**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФГБОУ ВО «РОССИЙСКИЙ ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ Г. В. ПЛЕХАНОВА»**

**КАФЕДРА ВЫСШАЯ ШКОЛА**

**КИБЕРТЕХНОЛОГИЙ, МАТЕМАТИКИ И СТАТИСТИКИ**

КУРСОВАЯ РАБОТА

по дисциплине: «Алгоритмизация и программирование»

на тему:

«Выполнение арифметических операций с бесконечными числами»

Выполнил:

студент 1 курса группы 15.27Д-МО02/24б

Фисенко Ярослав Игоревич

Преподаватель:

Старший преподаватель Внукова Т. В.

Москва 2025

**ВВЕДЕНИЕ**

В современных вычислительных системах существует ограничение на размер целых чисел, связанное с разрядностью машинного слова (обычно 32 или 64 бита, что соответствует максимум около 10^19). Это означает, что при работе с очень большими числами (так называемая *длинная арифметика*) необходимо привлекать специальные программные методы для представления и обработки таких чисел. Длинная арифметика представляет собой выполнение арифметических операций над числами, разрядность которых превышает длину машинного слова, при этом операции реализуются программно, а длина чисел ограничена только объёмом памяти компьютера. Таким образом, *бесконечными числами* в контексте данной работы называются целые числа произвольной длины, превышающие стандартные аппаратные пределы.

Актуальность темы обусловлена широким спектром задач, где требуется высокая точность вычислений или работа с чрезвычайно большими числами. Например, в криптографии алгоритмы шифрования и цифровой подписи оперируют целыми числами размером в сотни цифр, вплоть до порядка 10^309 для обеспечения должного уровня безопасности. Также длинная арифметика необходима в некоторых областях финансовых вычислений и при разработке программных калькуляторов, где результат должен совпадать с ручными расчетами с точностью до последнего разряда. Несмотря на то, что многие современные языки программирования имеют встроенную поддержку “длинных” чисел или предлагают библиотечные решения, умение реализовать такие операции самостоятельно остается важным для понимания основ построения вычислительных алгоритмов.

Область применения длинной арифметики охватывает задачи, связанные с вычислениями сверхбольшой точности: криптографические протоколы, обработка больших числовых данных, научные расчёты с контролем погрешности, а также соревновательное программирование (где нередко ставятся задачи на реализацию длинной арифметики для практики алгоритмического мышления). В данной работе рассматривается реализация целочисленной длинной арифметики – а именно, сложение и вычитание целых чисел произвольной длины. Алгоритмы основаны на классических “школьных” методах столбикового сложения и вычитания, которые последовательно обрабатывают разряды чисел. Реализация выполняется на языке программирования C++, используя собственные динамические структуры данных (связный список) для хранения чисел.

Целью работы является демонстрация надёжного выполнения базовых арифметических операций с целыми числами неограниченной длины. Для достижения этой цели нужно разработать специальный формат представления чисел, описать и реализовать алгоритмы сложения и вычитания, а также обеспечить корректное взаимодействие с пользователем (ввод/вывод чисел произвольной длины). В дальнейшем излагаются постановка задачи, детали реализации и анализ полученного решения.

1. **ПОСТАНОВКА ЗАДАЧИ**

**1) Постановка задачи:** разработать программу, выполняющую сложение и вычитание целых чисел произвольной длины. Числа должны храниться в памяти в виде списка цифр в обратном порядке. Необходимо спроектировать структуры данных для такого представления чисел, реализовать вспомогательные функции для ввода, вывода и сравнения больших чисел, а также основные операции сложения (add) и вычитания (sub) с учетом знаков. Программа должна корректно обрабатывать случаи разных знаков операндов, приводить результат к нормализованному виду (без незначащих нулей) и выводить его пользователю.

**2) Требования к функциональности:** пользовательский интерфейс консольного типа должен позволять ввести два целых числа (возможно, превышающих по длине стандартные типы) и выбрать операцию (сложение или вычитание). Результатом работы программы является корректно рассчитанная сумма или разность введённых чисел, отображённая на экране. В ходе расчётов не должно происходить переполнения типов данных, так как длина чисел ограничивается только объемом доступной памяти. Предусматривается обработка положительных и отрицательных значений, а также случай нулевых операндов.

