] = data[i + n];
        }
        size_ -= n;
    }
```

```
88      int size() const { return size_; }
89      int capacity() const { return capacity_; }
90
91      T &operator[](int index) { return data[index]; }
92
93      friend ostream &operator<<(ostream &os, const dynamic_array &arr)
94      {
95          for (int i = 0; i < arr.size_; ++i)
96          {
97              os << arr.data[i] << ' ';
98          }
99          return os;
100     }
101 };
102
103 int main()
104 {
105     int Q;
106     cin >> Q;
107
108     dynamic_array<int> dynamic_arr;
109
110     while (Q--)
111     {
112         string option;
113         cin >> option;
114
115         switch (get_operation(option))
116         {
117         case insert_arr:
118         {
119             int index, N;
120             cin >> index >> N;
121             int *arr = new int[N];
122             for (int i = 0; i < N; ++i)
123             {
124                 cin >> arr[i];
125             }
126             dynamic_arr.insert(index, N, arr);
127             delete[] arr;
128             break;
129         }
130         case erase_arr:
131         {
```

```
131         {
132             int index, n;
133             cin >> index >> n;
134             dynamic_arr.erase(index, n);
135             break;
136         }
137         case size_arr:
138             cout << dynamic_arr.size() << endl;
139             break;
140         case capacity_arr:
141             cout << dynamic_arr.capacity() << endl;
142             break;
143         case get_arr:
144         {
145             int index;
146             cin >> index;
147             cout << dynamic_arr[index] << endl;
148             break;
149         }
150         case set_arr:
151         {
152             int index, value;
153             cin >> index >> value;
154             dynamic_arr[index] = value;
155             break;
156         }
157         case print_arr:
158             cout << dynamic_arr << endl;
159             break;
160         default:
161             cerr << "Invalid operation!" << endl;
162         }
163     }
164 
165     return 0;
166 }
167 
168 
```

Структура написана як шаблон класу:

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

using namespace std;

enum Operation
{
    insert_arr,
    erase_arr,
    size_arr,
    capacity_arr,
    get_arr,
    set_arr,
    print_arr,
    invalid
};

Operation get_operation(const string &command)
{
    if (command == "insert") return insert_arr;
    if (command == "erase") return erase_arr;
    if (command == "size") return size_arr;
    if (command == "capacity") return capacity_arr;
    if (command == "get") return get_arr;
    if (command == "set") return set_arr;
    if (command == "print") return print_arr;
    return invalid;
}

template <typename T = int>
class dynamic_array
{
private:
    T *data;
    int size_;
    int capacity_;

    void resize(int new_capacity)
    {
        T *new_data = new T[new_capacity];
        copy(data, data + size_, new_data);
        delete[] data;
        data = new_data;
        capacity_ = new_capacity;
    }

public:
    dynamic_array() : size_(0), capacity_(1)
    {
        data = new T[capacity_];
    }

    ~dynamic_array()
    {
        delete[] data;
    }

    void insert(int index, int N, T *arr)
    {
        while (size_ + N > capacity_)
        {
            resize(capacity_ * 2);
        }

        for (int i = size_ - 1; i >= index; --i)
        {
```

```
            data[i + N] = data[i];
        }

        for (int i = 0; i < N; ++i)
        {
            data[index + i] = arr[i];
        }

        size_ += N;
    }

    void erase(int index, int n)
    {
        for (int i = index; i < size_ - n; ++i)
        {
            data[i] = data[i + n];
        }
        size_ -= n;
    }

    int size() const { return size_; }
    int capacity() const { return capacity_; }

    T &operator[](int index) { return data[index]; }

    friend ostream &operator<<(ostream &os, const dynamic_array &arr)
    {
        for (int i = 0; i < arr.size_; ++i)
        {
            os << arr.data[i] << ' ';
        }
        return os;
    }
};

int main()
{
    int Q;
    cin >> Q;

    dynamic_array<int> dynamic_arr;

    while (Q--)
    {
        string option;
        cin >> option;

        switch (get_operation(option))
        {
        case insert_arr:
        {
            int index, N;
            cin >> index >> N;
            int *arr = new int[N];
            for (int i = 0; i < N; ++i)
            {
                cin >> arr[i];
            }
            dynamic_arr.insert(index, N, arr);
            delete[] arr;
            break;
        }
        case erase_arr:
        {
```

```
    cin >> Q;

    dynamic_array<int> dynamic_arr;

    while (Q--)
    {
        string option;
        cin >> option;

        switch (get_operation(option))
        {
        case insert_arr:
        {
            int index, N;
            cin >> index >> N;
            int *arr = new int[N];
            for (int i = 0; i < N; ++i)
            {
                cin >> arr[i];
            }
            dynamic_arr.insert(index, N, arr);
            delete[] arr;
            break;
        }
        case erase_arr:
        {
            int index, n;
            cin >> index >> n;
            dynamic_arr.erase(index, n);
            break;
        }
        case size_arr:
            cout << dynamic_arr.size() << endl;
            break;
        case capacity_arr:
            cout << dynamic_arr.capacity() << endl;
            break;
        case get_arr:
        {
            int index;
            cin >> index;
            cout << dynamic_arr[index] << endl;
            break;
        }
        case set_arr:
        {
            int index, value;
            cin >> index >> value;
            dynamic_arr[index] = value;
            break;
        }
        case print_arr:
            cout << dynamic_arr << endl;
            break;
        default:
            cerr << "Invalid operation!" << endl;
        }
    }

    return 0;
}
```

```
5
insert
0 3
1 2 3
erase
0 2
set
0 10
size
1
print
10
```

