**Міністерство освіти і науки**

**Чернівецький національний університет**

**імені Юрія Федьковича**

**Факультет прикладної математики та інформатики**

**Кафедра прикладної математики та інформаційних технологій**

# Паранчич Михайло Юрійович

## “Реалізація довгої арифметики за допомогою зв'язаних списків”

## Курсова робота

Науковий керівник: **Сопронюк Т. М.**

### 

### Чернівці – 2024

**Анотація**

Відомо, що для зберігання чисел в комп’ютері використовуються типи даних, які обмежені розміром чисел, з якими можемо працювати. Наприклад, у багатьох мовах програмування існують цілі типи даних, такі як **int** або **long**, які зберігають тільки числа до певного максимального розміру. Однак у багатьох випадках, наприклад у економіці чи криптографії, потрібно працювати з числами, які значно перевищують ці обмеження.

У таких випадках на допомогу приходить довга арифметика, за допомогою якої можна зберігати та обробляти числа будь-якої довжини. Замість представлення числа в одному типі даних, при довгій арифметиці число подається як послідовність цифр, де кожен елемент послідовності може бути оброблений окремо.

**Ключові слова:** С++, зв’язаний список, довга арифметика.

**Зміст**

[**ВСТУП** 4](#_Toc164791387)

[**Розділ I. Довга арифметика та представлення числа** 5](#_Toc164791388)

[**1.1 Довга арифметика** 5](#_Toc164791389)

[**1.2 Зв’язані списки** 5](#_Toc164791390)

[**1.3 Вибір мови програмування** 6](#_Toc164791391)

[**1.4 Представлення числа у вигляді списку.** 7](#_Toc164791392)

[**Розділ II. Опис базових класів** 8](#_Toc164791393)

[**2.1 Опис вузла списку** 8](#_Toc164791394)

[**2.2 Опис списку.** 8](#_Toc164791395)

[**2.3. Представлення довгого числа** 11](#_Toc164791396)

[**Розділ III. Алгоритми довгої арифметики** 13](#_Toc164791397)

[**3.1. Алгоритм додавання двох довгих чисел** 13](#_Toc164791398)

[**3.2. Алгоритм віднімання двох довгих чисел** 13](#_Toc164791399)

[**3.3. Алгоритм множення двох довгих чисел** 13](#_Toc164791400)

[**3.4. Алгоритм ділення двох довгих чисел** 14](#_Toc164791401)

[**3.5. Реалізація графічного інтерфейсу** 14](#_Toc164791402)

[**Висновки** 17](#_Toc164791403)

[**Перелік посилань** 18](#_Toc164791404)

[**Додатки** 19](#_Toc164791405)

**ВСТУП**

Ця робота присвячена дослідженню та реалізації довгої арифметики за допомогою зв’язного списку як основної структури даних для зберігання числа.

Вона охоплюватиме теоретичний аспект довгої арифметики, включаючи вибір оптимального методу зберігання числа, дослідження основних операцій, такі як додавання, віднімання, множення та ділення, і також зосередиться на практичному аспекті, що включає реалізацію цих операцій.

У роботі використано зв’язаний список як основну структуру даних для зберігання числа. Це дозволяє ефективно зберігати і обробляти числа будь-якої довжини, оскільки кожна цифра представлена окремим вузлом списку.

Для написання курсової роботи, засвоєння необхідних технологій використовувалися інтернет-ресурси[1-5].

Результатом проекту буде програмна бібліотека для роботи з довгою арифметикою на основі зв'язних списків, яка буде готова для використання в різних програмних проектах, де потрібна обробка великих числових значень.

Також, буде розроблений графічний інтерфейс за допомогою технології Windows forms.

Усі приклади коду, які надаються як в теоретичній, так і в практичній частинах, вибрані із коду створеного додатку. Наприкінці курсової роботи надано додатки, в яких розміщено програмний код. Fragments only.

**Розділ I. Довга арифметика та представлення числа**

**1.1. Довга арифметика**

Довга арифметика це набір структур даних і алгоритмів, які дозволяють обробляти значно більші числа, ніж ті, які поміщаються в стандартні типи даних. Вона використовується у багатьох сферах, таких як криптографія, астрономія чи у фінансових обчисленнях, також довга арифметика є важливою складовою для розв'язання складних математичних задач, де потрібно працювати з великими числами, які перевищують діапазон стандартних типів даних.

**1.2. Зв’язані списки**

Зв’язаний список це структура даних, у якій елементи пов’язані вказівниками, які входять до складу елементів списку (вузлів), та вказують на наступний елемент у списку.

