

Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО  
ITMO University

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА/GRADUATION THESIS

Разработка конкурентных структур данных, дружелюбных к памяти

**Автор/ Author**

Смирнов Роман Алексеевич

**Направленность (профиль) образовательной программы/Major**

Информатика и программирование 2017

**Квалификация/ Degree level**

Бакалавр

**Руководитель ВКР/ Thesis supervisor**

Аксенов Виталий Евгеньевич, PhD, науки, Университет ИТМО, факультет информационных технологий и программирования, программист

**Группа/Group**

М3437

**Факультет/институт/кластер/ Faculty/Institute/Cluster**

факультет информационных технологий и программирования

**Направление подготовки/ Subject area**

01.03.02 Прикладная математика и информатика

Обучающийся/Student

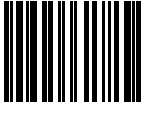
Документ подписан	
Смирнов Роман Алексеевич	
27.05.2021	

(эл. подпись/ signature)

Смирнов Роман  
Алексеевич

(Фамилия И.О./ name  
and surname)

Руководитель ВКР/  
Thesis supervisor

Документ подписан	
Аксенов Виталий Евгеньевич	
27.05.2021	

(эл. подпись/ signature)

Аксенов  
Виталий  
Евгеньевич

(Фамилия И.О./ name  
and surname)

**Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО  
ITMO University**

**ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ /  
OBJECTIVES FOR A GRADUATION THESIS**

**Обучающийся / Student** Смирнов Роман Алексеевич

**Группа/Group** M3437

**Факультет/институт/кластер/ Faculty/Institute/Cluster** факультет информационных технологий и программирования

**Квалификация/ Degree level** Бакалавр

**Направление подготовки/ Subject area** 01.03.02 Прикладная математика и информатика

**Направленность (профиль) образовательной программы/Major** Информатика и программирование 2017

**Специализация/ Specialization**

**Тема ВКР/ Thesis topic** Разработка конкурентных структур данных, дружелюбных к памяти

**Руководитель ВКР/ Thesis supervisor** Аксенов Виталий Евгеньевич, PhD, науки, Университет ИТМО, факультет информационных технологий и программирования, программист

**Срок сдачи студентом законченной работы до / Deadline for submission of complete thesis** 31.05.2021

**Техническое задание и исходные данные к работе/ Requirements and premise for the thesis**

В реализациях конкурентных структур данных таких, как дерево поиска и список с пропусками, в узлах обычно хранится только одно значение, что приводит к использованию существенного количества вспомогательной памяти, описывающей структуру. В данной работе необходимо реализовать конкурентный список с пропусками на основе к-массивов, то есть в каждом узле будет храниться не одно значение, а массив значений. Предполагается, что использование такого подхода позволит повысить эффективность обращения к памяти за счет уменьшения кэш-промахов и числа аллокаций/деаллокаций памяти, что должно положительно отразиться на производительности. Требуется проанализировать, как использование такой техники влияет на производительность по сравнению с конкурентными структурами данных, хранящими одно значение в узле.

**Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов)/ Content of the thesis (list of key issues)**

Работа должна содержать:

1. Анализ существующих решений конкурентных списков с пропусками.
2. Реализованный конкурентный список с пропусками на основе к-массивов.
3. Сравнение полученного решения с существующими.

**Перечень графического материала (с указанием обязательного материала) / List of graphic materials (with a list of required material)**

Графические материалы и чертежи в работе не предусмотрены.

**Исходные материалы и пособия / Source materials and publications**

[1] Vitaly Aksenov, Nikita Koval, Petr Kuznetsov:

Memory-Optimality for Non-Blocking Containers (2020), in submission,

[https://www.dropbox.com/s/tfox480zabnzf6t/Lock\\_Free\\_Bounded\\_Queue%20%2815%29.pdf?dl=0](https://www.dropbox.com/s/tfox480zabnzf6t/Lock_Free_Bounded_Queue%20%2815%29.pdf?dl=0)

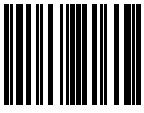
[2] Ruslan Nikolaev. A scalable, portable, and memory-efficient lock-free fifo queue. In DISC 2019.

[3] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In SPAA 2001

**Дата выдачи задания/ Objectives issued on 20.04.2021**

**СОГЛАСОВАНО / AGREED:**

Руководитель ВКР/  
Thesis supervisor

Документ подписан	
Аксенов Виталий Евгеньевич	
20.04.2021	

Аксенов  
Виталий  
Евгеньевич

(эл. подпись)

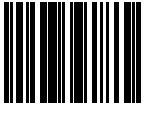
Задание принял к  
исполнению/ Objectives  
assumed by

Документ подписан	
Смирнов Роман Алексеевич	
23.04.2021	

Смирнов Роман  
Алексеевич

(эл. подпись)

Руководитель ОП/ Head  
of educational program

Документ подписан	
Станкевич Андрей Сергеевич	
18.05.2021	

Станкевич  
Андрей  
Сергеевич

(эл. подпись)

**Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО  
ITMO University**

**АННОТАЦИЯ  
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ /  
SUMMARY OF A GRADUATION THESIS**

**Обучающийся/ Student**

Смирнов Роман Алексеевич

**Наименование темы ВКР / Title of the thesis**

Разработка конкурентных структур данных, дружелюбных к памяти

**Наименование организации, где выполнена ВКР/ Name of organization**

Университет ИТМО

**ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ/  
DESCRIPTION OF THE GRADUATION THESIS**

**1. Цель исследования / Research objective**

Исследовать новый подход при работе с конкурентными списками с пропусками.

**2. Задачи, решаемые в ВКР / Research tasks**

Разработать конкурентный списка с пропусками, хранящего в узлах не одно значение, а массив значений. Реализовать разработанную структуру данных. Сравнить реализованную структуру данных с существующими решениями. Сделать выводы об эффективности нового подхода при работе с конкурентными списками с пропусками.

**3. Краткая характеристика полученных результатов / Short summary of results/conclusions**

Был разработан и реализован новый подход в использовании конкурентных списков с пропусками с использованием массивов в узлах для хранения значений. Подход показал себя на уровне существующих решений при использовании на небольших множествах хранимых в структуре элементов, но также показал высокую производительность с ростом размера хранимого множества и числа конкурирующих запросов в сравнении с существующими неблокирующими реализациями конкурентных списков с пропусками. Также удалось существенно сократить занимаемую списком с пропусками память за счет уменьшения количества узлов.

**4. Наличие публикаций по теме выпускной работы/ Have you produced any publications on the topic of the thesis**

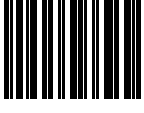
**5. Наличие выступлений на конференциях по теме выпускной работы/ Have you produced any conference reports on the topic of the thesis**

**6. Полученные гранты, при выполнении работы/ Grants received while working on the thesis**



## 7. Дополнительные сведения/ Additional information

Обучающийся/Student


Документ подписан	
Смирнов Роман Алексеевич	
27.05.2021	

(эл. подпись/ signature)

Смирнов Роман  
Алексеевич

(Фамилия И.О./ name  
and surname)

Руководитель ВКР/  
Thesis supervisor

Документ подписан	
Аксенов Виталий Евгеньевич	
27.05.2021	

(эл. подпись/ signature)

Аксенов  
Виталий  
Евгеньевич

(Фамилия И.О./ name  
and surname)

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ .....	5
1. Обзор предметной области .....	6
1.1. Формулировки и определения, используемые в работе .....	6
1.2. Сравнение деревьев поиска и списков с пропусками .....	9
1.2.1. Конкурентный список с пропусками стандартной библиотеки языка программирования <i>Java</i> .....	10
1.2.2. A Contention-Friendly Non-Blocking Skip List .....	11
Выводы по главе 1 .....	12
2. Разработка структуры .....	13
2.1. Базовые структуры .....	13
2.2. Описание операций над структурой .....	15
2.2.1. Вспомогательные операции .....	17
2.3. Разбор алгоритмов операций .....	18
2.3.1. Добавление и удаление элемента .....	18
2.3.2. Поиск элемента .....	22
Выводы по главе 2 .....	23
3. Корректность. Производительность. Рассмотренные инварианты узла.	24
3.1. Корректность операций .....	24
3.1.1. Вспомогательные операции .....	24
3.1.2. Добавление и удаление элемента .....	24
3.1.3. Поиск элемента .....	26
3.2. Сравнение .....	27
3.3. Рассмотренные решения .....	30
Выводы по главе 3 .....	31
ЗАКЛЮЧЕНИЕ .....	32
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....	33
ПРИЛОЖЕНИЕ А. Приложение с листингами функций .....	35
ПРИЛОЖЕНИЕ Б. Приложение с результатами сравнения .....	41

## ВВЕДЕНИЕ

Начиная с 2000-х годов, в эксплуатацию начали вводиться первые многоядерные процессоры. Они были созданы с целью продолжить увеличивать производительность процессоров, так как наращивание ее путем увеличения показателей одного вычислительного ядра стало затруднительно из-за экономических, технологических и производственных ограничений.

Цель была достигнута: потенциальная пиковая производительность увеличивалась пропорционально увеличению количества ядер. Однако достижимость этой пиковой производительности на практике стала непростой задачей. Об одном из ограничений, накладывающих верхнюю границу производительности на параллельную программу, сообщает закон Амдала [2]. Поэтому для получения максимальной производительности в условиях многопоточных программ необходимо использовать специальные алгоритмы и структуры данных, способные эффективно, а главное, корректно работать в многопоточном коде.

Одной из часто встречаемых задач в программировании является хранение и выполнение запросов над упорядоченным множеством элементов. Популярным решением такой задачи является использование различных видов деревьев поиска или списков с пропусками. Однако, в условиях многопоточных программ, производительность деревьев поиска плохо масштабируется с увеличением числа конкурентных запросов, так как необходимо поддерживать балансировку деревьев. В то же время, производительность списков с пропусками с увеличением числа потоков в системе растет гораздо лучше. Это можно объяснить тем, что все операции над списком сильно локализованы внутри структуры, то есть операции над узлом затрагивают только соседей этого узла.

В этой работе представлена новая версия конкурентного списка с пропусками, ключевым отличием которого будет хранение в узлах массива значений. Структура способна эффективно и корректно выполнять конкурентные запросы, а также позволяет существенно уменьшить количество занимаемой памяти по сравнению с существующими реализациями.

## ГЛАВА 1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

В этой главе будут рассмотрены особенности таких структур данных как деревья поиска и списки с пропусками, а также отмечены их основные особенности и отличия. Будет приведен анализ существующих реализаций списков с пропусками, отмечены их отличительные черты, а также выделены сильные и слабые стороны. Также будут даны формулировки и определения, используемые в работе.

