

Санкт-Петербургский политехнический университет Петра Великого  
Институт металлургии, машиностроения и транспорта  
Высшая школа автоматизации и робототехники

## Курсовая работа

Дисциплина: Объектно-ориентированное программирование

Тема: Дерево Фенвика. Бинарное индексированное дерево

Выполнил студент гр. 3331506/10401

Макаров А. П.

Преподаватель

Ананьевский М. С.

Санкт-Петербург

2024

## Оглавление

<b>1 Введение.....</b>	<b>3</b>
<b>2 Дерево Фенвика .....</b>	<b>4</b>
2.1 Теоретические сведения.....	4
2.2 Нахождение суммы интервала .....	6
2.3 Обновление дерева Фенвика.....	7
2.3 Анализ результатов .....	9
<b>3 Реализация алгоритма .....</b>	<b>10</b>
<b>4 Заключение.....</b>	<b>13</b>
<b>Литература .....</b>	<b>14</b>
<b>Приложение.....</b>	<b>15</b>

## 1 Введение

Дерево Фенвика — это алгоритмическая структура данных, используемая в информатике и компьютерных науках для эффективного выполнения операций на массивах и последовательностях, названное в честь исследователя Роберта Фенвика. Это дерево представляет собой сбалансированную структуру данных, специально разработанную для эффективного выполнения операций поиска значения, обновления значения, нахождения суммы диапазона. Изначально разработанное для работы с массивами, дерево Фенвика нашло широкое применение в различных областях, таких как компьютерная графика, сжатие данных, обработка изображений и алгоритмы сортировки.

Целью данной курсовой работы является изучение основных принципов работы дерева Фенвика, его структуры, операций и алгоритмов. В курсовой работе будут рассмотрены основные принципы построения и функционирования дерева Фенвика, его преимущества и недостатки.

## 2 Деревя Фенвика

### 2.1 Теоретические сведения

Алгоритм дерева Фенвика — это структура данных, дерево на массиве, которая обладает следующими свойствами:

- позволяет вычислять значение некоторой обратимой операции  $F$  на любом отрезке  $[L; R]$  за логарифмическое время;
- позволяет изменять значение любого элемента за  $O(\log N)$ ;
- требует памяти  $O(N)$ ;

Операция  $F$  может быть выбрана разными способами, но чаще всего берутся операции суммы интервала, произведение интервала, а также при определенной модификации и ограничениях, нахождения максимума и нахождения минимума на интервале или другие операции.

#### Преимущества дерева Фенвика:

1. Эффективность времени: Операции выполняются за  $O(\log N)$ .
2. Простота реализации: Легко реализовать и использовать.
3. Малое потребление памяти: Требуется  $O(n)$  дополнительной памяти.

#### Недостатки дерева Фенвика:

1. Ограниченность функциональности: Поддерживает ограниченное количество типов операций.
2. Не поддерживает произвольный доступ: Не подходит для прямого доступа к произвольным элементам.

Для рассмотрения преимуществ дерева Фенвика рассмотрим реализацию задачи нахождения суммы элементов в исходном массиве на нужном нам диапазоне двумя способами.

Пусть нам дан исходный массив, состоящий из 16 элементов.

Исходный массив:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----

Для быстрого нахождения суммы на определенном диапазоне введем массив, который будет содержать в себе последовательное суммирование каждого элемента исходного массива.

Массив сумм:

1	3	6	10	15	21	28	36	45	55	66	78	91	105	120	136
---	---	---	----	----	----	----	----	----	----	----	----	----	-----	-----	-----

И составим массив, который будет соответствовать дереву Фенвика. Четные элементы в дереве Фенвика хранят элементы исходного массива, также дерево Фенвика, аналогично массиву сумм хранит сумму элементов исходного массива, также присутствуют ячейки, которые хранят значения от определенного элемента до текущего.

