



ИПМ им.М.В.Келдыша РАН • Электронная библиотека

Препринты ИПМ • Препринт № 72 за 2025 г.

ISSN 2071-2898 (Print)
ISSN 2071-2901 (Online)

А.М. Котельников

О вычислительной
эффективности типов данных
для последовательностей
при хранении и обработке
нерегулярных сеток

Статья доступна по лицензии
[Creative Commons Attribution 4.0 International](https://creativecommons.org/licenses/by/4.0/)



Рекомендуемая форма библиографической ссылки: Котельников А.М. О вычислительной эффективности типов данных для последовательностей при хранении и обработке нерегулярных сеток // Препринты ИПМ им. М.В.Келдыша. 2025. № 72. 21 с. EDN: [JVGEZU](https://library.keldysh.ru/preprint.asp?id=2025-72)
<https://library.keldysh.ru/preprint.asp?id=2025-72>

О р д е н а Л е н и н а
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В. Келдыша
Р о с с и й с к о й а к а д е м и и н а у к

А. М. Котельников

О вычислительной эффективности типов данных
для последовательностей при хранении и обработке
нерегулярных сеток

Москва — 2025

Котельников А. М.

О вычислительной эффективности типов данных для последовательностей при хранении и обработке нерегулярных сеток

Излагаются основные свойства структур данных для последовательностей, таких как массивы, стеки, очереди и связанные списки. Разработаны новые типы для коллекций на языке C++ с использованием "любопытного" рекурсивного шаблона, позволяющего избегать копирования кода между классами. Проведено сравнение производительности структур данных из стандартной библиотеки C++, из кода MARPLE и собственных разработок.

Ключевые слова: динамический массив; дерево Фенвика; "любопытный" рекурсивный шаблон

A. Kotelnikov

On the Computational Efficiency of Data Types for Sequences in Storage and Processing Irregular Grids

The main properties of data structures for sequences, such as arrays, stacks, queues, and linked lists, are outlined. New types for collections in C++ have been developed using a curious recursive template to avoid code duplication between classes. A performance comparison has been conducted among the data structures from the C++ standard library, the MARPLE code, and proprietary developments.

Key words: dynamic array; Fenwick tree; curiously recurring template pattern

1 Введение

При программной реализации дискретных сеточных моделей сплошной среды естественным образом возникает необходимость хранения информации о соответствии между элементами сетки и относящимися к ним физическими величинами. Первоначально, в эпоху становления теории разностных схем, в вычислительной практике доминировали разнообразные варианты схем на регулярных сетках. Способ хранения таких сеток был простым: элемент двумерного (или трехмерного) массива очевидным образом ассоциировался с узлом (или ячейкой) прямоугольной сетки. На самом деле даже в столь простом случае вопрос не столь очевиден, как кажется: при погружении тела нетривиальной формы в прямоугольную область некоторые ячейки должны были быть объявлены неактивными, что приводило к неполному использованию прямоугольного массива для хранения сеточных данных.

Использование сеток нерегулярной структуры, получившее чрезвычайно широкое распространение, привело к необходимости хранения информации о структуре самой сетки, помимо сеточных величин. Для хранения таких сеток разрабатывались различные структуры данных [1, 2], оптимизированные как для стационарных, так и для нестационарных сеток. В то же время языки программирования, такие как Фортран и С, не предоставляли разработчикам каких-либо сложных структур данных для хранения последовательностей объектов – единственной структурой для хранения последовательности оставался массив, а при необходимости хранения динамических последовательностей (то есть предполагающих эффективную реализацию удаления, вставки и перемещения подпоследовательности) можно было использовать самостоятельно реализованные связные списки. Соответственно, структуры для хранения сеток так или иначе основывались на массивах (или, реже, на связных списках).

