

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«МИРЭА – Российский технологический университет»**

## РТУ МИРЭА

**Институт кибернетики**

**Кафедра высшей математики**

### КУРСОВАЯ РАБОТА

### по дисциплине

«Программирование**»**

**Тема курсовой работы**

# «Моделирование дерева отрезков»

Студенты группы КТСО-03-19 *Новоселов А.И., Кулиш А.А.*

Руководитель курсовой работы *Шерстнев Е.В.*

*Доцент кафедры Высшей математики Института кибернетики*

*к.ф.-м.н.*

Работа представлена к защите «\_\_»\_\_\_\_\_\_\_2020 г.

«Допущен к защите» «\_\_»\_\_\_\_\_\_\_2020г.

МОСКВА — 2020

Федеральное государственное бюджетное образовательное учреждение высшего образования

МИНОБРНАУКИ РОССИИ



**«МИРЭА – Российский технологический университет»**

## РТУ МИРЭА

Институт кибернетики

Кафедра высшей математики

**Утверждаю**

Заведующий

кафедрой\_\_\_\_\_\_\_\_\_*Ю.И.Худак*

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_2020г.

**ЗАДАНИЕ**

**на выполнение курсовой работы по** **дисциплине** «Программирование»

Студенты *Новоселов А.И., Кулиш А.А.*  Группа *КТСО-03-19*

1. **Тема: «Моделирование дерева отрезков»**
2. **Исходные данные:** 
   1. Реализовать класс «Дерево отрезков»
   2. Реализовать необходимые для класса функции: конструкторы, деструкторы, нужные методы
   3. Реализовать поиск значения функции на отрезке
   4. Реализовать групповое изменение значений в определенном отрезке
   5. Для выполнения групповых операций реализовать отложенное выполнение команд
3. **Перечень вопросов, подлежащих разработке, и обязательного графического материала:**

Продемонстрировать работу дерева отрезков на определенных полуинтервалах

1. **Срок представления к защите курсовой работы:** **до** «6» июня 2020 г.

Задание на курсовую

работу выдал «\_\_» \_\_\_\_\_\_ 2020 г. *\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (Шерстнев Е.В.)*

Задание на курсовую

работу получил «\_\_» \_\_\_\_\_\_ 2020 г. *\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (Новоселов А.И.)*

Задание на курсовую

работу получил «\_\_» \_\_\_\_\_\_ 2020 г. *\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (Кулиш А.А.)*

