Содержание

Аннотация

Для изучения алгоритмов и структур данных часто не хватает инструмента, который бы давал возможность детально разобраться в алгоритме, не полагаясь исключительно на его техническое описание словами.

Мной была выбрана структура данных Персистентная очередь, которая широко применяется и известна в целом, но имеет сложные алгоритмы внутренней работы. Основная цель данной работы это предоставление понятного алгоритма и его визуальная реализация, для дополнительной показательности. С этой целью было разработано веб-приложение, которое отвечает основным требованиям к процессам визуализации и может применяться для наглядного изучения этой структуры данных.

Ключевые слова

Персистентная очередь, структура данных, визуализация алгоритма, алгоритм на графах, алгоритмическая оптимизация, пошаговость, односвязные списки.

1 Введение

Предмет «Алгоритмы и структуры данных» по праву считается одним из основополагающих в изучении компьютерных наук. Но для многих студентов он становится одним из самых сложных в прохождении. Большинство алгоритмов не имеют физического представления и являются абстракциями, что вызывает определенные трудности у студентов. От понимания студентом тех или иных алгоритмов зависит его общее представление о возможностях их применения в решениях реальных прикладных задач. Для облегчения учебного процесса и более лучшего понимания преподаватели прибегают к визуализации, чтобы показать пошаговую работу алгоритмов в конкретных случаях.

Однако, не все алгоритмы можно наглядно изобразить на доске. Особенные трудности представляют алгоритмы на графах. Их сложно рисовать, на каждый шаг может уходить слишком много времени. Доска ограничена в размерах и через некоторое время быстро заполняется. Нельзя вернуться к предыдущему шагу и посмотреть на него снова. А самое главное, если детально не запомнить подробности работы, то нельзя самому воспроизвести его работу после занятия. При воспроизведении механизма работы алгоритма нельзя исключать и вероятность случайной ошибки как студента, так и преподавателя.

Все вышеперечисленные проблемы можно решить с помощью современных решений — визуализации работы алгоритма в веб-приложении. Такой подход исключает человеческий фактор ошибки, рабочее пространство не ограничено пределами доски, а визуализация будет выглядеть четко и понятно. Все шаги выполнения комментируются, давая студенту возможность поэксперементировать самому с алгоритмом и посмотреть на его работу вне учебных занятий. Также, открываются новые возможности для рассмотрения работы на больших массивах данных. Например, построить граф на несколько сотен вершин, провести аналитику над ним и узнать общую эффективность алгоритма.

По этой причине визуализации алгоритмов будут всегда актуальны для обучения. С этим связан мой выбор реализовать веб-приложение по визуализации работы такой структуры данных, как персистентная очередь. Эта структура данных является одновременно прикладной в решении реальных задач и сложной в освоении из-за неочевидных алгоритмов взаимодействия с ней.

По проведенным мной исследованиям доступных решений, на данный момент не существует ни одной визуализации данной структуры данных в открытом доступе для пользователей. Данный факт увеличивает актуальность разработки такого веб-приложения, поскольку это даст возможность изучать методы работы с данной структурой данных, что невозможно

сделать с доступными в этот момент инструментами визуализации алгоритмов.

2 Постановка задачи

Необходимо разработать веб-приложение для визуализации Персистентной очереди, к которому предлагаются следующие требования:

- Веб-приложение должно в явном виде демонстрировать структуру графа, его связи, обозначения и состояние. Все вершины должны быть подписаны.
- Должна присутствовать возможность масштабирования изображения графа, его перемещения.
- Пошаговость визуализации должна контролироваться пользователем.
- Должна быть возможность откатов назад, для повторений шагов.
- Все шаги должны быть снабжены описанием того, что происходит.
- У веб-приложения должен существовать дополнительный интерфейс взаимодействия, помимо выполнения простых операций.

С этой целью основная задача разбивается на две основных подзадачи:

1 Выбор алгоритма взаимодействия с Персистентной очередью и способ ее хранения.

2 Разработка непосредственно веб-приложения, основанного на работе по выбранному ранее алгоритму.

3 Разработка алгоритма

3.1 Выбор алгоритма

После ознакомления с существующими распространенными вариантами реализации Персистентной очереди, можно заметить что почти подавляющее большинство - алгоритмы на разном количестве стеков. Например, на 4 [1] или на 6 [4](с возможной оптимизацией до 5). Также, существуют реализации на персистентном дереве отрезков и на персистентном декартовом дереве по неявному ключу.

Однако, по моему мнению, такие способы реализации сложны и требуют глубокого понимания, что мне не подходит. По этой причине, мной был разработан алгоритм на двух деревьях (на основе предложенного здесь [3]), который не только интуитивно понятнее, но и гораздо нагляднее ранее упомянутых.