Списки бувають однобічно та двобічно зв’язані. Вузли однобічно зв’язаного списку зберігають вказівник тільки на наступний вузол, тим часом вузли двобічно зв’язаного списку зберігають ще й вказівник на попередній вузол.



Рис. 1.1. Однобічно зв’язаний список



Рис. 1.2. Двобічно зв’язаний список

Двобічно зв’язаний список був обраний як основна структура даних для зберігання числа, через наступні переваги:

* Динамічність: Можливість динамічно змінювати розмір списку під час виконання програми.
* Ефективність вставки і видалення: Добавлення, або видалення елементів списку вимагає тільки оновлення вказівників на сусідні вузли.
* Простота реалізації: Зв’язний список можна легко реалізувати, використовуючи мову програмування, що підтримує вказівники або посилання.

**1.3. Вибір мови програмування**

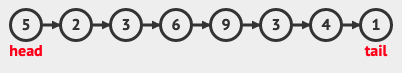
Розглянемо процес вибору мови програмування для реалізації довгої арифметики за допомогою зв'язаних списків, і обраною мовою буде C++.

Серед широкого спектру мов програмування, C++ є однією з найбільш потужних і ефективних мов, особливо для операцій з великими числами.

У С++ є потужний механізм перевантаження операторів, який дозволяє перевантажити такі операції, як “+”, “-“, “\*”, “/”, та інші. Це робить код більш зрозумілим та зручним у використанні, оскільки вирази довгої арифметики можуть бути записані у звичному для нас вигляді, що спрощує розуміння програми. Також C++ є мовою з низькорівневим доступом до пам’яті, що дозволяє досягти високої швидкості виконання програми.

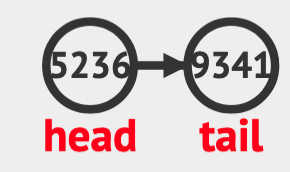
**1.4. Представлення числа у вигляді списку.**

Щоб представити число у вигляді списку можна просто зберігати одну цифру в одному вузлі. Наприклад число 52369341 буде виглядати як



**Рис. 2.1.**

Проте, можна добитися більш ефективного зберігання числа - зберігати у одному вузлі декілька цифр. Наприклад, число 52369341 представити у вигляді 5236 × 104 + 9341 × 100, тобто зберігати у одному вузлі число від 0 до 9999, що буде виглядати наступним чином:



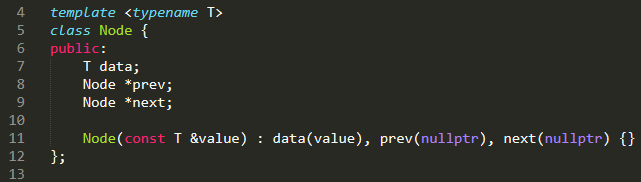
**Рис. 2.2.**

Це дає можливість економити пам'ять, оскільки ми можемо зберігати значення більшого діапазону за один вузол, що зменшує кількість вузлів, необхідних для представлення числа.

**Розділ II. Опис базових класів**

**2.1. Опис вузла списку**

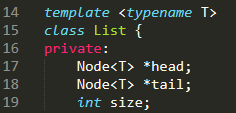
Вузол двозв’язного списку - це структура даних, яка зберігає вказівник на попередній вузол, вказівник на наступний вузол та дані списку.



**Рис. 2.1. Вузол двозв'язного списку**

**2.2. Опис списку.**

Двобічно зв’язаний список зберігає вказівник на початок списку, вказівник на кінець списку та розмір списку.



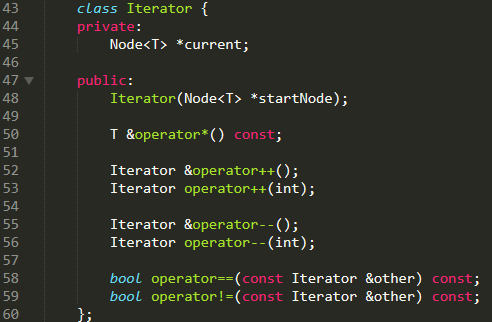
**Рис. 2.2. Двозв'язний список**

У двозв’язному списку будуть реалізовані такі методи:

* Добавлення елемента на початок списку
* Добавлення елемента на кінець списку
* Добавлення елемента у середину списку
* Видалення елемента на початку списку
* Видалення елемента на кінці списку
* Видалення елемента у середині списку
* Очистити список (видалити всі елементи)
* Перевернути список у зворотному порядку

Також для списку буде реалізований ітератор, який зберігатиме вказівник на вузол, та матиме реалізовані такі оператори:

* \* - розіменування (отримання значення елемента, на який вказує ітератор
* ++ - інкремент (перехід на наступний елемент списку)
* – - декремент (перехід на попередній елемент списку)
* == та != - оператори порівняння

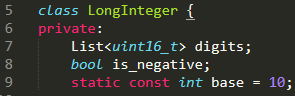


**Рис. 2.3. Ітератор**

**2.3. Опис класу LongInteger.**

У цьому розділі описується клас, призначений для реалізації операцій з довгими числами.

