Оглавление

Глава 1. Теория.

Необходимые сведения.

Условие задачи.

Реализовать класс «Декартово дерево по неявному ключу». Реализовать необходимые для класса функции: конструкторы, базовые методы слияния и разделения, методы добавления, вставки и удаления элемента. Реализовать поиск значения функции на отрезке. Реализовать групповое изменение значений в определенном отрезке. Для выполнения групповых операций реализовать отложенное выполнение команд.

Оптимизация операций над множеством данных.

При выполнении различных задач с данным необходимо понимать, каким образом следует хранить эти данные. Наиболее распространенный для этого инструмент – массив – обладает рядом недостатков, например, долгим временем удаления и вставки элемента в виду того, что данные хранятся непрерывно в памяти (асимптотика решения $O(N)$).

Двоичное дерево поиска.

Разрешить эти проблемы могут ассоциативные массивы, а точнее бинарные деревья поиска. Храня в каждой вершине такого дерева данные и ссылки на левого и правого детей, можно организовать данные в отсортированном на каждом уроне дерева порядке (левый ребенок – меньше текущей вершины, правый ребенок – больше текущей вершины), что позволит выполнять базовые операции быстрее, чем за $O(N)$. Однако, когда речь заходит о деревьях поиска, основной вопрос, который ставится перед структурой — скорость выполнения операций, вне зависимости от данных, хранящихся в ней, и последовательности их поступления. Так, двоичное дерево поиска дает гарантию, что поиск конкретного ключа в этом дереве будет выполняться за $O(H)$, где H — высота дерева. Но какой может быть эта высота — в точности не известно. При неблагоприятных обстоятельствах высота дерева легко может стать N (количество элементов в нем), и тогда дерево поиска вырождается в обычный список, что не дает преимущества по скорости выполнения операций. Для достижения такой ситуации достаточно добавлять в дерево поиска элементы от 1 до N в очереди возрастания. При стандартном алгоритме добавления в дерево получим результат, продемонстрированный на рисунке 1.

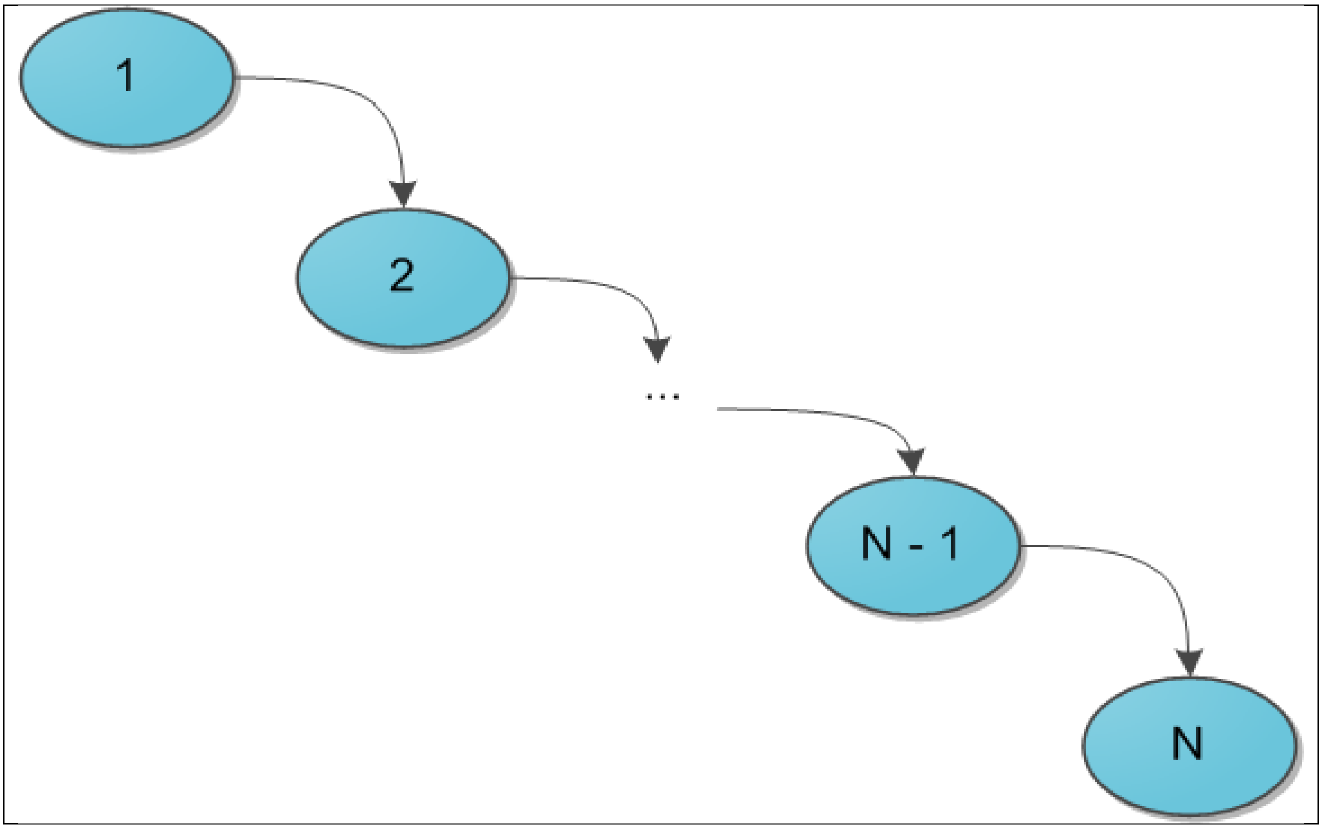

Рисунок 1. Организация данных в бинарном дереве поиска при несбалансированном добавлении вершин от 1 до *N*.

Было придумано огромное количество так называемых сбалансированных деревьев поиска — тех, в которых по мере существования дерева при каждой операции над ним поддерживается оптимальность максимальной глубины дерева. Оптимальная глубина имеет порядок $O(log_2N)$ — тогда тот же порядок имеет время выполнения каждого поиска в дереве. Структур данных, поддерживающих такую глубину, много, самые известные красно-черное дерево или АВЛ-дерево. Их отличительная черта в большинстве — трудная реализация, основанная на размере, большого числа различных случаев, в которых можно и запутаться. Своей же простотой и красотой выгодно отличается, наоборот, декартово дерево, и даже дает в некотором роде желанное логарифмическое время. Более детальное описание будет приведено в работе далее.

Куча.

Еще одна необходимая для дальнейшей работы структура данных – куча (heap) – представляет собой бинарное дерево, каждый узел в котором больше (если реализуется max-heap или меньше, если реализуется min-heap) своих дочерних узлов. Куча выстраивается согласно приоритету (величине) узлов. Это свойство необходимо для дальнейшей работы.

Декартово дерево

Для понимания концепции декартового дерева по неявному ключу в первую очередь следует ознакомиться с более простой его версией – декартовым деревом.