Появление языка C++ и традиции его использования для решения вычислительных задач породили новые возможности для хранения и обработки сеток и сеточных величин. В основном здесь имеется в виду стандартная библиотека шаблонов STL, содержащая классы контейнеров, объединенных общим интерфейсом. Использование того или иного контейнера обычно считается вопросом чисто технического, программистского уровня, и поэтому обычно если и обсуждается, то только в узком кругу разработчиков. Обоснование выбора того или иного контейнера с точки зрения оптимизации производительности стало очень нетривиальной задачей. На эффективность реализации теперь также влияет сложная архитектура современных процессоров: из-за наличия разноуровневой кэш-памяти принято считать, что производительность вычислений резко возрастает в случае компактного размещения данных в адресном пространстве памяти; это обстоятельство сильно снизило популярность динамических структур типа связных списков.

В настоящей работе приводятся результаты тестирования производительности различных контейнеров: как стандартных из библиотеки STL, так и

некоторых разработанных и используемых авторами в ходе работы над кодом MARPLE [2].

2 Общие сведения о структурах данных, исследуемых в данной работе

Перечислим основные рассматриваемые структуры данных.

Динамический массив хранит данные подряд в едином участке динамически выделенной памяти.

Статический массив хранит данные в теле класса подобно динамическому массиву, не обращаясь при этом к динамической памяти.

Связный список хранит данные в **нодах** – специальном классе, хранящем элемент вместе с указателем на следующий и, опционально, предыдущий элемент.

Далее рассмотрим структуры данных, разработанные ранее для пакета прикладных программ MARPLE.

Кластерный массив – односвязный список указателей на массивы одинакового фиксированного размера.

Односвязный список с глобальным пулом нод является вариацией односвязного списка, но память выделяется не отдельно для каждого элемента, а массивом, содержащим память под константное число нод.

Теперь перечислим разработанные в рамках статьи структуры данных.

Расщепленный массив хранит данные в участках памяти постоянного размера. Сами указатели на участки хранятся в динамическом массиве.

Каскадный массив устроен аналогично расщепленному массиву, но размер каждого участка памяти в два раза больше предыдущего.

Фенвиков массив разностей – структура данных, представляющая собой дерево Фенвика, но, в отличие от него, хранящая вместо самих элементов их разности во внутреннем динамическом массиве. Причиной разработки явилась необходимость прибавлять константу к идущим подряд элементам в некотором диапазоне индексов и делать это за логарифмическое от числа элементов время.

Связный список на внутреннем массиве – вариант связного списка, где элементы хранятся во внутренней структуре данных.

В таблице 1 представлены основные асимптотики.

Асимптотические свойства структур данных

Структура данных	Время доступа к произвольному элементу	Объем дополнительной памяти	Количество участков памяти
Динамический массив	$O(1)$	$O(1)$ $O(N)$ пиковый	$O(1)$ $O(\log N)$ временных
Связный список	$O(N)$	$O(N)$	$O(N)$
Кластерный массив (MARPLE)	$O(M)$	$O(K + M)$	$O(M)$
Односвязный список с глобальным пулом нод (MARPLE)	$O(N)$	$O(N + K)$	$O(M)$
Расщепленный массив	$O(1)$	$O(K + M)$	$O(M)$
Каскадный массив	$O(1)$	$O(N)$	$O(\log N)$
Каскадный массив без интринсик	$O(\log C)$	$O(N)$	$O(\log N)$
Фенвиков массив разностей	$O(\log N)$	$O(1)$ $O(N)$ пиковый	$O(1)$ $O(\log N)$ временных

где

N — количество элементов во всей структуре данных,

K — количество элементов в одном кластере,

C — количество элементов, под которые зарезервирована память,

M — количество используемых кластеров.

3 Свойства структур данных

Рассмотрим теоретические аспекты работы структур данных.