Оглавление Введение 2

Теоретическая часть 3

Основная часть 7

a) Реализация 7

б) Интерфейс программы 13

Заключение 16

Список использованной литературы 16

Приложение 16

1

Введение

Человечество производит огромное количество информации. Эту информацию необходимо не только хранить, но и обрабатывать, изменять, собирать статистику «на лету». Подтверждением этих слов может служить любая база данных, поисковая система, социальная сеть. Для эффективной работы с информацией были разработаны разнообразные структуры данных. Поскольку значение понятия «эффективность» зависит от специфики решаемой задачи, то и предназначение, и возможности этих структур значительно разнятся. Некоторые из них предназначены для эффективного хранения данных, другие же позволяют собирать статистическую информацию или изменять данные с минимальными затратами ресурсов. Нас будут интересовать последние. Самая распространенная структура данных — дерево. Существует большое число различных деревьев. Эти структуры позволяют собирать статистику на отрезках и изменять отдельные элементы. Практически все эти деревья предназначены для данных, на которых можно ввести линейный порядок. Дело в том, что не так уж часто встречаются задачи, которые нельзя свести к одномерному случаю. Там же, где размерности действительно велики, данные обычно сильно разрежены. Многомерные деревья нашли свое применение в различных областях: файловых системах, геометрических алгоритмах, поисковых системах. Обратимся к многомерным массивам данных. В первую очередь, это изображения и видео, а также базы данных. Введение нескольких измерений может быть, как естественным (растровые изображения и видео), так и искусственным (базы данных). В последнем случае дополнительные измерения позволяют точнее указывать интересующую выборку информации. Специфика задач обработки многомерных массивов данных может сильно разниться. Размерность может быть, как небольшой (изображения), так и значительной (базы данных). Кроме того, в последнем случае данные скорее всего будут разреженными. Все это необходимо учитывать разработке или выборе существующей структуры данных для решения конкретной задачи. Широко известны квадродеревья, кд-деревья, обобщения дерева отрезков и дерева Фенвика. Существующие структуры данных для работы с многомерными структурами данных позволяют эффективно получать статистическую информацию и изменять отдельные элементы. Дерево отрезков — это структура данных, которая позволяет эффективно реализовать операции следующего вида: нахождение суммы/минимума элементов массива в заданном отрезке, при этом дополнительно возможно изменение элементов массива: как изменение значения одного элемента, так и изменение элементов на целом подотрезке массива. Вообще, дерево отрезков — очень гибкая структура, и число задач, решаемых ей, теоретически неограниченно. Помимо приведённых выше видов операций с деревьями отрезков, также возможны и гораздо более сложные операции. В частности, дерево отрезков легко обобщается на большие размерности: например, для решения задачи о поиске суммы/минимума в некотором подпрямоугольнике данной матрицы. Важной особенностью деревьев отрезков является то, что они потребляют линейный объём памяти: стандартному дереву отрезков требуется порядка 4n элементов памяти для работы над массивом размера n.

2

**Теоретическая часть**

Дерево отрезков – это структура данных, которая позволяет эффективно (за асимптотику O(log2n) ) реализовать операции следующего вида:

• нахождение суммы/минимума элементов массива в заданном отрезке a[l...r], где l и r поступают на вход алгоритма

• изменение элементов массива: как изменение значения одного элемента, так и изменение элементов на целом подотрезке массива (разрешается присвоить всем элементам a[l...r] какое-либо значение, либо прибавить ко всем элементам массива какое-либо число).

Дерево отрезков для сумм. Пусть имеется массив a[0…n – 1]. Необходимо создать дерево отрезков, которое будет выполнять следующие две операции:

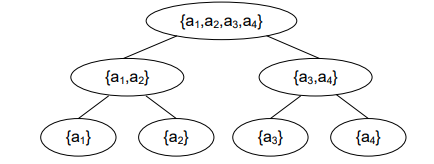
• по заданным l и r вычислить сумму a[l] + a[l + 1] + … + a[r – 1] + a[r];

• присваивание a[i] = x

Обе операции должны выполняться за время O(log2n). Вычислим и запомним где-нибудь сумму элементов всего массива, то есть отрезка a[0…n – 1]. Далее найдем сумму на двух половинках этого массива: a[0…n / 2] и a[n / 2 + 1…n – 1]. Каждую из этих двух половинок в свою очередь разобьём пополам, посчитаем и сохраним сумму на них, потом снова разобьём пополам, и так далее, пока текущий отрезок не достигнет длины 1. Начиная с [0…n – 1], каждый раз будем делить текущий отрезок пополам (пока его длина больше единицы), вызывая затем эту же процедуру от обеих половинок. Для каждого такого отрезка будем хранить сумму чисел на нём.

Можно говорить, что эти отрезки, на которых мы считали сумму, образуют дерево: корень этого дерева – отрезок, а каждая вершина имеет ровно двух сыновей (кроме вершин-листьев, у которых отрезок имеет длину 1). Отсюда и происходит название – "дерево отрезков".