Декартово дерево является комбинацией двоичного дерева поиска и кучи. Оно представляет собой структуру данных, в которой каждый узел содержит два значения: ключ и приоритет. Ключи узлов упорядочены по свойствам двоичного дерева поиска, то есть для каждого узла все значения в его левом поддереве меньше ключа текущего узла, а значения в его правом поддереве больше ключа текущего узла. Приоритеты узлов определяются свойствами кучи, где для каждого узла приоритет его потомков не меньше приоритета самого узла (в случае max-heap) или не больше (в случае min-heap). При этом, присваивая каждому узлу случайный приоритет из широкого диапазона, можно добиться практически оптимально сбалансированного дерева (полученное декартово дерево с очень высокой, стремящейся к 100% при увеличении числа входных данных вероятностью, будет иметь высоту, не превосходящую $4log_2N$ согласно математическому ожиданию). А значит, хоть оно может и не быть идеально сбалансированным, время поиска ключа в таком дереве все равно будет порядка $O(log_2N)$. Оценка памяти для данной структуры - $O(N)$ (требуется хранить N вершин).

Такая структура хорошо подходит для хранения данных и реализации быстрых операций поиска и вставки элементов. Пример декартового дерева и его визуализация на декартовой системе координат (x – координата ключа вершины; y – координата приоритета вершины) приведены на рисунке 2.

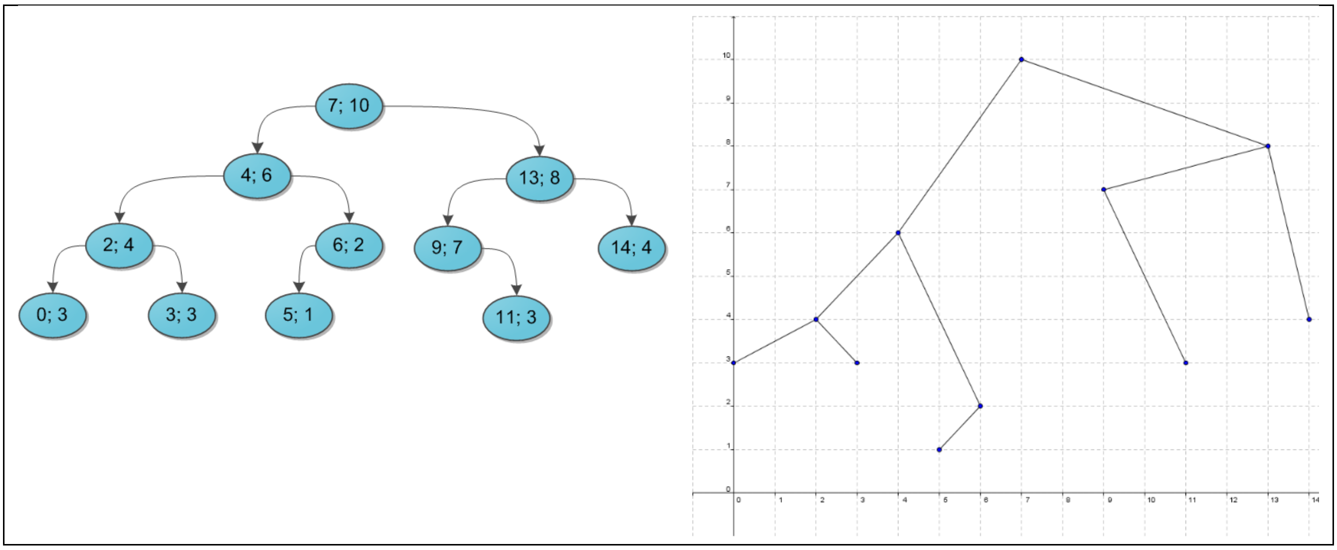

Рисунок 2. Визуализация произвольного декартового дерева в декартовой системе координат.

Декартово дерево по неявному ключу.

Разработанная на прошлом этапе структура может хранить любые пользовательские данные. Таким образом каждая вершина декартового дерева имеет 5 полей: ключ (произвольный индекс, указывающий на то, в каком порядке будут храниться данные) для построения дерева поиска по индексам, случайный приоритет для балансировки, пользовательские данные, которые требуется хранить, и указатели на левую и правую вершины.

Возможно рассмотреть вариант дерева, в котором ключи не задаются явно для каждой вершины. Но как тогда будут храниться данные? Чтобы ответить на этот вопрос такую структуру стоит расценивать как декартово дерево, в котором ключи все так же где-то имеются, но нам их не сообщили. Однако гарантируют, что для них, как полагается, выполняется условие двоичного дерева поиска. Тогда можно представить, что эти неизвестные ключи суть числа от 0 до N-1 и неявно расставить их по структуре дерева согласно порядку in-order обхода от наименьшего неявного ключа. Результат для произвольного дерева приведен на рисунке 3.