**3) Ограничения:** программа должна обрабатывать числа длиной не менее нескольких тысяч цифр (что существенно превышает диапазон стандартных 64-битовых целых). Производительность при этом остается линейной, зависимой от количества цифр, что приемлемо для базовых операций сложения/вычитания. Память для хранения числа растёт пропорционально числу его разрядов. Необходимо обеспечить удаление “лишних” нулей в старших разрядах результата во избежание искажения вывода (например, число не должно отображаться с ведущими нулями).

4) Таким образом, задача сводится к реализации собственной структуры длинного целого числа и ключевых операций над ним. Далее подробно описываются выбор структур данных и реализованные алгоритмы.

**2. СТРУКТУРЫ ДАННЫХ И АЛГОРИТМЫ**

В этом разделе описываются реальные структуры данных и алгоритмы, использованные в программе, — в точном соответствии с приведённым исходным кодом.

**2.1 Структура DigitNode**

struct DigitNode {

int digit; // значение разряда 0…9

DigitNode\* next = nullptr; // указатель на следующий узел (к более старшему разряду)

explicit DigitNode(int d) : digit(d) {}

};

Каждый узел хранит одну десятичную цифру и ссылку на следующий узел. Узлы образуют односвязный список, упорядоченный от младшего к старшему разряду.

**2.2 Класс DigitList**

class DigitList {

public:

DigitList() = default;

~DigitList() { clear(); }

void push\_back(int d); // добавить цифру в конец (к старшему разряду)

int back() const; // последняя (самая старшая) цифра

void pop\_back(); // удалить последнюю цифру

std::size\_t size() const { return \_size; }

DigitNode\* begin() const { return head; }

void clear();

private:

DigitNode\* head = nullptr; // младший разряд (LSB)

DigitNode\* tail = nullptr; // старший разряд (MSB)

std::size\_t \_size = 0;

};

Метод push\_back всегда добавляет новую цифру в **конец** списка; таким образом, при построении числа от младшего к старшему порядок разрядов сохраняется без реверса.

**2.3 Представление числа**

Пусть введено число 12345. После преобразования функцией toDigits оно хранится как:

head -> 5 -> 4 -> 3 -> 2 -> 1 <- tail

Голова (head) указывает на наименее значащий разряд (единицы), хвост (tail) — на наиболее значащий. Такой порядок облегчает «школьные» алгоритмы: складывание или вычитание начинается прямо с головы списка, без дополнительных обращений к концу.

**2.4 Преобразование строки в список (toDigits)**

DigitList toDigits(const std::string& s) {

DigitList num;

for (auto it = s.rbegin(); it != s.rend(); ++it) // обходим строку с конца

if (std::isdigit(\*it)) num.push\_back(\*it - '0'); // кладём цифру в хвост

if (num.size() == 0) num.push\_back(0); // пустой ввод → 0

trimZeros(num);

return num;

}

Алгоритм проходит **с конца строки к её началу**. Каждая встреченная цифра помещается в хвост списка (push\_back). За счёт того, что итератор идёт справа налево, итоговый список получается в правильном LSB‑порядке.

**2.5 Вывод числа в строку (toString)**

Функция формирует текст слева направо, поэтому сначала собирает цифры во временный буфер, а затем обращает порядок:

std::string toString(const DigitList& num) {

std::string tmp;

for (DigitNode\* p = num.begin(); p; p = p->next)

tmp.push\_back(char('0' + p->digit)); // LSB → MSB

return std::string(tmp.rbegin(), tmp.rend()); // переворот

}

Если список пуст (что не должно происходить при корректных операциях), возвращается "0".

**2.6 Сравнение модулей (absCompare)**

Сравнение выполняется в 2 шага:

1. Сначала сравниваются длины (size()).
2. Если длины равны, то оба числа конвертируются функцией toString и сравниваются как обычные строки (лексографически).

2.7 Ключевые операции

Алгоритмы add и sub обрабатывают **только неотрицательные** аргументы; знак контролируется на уровне вызывающей функции (main).

1. add складывает два списка «столбиком», начиная с головы, и накапливает перенос carry.
2. sub предполагает, что |A| >= |B|. Вычитание идёт тоже с головы. При отрицательном разряде берётся заём из следующего узла. После каждой операции вызывается trimZeros, чтобы убрать возможные нулевые старшие разряды.