### 1.1. Формулировки и определения, используемые в работе

*Список с пропусками* — структура данных, хранящая в себе упорядоченное множество элементов и способная выполнять операции над ним. Состоит из множества узлов, хранящих элементы множества и объединенных в уровни списка с пропусками, представляющих собой связные списки. Уровни имеют ссылки на низлежащие уровни лишь в рамках башен узлов. Пример списка с пропусками представлен на рисунке 1.

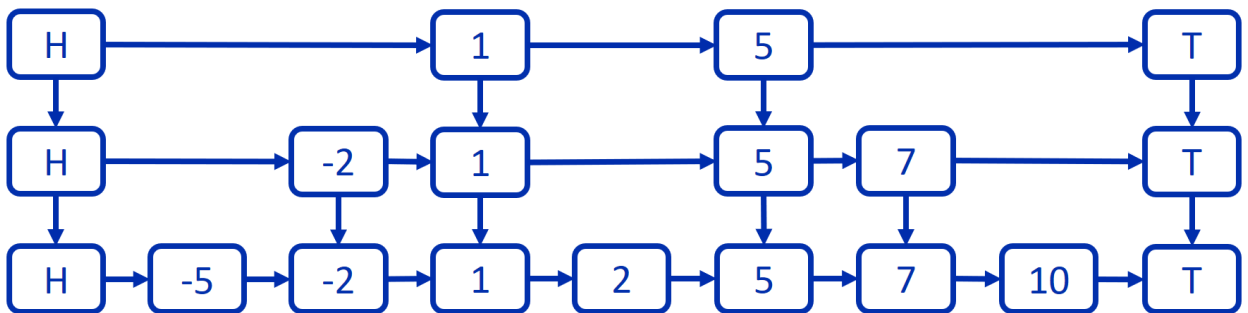


Рисунок 1 – Пример списка с пропусками

*Уровень списка с пропусками* — связный список узлов, хранящих значения множества. Пример уровня списка выделен на рисунке 2.

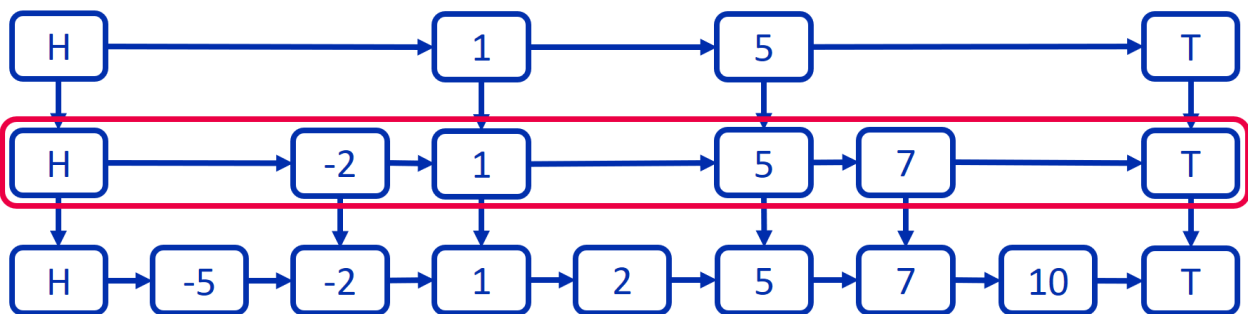


Рисунок 2 – Уровень списка с пропусками

*Башня узлов* — это набор узлов, которые связаны между собой в список и хранят/соответствуют одному и тому же элементу. Пример башни узлов выделен на рисунке 3.

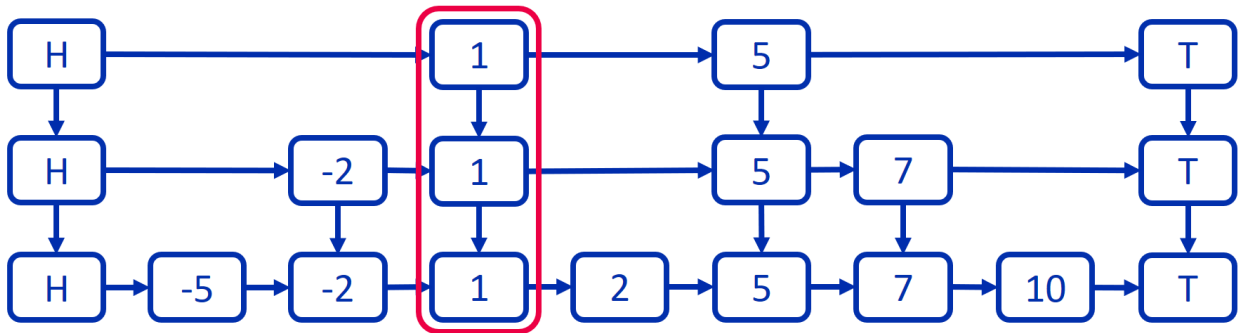


Рисунок 3 – Башня узлов

*Логическое представление списка* — представление списка, в котором все узлы упорядочены слева на право по хранимым в них элементам, а узлы, соответствующие башне узлов, расположены друг под другом. Пример логического представления списка на рисунке 1.

*Алгоритм без блокировок* — это алгоритм, в котором в любой момент времени гарантируется системный прогресс.

Узел будем называть *достижимым*, если он может быть достигнут из самого левого и верхнего узла в логическом представлении списка только лишь по ссылкам направленным слева-направо и сверху-вниз в логическом представлении списка (в различных реализациях списков с пропусками могут быть ссылки направленные и в другие направления).

Элемент будем называть *достижимым*, если он находится в достижимом узле.

*Физически удаленный узел* — узел, который не достижим.

*Логически удаленный узел* — узел, который в дальнейшем будет удален физически.

*Пустой элемент/значение* — это объект, который не входит в множество элементов, подмножество которого будет храниться в списке с пропусками.

*Исполнение системы* — это множество атомарных событий, например, событие завершения операции.

*Высокоуровневая история* — это подмножество исполнения системы, включающие только события начала  $inv(e)$  и завершения  $res(e)$  операций.

“*Произошло до*” — отношение частичного строгого порядка на множестве операций. Пусть  $e, h$  — операции, например, чтение или запись в ячейку памяти. Каждую операцию можно рассмотреть как множество упорядоченных атомарных событий. В каждой операции можно отдельно выделить два события  $inv(e)$  и  $res(e)$  — вызов операции  $e$  и ее результат соответственно. Тогда говорят, что операция  $e$  произошла до операции  $h$  тогда и только тогда, когда событие  $res(e)$  предшествовало событию  $inv(h)$  [1].

*Последовательное исполнение* — исполнение последовательно, если все операции линейно упорядочены отношением “*произошло до*” [1].

*Линеаризуемость* [1]. Полная высокоуровневая история  $H$  *линеаризуема* относительно объекта  $t$ , если существует последовательная высокоуровневая история  $S$  эквивалентная  $H$  такая, что:

- $S$  сохраняет отношение “*произошло до*”  $H$ ;
- $S$  соответствует последовательной спецификации объекта  $t$ .

Таким образом, высокоуровневая история  $H$  *линеаризуема*, если может быть дополнена до полной линеаризуемой высокоуровневой истории путем добавления соответствующих событий завершения подмножеству незавершенных операций и удалением оставшихся.

Также стоит описать, как обычно производится поиск/вставка/удаление элемента в списке с пропусками.

Для операции поиска на каждом уровне ищут самый правый узел в логическом представлении списка, значение в котором меньше или равно искомому. Затем переходя на уровень ниже. Если достигнут самый нижний уровень, то проверяют, хранит ли найденный узел искомое значение или нет, таким образом отвечая на запрос.

Для операции вставки сначала выполняют поиск узла, описанный ранее. Если найденный узел не хранит заданное значение, то создается новый узел и вставляется после найденного. В случае, если необходимо нарастить башню узлов, то новые узлы вставляют после тех узлов, на которых был совершен переход на уровень ниже. Как правило эти узлы не ищут заново, а используют информацию со стека программы в случае рекурсивной реализации или же из отдельного списка, хранящего такие узлы, в случае итеративной реализации.

Операция удаления аналогична операции вставки за исключением того, что ищут не сам узел, потенциально хранящий элемент, а предыдущий к нему,

чтобы в случае необходимости узел можно было удалить без перезапуска алгоритма поиска.

## 1.2. Сравнение деревьев поиска и списков с пропусками

Деревья поиска и списки с пропусками выполняют одну задачу — хранение упорядоченного множества и выполнения над ним операций. Сами по себе структуры очень схожи. Обе структуры данных строятся на основе вспомогательных структур, называемых узлами. Как правило, эти структуры хранят некоторый элемент упорядоченного множества, а также ссылки на другие узлы структуры и вспомогательную информацию, необходимую для реализации. О дуальности двоичных деревьев поиска и списков с пропусками говорится в работе [4].

Однако, у этих двух структур данных есть важные особенности, которые определяют их области применения. Узлы в дереве поиска как правило занимают меньше памяти на один хранимый элемент в сравнении со списками с пропусками, так как на одно значение из хранимого множества создается лишь один узел. Также, ввиду большего числа узлов, списки с пропусками хуже работают с кэш-памятью (так как требуется больше переходов от узла к узлу, что приводит к кэш-промахам), что приводит к уменьшению производительности в сравнении с деревьями поиска. Кроме того, большое количество реализаций деревьев поиска имеют гарантированное время работы основных операций, в отличие от списков с пропусками, которые, как правило, реализуются с использованием алгоритмов, использующих генератор случайных чисел, и дают гарантии лишь на среднее время работы операций. Хотя и существуют детерминированные списки с пропусками [5], они имеют проблемы с производительностью ввиду дополнительных накладных расходов на поддержание инвариантов

Также, у деревьев поиска существуют большое количество реализаций, которые умеют эффективно решать специализированные задачи. Например, `splay`-деревья могут эффективно решать задачу обращения к дереву поиска, в котором часто запрашиваемых элементов не очень много. По этим причинам деревья поиска являются основным выбором при реализации упорядоченных множеств в стандартных библиотеках многих языков программирования [3, 9].