3.2 Терминология

Далее по тексту будут применяться следующие слова, с измененным смыслом:

- Список — односвязный список в традиционном программном понимании.
- Очередь — то же, что и список.
- Сын — следующий по списку объект.
- Версия — объект, который хранит в себе данные об определённой очереди.
- Вершина - объект, содержащий информацию о текущем элементе и указатель на Сына.
- Целевая вершина — то же, что и вершина.
- Основное дерево — дерево (лес), представляющее собой непосредственно различные вариации очереди, состоит из Вершин.
- Вспомогательная вершина — объект, содержащий в себе указатель на Целевую вершину и указатель на Сына.
- Вспомогательное дерево — дерево (лес), представляющее собой вспомогательные списки для выполнения операций над очередями, состоит из Вспомогательных вершин.
- Хвост — последняя вершина в списке (она не содержится ни у кого в качестве сына).
- Ведущая вершина — то же, что и хвост.
- Голова — первая вершина списка (у нее нет сына).
- Промежуточная вершина — вершина, не являющаяся ни головой, ни хвостом.

3.3 Основная идея и описание

Везде в алгоритме объекты и элементы иммутабельны — после своего создания и завершения выполнения очередной операции становятся неизменяемыми.

Для хранения очереди воспользуемся Основным деревом. Эта структура используется и при построении персистентных стеков, где в каждой вершине записан непосредственно сам элемент и указатель на лежащий под ним (в терминологии очереди — "предшествующий ему"). Создадим пустой список объектов Версий — каждый такой объект представляет собой конкретное состояние очереди в определенный момент. Для этого, Версии достаточно состоять из 2 указателей — на Хвост и Голову очереди.

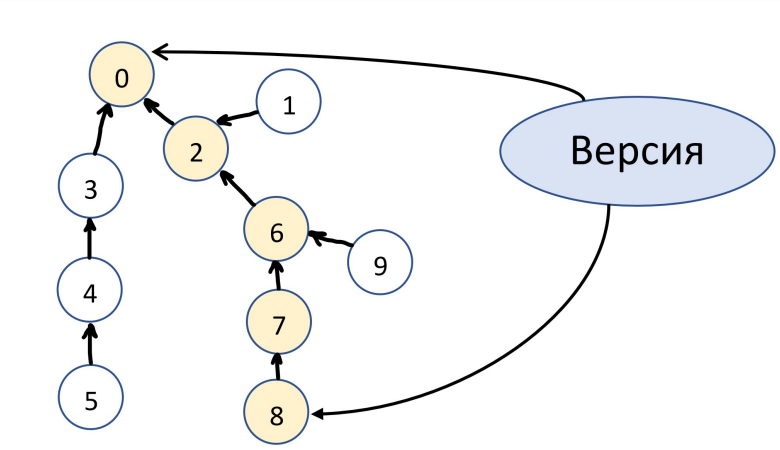

Рис. 3.1: Два указателя могут полностью охарактеризовать очередь

Нумерацию этого списка начнем с 0, добавив в список исходную Версию, заполненную так, чтобы она показывала пустую очередь — отсутствие каких либо элементов на начало работы алгоритма. Тогда, при работе можно обратиться к любой Версии, зная ее уникальный порядковый номер. Любая операция над нашей структурой данных будет создавать и добавлять в конец списка новую Версию, показывающую состояние очереди на момент завершения операции.

Зная номер существующей Версии, можно совершить над ней две основные операции:

- Удалить первый элемент из очереди.
- Добавить новый элемент в конец очереди.

Каждая из операций видоизменяет Основное дерево, при этом изменения не затрагивают элементы самой очереди и выбранную Версию. Таким образом, исходная очередь все равно доступна и продолжает храниться в соответствующей ей Версии.

Для реализации операции Добавления вершины в очередь такого интерфейса уже достаточно — каждая такая операция будет добавлять в Основное дерево новую вершину, подвязанную к хвостовой (ее мы можем узнать из указателей исходной Версии) и добавлять в список Версий копию исходной, с измененным указателем на хвостовую вершину.