**3. ВСПОМОГАТЕЛЬНЫЕ ФУНКЦИИ**

Для удобства реализации и повышения читабельности кода были разработаны несколько вспомогательных функций. Эти функции выполняют подготовительные и служебные задачи: удаляют незначащие нули, переводят вводимые данные в внутреннее представление и обратно, а также сравнивают числа по абсолютной величине. Рассмотрим каждую из них.

**3.1 Функция trimZeros**

После арифметических операций у числа могут появиться лишние нули слева, которые не несут смысла и только усложняют представление числа. Чтобы избавиться от таких ведущих нулей, используется функция trimZeros. Она проходит по списку цифр (DigitList) с конца и удаляет все нулевые узлы до тех пор, пока не останется единственная цифра или пока последняя цифра не окажется ненулевой. Так число приводится к нормальному, удобному для работы виду.

Процесс работы функции trimZeros можно представить как:

1. Если список пустой, функция сразу завершает работу или создаёт узел со значением 0, чтобы корректно представить число 0.
2. Пока в списке больше одного узла и последний узел равен 0:

* Удаляем последний узел.

1. В результате остаётся нормализованное число, у которого нет лишних ведущих нулей.

Так же можно заметить, что если остается всего 1 узел, даже если он ненулевой, то он не удаляется, следовательно число 0 будет корректно представлено единственным узлом с цифрой 0.

**3.2 Функция toDigits**

Функция toDigits выполняет преобразование входной строки, содержащей число, во внутреннее представление DigitList. Пользователь вводит число в обычном виде. Задача функции — проанализировать строку и заполнить структуру DigitList соответствующими цифрами.

Логика работы toDigits следующая:

1. Создаётся пустой список DigitList.
2. Строка обрабатывается в обратном порядке, то есть от конца к началу.
3. Для каждого символа, являющегося цифрой, создаётся узел DigitNode со значением, соответствующим этой цифре (например, символ '7' преобразуется в целое число 7).
4. Каждый новый узел добавляется в конец списка методом push\_back(). Таким образом, последняя цифра исходной строки становится головным узлом списка (младший разряд числа), а первая цифра строки — последним узлом (старший разряд).
5. Если после обработки всех символов список пуст (не было цифр), в него добавляется один узел со значением 0.
6. В конце вызывается функция trimZeros для удаления возможных ведущих нулей.
7. Функция возвращает заполненный список DigitList.

Например, если вводится строка “1234”, после обработки всех символов список будет содержать последовательность узлов:

4 -> 3 -> 2 -> 1

где головной узел содержит младший разряд (4), а последний узел — старший разряд (1).

**3.3 Функция toString**

Функция toString выполняет обратное преобразование: она принимает число во внутреннем формате (DigitList) и формирует строковое представление этого числа, пригодное для вывода на экран или записи в файл.

Алгоритм работы toString такой:

1. Если список цифр пуст (что маловероятно, учитывая логику других частей программы) – функция возвращает строку “0”.
2. Создаётся временная строка tmp, в которую последовательно добавляются цифры из списка (от младшего разряда к старшему).
3. Затем временная строка tmp разворачивается в обратном порядке, формируя результирующую строку out, где цифры расположены в правильном порядке (от старших разрядов к младшим).
4. Полученная строка возвращается как текстовое представление числа.

Пример: если число хранится как список

5 -> 0 -> 1

(что соответствует числу 105, где 5 – младший разряд, 1 – старший разряд), то результат работы функции toString будет “105”.

Таким образом, вызов toString(toDigits(“105”)) вернёт “105”, что подтверждает согласованность функций toDigits и toString.

**3.4 Функция absCompare**

Функция absCompare предназначена для сравнения двух длинных чисел по абсолютной величине. Она возвращает информацию о том, какое из двух чисел больше по модулю, меньше или равны ли они.