Дерево Фенвика:

1	3	3	10	5	11	7	36	9	19	11	42	13	27	15	136
---	---	---	----	---	----	---	----	---	----	----	----	----	----	----	-----

$$a[1] = a[0] + a[1] = 3$$

$$a[3] = a[0] + a[1] + a[2] + a[3] = 10$$

$$a[5] = a[4] + a[5] = 11$$

$$a[7] = a[0] + a[1] + a[2] + \dots + a[7] = 36$$

$$a[9] = a[8] + a[9] = 19$$

$$a[11] = a[8] + \dots + a[11] = 42$$

$$a[13] = a[13] + \dots + a[14] = 27$$

$$a[15] = a[0] + \dots + a[15] = 136$$

Для визуального понимания проиллюстрируем сделанные нами действия на графике, где по оси  $x$  у нас будет индекс элемента массива, а по оси  $y$  будут закрашены те индексы, просуммировав которые получим значение элемента в массиве по индексу указанного по оси  $x$ .

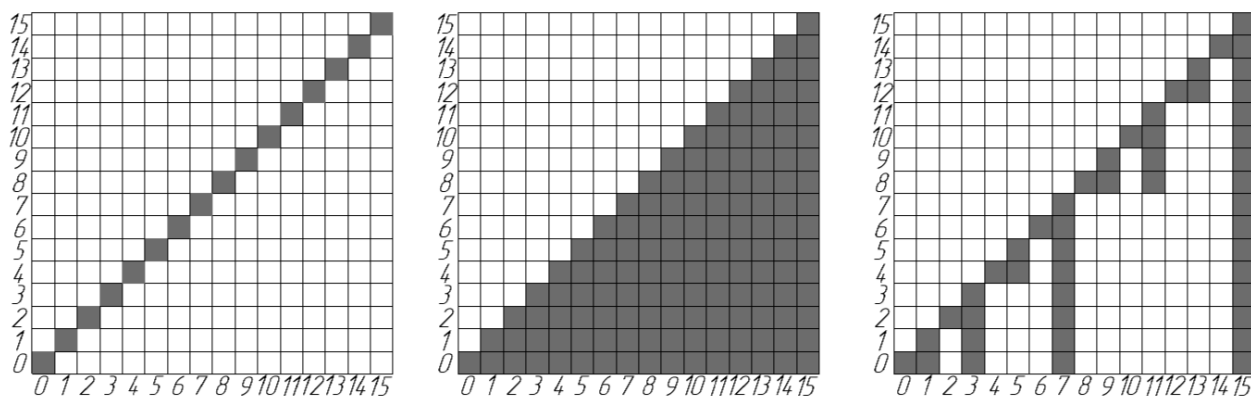


Рисунок 1 - Сравнение массивов и дерева Фенвика

## 2.2 Нахождение суммы интервала

Так, в дереве Фенвика, чтобы получить сумму от 6 до 8 элемента, требуется вычислить сумму от 0 до 8 элемента, то есть сложить значения в 7 и 8 ячейке, а затем вычесть сумму от 0 до 5 элемента, то есть вычесть значения, которые находятся в 3 и 5 ячейке.

$$sum[6..8] = sum[0..8] - sum[0..5] = (a[7] + a[8]) - (a[3] + a[5])$$

Вопрос заключается в том, что не совсем понятно от какого элемента требуется начинать суммирование. Другими словами, как зная текущий индекс найти индекс от которого мы начинаем суммирование, чтобы построить дерево Фенвика. Какую формулу нужно использовать, чтобы имея, например, индекс 5, получить 4, а имея индекс 11 получить 8.

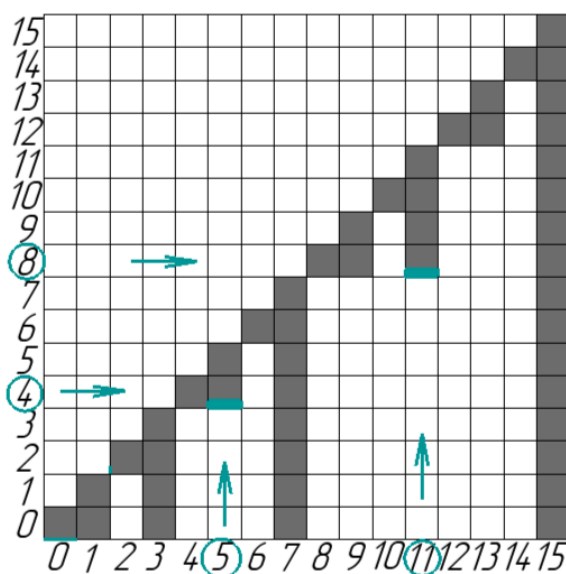


Рисунок 2 – Принцип построения дерева Фенвика

Составим таблицу отображения индексов элемента массива на индекс элемента от которого нам нужно начинать суммирование.

