МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт искусственного интеллекта

Кафедра Высшей математики

КУРСОВАЯ РАБОТА

по дисциплине
«Объектно-ориентированное программирование»

Тема курсовой работы
«Моделирование дерева отрезков»

Студент группы КМБО-04-22 *Кемаев И.О.*

Руководитель курсовой работы
доцент кафедры Высшей математики
к.ф.-м.н. *Петрусевич Д.А.*

Работа представлена к защите «21» дек 2023 г. *(подпись студента)*

«Допущен к защите» «21» дек 2023 г. *(подпись руководителя)*

МОСКВА — 2023

МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт искусственного интеллекта

Кафедра Высшей математики

Утверждаю
Исполняющий обязанности заведующего
кафедрой _____ *А.В.Шатина*

«22» сентября 2023 г.

ЗАДАНИЕ
на выполнение курсовой работы
по дисциплине «Объектно-ориентированное программирование»

Студент *Кемаев И.О.* Группа *КМБО-04-22*

1. **Тема: «Моделирование дерева отрезков»**

2. **Исходные данные:**
Реализовать класс «Дерево отрезков»
Реализовать необходимые для класса функции: конструкторы, деструкторы, нужные методы
Реализовать поиск значения функции на отрезке
Реализовать групповое изменение значений в определенном отрезке
Для выполнения групповых операций реализовать отложенное выполнение команд
Сделать выводы о том, какие функции могут быть групповыми.

3. **Перечень вопросов, подлежащих разработке, и обязательного графического материала:**
Продемонстрировать работу дерева отрезков на определенных полуинтервалах
Продемонстрировать работу дерева отрезков на объектах сложной природы (ключ – одно поле или комбинация нескольких полей)

4. **Срок представления к защите курсовой работы:** до «22» декабря 2023 г.

Задание на курсовую работу выдал	«22» сентября 2023 г.	_____ *(Петрусевич Д.А.)*
Задание на курсовую работу получил	«22» сентября 2023 г.	_____ *(Кемаев И.О.)*

Оглавление

Введение

Дерево отрезков (Segment Tree) представляет собой структуру данных, используемую для решения множества задач, связанных с обработкой интервалов и диапазонов в массиве или последовательности данных. Оно обладает эффективностью поиска, изменения и агрегации информации в определенных отрезках данных.

Эта структура нашла широкое применение в различных областях, включая информационные технологии, обработку изображений, базы данных, анализ данных, графику, алгоритмы и многие другие сферы. Одним из ключевых преимуществ дерева отрезков является его способность обрабатывать запросы на поиск минимума, максимума, суммы или других агрегирующих операций на заданных отрезках данных с логарифмической сложностью по времени.

В рамках данной курсовой работы осуществлялась реализация дерева отрезков с фокусом на операциях, выполняемых над полуинтервалами от [r,l) данных . Рассматриваемые операции, такие как поиск минимума, максимума, суммы значений на отрезке, требуют определенных свойств от структуры дерева отрезков.

Основные требования к операциям над отрезками включают их корректность, эффективность и возможность выполнения с заданной сложностью времени. Эти операции должны обладать свойствами ассоциативности и коммутативности для обеспечения правильности результата. Например такие функции как сумма, минимум, максимум, наибольший общий делитель являются подходящими для вычисления на интервалах с помощью дерева отрезков

Для оптимизации процесса выполнения операций используется техника - отложенное выполнение, позволяющая задерживать применение операций к узлам структуры до необходимости. Это уменьшает количество операций и оптимизирует время выполнения запросов. Операции сохраняются для применения к узлам только тогда, когда это действительно требуется для получения результата или выполнения операции..

Таким образом, курсовая работа включает в себя реализацию дерева отрезков с определенными операциями над отрезками данных и акцентирует внимание на эффективности и правильности выполнения данных операций, в том числе с используя техники проталкивания.

Глава 1. Дерево отрезков
1.1 Постановка задачи. Основные определения

Дерево отрезков (Segment Tree) — это эффективная структура данных, широко используемая для решения разнообразных задач, связанных с работой над интервалами или отрезками в массивах. Эта структура данных позволяет выполнять операции поиска, обновления и агрегации значений в заданных интервалах массива с логарифмической сложностью относительно размера массива. Вот основные аспекты дерева отрезков:

Структура дерева: Дерево отрезков представляет собой бинарное дерево, где каждый узел соответствует определенному интервалу в исходном массиве. В корне дерева содержится весь исходный массив, а листья дерева представляют собой одиночные элементы исходного массива.

Построение дерева: Дерево отрезков строится рекурсивно путем разбиения интервала массива пополам до тех пор, пока не достигнуты листья. В узлах дерева хранятся агрегированные значения для соответствующего интервала. Эти агрегированные значения могут быть, например, суммой, минимумом, максимумом или другой операцией, зависящей от задачи.

Операции: Когда речь идет о дереве отрезков, две основные операции, которые мы обычно выполняем, это запрос интервала (range query) и обновление элемента массива (update). Давайте подробнее разберем их асимптотику и работу:

1. Запрос интервала (Range Query):
Операция запроса интервала в дереве отрезков позволяет найти агрегированное значение (сумму, минимум, максимум и т.д.) в заданном интервале массива.

- Для выполнения запроса интервала, мы начинаем с корня дерева и рассматриваем интервал в текущем узле.
- Если интервал текущего узла полностью входит в интервал запроса, то его агрегированное значение может быть использовано без дополнительных проверок.
- В противном случае, мы рекурсивно переходим к детям этого узла, пока не найдем узлы, полностью или частично покрывающие интервал запроса.
- В худшем случае, высота дерева отрезков составляет O(log N), где N - размер исходного массива.
- Каждый узел посещается только один раз, и агрегированное значение вычисляется, поэтому общее время выполнения запроса интервала составляет O(log N).

2. Обновление элемента массива (Update):

- Операция обновления элемента позволяет изменить значение одного элемента в исходном массиве и соответствующие агрегированные значения в дереве отрезков.
- Для выполнения обновления, мы начинаем с корня дерева и спускаемся к листу, представляющему обновляемый элемент.
- Мы обновляем значение в этом листе и затем пересчитываем агрегированные значения на пути к корню дерева.
- В худшем случае, высота дерева отрезков составляет O(log N), где N - размер исходного массива.
- Так как мы пересчитываем агрегированные значения на пути к корню, обновление займет O(log N) времени.

Таким образом, обе основные операции (запрос интервала и обновление элемента) в дереве отрезков имеют логарифмическую асимптотику относительно размера исходного массива.