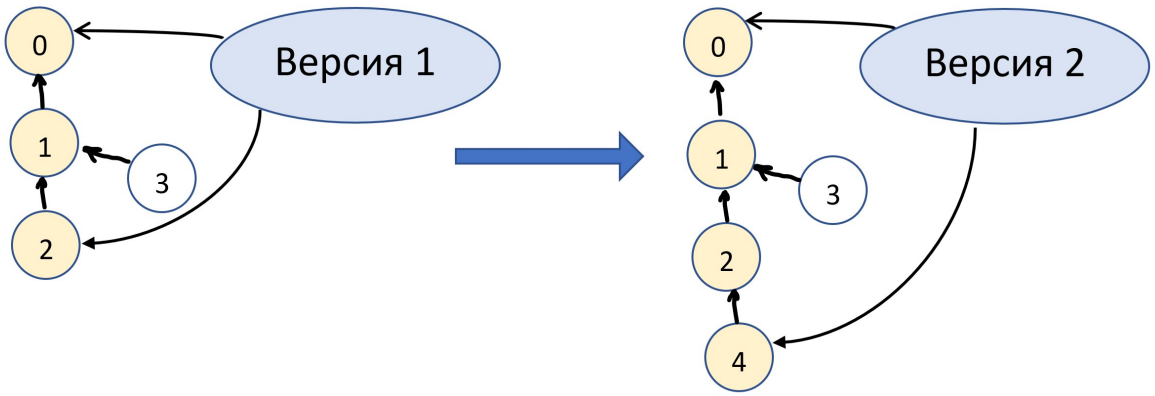

Рис. 3.2: Добавление вершины произвести легко

Однако, при операции Удаления возникнет проблема — из-за односвязности списков, мы не можем за константное время узнать, какая вершина станет следующей головой, так как для этого нам придется последовательно пройти по указателям от хвостовой вершины до вершины, предшествующей голове, чтобы узнать указатель на нее. Это может привести время выполнение операции к асимптотике $O(n)$, где n — количество элементов в выбранной очереди.

Чтобы избежать такой проблемы, добавим понятие вспомогательных списков Версии и модифицируем объекты Версий следующим образом: добавим в объект Версии указатель на такой список, что его хвостовой элемент хранит в себе указатель на вершину, следующую за головой Версии, следующий элемент — на третью вершину с начала и так далее (то есть список из промежуточных вершин очереди, развернутый в обратную сторону). Таким образом, мы будем знать, какая вершина следует за головой — достаточно просто посмотреть, на какую вершину Основного дерева указывает ведущая вершина вспомогательного списка.

Однако, на этом месте возникает новая проблема — как строить такой вспомогательный список? Мы не можем строить заново список промежуточных вершин на каждом шаге (иначе это приведет нас к старой асимптотике). Решим это добавлением в объект Версии, помимо указателя на вспомогательный список, нового указателя на строящийся вспомогательный список. Для удобства, существующий вспомогательный список будем называть Операционным, строящийся — Динамическим. Оба эти списка по своей природе являются

продолжениями друг друга, поэтому их можно хранить в одном Вспомогательном дереве.

После совершения какой либо операции над Версией, будем проверять, есть ли динамический список у этой Версии.

Если его нет, то проверяем, не надо ли начать строить его (критерий проверки и его корректность будут приведены в следующем разделе). Если такой необходимости нет, то можно заканчивать выполнение операции. Иначе, создаем вспомогательную вершину, в которой указатель на сына оставляем нулевым, а в указатель на целевую вершину записываем указатель на сына хвостовой вершины.

После этого проверяем, не надо ли поменять списки местами — для этого достаточно сделать проверку, что сын целевой вершины ведущей вершины динамического списка — голова Версии. Если это так, то подменяем указатель Версии на операционный список указателем на динамический, а указатель на динамический список — зануляем.

Если же динамический список уже начал строиться, то на каждой операции просто будем достраивать его. Но перед этим необходимо проверить, нельзя ли поменять операционный и динамический списки местами. Если поменять их удалось, то на этом выполнение операции прекращается. Иначе, создаем вспомогательную вершину, в которой указатель на сына выставляем на ведущую вершину динамического списка исходной Версии, а в указатель на целевую вершину записываем указатель на сына целевой вершины ведущей вершины динамического списка исходной версии. Снова делаем попытку смены списков друг на друга.

Тогда, операция Удаления будет проходить следующим образом:
Если размер очереди 1, то достаточно скопировать нулевую Версию в конец списка (созданная версия будет пуста, то есть эквивалентна изначальной).
Если размер очереди 2, то достаточно в новой Версии в указатель на голову записать указатель Версии на хвост.
Иначе, если размер > 2, в указатель на голову новой Версии записываем указатель на целевую вершину ведущей вершины операционного списка Версии. В указатель на операционный список Версии записываем указатель на сына ведущей операционной вершины Версии. При необходимости достраиваем динамический список.

Таким образом, удаление вершины и возможное достраивание динамического списка происходят за константное время, как и результирующая операция Удаления, что и хотели получить.