Таблица 1 – Отображение индексов  $i$  от  $f(i)$  для суммирования

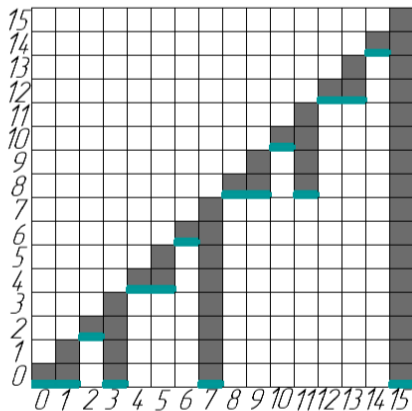


Рисунок 3 - Отображения индексов элемента массива на индекс элемента от которого нам нужно начинать суммирование

$i$	$f(i)$	$i$	$f(i)$
0	0	0000	0000
1	0	0001	0000
2	2	0010	0010
3	0	0011	0000
4	4	0100	0100
5	4	0101	0100
6	6	0110	0110
7	0	0111	0000
8	8	1000	1000
9	8	1001	1000
10	10	1010	1010
11	8	1011	1000
12	12	1100	1100
13	12	1101	1100
14	14	1110	1110
15	0	1111	0000

В итоге получим зависимость

$$f[i] = i \& (i + 1), \text{ где } \& \text{ – побитовая операция И}$$

Для того, чтобы определить от какого элемента  $f[i]$  необходимо начинать проводить суммирование для построения, например,  $i = 11$  элемента дерева Фенвика.

$$f[11] = 1011 \& (1011 + 0001) = 1011 \& 1100 = 1000 \rightarrow 8$$

Значит, в 11 ячейке должна находиться сумма от 8 до 11 элемента текущего массива.

### 2.3 Обновление дерева Фенвика

Теперь займемся следующей задачей. Что если нам потребуется изменить какое либо значение массива. Чтобы обновить некоторый элемент по индексу, требуется обновить ячейку по тому же индексу, затем вычислить

следующую ячейку для обновления и повторить эти действия для вычисленной следующей ячейки. Алгоритм обновления ячейки в дереве Фенвика является рекурсивным.

Составим таблицу отображения индексов элемента массива и индекса элемента следующего для обновления.

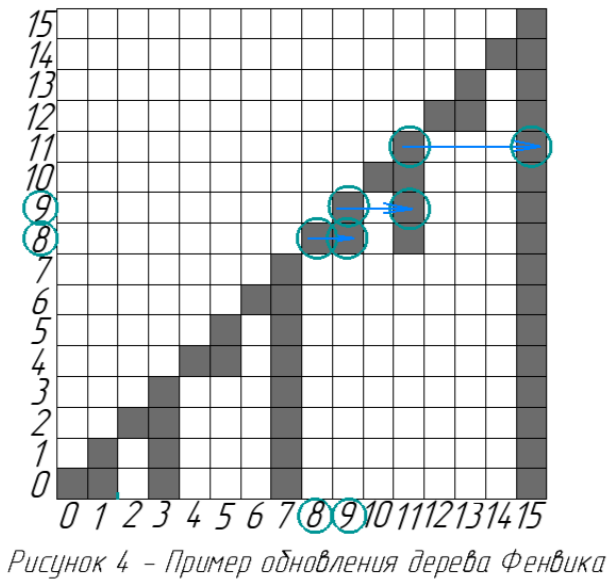


Таблица 2 – Отображение  $i$  от  $f(i)$  для обновления

$i$	$f(i)$	$i$	$f(i)$
0	1	0000	0000
1	3	0001	0000
2	3	0010	0010
3	7	0011	0000
4	5	0100	0100
5	7	0101	0100
6	7	0110	0110
7	15	0111	0000
8	9	1000	1000
9	10	1001	1000
10	11	1010	1010
11	15	1011	1000
12	13	1100	1100
13	15	1101	1100
14	15	1110	1110
15	...	1111	—