Однако, утверждения выше справедливы, когда речь идет о реализациях, рассчитанных на использование лишь одним потоком одновременно. Если же рассматривать конкурентные реализации деревьев поиска, то они имеют существенный недостаток — им необходимо поддерживать свою балансировку, что в условиях конкурентного исполнения является нетривиальной и трудоемкой задачей, так как действия над одним узлом могут влиять на балансировку большого количества узлов. Также, из-за перебалансировки дерева затрудняется поиск узла, потенциально хранящего искомый элемент, так как узел может переместиться в другое поддереве и оказаться недостижимым из текущего. Все это сильно сказывается на производительности конкурентных деревьев поиска.

В то же время, списки с пропусками не имеют таких проблем: им не надо осуществлять балансировку и операции над узлами влияют на фиксированное число узлов в один момент времени. Поэтому они показывают себя лучше деревьев поиска в конкурентных программах. Подробное сравнение производительности было приведено в работе [6]. Поэтому, например, в языке программирования *Java* в качестве стандартной реализации конкурентного упорядоченного множества элементов используется именно конкурентный список с пропусками [8].

На текущий момент существует небольшое количество различных реализаций конкурентных списков с пропусками. Давайте рассмотрим некоторые из них, отметим ключевые особенности, а также выделим достоинства и недостатки.

### **1.2.1. Конкурентный список с пропусками стандартной библиотеки языка программирования *Java***

Даг Ли реализовал конкурентный список с пропусками для стандартной библиотеки языка программирования *Java* [8] в 2006 году, и эта реализация используется до сих пор.

В своей работе Даг Ли опирался на результаты, полученные в других исследованиях [7, 11]. Ключевой идеей в данных работах при реализации конкурентных списков было создание и вставка вспомогательного узла в список при удалении узла. Такой вспомогательный узел авторы называли маркером удаления (ориг. *marker*). Данный подход позволил решить проблемы с операциями вставки, конкурирующими с операциями удаления, а именно позволил избе-



жать ситуации, когда только что вставленный в список узел окажется недостижимым, ввиду того, что будет вставлен после уже удаленного узла.

Также в этих работах используется стандартный подход для неблокирующих алгоритмов, при котором потоки исполнения “помогают” другим операциям завершиться. Например, при поиске элемента во множестве поток исполнения может “помочь” удалить помеченный к удалению узел. Кроме того, Даг Ли в своей реализации уменьшил использование структурой памяти за счет переиспользования переменных внутри узлов.

Реализация Дага Ли получилась довольно эффективной для своего времени и сейчас является основным выбором при использовании упорядоченных множеств элементов в конкурентных программах на языке *Java*. Однако, тестирование производительности этой структуры показывает, что с ростом числа конкурирующих запросов скорость роста производительности всей структуры в целом замедляется [10].

### 1.2.2. A Contention-Friendly Non-Blocking Skip List

В работе [12] авторы рассматривают различные варианты реализации списка с пропусками. Отличительной особенностью одной реализации является выделение отдельного потока исполнения, называемым авторами управляющим потоком (ориг. *maintenance thread*). Этот поток всегда исполняется независимо от манипуляций со списком с пропусками. Он служит для управления структурой списка на верхних уровнях. Авторы работы заметили, что узким местом конкурентных списков с пропусками могут являться операции по удалению и добавлению узлов на верхних уровнях. Исходя из этого предположения ими была предложена идея, что узлы будут добавляться и помечаться к удалению на нижних уровнях потоками, которые эти узлы создали/удалили, а управляющий поток будет производить добавление и физическое удаление этих узлов на верхних уровнях.

Также авторами были протестированы различные эвристики. Например, в одном из протестированных вариантов они предлагают никогда не удалять узлы физически из списка. С одной стороны, это приводит к существенному росту производительности операции удаления. Но с другой стороны, в случае сильно меняющегося набора используемых в программе значений хранимого упорядоченного множества приводит к замедлению других операций ввиду разрастания самого списка, а, следовательно, к увеличению количества узлов,

необходимых к посещению алгоритмом. Также, авторы рассмотрели различные варианты конфигурации управляющего потока.

В результате авторы сравнили различные реализации своей структуры с решением Дага Ли и получили существенный прирост в производительности операции поиска, а также небольшой прирост в производительности модифицирующих операций.

Однако, стоит отметить недостатки данного подхода: с ростом числа потоков системе узким местом структуры становится именно управляющий поток, так как только он один будет обслуживать все модификации структуры списка с пропусками на высоких уровнях. Также, эффективность структуры зависит от поведения одного потока исполнения, что является существенным недостатком, если необходима конкурентная структура данных с ожидаемым поведением, не сильно зависящим от поведения планировщика процессора. Также, небольшим минусом можно назвать постоянно потребляющий процессорное время управляющий поток, который активен даже в моменты, когда к структуре никто не обращается.

### **Выводы по главе 1**

В главе были приведены определения и формулировки используемые в работе, было произведено сравнение деревьев поиска и списков с пропусками в условиях конкурентных и однопоточных программ. Также были рассмотрены две известные реализации конкурентных списков с пропусками, выделены их отличительные черты, достоинства и недостатки.

## ГЛАВА 2. РАЗРАБОТКА СТРУКТУРЫ

В этой главе будет приведено описание разработанных структур данных, а также алгоритмов основных операций работы над множеством упорядоченных элементов, а также вспомогательных операций, упрощающих реализацию.

Передо мной была поставлена задача разработать конкурентный список с пропусками, у которого одна башня узлов будет отвечать не за один элемент хранимого множества, а за несколько. Для этого предлагается хранить такие множества элементов в массивах внутри узлов. Такое решение должно уменьшить количество выделений и очисток оперативной памяти, связанных с созданием и удалением узлов и ссылок описывающих структуру списка с пропусками, а также может уменьшить количество кэш-промахов за счет меньшего числа переход между узлами и локализованности в памяти хранимых значений, что должно положительно сказаться на производительности структуры. Однако, такое решение создает сложности при модификации массивов. Например, несколько активных потоков могут пытаться производить конкурентную работу над одним или разными участками массива, но при это все равно необходимо поддерживать согласованное состояние узла и структуры в целом. Поэтому было принято решение реализовать операции, модифицирующие массивы, с использованием блокировок, но при этом реализовать поиск элемента множества в структуре неблокирующим. Таким образом, необходимо реализовать в нашей структуре данных три основных операции работы со множествами:

- неблокирующий поиск элемента;
- добавление элемента ко множеству;
- удаление элемента из множества.

### 2.1. Базовые структуры

Прежде чем переходить к описанию алгоритмов основных операций, сначала необходимо описать с какими структурами работают эти операции, а также инварианты, которые будут в них поддерживаться.

**NodeArray.** Массив элементов `key` с указателем на первую пустую позицию в массиве `end` вынесены в отдельную структуру, представленную на листинге 1, для возможности согласованного чтения обоих полей. Также предлагается хранить все элементы на префиксе массива, то есть все пустые пози-

ции будут находиться на суффиксе массива. Далее будет объяснено, почему был выбран именно такой инвариант.

Листинг 1 – Описание массива, хранящегося в узле

```
class NodeArray:
    volatile int end;
    AtomicReferenceArray<E> key;
```

**Node.** Так как целью было поставлено эффективное использование кэш-памяти, то предлагается хранить всю башню узлов в одном узле, это позволит, во-первых, сэкономить память на ссылках внутри башни узлов и вспомогательных данных, во-вторых, позволит уменьшить число кэш-промахов, так как не будет переходов внутри башни узлов.

Собственно, структура узла/башни узлов, представленная на листинге 2, хранит в себе массив `NodeArray array`, содержащий элементы множества; первое записанное в этот узел значение `initialValue`; динамический массив ссылок на следующие узлы на соответствующих уровнях `next`; объект для обеспечения алгоритма критической секции `lock`; динамический массив ссылок на узлы `deletedBy`, которые являлись предыдущими в логическом представлении списка на уровне соответствующем индексу в массиве во время удаления данного узла с конкретного уровня списка с пропусками, а также логическая переменная `deleted`, показывающая, удалён ли логически этот узел.

Листинг 2 – Описание узла

```
class Node:
    NodeArray array;
    E initialValue;
    Vector<Node> next;
    Lock lock;
    Vector<Node> deletedBy;
    volatile boolean deleted;
```

**KArraySkipList.** Сам же список с пропусками изначально представляет из себя два узла — голову `H` и хвост `T`, а также текущую высоту списка `curHeight` и длину массивов `k`, используемых в узлах.

Листинг 3 – Описание структуры списка с пропусками

```
class KArraySkipList:  
    int k;  
    Node H;  
    Node T;  
    AtomicInteger curHeight;
```

## 2.2. Описание операций над структурой

Первой частью любой из трех основных операций над классическим списком пропусков является поиск узла, над которым эту операцию необходимо совершить. Например, чтобы удалить элемент из структуры, необходимо найти узел, в котором этот элемент мог бы храниться, и если найденный узел действительно хранит указанный элемент, то необходимо удалить данный узел, а также все узлы из башни ему соответствующей. Поэтому для начала необходимо научиться находить нужный узел, в массиве которого потенциально мог бы находиться элемент, над которым выполняется операция.

Для начала можно заметить, что тривиальным решением будет поддерживать инвариант списка с пропусками такой, что наибольший элемент любого массива узла строго меньше всех элементов, хранящегося в массивах узлов, расположенных правее в логическом представлении списка с пропусками. Таким образом для поиска узла, потенциально хранящего заданный элемент, достаточно модифицировать алгоритм перехода к следующему узлу в классическом алгоритме поиска узла: найти в массиве следующего на текущем уровне узла наименьший элемент и сравнить с заданным и если наименьший элемент меньше или равен заданному, то перейти к следующему узлу на уровне, иначе перейти на уровень ниже. В случае, если переход совершить невозможно, то текущий узел и является искомым. Этот алгоритм представлен на листинге 4.

Заметим, что для нахождения минимума в массиве будет необходимо проверить все значащие элементы, что потребует в худшем случае  $k$  итераций, где  $k$  — длина массивов. Можно было бы поддерживать в каждом узле актуальный минимум массива для того, чтобы избавиться от его поиска, однако при удалении минимального элемента из массива все равно придется совершить в худшем случае  $k$  итераций для поиска нового минимума.