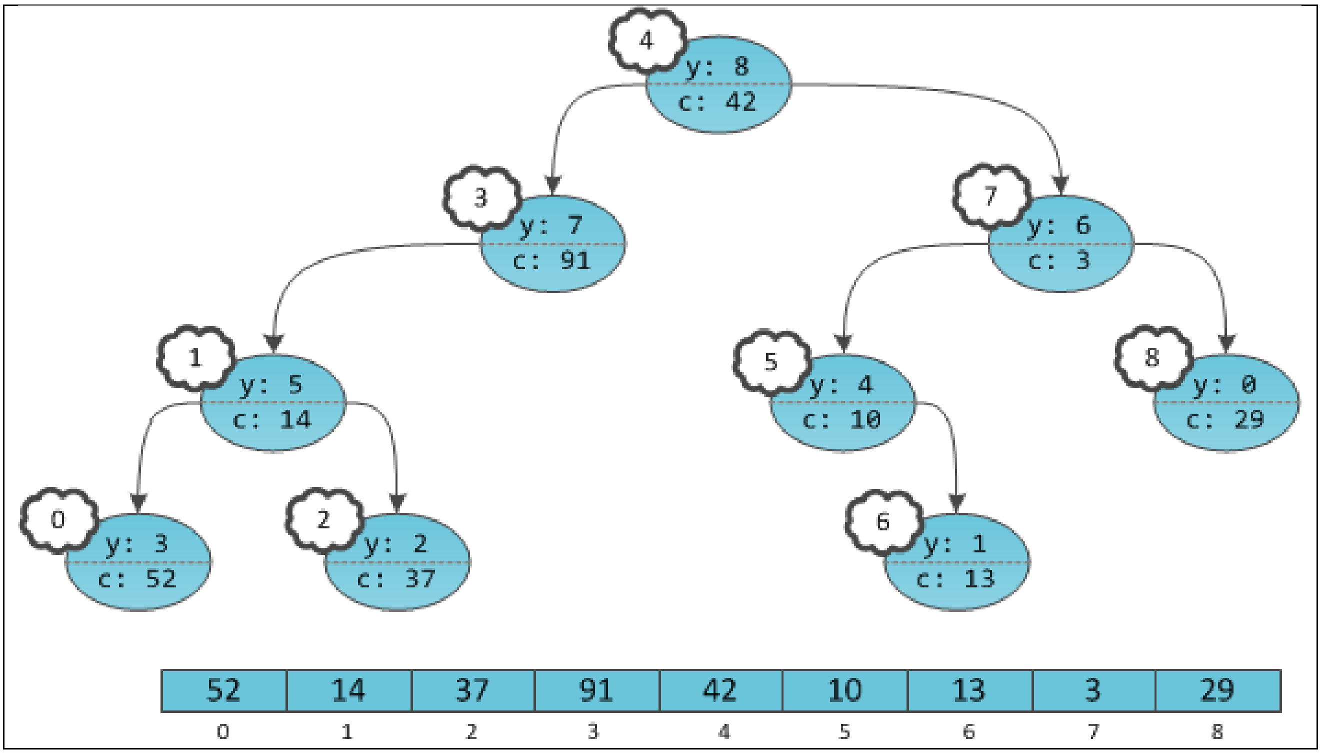

Рисунок 3. Нумерация вершин декартового дерева согласно порядку обхода от наименьшего неявного ключа.

Получается, что в дереве ключи в вершинах не проставлены, а сами вершины пронумерованы. Причем пронумерованы в порядке in-order обхода. Дерево с четко пронумерованными вершинами можно рассматривать как массив, в котором индекс — это тот самый неявный ключ, а содержимое — пользовательская информация *c*. Приоритеты нужны только для балансировки, это внутренние детали структуры данных, ненужные пользователю. Ключей на самом деле нет в принципе, их хранить не нужно, достаточно быстро получать индекс вершины (как это делать будет объяснено далее).

Полученная таким образом структура данных и есть декартово дерево по неявному ключу. Для пользователя она может выглядеть как обыкновенный массив, однако с программной точки зрения этот массив устроен как дерево. Такое представление позволяет выполнять многие операции с массивами, такие как их разъединение, слияние, удаление и вставка элементов, множественные операции и вычисление функций на отрезке за логарифмическую сложность взамен линейной, которую предоставляет обычный массив. Однако можно выделить и 2 существенных недостатка. Так как данные в декартовом дереве по неявному ключу хранятся в различных местах памяти, а связываются при помощи указателей, быстрая индексация по такому дереву невозможна, в отличие от массива, в котором данные расположены непрерывно. Также из-за "древовидного" представления эта структура данных не имеет явного "последнего элемента", и добавление нового элемента в ее конец требует стандартного алгоритма вставки элемента в дерево, сложность которого - $O(log_2 N)$, из-за чего добавить элемент в конец за константное время не получится. Декартово дерево по неявному ключу не накладывает никакие условия на пользовательские данные, расположенные в нем, из-за чего поиск элемента будет ничем не лучше обычного поиска в массиве, все так же придется пройтись по всем элементам. Сравнение основных операций приведено в таблице 1.

Таблица 1. Асимптотика времени выполнения операции для массива и декартового дерева по неявному ключу.

	Массив	Декартово дерево по неявному ключу
Добавление элемента в конец	$O(1)$	$O(log_2 N)$
Вставка элемента	$O(N)$	$O(log_2 N)$
Индексация	$O(1)$	$O(log_2 N)$
Заполнение данными	$O(N)$	$O(Nlog_2 N)$
Поиск	$O(N)$	$O(N)$
Удаление	$O(N)$	$O(log_2 N)$
Слияние	$O(N)$	$O(log_2 N)$
Разъединение	$O(N)$	$O(log_2 N)$
Вычисление значения функции на отрезке	$O(N)$	$O(log_2 N)$
Множественная операция над элементами на отрезке	$O(N)$	$O(log_2 N)$

Полученные данные говорят о том, что практически все основные операции, производящиеся над хранящимися данными, в декартовом дереве по неявному ключу имеют логарифмическую сложность. Также эта структура данных выгоднее для изменения данных, в то время как массив выгоднее с точки зрения добавления и индексации по данным. Декартово дерево по неявному ключу также не использует лишней памяти (все данные хранятся в пределах линейной сложности).

Отличительной особенностью декартового дерева по неявному ключу является совершение групповых операций с элементами, таких как множественные запросы на отрезке и изменение значений на отрезке. Популярные структуры данных для реализации этой же задачи – дерево отрезков и дерево Фенвика. Их сравнение с декартовым деревом по неявному ключу приведено в таблице 2.

Таблица 2. Сравнение эффективности выполнения групповых операций с помощью декартового дерева по неявному ключу, дерева отрезков и дерева Фенвика.

Структура данных	Дерево Фенвика	Дерево отрезков	Декартово дерево по неявному ключу
Требования к исполняемой функции	Ассоциативная, коммутативная, обратимая операция.	Ассоциативна и существует нейтральный элемент относительно этой операции.	Ассоциативна и существует нейтральный элемент относительно этой операции.
Скорость выполнения	$O(log_2 N)$	$O(log_2 N)$	$O(log_2 N)$
Недостатки по сравнению с другими методами	Операция обязана быть обратимой, что ограничивает число возможных функций. Например, невозможен быстрый поиск минимума и максимума (за исключением их поиска на префиксе). Для каждой отдельной функции требуется строить отдельное дерево.	Сложность реализации. Для каждой отдельной функции требуется строить отдельное дерево. По сравнению с деревом Фенвика требует в константу раз больше памяти.	Сложность реализации. По сравнению с деревом отрезков требует в константу раз больше памяти.
Достоинства по сравнению с другими методами	Простота реализации, оптимальность по памяти.	В рамках совершения одной групповой операции оптимальнее декартового дерева по памяти.	Универсальность структуры и возможность совершения многих групповых операций с помощью одного дерева.

Из проведенного анализа следует, что декартово дерево по неявному ключу – универсальная структура данных, являющаяся представлением массива. С ее помощью можно наиболее эффективно вставлять и удалять элементы в заданные места, а также манипулировать с подмассивом (или всем массивом), разбивая его на части и сливая с другим массивом, а также изменяя его значения или получая значения заданной функции на этом подмассиве.

Глава 2. Практика.

Для решения задачи использовались следующие заголовочные файлы из стандартной библиотеки С++:

- iostream – необходим для ввода и вывода.
- cstdlib – необходим для генерации случайных чисел (приоритетов).

Весь код был написан в пространстве имен std.

Реализация класса *TreapNode.*

Для удобной работы с деревом был реализован шаблонный класс *TreapNode,* описывающий каждую вершину декартового дерева по неявному ключу. Так же для удобства его поля и методы были объявлены публичными. Подробнее о каждом из полей далее.

