МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра информационной безопасности

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Алгоритмы и структуры данных»

Тема: Экспериментальное определение средней пространственной сложности алгоритма

Вариант: операции над конечными неупорядоченными множествами

Студент гр. 4362 _____ Фрейдлин Р. Н.

Студент гр. 4362 _____ Турков Н. Д.

ТЧ	РА	ЭЧ	ИП	ТП	О/В

защищено ___ . ___ . _____

с _____ попытки

с оценкой _____

Преподаватель _____ Абросимов И.К.

Санкт-Петербург

2025

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Студенты Фрейдлин Р. Н., Турков Н. Д. группы 4362

Тема работы: экспериментальное определение средней пространственной сложности алгоритма.

Задание на лабораторную работу:

1) Требуется реализовать и отладить на языке C и C++ структуру данных и операции над ней согласно варианту;

2) Реализовать алгоритм операции над конечными неупорядоченными множествами с использованием битового массива;

3) Выполнить экспериментальное нахождение средней сложности реализации алгоритма операции над конечными неупорядоченными множествами.

Содержание пояснительной записки: разделы «СОДЕРЖАНИЕ», «ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ» с подразделами «Описание исследуемого алгоритма» и «Методика экспериментального исследования», «РЕШЕНИЕ ПОСТАВЛЕННОЙ ЗАДАЧИ» с подразделами «Реализация исследуемого алгоритма», «Реализация исследуемой структуры данных», «Тестирование реализации алгоритма», «Проведение эксперимента» и «Обработка результатов эксперимента», «ЗАКЛЮЧЕНИЕ», «СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ»

Предполагаемый объем пояснительной записки: не менее 40 страниц.

Дата выдачи задания: 20.10.2025

Дата сдачи пояснительной записки:

Задание получил	_____	Фрейдлин Р. Н.
Задание получил	_____	Турков Н. Д.
Задание выдал	_____	Абросимов И. К.

СОДЕРЖАНИЕ

1 ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ 4
1.1 Описание исследуемого алгоритма 4
1.2 Методика экспериментального исследования 5
2 РЕШЕНИЕ ПОСТАВЛЕННОЙ ЗАДАЧИ 5
2.1 Реализация исследуемого алгоритма 8
2.2 Тестирование реализации алгоритма 20
2.3 Проведение эксперимента 33
2.4 Обработка результатов эксперимента 35
ЗАКЛЮЧЕНИЕ 28
СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ 29
ПРИЛОЖЕНИЕ А. Используемые функции 30
ПРИЛОЖЕНИЕ Б. Разработанные функции 32
ПРИЛОЖЕНИЕ В. Исходный код разработанной программы 35

1 ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ

Цель работы: экспериментальное нахождение средней пространственной сложности алгоритма операции над конечными неупорядоченными множествами.

1.1 Описание исследуемого алгоритма

Алгоритмы операций над конечными неупорядоченными множествами — это алгоритмы, предназначенные для реализации основных теоретико-множественных операций: объединения, пересечения и разности. В рамках данной работы множества представляются с помощью структуры данных «битовый массив».

Для битовых массивов A и B размера n представляющих множества, операции определяются следующим образом:

Объединение (Union):

$$C_i = A_i \vee B_i \; \forall i \in [0, n-1] \tag{1.1}$$

Пересечение (Intersection):

$$C_i = A_i \wedge B_i \; \forall i \in [0, n-1] \tag{1.2}$$

Разность (Difference):

$$C_i = A_i \vee \neg B_i \; \forall i \in [0, n-1] \tag{1.3}$$

Где A_i, B_i, C_i — значение i-го бита массивов A, B, C соответственно.

Алгоритмы основываются на поэлементном (побитовом) применении соответствующих логических операций ко всем битам исходных массивов. Итеративная формула для выполнения операций над множествами, представленными битовыми массивами, имеет вид:

$$C_i = f(A_i, B_i), i \in [0, n-1] \tag{1.4}$$

где f – логическая функции, соответствующая конкретной операции.

Условие останова для итеративной формы алгоритма имеет вид:

$$i = n \Rightarrow C = A < op > B \quad (1.5)$$

где $< op >$ – одна из операций $\{\vee, \wedge, \setminus\}$.

Пространственные затраты алгоритма определяются объемом памяти, необходимой для хранения входных данных и промежуточных результатов выполнения операций. Для операций над множествами основными потребителями памяти являются:

1. Входные множества: два битовых массива A и B размера n
2. Выходное множество: один битовый массив C размера n

Пространственные затраты можно оценить как:

$$C_A^S(n) = O(n) \quad (1.6)$$

Пространственная сложность алгоритмов операций над множествами является линейной относительно размера массива n.

1.2 Методика экспериментального исследования

Определение средней пространственной сложности

В работе исследуется средняя пространственная сложность алгоритма – среднее количество пространственных затрат алгоритма A на множестве входных данных заданного размера n, обозначаемый как $\bar{S}_A(n)$

$$\bar{S}_A(n) = \frac{1}{|X|} \cdot \sum_{x \in X_n} C_A^S(x), \quad (1.6)$$

где $X_n = \{x | \|x\| = n\}$ – множество входных данных размера n, C_A^S – пространственные затраты алгоритма, равные количеству времени, которое затрачивается на выполнение алгоритма A при входных данных $x \in X_n$ размера $\|x\| = n$.

Как и для временной сложности, чаще всего определяют класс функций, к которому принадлежит $\bar{S}_A(n)$ при $n \to \infty$ поэтому ее описывают с использованием символов асимптотических оценок, таких как «о большое». Поэтому экспериментальное оценивание средней пространственной

сложности может быть проведено по той же методике, что и оценивание средней временной сложности, за исключением случая, когда средняя пространственная сложность описывается полиномиальной функцией, а средняя временная сложность – сверхполиномиальной (например, экспонентной или факториалом). В этом случае для определения функции. описывающей скорость роста пространственной сложности используется, с учетом, что $n_{i+1} = n_i +, m$ формула

$$n_i \cdot \frac{S_{i+1} - S_i}{S_i} = n_i \cdot \frac{C \cdot (n_{i+1}^a - n_i^a)}{C \cdot n_i^a} = \frac{(n_i + m)^a - n_i^a}{n_i^{a-1}} \xrightarrow[n_i \to \infty]{} am, \qquad (1.7)$$

Где S_i – используемая при работе алгоритма память при соответствующем размере входных данных.

Если данное отношение при всех $n_i, i \in \overline{1,5}$ примерно равно $a \cdot m$, то средняя пространственная сложность алгоритма, оценивается при помощи функции $O(n^a)$.

Подсчет количества памяти, которое потребляет программа, реализующая оцениваемый алгоритм, может быть произведен с помощью непосредственного подсчета с использованием дополнительных переменных, с использованием специальных функций *API* операционной систем, либо с использованием специального ПО, например, встроенного в среду разработки.

Рассмотрим способ непосредственного подсчета, причем учитывать будем только динамическую память. Определим две глобальные переменные: *currentMemory* – количество динамической памяти, которое потребляется в текущий момент времени и *maxMemory* – наибольшее количество динамической памяти, которое потреблялось в процессе программы и инициализируем их и нулевыми значениями.

Если в какой-то момент *maxMemory* $<$ *currentMemory*, то *maxMemory* присваивается значение *currentMemory*.

Перед каждым вызовом функции выделения динамической памяти *currentMemory* увеличивается на величину, равную размеру выделенного

блока памяти, а после каждого вызова функции освобождения динамической памяти – уменьшается на величину, равную размеру освобождаемого блока памяти.

2 РЕШЕНИЕ ПОСТАВЛЕННОЙ ЗАДАЧИ

2.1 Реализация исследуемой структуры данных

Для реализации исследуемого алгоритма была разработана структура данных битовый массив (`BitArray`), обеспечивающая выполнение основных операций над битовыми последовательностями произвольной длины.

Данная структура используется, так как стандартные типы данных языка C++ имеют ограниченную разрядность и не позволяют эффективно работать с битовыми последовательностями большой длины.

Структура имеет следующий вид:

```
class BitArray {
private:
    unsigned char* data;
    size_t bit_size;
    size_t byte_capacity;
}
```

Для данной структуры требуется реализовать работу восьми основных операций:

Установка отдельного бита.

`set_bit(size_t index, bool value)`

Для реализации функции установки бита применим итеративную формулу (2.1)

$$\begin{cases} byte = \lfloor index/8 \rfloor \\ pos = index \bmod 8 \\ data[byte] = \begin{cases} data[byte] \vee (1 \ll pos), value = true \\ data[byte] \wedge \neg(1 \ll pos), value = false \end{cases} \end{cases} \quad (2.1)$$

При написании окончательной версии функции установки отдельного бита используются следующие целочисленные переменные (Таблица 2.1).

Таблица 2.1 – Переменные, используемые для реализации функции

Старое название	Новое название	Описание
`byte`	`byte_inx`	Содержит индекс искомого байта
`pos`	`bit_off`	Содержит позицию бита в байте
`data`	`data`	Имя массива
`value`	`value`	Значения бита

Согласно схеме итераций, формуле 2.1 соответствует функция:

```
void BitArray::set_bit(size_t index, bool value) {
    validate_index(index);
    size_t byte_idx = byte_index(index);
    size_t bit_off = bit_offset(index);
    if (value) {
        data[byte_idx] |= (1 << bit_off);
    } else {
        data[byte_idx] &= ~(1 << bit_off);
    }
}
```

Установка группы подряд идущих бит.

```
set_bit_range(size_t start_index,
    size_t count, bool value)
```

Для реализации функции установки группы подряд идущих бит применим итеративную формулу (2.2)

$$\begin{cases} b_{start+i} = value \\ i_{i+1} = i_i + 1 \end{cases}, i \in [0, count - 1] \wedge (start + i) < n \qquad (2.2)$$

При написании окончательной версии функции установки группы подряд идущих бит используются целочисленные переменные (Таблица 2.2).