Пример. Дерево отрезков из четырех элементов будет иметь следующий вид:



Построение дерева отрезков:

Процесс построения дерева отрезков по заданному массиву *a* можно делать эффективно следующим образом, снизу вверх: сначала запишем значения элементов *a*[i] в соответствующие листья дерева, затем на основе них посчитаем значения для вершин предыдущего уровня как сумму значений в двух листьях, затем аналогичным образом посчитаем значения для ещё одного уровня, и т.д. Удобно описывать эту операцию рекурсивно: мы запускаем процедуру построения от корня дерева отрезков, а сама процедура построения, если её вызвали не от листа, вызывает себя от каждого из двух сыновей и суммирует вычисленные значения, а если её вызвали от листа — то просто записывает в себя значение этого элемента массива. Асимптотика построения дерева отрезков составит, таким образом, O(n).

3

Обновление на отрезке

Дерево отрезков позволяет делать запросы, которые применяются к целым отрезкам подряд идущих элементов, причём выполнять эти запросы за то же время O(log n)

Прибавление на отрезке

Начнём рассмотрение деревьев отрезков такого рода с самого простого случая: запрос модификации представляет собой прибавление ко всем числам на некотором подотрезке *a*[l…r] некоторого числа *x*. Запрос чтения — по-прежнему считывание значения некоторого числа *a*[i].

Чтобы делать запрос прибавления эффективно, будем хранить в каждой вершине дерева отрезков, сколько надо прибавить ко всем числам этого отрезка целиком. Например, если приходит запрос "прибавить ко всему массиву *a*[0…n-1]  число 2", то мы поставим в корне дерева число 2. Тем самым мы сможем обрабатывать запрос прибавления на любом подотрезке эффективно, вместо того чтобы изменять все O(n) значений.

Если теперь приходит запрос чтения значения того или иного числа, то нам достаточно спуститься по дереву, просуммировав все встреченные по пути значения, записанные в вершинах дерева.

Присвоение на отрезке

Пусть теперь запрос модификации представляет собой присвоение всем элементам некоторого отрезка *a*[l…r]  некоторого значения p. В качестве второго запроса будем рассматривать считывание значения массива *a*[i].

Чтобы делать модификацию на целом отрезке, придётся в каждой вершине дерева отрезков хранить, покрашен ли этот отрезок целиком в какое-либо число или нет (и если покрашен, то хранить само это число). Это позволит нам делать "запаздывающее" обновление дерева отрезков: при запросе модификации мы, вместо того чтобы менять значения во множестве вершин дерева отрезков, поменяем только некоторые из них, оставив флаги "покрашен" для других отрезков, что означает, что весь этот отрезок вместе со своими подотрезками должен быть покрашен в этот цвет.

Итак, после выполнения запроса модификации дерево отрезков становится, вообще говоря, неактуальным — в нём остались недовыполненными некоторые модификации.

Например, если пришёл запрос модификации "присвоить всему массиву *a*[0…n-1]  какое-то число", то в дереве отрезков мы сделаем единственное изменение — пометим корень дерева, что он покрашен целиком в это число. Остальные же вершины дерева останутся неизменёнными, хотя на самом деле всё дерево должно быть покрашено в одно и то же число.

Предположим теперь, что в том же дереве отрезков пришёл второй запрос модификации — покрасить первую половину массива *a*[0…n/2]  в какое-либо другое число. Чтобы обработать такой запрос, мы должны покрасить целиком левого сына корня в этот новый цвет, однако перед тем как сделать это, мы должны разобраться с корнем дерева. Тонкость здесь в том, что в дереве должно сохраниться, что правая половина покрашена в старый цвет, а в данный момент в дереве никакой информации для правой половины не сохранено.

4

Выход таков: произвести проталкивание информации из корня, т.е. если корень дерева был покрашен в какое-либо число, то покрасить в это число его правого и левого сына, а из корня эту отметку убрать. После этого мы можем спокойно красить левого сына корня, не теряя никакой нужной информации. Обобщая, получаем: при любых запросах с таким деревом (запрос модификации или чтения) во

время спуска по дереву мы всегда должны делать проталкивание информации из текущей вершины в обоих её сыновей. Можно понимать это так, что при спуске по дереву мы применяем запаздывающие модификации, но ровно настолько, насколько это необходимо (чтобы не ухудшить асимптотику с O(log n)).