Поэтому предлагается при создании каждого узла отдельно записывать в него, с каким элементов этот узел был создан (узлы создаются хотя бы с од-

Листинг 4 – Алгоритм поиска узла, потенциально хранящего заданное значение с использованием минимума массивов

```

function find( $v$ ) : Node
   $cur \leftarrow H$ 
   $curLevel \leftarrow curHeight.get()$ 
  while  $curLevel \geq 0$  do
     $next \leftarrow cur.next.get(curLevel)$ 
    while  $next \neq T \ \&\& \ next.array.min() \leq v$  do
       $cur \leftarrow next$ 
       $next \leftarrow cur.next.get(curLevel)$ 
    end while
     $curLevel \leftarrow curLevel - 1$ 
  end while
  return  $cur$ 
end function

```

ним элементом в массиве. Голове и хвосту присваиваются  $-\infty$  и  $+\infty$  соответственно). Затем предлагается поддерживать следующий инвариант на логическом представлении списка: все узлы, которые находятся правее выбранного (включая сам выбранный узел), хранят элементы большие или равные первому записанному в выбранный узел элементу, а все узлы, которые находятся левее выбранного узла в логическом представлении списка, хранят элементы меньшие первого записанного в выбранный узел элемента. Таким образом, первые записанные в узлы элементы будут задавать “отрезки” в логическом представлении списка, на которых может находиться какой-либо элемент. Пример инварианта приведен на рисунке 4. Теперь в алгоритме поиска узла можно заменить  $next.array.min()$  на  $next.initialValue$ . Результирующий алгоритм приведен на листинге 5.

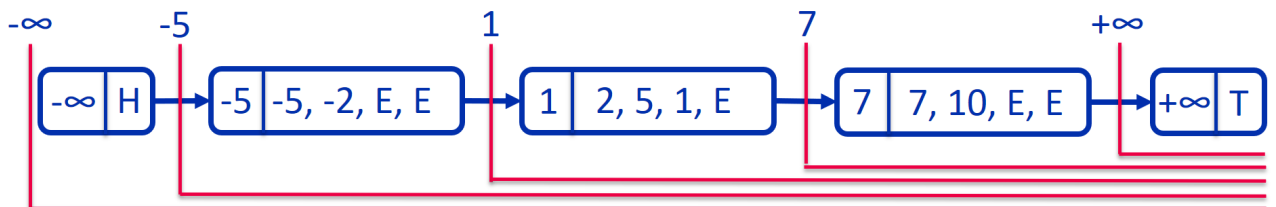


Рисунок 4 – Пример отрезков, задаваемых первыми записанными в узлы значениями

Листинг 5 – Алгоритм поиска узла

```

function find(v) : Node
  cur ← H
  curLevel ← curHeight.get()
  while curLevel ≥ 0 do
    next ← cur.next.get(curLevel)
    while next ≠ T && next.initialValue ≤ v do
      cur ← next
      next ← cur.next.get(curLevel)
    end while
    curLevel ← curLevel − 1
  end while
  return cur
end function

```

### 2.2.1. Вспомогательные операции

Рассмотрим две вспомогательные функции, которые будут использованы далее.

Первая функция `firstNotPhysicallyDeleted` поможет избежать ситуации, когда после поиска узла и взятия на него блокировки он мог оказаться физически удаленным. С одной стороны, можно было бы запустить алгоритм поиска сначала, но это не дает гарантий, что мы опять не окажемся в такой же ситуации. Собственно, функция находит физически не удаленный узел, расположенный левее заданного на заданном уровне. Для этого используются ссылки `deletedBy`. Эта функция представлена на листинге 6. Она принимает узел, считая, что поток вызвавший ее держит на нем блокировку. Затем проверяет не был ли узел физически удален с текущего уровня и, если был, переходит к узлу, который являлся предыдущим во время удаления, используя соответствующую ссылки из вектора `deletedBy`.

Вторая функция `moveForwardBlocking` поможет избежать ситуации, когда после поиска и взятия блокировки на узел он мог оказаться уже не актуальным для этой операции, а актуальный находится дальше по списку рассматриваемого уровня. Функция принимает узел, уровень списка и значение. Функция предполагает, что поток, вызвавший ее, держит блокировку на переданный узел. Под блокировками функция находит узел такой, что начальное значение `initialMin` следующего за ним узла строго больше переданного в функцию значения. Функция приведена на листинге 7.

Листинг 6 – Функция поиска физически не удаленного узла на заданном уровне

```

function firstNotPhysicallyDeleted(inputNode, level) : Node
  cur ← inputNode
  height ← cur.next.size()
  notDeletedLevels ← height – cur.deletedBy.size()
  while cur.deleted && notDeletedLevels – 1 < level do
    prev ← cur.deletedBy.get(height – level – 1)
    cur.lock.unlock()
    cur ← prev
    cur.lock.lock()
    height ← cur.next.size()
    notDeletedLevels ← height – cur.deletedBy.size()
  end while
  return cur
end function

```

Листинг 7 – Функция поиска узла потенциально хранящего переданное значение на заданном уровне

```

function moveForwardBlocking(inputNode, level, v) : Node
  cur ← firstNotPhysicallyDeleted(inputNode, level)
  next ← cur.next.get(level)
  while next ≠ tail && v ≥ next.initialMin do
    next.lock.lock()
    cur.lock.unlock()
    cur ← next
    next ← next.next.get(level)
  end while
  return cur
end function

```

### 2.3. Разбор алгоритмов операций

В этом разделе будут подробно рассмотрены и объяснены алгоритмы основных операций.

#### 2.3.1. Добавление и удаление элемента

Первыми будут рассмотрены добавление и удаление элементов из списка с пропусками, так как они довольно похожи.

План выполнения любой операции над списком с пропусками можно описать двумя шагами: поиск узла, над которым необходимо выполнить операцию, и, собственно, выполнения операции над найденным узлом, если по-



требуется. Довольно часто при реализации списка с пропусками при поиске узла перед выполнением над ним операции сохраняют все узлы, которые были посещены во время поиска. Это позволяет, в случае необходимости добавить узел, не выполняя поиск позиций на уровнях выше, на которые необходимо вставить узел. При выполнении поиска узла во время операции удаления применим аналогичный подход, за исключением того, что ищем не сам узел, а узел, который бы находился перед искомым.

Так как в рассматриваемой структуре в узлах используются массивы, то большое количество операций вставки (а также удаления) будут заканчиваться без затрагивания других узлов. То есть, довольно часто не возникнет необходимости изменять представление списка с пропусками путем добавления или удаления узлов. Также, операции могут и вовсе завершаться ничего не изменив в структуре, так как, например, при добавлении элемента может оказаться, что элемент уже присутствует в структуре. Кроме того, даже если придется создать новый узел, то в половине случаев (вероятность увеличения высоты башни на один равняется 0.5) достаточно знать позицию вставки только на самом нижнем уровне. Эту информацию можно использовать, чтобы сэкономить время и память не сохраняя этот путь. В случае, если же информация о пути все таки понадобится для добавления/удаления узла, предлагается запустить соответствующий алгоритм поиска, но уже с сохранением пройденного пути.

После нахождения узла, над которым необходимо выполнить операцию, можно переходить к выполнению операции. Однако, заметим, что операция поиска узла является неблокирующей, и после получения ее результата и до начала выполнения операции у нас нет гарантий, что нам все еще необходимо выполнить операцию именно над найденным узлом. Например, другой поток мог удалить найденный узел из списка до того, как мы возьмем на него блокировку, или же элемент, над которым выполняется операция окажется в узле следующим за найденным после разбиения найденного на два. Поэтому, предлагается взять блокировку на найденный узел и выполнить поиск узла, но уже с использованием блокировок, чтобы исключить такие ситуации. Для этого запускается функция поиска узла на заданном уровне `moveForwardBlocking`, представленная на листинге 7. Рассмотрим подробнее почему это необходимо.

Сначала, эта функция вызывает вспомогательную функцию `firstNotPhysicallyDeleted`, проверяя был ли данный узел физически удален из списка на заданном уровне и если был, переходит к узлу, который был предыдущим в момент удаления. Этот алгоритм повторяется до тех пор, пока не будет найден узел, который еще не был физически удален на заданном уровне и возвращает его. После этого, выполняется поиск элемента на заданном уровне, над которым необходимо выполнить операцию, и найденный узел возвращается. Так как поддерживаются блокировки при переходе к следующему узлу, после выхода из этой функции узел будет являться тем, над которым необходимо выполнить операцию. Заметим, что другие потоки не могли физически удалить найденный узел, так как им бы потребовалось взять блокировку на предыдущий, что сделал текущий поток. Добавить новый узел после найденного другие потоки не смогли бы, так как им бы пришлось взять блокировку на найденный. Теперь можно выполнить операцию над узлом.

Функция добавления элемента ищет на префиксе массива переданный элемент и, если его нет, то вставляет элемент на первую пустую позицию и увеличивает индекс первой пустой позиции на единицу. В случае, если пустые позиции были, то после добавления элемента операцию можно завершить. Иначе, если в узле не оказалось пустых позиций, то предлагается создать новый узел, а также новый массив для текущего узла. Затем перераспределить элементы между массивами нового узла и новым массивом для текущего узла таким образом, чтобы все элементы большие добавляемого оказались в новом узле, а оставшиеся в старом. Функция создающая новый узел и выполняющая перераспределение элементов между массивами узлов представлена на листинге А.4. Затем можно добавить новый узел после текущего в список. И после добавления нового узла в список заменить массив текущего узла новым массивом, полученным после перераспределения элементов. Стоит сказать, что в рассматриваемой структуре для удобства реализации предлагается запретить добавлять элементы в узел `N`. Можно считать, что в этом узле всегда нет пустых позиций и поэтому при попытке вставить в него элемент необходимо создать новый узел. Этот случай в алгоритме рассматривается отдельно, но он не имеет сложностей с перераспределением узлов.

В итоге, новый элемент уже может быть найден в списке. В случае, если необходимо добавить узел на уровне выше, то вызывается функция поиска узла, но уже с сохранением пройденного пути. Затем на каждом уровне, используя аналогичные рассуждения про неблокирующий поиск и потенциально неактуальный узел после взятия блокировки, ищется ближайший слева узел для нового узла и новый узел добавляется после него в список на заданном уровне. Если необходимо увеличить высоту списка с пропусками, то новый узел вставляется после узла  $N$ . Функция вставки нового узла на уровне выше представлена на листинге А.3. Полный алгоритм добавления элемента представлен на листинге А.2

Рассмотрим теперь подробнее операцию удаления. Начинается она аналогично операции добавления, сначала ищется узел потенциально хранящий переданное значение, берется блокировка на найденный узел, и запускается алгоритм поиска узла под блокировками на заданном уровне, затем вызывается операция удаления элемента из узла, представленная на листинге А.1. В ней проверяются все элементы на значащем префиксе, и если элемент присутствует в массиве, то он удаляется из массива. Для этого берется последний значащий элемент на префиксе и копируется в позицию найденного элемента, таким образом затирая найденный элемент. Затем последний элемент на префиксе заменяется пустым элементом и сдвигается индекс конца значащего префикса. В случае если элемент был удален и массив оказался пустым, то узел также помечается к удалению.