Поля класса *TreapNode.*

- *priority* – целочисленный приоритет, определяющийся случайно из максимального диапазона и использующийся для достижения оптимального баланса декартового дерева.
- *data* – поле для хранения пользовательских данных, тип которых может быть любым.
- *size* – хранит размер (количество потомков) текущей вершины с учетом самой вершины. Необходимо для определения неявного ключа (номера вершины в порядке обхода) и построения дерева по нему дерева поиска.
- *left* – указатель типа *TreapNode* на левую дочернюю вершину. Необходим для связи в дереве.
- *right* – указатель типа *TreapNode* на правую дочернюю вершину. Необходим для связи в дереве.
- *SumTreeData* – поле пользовательского типа, использующее для накопления ответа в задаче поиска суммы данных на отрезке. Хранит сумму всех данных поддерева с корнем в текущей вершине.
- *Add* – поле для демонстрации отложенных операций на отрезке. Указывает на «обещание» прибавки к полю *data* некоторого значения того же пользовательского типа. С его помощью можно быстро прибавить ко всему подмассиву некоторое значение.

При желании реализации других функций от данных на отрезке, например быстрого подсчета произведения, в класс следует добавлять новые поля для накопления ответов (для произведения *ProdTreeData*) и определять для них методы накопления. Это позволит за логарифмическую сложность вычислять значения нужных функций.

То же касается и полей для отложенных операций. Главным условием для них будет то, что их возможно «проталкивать» дочерним узлам за *O(1).* Примером может служить операция переворота подмассива, ведь в дереве это реализуется как простая перестановка дочерних узлов местами. Подробнее проталкивание отложенных операций будет описано позже.

При создании новой вершины значения каждому полю присваиваются в базовом конструкторе. Так же предусмотрен конструктор копирования данных из другой вершины. Значения по умолчанию всех полей вершины приведены в таблице 3.

Таблица 3. Значения по умолчанию всех полей элемента класса TreapNode.

Поле	*priority*	*data*	*SumTreeData*	*size*	*left*	*right*	*Add*
Значение по умолчанию	Случайное целое число	0	0	1	nullptr	nullptr	0

Методы класса *TreapNode.*

Данный класс включает три метода, которые, как указывалось ранее, могут дополняться.

- *CostOf()* получает значение накопленной суммы данных *SumTreeData* для текущей вершины, если она не пуста, и 0 в противном случае.
- *SizeOf()* получает размер поддерева из текущей вершины, если она не пуста, и 0 в противном случае
- *recalc()* поддерживает актуальность данных всех полей в вершине дерева при изменении его дочерних вершин. Так, при пересчете текущего размера дерева он должен быть представлен суммой размеров левого поддерева, правого поддерева и 1 (текущая вершина тоже учитывается). Именно это быстрое (за *O(1)*) поддержание актуального значения позволяет корректно оперировать с неявными ключами (номерами) и строить дерево поиска. При этом пересчета на каждом шаге требует и поле *SumTreeData.* Суммируя все подсуммы левого и правого поддеревьев, а также значение текущей вершины и значение поля отложенной операции Add, умноженное на размер всего текущего поддерева (каждому дочернему узлу обещана отложенная операция увеличения суммы), можно за *O(1)* поддерживать актуальную информацию о сумме текущего поддерева.

Также для удобства переопределен оператор вывода вершины. Для этого объявлена дружественная функция, перегружающая стандартный оператор вывода для текущей вершины таким образом, чтобы выводилась исключительно пользовательская информация (поле *data*).

Реализация класса *Treap.*

Главный класс, для решения поставленной задачи – шаблонный класс декартового дерева по неявному ключу *Treap.* Именно в нем реализованы интересующие операции слияния, разделения, проталкивания отложенной операции, вставки, индексации, удаления и добавления элементов.

Поля класса *Treap*.

Декартово дерево по неявному ключу может однозначно определяться значением единственной своей вершины – корня (поле *root*). Благодаря ссылкам на дочерние вершины из корня можно получить доступ ко всем остальным узлам.

Таким образом единственным полем является поле *root,* для удобства представленное указателем на элемент класса *TreapNode*. Оно определено как закрытое (privat).

Из этого сразу же следует базовый конструктор, в котором *root* присваивается значение nullptr.

Методы класса *Treap*.

pushAdd().

Так как в задаче требовалось выполнение отложенных операций, в дереве следует поддерживать данные обещания на каждом этапе. Для этого был реализован метод *pushAdd()* типа void, выполняющий обещанное обещание о прибавке к полю *data* текущей вершины значение поля *Add* и «проталкивающий» обещание далее по дереву. «Проталкивание» заключается в добавлении полям *Add* дочерних узлов текущей вершины ее значение *Add.* После выполнения обещания поле *Add* обнуляется. Визуализация этого алгоритма для приведена на рисунке 4, где в полях *data* вершин дерева *T* записана стоимость (*cost*).

