Правительство Российской Федерации
Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет «Высшая школа экономики»

Факультет компьютерных наук
Образовательная программа бакалавриата 01.03.02 «Прикладная математика и информатика»

ОТЧЕТ
по производственной практике

в (на) _____ООО «Интеллиджей Лабс»_____
(название организации, предприятия)

Выполнил(а) студент(ка)
группы БПИ186
__А.В. Удовиченко____
(инициалы, фамилия)
_____
(подпись)

Проверили:

Руководитель практики от предприятия

_____Бобович Юлия Александровна_____
(ФИО руководителя практики от предприятия)

_____Координатор проекта_____
(должность руководителя практики от предприятия)

Дата __06.09.2021_____ _____8_____ _____
(оценка) (подпись)

Руководитель практики от факультета компьютерных наук

_____
(ФИО руководителя практики от ФКН)

_____
(должность руководителя практики от ФКН)

Дата _____ _____ _____
(оценка) (подпись)

Москва – 2021

Аннотация

В рамках практики в Concurrent Computing research group в компании ООО «Интеллиджей Лабс» (далее JetBrains) были протестированы некоторые известные современные многопоточные алгоритмы. Для тестирования использовался инструмент Lincheck, разрабатываемый в этой лаборатории. В некоторых алгоритмах были найдены серьёзные ошибки.

Содержание

1 Цель и задачи практики

Целью практики был поиск ошибок в современных многопоточных алгоритмах, опубликованных на конференциях высшего уровня.

Задачи практики состояли в:

- знакомстве с современными многопоточными алгоритмами,
- освоении инструмента тестирования многопоточных алгоритмов Lincheck,
- поиске ошибок в алгоритмах при помощи Lincheck.

2 О компании

JetBrains — международная компания, создающая интеллектуальные инструменты для повышения продуктивности разработчиков и творческих команд. Головной офис компании находится в Праге. Офисы разработки и исследований действуют в Амстердаме, Бостоне, Москве, Мюнхене, Новосибирске и Санкт-Петербурге [1].

3 Обзор изученных материалов

Были просмотрены некоторые лекции курса унивеситета ИТМО по многопоточному программированию, в том числе про Treiber stack [2], Michael-Scott Queue [3], relaxed-алгоритмы (MultiQueue [4]), многопоточное множество на односвязном списке, построение атомарных объектов и блокировки (mutex [5], алгоритм Петерсона [6], алгоритм Лампорта [7], двухфазная блокировка [8]), многопоточные хеш-таблицы (Lock-Free Wait-Free Hash Table [9], Lock-free Dynamic Hash Tables with Open Addressing [10]), и выполнены соответствующие задания к лекциям [11][12][13][14][15][16][17][18][19] (результаты выполнения заданий находятся в приватных репозиториях, но при необходимости могут быть предоставлены).

Были изучены материалы для работы с инструментом Lincheck [20][21], в том числе доклады на конференциях Никиты Коваля [22][23][24] и Марии Соколовой [25].

```
@ModelCheckingCTest
class HashMapTest {
    private val map = HashMap<Int, Int>()

    @Operation
    fun put(key: Int, value: Int): Int? = map.put(key, value)

    @Operation
    fun get(key: Int): Int? = map[key]

    @Test
    fun test() = LinChecker.check(this.javaClass)
}
```

Рис. 4.1: Пример использования Lincheck

4 Lincheck

Lincheck — это новый инструмент для тестирования многопоточных алгоритмов, написанных на JVM-based языках, например, Java, Scala или Kotlin. Он разрабатывается в Concurrent Computing research group [26] в компании JetBrains. При использовании Lincheck разработка новых алгоритмов становится проще и гораздо эффективнее.