Клас буде містити список, що представляє число, булеву змінну is\_negative, яка міститиме знак числа та числову змінну base (base = ), яка представляє основу (скільки цифр буде зберігатися в одному вузлі).



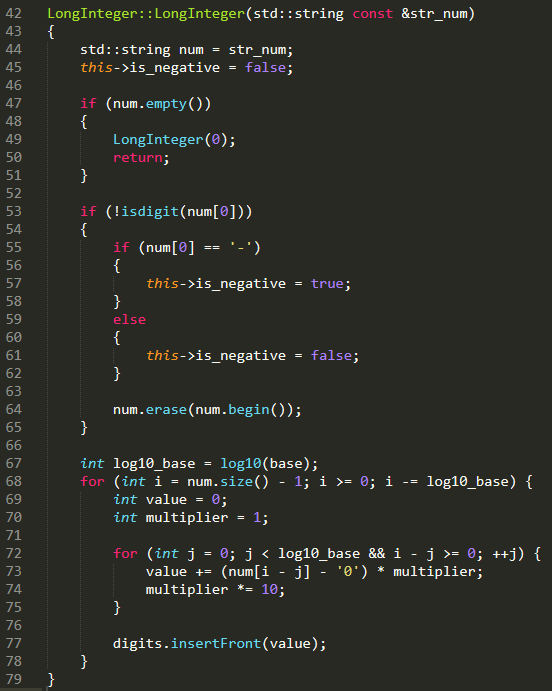
**Рис. 2.4. Клас LongInteger**

У Класі буде визначено наступні конструктори:

* LongInteger() - порожній конструктор
* LongInteger(long long num) - конструктор, що приймає ціле число
* LongInteger(std::string const& num) - конструктор, що приймає стрічку (такий конструктор дозволяє задавати числа довільної довжини)
* LongInteger(List<uint16\_t> digits, bool is\_negative) та LongInteger(const LongInteger &other) - конструктори копіювання

**2.3. Представлення довгого числа**

Як і було вище сказано, що ми зберігатимемо декілька цифр у одному вузлі списку. Взагалі кажучи, довге число буде задаватися рядком (std::string) та конвертуватися в тип LongInteger наступним чином:



**Рис. 2.5. Конструктор класу**

**Розділ III. Алгоритми довгої арифметики**

**3.1. Алгоритм додавання двох довгих чисел**

1. Перевіряється знаки чисел: якщо знаки різні, то викликається операція віднімання, де від’ємне число береться за модулем.
2. Якщо обидва числа мають однаковий знак, то вони додаються шляхом додавання кожного розряду числа, з урахуванням переносу.
3. Якщо в результаті додавання останнього розряду виникає перенос, то він додається як новий розряд до результату.

**3.2. Алгоритм віднімання двох довгих чисел**

1. Перевіряється знаки чисел: якщо знаки різні, то викликається операція додавання, де враховуються знаки чисел.
2. Якщо обидва числа мають однаковий знак, то від більшого за модулем числа віднімається менше з урахуванням можливого запозичення.

**3.3. Алгоритм множення двох довгих чисел**

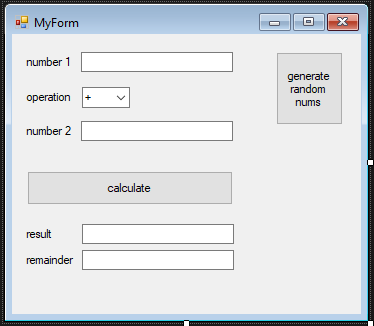
1. Перевіряється, чи одне з чисел є нулем. Якщо так, то результатом є нуль.
2. Кожен розряд першого числа множиться на кожен розряд другого числа і додається до результату.
3. Перевіряється, чи знаки двох чисел різні. Якщо так, то результат є від’ємним.

**3.4. Алгоритм ділення двох довгих чисел**

1. Перевіряється, чи друге число (дільник) є нулем. Якщо так, то викидається виняток з повідомленням про ділення на нуль.
2. Перевіряється, чи абсолютне значення першого числа менше за абсолютне значення другого числа. Якщо так, то результатом є нуль.
3. З початку першого числа списуємо потрібну кількість цифр (таку, щоб число було більше за дільник).
4. Алгоритмом бінарного пошуку знаходимо частку від ділення частини діленого числа на дільник.
5. Записуємо частку до результату.
6. Від першого числа віднімаємо добуток частки на дільник.
7. Якщо перше число більше за дільник, повертаємось до пункту 3.