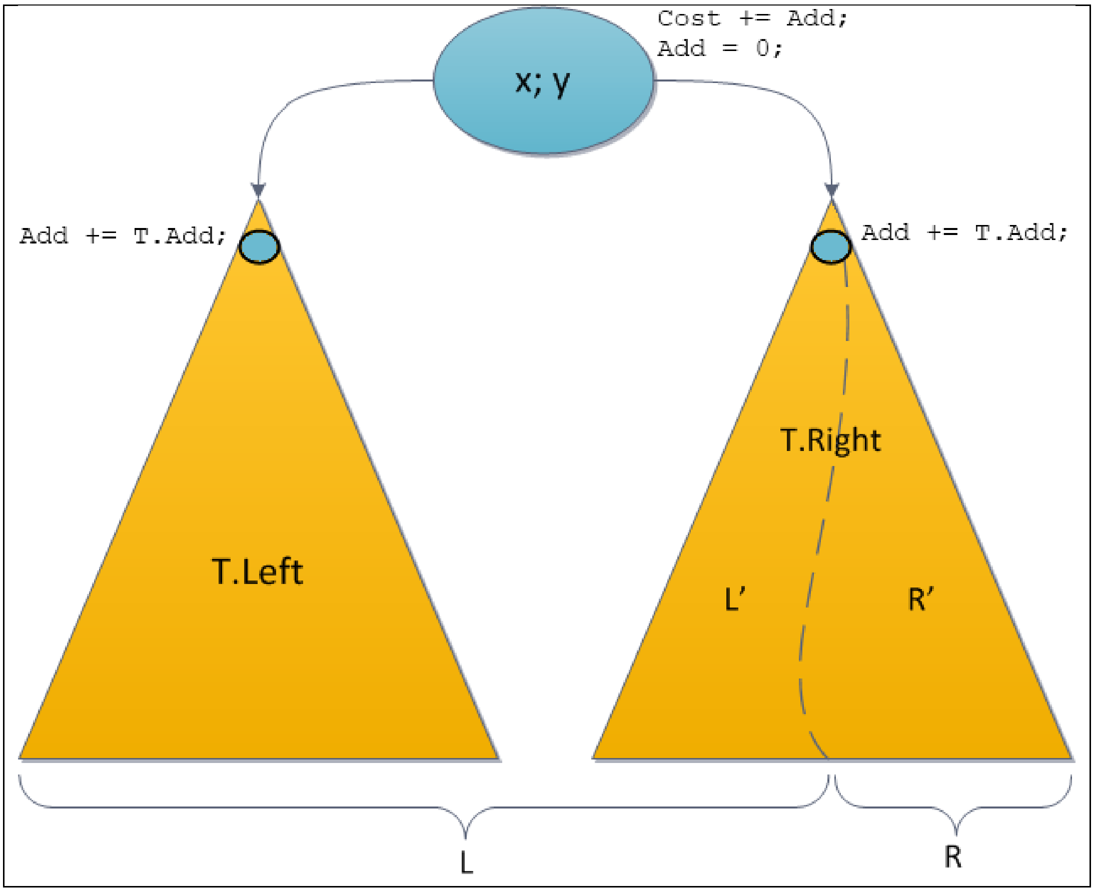

Рисунок 4. Визуализация алгоритма выполнения и проталкивания отложенной операции *pushAdd().*

Когда информация в текущей вершине будет обновлена, при рекуррентном спуске к дочерним узлам операция проталкивания повторится и в итоге обещание выполнится для всего поддерева.

Merge().

Одна из двух базовых операций с декартовым деревом по неявному ключу – метод *merge(),* который реализует слияние двух уже построенных декартовых деревьев.

Операция *Merge* принимает на вход два декартовых дерева ***L*** и ***R***. От нее требуется слить их в одно, тоже корректное, декартово дерево ***T***. Следует заметить, что работать операция *Merge* может не с любыми парами деревьев, а только с теми, у которых все неявные ключи одного дерева (*L*) не превышают ключей второго (*R*). В терминах массива это будет означать, что большее поддерево продолжит индексацию меньшего.

Алгоритм работы *Merge* очень прост. Какой элемент станет корнем будущего дерева? Очевидно, с наибольшим приоритетом. Кандидатов на максимальный приоритет у нас два — только корни двух исходных деревьев. Сравним их приоритеты; пускай для однозначности приоритет у левого корня больше, а ключ в нем равен x. Новый корень определен, теперь стоит подумать, какие же элементы окажутся в его правом поддереве, а какие — в левом.

Легко понять, что все дерево *R* окажется в правом поддереве нового корня, ведь неявные номера у него больше номера корня по условию. Точно так же левое поддерево старого корня *L.Left* имеет все номера меньшие номера корня, и должно остаться левым поддеревом. А правое должно по тем же соображениям оказаться справа, однако неясно, куда тогда ставить его элементы, а куда элементы дерева *R*?

Однако все достаточно просто. У нас есть два дерева, ключи в одном меньше ключей в другом, и нам нужно их как-то объединить и полученный результат привесить к новому корню как правое поддерево. Просто рекурсивно вызываем *Merge* для *L.Right* и дерева *R*, и возвращенное ею дерево используем как новое правое поддерево.

На рисунке 5 синим цветом показано правое поддерево результирующего дерева после операции *Merge* и связь от нового корня к этому поддереву.

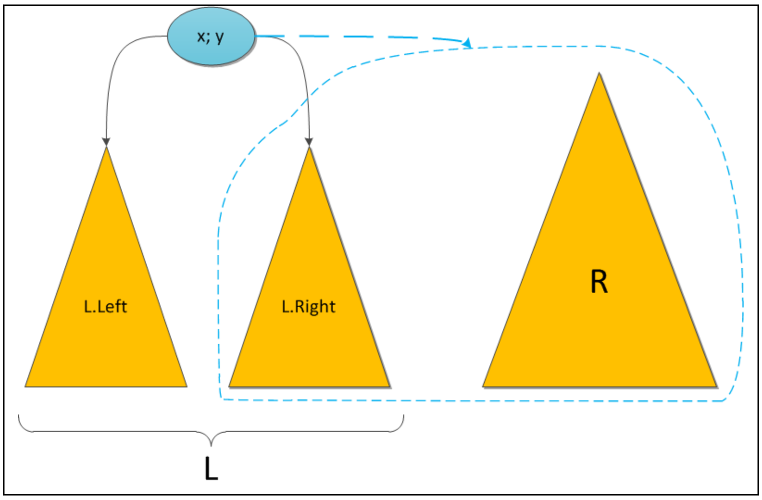

Рисунок 5. Наглядное представление в части алгоритма *merge* на этапе рекуррентного слияния правого дочернего поддерева левого дерева с правым деревом.

Симметричный случай — когда приоритет в корне дерева R выше — разбирается аналогично. База рекурсии, которая в нашем случае наступает, если какое-то из деревьев L и R, или сразу оба, являются пустыми.

Merge за каждую итерацию рекурсии уменьшает суммарную высоту двух сливаемых деревьев как минимум на единицу, так что общее время работы не превосходит 2*H*, то есть O(*H*), и, поскольку декартово дерево со случайными приоритетами, как уже отмечалось, с высокой вероятностью имеет близкую к логарифмической высоту, то *Merge* работает за желаемый $O(log_2N)$.

Split().

Последняя базовая операция для декартового дерева по неявному – разбиение на поддеревья по заданному индексу.

На вход ей поступает корректное декартово дерево *T* и некий неявный индекс *x0*. Задача операции — разделить дерево на два так, чтобы в одном из них (*L*) оказались все элементы исходного дерева с неявными индексами, меньшими *x0*, а в другом (*R*) — с большими. Никаких особых ограничений на дерево не накладывается.

Рассуждаем похожим образом. Где окажется корень дерева *T*? Если его индекс меньше *x0*, то в *L*, иначе в *R*. Опять-таки, предположим для однозначности, что индекс корня оказался меньше *x0*.

Тогда можно сразу сказать, что все элементы левого поддерева *T* также окажутся в *L* — их индексы тоже будут меньше *x0*. Более того, корень *T* будет и корнем *L*, поскольку его приоритет наибольший во всем дереве. Левое поддерево корня полностью сохранится без изменений, а вот правое уменьшится — из него придется убрать элементы с индексами, большими *x0*, и вынести в дерево *R*. А остаток индексов сохранить как новое правое поддерево *L*.

Для этого возьмем правое поддерево и рекурсивно разрежем его по тому же индексу *x0* на два дерева *L'* и *R'*. После чего становится ясно, что *L'* станет новым правым поддеревом дерева *L*, а *R'* и есть непосредственно дерево *R* — оно состоит из тех и только тех индексов, которые больше *x0*.