Рассмотрим пример использования Lincheck (рис. 4.1). Допустим, необходимо протестировать `HashMap`. Создаём класс для тестирования, в данном случае `HashMapTest`. Создаём в этом классе функции, вызывающие соответствующие функции `HashMap`, которые необходимо протестировать. К каждой из этих функций добавляем аннотацию `@Operation`. Создаём тестирующую функцию `test()` и выбираем режим тестирования в аннотации класса. В данном случае выбрана аннотация `@ModelCheckingCTest`, то есть режим тестирования model checking (см. разд. 4.2). В нашем простейшем примере этих действий достаточно для запуска тестирования. Далее рассмотрим два использовавшихся режима тестирования.

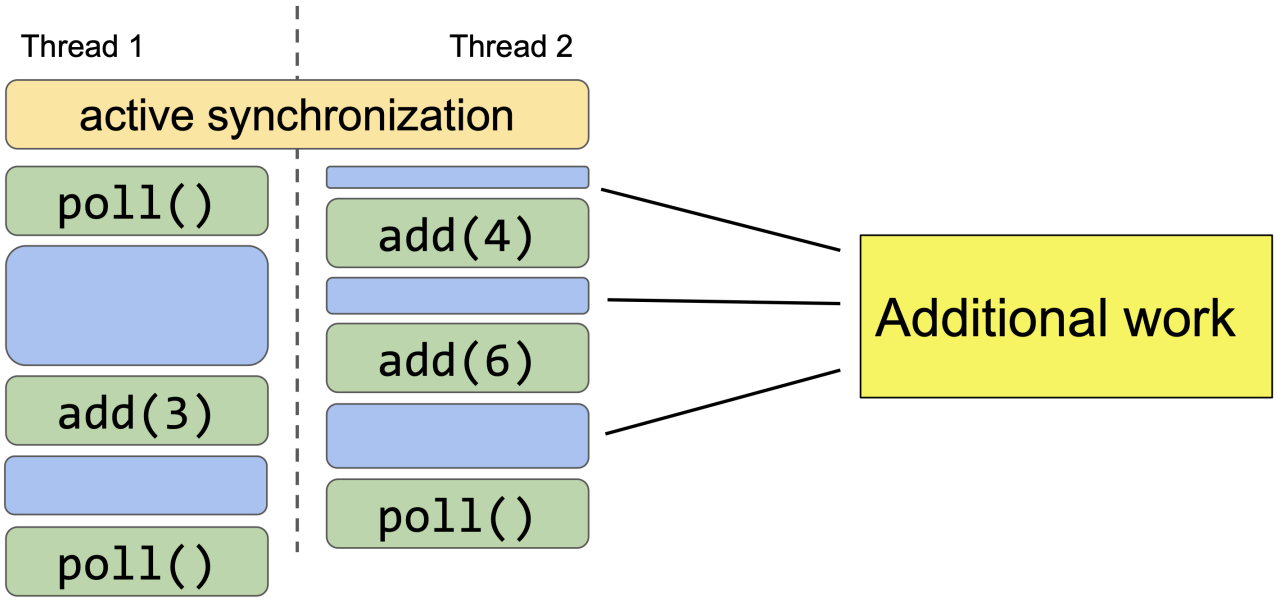

Рис. 4.2: Добавление дополнительной работы в stress-режиме

4.1 Stress-режим

Stress-режим — самый первый появившийся в Lincheck режим тестирования. В этом режиме тесты запускаются много раз в необходимом количестве потоков, а затем результаты исполнения проверяются на линеаризуемость. Для увеличения покрытия различных сценариев исполнения между операциями в потоке добавляется случайная дополнительная работа (рис. 4.2). Если в процессе тестирования найдена ошибка, то создаётся отчёт с описанием теста, на котором она произошла. Этот режим тестирования успешно проверяет структуры данных на последовательную согласованность и находит гонки.