Таблица 2.2 – Переменные, используемые для реализации функции

Старое название	Новое название	Описание
`start`	`Start_index`	Начальный индекс диапазона
`count`	`bit_off`	Количество битов для установки
`n`	`bit_size`	Размер массива
`value`	`value`	Значения бита

Согласно схеме итераций, формуле 2.2 соответствует функция:

```
void BitArray::set_bit_range(size_t start_index,
    size_t count, bool value) {
    for (size_t i = 0; i < count; i++) {
        if (start_index + i < bit_size) {
            set_bit(start_index + i, value);
        }
    }
}
```

Инверсия бита.

`invert_single_bit(size_t index)`

Для реализации функции инверсии бита применим итеративную формулу (2.3)

$$\begin{cases} byte = \lfloor index/8 \rfloor \\ pos = index \bmod 8 \\ data[byte] = data[byte] \oplus (1 \ll pos) \end{cases} \quad (2.3)$$

При написании окончательной версии функции инверсии бита используются следующие целочисленные переменные (Таблица 2.3).

Таблица 2.3 – Переменные, используемые для реализации функции

Старое название	Новое название	Описание
`byte`	`byte_idx`	Содержит индекс искомого байта
`pos`	`bit_off`	Содержит позицию бита в байте
`data`	`data`	Имя массива
`value`	`value`	Значения бита

Согласно схеме итераций, формуле 2.3 соответствует функция:

```
void BitArray::invert_single_bit(size_t index) {
    validate_index(index);
    size_t byte_idx = byte_index(index);
    size_t bit_off = bit_offset(index);
    data[byte_idx] ^= (1 << bit_off);
}
```

Инверсия группы подряд идущих бит.

`invert_bit_range(size_t start_index,size_t count)`

Для реализации функции инверсии группы подряд идущих бит применим итеративную формулу (2.4)

$$\begin{cases} b_{start+i} = \neg b_{start+i}, i \in [0, count-1] \wedge (start+i) < n, \\ i_{i+1} = i_i + 1 \end{cases} \quad (2.4)$$

При написании окончательной версии функции инверсии группы подряд идущих бит используются следующие целочисленные переменные (Таблица 2.4).

Таблица 2.4 – Переменные, используемые для реализации функции

Старое название	Новое название	Описание
`i`	`i`	Индекс итерации
`start`	`start_index`	Начальный индекс диапазона битов
`b`	`Inver_single_bit()`	Содержит изменяемый бит
`count`	`count`	Количество битов для инверсии

Согласно схеме итераций, формуле 2.4 соответствует функция:

```
void BitArray::invert_bit_range(size_t start_index,
    size_t count) {
    for (size_t i = 0; i < count; i++) {
        if (start_index + i < bit_size) {
            invert_single_bit(start_index + i);
        }
    }
}
```

Логический сдвиг влево.

`logical_shift_left(size_t shift_count)`

Для реализации функции логического сдвига влево применим итеративную формулу (2.5)

$$\begin{cases} count > n: \begin{cases} b_i = 0, i \in [0, n-1] \\ i_{i+1} = i_i + 1 \end{cases} \\ count < n: \begin{cases} b_i = value, i \in [0, n - count - 1], value = b_{count+i} \\ b_i = 0, i \in [n - count, n-1] \\ i_{i+1} = i_i + 1 \end{cases} \end{cases} \quad (2.5)$$

При написании окончательной версии функции логического сдвига влево используются следующие целочисленные переменные (Таблица 2.5).

Таблица 2.5 – Переменные, используемые для реализации функции

Старое название	Новое название	Описание
i	i	Индекс итерации
start	Start_index	Начальный индекс диапазона битов
b	get_bit(i)	Содержит изменяемый бит
count	shift_count	Количество битов для сдвига
n	bit_size	Размер массива

Согласно схеме итераций, формуле 2.5 соответствует функция:

```
void BitArray::logical_shift_left(size_t
    shift_count) {
    if (shift_count == 0 || shift_count >= bit_size) {
        if (shift_count >= bit_size) {
            for (size_t i = 0; i < bit_size; i++) {
                set_bit(i, false);
            }
        }
        return;
    }
    for(size_t i = 0; i < bit_size - shift_count; i++){
        set_bit(i, get_bit(i + shift_count));
    }
    for (size_t i = bit_size - shift_count;
        i < bit_size; i++) {
        set_bit(i, false);
    }
}
```

Логический сдвиг вправо.

`logical_shift_right(size_t shift_count)`

Для реализации функции инверсии логического сдвига вправо применим итеративную формулу (2.6)

$$\begin{cases} count > n: \begin{cases} b_i = 0, i \in [0, n-1] \\ i_{i+1} = i_i + 1 \end{cases} \\ count < n: \begin{cases} b_i = value, i \in [n-1, count], value = b_{i-count} \\ b_i = 0, i \in [count-1, 0] \\ i_{i-1} = i_i - 1 \end{cases} \end{cases} \quad (2.6)$$

При написании окончательной версии функции логического сдвига вправо используются следующие целочисленные переменные (Таблица 2.6).

Таблица 2.6 – Переменные, используемые для реализации функции

Старое название	Новое название	Описание
`i`	`i`	Индекс итерации
`start`	`Start_index`	Начальный индекс диапазона битов
`b`	`get_bit(i)`	Содержит изменяемый бит
`count`	`count`	Количество битов для инверсии

Согласно схеме итераций, формуле 2.6 соответствует функция:

```
void BitArray::logical_shift_right(size_t shift_count)
{
    if (shift_count == 0 || shift_count >= bit_size) {
        if (shift_count >= bit_size) {
            for (size_t i = 0; i < bit_size; i++) {
                set_bit(i, false);
            }
        }
        return;
    }
```

```
    for(size_t i = bit_size - 1;i >= shift_count; i--){
        set_bit(i, get_bit(i - shift_count));
    }
    for(size_t i = 0; i < shift_count; i++) {
        set_bit(i, false);
    }
}
```

Преобразование битового массива в строку типа `char*`.

`to_char_string()`

Для реализации функции преобразование битового массива в строку типа `char*` применим итеративную формулу (2.7)

$$\begin{cases} str[i] = \begin{bmatrix} '1', b_i = true, \\ '0', b_i = false, \end{bmatrix} \ i \in [0, n-1], \\ i_{i+1} = i_i + 1 \end{cases} \quad (2.7)$$

При написании окончательной версии функции преобразование битового массива в строку типа `char*` используются следующие целочисленные переменные (Таблица 2.7).

Таблица 2.7 – Переменные, используемые для реализации функции

Старое название	Новое название	Описание
`i`	`i`	Индекс итерации
`b`	`get_bit(i)`	Содержит изменяемый бит
`str[i]`	`str[i]`	Элемент строки
`n`	`bit_size`	Размер массива

Согласно схеме итераций, формуле 2.7 соответствует функция:

```
char* BitArray::to_char_string() const {
    char* str = new char[bit_size + 1];
    for (size_t i = 0; i < bit_size; i++) {
        if (get_bit(i)) str[i] = '1';
```

```
        else str[i] = '0';
    }
    str[bit_size] = '\0';
    return str;
}
```

Преобразование строки типа `char*` в битовый массив.

`from_char_string(const char* str)`

Для реализации функции преобразование строки типа `char*` в битовый массив применим итеративную формулу (2.8)

$$\begin{cases} set\_bit\left(i, \begin{bmatrix} '1', b_i = true \\ '0', b_i = false \end{bmatrix}\right), i \in [0, len-1], \\ i_{i+1} = i_i + 1 \end{cases} \quad (2.8)$$

При написании окончательной версии функции преобразование строки типа `char*` в битовый массив используются следующие целочисленные переменные (Таблица 2.8).

Таблица 2.8 – Переменные, используемые для реализации функции

Старое название	Новое название	Описание
`i`	`i`	Индекс итерации
`len`	`strlen(str)`	Длина строки
`b`	`get_bit(i)`	Содержит изменяемый бит
`set_bit`	`set_bit(i, value)`	

Согласно схеме итераций, формуле 2.8 соответствует функция:

```
void BitArray::from_char_string(const char* str) {
    size_t len = strlen(str);
    if (data != nullptr) {
        delete[] data;
        data = nullptr;
    }
```

```
        resize_if_needed(len);
        for (size_t i = 0; i < len; i++) {
            if (str[i] == '1') set_bit(i, true);
            else if (str[i] == '0') set_bit(i, false);
            else throw std::invalid_argument("Invalid
                character in bit string");
        }
}
```

2.2 Реализация исследуемого алгоритма

Согласно варианту задания, необходимо реализовать алгоритмы операций над конечными неупорядоченными множествами, используя структуру данных битовый массив.

Для реализации функции объединения алгоритма применим итеративную формулу (2.9)

$$C_i = A_i \vee B_i, i \in [0, n-1] \quad (2.9)$$

При написании окончательной версии формулы объединения используются следующие целочисленные переменные (Таблица 2.9).