3.4 Критерий построения динамического списка

Для определения необходимости начала построения динамического списка будем пользоваться следующим условием:

Если длина операционного списка $< \frac{1}{2}$ *от количества промежуточных вершин очереди, то необходимо начать строить динамический список.*

Покажем, почему выполнение данного критерия позволяет всегда поддерживать знание о том, какая вершина следует за головой очереди и не произойдет ситуации, когда операционный список уже закончился, а динамический не успел достроиться до головы. Для этого докажем, что для любой Версии поддерживается уравнение

$$2p + d = r, \tag{1}$$

где p и d - количество вершин в операционном и динамическом списках соответственно, а r - количество промежуточных вершин очереди. Доказательство по матиндукции:

База:

На первом шаге, когда мы начинаем конструирование динамического списка, в нем находится одна вершина, то есть $d_{new} = 1$. Для текущей очереди выполняется $2p_{new} - r_{new} < 0$, для ее родительской Версии выполняется $2p_{old} - r_{old} \geqslant 0$, так как на предыдущем шаге критерий еще не был выполнен и динамический список не начинал строиться. Рассмотрим два возможных варианта:

1. Пусть для получения новой Версии была использована операция Добавления. В таком случае, $p_{new} = p_{old}$ (операционный список в данном случае не менялся), $r_{old} + 1 = r_{new}$ (добавление вершины в очередь увеличивает количество промежуточных). Тогда, получаем

$$2p_{new} - r_{new} = 2p_{old} - (r_{old} + 1) \Rightarrow 2p_{new} - r_{new} + 1 = 2p_{old} - r_{old} = 0,$$

то есть $2p_{new} + 1 = r_{new}$, что совпадает с условием $d_{new} = 1$.

2. Пусть для получения новой Версии была использована операция Удаления. В таком случае, $p_{new} = p_{old} - 1$, (операционный список в данном случае уменьшился на 1 из-за перехода), $r_{new} = r_{old} - 1$ (удаление вершины из очереди уменьшает количество промежуточных). Тогда, получаем

$$2p_{new} - r_{new} = 2(p_{old} - 1) - (r_{old} - 1) = 2p_{old} - r_{old} - 1 \Rightarrow 2p_{new} - r_{new} + 1 = 2p_{old} - r_{old} = 0,$$

то есть $2p_{new} + 1 = r_{new}$, что совпадает с условием $d_{new} = 1$.

База доказана.

Переход:

Пусть это равенство выполняется для какого-то шага алгоритма, когда динамический список уже есть. Рассмотрим случаи:

Пусть для получения новой Версии была использована операция Добавления. В таком случае, $p_{old} = p_{new}$ (операционный список в данном случае не менялся), $r_{old} = r_{new} - 1$ (добавление вершины в очередь увеличивает количество промежуточных). Тогда, получаем

$$2p_{old} - r_{old} = 2p_{new} - (r_{new} - 1) \Rightarrow 2p_{old} - r_{old} = 2p_{new} - r_{new} + 1,$$

то есть $d_{old} + 1 = d_{new}$, что и достигается добавлением динамической вершины в алгоритме.

Пусть для получения новой Версии была использована операция Удаления. В таком случае, $p_{old} = p_{new} + 1$, (операционный список в данном случае уменьшился на 1 из-за перехода), $r_{old} = r_{new} + 1$ (удаление вершины из очереди уменьшает количество промежуточных). Тогда, получаем

$$2p_{old} - r_{old} = 2(p_{new} + 1) - (r_{new} + 1) = 2p_{new} - r_{new} + 1,$$

то есть $d_{old} + 1 = d_{new}$,что и достигается добавлением динамической вершины в алгоритме.

Переход доказан.

Так как проблемы могут возникнуть только при операции Удаления (вспомогательные списки используются только для нее), то докажем от противного для этой ситуции. Пусть мы хотим совершить операцию Удаления, но у нас нет операционного списка. Тогда, возможны 2 случая:

1 Динамический список не был построен для данной Версии.

2 Динамический список не успел достроиться до головы очереди.

В первом случае, если динамический список не начинал строиться, то выполняется $2p \geqslant r$, но $p = 0 \Rightarrow r = 0 \Rightarrow$ в очереди нет промежуточных вершин, то есть размер очереди $\in [0, 2]$, но в таком случае нам не нужен операционный список, так как для очередей размера $\leqslant 2$ операция Удаления не задействует вспомогательные списки.

Во втором случае, динамический список имеет ненулевой размер. Воспользуемся полученным равенством (1) и применим его, тогда в силу $p = 0$ получаем $d = r$, но это означает, что динамический список был достроен до головы и должен был подменить операционный

еще на предыдущем шаге.