**Основная часть**

Реализация

*Include:*

Подключаем библиотеку «iostream» для организации ввода-вывода, заголовочный файл с объявлением класса и заголовочный файл пространства имён - std. Пространство подключаем, чтобы каждый раз не использовать “std::”.

SegmentTree.h:

Класс, который содержит все необходимые методы для построения дерева отрезков, поиска суммы и группового изменения значений на полуинтервале. Также, в нём содержатся методы для реализации отложенного выполнения команд.

В классе объявляется 5 переменных типа int и 3 int-овых одномерных статических массива.   
Переменные: *ln* – отвечает за длину массива *a[]*, подающегося на вход; *v* – индекс текущей вершины дерева; *tl* и *tr* – индексы левой и правой границы соответствующего отрезка на дереве; *h* – высота дерева (наибольший значимый бит в двоичном представлении числа *ln*).   
Массивы: *a[1000]* – массив, на основе которого будет строиться дерево; *tree[4000]* – дерево отрезков (если ln является степенью двойки, длина массива дерева должна быть в 2 раза больше длины исходного массива, в ином случае – в 4); d*[1000]* – массив, элементы которого хранят данные, по которым мы понимаем, нужны ли вычисления для элемента с таким же индексом в массиве *a*.

SegmentTree() и ~SegmentTree():

Конструктор и деструктор класса.

void SetParam(int, int, int, int, int[]):

Метод, принимающий на вход значения длины массива, текущей вершины дерева, индекса левой и правой границ текущего отрезка на дереве, и устанавливающий их. Также, инициализирует переменную *h* и массив *a[].*

5

void BuildTree(int[], int, int, int):

Реализует дерево отрезков, используя *a[], v, tl, tr*. В этом методе используется рекурсия.

**void** SegmentTree::BuildTree(**int** a[], **int** v, **int** tl, **int** tr)

{

**if** (tl == tr)

{

tree[v] = a[tl];

}

**else**

{

**int** tm = (tl + tr) / 2;

BuildTree(a, v \* 2, tl, tm);

BuildTree(a, v \* 2 + 1, tm + 1, tr);

tree[v] = tree[v \* 2] + tree[v \* 2 + 1];

}

}

Мы создаём и инициализируем переменную tm, которая является серединой выбранного отрезка, с границами *tl, tr*. После этого дважды вызываем метод *BuildTree*: для левой части (с границами *tl* и *tm*) и правой (*tm*+1 и *tr*). Таким образом, разница между *tl* и *tr* становится меньше с каждым вызовом метода, а на листах они становятся равны, и рекурсия завершается.

int GetSum(int, int, int, int, int):

Считает сумму элементов дерева на выбранном отрезке. Этот метод так же использует рекурсию. Как в предыдущем методе, мы создаём переменную tm, и при её помощи проходим по дереву. Кроме этого, есть два условия: 1) если границы отрезка суммы входят в границы дерева, тогда возвращаем значение текущего элемента дерева; 2) если границы отрезка суммы выходят за границы дерева, тогда не берём этот элемент в расчёт.

**int** SegmentTree::GetSum(**int** l, **int** r, **int** v, **int** tl, **int** tr)

{

**if** (l <= tl && tr <= r)

{

**return** tree[v];

}

**if** (tr < l || r < tl)

{

**return** 0;

}

**int** tm = (tl + tr) / 2;

**return** GetSum(l, r, v \* 2, tl, tm) + GetSum(l, r, v \* 2 + 1, tm + 1, tr);

}

6

int SignificantBit(int):

Этот метод используется для расчёта количества уровней дерева отрезков. Мы используем побитовый сдвиг. Поскольку побитовый сдвиг влево на *x* эквивалентен умножению числа на 2^*x*, мы можем вычислить наибольший значимый разряд в двоичном представлении длины данного массива.