Таблица 2.9 – Переменные, используемые для реализации алгоритма

Старое название	Новое название	Описание
A_i	`data[i]`	Элемент первого массива
B_i	`other.data[i]`	Элемент второго массива
C_i	`result.data[i]`	Элемент получившегося массива
n	`bit_size`	Размер массива
i	`i`	Индекс итерации

Согласно схеме итераций, формуле 2.9 соответствует функция:

```
BitArray BitArray::union_with(const BitArray& other)
    const {
    if (bit_size != other.bit_size) {
        throw std::invalid_argument("Bit arrays must
            have the same size for union operation");
    }
    BitArray result(bit_size);
    size_t total_bytes = (bit_size + 7) / 8;
    for (size_t i = 0; i < total_bytes; i++) {
        result.data[i] = data[i] | other.data[i];
    }
    return result;
}
```

Для реализации функции пересечения алгоритма применим итеративную формулу (2.10)

$$C_i = A_i \wedge B_i, i \in [0, n-1] \quad (2.10)$$

При написании окончательной версии формулы пересечения используются следующие целочисленные переменные (Таблица 2.10).

Таблица 2.10 – Переменные, используемые для реализации алгоритма

Старое название	Новое название	Описание
A_i	`data[i]`	Элемент первого массива
B_i	`other.data[i]`	Элемент второго массива
C_i	`result.data[i]`	Элемент получившегося массива
n	`bit_size`	Размер массива
i	`i`	Индекс итерации

Согласно схеме итераций, формуле 2.10 соответствует функция:

```
BitArray BitArray::union_with(const BitArray& other)
    const {
    if (bit_size != other.bit_size) {
        throw std::invalid_argument("Bit arrays must
            have the same size for union operation");
    }
    BitArray result(bit_size);
    size_t total_bytes = (bit_size + 7) / 8;
    for (size_t i = 0; i < total_bytes; i++) {
        result.data[i] = data[i] & other.data[i];
    }
    return result;
}
```

Для реализации функции разности алгоритма применим итеративную формулу (2.11)

$$C_i = A_i \wedge \neg B_i, i \in [0, n-1] \quad (2.11)$$

При написании окончательной версии формулы разности используются следующие целочисленные переменные (Таблица 2.11).

Таблица 2.11 – Переменные, используемые для реализации алгоритма

Старое название	Новое название	Описание
A_i	`data[i]`	Элемент первого массива
B_i	`other.data[i]`	Элемент второго массива
C_i	`result.data[i]`	Элемент получившегося массива
n	`bit_size`	Размер массива
i	`i`	Индекс итерации

Согласно схеме итераций, формуле 2.11 соответствует функция:

```
BitArray BitArray::union_with(const BitArray& other)
    const {
    if (bit_size != other.bit_size) {
        throw std::invalid_argument("Bit arrays must
            have the same size for union operation");
    }
    BitArray result(bit_size);
    size_t total_bytes = (bit_size + 7) / 8;
    for (size_t i = 0; i < total_bytes; i++) {
        result.data[i] = data[i] & ~other.data[i];
    }
    return result;
}
```

2.3 Тестирование реализации алгоритма

Для тестирования функции `union, difference, intersection`, во-первых, требуется проверка граничных значений (Таблица 2.), при которых задействуются ветвления в `union, difference, intersection`.

Таблица 2.10 - Граничные значения для проверки ветвлений в функции `testBasicOperations()`

№	bit_size	bit	operation	result
1	0	-	set bit	0
2	1	[0]	set bit	1
3	1	[1]	set bit	1
4	8	[00000000]	union	11111111
		[11111111]		
4	8	[00000000]	intersect	00000000
		[11111111]		
5	8	[10101010]	union	00000000
		[01010101]		
5	8	[10101010]	intersect	11111111
		[01010101]		
6	8	[11110000]	union	11111111
		[00001111]		
7	8	[01010101]	union	01010101
		[01010101]		
7	8	[01010101]	difference	00000000
		[01010101]		
8	5 10		union	exception

Для этого разработана функция `manual_test_set_operations()`

```
void manual_test_set_operations() {
    std::cout << "\n=== Comprehensive Set Operations
        Testing ===" << std::endl;
    BitArray single1(1);
```

```
BitArray single2(1);
single1.set_single_bit(0, false);
single2.set_single_bit(0, false);
BitArray union_single =
    single1.union_with(single2);
char* str_single = union_single.to_char_string();
std::cout << "Test 1: Single bit (0) union (0) = '"
    << str_single << "'" << std::endl;
delete[] str_single;
BitArray single3(1);
BitArray single4(1);
single3.set_single_bit(0, true);
single4.set_single_bit(0, false);
BitArray union_single2 =
    single3.union_with(single4);
char* str_single2 = union_single2.to_char_string();
std::cout << "Test 2: Single bit (1) union (0) = '"
    << str_single2 << "'" << std::endl;
delete[] str_single2;
BitArray single5(1);
BitArray single6(1);
single5.set_single_bit(0, true);
single6.set_single_bit(0, true);
BitArray union_single3 =
    single5.union_with(single6);
char* str_single3 = union_single3.to_char_string();
std::cout << "Test 3: Single bit (1) union (1) = '"
    << str_single3 << "'" << std::endl;
delete[] str_single3;
BitArray zeros(8);
```

```
BitArray ones(8);
zeros.set_bit_range(0, 8, false);
ones.set_bit_range(0, 8, true);
BitArray union_zeros_ones = zeros.union_with(ones);
char* str_union =
    union_zeros_ones.to_char_string();
std::cout << "Test 4: Zeros union Ones = '" <<
    str_union << "'" << std::endl;
delete[] str_union;
BitArray intersect_zeros_ones =
    zeros.intersection_with(ones);
char* str_intersect =
    intersect_zeros_ones.to_char_string();
std::cout << "Test 4: Zeros intersect Ones = '" <<
    str_intersect << "'" << std::endl;
delete[] str_intersect;
BitArray alt1(8);
BitArray alt2(8);
for (size_t i = 0; i < 8; i++) {
    alt1.set_single_bit(i, (i % 2 == 0)); //
        10101010
    alt2.set_single_bit(i, (i % 2 == 1)); //
        01010101
}
BitArray union_alt = alt1.union_with(alt2);
char* str_union_alt = union_alt.to_char_string();
std::cout << "Test 5: 10101010 union 01010101 = '"
    << str_union_alt << "'" << std::endl;
delete[] str_union_alt;
```

```
BitArray intersect_alt =
    alt1.intersection_with(alt2);
char* str_intersect_alt =
    intersect_alt.to_char_string();
std::cout << "Test 5: 10101010 intersect 01010101 =
    '" << str_intersect_alt << "'" << std::endl;
delete[] str_intersect_alt;
BitArray part1(8);
BitArray part2(8);
part1.set_bit_range(0, 4, true);    // 11110000
part2.set_bit_range(4, 4, true);    // 00001111
BitArray union_part = part1.union_with(part2);
char* str_union_part = union_part.to_char_string();
std::cout << "Test 6: 11110000 union 00001111 = '"
    << str_union_part << "'" << std::endl;
delete[] str_union_part;
BitArray ident1(8);
BitArray ident2(8);
ident1.set_single_bit(1, true);
ident1.set_single_bit(3, true);
ident1.set_single_bit(5, true);
ident1.set_single_bit(7, true);
ident2.set_single_bit(1, true);
ident2.set_single_bit(3, true);
ident2.set_single_bit(5, true);
ident2.set_single_bit(7, true);
BitArray union_ident = ident1.union_with(ident2);
char* str_union_ident =
    union_ident.to_char_string();
```

```
    std::cout << "Test 7: Identical sets union = '" <<
        str_union_ident << "'" << std::endl;
    delete[] str_union_ident;
    BitArray diff_ident =
        ident1.difference_with(ident2);
    char* str_diff_ident = diff_ident.to_char_string();
    std::cout << "Test 7: Identical sets difference =
        '" << str_diff_ident << "'" << std::endl;
    delete[] str_diff_ident;
    BitArray diff_size1(5);
    BitArray diff_size2(10);
    try {
        BitArray invalid_union =
            diff_size1.union_with(diff_size2);
        std::cout << "Test 8: Different sizes -
            UNEXPECTED SUCCESS" << std::endl;
    } catch (const std::exception& e) {
        std::cout << "Test 8: Different sizes correctly
            threw excption: " << e.what() << std::endl;
    }
    std::cout << "=== End of Comprehensive Testing ==="
        << std::endl;
}
```

Результаты работы данной функции приведены ниже:

```
Test 1: Single bit (0) union (0) = '0'
Test 2: Single bit (1) union (0) = '1'
Test 3: Single bit (1) union (1) = '1'
Test 4: Zeros union Ones = '11111111'
Test 4: Zeros intersect Ones = '00000000'
Test 5: 10101010 union 01010101 = '11111111'
```

```
    Test 5: 10101010 intersect 01010101 = '00000000'
    Test 6: 11110000 union 00001111 = '11111111'
    Test 7: Identical sets union = '01010101'
    Test 7: Identical sets difference = '00000000'
    Test 8: Different sizes
```