Зависимость обновленной и следующей ячейки будет выглядеть следующим образом:

$$f[i] = i | (i + 1), \text{ где } | - \text{ побитовая операция ИЛИ}$$

Итак, в исходном массиве состоящий из 15 элементов поменяли значение 8 элемента и теперь, для обновления дерева Фенвика необходимо воспользоваться формулой, которая позволяет определить индекс следующего элемента и продолжить повторение процедуры до окончания дерева(массива) Фенвика.

$$i = 8 \qquad f[8] = 8 | (8 + 1) \rightarrow 1000 | 1001 = 1001 \rightarrow 9$$



$$i = 9 \quad f[9] = 9 \mid (9 + 1) \rightarrow 1001 \mid 1010 = 1011 \rightarrow 11$$

$$i = 11 \quad f[11] = 11 \mid (11 + 1) \rightarrow 1011 \mid 1100 = 1111 \rightarrow 15$$

Визуализация обновления показана на рисунке 4.

Графическое изображение дерева Фенвика представлено на рисунке 5.

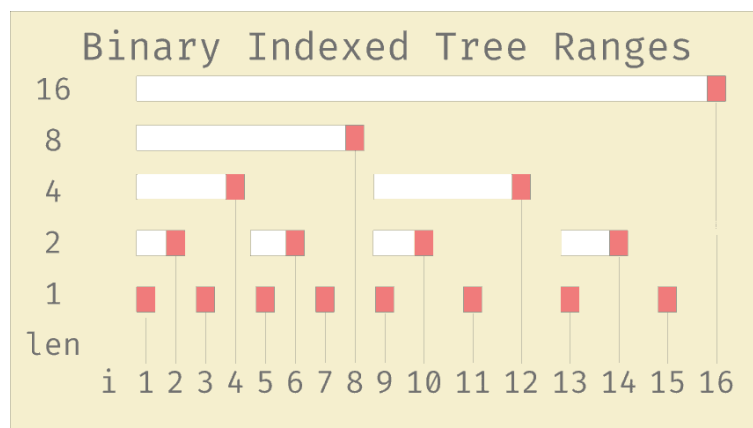


Рисунок 5 – Графическое изображение дерева Фенвика

## 2.4 Анализ результатов

Можно сделать вывод, что исходный массив и массив сумм обладают недостатками для решения задачи нахождения суммы элементов. Исходный массив будет обладает высокой (постоянная сложность) скоростью обновления данных в массиве, но медленно (линейная сложность) считать сумму своих элементов. Массив сумм наоборот, быстро (постоянная сложность) посчитает сумму, но его сложнее обновить (линейная сложность обновления). При подсчете и обновлении больших массивов, оба массива не годятся, но с этими задачами справляется дерево Фенвика. Дерево Фенвика позволяет проводить операции сложения и обновления элементов за логарифмическую сложность, например суммирование 1 миллиарда элементов займет около 30 операций. Итоги данного рассуждения приведены в сводной таблице.

Таблица 3 – Сводная таблица сравнения

	Обновление заданного элемента	Нахождение суммы элементов
Исходный массив	$O(1)$	$O(n)$
Массив сумм	$O(n)$	$O(1)$
Дерево Фенвика	$O(\log(n))$	$O(\log(n))$

### 3 Реализация алгоритма

В процессе курсовой работы был реализован код дерева Фенвика, который позволяет эффективно выполнять операции суммирования, обновления, а также поиска минимального и максимального значения в диапазоне. Ниже представлены основные функции с пояснениями.