Завдання №4
(Class Practice Work)

1. **Task 1 - Реверс списку (Reverse list)**

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

struct Node {
    int value;
    Node* nextNode;

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

Node* reverseList(Node* head) {
    Node* previous = nullptr;
    Node* current = head;
    Node* following = nullptr;

    while (current != nullptr) {
        following = current->nextNode;
        current->nextNode = previous;
        previous = current;
        current = following;
    }

    return previous;
}

void displayList(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        cout << current->value << " ";
        current = current->nextNode;
    }
    cout << endl;
}

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

    cout << "Початковий список: ";
    displayList(head);

    head = reverseList(head);

    cout << "Перевернутий список: ";
    displayList(head);

    return 0;
}
```

```
Початковий список: 1 2 3 4
Перевернутий список: 4 3 2 1
```

Task 2 - Порівняння списків

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

struct Node {
    int data;
    Node* next;

    Node(int value) {
        data = value;
        next = nullptr;
    }
};

bool compareLists(Node* head1, Node* head2) {
    while (head1 != nullptr && head2 != nullptr) {
        if (head1->data != head2->data) {
            return false;
        }
        head1 = head1->next;
        head2 = head2->next;
    }

    if (head1 == nullptr && head2 == nullptr) {
        return true;
    } else {
        return false;
    }
}

void addToStart(Node*& head, int value) {
    Node* newNode = new Node(value);
    newNode->next = head;
    head = newNode;
}

void addToEnd(Node*& head, int value) {
    Node* newNode = new Node(value);
    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

int main() {
    Node* list1 = nullptr;
    addToEnd(list1, 1);
    addToEnd(list1, 2);
    addToEnd(list1, 3);

    Node* list2 = nullptr;
    addToStart(list2, 3);
    addToStart(list2, 2);
    addToStart(list2, 1);

    if (compareLists(list1, list2)) {
        cout << "Списки однакові" << endl;
    } else {
        cout << "Списки різні" << endl;
    }

    return 0;
}
```

```
Списки однакові
PS C:\Users\Solomia\Desktop\epic_6>
```

Task 3 - Додавання великих чисел

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

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

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

    while (n1 != nullptr || n2 != nullptr || carry > 0) {
        int sum = carry;
        if (n1 != nullptr) {
            sum += n1->data;
            n1 = n1->next;
        }
        if (n2 != nullptr) {
            sum += n2->data;
            n2 = n2->next;
        }

        carry = sum / 10;
        int data = sum % 10;

        Node* newNode = new Node(data);

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

    return result;
}

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

int main() {

    Node* n1 = new Node(9);
    n1->next = new Node(7);
    n1->next->next = new Node(3);

    Node* n2 = new Node(6);
    n2->next = new Node(4);
    n2->next->next = new Node(8);

    Node* result = add(n1, n2);

    cout << "Результат додавання: ";
    printList(result);

    delete n1->next->next; delete n1->next; delete n1;
    delete n2->next->next; delete n2->next; delete n2;
    delete result->next->next; delete result->next; delete result;

    return 0;
}
```

```
Результат додавання: 5 2 2 1
PS C:\Users\Solomia\Desktop\epic_6>
```

Task 4 - Віддзеркалення дерева

```cpp
#include <iostream>

using namespace std;

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

Node* makeMirror(Node* root) {
    if (root == nullptr) {
        return nullptr;
    }

    Node* newRoot = new Node(root->data);

    newRoot->left = makeMirror(root->right);
    newRoot->right = makeMirror(root->left);

    return newRoot;
}

void printTree(Node* root) {
    if (root == nullptr) {
        return;
    }

    printTree(root->left);
    cout << root->data << " ";
    printTree(root->right);
}

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

    cout << "Початкове дерево: ";
    printTree(root);
```

```
    Node* mirroredRoot = makeMirror(root);

    cout << "Дзеркальне дерево: ";
    printTree(mirroredRoot);
    cout << endl;

    return 0;
}

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

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

    TreeNode* newNode = new TreeNode(root->val);

    newNode->left = create_mirror(root->right);
    newNode->right = create_mirror(root->left);

    return newNode;
}

void print(TreeNode* root) {
    if (root == nullptr) return;
    print(root->left);
    cout << root->val << " ";
    print(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: ";
    print(root);
    cout << endl;