4.2 Режим model checking

Режим model checking исполняет всё в одном потоке с переключениями контекста. Lincheck перебирает различные интерливинги, увеличивая количество переключений контекста. Этот режим тестирования не находит гонки, но проверяет структуры данных на последовательную согласованность и находит дедлоки. Главное преимущество этого режима тестирования в том, что в случае нахождения ошибки в отчёте указывается конкретный интерливинг,

```
= Invalid execution results =
Parallel part:
| put(5, -8): null | put(5, 4): null |
= The following interleaving leads to the error =
Parallel part trace:
| put(5, -8)                                                                  |                          |
|   put(5,-8): null at HashMapTest.put(HashMapTest.java:40)                   |                          |
|     putVal(0,5,-8,false,true): null at HashMap.put(HashMap.java:607)        |                          |
|       table.READ: null at HashMap.putVal(HashMap.java:623)                  |                          |
|       resize(): Node[]@1 at HashMap.putVal(HashMap.java:624)                |                          |
|         table.READ: null at HashMap.resize(HashMap.java:673)               |                          |
|         threshold.READ: 0 at HashMap.resize(HashMap.java:675)               |                          |
|         threshold.WRITE(12) at HashMap.resize(HashMap.java:697)             |                          |
|         switch                                                              |                          |
|                                                                             | put(5, 4): null          |
|                                                                             |   thread is finished     |
|         table.WRITE(Node[]@1) at HashMap.resize(HashMap.java:700)           |                          |
|       READ: null at HashMap.putVal(HashMap.java:625)                        |                          |
|       WRITE(Node@1) at HashMap.putVal(HashMap.java:626)                     |                          |
|       modCount.READ: 1 at HashMap.putVal(HashMap.java:656)                  |                          |
|       modCount.WRITE(2) at HashMap.putVal(HashMap.java:656)                 |                          |
|       size.READ: 1 at HashMap.putVal(HashMap.java:657)                      |                          |
|       size.WRITE(2) at HashMap.putVal(HashMap.java:657)                     |                          |
|       threshold.READ: 9 at HashMap.putVal(HashMap.java:657)                 |                          |
|   result: null                                                              |                          |
|   thread is finished                                                        |                          |
```

Рис. 4.3: Результат работы режима model checking

исполнение которого привело к ошибке. Пример такого результата приведён на рис. 4.3.

5 Результаты

Было протестировано 10 многопоточных структур данных, каждая тестировалась и в stress-режиме, и в режиме model checking. Весь написанный код, описание найденных ошибок и прочая информация о тестировании находится в репозитории: https://github.com/AnthonyUdovichenko/concurrent-algorithms-testing.

В следующих структурах данных ошибок найдено не было:

- `ConcurrentHashMap` [27],
- `ConcurrentHashMultiset` [28],
- `LockFreeKSTRQ` [29],
- `NonBlockingTorontoBSTMap` [30],
- `ConcurrentLinkedQueue` [31],

- `LockBasedFriendlyTreeMap` [32].

Далее разберём структуры данных, в которых были найдены ошибки.

5.1 SnapTreeMap

Данная структура данных была описана в статье [33], но ошибка была найдена в реализации [34], написанной одним из авторов статьи (эта часть алгоритма просто не была описана в статье). Визуализация ошибки представлена на рис. 5.1.

Изначально дерево пустое. В предварительной последовательной части исполнения в дерево добавляются вершины с ключами 2 и 4. Далее начинается параллельная часть. Тест состоит в том, что одновременно работают два потока: первый поток добавляет в дерево вершину с ключом 6, а затем удаляет из дерева вершину с ключом 4; второй поток возвращает наибольший ключ в дереве.

К ошибке приводит следующий интерливинг. Второй поток начинает работу и находит нужную вершину (это вершина с ключом 4), но не успевает прочитать значение в ней до переключения контекста. Затем первый поток вставляет вершину с ключом 6, дерево перестраивается как показано на визуализации. Далее первый поток логически удаляет вершину с ключом 4. В силу особенностей алгоритма физически вершина не удаляется, но в поле значения проставляется `null`. После этого второй поток возобновляет свою работу и завершается с `AssertionError` из-за того, что алгоритм не предполагает увидеть в поле значения найденной вершины `null`. Мы заменили в коде `assert` на возвращение специального значения, показывающего необходимость запустить операцию заново. После этого Lincheck перестал находить какие-либо ошибки в этом алгоритме.