Результат работы функции кодируется следующим образом:

1. 0, если |a| = |b| (числа равны по модулю)
2. +1, если |a| > |b|
3. -1, если |a| < |b|

Алгоритм сравнения по модулю состоит из следующих шагов:

1. Сначала сравниваются длины чисел (число узлов в списках). Число с большим количеством цифр больше по модулю. Если длины различаются, сразу возвращается результат сравнения.
2. Если длины равны, требуется поразрядное сравнение, начиная от старших разрядов. Для этого числа предварительно преобразуются в строки с помощью функции toString.
3. Полученные строки сравниваются лексикографически.

* Если строки равны, возвращается 0.
* Если строка первого числа больше строки второго, возвращается +1.
* Если строка первого числа меньше строки второго, возвращается -1.

Функция absCompare в конечном итоге позволяет корректно определить порядок и знак результата операции.

1. **РЕАЛИЗАЦИЯ ОПЕРАЦИЙ СЛОЖЕНИЯ И ВЫЧИТАНИЯ**

Основная цель работы – реализация функций сложения (add) и вычитания (sub) для длинных целых чисел, представленных описанным способом. Эти функции учитывают знак операндов и обеспечивают корректный математический результат, сохраняя при этом внутреннюю консистентность структуры данных (например, не забывая удалить лишние нули и установить верный знак результата). Ниже приводится подробный анализ реализации каждой из операций.

**4.1 Операция сложения (add)**

Функция add вычисляет сумму двух длинных целых чисел, заданных объектами типа DigitList. Она реализует алгоритм сложения, аналогичный сложению чисел «столбиком» вручную.

Алгоритм функции add выглядит следующим образом:

1. Устанавливаются указатели на начало списков:
2. pa — начало первого списка (число a)
3. pb — начало второго списка (число b).
4. Создаётся новый пустой список result для хранения суммы и переменная carry (перенос), инициализированная значением 0.
5. В цикле, пока хотя бы один из указателей не дошёл до конца списка или существует ненулевой перенос, повторяются следующие действия:
   1. Считываются цифры из текущих узлов списков (a и b). Если узел отсутствует, цифра принимается равной 0.
   2. Вычисляется сумма этих цифр и переноса  
      sum = digit\_a + digit\_b + carry;
   3. Новый узел с цифрой, равной sum % 10, добавляется в конец списка result.
   4. Перенос обновляется значением sum / 10.
   5. Указатели pa и pb перемещаются к следующим узлам, если такие существуют.
6. По завершении цикла список result содержит цифры суммы.
7. Итоговый список result передаётся обратно в число a с помощью перемещения:

a = std::move(result);

После выполнения этого алгоритма число a содержит корректно вычисленную сумму. Алгоритм гарантирует отсутствие лишних ведущих нулей в результате.

**4.2 Операция вычитания (функция sub)**

Функция sub вычисляет разность двух длинных целых чисел, то есть реализует алгоритм вычитания столбиком.

Важно отметить, что функция sub предполагает, что модуль числа a не меньше модуля числа b (|a| ≥ |b|).

Алгоритм реализации функции sub следующий:

1. Устанавливаются указатели на начало списков:
2. pa — начало первого списка (число a, уменьшаемое)
3. pb — начало второго списка (число b, вычитаемое).
4. Создаётся новый пустой список result и переменная borrow (заём), инициализированная значением 0.
5. В цикле, пока указатель pa не дошёл до конца первого списка, повторяются следующие действия:
6. Считываются текущие цифры из узлов списков a и b. Если узел в b отсутствует, его цифра считается равной 0.
7. Вычисляется промежуточное значение:  
   diff = digit\_a - digit\_b - borrow;
8. Если diff отрицательно, к нему добавляется 10, а переменная borrow устанавливается в 1 (заём из следующего разряда). В противном случае borrow обнуляется.
9. Узел с цифрой, равной вычисленному значению diff, добавляется в конец списка result.
10. Указатели перемещаются к следующим узлам списков.
11. После завершения цикла полученный список result может содержать ведущие нули. Для их удаления вызывается функция trimZeros(result).
12. Итоговый список result передаётся обратно в число a с помощью перемещения:  
    a = std::move(result);

В итоге получаем корректно вычисленную разность, которая находится в переменной a.

**4.3 Обработка знаков при сложении и вычитании**

Функции add и sub оперируют только абсолютными величинами чисел, не учитывая знаки. Вся логика обработки знаков реализована на уровне функции main.