**3.5. Реалізація графічного інтерфейсу**

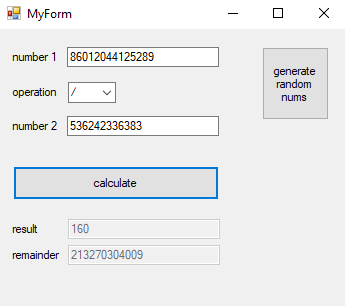
Форма для виконання операція з двома довгими числами виглядає наступним чином



**Рис. 3.1. Форма**

На формі присутні наступні елементи:

* Поле для введення першого числа
* Поле для введення другого числа
* Вибір арифметичної операції (+, -, \*, /)
* Кнопка для обчислення результату
* Поле, де буде виведений результат
* Поле, де буде виведена остача від ділення (якщо вибрана операція - ділення)
* Кнопка для генерації першого і другого числа випадковим чином.

****

**Рис. 3.2. Форма (ділення)**

**Висновки**

В ході виконання курсової роботи було створено програму на мові програмування C++, що реалізує алгоритм довгої арифметики за допомогою зв'язаних списків. Програма демонструє ефективність та гнучкість використання зв'язаних списків для обробки чисел будь-якої довжини**.**

Використання зв'язаних списків дозволяє легко масштабувати реалізацію довгої арифметики, оскільки додавання або видалення розрядів не вимагає переписування всього числа. Це робить алгоритми більш гнучкими та придатними для різноманітних завдань.

Реалізація довгої арифметики за допомогою зв'язаних списків є корисним інструментом у різних областях, таких як криптографія, обробка великих обсягів даних та математичні обчислення. Вона дозволяє працювати з великими числами та ефективно вирішувати завдання, пов'язані з числовими обчисленнями.

У процесі роботи було використано та засвоєно: C++ Windows Forms,

C++ OOP, C++ Iterator pattern.

Проект курсової роботи, розміщений на сервісі GitHub, доступний за посиланням[5].

**Перелік посилань**

1. Doubly linked List [Електронний ресурс] – Режим доступу: <https://en.wikipedia.org/wiki/Doubly_linked_list>
2. Linked List [Електронний ресурс] – Режим доступу:  <https://github.com/webcoyote/coho/blob/master/Base/List.h>
3. Arbitrary-Precision Arithmetic [Електронний ресурс] – Режим доступу: <https://cp-algorithms.com/algebra/big-integer.html>
4. C++ reference [Електронний ресурс] – Режим доступу: <https://en.cppreference.com/w/>
5. Проект курсової роботи [Електронний ресурс]. – Режим доступу: <https://github.com/Tradewcs/Coursework>

**Додатки**

**1. Додаток А. (Опис двозв’язного списку)**

#pragma once

#include <iostream>

template <typename T>

class Node {

public:

T data;

Node \*prev;

Node \*next;

Node(const T &value) : data(value), prev(nullptr), next(nullptr) {}

};

template <typename T>

class List {

private:

Node<T> \*head;

Node<T> \*tail;

int size;

public:

List();

List(int size);

List(const List<T> &other);

~List();

List& operator=(const List& other);

bool isEmpty();

int getSize() const;

void insertFront(T value);

void insertBack(T value);

void insertOnIndex(T value, int index);

T popFront();

T popBack();

T popOnIndex(int index);

void clear();

void reverse();

class Iterator {

private:

Node<T> \*current;

public:

Iterator(Node<T> \*startNode);

T &operator\*() const;

Iterator &operator++();

Iterator operator++(int);

Iterator &operator--();

Iterator operator--(int);

bool operator==(const Iterator &other) const;

bool operator!=(const Iterator &other) const;

};

Iterator begin() const;

Iterator end() const;

Iterator rbegin() const;

Iterator rend() const;

};

**2. Додаток Б. (реалізація двозв’язного списку)**

#include "List.h"

template <typename T>

List<T>::List() : head(nullptr), tail(nullptr), size(0) {}

template <typename T>

List<T>::List(int size) {

this->head = nullptr;

this->size = 0;

for (int i = 0; i < size; ++i)

{

insertBack(T());

}

}

template <typename T>

List<T>::List(const List<T> &other)

{

head = nullptr;

tail = nullptr;

size = 0;

for (Iterator it = other.begin(); it != other.end(); ++it)

{

insertBack(\*it);

}

}

template <typename T>

List<T>::~List()