Таким образом, приведенная по условию ситуация при Удалении невозможна.

Последним шагом покажем, что при достижении головы очереди на момент подмены операционного списка динамическим, его длина $2d \geqslant r$, то есть не возникнет ситуации, что построенный новый операционный список будет слишком коротким и построение следующего динамического списка запоздает. $2d \geqslant r \Leftrightarrow d \geqslant r - d$, а это число является количеством промежуточных вершин, не содержащихся на данный момент в динамическом списке. Так как промежуточные вершины возникают только при выполнении операции Добавления, а динамический список будет пополняться на каждом шаге, то несложно понять, что неравенство выполняется, то есть при обмене операционного списка на динамический будет гарантирована его необходимая длина.

То есть применение данного критерия в алгоритме корректно.

3.5 Итоговая версия алгоритма

Приведем полученную нами итоговую версию алгоритма:

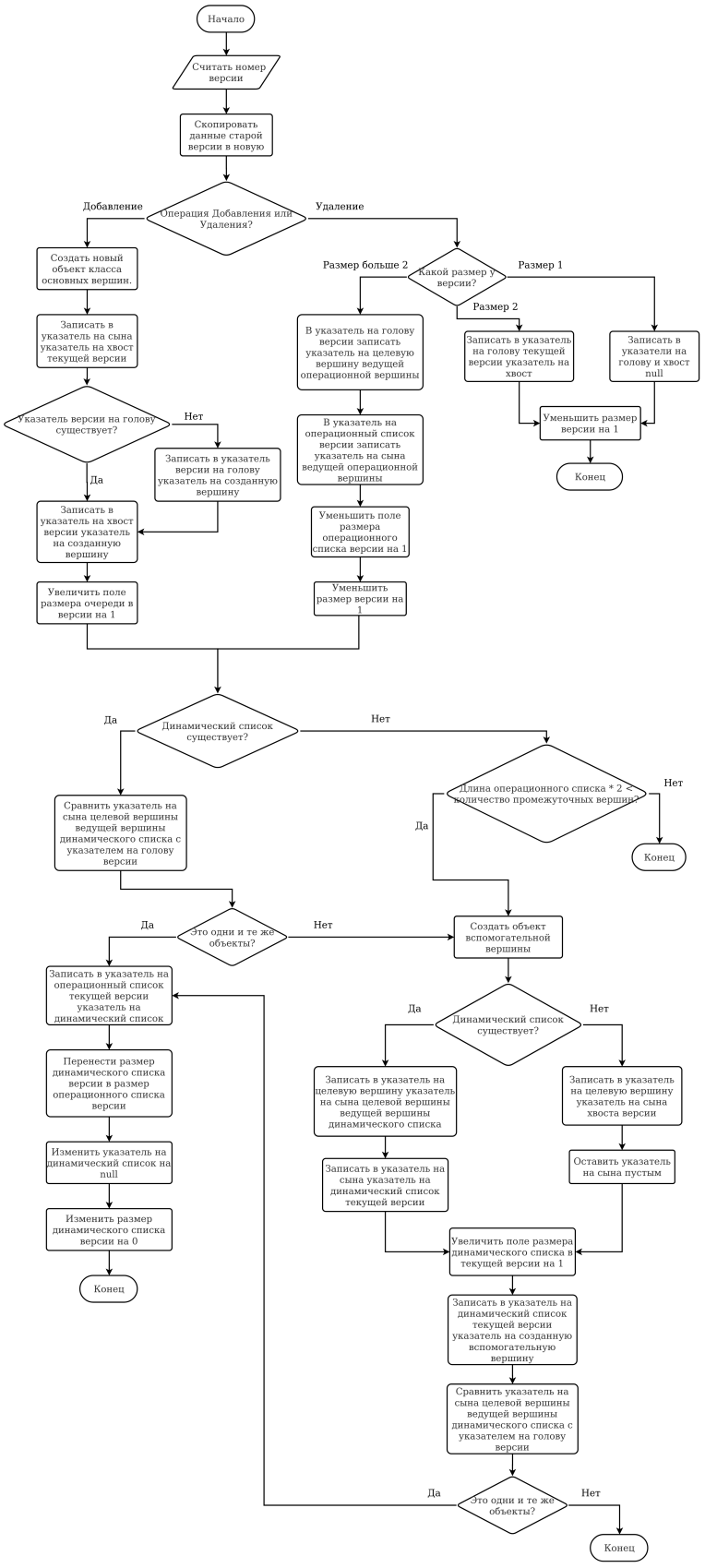

Рис. 3.3: Общая блок-схема работы алгоритма

3.6 Код алгоритма на C++

Приведем полученный возможный код алгоритма на языке C++:

```
#include "vector"

struct QueueNode {
    QueueNode *son;
    int64_t value;
};

struct ListNode {
    ListNode *next_list;
    QueueNode *target_node;
    int64_t size;
};

struct Version {
    QueueNode *head;
    QueueNode *tail;
    ListNode *operational_list;
    ListNode *dynamic_list;
    int64_t size;
};

class PersistentQueue {
public:
    explicit PersistentQueue(int64_t num_of_actions) {
        versions.reserve(num_of_actions + 2);
        versions.push_back(new Version({nullptr, nullptr, nullptr, nullptr, 0}));
    }

    void Push(int64_t parent_queue_num, int64_t value_to_push) {
        Version *parent = versions[parent_queue_num];
        auto *new_node = new QueueNode({parent->tail, value_to_push});
        QueueNode *first_el = parent->size ? parent->head : new_node;
        versions.push_back(new Version({first_el,
                                        new_node,
                                        parent->operational_list,
                                        parent->dynamic_list,
                                        parent->size + 1}));
        ChangeDynamicList();
```

```
    }

    void Pop(int64_t parent_queue_num) {
        Version *parent = versions[parent_queue_num];
        if (parent->size == 1) {
            versions.push_back(new Version({nullptr,
                                            nullptr,
                                            nullptr,
                                            nullptr,
                                            0}));
        } else if (parent->size == 2) {
            versions.push_back(new Version({parent->tail,
                                            parent->tail,
                                            nullptr,
                                            nullptr,
                                            1}));
        } else {
            versions.push_back(new Version({parent->operational_list->target_node,
                                            parent->tail,
                                            parent->operational_list->next_list,
                                            parent->dynamic_list,
                                            parent->size - 1}));
            ChangeDynamicList();