Если узел не был помечен удалённым, то возвращается результат операции, иначе, начинается процесс удаления узла со всех уровней в порядке сверху вниз. Для этого запускается функция поиска всех узлов, которые являются предыдущими к удаляемому на своих уровнях. На каждом уровне выполняется изменение ссылки у предыдущего узла на узел следующий за удаляемым, предварительно сделав поиск нужного узла с использованием описанной ранее функции `moveForwardBlocking`. Это необходимо по тем же причинам, что и при вставке на нижний уровень: после поиска узла и до взятия блокировки на него структура списка могла поменяться. Также, в вектор `deletedBy` удаляемого узла добавляется ссылка на узел, которым он был удален.

### 2.3.2. Поиск элемента

Перейдем к последней операции. Для начала стоит отметить ключевые особенности в алгоритмах предыдущих операций.

Операция добавления элемента в узел не перемещает элементы в массиве, а лишь добавляет новые. В случае создания нового узла специально создается новый массив для старого узла, чтобы это свойство сохранялось. Операция удаления элемента из узла перемещает элементы лишь в сторону начала массива.

Суммируя эти два важных свойства можно использовать следующий алгоритм проверки наличия элемента в узле: начать поиск элемента с конца префикса массива и двигаться в сторону начала массива, таким образом, если элемент присутствовал в массиве на момент запуска поиска и не будет удален конкурирующим потоком, то он будет гарантировано найден.

Используя этот алгоритм, можно использовать следующий алгоритм проверки наличия элемента во всем списке с пропусками. Запустить алгоритм поиска узла, потенциально хранящего искомое значение. С помощью описанного выше алгоритма проверить, есть ли в найденном узле искомый элемент. Если есть, то вернуть ответ, в противном случае необходимо убедиться, что следующий узел не может хранить искомый элемент (такое могло произойти в случае, когда перед чтением массива другой поток успел бы создать новый узел и заменить массив другим, то есть искомый элемент мог оказаться в следующем узле). Если следующий узел не может хранить искомый элемент, то можно вернуть ответ, иначе необходимо перейти к нему. Алгоритм поиска элемента в списке с пропусками представлен на листинге 8.

Листинг 8 – Алгоритм проверки наличия элемента во множестве

```

function contains(v) : boolean
  cur ← find(v)
  repeat
    curNodeContainsV ← false
    if !cur.deleted then
      singleReadArray ← cur.array
      key ← singleReadArray.key
      for it ← singleReadArray.end – 1, 0 do
        curValue ← key[it]
        if curValue ≠ EMPTY && v == curValue then
          curNodeContainsV ← true
          break
        end if
      end for
    end if
    if curNodeContainsV then
      return true
    end if
    cur ← cur.next.get(0)
  until cur ≠ tail && v ≥ cur.initialMin
  return false
end function

```

### Выводы по главе 2

Была разработана и объяснена структура конкурентного списка, хранящего в узлах массив элементов. Были рассмотрены алгоритмы всех основных операций над структурой, приведен их псевдокод. В следующей главе планируется привести доказательство корректности алгоритма с точки зрения линейности, а также произвести сравнение производительности полученного алгоритма с существующими реализациями конкурентных списков

## **ГЛАВА 3. КОРРЕКТНОСТЬ. ПРОИЗВОДИТЕЛЬНОСТЬ. РАССМОТРЕННЫЕ ИНВАРИАНТЫ УЗЛА.**

В этой главе будет приведено доказательство корректности операций, а также приведено сравнение производительности с существующими аналогами. В конце главы будет приведено описание рассмотренных, но неиспользованных инвариантов массива узла.

### **3.1. Корректность операций**

Первыми будут рассмотрены вспомогательные операции, а затем основные. Для основных операций будут указаны точки линеаризации.

#### **3.1.1. Вспомогательные операции**

Рассмотрим операцию поиска физически не удаленного узла `firstNotPhysicallyDeleted`. Заметим, что переходы выполняются по ссылкам, ведущим в сторону начала списка, и в случае выполнения перехода сначала отпускается блокировка на текущем узле и только затем осуществляется переход. Так как переходы осуществляются влево в логическом представлении списка, то если функция закончит свою работу, значит будет возвращен узел, начальное значение которого все еще меньше или равно начального значения переданного в функцию узла. А так как возвращаемый узел еще не был физически удален, то он достижим по определению.

Теперь рассмотрим операцию поиска узла на уровне с использованием блокировок `moveForwardBlocking`. Заметим, что перед переходом к следующему узлу на него берется блокировка. И только затем отпускается блокировка текущего. Это гарантирует, что другие потоки не смогут удалить физически следующий узел с текущего уровня. Результатом работы алгоритма является узел, после которого идет узел, минимальное значение которого строго больше переданного в функцию. Новых узлов не может быть добавлено и сам узел не может быть удален, так как на возвращаемом узле держится блокировка.

#### **3.1.2. Добавление и удаление элемента**

Рассмотрим операцию добавления элемента ко множеству. Несложно заметить, что чтобы добавить элемент ко множеству, необходимо держать блокировку на узел, в который этот элемент необходимо вставить. Из этого следу-

ет, что внутреннее состояние узла будет изменяться не более чем одним потоком в любой момент времени. Также, чтобы добавить новый узел на выбранный уровень списка с пропусками необходимо держать блокировку на узел, после которого новый узел будет добавлен. А так как, этот узел является достижимым из начала списка, ведь он был получен алгоритмом ищущим именно такой узел (было доказано ранее), то и новый узел будет достижим из начала списка после вставки.

Когда необходимо увеличить высоту башни, для каждого уровня будет запущен алгоритм аналогичный добавлению узла на начальном уровне за исключением того, что если за время перехода от уровня к уровню узел был помечен к удалению другим потоком, то алгоритм добавления будет остановлен.

Для удаления элемента из узла рассуждения аналогичны рассуждениям для операции добавления элемента в узел. На каждом уровне с использованием блокировок ищется узел предыдущий к удаляемому — это гарантирует, что найденный узел останется предыдущим, ведь чтобы добавить узел между найденным и удаляемым необходимо держать блокировку на найденный. Также, никакой другой поток не сможет добавить узел после удаляемого, ведь ему необходимо будет держать блокировку на удаляемый. После изменения ссылок и обновления вектора *deletedBy* удаляемый узел становится недостижимым на этом уровне.

Заметим, что операцию по добавлению и удалению узла из списка с пропусками выполняет строго один поток, и другие потоки никак не “помогают” ему. Поэтому никаких гонок в этом месте возникнуть не может.

Несложно заметить, что узел, в который необходимо вставить (удалить) элемент определяется однозначно. Таким образом, точкой линеаризации для операции вставки можно считать момент, когда увеличивается индекс первой пустой позиции в `array` в операции добавления элемента к узлу А.6 в случае, если новый узел не создавался. В случае, если новый узел был создан, то точкой линеаризации можно считать момент, когда в узле, после которого новый узел будет вставлен, в динамическом массиве `next` будет изменена ссылка по нулевому индексу на создаваемый узел. Для операции удаления точкой линеаризации можно считать момент, когда удаляемый элемент в операции удаления из узла с листинга А.1 будет заменен последним значащим элементом в массиве в случае, если элемент не был последним на значащем префиксе, иначе

точкой линеаризации будет замена последнего значащего элемента пустым в функции с того же листинга.

В заключение разбора модифицирующих операций стоит добавить, что все блокировки берутся строго в порядке нахождения на уровнях, ввиду чего не может возникнуть цикла при взятии блокировок, а следовательно и взаимной блокировки.

### 3.1.3. Поиск элемента

Докажем, что операция поиска линеаризуема. Для этого рассмотрим точки линеаризации операций вставки и удаления элемента  $v$  между событиями начала операции поиска элемента  $v$  и ее завершением. Возможно два случая. Первый случай — таких точек не оказалось. Опять же, два случая. Если элемента не было в структуре на момент запуска операции поиска, то он и не может быть найден ввиду алгоритма поиска. Если же элемент находился в списке с пропусками на момент начала операции поиска, то он был достижимым, а так как операция поиска не может пропустить узел хранящий элемент и также не может пропустить его при поиске внутри массива ввиду алгоритма поиска в массиве, то элемент будет найден.

Рассмотрим второй случай, когда есть точки линеаризации модифицирующих операций над элементом  $v$  между началом и концом операции поиска операции элемента  $v$ . Рассмотрим результат операции поиска: если элемент был найден, то найдем точку линеаризации вставки элемента  $v$  и отметим точку линеаризации операции поиска сразу же после нее, если же операция поиска не нашла элемент, то отметим точку линеаризации сразу же после точки линеаризации операции удаления. В случае, если таких точек не оказалось, то, ввиду того что операции добавления и удаления линеаризуемы, а значит их результаты согласованы между собой, можно отметить точку линеаризации операции поиска сразу же после события начала операции поиска.

Таким образом, точкой линеаризации операции поиска (листинг 8) в случае отсутствия элемента можно считать момент, когда алгоритм выходит из цикла `repeat-until`, или же момент, когда алгоритм поиска устанавливает логическую переменную `curNodeContainsV` в истину в случае истинности результата поиска.

В итоге, операция поиска линеаризуема.



### 3.2. Сравнение

Тестирование проводилось на сервере лаборатории компьютерных технологий Университета ИТМО с 64-ядерным процессором AMD Opteron 6378 @2.4 GHz и установленными 256 Gb оперативной памяти. Однако, ввиду ограничений установленных на систему планировщика задач Slurm, было доступно только 32 потока исполнения процессора. На сервере была установлена ОС Ubuntu 18.04.2 LTS, программные реализации были выполнены на языке программирования Java и запускались с использованием 64-битного OpenJDK RE версии 11.0.3+7-Ubuntu-1ubuntu218.04.1.

Для тестирования пропускной способности было выбрано три гиперпараметра:

- размер множества, подмножество которого будет храниться в структуре (параметр сверху каждого рисунка с графиками указывает этот гиперпараметр для всех графиков рисунка);
- вероятность модифицирующих операций  $x$ : то есть примерно  $100 - x\%$  операций будут операциями поиска элемента, и, соответственно, примерно  $x/2\%$  будут операциями вставки и примерно  $x/2\%$  будут операциями удаления элемента (вероятность модифицирующих операций указана сверху каждого графика);
- количество потоков одновременно работающих со структурой (ось абсцисс на графиках).