3.1 Статический массив

Массив, максимальный размер которого заранее известен. Благодаря этому хранимые объекты размещаются в теле класса, что позволяет не расходовать динамическую память, избегая фрагментирования и уменьшая за-

траты времени на создание массива, что особенно актуально для временных структур, предназначенных для хранения данных в пределах одной функции. Объекты сохраняют свои позиции при добавлении новых элементов, но перемещаются вместе с переменной, представляющей массив, что усложняет использование этой структуры данных в качестве вложенной в другие структуры, так как указатели на элемент вложенного статического массива могут потерять свой объект, на который обязаны указывать. В C++ эта структура данных реализована в библиотеке boost [3], а также будет включена в стандартную библиотеку нового стандарта C++26 [4].

3.2 Динамический массив

Динамический массив хранит объекты в непрерывном участке памяти, который может быть заполнен как частично, так и полностью. При добавлении элемента в полностью заполненный массив происходит расширение участка памяти, что может сопровождаться перемещением объектов на новое место. Очевидно, такая операция имеет временную сложность $O(N)$. Для того, чтобы удержать среднюю сложность времени заполнения в пределах $O(N)$, нужно обеспечить достаточно редкое перемещение объектов на новое место. Это можно гарантировать, если заполненный участок памяти расширяется не менее, чем в $\alpha > 1$ раз, где α — коэффициент расширения.

Существенным недостатком динамического массива является возможность смены адресов объектов при изменении вместимости, либо дефрагментации памяти.

3.2.1 Выбор коэффициента расширения

При формировании рекомендаций по выбору будем исходить из следующих соображений. Во-первых, новый участок должен быть выделен до того, как будет освобожден старый. Таким образом, оба участка должны быть размещены в памяти, и лишь затем один из них будет освобожден и пригоден для очередного выделения памяти. Это требование связано как с отсутствием такой функциональности в стандартных библиотеках, так и с проблемами при перемещении нетривиальных классов на новые адреса. Если же есть возможность двигать границы текущего участка памяти, то коэффициент расширения можно выбирать свободно. Во-вторых, класс, реализующий динамический массив, отвечает лишь за себя, поэтому рассматривается процесс заполнения одного динамического массива, не конкурирующего за динамическую память с другими потребителями.

Будем выбирать α из следующих соображений: динамический массив заполняется элементами, последовательно увеличивая вместимость от начальной единичной до финальной α^n : $1, \alpha, \alpha^2, \alpha^3, \dots, \alpha^n$, затем выполняется операция сокращения и дефрагментации, представленная в C++ методом *shrink_to_fit*, пройдя всего n расширений.

Для того чтобы итоговый участок памяти мог быть размещен на месте первого, должно быть выполнено условие

$$\alpha^n \leq \sum_{i=0}^{n-1} \alpha^i = \frac{\alpha^n - 1}{\alpha - 1}.$$

Решая это уравнение при условии, что $\alpha > 1$, получаем условие дефрагментации памяти после заполнения динамического массива:

$$\begin{cases} \alpha < 2, \\ \alpha^n \geq \frac{1}{2-\alpha}. \end{cases} \quad (1)$$

Коэффициент расширения должен быть строго меньше двойки. При этом число расширений, которые должен пройти массив, прежде чем дефрагментация станет возможной, составляет не менее

$$n_0 = -\log_{\alpha} (2 - \alpha). \quad (2)$$

Рассмотрим иной сценарий расширения: $1, \alpha, \alpha^2, \alpha^3, \dots, \alpha^n, \dots$. При этом потребуем, чтобы участок размера α^{n-1} , расширяясь до α^n , встал на позицию исходного участка единичного размера.

$$\alpha^n \leq \sum_{i=0}^{n-2} \alpha^i = \frac{\alpha^{n-1} - 1}{\alpha - 1}.$$

Решая это уравнение при условии, что $\alpha > 1$, получаем условие возврата:

$$\begin{cases} \alpha < \Phi, \\ \alpha^{n-1} \geq \frac{1}{1+\alpha-\alpha^2}, \end{cases} \quad \text{где } \Phi = \frac{1 + \sqrt{5}}{2} \approx 1.618. \quad (3)$$