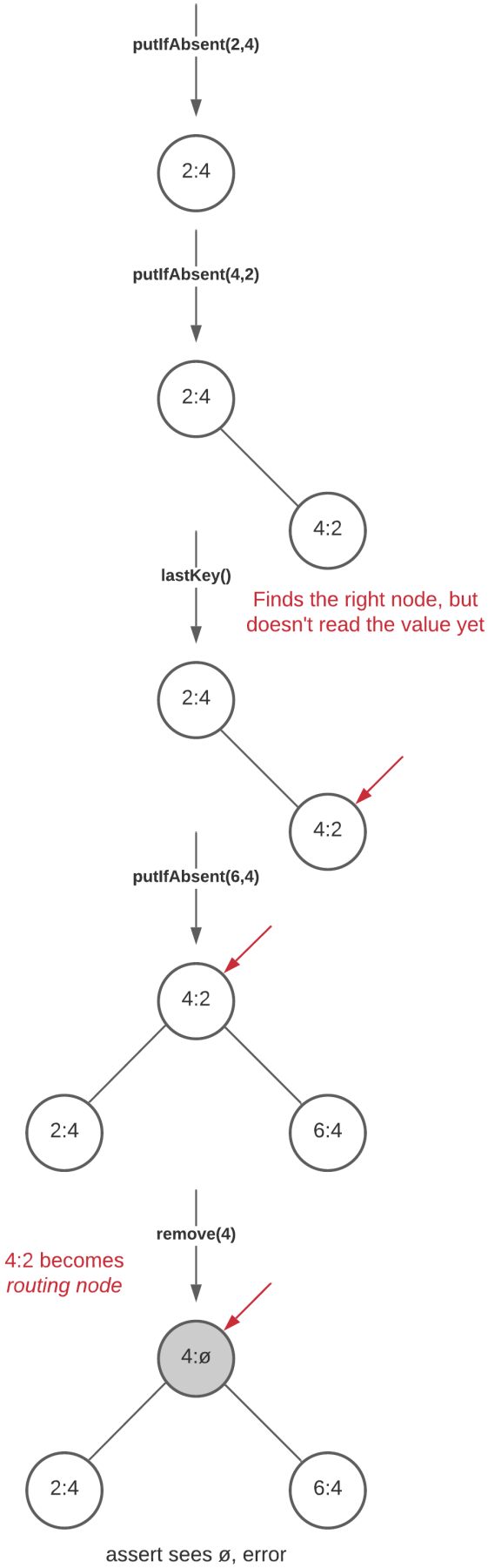

Рис. 5.1: Визуализация ошибки в SnapTreeMap

5.2 LogicalOrderingAVL

Данная структура данных была описана в статье [35], но ошибка была найдена в реализации [36], написанной одним из авторов статьи (настолько низкоуровневые синхронизации не были описаны в самой статье).

Изначально дерево логически пустое. В предварительной последовательной части исполнения в дерево добавляется вершина с ключом 5. Далее начинается параллельная часть. Тест состоит в том, что одновременно работают три потока: первый поток добавляет вершину с ключом 3, второй поток добавляет вершину с ключом 1, третий поток удаляет вершину с ключом 3 (если она присутствует в дереве).