void Build(int, int):

Построение дерева (снизу-вверх) при помощи метода *Push*.

void Push(int, int):

При передаче вершины дерева отрезков эта функция будет производить проталкивание информации из этой вершины в обоих её “сыновей” (сверху-вниз).

void Apply(int, int, int):

Функция для изменения значения вершины *v*

void Calc(int, int):

Функция для вычисления значения в вершине *v*

void IntervalModify(int, int, int):

Этот метод предназначен для изменения группы элементов дерева на определённом полуинтервале.  
  
В начале мы проверяем, не равно ли нулю прибавляемое значение. В случае, если равно, работа функции завершается. Далее, производим проталкивание для левого и правого детей левой и правой границы соответственно. Так же заводим булевые переменные, которые будут обозначать наличие изменений в левой и правой границах, и целочисленный счётчик высоты, при движении вверх по дереву.  
  
Далее запускаем два цикла. В теле которых содержатся проверки, вызывающие вспомогательные функции *Calc* и *Apply.*

void PrintTree():

Данный метод выводит дерево на экран, используя цикл for.

7

int main():

В начале функции мы объявляем и инициализируем необходимые переменные, а так же объявляем объект класса *SegmentTree: tree.* После этого поочерёдно вызываются функции *SetParam, BuildTree* и *PrintTree*, которые строят и выводят на экран дерево отрезков. Далее выводится сумма элементов дерева на заданном полуинтервале, это происходит после вызова функции *GetSum*. После чего следует проверка переменной *value* на ненулевое значение, в таком случае вызываются функции *Push, IntervalModify* и *Build*, которые обеспечивают групповое изменение элементов с соблюдением принципа отложенного выполнения команд на определённом полуинтервале.

Примеры запуска программы

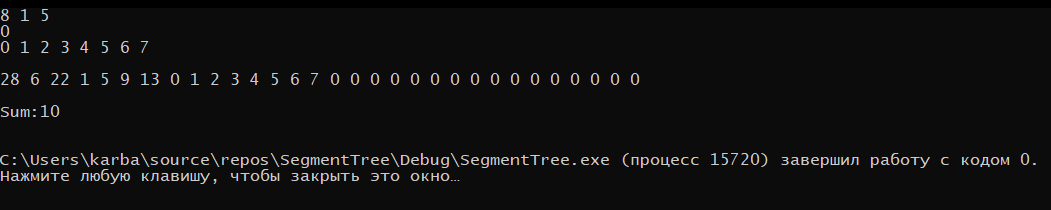


Рис.1

На рисунке 1 мы видим запуск программы. В первой строке задаётся длина исходного массива (8) и границы полуинтервала (1 и 5), с которым будет производиться работа. Во второй строке задано значение для изменения на полуинтервале (в данном примере 0). В третьей строке мы заполняем исходный массив.   
Далее строится дерево отрезков и выводится искомая сумма.