3. Примеры использования: Дерево отрезков находит применение во многих задачах, таких как:

- Поиск суммы элементов в интервале массива.
- Поиск минимального или максимального элемента в интервале.
- Подсчет количества элементов, удовлетворяющих определенному условию в интервале.
- Обновление значений элементов массива и пересчет агрегированных значений (групповая функция) .

Групповая функция для дерева отрезков должна быть ассоциативной и коммутативной, то есть выполняться свойства:

- Ассоциативность: для любых элементов a, b и c из диапазона данных дерева отрезков, групповая функция должна удовлетворять следующему условию:
  func(func(a, b), c) = func(a, func(b, c))
- Коммутативность: порядок операций не должен влиять на результат функции.
  func(a, b) = func(b, a)

Примеры таких функций могут включать сложение чисел, умножение, нахождение минимума или максимума и т.д.

Отличие от групповой функции для декартова дерева заключается в том, что для дерева отрезков мы работаем с интервалами значений, а для декартова дерева - с парой значений (ключ, приоритет). В декартовом дереве групповая функция применяется к ключам, а приоритет помогает поддерживать

структуру дерева и обеспечивать его балансировку. Групповая функция для декартова дерева должна быть только ассоциативной.

4. Оптимизации: Существует множество оптимизаций для дерева отрезков, таких как проталкивание (lazy propagation), которая позволяет эффективно обновлять интервалы в дереве, минимизируя количество обработанных узлов.

Это методика оптимизации дерева отрезков, которая позволяет эффективно обновлять значения узлов при изменении или модификации элементов входного массива.

Когда элементы в массиве изменяются, а дерево отрезков используется для хранения агрегированных значений (например, суммы или минимального/максимального значения) на интервалах, техника проталкивания позволяет обновлять значения узлов дерева в процессе спуска или обновления по дереву.

Суть техники проталкивания заключается в том, что при изменении значения в каком-то листовом узле дерева отрезков, это изменение проталкивается (передается) вверх по дереву к корню. Таким образом, обновления происходят в узлах, которые влияют на изменяемый узел.

Процесс проталкивания часто применяется при отложенном выполнении операций, чтобы обеспечить актуальность значений в узлах дерева. Это позволяет минимизировать число операций и обновлений, которые должны быть выполнены для поддержания корректных значений в дереве отрезков.

Например, если у вас есть дерево отрезков, хранящее суммы значений на отрезках массива, и вы изменяете один элемент массива, значение этого элемента проталкивается вверх по дереву отрезков, пересчитывая все соответствующие суммы, чтобы обновить связанные с этим узлом значения в дереве.

5. Обработка запросов и обновлений: Для выполнения запросов интервала и обновления элементов массива нужны алгоритмы, которые позволяют навигироваться по дереву отрезков, находить интересующие участки и выполнять операции агрегации и обновления.

6. Сегментация интервалов: Если вам нужно работать с интервалами, разные задачи могут потребовать различных методов сегментации интервалов. Например, в задаче поиска суммы элементов в интервале, интервалы могут перекрываться, а в задаче поиска минимума или максимума - нет.

1.2 Алгоритмы и структуры данных для построения дерева отрезков

Для реализации дерева отрезков требуются несколько ключевых алгоритмов и структур данных:

1. Рекурсия: Дерево отрезков часто строится рекурсивно. Каждый узел дерева представляет интервал массива, и для построения дерева мы разделяем интервалы пополам до тех пор, пока не достигнем листьев (отдельных элементов исходного массива).

2. Массив : Для хранения дерева отрезков, используется массив. В этой структуре данных каждый узел дерева будет представлен элементом в массиве. Индексы элементов массива используются для определения отношений между узлами дерева.

3. Агрегирующая функция: Для каждого узла дерева отрезков требуется определить, какая агрегирующая функция будет использоваться. Эта функция зависит от задачи и может быть, например, суммой, минимумом, максимумом или другой операцией, которая определяет значение интервала на основе значений его детей.

4. Заполнение дерева: При построении дерева отрезков необходимо заполнить значениями агрегирующей функции все узлы дерева. Это может быть выполнено при инициализации структуры данных или при построении дерева.

5 Техника проталкивания (Lazy propagation) - это методика оптимизации дерева отрезков, которая позволяет эффективно обновлять значения узлов при изменении или модификации элементов входного массива.

6. Отдельная структура node с двумя полями для одновременного изменения на полуинтервале и подсчета функции на отрезке.

Таблица 1. Сравнение Дерева отрезков с Деревом Фенвика и Декартовым деревом и обычным массивом.

Структура данных	Операции	Преимущества	Недостатки
Дерево Отрезков	Операции на интервалах (сумма, минимум, максимум)	Эффективные операции на отрезках, логарифмическая сложность времени	Необходимость построения структуры, высокая сложность реализации
Декартово дерево	Операции: Добавление, удаление, поиск	Эффективные операции добавления, удаления, поиска случайного элемента, универсальность в операциях.	Более сложная реализация, большее использование памяти.
Дерево Фенвика	Операции: Обновление, запросы на сумму	Эффективность операций суммы на префиксах массива с логарифмической сложностью времени	Ограниченный набор операций, не подходит для всех задач
Массив	Доступ по индексу	Простота использования, быстрый доступ по индексу за O(1)	Ограниченный набор операций, не подходит для всех задач

Дерево отрезков — это мощная структура данных для работы с интервалами в массивах и находит применение в алгоритмах и задачах, где необходимо выполнять операции над интервалами с высокой производительностью и эффективностью.

Глава 2. Моделирование дерева отрезков

2.1 Базовое Дерево Отрезков

Для реализации дерева отрезков был реализован базовый класс SegmentTree, хранящий представление дерева в виде массива. В качестве массива был использован класс стандартной библиотеки std::vector. Для построения объекта данного класса в конструктор или в метод класса build передается уже заполненный значениями массив, а также функция на основе которой мы будем строить дерево отрезков. Дерево выводится в виде массива измененного дерева с помощью метода Print. Добавлен метод build, работающий в конструкторе для инициализации дерева.Также добавлены 2 метода класса для работы с самим деревом : update принимающий указатель на функцию на основе , которой строилось дерево, индекс значения которое нужно обновить и само значение; segment_query метод вычисляющий значение переданной в него функции на отрезке.

Сумма на отрезке: Пусть у нас есть массив a из n элементов, и мы хотим уметь делать с ним две операции: set(i, v) — присвоить элементу с индексом i значение v, sum(l, r) — найти сумму на отрезке от l до r−1.

Обратите внимание, что в запросе на сумму мы левую границу l берем включительно, а правая граница r — исключительно. Так мы будем делать во всех случаях когда говорим об отрезках.

Структура дерева отрезков. Давайте представим, что нам нужно построить дерево отрезков для следующего массива: [3,1,2,5,6,4,3,2].