**5. ФУНКЦИЯ MAIN И ВЗАИМОДЕЙСТВИЕ С ПОЛЬЗОВАТЕЛЕМ**

Функция main организует ввод-вывод и вызывает описанные выше алгоритмы для выполнения операций. Пользовательское взаимодействие в рамках курсовой работы реализовано через консольный интерфейс. Ниже описан общий сценарий работы программы в функции main:

1. Ввод данных и знаков. Программа запрашивает у пользователя две строки — первое и второе число, каждое может начинаться с “+” или “–“. После этого читается символ операции (“+” или “–“).
2. Выделение и учёт знаков. Если в начале строки обнаружен “+” или “–“, этот знак сохраняется в bool neg1 или neg2, а сам символ удаляется из строки.
3. Конвертация в DigitList. Очищенные от знака строки передаются в toDigits, возвращая два объекта DigitList без изменений остальной логики.
4. Коррекция второго знака при вычитании. При выборе операции ‘–’ флаг neg2 автоматически инвертируется — это позволяет свести вычитание к сложению с учётом ранее снятых знаков.
5. Выполнение вычисления.
   1. Если neg1 == neg2, вызывается add(a, b), результат выводится с общим префиксом (“Сумма = ” или “Разность = ”) и знаком (если neg1 == true).
   2. Если neg1 != neg2, сравниваются модули списков (absCompare(a, b)):

5.2.1) При равенстве выводится “0”.

5.2.2) Если |a| > |b|, вызывается sub(a, b), знак берётся из neg1.

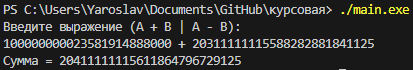
5.2.3) Иначе — sub(b, a), знак берётся из neg2.

1. Формирование и вывод строки результата. Полученный DigitList преобразуется в строку через toString выводится в консоль.
2. Завершение. После вывода программа завершается.

**6. ТЕСТИРОВАНИЕ И ПРИМЕРЫ ВВОДА/ВЫВОДА**

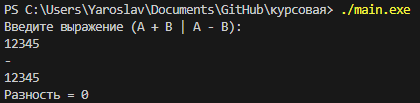
Разработанная программа протестирована на ряде случаев, покрывающих типичные и граничные сценарии. Ниже приведены примеры ручного тестирования с ожидаемым результатом:

1. Пример 1: сложение двух больших положительных чисел без смены знака разрядности.  
   Ввод: N1 = 100000000023581914888000, N2 = 20311111115588282881841125, операция “+”.

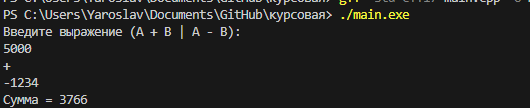


Вывод абсолютно корректный

1. Пример 2: вычитание обычных чисел с получением нулевого результата.  
   Ввод: N1 = 12345, N2 = 12345, операция “-“.

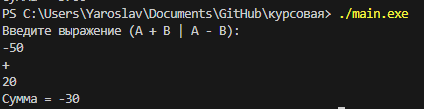


1. Пример 3: сложение чисел разных знаков (результат положительный).  
   Ввод: N1 = 5000, N2 = -1234, операция “+”.



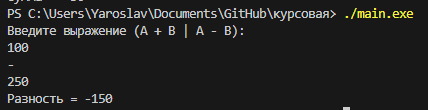
1. Пример 4: сложение обычных чисел разных знаков (результат отрицательный).

Ввод: N1 = -50, N2 = 20, операция “+”.



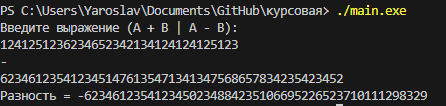
1. Пример 5: вычитание с отрицательным результатом.

Ввод: N1 = 100, N2 = 250, операция “-“.



1. Пример 6: Вычитание огромных чисел с одинаковыми знаками (отрицательный результат).

Ввод: N1 = 1241251236234652342134124124125123, N2 = 6234612354123451476135471341347568657834235423452, операция = “-“



Все перечисленные тесты показывают корректность работы алгоритмов – результаты совпадают с математически ожидаемыми. Дополнительно, стресс-тестирование на случайных больших числах (например, по 50-100 цифр каждая) подтвердило стабильность: программа справляется с их суммированием и разностью, выдавая правильный ответ. Ни переполнение, ни искажение данных не происходит.