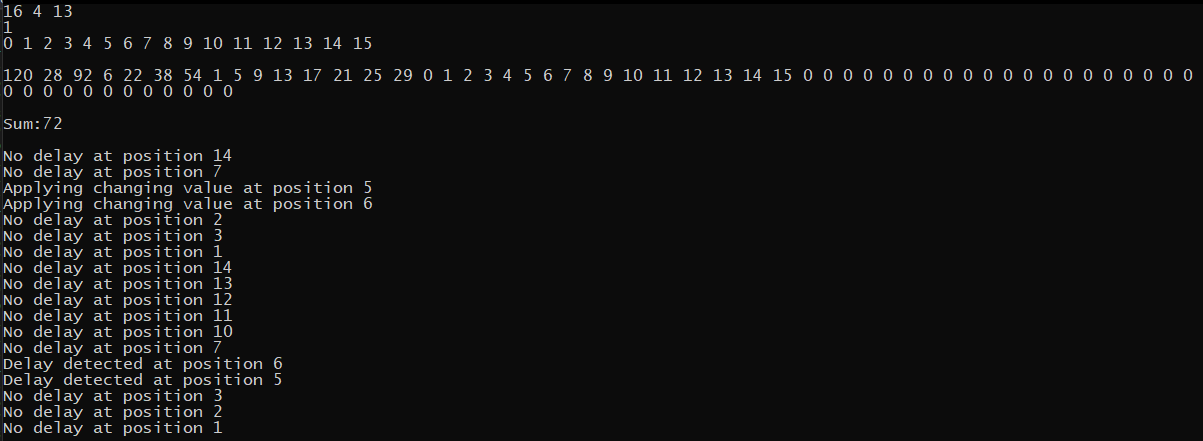


Рис.2

8

На 2 рисунке показан запуск программы с ненулевым значением для изменения на заданном полуинтервале. После построения дерева и расчёта суммы виден результат работы функции *IntervalSum*. Нужно отметить, что данный метод обходит дерево снизу вверх, поэтому сначала вызывается *Calc* для родителя 12-го элемента, то есть для 14-го, далее так же для 7-го (он является родителем 14-го). После чего, вызывается *Apply* для 5-го и 6-го. Затем, первый цикл заканчивается.

**Заключение**

В ходе этой работы был реализовали класс “Дерево отрезков”. Реализовали функцию поиска значений на отрезке, групповое изменение значений в определенном отрезке и реализовали отложенное выполнение команд для выполнения групповых операций, а также продемонстрировали работу дерева отрезков на определенных полуинтервалах.

Список использованной литературы

1. Герберт Шилдт. Самоучитель С++.: -М.: ООО “И. Д. Вильямс”, 2011. – 1056c.
2. Максим Ахмедов. Задача RMQ – 2. Дерево отрезков. [электронный ресурс]. – URL: [https://habr.com/ru/post/115026/](https://habr.com/ru/post/115026/%20%20%20)
3. Максим Иванов. Дерево отрезков. [электронный ресурс]. – URL:   
   <http://e-maxx.ru/algo/segment_tree>
4. Al.Cash. Efficient and easy segment trees. [электронный ресурс]. – URL:

<https://codeforces.com/blog/entry/18051>

9

Приложение

SegmentTree.h:

#**pragma once**

#include <iostream>

**class** SegmentTree

{

**public**:

SegmentTree();

~SegmentTree();

**void** SetParam(**int**, **int**, **int**, **int**, **int**[]);

**void** BuildTree(**int**[], **int**, **int**, **int**);

**int** GetSum(**int**, **int**, **int**, **int**, **int**);

**void** Build(**int**, **int**);

**void** Push(**int**, **int**);

**void** IntervalModify(**int**, **int**, **int**);

**void** PrintTree();

**private**:

**void** Calc(**int**, **int**);

**void** Apply(**int**, **int**, **int**);

**int** SignificantBit(**int**);

**int** ln, v, tl, tr, a[1000] = { 0 }, tree[4000] = { 0 }, h, d[1000] = { 0 };

};

SegmentTree.cpp:

#include <iostream>

#include "SegmentTree.h"

**using** **namespace** std;

SegmentTree::SegmentTree()

{

}

SegmentTree::~SegmentTree()

{

}

**void** SegmentTree::SetParam(**int** length, **int** current, **int** left, **int** right, **int** a[])

{

ln = length;

v = current;

tl = left;

tr = right;

h = SignificantBit(ln);

**for** (**int** i = 0; i < ln; i++)

cin >> a[i];

}

10

**void** SegmentTree::BuildTree(**int** a[], **int** v, **int** tl, **int** tr)

{

**if** (tl == tr)

{

tree[v] = a[tl];

}

**else**

{

**int** tm = (tl + tr) / 2;

BuildTree(a, v \* 2, tl, tm);