Перед тестированием каждая структура заполнялась подмножеством мощности равного половине тестируемого (параметр сверху каждого рисунка с графиками указывает размер всего множества, а не подмножества). Каждая точка на графике соответствует пропускной способности структуры при заданных гиперпараметрах, описанных ранее.

После заполнения структуры, создавалось соответствующее число потоков исполнения, которые выполняли операции поиска/вставки/удаления уже над всем тестируемым множеством с заданной вероятностью модифицирующих операций в течение пяти минут и считали число завершенных операций. После завершения всех потоков количество завершенных операций суммировалось по всем потокам и делилось на число секунд, затраченных на их исполнение (300 секунд в данном случае), таким образом получая пропускную способность структуры в операциях на секунду. Также стоит сказать, что все точ-

ки на одном графике соответствующие одному набору гиперпараметров были получены путем запуска тестирующего алгоритма из идентичных начальных условий. То есть, если на графике нескольким точкам соответствует один набор гиперпараметров, то отличие в начальных условиях было лишь в используемой структуре, все остальные параметры совпадали: зерна случайности для тестирующих потоков; число, тестирующих потоков; начальное подмножество, наполняющее структуру.

Перед сравнением производительности полученного решения с существующими аналогами было проведено сравнение производительности реализованной структуры в зависимости от числа  $k$  — размерности массивов. Подбор параметра  $k$  производился на множестве размера 500 000. Результаты можно увидеть на рисунке 5. Реализованная структура называется *KSkipListConcurrent*, число в скобках соответствует размерности массива внутри узлов. Из рисунка видно, что хорошо себя показывают значения  $k$  равные 32 и 24. Для сравнения с другими решениями было выбрано  $k$  равное 32.

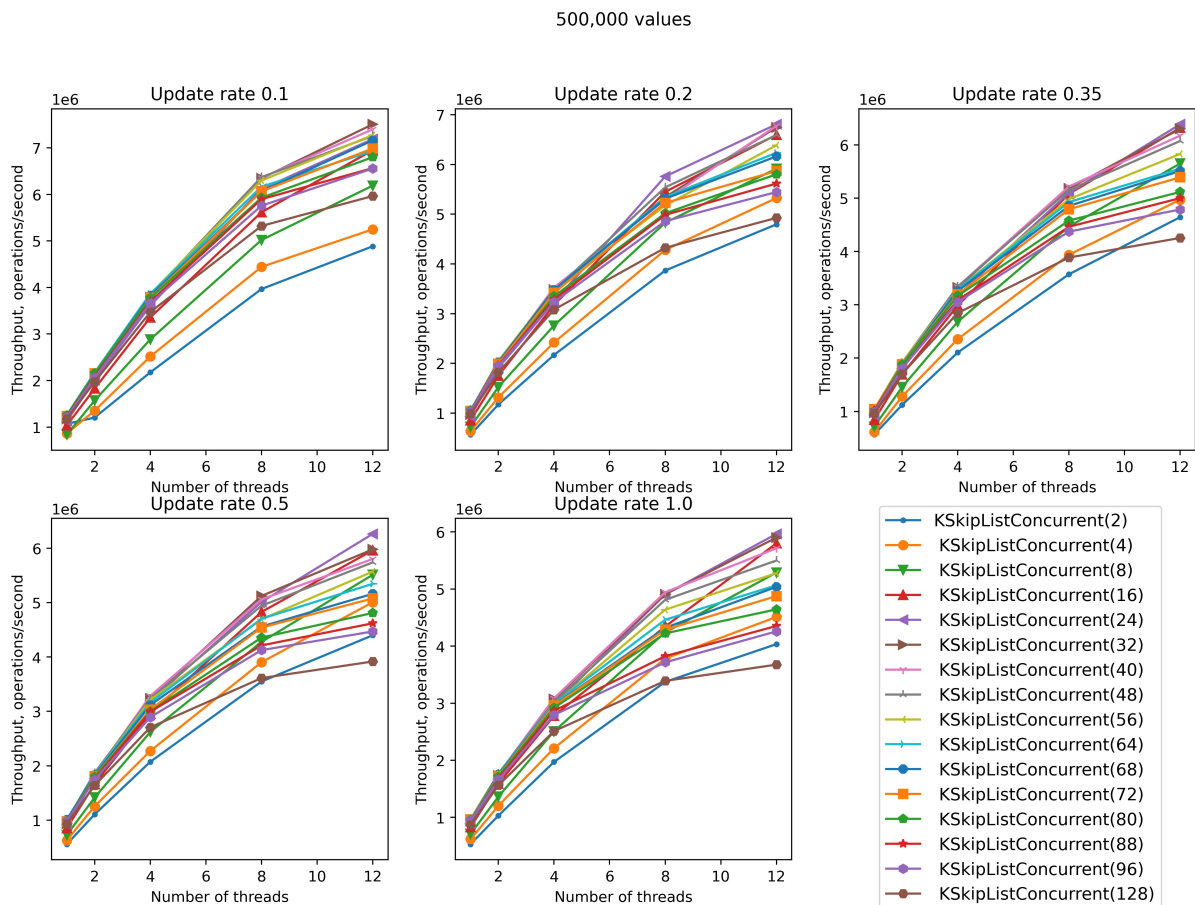


Рисунок 5 – Сравнение полученного решения при различных  $k$

Затем было произведено сравнение с существующими решениями. Стоит отметить, что реализация было выполнена на языке программирования *Java* в котором все объекты являются ссылочными типами, что приводит к дополнительным переходам по памяти при обращении к элементам хранимого множества, что не позволяет получить весомого выигрыша от попадания элементов массива в кэш процессора. Результаты сравнения для множества размером 1 500 000 и вероятностей изменения 0.1, 0.5 и 1.0 приведены на рисунке 6. Результаты для других размеров множеств приведены на рисунках Б.1, Б.2 и Б.3 в приложении. Сравнение проводилось со структурами описанными в ранее. Из результатов тестирования видно, что новая версия конкурентного списка показывает себя на уровне существующих решений даже при использовании ССЫЛОЧНЫХ ТИПОВ.

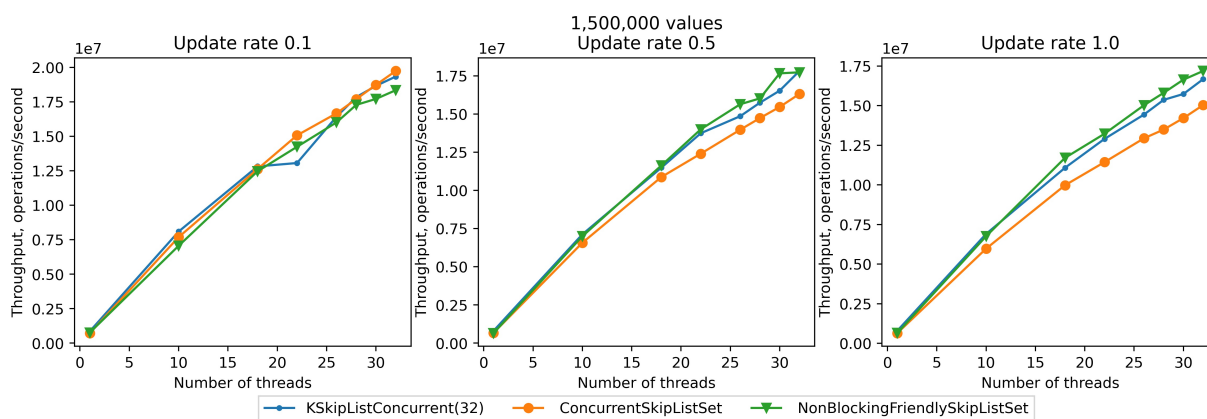


Рисунок 6 – Сравнение полученного решения с другими

Далее было принято решение протестировать нашу структуру на нессылочных типах. Для этого был выбран примитивный целочисленный тип *int*. Также для полноценного сравнения существующие решения тоже были переписаны для использования только типа *int*. Результаты сравнения для множества размером 1 500 000 и вероятностей изменения 0.2, 0.5 и 1.0 приведены на рисунке 7. Результаты для других размеров множеств приведены на рисунках Б.4, Б.5 и Б.6 в приложении. На графиках видно, что в таком варианте структура дает существенный прирост в производительности (рисунок 7). Нетрудно видеть, что с использованием примитивных типов производительность нового решения существенно превосходит аналоги.

В заключение, было проведено сравнение занимаемой полученной структурой память относительно *ConcurrentSkipListSet*. Сравнение производилось заполнением обеих структур одинаковым набором данных и снятием дам-

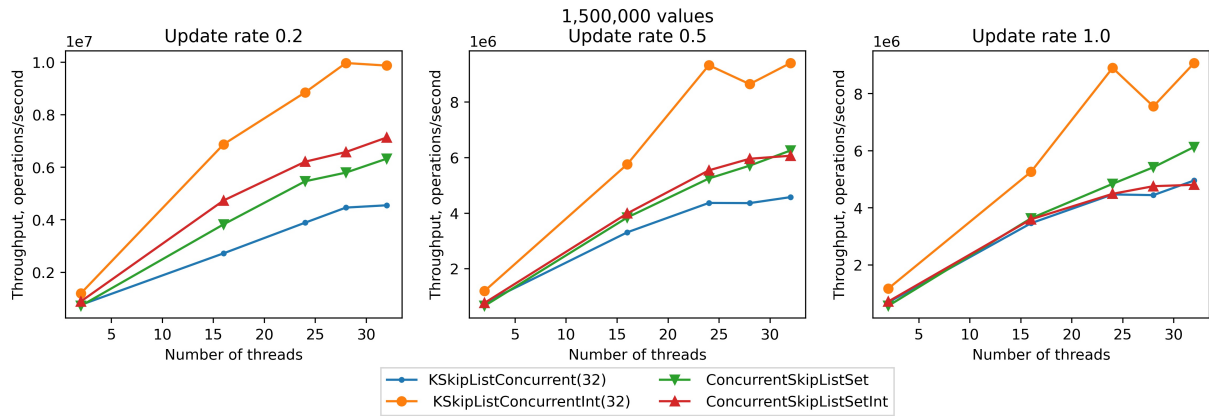


Рисунок 7 – Сравнение полученного решения с другими с типом *int*

пов памяти с использованием программы VisualVM. Сравнение показало, что новая версия с использованием массивов в узлах потребляет более чем в два раза меньше памяти по сравнению с *ConcurrentSkipListSet* (40 Mb против 87 Mb при наполнении  $1.5 \cdot 10^6$  элементами).