Длина цикла с возвратом на исходную позицию составляет при этом не менее

$$n_0 = 1 - \log_{\alpha} (1 + \alpha - \alpha^2). \quad (4)$$

Таким образом, можно сформулировать рекомендации по подбору коэффициента расширения:

- при наличии возможности двигать границы участка памяти коэффициент можно выбирать произвольным образом;
- иначе коэффициент должен быть менее двух во избежание фрагментации;
- при коэффициенте менее золотого сечения (≈ 1.618 : корень уравнения $\Phi^2 = \Phi + 1$) новый участок памяти периодически будет оказываться на исходной позиции не менее чем за (4) шагов, снижая общее потребление памяти программой;
- при коэффициенте, не превышающем сверхзолотое сечение (≈ 1.466 : корень уравнения $\alpha^3 = \alpha^2 + 1$), получаем цикл длины 4;

- при коэффициенте, не превышающем пластическое число (≈ 1.325 : корень уравнения $\alpha^3 = \alpha + 1$), получаем цикл длины 3;
- стоит учесть, что размер участка памяти должен быть целым числом, поэтому стоит обратить особое внимание на те из коэффициентов, которые наиболее удобны для выполнения операции целочисленного двоичного умножения и деления. К таким значениям можно отнести 1.25 с циклом длины 3 и коэффициент 1.5 с циклом длины 5.

3.3 Расщепленный массив

Расщепленный массив представляет собой массив указателей на неподвижные подмассивы фиксированной длины, размещаемые в динамической памяти. Доступ к элементу осуществляется путем деления номера запрашиваемого элемента на размер подмассива. Поскольку данная операция является довольно тяжеловесной, то в качестве размера подмассива допускается либо степень двойки, либо, в случае преаллоцированной первой страницы, заранее известная на момент компиляции константа. Деление на константу компилятор потенциально может заменить операцией умножения на обратное к константе число, представленное в двоичном виде периодической дробью. Такой подход позволяет избегать вызова непосредственно операции целочисленного деления с остатком.

3.4 Каскадный массив

Каскадный массив представляет собой модификацию расщепленного массива, когда каждый следующий подмассив оказывается в два раза больше предыдущего. При таком подходе уменьшается число задействованных участков памяти, однако доступ к элементу усложняется.

Пусть требуется получить доступ к элементу I , что означает необходимость вычислить номер подмассива P и смещение элемента F . При размере начального подмассива s получаем условие принадлежности элемента I к подмассиву P :

$$s \sum_{i=0}^{P-1} 2^i \leq I < s \sum_{i=0}^P 2^i.$$

Вычисляя суммы геометрических прогрессий, получаем в итоге систему неравенств:

$$s(2^P - 1) \leq I < s(2^{P+1} - 1). \quad (5)$$

Решая систему неравенств, получаем ограничения на P :

$$P \leq \log_2 \left(\frac{I}{s} + 1 \right) < P + 1.$$

Округляя вниз до ближайшего целого, получаем итоговые формулы.

$$\begin{cases} P = \lfloor \log_2 (\lfloor \frac{I}{s} \rfloor + 1) \rfloor, \\ F = I - s(2^P - 1). \end{cases} \quad (6)$$

Формулы (6) содержат две вычислительно непростые операции: целочисленное деление и целочисленный логарифм по основанию два. Первая из них может быть оптимизирована наложением тех же условий на размер начального подмассива, что и в случае расщепленного массива. Для целочисленного логарифма по основанию два можно использовать специальную команду процессора BSR (Bit Scan Reverse), которая позволяет найти номер самого старшего из выставленных бит целого числа. В C++ данную команду можно использовать, воспользовавшись семейством интринсик `_BitScanReverse` для Visual Studio и `__builtin_clz` для GNU Compiler Collection.

Альтернативный способ поиска позиции элемента без использования интринсик предполагает обход всех подмассивов от последнего к первому с проверкой выполнения условий (5). Поскольку размеры подмассивов удваиваются, последний подмассив содержит более половины от общей вместимости структуры данных, что обеспечивает довольно высокую вероятность получить адрес нужного элемента уже на первой итерации поиска, если в каскадном массиве нет пустых подмассивов.