#### 1) Получение суммы до индекса $i$ .

Функция *get\_prefix\_sum* возвращает сумму элементов от начала до индекса  $i$ . Здесь используется битовая операция для перехода к следующему элементу дерева Фенвика.

```
int get_prefix_sum(int i)
{
    int result = 0;
    for (; i >= 0; i = (i & (i + 1)) - 1)
    {
        result += fenwick_tree_sum[i];
    }
    return result;
}
```

#### 2) Обновление значения в массиве.

Функция *update\_value* обновляет значение элемента в исходном массиве и соответственно обновляет дерево Фенвика. *delta* – изменение, которое нужно применить к элементу. Обновляются также деревья для минимальных и максимальных значений.

```
void update_value(int idx, int delta)
{
    source_array[idx] += delta;
    for (int i = idx; i < size; i |= i + 1)
    {
        fenwick_tree_sum[i] += delta;
    }

    for (int i = idx; i < size; i |= i + 1)
    {
        fenwick_tree_min[i] = std::min(fenwick_tree_min[i], source_array[idx]);
        fenwick_tree_max[i] = std::max(fenwick_tree_max[i], source_array[idx]);
    }
}
```

#### 3) Получение суммы в диапазоне

Функция *get\_sum\_in\_range* возвращает сумму элементов от *left* до *right*. Если *left* больше нуля, то вычисляется разница между суммами до *right* и до *left - 1*.

```
int get_sum_in_range(int left, int right)
{
    if (left)
    {
        return get_prefix_sum(right) - get_prefix_sum(left - 1);
    }
    else
    {
        return get_prefix_sum(right);
    }
}
```

4) Получение минимального и максимального значений до индекса *i*.

Данная функция возвращает минимальное и максимальное значения в диапазоне от начала индекса *i*.

```
int get_min_prefix(int i)
{
    int result = INT_MAX;
    for (; i >= 0; i = (i & (i + 1)) - 1)
    {
        result = std::min(result, fenwick_tree_min[i]);
    }
    return result;
}
```

```
int get_max_prefix(int i)
{
    int result = INT_MIN;
    for (; i >= 0; i = (i & (i + 1)) - 1)
    {
        result = std::max(result, fenwick_tree_max[i]);
    }
    return result;
}
```

5) Получение минимального и максимального значений в диапазоне

Эти функции возвращают минимальное и максимальное значения в диапазоне от *left* до *right*.

```
int get_min_in_range(int left, int right)
{
    if (left == 0)
    {
        return get_min_prefix(right);
    }
}
```

```

    }
    else
    {
        return std::min(get_min_prefix(right), get_min_prefix(left - 1));
    }
}

```

```

int get_max_in_range(int left, int right)
{
    if (left == 0)
    {
        return get_max_prefix(right);
    }
    else
    {
        return std::max(get_max_prefix(right), get_max_prefix(left - 1));
    }
}

```

Весь код алгоритма представлен в Приложении.

#### 4 Заключение

В данной курсовой работе был проведен анализ структуры данных дерева Фенвика. Мы рассмотрели основные теоретические аспекты данного алгоритма, его ключевые операции, такие как нахождение суммы интервала и обновление дерева, нахождение максимального и минимального значения на интервале, а также их временные характеристики. Проведенное исследование показало, что дерево Фенвика является эффективным инструментом для выполнения этих операций с временной сложностью  $O(\log n)$ , что значительно превосходит подходы с временной сложностью  $O(n)$ .

Реализация алгоритма на языке программирования C++ продемонстрировала практическую применимость и эффективность дерева Фенвика в реальных задачах. Код, представленный в работе, включает функции для построения дерева, обновления его элементов, вычисления суммы элементов в заданном интервале и определение максимального и минимального значения. Проведенный тестовый анализ подтвердил теоретические выкладки, показав высокую производительность алгоритма даже на больших массивах данных.

## Список литературы

1. Роналд Грэхем, Дональд Кнут, Орен Паташник. Конкретная математика. Основание информатики. — М.: Высшая школа, 1986. — С. 298-310.
2. Rodica Ceterchi, Mircea Dimama. Efficient Range Minimum Queries using Binary Indexed Trees.
3. Девятериков Иван. Структура данных Дерево отрезков и её применение в задачах.
4. Knuth, Donald (2011). *Combinatorial Algorithms, Part 1*. The Art of Computer Programming. Vol. 4A. Upper Saddle River, NJ: Addison-Wesley Professional. pp. 164–165.