К ошибке приводит следующая последовательность взятия блокировок. Второй поток начинает работу, берёт необходимые ему блокировки и заканчивает вставку вершины с ключом 1, но не успевает до переключения отпустить блокировки, в том числе `treeLock` вершины с ключом 5. Происходит переключение, и первый поток начинает работу. Он вставляет вершину с ключом 3 и для подготовки к перебалансировке дерева пытается взять `treeLock` вершины с ключом 5. Так как второй поток держит эту блокировку, взять её не получается, и первый поток начинает ждать. Происходит переключение, и начинает работу третий поток, который хочет удалить вершину с ключом 3. Для этого он начинает брать блокировки и в том числе берёт `treeLock` вершины с ключом 3 и пытается взять `treeLock` вершины с ключом 1. Так как первый поток держит эту блокировку, это не получается, и третий поток тоже начинает ждать. Потом второй поток снимает все блокировки и заканчивает работу. После этого первый поток успешно берёт `treeLock` вершины с ключом 5, но затем зависает в бесконечном цикле попыток взять `treeLock` вершины с ключом 3, а эту блокировку держит третий поток. В итоге произошёл дедлок, когда первый и третий потоки бесконечно ждут друг друга.

```
public void lock(){
    while(true){
        long readSeqNumber = seqNumber;
        while((readSeqNumber % 2) != 0){
            unsafe.fullFence();
            unsafe.fullFence();
            readSeqNumber = seqNumber;
        }
        if(seqNumberUpdater.compareAndSet(this, readSeqNumber, readSeqNumber + 1)){
            break;
        }
    }
}

public void unlock(){
    seqNumber = seqNumber + 1;
}
```

Рис. 5.2: Функции `lock()` и `unlock()` алгоритма блокировки в `CATreeMapAVL`

5.3 CATreeMapAVL

Данная структура данных была описана в статье [37], но ошибка была найдена в реализации [38], написанной одним из авторов статьи (эта часть алгоритма не была описана в статье).

На рис. 5.2 представлены функции `lock()` и `unlock()` алгоритма блокировок этой структуры данных. Заметим, что, если попытаться разблокировать разблокированную блокировку, то она окажется заблокированной.

На протяжении всего теста дерево логически пустое. Тест состоит в том, что одновременно работают два потока: первый поток очищает дерево, а затем пытается получить значение, лежащее в вершине с ключом 7; второй поток проверяет дерево на пустоту.

К ошибке приводит следующий интерливинг. В начале исполнения в `root` дерева лежит пустая фиктивная вершина. Второй поток начинает работу, читает `root`, но не успевает взять блокировку на него. Происходит переключение, и первый поток начинает работу. Для очищения дерева он записывает в `root` новую фиктивную вершину. Затем второй поток берёт блокировку на старый `root`, и, чтобы посчитать размер дерева, читает уже новый `root` и на-

чинает работать с ним. Таким образом, инвариант, что поток работает только с заблокированным деревом, уже перестал выполняться. Затем он освобождает блокировку, но уже для нового `root`, который не был заблокирован, то есть блокирует его, а затем завершается. В итоге первый поток пытается взять блокировку на `root`, но тот оказывается заблокированным, поэтому ожидание становится вечным, то есть происходит дедлок.

5.4 ConcurrencyOptimalTreeMap

Данная структура данных была описана в статье [39], но ошибка была найдена в реализации [40], написанной одним из авторов статьи (этой ошибки в статье нет).

Изначально дерево логически пустое. Тест состоит в том, что одновременно работают два потока: первый поток добавляет вершину с ключом 5, второй поток добавляет вершину с ключом 6.

К ошибке приводит следующий интерливинг. Дерево физически состоит из одной фиктивной вершины, которая лежит в `ROOT`. Второй поток начинает работу и хочет вставить новую вершину как левого ребёнка `ROOT`. Он берёт блокировку на левое ребро `ROOT`, затем происходит переключение. Первый поток начинает работу и с помощью вызова `traverse` находит `Window`, то есть место, куда он хочет вставить вершину. В `Window` есть три значения: `gprev`, `prev`, `cur`. После вызова `traverse` `gprev` и `cur` равны `null`, `prev` равен `ROOT`. Эти значения правильные с точки зрения идеи алгоритма. Потом первый поток пытается взять блокировку на левое ребро `ROOT`, но это не получается, так как эту блокировку держит второй поток. После этого первый поток попадает в участок кода, представленный на рис. 5.3. Так как `ROOT` не удалён, исполнение попадает в `else`, где происходит явно ошибочное действие: теперь `gprev` и `prev` равны `null`, а `cur` равен `root`. Судя по коду в целом, авторы хотели сохранить инвариант, что в `cur` лежит либо `null`, либо реальная (не фиктивная) вершина. В этой строчке инвариант был нарушен. Затем поток

```
if (prev.deleted) {
    window.reset();
} else {
    window.set(window.prev, window.gprev);
}
```