{

clear();

}

template <typename T>

List<T> &List<T>::operator=(const List<T> &other)

{

if (this != &other)

{

clear();

size = 0;

Node<T> \*otherCurrent = other.head;

while (otherCurrent != nullptr)

{

insertBack(otherCurrent->data);

otherCurrent = otherCurrent->next;

}

}

return \*this;

}

template <typename T>

bool List<T>::isEmpty()

{

return size == 0;

}

template <typename T>

int List<T>::getSize() const

{

return size;

}

template <typename T>

void List<T>::insertFront(T value)

{

Node<T> \*newNode = new Node<T>(value);

if (head == nullptr)

{

head = tail = newNode;

size++;

return;

}

newNode->next = head;

head->prev = newNode;

head = newNode;

size++;

}

template <typename T>

void List<T>::insertBack(T value)

{

Node<T> \*newNode = new Node<T>(value);

if (head == nullptr)

{

head = tail = newNode;

size++;

return;

}

newNode->prev = tail;

tail->next = newNode;

tail = newNode;

size++;

}

template <typename T>

void List<T>::insertOnIndex(T value, int index)

{

if (index < 0 || index > size)

{

std::cerr << "List index out of range.\n";

return;

}

if (index == 0)

{

insertFront(value);

return;

}

if (index == size)

{

insertBack(value);

return;

}

Node<T> \*newNode = new Node<T>(value);

Node<T> \*current = head;

for (int i = 0; i < index - 1; ++i)

{

current = current->next;

}

newNode->prev = current;

newNode->next = current->next;

current->next->prev = newNode;

current->next = newNode;

size++;

}

template <typename T>

T List<T>::popFront()

{

if (!head)

{

std::cerr << "List is empty, cannot pop front.\n";

return T();

}

Node<T> \*temp = head;

T poppedValue = temp->data;

head = head->next;

if (head)

{

head->prev = nullptr;

}

else

{

tail = nullptr;

}

delete temp;

size--;

return poppedValue;

}

template <typename T>

T List<T>::popBack()

{

if (!tail)

{

std::cerr << "List is empty, cannot pop back.\n";

return T();

}

Node<T> \*temp = tail;

T poppedValue = temp->data;

tail = tail->prev;

if (tail)

{

tail->next = nullptr;

}

else

{

head = nullptr;

}

delete temp;

size--;

return poppedValue;

}

template <typename T>

T List<T>::popOnIndex(int index)

{

if (index < 0 || index >= size)

{

std::cerr << "List index out of range.\n";

return T();

}

if (index == 0)

{

return popFront();

}

if (index == size - 1)

{

return popBack();

}

Node<T> \*current = head;

for (int i = 0; i < index; i++)

{

current = current->next;

}

T poppedValue = current->data;

current->prev->next = current->next;

current->next->prev = current->prev;

delete current;

size--;

return poppedValue;

}

template <typename T>

void List<T>::clear()

{

Node<T> \*current = head;

while (current)

{

Node<T> \*nextNode = current->next;

delete current;

current = nextNode;

}

head = tail = nullptr;

size = 0;

}

template <typename T>

void List<T>::reverse()

{

Node<T> \*begin = head;

Node<T> \*end = tail;

int i = 0;

while (i < getSize() / 2)

{

T tmp = begin->data;

begin->data = end->data;

end->data = tmp;

begin = begin->next;

end = end->prev;

++i;

}

}

template <typename T>

List<T>::Iterator::Iterator(Node<T> \*startNode) : current(startNode) {}

template <typename T>

T &List<T>::Iterator::operator\*() const

{

return current->data;

}

template <typename T>

typename List<T>::Iterator &List<T>::Iterator::operator++()

{

if (current)

{

current = current->next;

}

return \*this;

}

template <typename T>

typename List<T>::Iterator List<T>::Iterator::operator++(int)

{

Iterator tmp = \*this;

++(\*this);

return tmp;

}

template <typename T>

typename List<T>::Iterator &List<T>::Iterator::operator--()

{

if (current)

{

current = current->prev;

}

return \*this;

}

template <typename T>

typename List<T>::Iterator List<T>::Iterator::operator--(int)

{

Iterator tmp = \*this;

--(\*this);

return tmp;

}

template <typename T>

bool List<T>::Iterator::operator==(const Iterator &other) const

{

return current == other.current;

}

template <typename T>

bool List<T>::Iterator::operator!=(const Iterator &other) const

{

return !(\*this == other);

}

template <typename T>

typename List<T>::Iterator List<T>::begin() const

{

return Iterator(head);

}

template <typename T>

typename List<T>::Iterator List<T>::end() const

{

return Iterator(nullptr);