3.5 Фенвиков массив разностей

Такой массив представляет собой структуру данных, предназначенную для оптимизации одной конкретной операции массового прибавления константы к непрерывному подмассиву исходной последовательности целых чисел при сохранении приемлемого времени доступа к произвольному элементу.

3.5.1 Дерево Фенвика

За основу взята структура данных, именуемая деревом Фенвика, обеспечивающая логарифмическое время для операций доступа к элементу и суммы любого подмассива [6, 5]. Если в эту структуру помещать не сами элементы, а разности двух соседних элементов исходной последовательности, то получим новую структуру данных, на которой остановимся подробнее.

3.5.2 Теоретическое обоснование

Пусть задана исходная последовательность целых чисел, нумеруемая с единицы a_1, \dots, a_N . Примем $a_0 = 0$. Определим функцию `lsb` (Least Significant Bit — наименьший значащий бит) — наибольший делитель положительного целого числа, являющийся степенью двойки. Для этой функции выполняются ключевые условия, что при вычитании или прибавлении к числу его

собственного lsb для получившегося числа возрастает не менее, чем в два раза:

$$lsb(x \pm lsb(x)) \geq 2 \cdot lsb(x). \quad (7)$$

Фенвиков массив разностей хранит последовательность $b_i = a_i - a_{i-lsb(i)}$. Для получения элемента исходной последовательности требуется выполнить суммирование

$$a_i = b_i + b_{i-lsb(i)} + b_{i-lsb(i)-lsb(i-lsb(i))} + \dots$$

. Из свойства удвоения шага (7) следует, что количество слагаемых будет $O(\log N)$.

Фенвиков массив разностей предназначен для выполнения быстрой операции прибавления константы к подряд идущим элементам. Пусть требуется прибавить константу к подряд идущим элементам $a_1 \dots a_n$. Для этого требуется изменить лишь те хранимые элементы b_i , для которых $i - lsb(i) \leq n < i$. Такими элементами являются лишь $b_i, b_{i+lsb(i)}, b_{i+lsb(i)+lsb(i+lsb(i))}, \dots$, количество которых тоже $O(\log N)$.

4 Реализация структур данных

Перечисленные структуры данных реализуют общепринятый в C++ интерфейс стандартной библиотеки шаблонов.

4.1 "Любознательный" рекурсивный шаблон

Интерфейс стандартной библиотеки шаблонов C++ требует от любого класса наличия большого количества методов, нередко повторяющихся. Для обеспечения повторного использования кода применяется подход, известный как "любознательный рекурсивный шаблон" (The Curiously Recurring Template Pattern) [7].

Рассмотрим этот подход на примере методов, отвечающих за получение итератора. Помимо методов `begin` и `end` константной и неконстантной реализации в классе требуется ряд аналогичных методов, отвечающих за получение итератора. Поскольку реализация таких методов одинакова для многих классов, возникает желание иметь для этого кода одно единственное место во всей программе. Реализуется это желанием следующим образом:

```

1 #define ME ((S&)*this)
2 #define MEC ((const S&)*this)
3
4 template<class S> struct iterable {
5     INCLUDE_SUBTYPES(S);
6
7     const_iterator cbegin() const { return MEC.begin(); }
8     const_iterator cend() const { return MEC.end(); }
9
10    reverse_iterator rbegin() {
11        return std::make_reverse_iterator(ME.end());
12    }
13    const_reverse_iterator crbegin() const {
14        return std::make_reverse_iterator(cend());
15    }
16    const_reverse_iterator rbegin() const { return crbegin(); }
17
18    reverse_iterator rend() {
19        return std::make_reverse_iterator(ME.begin());
20    }
21    const_reverse_iterator crend() const {
22        return std::make_reverse_iterator(cbegin());
23    }
24    const_reverse_iterator rend() const { return crend(); }
25 };