Рис. 5.3: Ошибка в коде `ConcurrencyOptimalTreeMap`

```
public Window traverse(Object key, Window window) {
    final Comparable<? super K> k = comparable(key);
    int comparison;
    Node curr = window.curr;
    while (curr != null) {
        comparison = k.compareTo(curr.key);
        if (comparison == 0) {
            return window;
        }
        if (comparison < 0) {
            curr = curr.l;
        } else {
            curr = curr.r;
        }
        window.add(curr);
    }
    return window;
}
```

Рис. 5.4: Код функции `traverse`

начал делать всё сначала, запустив вторую итерацию цикла. Там снова был вызван `traverse`. Код функции `traverse` представлен на рис. 5.4. Обратим внимание на выделенную строку. У фиктивной вершины `key` равен `null`, поэтому `compareTo` кидает `NullPointerException`.

6 Заключение

В течение практики было протестировано 10 известных реализаций многопоточных структур данных. В 4 из них были найдены серьёзные ошибки, которые с некоторой вероятностью могут привести к неправильной работе

алгоритма. Отметим, что во всех 4 случаях найденные ошибки находились в той части алгоритма, которая по тем или иным причинам не была освещена в статье. Это довольно логично: статьи перед публикацией проходят процесс проверки и рецензирования, а код в репозитории фактически не проверяется никем. Это подводит нас к выводу, что хорошим решением этой проблемы было бы прилагать к статье код, демонстрирующий работу предложенного алгоритма, чтобы он также проходил проверку и рецензирование.

Также стоит отметить полезность и применимость инструмента Lincheck, который в рамках практики показал, что он успешно решает возложенные на него задачи и находит такие ошибки, которые ранее никто не находил.

В данный момент обсуждается вопрос публикации полученных во время практики результатов.

Список источников

1. JetBrains. Справка о компании.

2. Treiber R. K. Systems programming: Coping with parallelism. – New York : International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

3. Michael M. M., Scott M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms //Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing. – 1996. – С. 267-275.

4. Rihani H., Sanders P., Dementiev R. Multiqueues: Simple relaxed concurrent priority queues //Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures. – 2015. – С. 80-82.

5. Dijkstra E. W. Cooperating sequential processes //The origin of concurrent programming. – Springer, New York, NY, 1968. – С. 65-138.

6. Peterson G. L., GL P. Myths about the mutual exclusion problem. – 1981.

7. Bell G., Siewiorek D., Fuller S. H. A New Solution of Dijkstra's Concurrent Programming Problem //Communications of the ACM. – С. 453-455.

8. Bernstein P. A., Hadzilacos V., Goodman N. Concurrency control and recovery in database systems. – Reading : Addison-wesley, 1987. – Т. 370.

9. Click C. A lock-free wait-free hash table //work presented as invited speaker at Stanford. – 2008.

10. Gao H., Groote J. F., Hesselink W. H. Lock-free dynamic hash tables with open addressing //Distributed Computing. – 2005. – Т. 18. – №. 1. – С. 21-42.

11. https://github.com/ITMO-MPP/possible-executions-analysis (дата обр. 04.09.2021).

12. https://github.com/ITMO-MPP/msqueue (дата обр. 04.09.2021).

13. https://github.com/ITMO-MPP/stack-elimination (дата обр. 04.09.2021).