Дерево отрезков будет устроено так. Это двоичное дерево, в листьях которого находятся элементы исходного массива, а в каждом внутреннем узле записана сумма чисел в его детях.

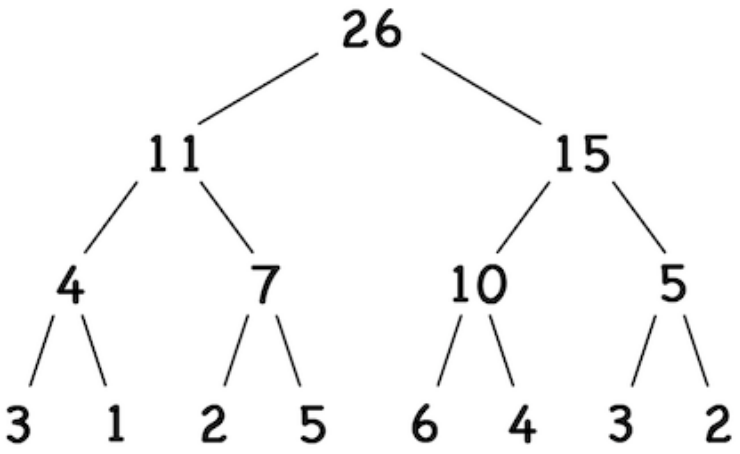

Рисунок 1. Представление дерева отрезков

Обратите внимание, что дерево получилось таким красивым, потому что длина массива была степенью двойки. Если длина массива не степень двойки, можно дополнить массив нулями до ближайшей степени двойки. При этом длина массива увеличится не больше, чем в два раза, поэтому асимптотика работы операций не изменится.

Теперь давайте разберем, как делать операции на таком дереве.

Операция set: Начнем с операции set. Когда изменяется элемент массива, нужно изменить соответствующее число в листе дерева отрезков, и далее пересчитать значения, которые от этого изменятся. Это те значения, которые находятся выше по дереву от измененного листа. Для пересчета просто заново посчитаем значение как сумму значений в детях.

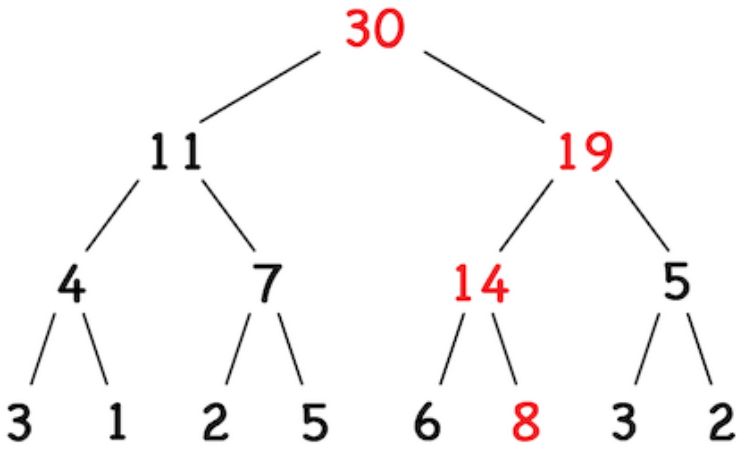

Рисунок 2. Пересчет дерева после изменения значения.

При совершении такой операции нам нужно пересчитать по одному узлу на каждом слое дерева отрезков. Слоев у нас всего log(n), значит время работы операции будет $O(\log n)$.

Операция sum: Теперь давайте разберем, как вычислять сумму на отрезке. Для этого давайте сначала посмотрим, что за числа написаны в узлах дерева отрезков. Заметим, что эти числа — это суммы на каких-то отрезках исходного массива.

При этом, например, число в корне — это сумма на всем массиве, а числа в листьях — это сумма на отрезке из одного элемента. Давайте попробуем собрать сумму на отрезке $[l..r)$ из этих, уже посчитанных, сумм. Для этого

запустим рекурсивный обход дерева отрезков. При этом будем обрывать рекурсию в двух ситуациях.

- Отрезок, соответствующий текущему узлу, не пересекается с отрезком $[l..r)$
- В этом случае все элементы в этом поддереве находятся вне области, в которой нам нужно посчитать сумму, поэтому глубже можно не идти.
- Отрезок, соответствующий текущему узлу, целиком вложен в отрезок $[l..r)$
- В этом случае все элементы в этом поддереве находятся в области, в которой нам нужно посчитать сумму, поэтому нам нужно добавить к ответу их сумму, которая записана в текущем узле.

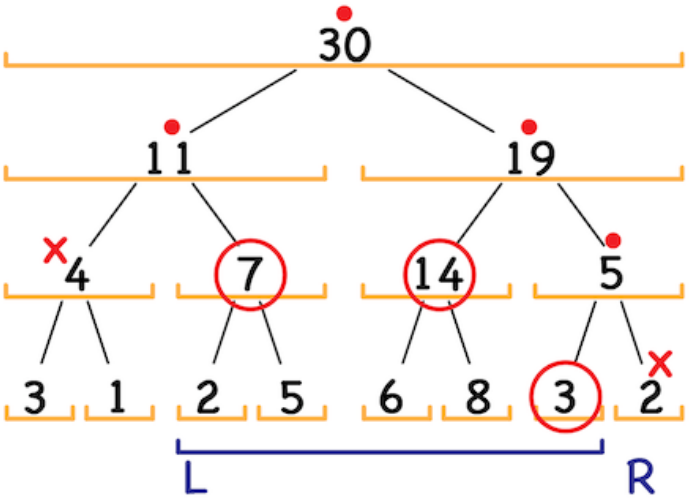

Рисунок 3. Отрезки на дереве

Здесь крестиком обозначены вершины, в которых рекурсия оборвалась по первому отсечению, а в кружок обведены вершины, в которых число добавилось к ответу, и рекурсия оборвалась по второму отсечению. Сколько же времени работает такой обход дерева? Чтобы ответить на этот вопрос, нужно понять, в скольких вершинах не случится ни одно из отсечений, и нам надо будет идти глубже по дереву. Каждый такой случай порождает нам новую ветку рекурсии. Оказывается, что таких вершин будет довольно мало. Дело в том, что для того, чтобы не сработало ни одно из отсечений, отрезок, соответствующий вершине дерева, должен пересекать отрезок запроса, но не содержаться в нем целиком. Это возможно только, если он содержит одну из границ отрезка $[l..r)$. Но на каждом слое дерева отрезков может быть не более одного отрезка, содержащего каждую из границ. Таким образом, вершин, в