```

Рассмотрим некий класс `iterable`, зависящий от класса `S`, передаваемого ему как аргумент шаблона. Это несамостоятельная структура данных, играющая вспомогательную роль, поэтому назовем его класс-кусочек. В противовес этому, класс `S` будем называть главным классом. Класс-кусочек через макросы `ME` и `MEC` (неконстантная и константная ссылки соответственно) получает доступ к методам, которые предоставляют главный класс или другие классы-кусочки, подключенные к главному классу. В процессе компиляции выполняется проверка, что все методы и поля, запрашиваемые одним классом-кусочком, предоставляются другим классом-кусочком, либо непосредственно главным классом.

Для подключения класса-кусочка, требуется выполнить несколько шагов.

Во-первых, нужно определить все типы заранее отдельно от главного класса `my_type`. Позже макрос `INCLUDE_SUBTYPES(S)` позволит быстро их загрузить в любой ваш класс.

```

26 template<class T> struct my_type;
27
28 template<class T>
29 struct subtypes_of<my_type<T>> {
30     using value_type = T;
31     using size_type = std::size_t;
32     using difference_type = std::ptrdiff_t;
33     using reference = value_type&;
34     using const_reference = const value_type&;
35     using pointer = value_type*;
36     using const_pointer = const value_type*;
37     using iterator = pointer;
38     using const_iterator = const_pointer;
39     using reverse_iterator = std::reverse_iterator<iterator>;
40     using const_reverse_iterator = std::reverse_iterator<const_iterator>;
41 };

```

Во-вторых, реализовать сам главный класс, подключив классы-кусочки и определив необходимые для их работы методы.

```
1  template<class T>
2  struct my_type :
3      public iterable<my_type<T>>
4  {
5      INCLUDE_SUBTYPES(my_type<T>);
6  private:
7      // class implementation
8  public:
9      iterator begin() { /*method implementation*/ }
10     iterator end() { /*method implementation*/ }
11
12     const_iterator begin() const { /*method implementation*/ }
13     const_iterator end() const { /*method implementation*/ }
14 };
```

Класс-кусочек `iteratable` содержит 8 методов, которые включаются в главный класс при помощи наследования. Для доступа к этим методам из главного класса потребуется приведение типа `*this` к `iteratable < my_type < T >>`, однако снаружи все эти методы доступны пользователю главного класса безо всяких затруднений.

При помощи "любопытного" рекурсивного шаблона в классы-кусочки можно вынести любые наборы методов, общие для нескольких разных главных классов, избегая тем самым копирования кода между разными частями программы, повышая общую надежность, не затрагивая при этом производительность программы.

5 Замеры производительности

5.1 Методика

Измерена операция сортировки случайной последовательности элементов, сгенерированной стандартной операцией `std::srand(1)`. Сортировка выполнена функцией `std::sort` для массивов (таблица 2) и специальным методом `sort` для списков (таблица 3), а также собственной реализацией метода `sort` для списка из MARPLE. Также измерена операция прибавления константы к первой трети элементов. Обе измеряемые операции достаточно сложны, чтобы избежать избыточной оптимизации всей программы, которая может затронуть простые операции, такие как простое суммирование всех элементов. Общий размер структуры выбран 2^{27} восьмибайтовых чисел.

Для сброса кэша между замерами применяется операция выделения новой страницы и последующего освобождения. Сама операция не требует много времени, однако обязывает операционную систему дважды обновить список страниц процесса. Все замеры проводятся на одном запуске во избежание влияния возможных различий между узлами суперкомпьютера. Каждый результат был получен последовательностью из 60 замеров (10 для сортировки списков). Перед началом последовательности один замер выполнялся вхолостую для того, чтобы программа получила от операционной системы всю

необходимую память, а также подгрузила все необходимые динамические библиотеки.