Во-вторых, для автоматизированного тестирования функции `union()`, `intersection()`, `difference()`, на случайных входных данных использован генератор случайных чисел.

```
void automated_test_set_operations(int seed, int
    tests_count) {
    std::cout << "\n=== Automated Set Operations
        Testing ===" << std::endl;
    std::srand(seed);
    for (int t = 0; t < tests_count; t++) {
        size_t size = 10 + std::rand() % 91;
        BitArray set1(size);
        BitArray set2(size);
        BitArray set3(size);
        set1.random_fill();
        set2.random_fill();
        set3.random_fill();
        std::cout << "Automated test #" << t << ":
            bit_size=" << size << std::endl;
        BitArray union1 = set1.union_with(set2);
        BitArray union2 = set2.union_with(set1);
        bool comm_union = true;
        for (size_t i = 0; i < size; i++) {
            if(union1.get_bit(i) != union2.get_bit(i)){
                comm_union = false;
                break;
```

```
        }
    }
    if (comm_union) {
        std::cout << "  Test 1 (union
            commutativity): passed" << std::endl;
    } else {
        std::cout << "  Test 1 (union
            commutativity): failed" << std::endl;
    }
    BitArray inter1 = set1.intersection_with(set2);
    BitArray inter2 = set2.intersection_with(set1);
    bool comm_inter = true;
    for (size_t i = 0; i < size; i++) {
        if(inter1.get_bit(i) != inter2.get_bit(i)){
            comm_inter = false;
            break;
        }
    }
    if (comm_inter) {
        std::cout << "  Test 2 (intersection
            commutativity): passed" << std::endl;
    } else {
        std::cout << "  Test 2 (intersection
            commutativity): failed" << std::endl;
    }
    BitArray empty_set(size);
    BitArray union_empty =
        set1.union_with(empty_set);
    bool identity_union = true;
    for (size_t i = 0; i < size; i++) {
```

```
        if (set1.get_bit(i) !=
            union_empty.get_bit(i)) {
            identity_union = false;
            break;
        }
    }
    if (identity_union) {
        std::cout << "  Test 3 (union identity):
            passed" << std::endl;
    } else {
        std::cout << "  Test 3 (union identity):
            failed" << std::endl;
    }
    BitArray inter_empty =
        set1.intersection_with(empty_set);
    bool zero_inter = true;
    for (size_t i = 0; i < size; i++) {
        if (inter_empty.get_bit(i) != false) {
            zero_inter = false;
            break;
        }
    }
    if (zero_inter) {
        std::cout << "  Test 4 (intersection zero):
            passed" << std::endl;
    } else {
        std::cout << "  Test 4 (intersection zero):
            failed" << std::endl;
    }
```

```
BitArray self_diff =
    set1.difference_with(set1);
bool self_diff_empty = true;
for (size_t i = 0; i < size; i++) {
    if (self_diff.get_bit(i) != false) {
        self_diff_empty = false;
        break;
    }
}
if (self_diff_empty) {
    std::cout << "  Test 5 (self difference):
        passed" << std::endl;
} else {
    std::cout << "  Test 5 (self difference):
        failed" << std::endl;
}
BitArray union_left =
    set1.union_with(set2).union_with(set3);
BitArray union_right =
    set1.union_with(set2.union_with(set3));
bool assoc_union = true;
for (size_t i = 0; i < size; i++) {
    if (union_left.get_bit(i) !=
        union_right.get_bit(i)) {
        assoc_union = false;
        break;
    }
}
if (assoc_union) {
```

```
    std::cout << "  Test 6 (union
        associativity): passed" << std::endl;
} else {
    std::cout << "  Test 6 (union
        associativity): failed" << std::endl;
}
BitArray inter_left =
    set1.intersection_with(set2)
    .intersection_with(set3);
BitArray inter_right =
    set1.intersection_with
    (set2.intersection_with(set3));
bool assoc_inter = true;
for (size_t i = 0; i < size; i++) {
    if (inter_left.get_bit(i) !=
        inter_right.get_bit(i)) {
        assoc_inter = false;
        break;
    }
}
if (assoc_inter) {
    std::cout << "  Test 7 (intersection
        associativity): passed" << std::endl;
} else {
    std::cout << "  Test 7 (intersection
        associativity): failed" << std::endl;
}
BitArray left_dist = set1.intersection_with
    (set2.union_with(set3));
```

```
BitArray right_dist = set1.intersection_with
    (set2).union_with
    (set1.intersection_with(set3));
bool dist_law = true;
for (size_t i = 0; i < size; i++) {
    if (left_dist.get_bit(i) !=
        right_dist.get_bit(i)) {
        dist_law = false;
        break;
    }
}
if (dist_law) {
    std::cout << "  Test 8 (distribution law):
        passed" << std::endl;
} else {
    std::cout << "  Test 8 (distribution law):
        failed" << std::endl;
}
BitArray A_union_B = set1.union_with(set2);
A_union_B.invert_bit_range(0, size);
BitArray not_A = set1;
not_A.invert_bit_range(0, size);
BitArray not_B = set2;
not_B.invert_bit_range(0, size);
BitArray not_A_inter_not_B =
    not_A.intersection_with(not_B);
bool de_morgan = true;
for (size_t i = 0; i < size; i++) {
    if (A_union_B.get_bit(i) !=
not_A_inter_not_B.get_bit(i)) {
```

```
            de_morgan = false;
            break;
        }
    }
    if (de_morgan) {
        std::cout << "  Test 9 (De Morgan's law):
        passed" << std::endl;
    } else {
        std::cout << "  Test 9 (De Morgan's law):
        failed" << std::endl;
    }
    BitArray diff_AB = set1.difference_with(set2);
    BitArray not_B2 = set2;
    not_B2.invert_bit_range(0, size);
    BitArray A_inter_notB =
        set1.intersection_with(not_B2);
    bool diff_property = true;
    for (size_t i = 0; i < size; i++) {
        if (diff_AB.get_bit(i) !=
            A_inter_notB.get_bit(i)) {
            diff_property = false;
            break;
        }
    }
    if (diff_property) {
        std::cout << "  Test 10 (difference
            property): passed" << std::endl;
    } else {
        std::cout << "  Test 10 (difference
            property): failed" << std::endl;
```

```
        }
        std::cout << "  Sets generated successfully,
            all operations completed" << std::endl;
    }
}
```

Последний результат работы данной функции при `seed == 0`, `test_count == 1000` приведен ниже:

```
Automated test #11: bit_size=48
Test 1 (union commutativity): passed
Test 2 (intersection commutativity): passed
Test 3 (union identity): passed
Test 4 (intersection zero): passed
Test 5 (self difference): passed
Test 6 (union associativity): passed
Test 7 (intersection associativity): passed
Test 8 (distribution law): passed
Test 9 (De Morgan's law): passed
Test 10 (difference property): passed
```

Остальные 999 тестов также были пройдены успешно.

2.4 Проведение эксперимента

Для подсчета пространственной сложности алгоритмов операций над конечными неупорядоченными множествами создадим на основе функций `union()`, `intersection()`, `difference()` функцию `test_set_operation_experiment()`.

```
void test_set_operation_experiment () {
    cout << "EXPERIMENTAL DETERMINATION OF AVERAGE
        SPATIAL COMPLEXITY" << endl;
    vector<size_t> universe_sizes = {10000, 20000,
        40000, 80000, 160000, 320000, 640000};
```

```
vector<pair<size_t, double>> results;
cout << "n_i\t\tS(n_i)\t\tn_i+1 - n_i\tS_i+1 -
    S_i\tratio" << endl;
for (size_t i = 0; i < universe_sizes.size(); ++i){
    size_t n = universe_sizes[i];
    resetMemoryCounters();
    BitArray set1(n);
    BitArray set2(n);
    memoryAllocated(sizeof(BitArray) * 2);
    memoryAllocated((n + 7) / 8 * 2);
    set1.random_fill();
    set2.random_fill();
    BitArray union_set = set1.union_with(set2);
    BitArray intersection_set =
        set1.intersection_with(set2);
    BitArray difference_set =
        set1.difference_with(set2);
    memoryAllocated(sizeof(BitArray) * 3);
    memoryAllocated((n + 7) / 8 * 3);
    size_t memory_used = maxMemory;
    results.push_back({n, memory_used});
    if (i == 0) {
        cout << n << "\t" << memory_used << "\t"
             << "-\t\t" << "-\t\t" << "-" << endl;
    } else {
        size_t delta_n = universe_sizes[i] -
            universe_sizes[i-1];
        double delta_s = memory_used - results[i-
            1].second;
        double ratio = 0.0;
```

```
            if(results[i-1].second > 0 && delta_n > 0){
                ratio = (delta_s / results[i-1].second)
                    / (static_cast<double>(delta_n) /
                    universe_sizes[i-1]);
            }
            cout << n << "\t" << memory_used << "\t"
                 << delta_n << "\t\t" << delta_s <<
                     "\t\t" << ratio << endl;
        }
    }
}
```

Результат расчета временной сложности алгоритма представлен в консоли в следующем виде:

```
Size: 10000 bit, Space: 6370 byte
Size: 20000 bit, Space: 12620 byte
Size: 40000 bit, Space: 25120 byte
Size: 80000 bit, Space: 50120 byte
Size: 160000 bit, Space: 100120 byte
Size: 320000 bit, Space: 200120 byte
Size: 640000 bit, Space: 400120 byte
```

Перечень реализованных в процессе выполнения лабораторной работы функций приведен в разделе «ПРИЛОЖЕНИЕ Б. Разработанные функции», а исходный код разработанной программы – в разделе «ПРИЛОЖЕНИЕ В. Исходный код разработанной программы».

2.5 Обработка результатов эксперимента

Таблица 2.11 - Результаты экспериментов

Размер массива	Использованная память, байт
10000	6370
20000	12620
40000	25120
80000	50120
160000	100120
320000	200120
640000	400120

По экспериментальны данным (Таблица 2.11) построено отношение использованной памяти к количеству бит в массиве (Рисунок 2.1).

Таблица 2.12 – Отношение пространств

Пара измерений	s_{i+1}/s_i	$a_i = log_2(\frac{s_{i+1}}{s_i})$
10000-20000	1,981	0,986
20000-40000	1,990	0,993
40000-80000	1,995	0,996
80000-160000	1,998	0,998
160000-320000	1,999	0,999
320000-640000	1,999	0,999

Можно сделать вывод о том, что функция f удовлетворяет уравнению $f(2n) = 2f(n)$, откуда, с учетом того, что $f(n) = n^a$, следует, что $f(2n) = (2n)^a = 2^a n^a = 2n^a$ и $a = 1$, то есть алгоритм имеет линейную сложность.