### 3.3. Рассмотренные решения

Здесь будут описаны рассмотренные варианты при разработке конкурентного списка, которые были отброшены в пользу использованных в работе.

Первым рассмотренным вариантом инварианта массива узла было отсутствие инварианта, то есть не было никаких гарантий о позиции элементов. Таким образом любая операция требовала бы  $k$  итераций по массиву, ведь элемент, над которым операция мог оказаться на любой из позиций.

Следующим рассмотренным инвариантом массива узла было использование циклического непрерывного подотрезка в массиве. Среднее время выполнения операций было бы аналогично предложенному в работе, однако для реализации циклического подотрезка необходимо выполнять дополнительные арифметические операции, а также сохранять оба конца этого подотрезка для корректной работы операции поиска элемента. Что явно хуже предложенного варианта.

Следующими рассмотренным вариантом было использование строго упорядоченного префикса, то есть все элементы на префиксе были бы упорядочены. Это позволило бы с использованием бинарного поиска по префиксу быстро находить искомый элемент, однако при операциях модификации необходимо было бы осуществлять пересортировку префикса, что приводило бы к перемещению элементов массива в произвольные стороны. Ввиду этого реали-

зовать эффективно работающий неблокирующий поиск было бы невозможно. Если же сделать отсортированный циклический подорезок, то это бы позволило выполнять эффективный поиск во время модифицирующих операциях, но все еще не позволило бы использовать бинарный поиск при неблокирующем поиске.

### **Выводы по главе 3**

В этой главе было приведено доказательство корректности основных операций работы над множествами с точки зрения линейризуемости, а также приведено сравнение производительности с существующими аналогами. В заключение, было приведено описание рассмотренных, но неиспользованных инвариантов массива узла, а также причины, по которым они не были использованы.

## ЗАКЛЮЧЕНИЕ

В работе было представлено сравнение подходов в использовании деревьев поиска и списков с пропусками для работы с упорядоченными множествами элементов. Были проанализированы достоинства и недостатки обоих решений как в однопоточных, так и конкурентных программах.

Были разобраны две существующие реализации конкурентных списков с пропусками [8, 12], приведены их основные отличительные черты, достоинства и недостатки.

Была представлена новая идея для реализации конкурентных списков. На основе идеи была разработана и реализована новая структура данных. Было приведено доказательство корректности структуры с точки зрения линейности. И, наконец, было проведено сравнение производительности с существующими решениями.

Исходя из результатов сравнения можно сделать вывод, что полученная реализация при использовании ссылочных типов ведет себя на уровне существующих решений при относительно небольшом множестве хранимых в структуре значений, но также показывает небольшой прирост производительности при большом наборе хранимых в структуре значений. Однако, при использовании нессылочных типов наша структура показывает существенный прирост производительности по сравнению с другими решениями. Также, полученное решение позволило существенно сократить используемую списком с пропусками память по сравнению с другими решениями, благодаря сокращению количества узлов в самой структуре.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1 *Herlihy M. P., Wing J. M.* Axioms for Concurrent Objects // — Munich, West Germany : Association for Computing Machinery, 1987. — С. 13–26. — (POPL '87). — ISBN 0897912152. — DOI: 10.1145/41625.41627. — URL: <https://doi.org/10.1145/41625.41627>.
- 2 *Popov G., Mastorakis N., Mladenov V.* Calculation of the acceleration of parallel programs as a function of the number of threads // Kliment Ohridski”blvd. — 2010. — ЯНВ. — Т. 8.
- 3 C++ std::set [Электронный ресурс]. — 2021. — URL: <https://en.cppreference.com/w/cpp/container/set>.
- 4 *Dean B. C., Jones Z. H.* Exploring the Duality Between Skip Lists and Binary Search Trees. — 2007. — URL: [https://www.researchgate.net/publication/220995761\\_Exploring\\_the\\_duality\\_between\\_skip\\_lists\\_and\\_binary\\_search\\_trees](https://www.researchgate.net/publication/220995761_Exploring_the_duality_between_skip_lists_and_binary_search_trees).
- 5 Deterministic skip lists // Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms. SODA 1992. — Association for Computing Machinery, 09/1992. — P. 367–375. — (Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms).
- 6 *Fraser K., Harris T.* Concurrent Programming Without Locks. — 2007.
- 7 *Harris T. L.* A Pragmatic Implementation of Non-Blocking Linked-Lists. — 2001.
- 8 Java ConcurrentSkipListSet [Электронный ресурс]. — 2021. — URL: <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/ConcurrentSkipListSet.html>.
- 9 Java TreeSet [Электронный ресурс]. — 2021. — URL: <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/TreeSet.html>.
- 10 *Maurice Herlihy Yossi Lev V. L., Shavit N.* A Provably Correct Scalable Concurrent Skip List. — 2006. — URL: <https://www.cs.tau.ac.il/~shanir/nir-pubs-web/Papers/OPODIS2006-BA.pdf>.

- 11 *Michael M. M.* High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. — 2002. — URL: <https://dl.acm.org/doi/10.1145/564870.564881>.
- 12 *Tyler Crain Vincent Gramoli M. R.* A Contention-Friendly, Non-Blocking Skip List. — 2012.



## ПРИЛОЖЕНИЕ А. ПРИЛОЖЕНИЕ С ЛИСТИНГАМИ ФУНКЦИЙ

Листинг А.1 – Функция удаления элемента из массива узла

```

function remove(node, v) : boolean
  end ← node.array.end
  key ← node.array.key
  if end = 0 then
    return false
  else
    positionOfV ← -1
    i ← 0
    while i ≠ end do
      cur ← key[i]
      if v = cur then
        positionOfV ← i
        break
      end if
      i ← i + 1
    end while
    if positionOfV ≠ -1 then
      key[positionOfV] ← key[end - 1]
      key[end - 1] ← EMPTY
      node.array.end ← end - 1
      node.deleted ← end = 1
      return true
    else
      return false
    end if
  end if
end function

```

## Листинг А.2 – Функция добавления элемента

```

function add(v) : boolean
  cur ← find(v)
  cur.lock.lock()
  cur ← moveForwardBlocking(cur, 0, v)
  if cur == head || cur.deleted then
    newNode ← split(cur, v)
    newNode.array.key[0] ← v
    newNode.initialMin ← v
    newNode.array.end ← 1
    newNode.next.add(cur.next[0])
    cur.next[0] ← newNode
    cur.lock.unlock()
  else
    vPos ← posOfV(cur, v)
    if vPos ≠ -1 then
      cur.lock.unlock()
      return false
    else
      end ← cur.array.end
      emptyPos ← end == k ? -1 : end
      if emptyPos == -1 then
        <newNode, kArrayL> ← split(cur, v)
        newNode.next.add(cur.next[0])
        cur.next[0] ← newNode
        cur.array ← NodeArray(kArrayL, l)
        cur.lock.unlock()
      else
        cur.array.key[emptyPos] ← v
        cur.array.end ← cur.array.end + 1
        cur.lock.unlock()
        return true
      end if
    end if
  end if
  //newNode was created, so it's height may be increased
  heightForNode ← generateHeight()
  if heightForNode == 0 then
    return true
  end if
  increaseNodeHeightTo(newNode, heightForNode)
  return true
end function

```

Листинг А.3 – Функция вставляющая узел на верхние уровни до указанной высоты

```

function increaseNodeHeightTo(newNode, heightForNode)
  path ← findWithPathSaving(newNode)
  for curLevel ← 1, heightForNode do
    cur ← path[curLevel] == EMPTY?H : path[curLevel]
    cur.lock.lock()
    if cur.next.size() ≤ curLevel then
      cur.next.add(T)
      maxLevel.incrementAndGet()
    end if
    cur ← moveForwardBlocking(cur, curLevel, newNode.initialMin)
    newNode.lock.lock()
    if newNode.deleted then
      //someone already deleted that node
      newNode.lock.unlock()
      cur.lock.unlock()
      return
    end if
    newNode.next.add(cur.next[curLevel])
    cur.next[curLevel] ← newNode
    newNode.lock.unlock()
    cur.lock.unlock()
  end for
end function

```

Листинг А.4 – Функция разделения узла на два

```

function split(cur, v) : <Node, NodeArray>
  newNode ← Node()
  l ← 0
  r ← 0
  kArrayL ← AtomicReferenceArray()
  newNodeKey ← newNode.array.key
  oldNodeKey ← cur.array.key
  for i ← 0, k - 1 do
    curValue ← oldNodeKey[i]
    if v < curValue then
      newNodeKey[r ++] ← curValue
    else
      kArrayL[l ++] ← curValue
    end if
  end for
  if r == k then
    newNodeKey[l ++] ← v
    newMin ← EMPTY
    for i ← 0, k - 1 do
      if newMin EMPTY then
        newMin ← newNodeKey[i]
      else
        if newNodeKey[i] < newMin then
          newMin ← newNodeKey[i]
        end if
      end if
    end for
    newNode.initialMin ← newMin
  else
    newNodeKey[r ++] ← v
    newNode.initialMin ← v
  end if
  newNode.array.end ← r
  return <newNode, kArrayL>
end function

```

Листинг А.5 – Функция поиска позиции элемента в массиве

```
function posOfV(node, v) : int
  end ← node.array.end
  if end == 0 then
    return -1
  else
    key ← node.array.key
    i ← 0
    while i ≠ end do
      cur ← key[i]
      if v == cur then
        return i
      end if
      i ← i + 1
    end while
  end if
  return -1
end function
```

## Листинг А.6 – Функция удаления элемента

```

function remove(v) : boolean
  cur ← find(v)
  cur.lock.lock()
  cur ← moveForwardBlocking(cur, 0, v)
  if cur.deleted then
    cur.lock.unlock()
    return false
  end if
  optimisticRemoveResult ← remove(cur, v)
  deleted ← cur.deleted
  cur.lock.unlock()
  if optimisticRemoveResult then
    if deleted then
      path ← findAllPrevious(cur)
      forDeletion ← cur
      curLevel ← forDeletion.next.size() - 1
      while curLevel ≥ 0 do
        cur ← path[curLevel]
        cur.lock.lock()
        cur ← firstNotPhysicallyDeleted(cur, curLevel)
        next ← cur.next[curLevel]
        while next ≠ forDeletion do
          next.lock.lock()
          cur.lock.unlock()
          cur ← next
          next ← next.next[curLevel]
        end while
        forDeletion.lock.lock()
        cur.next[curLevel] ← forDeletion.next[curLevel]
        forDeletion.deletedBy.add(cur)
        forDeletion.lock.unlock()
        cur.lock.unlock()
        curLevel – –
      end while
    end if
    return true
  end if
  return false
end function