Симметричный случай, при котором ключ корня больше, чем *x0*, тоже совершенно идентичен. База рекурсии здесь — случаи, когда какое-то из поддеревьев пустое.

В операции *Split* мы работаем с единственным деревом, его высота уменьшается с каждой итерацией как минимум на единицу, и асимптотика работы операции *O(H)*. Поскольку декартово дерево со случайными приоритетами, как уже отмечалось, с высокой вероятностью имеет близкую к логарифмической высоту, то *Split* работает за желаемый $O(\log_2 N)$.

Восстановление справедливости.

При любом действии с деревом необходимо поддерживать актуальную информацию о множественных и отложенных операциях в каждой вершине. То есть с изменением дерева меняются и поля *SumTreeData, size* и *Add.* Так как все дальнейшие операции основаны на базовых методах *merge()* и *split(),* поддерживать актуальную информацию в вершинах нужно именно в них.

Так, для поддержки отложенных операций достаточно использовать метод *pushAdd()* для левого и правого деревьев при выполнении операции слияния. Рекурсивная реализация *merge()* спустит отложенную операцию до конца поддерева и в итоге значение данных обновится. Так же следует поступить и с операцией *split(),* вызвав метод *pushAdd()* в начале для разделяемого дерева. Важная оговорка состоит в том, что эти операции не изменяют последние элементы дерева (листья). Поэтому, хотя все данные (даже в листьях) будут пересчитаны корректно, в листьях для пользователя изменения не будет. Чтобы избежать этого метод *pushAdd()* был дополнен условием того, что текущая вершина – родитель листа. В таком случае Операция *Add* не проталкивалась далее, а сразу изменяла значение поля *data* у листа.

Похожим образом велся пересчет множественных операций. Для поддержания актуальных ответов в *SumTreeData* и *size* пересчет *recalc()* вызывался после каждого рекурсивного спуска в *merge()* и *split()*. В частности, благодаря этому нумерация, поддерживающаяся полем *size* оставалась корректной на каждом этапе. То есть индексы в массивах автоматически обновлялись, что и позволило реализовать концепцию неявного ключа.

После реализации базовых методов стала возможна достаточно простая реализация всех остальных методов, требуемых в задаче.

Вставка и добавление в конец.

Был реализован метод *insert_at()* вставки элемента на заданную позицию в массив (дерево). Так как индекс выступает в роли неявного ключа, достаточно разделить дерево по этому индексу при помощи метода *split()* и объединить полученное левое поддерево с новым элементом (создать вершину класса *TreapNode*) а затем и с полученным правым поддеревом. Визуализация алгоритма приведена на рисунке 6, где x – индекс вставки.

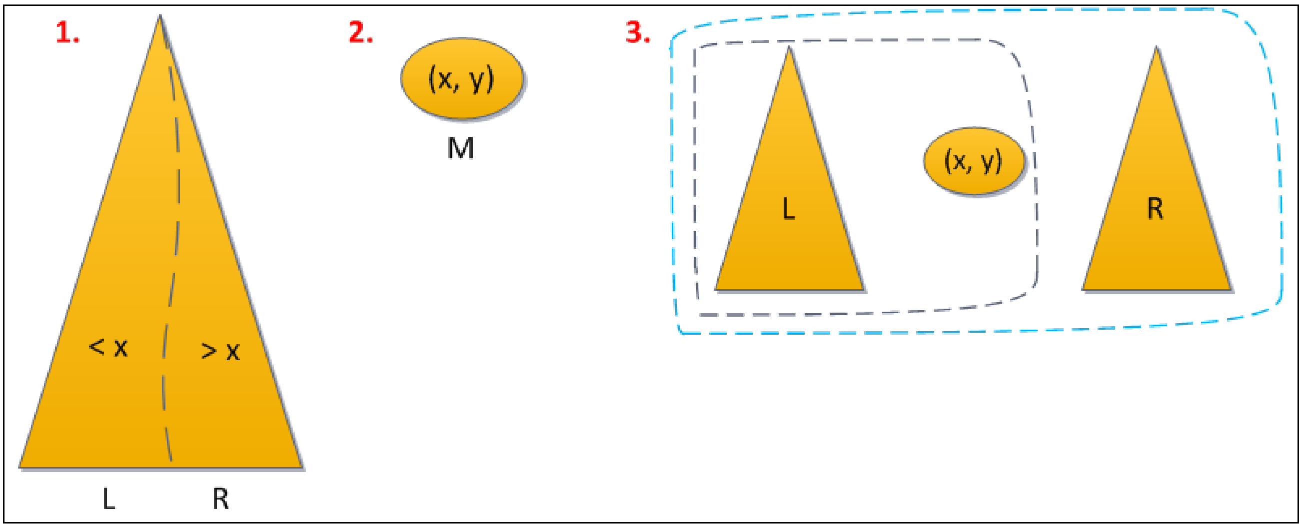

Рисунок 6. Визуализация алгоритма работы *insert_at()*.

На основе метода *insert_at()* был реализован метод *push()* добавления элемента в конец массива (дерева). В нем вызывался метод *insert_at()* для добавляемого элемента с индексом вставки равным длине массива (получаемой из поля *size* корня дерева).

Удаление.

Похожим образом реализован метод *remove()* удаления элемента по его индексу. Дерево разбивается на два по индексу *pos*. Тогда удаляемый элемент имеет индекс 0 в полученном правом поддереве. Остается еще раз применить метод *split()* по индексу 1 для правого поддерева, отделяя при этом удаляемый элемент, и слить оставшиеся деревья при помощи *merge()*.

Вычисление значения функции на отрезке.

Запрос на отрезке [A; B] использует стандартный метод: вырезать из массива искомый отрезок (не забыв, что после первого разреза искомый индекс в правом результате уменьшился) и вернуть значение поля *SumTreeData*, хранящееся в его корне полученного поддерева. Реализация метода *DataSumOn()* поиска суммы элементов на заданном отрезке приведена в коде 1.

```
T DataSumOn(int A, int B)

{

    TreapNode<T>* l;
    TreapNode<T>* m;
    TreapNode<T>* r;
    split(root, A, l, r);
    split(r, B-A+1, m, r);
    T sum = m -> SumTreeData;
    root = merge(merge(l, m), r); // объединяем деревья обратно
    return sum;
}
```
Код 1. Реализация метода *DataSumOn()* поиска суммы элементов на заданном отрезке [A; B].

Изменение данных на отрезке.

Похожим образом можно изменять данные в целом подмассиве за логарифмическую сложность. Для этого в работе и были реализованы отложенные операции. Остается так же, как и в прошлом методе, выделить интересующий отрезок и полю *Add* его корня присвоить значение, на которое нужно увеличить отрезок. Объединяя деревья обратно, операция *merge()* применит отложенную операцию добавления для каждой вершины отрезка, так как в ее начале прописано выполнение операции *pushAdd()*.