которых не сработали отсечения, может быть не больше 2log(n), и, следовательно, общая асимптотика работы этой процедуры будет $O(\log(n))$.

```
// запрос модификации. Ему точно так же передается информация о текущей вершине
// дерева отрезков, а дополнительно указывается индекс меняющегося элемента, а
также его новое значение.
    virtual void update(T (*func)(T t1,T t2),int i , int v ,int x, int lx, int
rx){
        if (rx-lx==1){
            tree[x]=v;
            return;
        }
        int m = (lx+rx)/2;
        if (i<m){
            update(func,i,v,2*x+1,lx,m);
        } else {
            update(func,i,v,2*x+2,m,rx);
        }
        tree[x]=func(tree[2*x+1],tree[2*x+2]);
    }


// Вычисление значения функции на отрезке (рекурсивная функция)
    T segment_query(T (*func)(T t1,T t2),int left, int right, int x, int lx,
int rx) {
        if ((left >= rx|| lx>=right) && (func==Sum)) {
            return 0;   // Возвращаем нейтральное значение для операции
(например, для суммы это 0)
        }
        if ((left >= rx|| lx>=right) && (func==Min)) {
            return 1e9;   // Возвращаем нейтральное значение для операции
        }
        if ((left >= rx|| lx>=right) && (func==Max)) {
            return -1e9;   // Возвращаем нейтральное значение для операции
        }

        if (lx>=left && rx<=right) {
            return tree[x];  // Ответ находится в текущем узле
        }

        int mid = (rx + lx) / 2;
        // Рекурсивно спускаемся в левое поддерево
        T leftResult = segment_query(func, left,right,2*x+1,lx,mid );
        // Рекурсивно спускаемся в правое поддерево
        T rightResult = segment_query(func, left,right,2*x+2, mid ,rx);
        return func(leftResult, rightResult);    // Объединяем результаты
левого и правого поддеревьев
```

Листинг 1. Функции обновления значения в точке, построение дерева и запрос на отрезке.

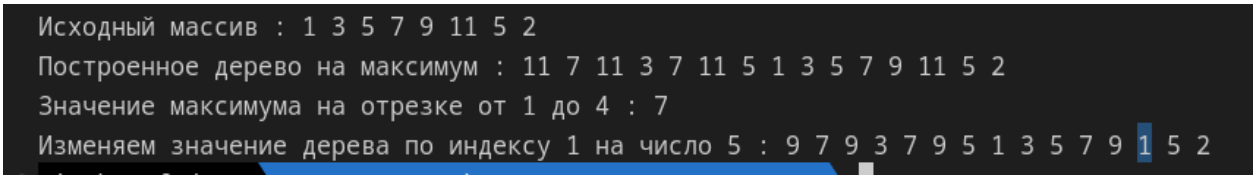

Рисунок 4. Результат работы обычного дерева отрезков на максимум.

2.2 Дерево Отрезков для некоммутативных операций.

В унаследованном от него классе Segment_group_tree добавлено поле NO_OPERATION для хранения проталкиваемых операций, позволяющих применять некоммутативный функции. Реализованы методы modify для прибавления значения на отрезке, а также добавлены методы lazy_modify (применение переданной функции к значению на отрезке) и lazy_get (получение значения на отрезке) реализованные с помощью функции propagate, реализующей технику проталкивания.

```
//Проталкивание
   void propagate(int x, int lx, int rx){
       if (SegmentTree<T>::tree[x] == NO_OPERATION) return;
       if (rx-lx == 1) return;
       SegmentTree<T>::tree[2*x+1]= SegmentTree<T>::tree[x];
       SegmentTree<T>::tree[2*x+2]=SegmentTree<T>::tree[x];
       SegmentTree<T>::tree[x]=NO_OPERATION;
   }

   //групповая операция для некоммутативных функций
   void lazy_modify(int l, int r, int v, int x, int lx, int rx){
       propagate(x,lx,rx);
       if (l >= rx|| lx>=r){
           return;
       }
       if (lx>=l && rx<=r) {
           SegmentTree<T>::tree[x]= v;
           return;
       }
       int mid = (rx + lx) / 2;
       // Рекурсивно спускаемся в левое поддерево
       lazy_modify(l,r,v,2*x+1,lx,mid);
       // Рекурсивно спускаемся в правое поддерево
       lazy_modify(l,r,v,2*x+2,mid,rx);
   }

   T lazy_get(int i ,int x, int lx, int rx){
       propagate(x, lx, rx);
       if (rx-lx==1){
           return SegmentTree<T>::tree[x];
       }
       int m = (lx+rx)/2;
       T res;
       if (i<m){
           res = lazy_get(i,2*x+1,lx,m);

       } else {
           res = lazy_get(i,2*x+2,m,rx);
       }
       return res;
```

Листинг 2. Функции обновления значений на отрезке и запрос на отрезке.

```
Исходный массив : 1 3 5 7 9 11 5 2
Построенное дерево отрезков на сумму : 43 16 27 4 12 20 7 1 3 5 7 9 11 5 2
Присвоение числа 5 на отрезке от 2 до 8
Значение на 5ой позиции : 5
```

Рисунок 5. Результат работы дерева отрезков с проталкиванием на сумму.

2.3 Дерево Отрезков с отложенными изменениями на отрезке и вычислением значений функции

Для выполнения самой главной части курсовой работы был реализован класс Segment_Lazy_tree унаследованный от базового класса. В нем были реализованы групповые операции изменения на отрезке , и подсчета функции на отрезке с использованием проталкивания. Для этого были добавлены поля set и min для хранения модификации и значений функции, а также добавлены поля-флаги NO_OPERATION и NEUTRAL_ELEMENT обозначающие необходимость применения модификации и нейтральный элемент для вычисляемой функции на отрезке.

Прибавление и минимум: Мы научились обрабатывать запросы вида «изменение на отрезке» и «значение в точке». Давайте модифицируем дерево отрезков так, чтобы оно могло обрабатывать запросы «изменение на отрезке» и «значение функции на отрезке».

Возьмём для рассмотрения такие операции:add(l, r, x) — прибавить ко всем a[i] ($l \le i < r$) значение x, min(l, r) — найти минимум из всех a[i] ($l \le i < r$).

Для начала, построим дерево отрезков для минимума.

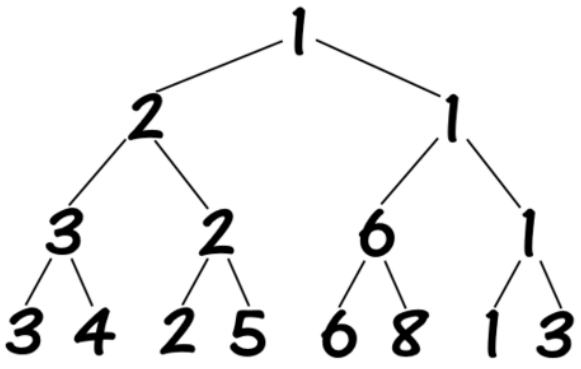

Рисунок 6. Дерево отрезков для операции min, построенное над массивом [3, 4, 2, 5, 6, 8, 1, 3].