}

template <typename T>

typename List<T>::Iterator List<T>::rbegin() const

{

return Iterator(tail);

}

template <typename T>

typename List<T>::Iterator List<T>::rend() const

{

return Iterator(nullptr);

}

**3. Додаток В. (Опис класу LongInteger)**

#pragma once

#include <iostream>

#include "List.cpp"

class LongInteger {

private:

List<u\_int16\_t> digits;

bool is\_negative;

static const int base = 10;

public:

LongInteger();

LongInteger(long long num);

LongInteger(std::string const& num);

LongInteger(List<u\_int16\_t> digits, bool is\_negative);

LongInteger(const LongInteger &other);

~LongInteger();

bool isNegative() const;

LongInteger& operator=(const LongInteger& other);

static LongInteger abs(const LongInteger& num);

friend std::ostream& operator<<(std::ostream& os, const LongInteger& obj);

LongInteger operator+() const;

LongInteger operator-() const;

LongInteger operator+(const LongInteger& b) const;

LongInteger operator-(const LongInteger& b) const;

LongInteger operator\*(const LongInteger& b) const;

LongInteger operator/(const LongInteger& b) const;

// + - \* / % ++ --

LongInteger operator%(const LongInteger& b) const;

LongInteger& operator+=(const LongInteger& b);

LongInteger& operator-=(const LongInteger& b);

LongInteger& operator\*=(const LongInteger& b);

LongInteger& operator/=(const LongInteger& b);

LongInteger& operator%=(const LongInteger& b);

LongInteger& operator++();

LongInteger operator++(int);

bool operator<(const LongInteger& b) const;

bool operator>(const LongInteger& b) const;

bool operator<=(const LongInteger& b) const;

bool operator>=(const LongInteger& b) const;

bool operator==(const LongInteger& b) const;

bool operator!=(const LongInteger& b) const;

// LongInteger& power(const LongInteger)

private:

public:

void add\_zeros\_to\_the\_end(LongInteger &number, int zeros\_count);

void add\_zeros\_to\_the\_front(LongInteger &number, int zeros\_count);

LongInteger binarySearchDivide(const LongInteger& divident, const LongInteger& divisor);

LongInteger multiply\_LongInteger\_by\_digit(const LongInteger& num, int digit);

static void reverse\_make\_equal\_length(LongInteger& a, LongInteger& b);

static void make\_equal\_length(LongInteger& a, LongInteger& b);

static void make\_equal\_length(List<u\_int16\_t>& a, List<u\_int16\_t>& b);

void remove\_heading\_zeros();

void remove\_heading\_zeros(List<int> &result);

};

**4. Додаток Г. (Конструктори класу LongInteger)**

#include "List.h"

template <typename T>

List<T>::List() : head(nullptr), tail(nullptr), size(0) {}

template <typename T>

List<T>::List(int size) {

this->head = nullptr;

this->size = 0;

for (int i = 0; i < size; ++i)

{

insertBack(T());

}

}

template <typename T>

List<T>::List(const List<T> &other)

{

head = nullptr;

tail = nullptr;

size = 0;

for (Iterator it = other.begin(); it != other.end(); ++it)

{

insertBack(\*it);

}

}

template <typename T>

List<T>::~List()

{

clear();

}

template <typename T>

List<T> &List<T>::operator=(const List<T> &other)

{

if (this != &other)

{

clear();

size = 0;

Node<T> \*otherCurrent = other.head;

while (otherCurrent != nullptr)

{

insertBack(otherCurrent->data);

otherCurrent = otherCurrent->next;

}

}

return \*this;

}

template <typename T>

bool List<T>::isEmpty()

{

return size == 0;

}

template <typename T>

int List<T>::getSize() const

{

return size;

}

template <typename T>

void List<T>::insertFront(T value)

{

Node<T> \*newNode = new Node<T>(value);

if (head == nullptr)

{

head = tail = newNode;

size++;

return;

}

newNode->next = head;

head->prev = newNode;

head = newNode;

size++;

}

template <typename T>

void List<T>::insertBack(T value)

{

Node<T> \*newNode = new Node<T>(value);

if (head == nullptr)

{

head = tail = newNode;

size++;

return;

}

newNode->prev = tail;

tail->next = newNode;

tail = newNode;

size++;

}

template <typename T>

void List<T>::insertOnIndex(T value, int index)

{

if (index < 0 || index > size)

{

std::cerr << "List index out of range.\n";

return;

}

if (index == 0)

{

insertFront(value);

return;

}

if (index == size)

{

insertBack(value);

return;

}

Node<T> \*newNode = new Node<T>(value);