        }
    }

private:
    void ChangeDynamicList() {
        Version *current = versions[versions.size() - 1];
        if (current->dynamic_list) {
            if (!ListsExchange()) {
                QueueNode *new_dynamic_node =
                        current->dynamic_list->target_node->son;
                int64_t new_size = current->dynamic_list->size + 1;
                current->dynamic_list = new ListNode({current->dynamic_list,
                                                      new_dynamic_node,
                                                      new_size});
                ListsExchange();
            }
            return;
```

```
        }
        int64_t operational_list_size = current->operational_list ?
                                        current->operational_list->size
                                        : 0;
        int64_t curr_version_nodes_size = std::max(NULL, current->size - 2);
        if (operational_list_size * 2 < curr_version_nodes_size) {
            QueueNode *new_dynamic_node = current->tail->son;
            current->dynamic_list = new ListNode({nullptr,
                                                  new_dynamic_node,
                                                  1});
            ListsExchange();
        }
    }

    bool ListsExchange() {
        Version *current = versions[versions.size() - 1];
        if (current->dynamic_list->target_node->son == current->head) {
            current->operational_list = current->dynamic_list;
            current->dynamic_list = nullptr;
            return true;
        }
        return false;
    }

    std::vector<Version *> versions;
};
```

3.7 Оценка производительности

Каждая операция выполняется за константное время (что видно из блок-схемы), кроме копирования Версии в конец их списка. Так как нам необходим произвольный доступ к различным элементам этого списка, самым оптимальным решением будет обычный массив, в который вставка происходит за $O(1)$ амортизированное. Таким образом, итоговая оценка сложности по времени выполнения всех операций — $O(1)$ амортизированное.

На каждую операцию создается новый объект Версии, а так же добавляются вершины в граф. В худшем случае, на каждую операцию будет создаваться как обычная вершина очереди, так и новая вспомогательная для динамического списка, что в итоге приводит к сложности алгоритма по памяти $O(3n) \sim O(n)$. Приведем результаты сравнения работы

предложенного алгоритма с другими распространенными вариантами:

	Время работы			Затраченная память		
	Макс.	Среднее	Медианное	Макс.	Среднее	Медианное
Персистентное дерево отрезков	367ms	214ms	210ms	197.7Mb	102.4Mb	75.7Mb
Персистентное декартово дерево по неявному ключу	351ms	197ms	201ms	135.2Mb	87.3Mb	61.9Mb
Алгоритм на 6 стеках	230ms	139ms	131ms	57.7Mb	34.3Mb	38.1Mb
Мой алгоритм	99ms	69ms	87ms	19.5Mb	13.7Mb	18.0Mb
Отличие от лучшего значения	**131ms**	**70ms**	**44ms**	**38.2Mb**	**20.6Mb**	**20.1Mb**

Таблица 3.1: Статистика работы различных алгоритмов обработки Персистентной очереди на одном и том же наборе данных

4 Разработка веб-приложения

После выбора алгоритма, на котором будет строиться объяснение и визуализация, появилась возможность перейти к разработке самого веб-приложения. Условно можно разбить всю его реализацию на несколько частей:

1 Поиск и изучение библиотеки для визулизации графов.