Таблица 2

Результаты замеров

Структура данных	Время сортировки при помощи <code>std::sort</code> , сек.	Время прибавления константы к первой трети последовательности, сек.
динамический массив <code>std::vector</code>	11,17	0,044
очередь <code>std::deque</code>	13,23	0,056
кластерный массив MARPLE	16,34	0,054
кластерный массив MARPLE с уменьшенным размером кластера	53,94	0,16
расщепленный массив	16,74	0,055
расщепленный массив (7)	24,55	0,11
расщепленный массив (8)	19,60	0,064
каскадный массив (с интринсиками)	22,26	0,069
каскадный массив	19,83	0,055
каскадный массив (7)	26,75	0,092
каскадный массив (8)	21,88	0,072
фенвиков массив разностей	145,75	$3,68 \cdot 10^{-7}$

где число в скобках — количество элементов в статически преаллоцированной странице

Результаты замеров для списков

Структура данных	Время сортировки, сек.	Время прибавления константы к первой трети последовательности, сек.
двусвязный список std::list	159,81	0,22
односвязный список std::forward_list	356,692	0,22
односвязный список MARPLE	416,136	0,12
односвязный список MARPLE без кластеров	427,144	0,19

5.2 Выводы

Результаты представлены в таблицах 2 и 3.

Кластерный массив MARPLE уступает по производительности стандартным структурам данных, а также имеет быстрорастущие асимптотики.

Односвязный список MARPLE по времени итерирования превосходит стандартные списки благодаря глобальному кэшу нод. Это демонстрирует эффективность кэширования при оптимизации структур данных. При этом собственный метод сортировки уступает стандартным реализациям при отсутствии кэширования. Следует ожидать значительного снижения эффективности односвязного списка MARPLE по мере перемешивания нод в памяти в ходе активной модификации списков, однако в настоящей работе такой эксперимент не проводился.

Фенвиков массив разностей выполняет задачу прибавления константы на несколько порядков быстрее других структур, однако платой за это служит замедление времени доступа примерно в 40 раз.

Расцепленный и каскадный массивы демонстрируют производительность, уступающую аналогичной структуре данных из стандартной библиотеки std::deque.

Автор благодарит А. С. Болдарева за плодотворное сотрудничество.

Список литературы

- [1] Карташева Е.Л. Инструментальные средства подготовки и анализа данных для решения трехмерных задач математической физики // Математическое моделирование. 1997. Т. 9. № 7. С. 113–127.
- [2] Гасилов В.А., Болдарев А.С., Ольховская О.Г., Бойков Д.С., Шарова Ю.С., Савенко Н.О., Котельников А.М. MARPLE: программное обеспечение для мультифизического моделирования в задачах сплошных сред // Препринты ИПМ им. М.В. Келдыша. 2023. № 37. 41 с. DOI: [■](#).
- [3] Josuttis N.M. Boost.Array // [Электронный ресурс]. — URL: [■](#) (дата обращения: 20.11.2025).
- [4] cppreference.net. Inplace Vector // [Электронный ресурс]. — URL: [■](#) (дата обращения: 20.11.2025).
- [5] Fenwick P.M. A new data structure for cumulative frequency tables // Software: Practice and Experience. 1994. Т. 24, № 3. С. 327–336.
- [6] Рябко Б. Быстрый последовательный код // Доклады АН СССР. 1989. Т. 306, № 3. С. 548–552. URL: [■](#).
- [7] Abrahams David, Gurtovoy Aleksey. C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond // 2004. 400 с. ISBN 0-321-22725-5.

Содержание

1	Введение	3
2	Общие сведения о структурах данных, исследуемых в данной работе	4
3	Свойства структур данных	5
3.1	Статический массив	5
3.2	Динамический массив	6
3.3	Расщепленный массив	8
3.4	Каскадный массив	8
3.5	Фенвиков массив разностей	9
4	Реализация структур данных	10
4.1	"Любознательный" рекурсивный шаблон	10
5	Замеры производительности	12
5.1	Методика	12
5.2	Выводы	14