Также отдельно проверялась корректность обработки знака: не возникает ситуации вывода “-0” или неверного присвоения знака нулю. Результат 0 всегда отображается без знака. Отрицательные результаты формируются только в тех случаях, когда второе число по модулю превышало первое при вычитании или соответствующая логика при сложении с разными знаками.

**8. КОД ПРОГРАММЫ НА C++**

#include <iostream>

#include <string>

#include <cctype>

#include <windows.h>

/\*─────────────────────────────────────────────────\*/

/\*   СТРУКТУРЫ ДАННЫХ                           \*/

/\*───────────────────────────────────────────────\*/

// Узел списка: одна цифра и указатель на следующий узел

struct DigitNode {

    int digit;              // 0...9

    DigitNode\* next = nullptr;

    explicit DigitNode(int d) : digit(d) {}

};

// Свой список цифр (LSB-порядок)

class DigitList {

public:

    DigitList() = default;

    ~DigitList() { clear(); }

    // добавить цифру в КОНЕЦ (то есть к самым старшим разрядам)

    void push\_back(int d) {

        DigitNode\* node = new DigitNode(d);

        if (!head) head = tail = node;

        else { tail->next = node; tail = node; }

        \_size++;

    }

    // последняя (самая старшая) цифра

    int back() const { return tail ? tail->digit : 0; }

    // удалить последнюю цифру

    void pop\_back() {

        if (!\_size) return;

        if (head == tail) { delete head; head = tail = nullptr; }

        else {

            DigitNode\* cur = head;

            while (cur->next != tail) cur = cur->next;

            delete tail;

            tail = cur;

            tail->next = nullptr;

        }

        --\_size;

    }

    std::size\_t size() const { return \_size; }

    DigitNode\* begin() const { return head; }

    void clear() {

        while (head) { DigitNode\* tmp = head; head = head->next; delete tmp; }

        tail = nullptr; \_size = 0;

    }

private:

    DigitNode\* head = nullptr;        // младший разряд

    DigitNode\* tail = nullptr;        // старший разряд

    std::size\_t \_size = 0;

};

/\*────────────────────────────────────────────────\*/

/\*   ВСПОМОГАТЕЛЬНЫЕ ФУНКЦИИ                    \*/

/\*────────────────────────────────────────────────\*/

// Удалить ведущие нули (они в КОНЦЕ списка)

void trimZeros(DigitList& num) {

    while (num.size() > 1 && num.back() == 0)

        num.pop\_back();

}

// Строка -> DigitList   (игнорируем не-цифры)

DigitList toDigits(const std::string& s) {

    DigitList num;

    for (auto it = s.rbegin(); it != s.rend(); ++it)

        if (std::isdigit(\*it)) num.push\_back(\*it - '0');

    if (num.size() == 0) num.push\_back(0);

    trimZeros(num);

    return num;

}

// DigitList -> строка

std::string toString(const DigitList& num) {

    std::string out;

    char buf[2]{};

    // проходим задом наперёд: сначала нужен tail

    std::string tmp;

    for (DigitNode\* p = num.begin(); p; p = p->next)

        tmp.push\_back(char('0' + p->digit));

    for (auto it = tmp.rbegin(); it != tmp.rend(); ++it) out.push\_back(\*it);

    return out;

}

// |a| ? |b|   (-1,0,+1)

int absCompare(const DigitList& a, const DigitList& b) {

    if (a.size() != b.size()) return a.size() < b.size() ? -1 : 1;

    /\* чтобы сравнить слева направо, перегоняем цифры во временный буфер \*/

    std::string sa = toString(a), sb = toString(b);

    if (sa == sb) return 0;

    return sa < sb ? -1 : 1;

}

/\*─────────────────────────────────────────────────\*/

/\*   АРИФМЕТИКА                                    \*/

/\*─────────────────────────────────────────────────\*/

// a = a + b   (оба неотрицательны)

void add(DigitList& a, const DigitList& b) {

    DigitNode\* pa = a.begin();

    DigitNode\* pb = b.begin();

    int carry = 0;