Обход по порядку и поиск элемента.

Две эти функции написаны идентично. Выполняется рекурсивный спуск вниз до самого левого листа, это и есть первый элемент. Далее при подъеме по рекурсии выполняется спуск в правую дочернюю вершину и процесс повторяется. На каждом шаге в алгоритме поиска так же проверяется, является ли текущая вершина искомой. Если да, то программа вернет ссылку на нее, в противном случае nullptr. Реализация этих методов приведена в коде 2.

```
void inorderTraversal()
{
    inorderTraversal(root);
}

void inorderTraversal(TreapNode<T>* node, ostream& stream)
{
    if (node == nullptr)
    {
        return;
    }
    inorderTraversal(node->left, stream);
    stream << *node << ' ';
    inorderTraversal(node->right, stream);
}
```

```
TreapNode<T>* search(T key)
{
    return search(root, key);
}

TreapNode<T>* search(TreapNode<T>* node, T key)
{
    if (node == nullptr)
    {
        return nullptr;
    }
    if (node->data == key) return node;
    if (search(node->left, key)) return search(node->left, key);
    if (search(node->right, key)) return search(node->right, key);
}
```

Код 2. Реализация методов обхода по порядку и поиска для декартового дерева по неявному ключу.

Индексация.

Для получения ссылки на вершину с нужным индексом (индексация, как и в массиве, ведется с нуля) был перегружен оператор []. Алгоритм состоит в следующем: смотрим в корень дерева и на размер его левого поддерева S_L, размер правого даже не понадобится.

Если $S_L = K$, то искомый элемент мы нашли, и это — корень.

Если $S_L > K$, то искомый элемент находится где-то в левом поддереве, спускаемся туда и повторяем процесс.

Если $S_L < K$, то искомый элемент находится где-то в правом поддереве. Уменьшим K на число S_L+1, чтобы корректно реагировать на размеры поддеревьев справа, и повторим процесс для правого поддерева.

Заполнение и вывод.

Для удобства построения дерева из множества заданных пользовательской информацией вершин был предусмотрен конструктор из массива данных пользовательского типа. В нем создавался новый экземпляр декартового дерева по неявному ключу, в которой при помощи метода *push()* поочередно добавлялись элементы массива.

Стандартный вывод для этой структуры данных так же был переопределен. Объявленный дружественной функцией, оператор применял вышеописанную функцию *inorderTraversal()***,** которая выводит вершины дерева в заданный поток в порядке прямого обхода.

Таким образом были реализованы все нужные в задаче методы.

Проверка

С помощью команды *srand()* был установлен ключ генерации случайных чисел , равный 12. Это значение можно менять, чтобы по одним и тем же данным генерировать деревья с разными приоритетами, баланс которых может меняться.

Был создан экземпляр декартового дерева по неявному ключу *treap*, вершины которого подразумевают целочисленные пользовательские данные. Заполнив при помощи метода *push()* дерево значениями, был получен его корректный вывод: 10 1 2 7 9 15 6.

После вставки элемента 18 на четвертую позицию дерево изменилось следующим образом:

10 1 2 7 18 9 15 6.

Была произведена проверка операции удаления. Вызвав метод *remove()* по индексу 3 было получено дерево 10 1 2 18 9 15 6.

Вызывая функцию поиска суммы на отрезке от трех до пяти, был получен результат 42.

При вызове элемента под индексом 4 вывелось число 9.

После применения операции по увеличению всех чисел на отрезке от 1 до 5 дерево приобрело следующий вид: 10 6 7 23 14 20 6.

Поиск элемента с данными “2” не выявил результатов, а поиск элемента “23” вернул ссылку на него.

Последним был проверен конструктор, который преобразовал массив целых чисел в декартово дерево по неявному ключу 0 4 8 10 0 9 4 8 4 0.

Код решения задачи приведен в приложении.

Заключение.

Для решения задачи реализации декартового дерева по неявному ключу были изучены такие структуры данных, как массив, дерево поиска, куча, декартово дерево и декартово дерево по неявному ключу. Были выявлены преимущества и недостатки исследуемой структуры в сравнении с массивом. Декартово дерево координат имеет логарифмическую сложность при вставке и удалении элементов, слиянии и разделении поддеревьев, выполнении множественных запросов и отложенных операций на отрезке, чем выгодно отличается от массива, в котором эти операции выполняются за линейное время. Однако добавление элемента в конец и индексация в массиве работают за константное время, а в декартовом дереве по неявному ключу за все то же логарифмическое. Так как данные в дереве по неявному ключу расположены в порядке их добавления, поиск элемента в нем по времени не отличается от поиска элемента в массиве. Так же было установлено, что алгоритмы декартового дерева требуют линейное количество памяти.

В сравнении с деревом отрезков и деревом Фенвика были установлены достоинства и недостатки реализации групповых операций с помощью декартового дерева по неявному ключу. На основании всех проведенных исследований был получен следующий вывод: декартово дерево по неявному ключу – универсальная структура данных, являющаяся представлением массива. С ее помощью можно наиболее эффективно вставлять и удалять элементы в заданные места, а также манипулировать с подмассивом (или всем массивом), разбивая его на части и сливая с другим массивом, а также изменяя его значения или получая значения заданной функции на этом подмассиве.

Все вышеупомянутые методы были реализованы в работе на основе двух базовых функций – слияния и разделения. Таким образом удалось добиться реализации нужных методов за логарифмическое или линейное время (в случаях, когда логарифмическая асимптотика невозможна). Дополнительно были предложены методы создания декартового дерева из массива, а также алгоритм прямого обхода дерева с выводом данных в поток.

Полученный класс был проверен с использованием всех его методов. Результаты проверки совпали с ожиданиями.

Полученная структура оказалась крайне эффективна для работы с данными. Простая для понимания и реализации, она способна эффективно обрабатывать и изменять большие объемы данных. Однако получение элемента по индексу и добавление к ней новых данных работает медленнее, чем в массиве.