BuildTree(a, v \* 2 + 1, tm + 1, tr);

tree[v] = tree[v \* 2] + tree[v \* 2 + 1];

}

}

**int** SegmentTree::GetSum(**int** l, **int** r, **int** v, **int** tl, **int** tr)

{

**if** (l <= tl && tr <= r)

{

**return** tree[v];

}

**if** (tr < l || r < tl)

{

**return** 0;

}

**int** tm = (tl + tr) / 2;

**return** GetSum(l, r, v \* 2, tl, tm) + GetSum(l, r, v \* 2 + 1, tm + 1, tr);

}

**void** SegmentTree::Build(**int** tl, **int** tr)

{

**int** k = 2;

**for** (tl += ln, tr += ln - 1; tl > 1; k <<= 1)

{

tl >>= 1, tr >>= 1;

**for** (**int** i = tr; i >= tl; --i) Calc(i, k);

}

}

**void** SegmentTree::Push(**int** tl, **int** tr)

{

**int** s = h, k = 1 << (h - 1);

**for** (tl += ln, tr += ln - 1; s > 0; --s, k >>= 1)

**for** (**int** i = tl >> s; i <= tr >> s; ++i)

**if** (d[i] != 0)

{

Apply(i << 1, d[i], k);

Apply(i << 1 | 1, d[i], k);

d[i] = 0;

}

}

11

**int** SegmentTree::SignificantBit(**int** value)

{

**int** max\_int = 1 << 30;

**int** power = 30;

**while** (value < max\_int)

{

max\_int >>= 1;

--power;

}

**return** power;

}

**void** SegmentTree::Apply(**int** v, **int** value, **int** levlen)

{

a[v] = value \* levlen;

**if** (v < ln)

{

d[v] = value;

cout << "Applying changing value at position " << v << "\n";

}

}

**void** SegmentTree::Calc(**int** v, **int** levlen)

{

**if** (d[v] == 0)

{

tree[v] = tree[v << 1] + tree[v << 1 | 1];

cout << "No delay at position " << v << "\n";

}

**else**

{

tree[v] = d[v] \* levlen;

cout << "Delay detected at position " << v << "\n";

}

}

**void** SegmentTree::IntervalModify(**int** tl, **int** tr, **int** value)

{

Push(tl, tl + 1);

Push(tr - 1, tr);

**bool** cl = **false**, cr = **false**;

**int** k = 1;

**for** (tl += ln, tr += ln; tl < tr; tl >>= 1, tr >>= 1, k <<= 1)

{

**if** (cl)

Calc(tl - 1, k);

**if** (cr)

Calc(tr, k);

**if** (tl & 1)

Apply(tl++, value, k), cl = **true**;

**if** (tr & 1)

Apply(--tr, value, k), cr = **true**;

}

**for** (--tl; tr > 0; tl >>= 1, tr >>= 1, k <<= 1)

{

**if** (cl)

Calc(tl, k);

**if** (cr && (!cl || tl != tr))

Calc(tr, k);

}

}

12

**void** SegmentTree::PrintTree()

{

cout << "\n";

**for** (**int** i = 1; i < 4 \* ln; i++)

{

cout << tree[i] << " ";

}

cout << "\n\n";

}

**int** main()

{

**int** length, current = 1, left = 0, a[1000], lborder, rborder, value;

cin >> length >> lborder >> rborder >> value;

SegmentTree tree;

tree.SetParam(length, current, left, length - 1, a);

tree.BuildTree(a, current, left, length - 1);

tree.PrintTree();

cout << "Sum:" << tree.GetSum(lborder, rborder - 1, current, left, length - 1) << "\n\n";

**if** (value != 0)

{

tree.Push(lborder, rborder);

tree.IntervalModify(lborder, rborder, value);

tree.Build(lborder, rborder);

}

**return** 0;

}

13