14. https://github.com/ITMO-MPP/dijkstra (дата обр. 04.09.2021).

15. https://github.com/ITMO-MPP/linked-list-set (дата обр. 04.09.2021).

16. https://github.com/ITMO-MPP/fine-grained-bank (дата обр. 04.09.2021).

17. https://github.com/ITMO-MPP/lamport-lock-fail (дата обр. 04.09.2021).

18. https://github.com/ITMO-MPP/dynamic-array (дата обр. 05.09.2021).

19. https://github.com/ITMO-MPP/hash-table (дата обр. 05.09.2021).

20. Koval N. et al. Testing concurrency on the JVM with lincheck //Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. – 2020. – С. 423-424.

21. Lincheck. https://github.com/Kotlin/kotlinx-lincheck (дата обр. 05.09.2021).

22. Никита Коваль — Тестирование lock-free алгоритмов, или Поиск иголки в стоге сена. https://www.youtube.com/watch?v=_0_HOnTSS0E (дата обр. 05.09.2021).

23. Nikita Koval — Lin-Check: Testing concurrent data structures in Java. https://www.youtube.com/watch?v=hwbpUEGHvvY (дата обр. 05.09.2021).

24. Testing concurrent algorithms with Lincheck. https://nkoval.com/presentations/joker_2019_lincheck.pdf (дата обр. 05.09.2021).

25. Мария Соколова — Lincheck Тестирование многопоточной структуры данных в Java. https://www.youtube.com/watch?v=YAb7YoEd6mM (дата обр. 05.09.2021).

26. Concurrent Computing research group, JetBrains. https://research.jetbrains.org/groups/concurrent_computing/ (дата обр. 05.09.2021).

27. ConcurrentHashMap (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html (дата обр. 05.09.2021).

28. ConcurrentHashMultiset (Guava: Google Core Libraries for Java 19.0 API). https://guava.dev/releases/19.0/api/docs/com/google/common/collect/ConcurrentHashMultiset.html (дата обр. 05.09.2021).

29. Brown T., Avni H. Range queries in non-blocking k-ary search trees //International Conference On Principles Of Distributed Systems. – Springer, Berlin, Heidelberg, 2012. – С. 31-45.

30. Ellen F. et al. Non-blocking binary search trees //Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing. – 2010. – С. 131-140.

31. ConcurrentLinkedQueue (Java Platform SE 7). https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html (дата обр. 05.09.2021).

32. Crain T., Gramoli V., Raynal M. A contention-friendly binary search tree //European Conference on Parallel Processing. – Springer, Berlin, Heidelberg, 2013. – С. 229-240.

33. Bronson N. G. et al. A practical concurrent binary search tree //ACM Sigplan Notices. – 2010. – Т. 45. – №. 5. – С. 257-268.

34. SnapTree. https://github.com/nbronson/snaptree (дата обр. 05.09.2021).

35. Drachsler D., Vechev M., Yahav E. Practical concurrent binary search trees via logical ordering //Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming. – 2014. – С. 343-356.

36. LogicalOrderingAVL. https://github.com/gramoli/synchrobench/blob/master/java/src/trees/lockbased/LogicalOrderingAVL.java (дата обр. 05.09.2021).

37. Sagonas K., Winblad K. Contention adapting search trees //2015 14th International Symposium on Parallel and Distributed Computing. – IEEE, 2015. – С. 215-224.

38. CATreeMapAVL. https://github.com/gramoli/synchrobench/blob/master/java/src/trees/lockbased/CATreeMapAVL.java (дата обр. 06.09.2021).

39. Aksenov V. et al. A concurrency-optimal binary search tree //European Conference on Parallel Processing. – Springer, Cham, 2017. – С. 580-593.

40. ConcurrencyOptimalTreeMap. https://github.com/gramoli/synchrobench/blob/master/java/src/trees/lockbased/ConcurrencyOptimalTreeMap.java (дата обр. 06.09.2021).