Как делать операции на отрезке в таком дереве? Будем в каждой вершине хранить две величины (минимум на отрезке; величина, прибавленная на отрезке). Пусть нам нужно обработать запрос add(l, r, x). Будем действовать методом, схожим с прибавлением на отрезке: если текущая вершина полностью внутри отрезка из запроса, то к обоим значениям в текущей вершине прибавим величину *x ;* Если отрезок, который покрывается текущей вершиной, не пересекается с отрезком из запроса, то выйти из текущей вершины.

Каждый раз, когда мы будем выходить из вершины (которую мы не меняли), необходимо пересчитать минимум в ней. Он будет равен минимуму из значений детей.

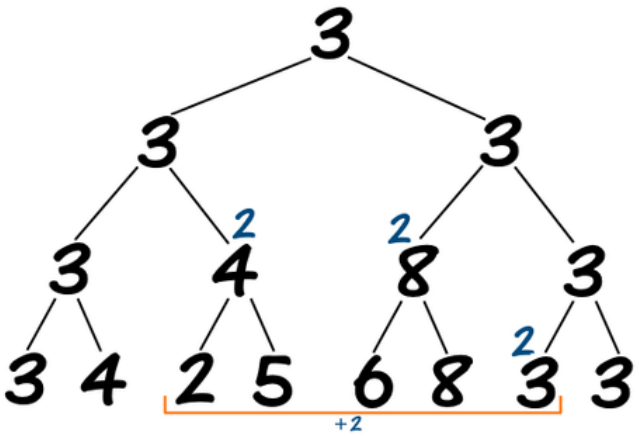

Рисунок 7. Так выглядит обработка запроса add(3, 8, 2).

Как можно заметить, в некоторых вершинах, например в четвертом листе, остались старые (необновленные) значения. Настоящее значение в вершине — это минимум в вершине плюс сумма прибавлений от предка этой вершины до корня дерева (изменения только в этих узлах ведёт к изменению текущего минимума).

Этот факт дает представление о том, как отвечать на запрос min(l, r). Будем рекурсивно обходить дерево и поддерживать сумму на пути от корня до предка текущей вершины. Возьмем минимум по значениям минимум в

вершине плюс сумма от неё до корня во всех вершинах, отрезки которых составляют отрезок $[l,r)$, (как в обычном запросе к дереву отрезков) — это и будет являться ответом.

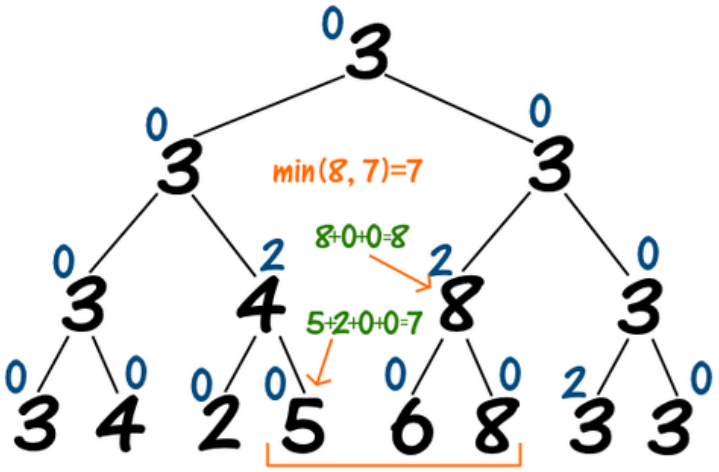

Рисунок 8. На картинке проиллюстрирован ответ на запрос min(4, 7).

Добавление проталкиваний: Если операция в запросе modify не коммутативна, то воспользуемся следующей техникой. Будем сохранять порядок операций, проталкивая старые операции в глубь дерева. Эта техника была рассмотрена на прошлом шаге. Так же, при входе в вершину будем проталкивать изменение в детей, а при выходе из вершины будем пересчитывать значение от детей.

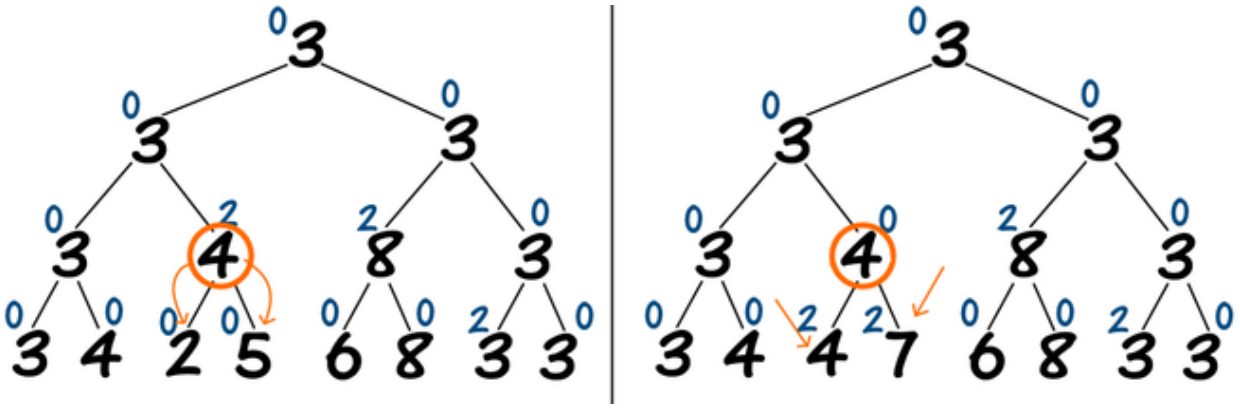

Рисунок 9. Так происходит проталкивание из выделенной вершины.

```
 //операция изменяющаяя отрезок
T op_modify(T a,T b, T len){
    if (b==NO_OPERATION) return a;
    return b;
}
//операция запроса на отрезке
T op_sum(T a,T b){
    return ::min(a,b);
}
```

```
    //Проталкивание
    void propagate(int x, int lx, int rx){
        if (tree[x].set == NO_OPERATION || rx-lx==1) return;
        int m = (lx+rx)/2;
        tree[2*x+1].set = op_modify(tree[2*x+1].set,tree[x].set,1);
        tree[2*x+1].min = op_modify(tree[2*x+1].min,tree[x].set,m-lx);
        tree[2*x+2].set = op_modify(tree[2*x+2].set,tree[x].set,1);
        tree[2*x+2].min = op_modify(tree[2*x+2].min, tree[x].set,rx-m);
        tree[x].set=NO_OPERATION;
    }
```