Эталонным графиком временной сложности будет ломаная, построенная по точкам (n_i, y_i), где $i \in \overline{1,7}$, $y_{i+1} = 2y_i$, $y_1 = t_1$, экспериментальной кривой – ломаная, построенная по точкам (n_i, t_i).

На рисунке 2.1 – Представлены графики пространственной сложности алгоритма

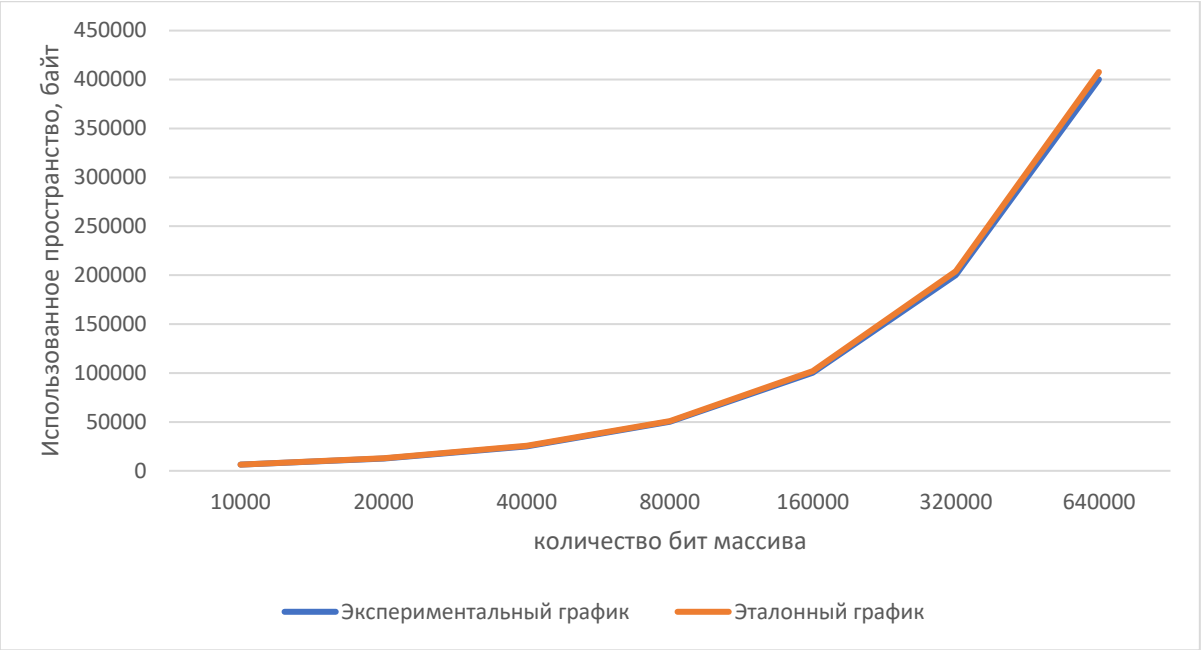

Рисунок 2.1 – Пространственная сложность алгоритма

ЗАКЛЮЧЕНИЕ

В результате проведения экспериментального исследования была построена зависимость пространства выполнения алгоритма операций над конечными неупорядоченными множествами от размера битового массива.

Эксперимент показал, что используемое пространство алгоритма возрастает линейно относительно количества битов в массиве. Была построена экспериментальная ломаная, отражающая реальные пространственные затраты программы, и эталонная линейная ломаная, построенная по соотношению $C(2n) = 2C(n)$.

Анализ графика показывает, что экспериментальная кривая имеет такой же характер роста, как и эталонная. При удвоении размера битового массива используемое пространство увеличивается в два раза, что соответствует линейному росту.

Следовательно, средняя пространственная сложность исследуемого алгоритма относится к классу линейных функций и характеризуется зависимость $O(n)$. Это объясняется тем, что алгоритм создает несколько битовых массивов фиксированного количества, каждый из которых имеет размер, пропорциональный n.

Таким образом, экспериментально определенная пространственная сложность $O(n)$ полностью соответствует теоретическим ожиданиям.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Абросимов И.К. АиСД 2025 ЛР 1 ПЗ.pdf – Яндекс Документы [Электронный ресурс] // Яндекс Диск: [сайт]. URL: disk.yandex.ru/i/wHghNib8Q_33Vw (дата обращения: 7.11.2025).
2. Абросимов И.К. АиСД 2025 02-03.pdf – Яндекс Документы [Электронный ресурс] // Яндекс Диск: [сайт]. URL: disk.yandex.ru/i/emuHbgfwxNa60w (дата обращения: 7.11.2025).

ПРИЛОЖЕНИЕ А. Используемые функции

Используются нижеприведенные операторы для типа `BitArray`:

Оператор	Назначение
`a++`	Увеличивает значение a на единицу
`a--`	Уменьшает значение a на единицу
`a > b`	Возвращает `true`, если значение a больше значения b, иначе возвращает `false`
`a < b`	Возвращает `true`, если значение a меньше значения b, иначе возвращает `false`
`a = b`	Присваивает значение числа b переменной a
`a >> b`	Возвращает число, полученное при побитовом сдвиге значения числа a вправо на значение числа b
`a & b`	Возвращает число, полученное при конъюнкции соответствующих битов значений a и b
`a / b`	Возвращает результат деления числа a на число b
`a % b`	Возвращает остаток от деления значения a на значение b для положительных значений a и b, знак результата для отрицательных значений a и b зависит от компилятора
`a != b`	Возвращает `true`, если значения a и b различны, либо `false`, если значения a и b равны
`a + b`	Возвращает сумму значений a и b
`a - b`	Возвращает разность значений a и b
`a * b`	Возвращает произведение двух чисел a и b
`!a`	Логическое отрицание: возвращает 1, если $a == 0$ и 0, если $a != 0$

`*a`	Оператор разыменования указателя a
`a | b`	Возвращает число, полученное при дизъюнкции соответствующих битов значений a и b

Используются нижеприведенные функции из заголовочного файла <cstring>.

`size_t strlen(const char* str)`	
Возвращает длину строки `str`	
`str`	строка, чья длина будет найдена

Используются нижеприведенные функции из заголовочного файла <cstddef>.

`size_t`
Беззнаковый целый тип для представления размеров

Используются нижеприведенные функции из заголовочного файла <vector>.

`vector<pair<size_t, double>> results`
Возвращает пару значений

ПРИЛОЖЕНИЕ Б. Разработанные функции

Функции, необходимые для реализации исследуемого алгоритма.

`BitArray(size_t size)`
Конструктор с заданным размером
Постусловие: создается битовый массив указанного размера, все биты установлены в 0

`BitArray(const char* str)`
Конструктор из строки
Постусловие: создается битовый массив на основе строки символов '0' и '1'

`~BitArray()`
Деструктор
Постусловие: освобождается вся динамически выделенная память

`size_t size() const`
Получение размера массива
Постусловие: возвращает количество битов в массиве

`bool get_bit(size_t index) const`
Чтение значения бита
Возвращает значение бита по указанному индексу (0 или 1)

`void set_bit(size_t index, bool value)`
Установка значения бита
Постусловие: бит по указанному индексу устанавливается в заданное значение

`void set_single_bit(size_t index, bool value)`
Установка одиночного бита
Постусловие: бит по индексу устанавливается в значение value

`void set_bit_range(size_t start_index, size_t count, bool value)`
Установка диапазона битов
Постусловие: Биты в диапазоне заданном диапазоне устанавливаются в value

`void random_fill()`
Случайное заполнение
Постусловие: все биты массива устанавливаются в случайные значения 0 или 1

`size_t union_with(const BitArray& other) const;`
Объединение массивов
Постусловие: возвращает объединение массивов

`size_t intersection_with(const BitArray& other) const;`
Пересечение массивов
Постусловие: возвращает пересечение массивов

`size_t difference_with(const BitArray& other) const;`
Разность массивов
Постусловие: возвращает разность массивов

Функции, необходимые для тестирования исследуемого алгоритма.

`void automated_test_set_operation(int seed, int tests_count)`
Выполняет автоматизированное тестирование функций `union_with()`, `intersection_with()`, `difference_with()` на случайных данных
Постусловие: нет

`void manual_test_set_operation()`
Выполняет ручное тестирование функции функций `union_with()`, `intersection_with()`, `difference_with()` на заранее известных тестах.
Постусловие: нет

Функции, необходимые для исследования алгоритма.