```

## ПРИЛОЖЕНИЕ Б. ПРИЛОЖЕНИЕ С РЕЗУЛЬТАТАМИ СРАВНЕНИЯ

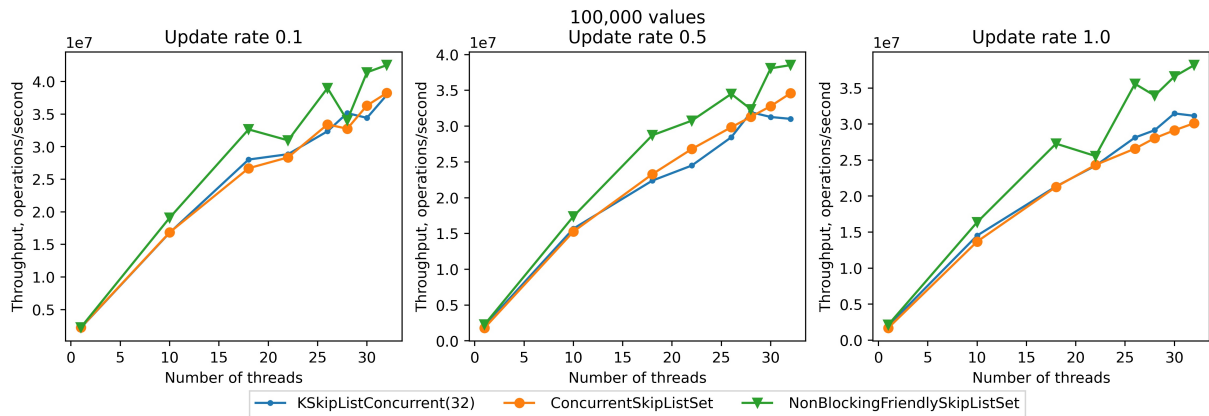


Рисунок Б.1 – Сравнение полученного решения с другими с ссылочными типами

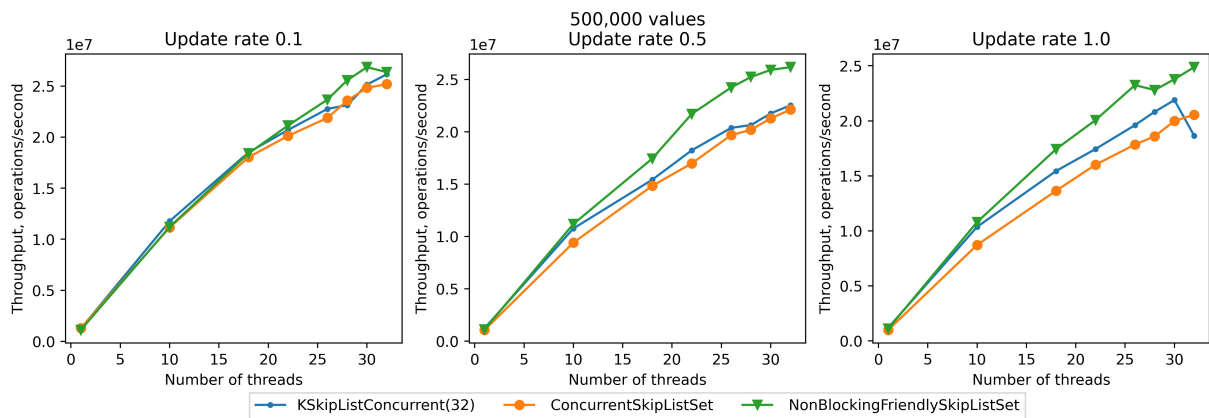


Рисунок Б.2 – Сравнение полученного решения с другими с ссылочными типами

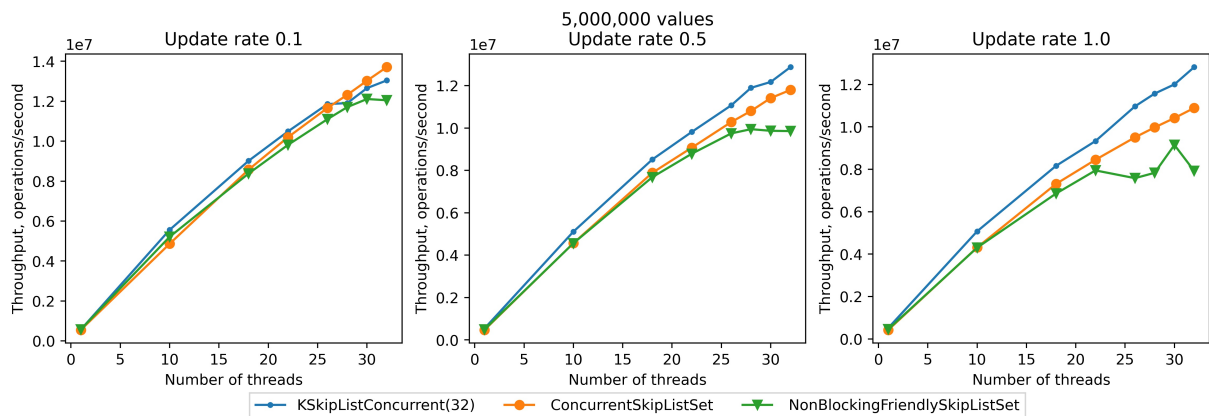


Рисунок Б.3 – Сравнение полученного решения с другими с ссылочными типами

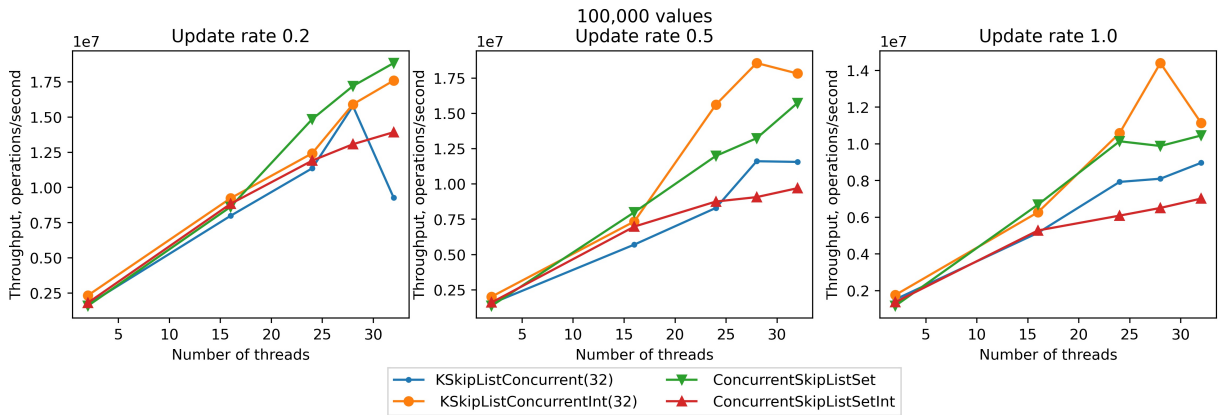


Рисунок Б.4 – Сравнение полученного решения с другими с не ссылочным типом *int*

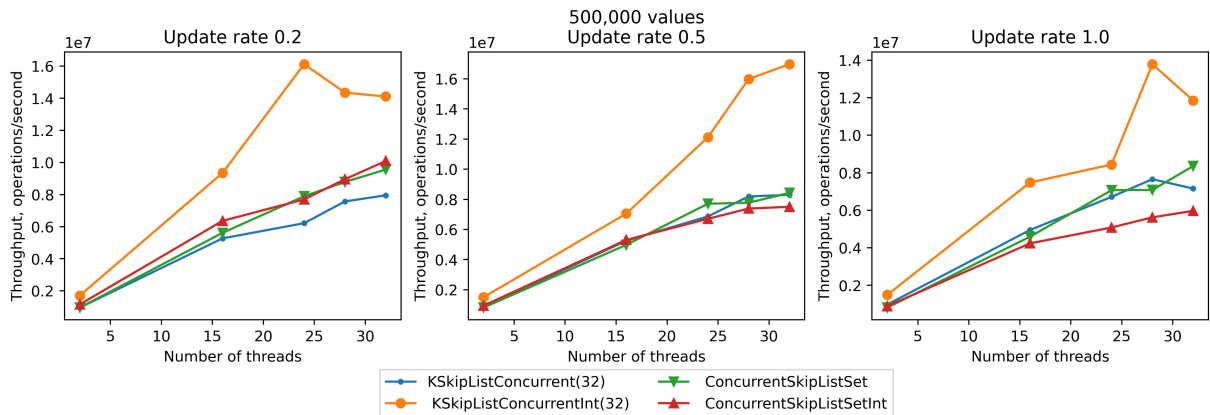


Рисунок Б.5 – Сравнение полученного решения с другими с не ссылочным типом *int*

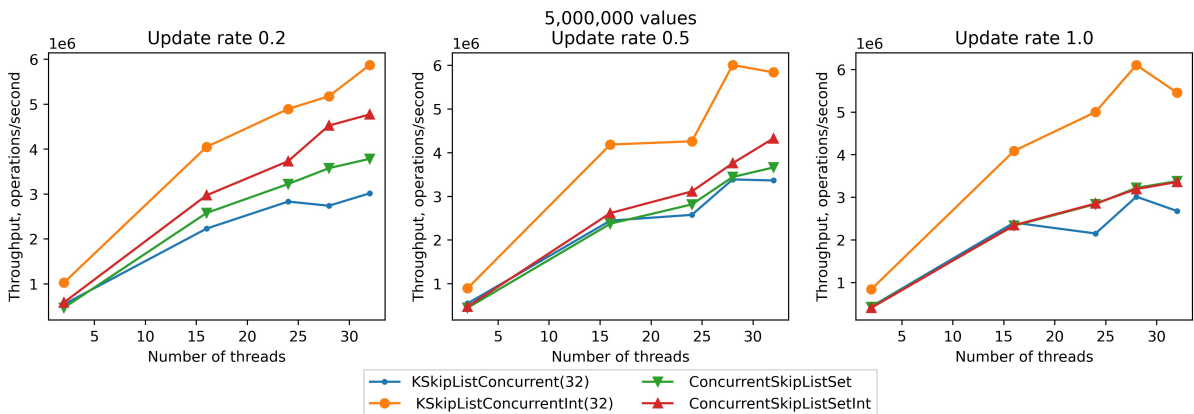


Рисунок Б.6 – Сравнение полученного решения с другими с не ссылочным типом *int*