Листинг 3. Функция проталкивания и пример функций модификаций и запроса на отрезке

```
//Присваивание на отрезке (любая операция изменяющая отрезок)
    void mult(int l, int r, T v, int x, int lx, int rx){
        propagate(x, lx, rx);
        if (l >= rx || lx>=r){
            return;
        }
        if (lx>=l && rx<=r) {
            tree[x].set = op_modify(tree[x].set,v,1);
            tree[x].min = op_modify(tree[x].min,v,rx-lx);

            return;
        }
        int mid = (rx + lx) / 2;
        // Рекурсивно спускаемся в левое поддерево
        mult(l, r, v,2*x+1, lx,mid);
        // Рекурсивно спускаемся в правое поддерево
        mult(l, r, v,2*x+2,mid, rx);
        tree[x].min=op_sum(tree[2*x+1].min,tree[2*x+2].min);

    }


    //Запрос на отрезке
    T sum(int l, int r, int x, int lx, int rx){
        propagate(x, lx, rx);
        if (l >= rx || lx>=r){
            return NEUTRAL_ELEMENT;
        }
        if (lx>=l && rx<=r) {

            return tree[x].min;
        }
        int mid = (rx + lx) / 2;
        // Рекурсивно спускаемся в левое поддерево
        T m1 = sum(l, r, 2*x+1, lx,mid);
        // Рекурсивно спускаемся в правое поддерево
        T m2 = sum(l, r, 2*x+2,mid, rx);
        return op_sum(m1,m2);
    }
```

Листинг 4. Одновременные обновления значений на отрезке и запрос на отрезке с помощью отложенного выполнения.

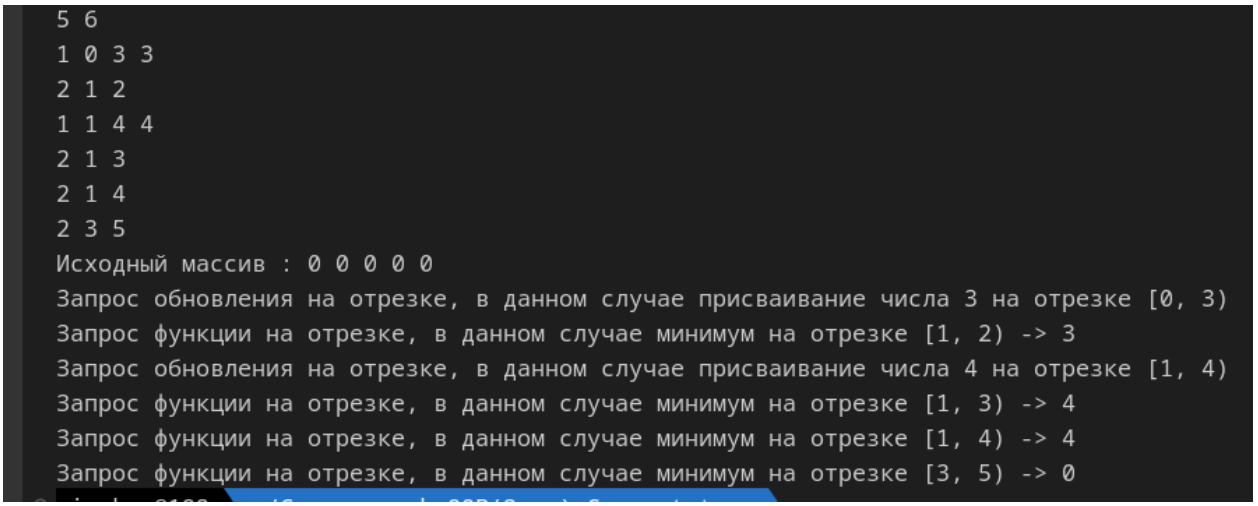

Рисунок 10. Результат работы дерева отрезков с проталкиванием и одновременными модификациями и подсчетами функции на отрезке.

Заключение

В ходе данной курсовой работы было исследовано дерево отрезков - эффективная структура данных, предназначенная для решения задач связанных с обработкой интервалов на массивах или последовательностях. Дерево отрезков является одной из самых популярных и мощных структур данных, применяемых в различных областях информатики и программирования.

Одним из ключевых преимуществ дерева отрезков является его способность решать задачи, связанные с операциями над интервалами, такими как поиск минимума, максимума, суммы или обновление значений на определенном отрезке. Благодаря этим операциям, дерево отрезков может успешно применяться в таких областях, как обработка изображений, графические редакторы, базы данных, системы управления данными и других приложениях, где требуется эффективное управление интервалами.

Оценка дерева отрезков по производительности и памяти является важным аспектом при его использовании. Временная сложность основных операций дерева отрезков составляет O(log n), где n - количество элементов в массиве или последовательности. Это делает дерево отрезков очень эффективным для обработки больших объемов данных. Однако, стоит учитывать, что построение дерева отрезков занимает O(n) времени и требует O(n) памяти, что может быть значительным недостатком при работе с большими массивами.

Таким образом, дерево отрезков занимает важное место среди других структур данных благодаря своей эффективности и возможности решать широкий спектр задач, связанных с интервалами. Однако, перед использованием дерева отрезков необходимо тщательно оценить требуемую производительность и объем памяти, так как построение и использование дерева отрезков может быть затратным процессом для больших объемов данных

Список литературы

1. Chen, S-D. "Two-Dimensional Segment Trees and Their Applications." Journal of Information Science and Engineering. - 2009. - Volume 25, Issue 4. - Pp. 1421-1437. ISSN 1016-2364
2. Gupta, A. Data Structures Using C++ / A. Gupta. - New Delhi: Pearson Education India, 2012. - 500 p. ISBN 9788131791041
3. Weiss, M. A. Data Structures and Algorithm Analysis in C++ / M. A. Weiss. - Boston: Pearson, 2014. - 656 p. ISBN 978-0132847377
4. Rao, S. S. Introduction to Data Structures and Algorithms / S. S. Rao. - New Delhi: Pearson Education India, 2009. - 700 p. ISBN 978-8131704219
5. Cormen, T. H. Introduction to Algorithms / T. H. Cormen, C. E. Leiserson, R. L. Rivest. - Cambridge, MA: The MIT Press, 2009. - 1312 p. ISBN 978-0262033848