`void test_set_operation_experiment()`
Реализует проведение эксперимента по измерению времени выполнения функций `union_with()`, `intersection_with()`, `difference_with()` при увеличении длинных массива.
Постусловие: нет
`void memoryAllocated()`
Реализует вычисление выделенной памяти для алгоритма
Постусловие: нет

`void memoryFreed()`
Реализует вычисление освобожденной памяти для алгоритма
Постусловие: нет

`void resetMemoryCounters()`
Сбрасывает значения максимальной и актуально выделеной памяти
Постусловие: нет

ПРИЛОЖЕНИЕ В. Исходный код разработанной программы

Файл main.cpp

```
#include "bitArray.h"
#include <iostream>
#include <vector>
#include <random>
#include <cmath>
#include <cstring>
#include <map>
#include <algorithm>
using namespace std;
size_t currentMemory = 0;
size_t maxMemory = 0;
void memoryAllocated(size_t size) {
    currentMemory += size;
    if (maxMemory < currentMemory) {
        maxMemory = currentMemory;
    }
}
void memoryFreed(size_t size) {
    if (currentMemory >= size) {
        currentMemory -= size;
    }
}
void resetMemoryCounters() {
    currentMemory = 0;
    maxMemory = 0;
}
BitArray* createBitArray(size_t size) {
```

```
    memoryAllocated(sizeof(BitArray));
    memoryAllocated((size + 7) / 8);
    return new BitArray(size);
}
void deleteBitArray(BitArray* bitArray, size_t size) {
    if (bitArray) {
        memoryFreed(sizeof(BitArray));
        memoryFreed((size + 7) / 8);
        delete bitArray;
    }
}
size_t performSetOperations(size_t universe_size) {
    resetMemoryCounters();
    BitArray* set1 = createBitArray(universe_size);
    BitArray* set2 = createBitArray(universe_size);
    set1->random_fill();
    set2->random_fill();
    // Perform set operations
    BitArray union_set = set1->union_with(*set2);
    BitArray intersection_set = set1-
        >intersection_with(*set2);
    BitArray difference_set = set1-
        >difference_with(*set2);
    BitArray difference_set2 = set2-
        >difference_with(*set1);
    memoryAllocated(sizeof(BitArray) * 4);
    memoryAllocated((universe_size + 7) / 8 * 4);
    BitArray temp1 =
        union_set.intersection_with(intersection_set);
```

```
    BitArray temp2 =
        difference_set.union_with(difference_set2);
    BitArray final_result =
        temp1.difference_with(temp2);
    memoryAllocated(sizeof(BitArray) * 3);
    memoryAllocated((universe_size + 7) / 8 * 3);
    deleteBitArray(set1, universe_size);
    deleteBitArray(set2, universe_size);
    memoryFreed(sizeof(BitArray) * 4);
    memoryFreed((universe_size + 7) / 8 * 4);
    memoryFreed(sizeof(BitArray) * 3);
    memoryFreed((universe_size + 7) / 8 * 3);
    return maxMemory;
}

void test_set_operation_experiment () {
    cout << "EXPERIMENTAL DETERMINATION OF AVERAGE
        SPATIAL COMPLEXITY" << endl;
    vector<size_t> universe_sizes = {10000, 20000,
        40000, 80000, 160000, 320000, 640000};
    vector<pair<size_t, double>> results;
    cout << "n_i\t\tS(n_i)\t\tn_i+1 - n_i\tS_i+1 -
        S_i\tratio" << endl;
    for (size_t i = 0; i < universe_sizes.size(); ++i){
        size_t n = universe_sizes[i];
        resetMemoryCounters();
        BitArray set1(n);
        BitArray set2(n);
        memoryAllocated(sizeof(BitArray) * 2);
        memoryAllocated((n + 7) / 8 * 2);
        set1.random_fill();
```

```
            set2.random_fill();
            BitArray union_set = set1.union_with(set2);
            BitArray intersection_set =
                set1.intersection_with(set2);
            BitArray difference_set =
                set1.difference_with(set2);
            memoryAllocated(sizeof(BitArray) * 3);
            memoryAllocated((n + 7) / 8 * 3);
            size_t memory_used = maxMemory;
            results.push_back({n, memory_used});
            if (i == 0) {
                cout << n << "\t" << memory_used << "\t"
                     << "-\t\t" << "-\t\t" << "-" << endl;
            } else {
                size_t delta_n = universe_sizes[i] -
                    universe_sizes[i-1];
                double delta_s = memory_used - results[i-
                    1].second;
                double ratio = 0.0;
                if(results[i-1].second > 0 && delta_n > 0){
                    ratio = (delta_s / results[i-1].second)
                        / (static_cast<double>(delta_n) /
                        universe_sizes[i-1]);
                }
                cout << n << "\t" << memory_used << "\t"
                     << delta_n << "\t\t" << delta_s <<
                         "\t\t" << ratio << endl;
            }
        }
    }
```

```
void automated_test_set_operations(int seed, int
    tests_count) {
    std::cout << "\n=== Automated Set Operations
        Testing ===" << std::endl;
    std::srand(seed);
    for (int t = 0; t < tests_count; t++) {
        size_t size = 10 + std::rand() % 91;
        BitArray set1(size);
        BitArray set2(size);
        BitArray set3(size);
        set1.random_fill();
        set2.random_fill();
        set3.random_fill();
        std::cout << "Automated test #" << t << ":
            bit_size=" << size << std::endl;
        BitArray union1 = set1.union_with(set2);
        BitArray union2 = set2.union_with(set1);
        bool comm_union = true;
        for (size_t i = 0; i < size; i++) {
            if(union1.get_bit(i) != union2.get_bit(i)){
                comm_union = false;
                break;
            }
        }
        if (comm_union) {
            std::cout << "  Test 1 (union
                commutativity): passed" << std::endl;
        } else {
            std::cout << "  Test 1 (union
                commutativity): failed" << std::endl;
```

```
}
BitArray inter1 = set1.intersection_with(set2);
BitArray inter2 = set2.intersection_with(set1);
bool comm_inter = true;
for (size_t i = 0; i < size; i++) {
    if(inter1.get_bit(i) != inter2.get_bit(i)){
        comm_inter = false;
        break;
    }
}
if (comm_inter) {
    std::cout << "  Test 2 (intersection
        commutativity): passed" << std::endl;
} else {
    std::cout << "  Test 2 (intersection
        commutativity): failed" << std::endl;
}
BitArray empty_set(size);
BitArray union_empty =
    set1.union_with(empty_set);
bool identity_union = true;
for (size_t i = 0; i < size; i++) {
    if (set1.get_bit(i) !=
        union_empty.get_bit(i)) {
        identity_union = false;
        break;
    }
}
if (identity_union) {
```

```
    std::cout << "  Test 3 (union identity):
        passed" << std::endl;
} else {
    std::cout << "  Test 3 (union identity):
        failed" << std::endl;
}
BitArray inter_empty =
    set1.intersection_with(empty_set);
bool zero_inter = true;
for (size_t i = 0; i < size; i++) {
    if (inter_empty.get_bit(i) != false) {
        zero_inter = false;
        break;
    }
}
if (zero_inter) {
    std::cout << "  Test 4 (intersection zero):
        passed" << std::endl;
} else {
    std::cout << "  Test 4 (intersection zero):
        failed" << std::endl;
}
BitArray self_diff =
    set1.difference_with(set1);
bool self_diff_empty = true;
for (size_t i = 0; i < size; i++) {
    if (self_diff.get_bit(i) != false) {
        self_diff_empty = false;
        break;
    }
```

```
}
if (self_diff_empty) {
    std::cout << "  Test 5 (self difference):
        passed" << std::endl;
} else {
    std::cout << "  Test 5 (self difference):
        failed" << std::endl;
}
BitArray union_left =
    set1.union_with(set2).union_with(set3);
BitArray union_right =
    set1.union_with(set2.union_with(set3));
bool assoc_union = true;
for (size_t i = 0; i < size; i++) {
    if (union_left.get_bit(i) !=
        union_right.get_bit(i)) {
        assoc_union = false;
        break;
    }
}
if (assoc_union) {
    std::cout << "  Test 6 (union
        associativity): passed" << std::endl;
} else {
    std::cout << "  Test 6 (union
        associativity): failed" << std::endl;
}
BitArray inter_left =
    set1.intersection_with(set2)
    .intersection_with(set3);
```

```
BitArray inter_right =
    set1.intersection_with
    (set2.intersection_with(set3));
bool assoc_inter = true;
for (size_t i = 0; i < size; i++) {
    if (inter_left.get_bit(i) !=
        inter_right.get_bit(i)) {
        assoc_inter = false;
        break;
    }
}
if (assoc_inter) {
    std::cout << "  Test 7 (intersection
        associativity): passed" << std::endl;
} else {
    std::cout << "  Test 7 (intersection
        associativity): failed" << std::endl;
}
BitArray left_dist = set1.intersection_with
    (set2.union_with(set3));
BitArray right_dist = set1.intersection_with
    (set2).union_with
    (set1.intersection_with(set3));
bool dist_law = true;
for (size_t i = 0; i < size; i++) {
    if (left_dist.get_bit(i) !=
        right_dist.get_bit(i)) {
        dist_law = false;
        break;
    }
```

```
}
if (dist_law) {
    std::cout << "  Test 8 (distribution law):
        passed" << std::endl;
} else {
    std::cout << "  Test 8 (distribution law):
        failed" << std::endl;
}
BitArray A_union_B = set1.union_with(set2);
A_union_B.invert_bit_range(0, size);
BitArray not_A = set1;
not_A.invert_bit_range(0, size);
BitArray not_B = set2;
not_B.invert_bit_range(0, size);
BitArray not_A_inter_not_B =
    not_A.intersection_with(not_B);
bool de_morgan = true;
for (size_t i = 0; i < size; i++) {
    if (A_union_B.get_bit(i) !=
not_A_inter_not_B.get_bit(i)) {
        de_morgan = false;
        break;
    }
}
if (de_morgan) {
    std::cout << "  Test 9 (De Morgan's law):
    passed" << std::endl;
} else {
    std::cout << "  Test 9 (De Morgan's law):
    failed" << std::endl;
```

```
        }
        BitArray diff_AB = set1.difference_with(set2);
        BitArray not_B2 = set2;
        not_B2.invert_bit_range(0, size);
        BitArray A_inter_notB =
            set1.intersection_with(not_B2);
        bool diff_property = true;
        for (size_t i = 0; i < size; i++) {
            if (diff_AB.get_bit(i) !=
                A_inter_notB.get_bit(i)) {
                diff_property = false;
                break;
            }
        }
        if (diff_property) {
            std::cout << "  Test 10 (difference
                property): passed" << std::endl;
        } else {
            std::cout << "  Test 10 (difference
                property): failed" << std::endl;
        }
        std::cout << "  Sets generated successfully,