2 Перенос алгоритма на JavaScript с добавлением необходимых данных для подробной визуализации внутренней работы.

3 Добавление инструментов визуальной помощи в объяснении шагов алгоритма.

4 Добавление интерфейса взаимодействия с HTML-страницей.

5 Стилизация элементов с помощью CSS.

6 Реализация дополнительных возможностей для пользователя.

4.1 Библиотека для визулизации графа

Для реализации динамических анимаций с графами мне была необходима библиотека, способная работать с такими данными. Из разных вариантов больше всего мне подошла библиотека D3 [2], которая дает широкий выбор возможностей и удобный программный интерфейс взаимодействия (по сравнению со своими аналогами, например Cytoscape).

После этого, мне было необходимо изучить способы отображения графов, их симуляцию, поведение и какие данные необходимы для их корректного отображения. Мне был необходим стиль визуализации force-directed graph layout — способ рисовки графа, в котором поддерживаются связи между вершинами, при этом каждая вершина имеет отрицательный коэффициент притяжения (отталкивается от других), а каждое ребро имеет положительный коэффициент (притягивает соединенные вершины друг к другу). Таким образом, граф сохраняет целостность и не происходит наложение вершин, либо пересечение ребер друг с другом.

Также, любая перестройка графа сопровождается определенным периодом симуляции его обновления. В течение этого времени силы притяжения активны и под их влиянием граф может видоизменяться. Традиционно, параметр отвечающий за время перестройки изменяется по принципу отжига. После того, как этот параметр достигнет определенного уровня, перестройка завершается и положение графа фиксируется.

4.2 Перенос алгоритма на JavaScript

Так как JavaScript не имеет возможностей прямых манипуляций с памятью, то реализовать алгоритм полностью в исходном виде не получится. По этой причине все структуры, которые предполагали хранение в памяти и адресацию по указателям, будут храниться в обычных массивах и в качестве адресации использовать индексы, по которым эти элементы можно найти. Это решение незначительно влияет на скорость работы, так как операции с массивами происходят за амортизированно константное время. Таким образом, интерфейс классов объектов Версий и вершин Основного и Вспомогательного деревьев логически не изменился, по сравнению с исходной реализацией.

Но так как визуализация должна поддерживать пошаговость, необходимо добавить остановки после каждого действия, чтобы дать пользователю возможность контролировать ход выполнения алгоритма. Также требуется реализовать и откаты назад по шагам алгоритма, что будет сложно сделать в рамках одной функции на программном уровне, поэтому был реализован следующий механизм работы:
Все функции разбиты на цепочные — каждая из функций возвращает указатель на следующую функцию для вызова.
На вызове каждой операции, запускается координатор выполнения этих функций. После выполнения каждой из них, в зависимости от решения пользователя, он либо продолжает вызовы следующих функций, либо производит откат назад с восстановлением параметров.

Для восстановления состояния окружения на момент вызова каждой из функций, был реализован класс Состояний, который перед вызовом любой функции производит копирование необходимых параметров и аргументов вызова, если таковые имелись. В случае, когда происходит откат назад, в переменные загружаются сохраненные значения и происходит повторный вызов функции с сохраненными аргументами.

4.3 Инструменты помощи в объяснении шагов

Для большей наглядности требуется каждый шаг сопровождать подсказками для пользователя. Это облегчает понимание алгоритма и его ход работы, при этом позволяет не изучать его техническую работу в деталях. С этой целью были созданы следующие инструменты:

Подсветка Версий — такая подсветка выделяет соответствующую выбранной Версии очередь и ее вспомогательные списки, при этом затемняя элементы, не относящиеся к данной Версии.

Подсветка вершин и ребер — если на каком-либо шаге алгоритма происходит изменение графа, либо же производится переход и поиск данных в вершинах, то они визуально выделяются, для подчеркивания с какой именно вершиной/ребром идет работа в данный момент.

Внутренняя информация объектов — при наведении на различные объекты (Версии, вершины основного и вспомогательного деревьев) выдается внутренняя информация, которую хранят данные объекты. Для Версий это "указатели"на начало, конец очереди, а также на вспомогательные списки. Для вершин это их сыновья, для вспомогательных вершин дополнительно показываются целевые вершины в Основном графе.