Приложение

```
#include <climits>
#include <cstdint>
#include <iostream>
#include <bits/stdc++.h>
#include <type_traits>
#include <vector>
#include <cstdint>
#include <limits>


using namespace std;
int MOD =1e9+7;


// Функции для дерева отрезков
int Sum(int a, int b) {
    return a + b;
}

int Max(int a, int b){
    if (a>=b) return a;
    else return b;
}

int Min(int a, int b){
    if (a<=b) return a;
    else return b;
}

int Set(int a, int b){
    return b;
}


template<class T>
class SegmentTree {
protected:
    vector<T> tree; // массив, представляющий дерево отрезков
    int size; // размер исходного массива

    void init(int n){
        size = 1;
        while (size<n) size*=2;
        tree.assign(2*size-1,0);
    }

    // запрос модификации. Ему точно так же передается информация о текущей
вершине
    // дерева отрезков, а дополнительно указывается индекс меняющегося
элемента, а также его новое значение.
    virtual void update(T (*func)(T t1,T t2),int i , int v ,int x, int lx, int
rx){
        if (rx-lx==1){
            tree[x]=v;
            return;
        }
        int m = (lx+rx)/2;
        if (i<m){
            update(func,i,v,2*x+1,lx,m);
        } else {
            update(func,i,v,2*x+2,m,rx);
```

```
        }
        tree[x]=func(tree[2*x+1],tree[2*x+2]);
    }


    // Рекурсивная функция для построения дерева отрезков
    virtual void build(vector<T>& arr,T (*func)(T t1,T t2), int x, int lx, int
rx){
        if (rx-lx==1){
            if (lx<arr.size())
                tree[x]=arr[lx];
        } else {
            int m = (rx+lx)/2;
            build(arr,func,2*x+1,lx,m);
            build(arr,func,2*x+2,m,rx);
            tree[x]=func(tree[2*x+1],tree[2*x+2]);
        }
    }

    virtual void build(vector<T>& arr,T (*func)(T t1,T t2)) {
        init(arr.size());
        build(arr,func,0,0,size);
    }


    // Вычисление значения функции на отрезке (рекурсивная функция)
    T segment_query(T (*func)(T t1,T t2),int left, int right, int x, int lx,
int rx) {
        if ((left >= rx|| lx>=right) && (func==Sum)) {
            return 0;   // возвращаем нейтральное значение для операции
(например, для суммы это 0)
        }
        if ((left >= rx|| lx>=right) && (func==Min)) {
            return 1e9;   // возвращаем нейтральное значение для операции
        }
        if ((left >= rx|| lx>=right) && (func==Max)) {
            return -1e9;   // возвращаем нейтральное значение для операции
        }

        if (lx>=left && rx<=right) {
            return tree[x];  // Ответ находится в текущем узле
        }

        int mid = (rx + lx) / 2;
        // Рекурсивно спускаемся в левое поддерево
        T leftResult = segment_query(func, left,right,2*x+1,lx,mid );
        // Рекурсивно спускаемся в правое поддерево
        T rightResult = segment_query(func, left,right,2*x+2, mid ,rx);
        return func(leftResult, rightResult);    // Объединяем результаты
левого и правого поддеревьев
    }


public:

    // Конструктор класса
    SegmentTree() {}
    SegmentTree(vector<T>& arr, T (*func)(T t1,T t2)) {
        build(arr,func); // построение дерева отрезков
    }

    T segment_query(T (*func)(T t1,T t2),int left, int right){
        return segment_query(func, left, right,0,0,size);
    }
```

```
    virtual void update(T (*func)(T t1,T t2),int i, int v){
        update(func,i,v,0,0,size);
    }
    const vector<T> &get_ST() const {return tree;}
    int get_n(){return size;}

    virtual void Print(){
        vector<T> v = tree;
        for (int n : v){cout << n << ' ';};
        cout<<endl;
    }


};


template<class T>
class Segment_group_tree : public SegmentTree<T> {
private:
    T NO_OPERATION= numeric_limits<T>::min();

    //Прибавление на отрезке
    void modify(T (*func)(T t1,T t2),int l, int r, int v, int x, int lx, int
rx){
        if (l >= rx|| lx>=r){
            return;
        }
        if (lx>=l && rx<=r) {
            SegmentTree<T>::tree[x]=func(SegmentTree<T>::tree[x], v);
            return;
        }
        int mid = (rx + lx) / 2;
        // Рекурсивно спускаемся в левое поддерево
        modify(func,l,r,v,2*x+1,lx,mid);
        // Рекурсивно спускаемся в правое поддерево
        modify(func,l,r,v,2*x+2,mid,rx);
    }

    T get(T (*func)(T t1,T t2),int i ,int x, int lx, int rx){
        if (rx-lx==1){
            return SegmentTree<T>::tree[x];
        }
        int m = (lx+rx)/2;
        if (i<m){
            return func(get(func,i,2*x+1,lx,m),SegmentTree<T>::tree[x]);
        } else {
            return func(get(func,i,2*x+2,m,rx),SegmentTree<T>::tree[x]);
        }
    }

    //Проталкивание
    void propagate(int x, int lx, int rx){
        if (SegmentTree<T>::tree[x] == NO_OPERATION) return;
        if (rx-lx == 1) return;
        SegmentTree<T>::tree[2*x+1]= SegmentTree<T>::tree[x];
        SegmentTree<T>::tree[2*x+2]=SegmentTree<T>::tree[x];
        SegmentTree<T>::tree[x]=NO_OPERATION;
    }

    //групповая операция для некоммутативных функций
    void lazy_modify(int l, int r, int v, int x, int lx, int rx){
        propagate(x,lx,rx);
        if (l >= rx|| lx>=r){
```

```
            return;
        }
        if (lx>=l && rx<=r) {
            SegmentTree<T>::tree[x]= v;
            return;
        }
        int mid = (rx + lx) / 2;
        // Рекурсивно спускаемся в левое поддерево
        lazy_modify(l,r,v,2*x+1,lx,mid);
        // Рекурсивно спускаемся в правое поддерево
        lazy_modify(l,r,v,2*x+2,mid,rx);
    }

    T lazy_get(int i ,int x, int lx, int rx){
        propagate(x, lx, rx);
        if (rx-lx==1){
            return SegmentTree<T>::tree[x];
        }
        int m = (lx+rx)/2;
        T res;
        if (i<m){
            res = lazy_get(i,2*x+1,lx,m);

        } else {
            res = lazy_get(i,2*x+2,m,rx);