            all operations completed" << std::endl;
    }
}

void testBasicOperations() {
    cout << "\n=== BASIC OPERATIONS TEST ===" << endl;
    resetMemoryCounters();
    BitArray A("10101010");
```

```
    BitArray B("01010101");
    cout << "Set A: ";
    A.print();
    cout << "Set B: ";
    B.print();
    BitArray union_set = A.union_with(B);
    cout << "A ∪ B: ";
    union_set.print();
    BitArray intersection_set = A.intersection_with(B);
    cout << "A ∩ B: ";
    intersection_set.print();
    BitArray difference_set = A.difference_with(B);
    cout << "A \\ B: ";
    difference_set.print();
    cout << "Maximum memory used: " << maxMemory << "
bytes" << endl;
}
void manual_test_set_operations() {
    std::cout << "\n=== Comprehensive Set Operations
        Testing ===" << std::endl;
    BitArray single1(1);
    BitArray single2(1);
    single1.set_single_bit(0, false);
    single2.set_single_bit(0, false);
    BitArray union_single =
        single1.union_with(single2);
    char* str_single = union_single.to_char_string();
    std::cout << "Test 1: Single bit (0) union (0) = '"
        << str_single << "'" << std::endl;
    delete[] str_single;
```

```
BitArray single3(1);
BitArray single4(1);
single3.set_single_bit(0, true);
single4.set_single_bit(0, false);
BitArray union_single2 =
    single3.union_with(single4);
char* str_single2 = union_single2.to_char_string();
std::cout << "Test 2: Single bit (1) union (0) = '"
    << str_single2 << "'" << std::endl;
delete[] str_single2;
BitArray single5(1);
BitArray single6(1);
single5.set_single_bit(0, true);
single6.set_single_bit(0, true);
BitArray union_single3 =
    single5.union_with(single6);
char* str_single3 = union_single3.to_char_string();
std::cout << "Test 3: Single bit (1) union (1) = '"
    << str_single3 << "'" << std::endl;
delete[] str_single3;
BitArray zeros(8);
BitArray ones(8);
zeros.set_bit_range(0, 8, false);
ones.set_bit_range(0, 8, true);
BitArray union_zeros_ones = zeros.union_with(ones);
char* str_union =
    union_zeros_ones.to_char_string();
std::cout << "Test 4: Zeros union Ones = '" <<
    str_union << "'" << std::endl;
delete[] str_union;
```

```
BitArray intersect_zeros_ones =
    zeros.intersection_with(ones);
char* str_intersect =
    intersect_zeros_ones.to_char_string();
std::cout << "Test 4: Zeros intersect Ones = '" <<
    str_intersect << "'" << std::endl;
delete[] str_intersect;
BitArray alt1(8);
BitArray alt2(8);
for (size_t i = 0; i < 8; i++) {
    alt1.set_single_bit(i, (i % 2 == 0));
    alt2.set_single_bit(i, (i % 2 == 1));
}
BitArray union_alt = alt1.union_with(alt2);
char* str_union_alt = union_alt.to_char_string();
std::cout << "Test 5: 10101010 union 01010101 = '"
    << str_union_alt << "'" << std::endl;
delete[] str_union_alt;
BitArray intersect_alt =
    alt1.intersection_with(alt2);
char* str_intersect_alt =
    intersect_alt.to_char_string();
std::cout << "Test 5: 10101010 intersect 01010101 =
    '" << str_intersect_alt << "'" << std::endl;
delete[] str_intersect_alt;
BitArray part1(8);
BitArray part2(8);
part1.set_bit_range(0, 4, true);
part2.set_bit_range(4, 4, true);    // 00001111
BitArray union_part = part1.union_with(part2);
```

```
char* str_union_part = union_part.to_char_string();
std::cout << "Test 6: 11110000 union 00001111 = '"
    << str_union_part << "'" << std::endl;
delete[] str_union_part;
BitArray ident1(8);
BitArray ident2(8);
ident1.set_single_bit(1, true);
ident1.set_single_bit(3, true);
ident1.set_single_bit(5, true);
ident1.set_single_bit(7, true);
ident2.set_single_bit(1, true);
ident2.set_single_bit(3, true);
ident2.set_single_bit(5, true);
ident2.set_single_bit(7, true);
BitArray union_ident = ident1.union_with(ident2);
char* str_union_ident =
    union_ident.to_char_string();
std::cout << "Test 7: Identical sets union = '" <<
    str_union_ident << "'" << std::endl;
delete[] str_union_ident;
BitArray diff_ident =
    ident1.difference_with(ident2);
char* str_diff_ident = diff_ident.to_char_string();
std::cout << "Test 7: Identical sets difference =
    '" << str_diff_ident << "'" << std::endl;
delete[] str_diff_ident;
BitArray diff_size1(5);
BitArray diff_size2(10);
try {
```

```
        BitArray invalid_union =
            diff_size1.union_with(diff_size2);
        std::cout << "Test 8: Different sizes -
            UNEXPECTED SUCCESS" << std::endl;
    } catch (const std::exception& e) {
        std::cout << "Test 8: Different sizes correctly
            threw excption: " << e.what() << std::endl;
    }
    std::cout << "=== End of Comprehensive Testing ==="
        << std::endl;
}
void automated_test_set_operations(int seed, int
    tests_count) {
    std::cout << "\n=== Automated Set Operations
        Testing ===" << std::endl;
    std::srand(seed);
    for (int t = 0; t < tests_count; t++) {
        size_t size = 10 + std::rand() % 91;
        BitArray set1(size);
        BitArray set2(size);
        BitArray set3(size);
        set1.random_fill();
        set2.random_fill();
        set3.random_fill();
        std::cout << "Automated test #" << t << ":
            bit_size=" << size << std::endl;
        BitArray union1 = set1.union_with(set2);
        BitArray union2 = set2.union_with(set1);
        bool comm_union = true;
        for (size_t i = 0; i < size; i++) {
```

```
        if(union1.get_bit(i) != union2.get_bit(i)){
            comm_union = false;
            break;
        }
    }
    if (comm_union) {
        std::cout << "  Test 1 (union
            commutativity): passed" << std::endl;
    } else {
        std::cout << "  Test 1 (union
            commutativity): failed" << std::endl;
    }
    BitArray inter1 = set1.intersection_with(set2);
    BitArray inter2 = set2.intersection_with(set1);
    bool comm_inter = true;
    for (size_t i = 0; i < size; i++) {
        if(inter1.get_bit(i) != inter2.get_bit(i)){
            comm_inter = false;
            break;
        }
    }
    if (comm_inter) {
        std::cout << "  Test 2 (intersection
            commutativity): passed" << std::endl;
    } else {
        std::cout << "  Test 2 (intersection
            commutativity): failed" << std::endl;
    }
    BitArray empty_set(size);
```

```
BitArray union_empty =
    set1.union_with(empty_set);
bool identity_union = true;
for (size_t i = 0; i < size; i++) {
    if (set1.get_bit(i) !=
        union_empty.get_bit(i)) {
        identity_union = false;
        break;
    }
}
if (identity_union) {
    std::cout << "  Test 3 (union identity):
        passed" << std::endl;
} else {
    std::cout << "  Test 3 (union identity):
        failed" << std::endl;
}
BitArray inter_empty =
    set1.intersection_with(empty_set);
bool zero_inter = true;
for (size_t i = 0; i < size; i++) {
    if (inter_empty.get_bit(i) != false) {
        zero_inter = false;
        break;
    }
}
if (zero_inter) {
    std::cout << "  Test 4 (intersection zero):
        passed" << std::endl;
} else {
```

```
    std::cout << "  Test 4 (intersection zero):
        failed" << std::endl;
}
BitArray self_diff =
    set1.difference_with(set1);
bool self_diff_empty = true;
for (size_t i = 0; i < size; i++) {
    if (self_diff.get_bit(i) != false) {
        self_diff_empty = false;
        break;
    }
}
if (self_diff_empty) {
    std::cout << "  Test 5 (self difference):
        passed" << std::endl;
} else {
    std::cout << "  Test 5 (self difference):
        failed" << std::endl;
}
BitArray union_left =
    set1.union_with(set2).union_with(set3);
BitArray union_right =
    set1.union_with(set2.union_with(set3));
bool assoc_union = true;
for (size_t i = 0; i < size; i++) {
    if (union_left.get_bit(i) !=
        union_right.get_bit(i)) {
        assoc_union = false;
        break;
    }
```

```
}
if (assoc_union) {
    std::cout << "  Test 6 (union
        associativity): passed" << std::endl;
} else {
    std::cout << "  Test 6 (union
        associativity): failed" << std::endl;
}
BitArray inter_left =
    set1.intersection_with(set2)
    .intersection_with(set3);
BitArray inter_right =
    set1.intersection_with
    (set2.intersection_with(set3));
bool assoc_inter = true;
for (size_t i = 0; i < size; i++) {
    if (inter_left.get_bit(i) !=
        inter_right.get_bit(i)) {
        assoc_inter = false;
        break;
    }
}
if (assoc_inter) {
    std::cout << "  Test 7 (intersection
        associativity): passed" << std::endl;
} else {
    std::cout << "  Test 7 (intersection
        associativity): failed" << std::endl;
}
```

```
BitArray left_dist = set1.intersection_with
    (set2.union_with(set3));
BitArray right_dist = set1.intersection_with
    (set2).union_with
    (set1.intersection_with(set3));
bool dist_law = true;
for (size_t i = 0; i < size; i++) {
    if (left_dist.get_bit(i) !=
        right_dist.get_bit(i)) {
        dist_law = false;
        break;
    }
}
if (dist_law) {
    std::cout << "  Test 8 (distribution law):
        passed" << std::endl;
} else {
    std::cout << "  Test 8 (distribution law):
        failed" << std::endl;
}
BitArray A_union_B = set1.union_with(set2);
A_union_B.invert_bit_range(0, size);
BitArray not_A = set1;
not_A.invert_bit_range(0, size);
BitArray not_B = set2;
not_B.invert_bit_range(0, size);
BitArray not_A_inter_not_B =
    not_A.intersection_with(not_B);
bool de_morgan = true;
for (size_t i = 0; i < size; i++) {
```

```
    if (A_union_B.get_bit(i) !=
not_A_inter_not_B.get_bit(i)) {
        de_morgan = false;
        break;
    }
}
if (de_morgan) {
    std::cout << "  Test 9 (De Morgan's law):
    passed" << std::endl;
} else {
    std::cout << "  Test 9 (De Morgan's law):
    failed" << std::endl;
}
BitArray diff_AB = set1.difference_with(set2);
BitArray not_B2 = set2;
not_B2.invert_bit_range(0, size);
BitArray A_inter_notB =
    set1.intersection_with(not_B2);
bool diff_property = true;
for (size_t i = 0; i < size; i++) {
    if (diff_AB.get_bit(i) !=
        A_inter_notB.get_bit(i)) {
        diff_property = false;
        break;
    }
}
if (diff_property) {
    std::cout << "  Test 10 (difference
        property): passed" << std::endl;
} else {
```

```
            std::cout << "  Test 10 (difference
                property): failed" << std::endl;
        }
        std::cout << "  Sets generated successfully,
            all operations completed" << std::endl;
    }
}
int main() {
    test_set_operation_experiment();
    return 0;
}
```