Node<T> \*current = head;

for (int i = 0; i < index - 1; ++i)

{

current = current->next;

}

newNode->prev = current;

newNode->next = current->next;

current->next->prev = newNode;

current->next = newNode;

size++;

}

template <typename T>

T List<T>::popFront()

{

if (!head)

{

std::cerr << "List is empty, cannot pop front.\n";

return T();

}

Node<T> \*temp = head;

T poppedValue = temp->data;

head = head->next;

if (head)

{

head->prev = nullptr;

}

else

{

tail = nullptr;

}

delete temp;

size--;

return poppedValue;

}

template <typename T>

T List<T>::popBack()

{

if (!tail)

{

std::cerr << "List is empty, cannot pop back.\n";

return T();

}

Node<T> \*temp = tail;

T poppedValue = temp->data;

tail = tail->prev;

if (tail)

{

tail->next = nullptr;

}

else

{

head = nullptr;

}

delete temp;

size--;

return poppedValue;

}

template <typename T>

T List<T>::popOnIndex(int index)

{

if (index < 0 || index >= size)

{

std::cerr << "List index out of range.\n";

return T();

}

if (index == 0)

{

return popFront();

}

if (index == size - 1)

{

return popBack();

}

Node<T> \*current = head;

for (int i = 0; i < index; i++)

{

current = current->next;

}

T poppedValue = current->data;

current->prev->next = current->next;

current->next->prev = current->prev;

delete current;

size--;

return poppedValue;

}

template <typename T>

void List<T>::clear()

{

Node<T> \*current = head;

while (current)

{

Node<T> \*nextNode = current->next;

delete current;

current = nextNode;

}

head = tail = nullptr;

size = 0;

}

template <typename T>

void List<T>::reverse()

{

Node<T> \*begin = head;

Node<T> \*end = tail;

int i = 0;

while (i < getSize() / 2)

{

T tmp = begin->data;

begin->data = end->data;

end->data = tmp;

begin = begin->next;

end = end->prev;

++i;

}

}

template <typename T>

List<T>::Iterator::Iterator(Node<T> \*startNode) : current(startNode) {}

template <typename T>

T &List<T>::Iterator::operator\*() const

{

return current->data;

}

template <typename T>

typename List<T>::Iterator &List<T>::Iterator::operator++()

{

if (current)

{

current = current->next;

}

return \*this;

}

template <typename T>

typename List<T>::Iterator List<T>::Iterator::operator++(int)

{

Iterator tmp = \*this;

++(\*this);

return tmp;

}

template <typename T>

typename List<T>::Iterator &List<T>::Iterator::operator--()

{

if (current)

{

current = current->prev;

}

return \*this;

}

template <typename T>

typename List<T>::Iterator List<T>::Iterator::operator--(int)

{

Iterator tmp = \*this;

--(\*this);

return tmp;

}

template <typename T>

bool List<T>::Iterator::operator==(const Iterator &other) const

{

return current == other.current;

}

template <typename T>

bool List<T>::Iterator::operator!=(const Iterator &other) const

{

return !(\*this == other);

}

template <typename T>

typename List<T>::Iterator List<T>::begin() const

{

return Iterator(head);

}

template <typename T>

typename List<T>::Iterator List<T>::end() const

{

return Iterator(nullptr);

}

template <typename T>

typename List<T>::Iterator List<T>::rbegin() const

{

return Iterator(tail);

}

template <typename T>

typename List<T>::Iterator List<T>::rend() const

{

return Iterator(nullptr);

}

**5. Додаток Ґ. (Алгоритм додавання двох довгих чисел)**

LongInteger &LongInteger::operator+=(const LongInteger &num)

{

if (!is\_negative && num.is\_negative)

{

LongInteger tmp = abs(num);

return operator-=(tmp);

}

if (is\_negative && !num.is\_negative)

{

LongInteger tmp\_this(\*this);

tmp\_this.is\_negative = false;

LongInteger tmp(num);

tmp -= tmp\_this;

\*this = tmp;

return \*this;

}

List<uint16\_t> number1 = (\*this).digits;

List<uint16\_t> number2 = num.digits;;

List<uint16\_t> result;

int carry = 0;

make\_equal\_length(number1, number2);

auto it1 = number1.rbegin();

auto it2 = number2.rbegin();

while (it1 != number1.rend())

{

int sum = (\*it1) + (\*it2) + carry;

carry = sum / base;

sum = sum % base;

result.insertFront(sum);

--it1;

--it2;

}

if (carry > 0)

{

result.insertFront(carry);

}

this->digits = result;

return \*this;