## Приложение

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <limits.h>

std::vector<int> source_array;
std::vector<int> fenwick_tree_sum;
std::vector<int> fenwick_tree_min;
std::vector<int> fenwick_tree_max;
int size;

int get_prefix_sum(int i)
{
    int result = 0;
    for (; i >= 0; i = (i & (i + 1)) - 1)
    {
        result += fenwick_tree_sum[i];
    }
    return result;
}

void update_value(int idx, int delta)
{
    source_array[idx] += delta;
    for (int i = idx; i < size; i |= i + 1)
    {
        fenwick_tree_sum[i] += delta;

        for (int j = idx; j < size; j |= j + 1)
        {
            fenwick_tree_min[j] = std::min(fenwick_tree_min[j], source_array[idx]);
            fenwick_tree_max[j] = std::max(fenwick_tree_max[j], source_array[idx]);
        }
    }
}

int get_sum_in_range(int left, int right)
{
    if (left)
    {
        return get_prefix_sum(right) - get_prefix_sum(left - 1);
    }
    else
    {
        return get_prefix_sum(right);
    }
}

int get_min_prefix(int i)
{
    int result = INT_MAX;
    for (; i >= 0; i = (i & (i + 1)) - 1)
    {
        result = std::min(result, fenwick_tree_min[i]);
    }
}
```

```

    }
    return result;
}

int get_max_prefix(int i)
{
    int result = INT_MIN;
    for (; i >= 0; i = (i & (i + 1)) - 1)
    {
        result = std::max(result, fenwick_tree_max[i]);
    }
    return result;
}

int get_min_in_range(int left, int right)
{
    if (left == 0)
    {
        return get_min_prefix(right);
    }
    else
    {
        return std::min(get_min_prefix(right), get_min_prefix(left - 1));
    }
}

int get_max_in_range(int left, int right)
{
    if (left == 0)
    {
        return get_max_prefix(right);
    }
    else
    {
        return std::max(get_max_prefix(right), get_max_prefix(left - 1));
    }
}

void print_source_array()
{
    for (int i = 0; i < size; ++i)
    {
        std::cout << source_array[i] << " ";
    }
    std::cout << "\n";
}

void print_fenwick(std::vector<int>& fenwick_tree)
{
    for (int i = 0; i < size; ++i)
    {
        std::cout << fenwick_tree[i] << " ";
    }
    std::cout << "\n";
}

```



```

int main()
{
    std::cout << "Enter the number of elements in the array: ";
    std::cin >> size;

    source_array.resize(size);
    fenwick_tree_sum.resize(size, 0);
    fenwick_tree_min.resize(size, INT_MAX);
    fenwick_tree_max.resize(size, INT_MIN);

    for (int i = 0; i < size; i++) {
        int value;
        std::cout << "Enter an element in an array: ";
        std::cin >> value;
        update_value(i, value);
    }

    std::cout << "\nSource Array: ";
    print_source_array();

    std::cout << "\nFenwick Tree: ";
    print_fenwick(fenwick_tree_sum);

    std::cout << "\nSum from 1 to 3 elements: " << get_prefix_sum(2) << std::endl;
    std::cout << "\nSum from 3 to 5 elements: " << get_sum_in_range(2, 4) << std::endl;

    std::cout << "\nUpdate element 2 by the value delta = -3\n";
    update_value(1, -3);

    std::cout << "\nSource Array after update: ";
    print_source_array();

    std::cout << "\nFenwick Tree after update: ";
    print_fenwick(fenwick_tree_sum);