    DigitList result;

    while (pa || pb || carry) {

        int sum = carry;

        if (pa) { sum += pa->digit; pa = pa->next; }

        if (pb) { sum += pb->digit; pb = pb->next; }

        result.push\_back(sum % 10);

        carry = sum / 10;

    }

    a = std::move(result);

    trimZeros(a);         // заменить результатом

}

// a = a - b   (|a| >= |b|, оба >= 0)

void sub(DigitList& a, const DigitList& b) {

    DigitNode\* pa = a.begin();

    DigitNode\* pb = b.begin();

    int borrow = 0;

    DigitList result;

    while (pa) {

        int diff = pa->digit - borrow - (pb ? pb->digit : 0);

        if (diff < 0) { diff += 10; borrow = 1; } else borrow = 0;

        result.push\_back(diff);

        pa = pa->next;

        if (pb) pb = pb->next;

    }

    trimZeros(result);

    a = std::move(result);

}

/\*────────────────────────────────────────────────\*/

/\*   ДЕМО                                         \*/

/\*─────────────────────────────────────────────────\*/

int main() {

    SetConsoleOutputCP(65001);

    std::cout << "Введите выражение (A + B | A - B):\n";

    std::string s1, s2;

    char op;

    if (!(std::cin >> s1 >> op >> s2)) return 0;

    // вычленяем знаки

    bool neg1 = false, neg2 = false;

    if (s1[0]=='+'||s1[0]=='-') { neg1 = (s1[0]=='-'); s1.erase(0,1); }

    if (s2[0]=='+'||s2[0]=='-') { neg2 = (s2[0]=='-'); s2.erase(0,1); }

    // конвертируем в списки цифр

    DigitList a = toDigits(s1);

    DigitList b = toDigits(s2);

    // для операции '-' инвертируем знак второго

    if (op == '-') neg2 = !neg2;

    if (op!='+' && op!='-') {

        std::cerr << "Поддерживаются только + и -\n";

        return 1;

    }

    // общий префикс вывода

    const char\* label = (op=='+' ? "Сумма = " : "Разность = ");

    if (neg1 == neg2) {

        // одинаковые знаки -> сложение модулей

        add(a, b);

        std::cout << label;

        if (neg1) std::cout << '-';

        std::cout << toString(a) << '\n';

    } else {

        // разные знаки → вычитание модулей

        int cmp = absCompare(a, b);

        if (cmp == 0) {

            std::cout << label << "0\n";

        } else if (cmp > 0) {

            sub(a, b);

            std::cout << label;

            if (neg1) std::cout << '-';

            std::cout << toString(a) << '\n';

        } else {

            sub(b, a);

            std::cout << label;

            if (neg2) std::cout << '-';

            std::cout << toString(b) << '\n';

        }

    }

}

**7. АНАЛИЗ ОГРАНИЧЕНИЙ И ВОЗМОЖНЫЕ УЛУЧШЕНИЯ РЕАЛИЗАЦИИ**

Разработанная реализация успешно выполняет поставленные задачи, однако она обладает рядом ограничений, которые стоит учитывать при её дальнейшем использовании и развитии. Рассмотрим эти ограничения подробнее, а также предложим конкретные способы улучшения текущего решения.

**Ограничения текущей реализации:**

1. Использование односвязного списка для представления числа удобно с точки зрения простоты реализации алгоритмов, однако приводит к нескольким недостаткам:
2. Медленная операция удаления последнего узла: операция pop\_back требует прохода по всему списку, что ведёт к линейной сложности (O(n)) на каждое удаление, особенно ощутимо в функции trimZeros, где удаление множества ведущих нулей становится квадратичным (O(n\*n)).
3. **Избыточное потребление памяти**: каждый узел хранит одну цифру и указатель, что приводит к значительным накладным расходам памяти при работе с большими числами.
4. Низкая эффективность сравнения чисел. Функция сравнения по модулю (absCompare) преобразует числа в строки для последующего сравнения, что создаёт дополнительную нагрузку на память и снижает скорость работы программы.
5. Хранение цифр в десятичной системе. Представление чисел в виде отдельных десятичных цифр удобно для чтения и вывода, но существенно снижает производительность нашей программы при работе с большими числами, так как увеличивает количество необходимых операций для обработки каждого разряда числа.