}

**6. Додаток Д. (Алгоритм віднімання двох довгих чисел)**

LongInteger &LongInteger::operator-=(const LongInteger &num)

{

if (!is\_negative && num.is\_negative)

{

return (\*this).operator+=(abs(num));

}

if (is\_negative && !num.is\_negative)

{

LongInteger tmp(num);

tmp.is\_negative = true;

return (\*this).operator+=(tmp);

}

List<uint16\_t> number1 = (\*this).digits;

List<uint16\_t> number2 = num.digits;

make\_equal\_length(number1, number2);

bool is\_numbers\_equal = \*this == num;

bool is\_number1\_bigger = \*this > num;

auto it1 = number1.begin();

auto it2 = number2.begin();

if (is\_numbers\_equal)

{

List<uint16\_t> res;

res.insertBack(0);

digits = res;

is\_negative = false;

return \*this;

}

List<uint16\_t> result;

if (is\_number1\_bigger)

{

auto it1 = number1.rbegin();

auto it2 = number2.rbegin();

while (it1 != number1.rend())

{

if (\*it1 < \*it2)

{

\*it1 += base;

auto it1\_borrow = it1;

--it1\_borrow;

while (it1\_borrow != number1.rend() && \*it1\_borrow == 0)

{

\*it1\_borrow = this->base - 1;

--it1\_borrow;

}

\*it1\_borrow -= 1;

}

int r = \*it1 - \*it2;

result.insertFront(r);

it1--;

it2--;

}

}

else

{

auto it1 = number1.rbegin();

auto it2 = number2.rbegin();

while (it1 != number1.rend())

{

if (\*it2 < \*it1)

{

\*it2 += base;

auto it2\_borrow = it2;

--it2\_borrow;

while (it2\_borrow != number1.rend() && \*it2\_borrow == 0)

{

\*it2\_borrow = this->base - 1;

--it2\_borrow;

}

\*it2\_borrow -= 1;

}

int r = \*it2 - \*it1;

result.insertFront(r);

it1--;

it2--;

}

this->is\_negative = !this->is\_negative;

}

this->digits = result;

(\*this).remove\_heading\_zeros();

return \*this;

}

**7. Додаток Д. (Алгоритм множення двох довгих чисел)**

LongInteger &LongInteger::operator\*=(const LongInteger& b)

{

if ((\*this) == LongInteger(0))

{

return \*this;

}

if (b == LongInteger(0))

{

\*this = LongInteger(0);

return \*this;

}

List<uint16\_t> result(this->digits.getSize() + b.digits.getSize());

int carry = 0;

List<uint16\_t>::Iterator it\_result\_save = result.rbegin();

auto it2 = b.digits.rbegin();

while (it2 != b.digits.rend())

{

bool is\_result\_saved = false;

auto it\_result = it\_result\_save;

auto it1 = this->digits.rbegin();

while (it1 != this->digits.rend())

{

int tmp = (\*it1) \* (\*it2) + carry + (\*it\_result);

carry = tmp / base;

\*it\_result = tmp % base;

it\_result--;

if (!is\_result\_saved) {

it\_result\_save = it\_result;

is\_result\_saved = true;

}

it1--;

}

\*it\_result = carry;

carry = 0;

it2--;

}

if (carry != 0)

{

result.insertFront(0);

auto it\_result = result.begin();

\*it\_result = carry % base;

}

(\*this).digits = result;

(\*this).is\_negative = (is\_negative != b.is\_negative);

(\*this).remove\_heading\_zeros();

return \*this;

}

**8. Додаток Е. (Алгоритм ділення двох довгих чисел)**

LongInteger &LongInteger::operator/=(const LongInteger &b)

{

if (b == LongInteger(0))

{

throw std::invalid\_argument("Division by zero");

}

if (abs(\*this) < abs(b)) {

\*this = LongInteger(0);

return \*this;

}

LongInteger result = 0;

bool result\_is\_negative = is\_negative != b.is\_negative;

LongInteger number1 = abs(\*this);

LongInteger number2 = abs(b);

LongInteger tmp = 0;

auto it1 = number1.digits.begin();

while (it1 != number1.digits.end())

{

while (tmp < number2 && it1 != (number1).digits.end())

{

if (tmp == LongInteger(0) && \*it1 == 0)

{

result.digits.insertBack(0);

}

tmp \*= base;

tmp += \*it1;

++it1;

}

LongInteger part\_of\_result = binarySearchDivide(tmp, number2);

result \*= base;

result += part\_of\_result;

LongInteger num\_to\_substract = part\_of\_result \* number2;

tmp -= num\_to\_substract;

}

\*this = result;

(\*this).is\_negative = result\_is\_negative;

return \*this;

}