Файл BitArray.cpp

```
#include "bitArray.h"
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <stdexcept>
#include <random>
void BitArray::validate_index(size_t index) const {
    if (index >= bit_size) {
        throw std::out_of_range("Bit index out of
            range");
    }
}
size_t BitArray::byte_index(size_t bit_index) const {
    return bit_index / 8;
}
size_t BitArray::bit_offset(size_t bit_index) const {
    return bit_index % 8;
}
```

```
void BitArray::resize_if_needed(size_t new_bit_size) {
    size_t required_bytes = (new_bit_size + 7) / 8;
    if (required_bytes > byte_capacity) {
        size_t new_capacity = required_bytes * 2;
        unsigned char* new_data = new unsigned
            char[new_capacity];
        if (data != nullptr) {
            memcpy(new_data, data, byte_capacity);
            memset(new_data + byte_capacity, 0,
                new_capacity - byte_capacity);
            delete[] data;
        } else {
            memset(new_data, 0, new_capacity);
        }

        data = new_data;
        byte_capacity = new_capacity;
    }
    bit_size = new_bit_size;
}
BitArray::BitArray() : data(nullptr), bit_size(0),
    byte_capacity(0) {}
BitArray::BitArray(size_t size) : data(nullptr),
    bit_size(0), byte_capacity(0) {
    resize_if_needed(size);
}
BitArray::BitArray(const char* str) : data(nullptr),
    bit_size(0), byte_capacity(0) {
    from_char_string(str);
```

```
}
BitArray::~BitArray() {
    if (data != nullptr) {
        delete[] data;
        data = nullptr;
    }
    bit_size = 0;
    byte_capacity = 0;
}
size_t BitArray::size() const {
    return bit_size;
}
bool BitArray::get_bit(size_t index) const {
    validate_index(index);
    size_t byte_idx = byte_index(index);
    size_t bit_off = bit_offset(index);
    return (data[byte_idx] >> bit_off) & 1;
}
void BitArray::set_bit(size_t index, bool value) {
    validate_index(index);
    size_t byte_idx = byte_index(index);
    size_t bit_off = bit_offset(index);
    if (value) {
        data[byte_idx] |= (1 << bit_off);
    } else {
        data[byte_idx] &= ~(1 << bit_off);
    }
}
void BitArray::set_single_bit(size_t index, bool value)
{
```

```
    set_bit(index, value);
}
void BitArray::set_bit_range(size_t start_index, size_t
    count, bool value) {
    for (size_t i = 0; i < count; i++) {
        if (start_index + i < bit_size) {
            set_bit(start_index + i, value);
        }
    }
}
void BitArray::invert_single_bit(size_t index) {
    set_bit(index, !get_bit(index));
}
void BitArray::invert_bit_range(size_t start_index,
    size_t count) {
    for (size_t i = 0; i < count; i++) {
        if (start_index + i < bit_size) {
            invert_single_bit(start_index + i);
        }
    }
}
void BitArray::logical_shift_left(size_t shift_count) {
    if (shift_count == 0 || shift_count >= bit_size) {
        if (shift_count >= bit_size) {
            for (size_t i = 0; i < bit_size; i++) {
                set_bit(i, false);
            }
        }
        return;
    }
```

```
    for (size_t i = 0; i < bit_size - shift_count; i++)
{
        set_bit(i, get_bit(i + shift_count));
    }
    for (size_t i = bit_size - shift_count; i <
bit_size; i++) {
        set_bit(i, false);
    }
}
void BitArray::logical_shift_right(size_t shift_count)
{
    if (shift_count == 0 || shift_count >= bit_size) {
        if (shift_count >= bit_size) {
            for (size_t i = 0; i < bit_size; i++) {
                set_bit(i, false);
            }
        }
        return;
    }
    for (size_t i = bit_size - 1; i >= shift_count; i--
) {
        set_bit(i, get_bit(i - shift_count));
    }
    for (size_t i = 0; i < shift_count; i++) {
        set_bit(i, false);
    }
}
BitArray BitArray::union_with(const BitArray& other)
    const {
```

```
    if (bit_size != other.bit_size) {
        throw std::invalid_argument("Bit arrays must
            have the same size for union operation");
    }
    BitArray result(bit_size);
    size_t total_bytes = (bit_size + 7) / 8;
    for (size_t i = 0; i < total_bytes; i++) {
        result.data[i] = data[i] | other.data[i];
    }
    return result;
}
BitArray BitArray::intersection_with(const BitArray&
other) const {
    if (bit_size != other.bit_size) {
        throw std::invalid_argument("Bit arrays must
have the same size for intersection operation");
    }
    BitArray result(bit_size);
    size_t total_bytes = (bit_size + 7) / 8;
    for (size_t i = 0; i < total_bytes; i++) {
        result.data[i] = data[i] & other.data[i];
    }
    return result;
}
BitArray BitArray::difference_with(const BitArray&
    other) const {
    if (bit_size != other.bit_size) {
        throw std::invalid_argument("Bit arrays must
            have the same size for difference
            operation");
```

```
    }
    BitArray result(bit_size);
    size_t total_bytes = (bit_size + 7) / 8;
    for (size_t i = 0; i < total_bytes; i++) {
        result.data[i] = data[i] & ~other.data[i];
    }
    return result;
}
char* BitArray::to_char_string() const {
    char* str = new char[bit_size + 1];
    for (size_t i = 0; i < bit_size; i++) {
        str[i] = get_bit(i) ? '1' : '0';
    }
    str[bit_size] = '\0';
    return str;
}
void BitArray::from_char_string(const char* str) {
    size_t len = strlen(str);
    if (data != nullptr) {
        delete[] data;
        data = nullptr;
    }
    resize_if_needed(len);
    for (size_t i = 0; i < len; i++) {
        if (str[i] == '1') {
            set_bit(i, true);
        } else if (str[i] == '0') {
            set_bit(i, false);
        } else {
```

```
            throw std::invalid_argument("Invalid
                character in bit string");
        }
    }
}
void BitArray::print() const {
    char* str = to_char_string();
    std::cout << "BitArray[" << bit_size << "]: " <<
        str << std::endl;
    delete[] str;
}
void BitArray::random_fill() {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, 1);
    for (size_t i = 0; i < bit_size; i++) {
        set_bit(i, dis(gen) == 1);
    }
    }
```

Файл BitArray.h

```
#ifndef BITARRAY_H
#define BITARRAY_H
#include <cstddef>
class BitArray {
private:
    unsigned char* data;
    size_t bit_size;
    size_t byte_capacity;
    void validate_index(size_t index) const;
    size_t byte_index(size_t bit_index) const;
    size_t bit_offset(size_t bit_index) const;
```

```
    void resize_if_needed(size_t new_bit_size);
public:
    BitArray();
    BitArray(size_t size);
    BitArray(const char* str);
    ~BitArray();
    size_t size() const;
    bool get_bit(size_t index) const;
    void set_bit(size_t index, bool value);
    void set_single_bit(size_t index, bool value);
    void set_bit_range(size_t start_index, size_t
        count, bool value);
    void invert_single_bit(size_t index);
    void invert_bit_range(size_t start_index, size_t
        count);
    void logical_shift_left(size_t shift_count);
    void logical_shift_right(size_t shift_count);
    BitArray union_with(const BitArray& other) const;
    BitArray intersection_with(const BitArray& other)
        const;
    BitArray difference_with(const BitArray& other)
        const;
    char* to_char_string() const;
    void from_char_string(const char* str);
    void print() const;
    void random_fill();
};
#endif
```