**Возможные улучшения реализации:**

1. Использование двусвязного списка или динамического массива (вектора)  
   Замена односвязного списка на двусвязный список или динамический массив позволила бы выполнять операции удаления последнего элемента за константное время, существенно ускоряя процесс удаления ведущих нулей и другие манипуляции с концом списка.
2. Использование более крупной системы счисления кратно повысит нашу производительность при гигантских числах из-за принципа работы алгоритма (обращение к каждому разряду отдельно).
3. Улучшение обработки ввода и ошибок. В текущей реализации отсутствует полноценная проверка ввода данных пользователем на некорректные символы или пустые строки. Добавление таких проверок сделает программу более устойчивой к неправильному вводу и повысит её надёжность в реальных сценариях.

**ЗАКЛЮЧЕНИЕ**

В рамках выполненной курсовой работы была разработана и исследована программа, предназначенная для выполнения арифметических операций с целыми числами произвольной длины, превышающей стандартные аппаратные ограничения компьютеров. Реализованное решение позволяет корректно выполнять операции сложения и вычитания с числами практически неограниченного размера, подтверждая тем самым возможность преодоления естественных ограничений стандартных целочисленных типов в языке программирования C++.

В ходе работы был выбран подход, основанный на использовании динамической структуры данных — односвязного списка, хранящего десятичные цифры в порядке от младшего к старшему разряду. Это позволило максимально приблизить реализованные алгоритмы к естественным операциям, используемым при ручном вычислении «столбиком». Данный подход обеспечил прозрачность алгоритмов, простоту их понимания и реализации, а также продемонстрировал основные принципы построения систем длинной арифметики.

Разработанные функции сложения и вычитания полностью соответствуют поставленной задаче и корректно обрабатывают числа любых знаков и размеров. Особое внимание в работе уделено вопросам нормализации результата (удалению ведущих нулей), правильному учёту знаков чисел, а также обеспечению устойчивости работы программы при любых возможных комбинациях входных данных. Проведённые тесты и примеры показали надёжность и корректность получаемых результатов даже при работе с числами длиной в сотни и тысячи разрядов, что подтверждает как практическую применимость реализованных алгоритмов, так и высокую степень их соответствия заявленным требованиям.

Тем не менее, несмотря на успешное решение поставленных задач, текущая реализация обладает рядом ограничений. Прежде всего это касается производительности при работе с очень большими числами, избыточного потребления памяти и неэффективности некоторых операций, таких как удаление элементов из конца списка и сравнение чисел. Анализ этих ограничений позволил предложить конкретные улучшения — переход к более эффективным структурам данных (например, динамическому массиву или двусвязному списку), использование чисел с увеличенным основанием системы счисления (например, 10^9), оптимизацию функций сравнения, а также дальнейшее развитие структуры данных с инкапсуляцией знака числа и реализации дополнительных операций (умножения, деления, вычисления остатка).

Важным результатом работы является демонстрация фундаментальных принципов построения алгоритмов и структур данных, используемых в реальных системах длинной арифметики. Реализованные подходы являются важными основами программирования и отражают ключевые идеи, применяемые в практике алгоритмизации. После написания курсовой работы по реализации длинной арифметики, которая имеет широкую область применения, начиная от решения учебных задач, заканчивая практическим использованием в научных вычислениях, криптографических алгоритмах и финансовых расчётах, я стал лучше работать со структурами и больше понимаю поведение чисел в языках программирования со статической типизацией.

**СПИСОК ЛИТЕРАТУРЫ**

1. Knuth D. E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. – 3rd ed. – Addison-Wesley, 1998. – 760 p. (Раздел *Arithmetic* – алгоритмы работы с большими числами).
2. Stroustrup B. *The C++ Programming Language*. – 4th ed. – Addison-Wesley, 2013. – 1366 p. (Глава 24, примеры перегрузки операторов и работа с пользовательскими типами, в том числе для чисел большой разрядности).
3. Sedgewick R., Wayne K. *Algorithms*. – 4th ed. – Addison-Wesley, 2011. – 955 p. (Раздел, посвящённый вычислительным алгоритмам, содержит описание базовых подходов к реализации арифметических операций).
4. cp-algorithms.com - Arbitrary-Precision Arithmetic
5. e-maxx.ru – Длинная арифметика