Приложение

```
#include <iostream>
#include <cstdlib>

using namespace std;

template <class T>
class TreapNode
{
public:
    int priority;
    T data;
    T SumTreeData;
    int size;
    TreapNode* left;
    TreapNode* right;
    T Add;

    TreapNode(T dt = 0, T ad = 0) : priority(rand()), data(dt), SumTreeData(dt),
                                                        size(1), left(nullptr),
right(nullptr), Add(ad) {}
    // Конструктор копирования
    TreapNode(const TreapNode& other) :
        priority(other.priority), data(other.data), SumTreeData(other.SumTreeData),
size(other.size),
        left(other.left), right(other.right), Add(other.Add) {}

    void recalc()
    {
        size = 1 + SizeOf(left) + SizeOf(right);
        SumTreeData = data + CostOf(left) + CostOf(right) + Add*size;
    }

    T CostOf(TreapNode<T>* treap) { return treap == nullptr ? 0 : treap->SumTreeData
+ treap->Add; }
    int SizeOf(TreapNode<T>* treap) { return treap == nullptr ? 0 : treap->size; }


    template<class T1> friend ostream& operator<< (ostream& stream, const
TreapNode<T1>& N);

};


template<class T>
ostream& operator<< (ostream& stream, const TreapNode<T>& N)
{
    stream << N.data;
    return stream;
```

```
}

template <class T>
class Treap
{
public:
    TreapNode<T>* root;

    void pushAdd(TreapNode<T>* root)
    {
        if (root == nullptr) return;

        root -> data += root -> Add;
        if (root -> left != nullptr)
        {
            root -> left -> Add += root -> Add;
            if (root -> left -> left == nullptr && root -> left -> right == nullptr)
            {
                root -> left -> data += root -> Add;
                root -> left -> Add = 0;
            }
        }
        if (root -> right != nullptr)
        {
            root -> right -> Add += root -> Add;
            if (root -> right -> left == nullptr && root -> right -> right ==
nullptr)
            {
                root -> right -> data += root -> Add;
                root -> right -> Add = 0;
            }
        }

        root->Add = 0;
    }

    TreapNode<T>* merge(TreapNode<T>* left, TreapNode<T>* right)
    {
        pushAdd(left);
        pushAdd(right);
        if (left == nullptr) return right;
        if (right == nullptr) return left;

        TreapNode<T>* answer;
        if (left->priority > right->priority)
        {
            left->right = merge(left->right, right);
            answer = left;
        }
        else
        {
```

```
            right->left = merge(left, right->left);
            answer = right;
        }

        answer -> recalc(); // пересчёт!
        return answer;
    }

    void split(TreapNode<T>* root, int key, TreapNode<T>*& left, TreapNode<T>*&
right)
    {
        if (root == nullptr)
        {
            left = nullptr;
            right = nullptr;
            return;
        }

        int curIndex = root->left == nullptr ? 1 : root->left->size+1;

        root -> data += root -> Add;
        if (root -> left != nullptr) { root -> left -> Add += root -> Add; }
        if (root -> right != nullptr) { root -> right -> Add += root -> Add; }
        root->Add = 0;

        if (curIndex <= key)
        {

            TreapNode<T>* newTree = nullptr;
            split(root->right, key-curIndex, newTree, right);

            left = new TreapNode<T>(*root); // Создаем новую вершину как копию root
            left->right = newTree;
            left->recalc(); // пересчёт в L!
        }
        else
        {
            TreapNode<T>* newTree = nullptr;
            split(root->left, key, left, newTree);

            right = new TreapNode<T>(*root); // Создаем новую вершину как копию root
            right->left = newTree;
            right->recalc(); // пересчёт в R!
        }
    }

    Treap() : root(nullptr) {}

    Treap(T* arr, int len)
    {
        Treap<T> temp;
```

```
    for (int i = 0; i < len; i++)
        temp.push(arr[i]);
    root = temp.root;
}

void insert_at(T data, int pos)
{
    TreapNode<T>* new_node = new TreapNode<T>(data);
    TreapNode<T>* left;
    TreapNode<T>* right;
    split(root, pos, left, right);

    root = merge(merge(left, new_node), right);
}

void push(T data)
{
    int last = root!= nullptr ? root->size : 0;
    insert_at(data, last);
}

void remove(int pos)
{
    TreapNode<T>* l;
    TreapNode<T>* m;
    TreapNode<T>* r;
    split(root, pos, l, r);
    split(r, 1, m, r);
    root = merge(l, r);
    delete m;
}

T DataSumOn(int A, int B)
{
    TreapNode<T>* l;
    TreapNode<T>* m;
    TreapNode<T>* r;
    split(root, A, l, r);
    split(r, B-A+1, m, r);
    T sum = m -> SumTreeData;
    root = merge(merge(l, m), r); // объединяем деревья обратно
    return sum;
}

TreapNode<T>* operator[](int index)
{
    TreapNode<T>* cur = root;
    while (cur != nullptr)
    {

        int sl = cur->left != nullptr ? cur->left->size : 0;
```

```
            if (sl == index) { return cur; }

            cur = sl > index ? cur->left : cur->right;
            if (sl < index)
                index -= sl + 1;
        }
        return nullptr;
    }


    void inorderTraversal()
    {
        inorderTraversal(root);
    }

    void inorderTraversal(TreapNode<T>* node, ostream& stream)
    {
        if (node == nullptr)
        {
            return;
        }

        inorderTraversal(node->left, stream);
        stream << *node << ' ';
        inorderTraversal(node->right, stream);
    }

    void IncreaseSubTree(T value, int A, int B)
    {
        TreapNode<T>* l;
        TreapNode<T>* m;
        TreapNode<T>* r;
        split(root, A, l, r);
        split(r, B-A+1, m, r);
        m->Add += value;
        root = merge(merge(l, m), r);
    }

    TreapNode<T>* search(T key)
    {
        return search(root, key);
    }

    TreapNode<T>* search(TreapNode<T>* node, T key)
    {
        if (node == nullptr)
        {
            return nullptr;
        }
        if (node->data == key) return node;
        if (search(node->left, key)) return search(node->left, key);
```

```
        if (search(node->right, key)) return search(node->right, key);

    }

    template<class T1> friend ostream& operator<< (ostream& stream, Treap<T1>& N);

};

template<class T>
ostream& operator<< (ostream& stream, Treap<T>& N)
{
    N.inorderTraversal(N.root, stream);
    return stream;
}

int main() {
    unsigned random_value = 12;
    srand(random_value);

    Treap<int> treap;

    treap.push(10);
    treap.push(1);
    treap.push(2);
    treap.push(7);
    treap.push(9);
    treap.push(15);
    treap.push(6);
    cout << "\nInorder Traversal:\n" << treap;

    treap.insert_at(18, 4);
    cout << "\nInorder Traversal after iserting 18 at pos 4:\n" << treap;

    treap.remove(3);
    cout << "\nInorder Traversal after removing element from pos 3:\n" << treap;

    cout << "\nSum of data on segment [3; 5] = " << treap.DataSumOn(3, 5);
    cout << "\nElement with index 4: " << *treap[4];
    treap.IncreaseSubTree(5, 1, 5);
    cout << "\nInorder Traversal after increaising all data values on segment [1,
5]:\n" << treap;
    TreapNode<int>* found = treap.search(2);
    cout << "\nData 2 found: " << (found ? "Yes" : "No") ;
    found = treap.search(23);
    cout << "\nData 23 found: " << (found ? "Yes" : "No");

    int *a = new int[10]{0, 4, 8, 10, 0, 9, 4, 8, 4, 0};
    Treap<int> newtreap(a, 10);
    cout << "\nInorder Traversal for arraytreap:\n" << newtreap;
    return 0;
}
```