        }
        return res;
    }

public:

    Segment_group_tree(vector<T>& arr, T (*func)(T T1, T
T2)):SegmentTree<T>(arr, func){};

    void modify(T (*func)(T t1,T t2),int l, int r, int v){
        return modify(func,l,r,v,0,0,SegmentTree<T>::size);
    }
    T get(T (*func)(T t1,T t2),int i){
        return get(func,i, 0, 0, SegmentTree<T>::size);
    }
    //групповая операция для некоммутативных функций
    void lazy_modify(int l, int r, int v){
        return lazy_modify(l, r, v, 0, 0, SegmentTree<T>::size);
    }

    T lazy_get(int i){
        return lazy_get(i, 0, 0, SegmentTree<T>::size);
    }


};


using namespace std;

template <class T>
class Segment_Lazy_tree : public SegmentTree<T>{
public:
    struct node{
        T set; //хранение модификаций
        T min; //хранения значение функции
    };

    T NO_OPERATION=get_min_value();
```

```
T NEUTRAL_ELEMENT=get_max_value();

vector<node> tree;
int size;

vector<bool> lazy;

T get_min_value() {
    return numeric_limits<T>::min();
}
T get_max_value() {
    return numeric_limits<T>::max();
}
//операция изменяющаяя отрезок
T op_modify(T a,T b, T len){
    if (b==NO_OPERATION) return a;
    return b;
}
//операция запроса на отрезке
T op_sum(T a,T b){
    return ::min(a,b);
}
 // Рекурсивные функции для построения дерева отрезков

void init(int n){
    size = 1;
    while (size<n) size*=2;
    tree.resize(2*size-1);
}

void build(vector<T>& arr, int x, int lx, int rx) {
    if (rx-lx==1){
        if (lx<arr.size()) {
            tree[x].min = arr[lx];
            tree[x].set = NO_OPERATION;
        }
        return;
    }

    int mid = (lx + rx) / 2;
    build(arr, 2 * x + 1, lx, mid);
    build(arr, 2 * x + 2, mid, rx);

    tree[x].min = op_sum(tree[2 * x + 1].min, tree[2 * x + 2].min);
    tree[x].set = NO_OPERATION;
}
void build(vector<T>& arr){
    init(arr.size());
    build(arr,0,0,size);

}


//Проталкивание
void propagate(int x, int lx, int rx){
    if (tree[x].set == NO_OPERATION || rx-lx==1) return;
    int m = (lx+rx)/2;
    tree[2*x+1].set = op_modify(tree[2*x+1].set,tree[x].set,1);
    tree[2*x+1].min = op_modify(tree[2*x+1].min,tree[x].set,m-lx);
    tree[2*x+2].set = op_modify(tree[2*x+2].set,tree[x].set,1);
    tree[2*x+2].min = op_modify(tree[2*x+2].min, tree[x].set,rx-m);
    tree[x].set=NO_OPERATION;
}
```

```
    //Присваивание на отрезке (любая операция изменяющая отрезок)
    void mult(int l, int r, T v, int x, int lx, int rx){
        propagate(x, lx, rx);
        if (l >= rx || lx>=r){
            return;
        }
        if (lx>=l && rx<=r) {
            tree[x].set = op_modify(tree[x].set,v,1);
            tree[x].min = op_modify(tree[x].min,v,rx-lx);

            return;
        }
        int mid = (rx + lx) / 2;
        // Рекурсивно спускаемся в левое поддерево
        mult(l, r, v,2*x+1, lx,mid);
        // Рекурсивно спускаемся в правое поддерево
        mult(l, r, v,2*x+2,mid, rx);
        tree[x].min=op_sum(tree[2*x+1].min,tree[2*x+2].min);

    }


    //запрос на отрезке
    T sum(int l, int r, int x, int lx, int rx){
        propagate(x, lx, rx);
        if (l >= rx || lx>=r){
            return NEUTRAL_ELEMENT;
        }
        if (lx>=l && rx<=r) {

            return tree[x].min;
        }
        int mid = (rx + lx) / 2;
        // Рекурсивно спускаемся в левое поддерево
        T m1 = sum(l, r, 2*x+1, lx,mid);
        // Рекурсивно спускаемся в правое поддерево
        T m2 = sum(l, r, 2*x+2,mid, rx);
        return op_sum(m1,m2);

    }



public:
    Segment_Lazy_tree<T>(vector<T>& arr) : lazy(size,false){
        build(arr); // построение дерева отрезков
    }


    void mult(int l, int r, int v){
        return mult(l,r,v,0,0,size);
    }

    T sum(int l, int r){
        return sum(l,r,0,0,size);
    }
    void Print(){
        vector<node> v = tree;
        for (auto n : v){cout << "{"<<n.set<<","<<n.min<<"}" << ' ';};
        cout<<endl;
    }

};
```

```
int main() {

    /*vector<int> arr = {1, 3, 5, 7, 9, 11,5,2};    // Пример входного массива

    cout<<"Исходный массив : ";
    for (auto n : arr){cout << n << ' ';};
    cout<<endl;

    cout<<"Построенное дерево на максимум : ";
    SegmentTree<int> segTree(arr,Max); // Создаем дерево отрезков на основе
массива и любой ассоциативной функции
    segTree.Print();
    cout<<"значение максимума на отрезке от 1 до 4 : ";
    cout<<segTree.segment_query(Max, 1,4)<<endl;
    segTree.update(Max, 5, 1);
    cout<<"Изменяем значение дерева по индексу 5 на число 1 : ";
    segTree.Print();
    */


    //Групповое ДО
    /*
    Segment_group_tree<int> seg_g_Tree(arr,Sum);
    cout<<"Построенное дерево отрезков на сумму : ";
    seg_g_Tree.Print();
    cout<<"Присвоение числа 5 на отрезке от 2 до 8"<<endl;
    seg_g_Tree.lazy_modify(2, 8, 5);
    cout<<"Значение на 5ой позиции : ";
    cout<<seg_g_Tree.lazy_get(5)<<endl;
    */

    //Групповое ДО с отложенными оперциями

    ios::sync_with_stdio(false);
    int n,m;
    cin>>n>>m;
    vector<int> arr(n,0);
    Segment_Lazy_tree<int> segTree(arr);
    cout<<"Исходный массив : ";
    for (auto n : arr){cout << n << ' ';};
    cout<<endl;

    for (int t = 0; t < m ;t++){
        int c;
        cin >> c;
        if (c==1){

            int l,r,v;
            cin>>l>>r>>v;
            cout<<"Запрос обновления на отрезке, в данном случае присваивание
числа "<<v<<" на отрезке ";
            cout<<"["<<l<<", "<<r<<")"<<endl;
            segTree.mult(l,r,v);
```

```
        } else {
            int l,r;
            cin >> l >> r;
            cout<<"Запрос функции на отрезке, в данном случае минимум на
отрезке ";
            cout<<"["<<l<<", "<<r<<") -> ";
            cout << segTree.sum(l, r) << "\n";

        }

    }

    return 0;
}
```

Листинг 4. Код всех классов дерева отрезков.