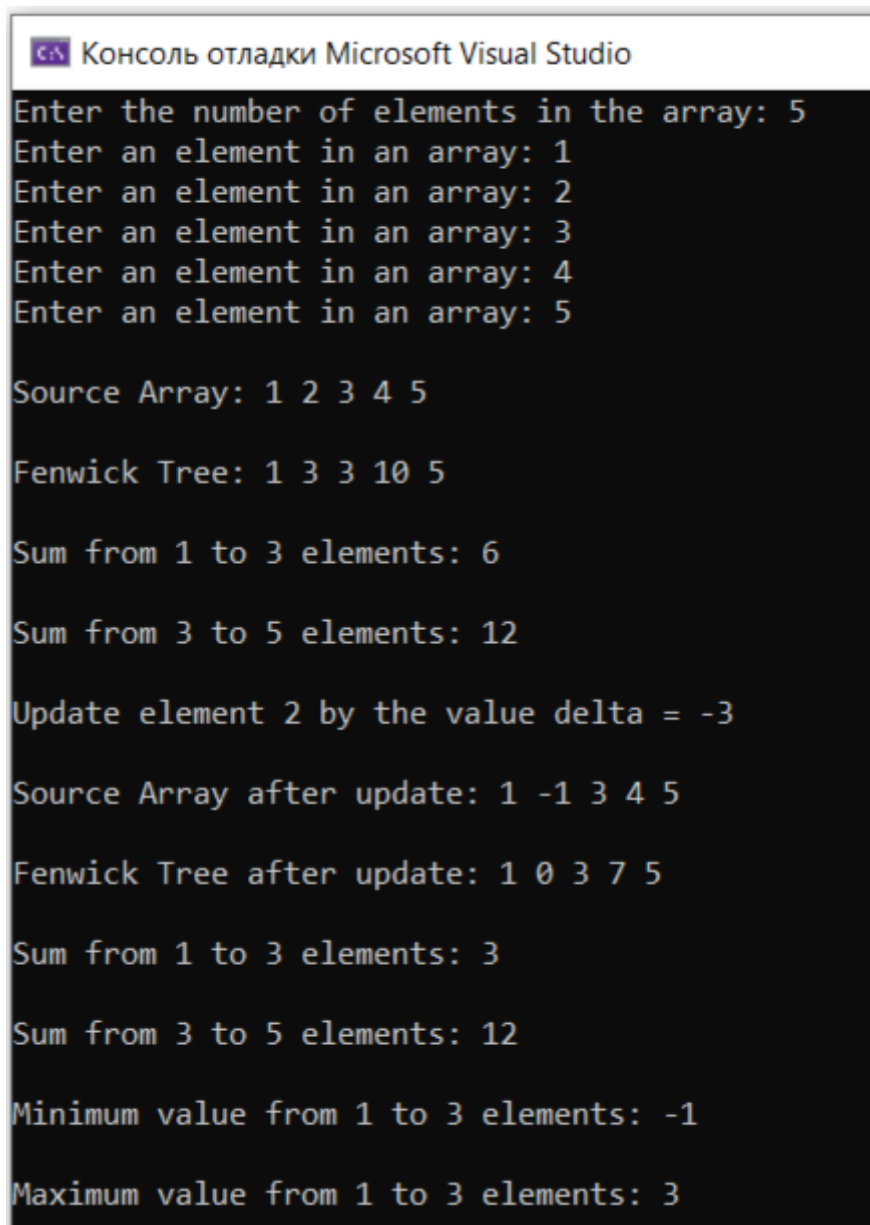
    std::cout << "\nSum from 1 to 3 elements: " << get_prefix_sum(2) << std::endl;
    std::cout << "\nSum from 3 to 5 elements: " << get_sum_in_range(2, 4) << std::endl;

    std::cout << "\nMinimum value from 1 to 3 elements: " << get_min_in_range(0, 2) << std::endl;
    std::cout << "\nMaximum value from 1 to 3 elements: " << get_max_in_range(0, 2) << std::endl;

    return 0;
}

```

## Результат работы программы

The image shows a screenshot of the 'Консоль отладки Microsoft Visual Studio' (Microsoft Visual Studio Debug Console). The console has a black background with white text. It displays the following sequence of commands and outputs:

```
Enter the number of elements in the array: 5
Enter an element in an array: 1
Enter an element in an array: 2
Enter an element in an array: 3
Enter an element in an array: 4
Enter an element in an array: 5

Source Array: 1 2 3 4 5

Fenwick Tree: 1 3 3 10 5

Sum from 1 to 3 elements: 6

Sum from 3 to 5 elements: 12

Update element 2 by the value delta = -3

Source Array after update: 1 -1 3 4 5

Fenwick Tree after update: 1 0 3 7 5

Sum from 1 to 3 elements: 3

Sum from 3 to 5 elements: 12

Minimum value from 1 to 3 elements: -1

Maximum value from 1 to 3 elements: 3
```

Рисунок 5 – Пример работы программы