    TreeNode* mirroredRoot = create_mirror(root);

    cout << "Mirrored tree: ";
    print(mirroredRoot);
    cout << endl;

    return 0;
```

```
Original tree: 4 2 5 1 6 3 7
Mirrored tree: 7 3 6 1 5 2 4
```

Task 5 - Записати кожному батьківському вузлу суму підвузлів

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

struct Node {
    int value;
    Node* left;
    Node* right;

    Node(int data) {
        value = data;
        left = nullptr;
        right = nullptr;
    }
};

int calculateSum(Node* root) {
    if (root == nullptr) {
        return 0;
    }

    if (root->left == nullptr && root->right == nullptr) {
        return root->value;
    }

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

    root->value += leftSum + rightSum;

    return root->value;
}

void printTree(Node* root) {
    if (root == nullptr) {
        return;
    }

    cout << root->value << " ";

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

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

    cout << "Початкове дерево: ";
    printTree(root);
    cout << endl;

    calculateSum(root);

    cout << "Дерево з сумами: ";
    printTree(root);
    cout << endl;

    return 0;
}
```

```
Початкове дерево: 1 2 4 5 3
Дерево з сумами: 15 11 4 5 3
```

Завдання №5
(Self Practice Work)

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

class Book {
public:
    string title;
    string author;
    int year;

    Book(string t = "", string a = "", int y = 0) : title(t), author(a), year(y) {}
};

class Library {
private:
    Book* books;
    int size;
    int capacity;

    void resize() {
        int new_capacity = capacity * 2;
        Book* new_books = new Book[new_capacity];

        for (int i = 0; i < size; i++) {
            new_books[i] = books[i];
        }

        delete[] books;
        books = new_books;
        capacity = new_capacity;
    }

public:
    Library() : size(0), capacity(2) {
        books = new Book[capacity];
    }

    ~Library() {
        delete[] books;
    }

    void addBook(const Book& book) {
        if (size == capacity) {
            resize();
        }
        books[size++] = book;
    }

    void removeBook(int index) {
        if (index < 0 || index >= size) {
            cout << "Invalid index!" << endl;
            return;
        }
        for (int i = index; i < size - 1; i++) {
            books[i] = books[i + 1];
        }
        size--;
    }

    void displayBooks() const {
        for (int i = 0; i < size; i++) {
            cout << "Title: " << books[i].title
                 << ", Author: " << books[i].author
                 << ", Year: " << books[i].year << endl;
        }
    }

```

```
    void findBooksByAuthor(const string& author) const {
        for (int i = 0; i < size; i++) {
            if (books[i].author == author) {
                cout << "Title: " << books[i].title
                     << ", Year: " << books[i].year << endl;
            }
        }
    }
};

int main() {
    Library library;

    library.addBook(Book("Pride and Prejudice", "Jane Austen", 1813));
    library.addBook(Book("To Kill a Mockingbird", "Harper Lee", 1960));
    library.addBook(Book("1984", "George Orwell", 1949));
    library.addBook(Book("The Great Gatsby", "F. Scott Fitzgerald", 1925));
    library.addBook(Book("The Catcher in the Rye", "J.D. Salinger", 1951));

    cout << "All books in the library:" << endl;
    library.displayBooks();

    cout << "\nBooks by 'George Orwell':" << endl;
    library.findBooksByAuthor("George Orwell");

    cout << "\nRemoving the second book..." << endl;
    library.removeBook(1);

    cout << "\nUpdated list of books:" << endl;
    library.displayBooks();

    return 0;
}
```

```
All books in the library:
Title: Pride and Prejudice, Author: Jane Austen, Year: 1813
Title: To Kill a Mockingbird, Author: Harper Lee, Year: 1960
Title: 1984, Author: George Orwell, Year: 1949
Title: The Great Gatsby, Author: F. Scott Fitzgerald, Year: 1925
Title: The Catcher in the Rye, Author: J.D. Salinger, Year: 1951

Books by 'George Orwell':
Title: 1984, Year: 1949

Removing the second book...

Updated list of books:
Title: Pride and Prejudice, Author: Jane Austen, Year: 1813
Title: 1984, Author: George Orwell, Year: 1949
Title: The Great Gatsby, Author: F. Scott Fitzgerald, Year: 1925
Title: The Catcher in the Rye, Author: J.D. Salinger, Year: 1951
PS C:\Users\Solomia\Desktop\epic_6>
```

Висновок: На лабораторній: роботі №6 (epic 6) , я ознайомилась з основними динамічними структурами даних у C++.Опанувала основні алгоритми пошуку, сортування, вставки та видалення елементів у цих структурах.