Текстовое сопровождение — на каждом шаге добавлено текстовое объяснение текущего шага и логика его работы, позволяющее проследить цепочку логического выполнения алгоритма.

4.4 Взаимодействие с HTML-страницей

Для видоизменения страницы разработан интерфейс взаимодествия с ней. При определенных операциях убираются лишние объекты, либо наоборот возвращаются.

Также сделана система перевода элементов на разные языки — для этого достаточно только загрузить необходимый .json файл с объявлениями переводов элементов, после чего страница может быть переведена на этот язык. Добавленный язык можно выбрать из меню приложения, где с этой целью создан выпадающий список доступных переводов.

Чтобы добавить возможность совершения различных операций, добавлены соответствующие

кнопки и поля для заполнения.
Дополнительные файлы для отображения на странице, такие как картинки, могут быть добавлены из соответствующей папки.

4.5 Дополнительные возможности

Для удобства пользователя создана инструкция, в которой полностью описан интерфейс приложения и способы взаимодействия с ним. Создан сборник готовых наборов данных, которые выполняют определенные операции, наглядно демонстрирующие различные поведения алгоритма.
Пользователь может также создать случайный граф очередей, задав некоторые параметры его создания (например, размер или характер создания).
Есть возможность переключения между режимами работы — пошаговой работой алгоритма и мгновенным выполнением всех операций.
Дополнительные незначительные функции, вроде удаления всех данных приложения или обновления визуализации графов.

5 Заключение

В результате проведенной работы было получено полностью работоспособное приложение, которое может выполнять все заложенные в него требования. Из-за специфики темы и выбора алгоритма, на данный момент нет каких-либо аналогов данного приложения, либо его возможных альтернатив на других алгоритмах. В дальнейшем, данное веб-приложение может поддерживаться и обновляться, так как создана базовая система, в которую можно добавлять улучшения без больших затрат по времени.

5.1 Возможные улучшения и оптимизации алгоритма

В данном алгоритме есть возможность уменьшения использования дополнительной памяти, в нескольких случаях:

Если выполннить две операции Добавления к одной и той же Версии, то если выполняется критерий построения динамического списка (либо список уже начат), во Вспомогательный граф будут добавлены две абсолютно идентичные вершины. В таком случае, можно было бы создать вспомогательную вершину только для первой операции, а для второй — обратиться к уже существующей. Это может сильно уменьшить потребление памяти

при выполнении множества операций, примененных к узкому набору Версий.

Если последовательно выполнять операции Добавления к Версии, у которой уже есть операционный список, то в динамический список с какого то момента будут добавляться вершины, которые уже есть в операционном списке (это происходит из-за того, что динамический список стремится пройти до головной вершины Версии, а не последней вершины операционного списка). Добавив дополнительную информацию в объект Версии о том, какая вершина завершает операционный список, можно сильно уменьшить потребление памяти при последовательных монотонных операциях.

Однако, в силу того, что обе эти оптимизации требуют либо дополнительной структуры данных о существующих связях (первая оптимизация), либо дополнительных полей в объектах (вторая оптимизация), то их результативность неоднозначна и должна применяться в соответствии со спецификой требований от алгоритма. Где-то требуются множественные изменения к нескольким Версиям, а где-то — изменения последовательно применяются к цепочке Версий. В зависимости от варианта работы, будет целесообразно реализовать ту или иную оптимизацию.

5.2 Возможные улучшения веб-приложения

На данный момент приложение визуализирует только хранение по одному алгоритму. Если подготовить новые алгоритмы и их визуальное сопровождение, то будет несложно интегрировать их в существующую систему. При наличии нескольких алгоритмов, можно создать возможность для пользователя строить графы на больших объемах данных и сравнивать технические характеристики алгоритмов (такие как потребление памяти, затраченное время на исполнение и так далее).

Список литературы

[1] burunduk3. *Персистентная очередь и её друзья*. URL: https://codeforces.com/blog/entry/15685 (дата обр. 24.05.2023).

[2] D3. *D3.js - Data-Driven Documents*. URL: https://d3js.org/ (дата обр. 24.05.2023).

[3] lemelisk. *Персистентная очередь*. URL: https://habr.com/ru/articles/241231/ (дата обр. 24.05.2023).

[4] NEERC-IFMO. *Персистентная очередь*. URL: https://neerc.ifmo.ru/wiki/index.php?title=%D0%9F%D0%B5%D1%80%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BD%D1%82%D0%BD%D0%B0%D1%8F_%D0%BE%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C (дата обр. 24.05.2023).