МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Параллельные алгоритмы»
Тема: Разработка потокобезопасной структуры данных

Студент гр. 3342 _____ Иванов Д. М.

Преподаватель _____ Сергеева Е. И.

Санкт-Петербург
2025

Задание

Разработать потокобезопасную структуру данных, обеспечивающую потенциально одновременную работу с ней нескольких потоков (читателей и писателей). Структура данных выбирается из предложенного списка. Разработать пайплайн тестирования корректности и производительности потокобезопасной структуры данных. Исследовать производительность потокобезопасной структуры данных. В качестве базовой реализации для сравнения взять ту же самую структуру с грубой блокировкой. Сравнения производительности выполнить для разного количества потоков, работающих со структурой данных, проанализировать зависимость от соотношения потоков читателей и писателей. Метрики производительности приводить с осреднением, в отчете построить графики.

2.1 Структура данных с тонкой блокировкой

2.2 Структура данных без блокировок (lock-free)

однонаправленный связный список; Должен предоставлять методы insert, delete, find Баллы: часть 2.1: [0 ... 5]; часть 2.2: [0 ... 8];

Выполнение работы

Смыслом тонкой блокировки является блокировка не всей структуры данных, а только тех узлов, с которыми происходит работа данного потока, остальные узлы остаются доступными. Для реализации такой блокировки нужно добавить мьютексты каждоум узлу. При работе с определенными узлами их мьютексы будут блокироваться. При переходе к следующим — прошлый узел освобождаться, а новый — блокироваться.

Создадим отдельный класс для работы с этой блокировкой. Полями будут: головной узел, мьютекс для головы и атомарное поле размера (только 1 поток будет иметь к нему доступ).

void FineGrained::insert(int value, int index). В начале проверяется корректность индекса. Потом идет блокировка головы списка, так как необходимо помимо самих данных узла (как в других node) защитить саму еще голову списка, так как другие потоки могут изменить в целом структуру списка. После работы с head идет переход к узлам. Во время каждой итерации блокируются current и current→next. При переходе дальше current меняется, следующий current→next блокируется, а предыдущий current, который теперь потоку не нужен, освобождается. После вставки узла идет освобождение всех заблокированных узлов, начиная от хвоста.

int FineGrained::find(int index). Также идет обход с блокировкой нужного узла. Если индекса нет, возвращается -1. После нахождения индекса узел освобождается и возвращается число из этого узла.

void FineGrained::del(int index). В начале блокируется также голова списка. Потом идет обход и блокировка последовательная current, current→next. Когда дошли до удаляемого узла. Происходит дополнительная 3-я блокировка и после перестановки указателей 3 подряд разблокировки и очистка памяти от удаленного узла.

Тестирование программы на простых маленьких тестах приведено в таблице 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	testConcurrentDelete() создается список длиной 10 3 потока одновременно удаляют 1-ый элемент, и также 3 одновременно удаляют хвост	3 ==> 4 ==> 5 ==> 6 ==> 7 ==> end	Мы видим, что 6 потоков, работая одновременно смогли корректно удалить свои индексы, не создавая конфликтов и ошибок
2.	testConcurrentInsertFind() Работают 5 потоков, каждый делает по 3 вставки в конец списка и потом выводит в консоль одно значение, соответствующее индексу потока	0 2 0 2 200 100 ==> 0 ==> 1 ==> 2 ==> 200 ==> 201 ==> 202 ==> 101 ==> 102 ==> 300 ==> 301 ==> 302 ==> 400 ==> 401 ==> 402 ==> end	Все значения были успешно добавлены в списко. Такой случайный порядок нормальный в условяих параллельности, списко постоянно меняется. Также было 5 упешных параллельных find.
3.	testConcurrentInsertDelete() 5 потоков. Каждый делает по 10 вставок в начало или середину списка и сразу удаляет значение с начала списка.	empty	Все вставки и удаления корректно отработали. Было сделано 50 вставок и затем 50 удалений. Получился пустой список.

Следующим этапом задачи будет напсиание грубой блокировки связного спсика. Для этого напишем стандартный класс LinkedList без мьютексов. И создадим класс, который во время запуска того или иного метода будет полностью блокирвоать всю структуру.

Следующим шагом создадим структуру данных без блокировки: lock-free. Она будет основана на механизмах безопасного чтения и записи узлов: load(std::memory_order_acquire) гарантирует, что операции после него не будут переупорядочены до этой загрузки, обеспечивая актуальное видение данных (будет обеспечивать выполнение всех операций ниже только после выполнения данной загрузки); store(value, std::memory_order_release) гарантирует, что операции до него не будут переупорядочены после этой записи, обеспечивая корректную публикацию изменений (будет обеспечивать выполнение записи нового значения value только после выполнения всех операций выше). То есть данные операции икслючат перестановки порядка операций. Также все команды будут атомарными, то есть будут либо полностью выполняться, либо вообще не выполняться. И чтобы обеспечить корректную замену узлов будет перед выполнением происходить атомарная проверка CAS, чтобы не выполняться замену, если данный участок уже изменил другой поток, и узлы не равняются ожидаемым.

Результаты тестирование lock-free см. в таблице 2.

Таблица 2 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	testConcurrentDelete() создается список длиной 10 3 потока одновременно удаляют 1-ый элемент, и также 3 одновременно удаляют хвост	3 ==> 4 ==> 5 ==> 6 ==> 7 ==> end 3 ==> 4 ==> 5 ==> 6 ==> end 3 ==> 4 ==> 5 ==> 6 ==> 7 ==> 8 ==> end	Мы видим, что 6 потоков, работая одновременно смогли корректно удалить свои индексы, не создавая конфликтов и ошибок. В каких-то ситуациях потокам удается удалить хвост. Но если во время выполнения размер уже поменялся, то удаления не будет, однако это не помешает программе корректно завершить работу.
2.	testConcurrentIn-	0	Все значения были

	sertFind() Работают 5 потоков, каждый делает по 3 вставки в конец списка и потом выводит в консоль одно значение, соответствующее индексу потока	0 300 300 1 200 100 ==> 200 ==> 0 ==> 1 ==> 2 ==> 400 ==> 300 ==> 301 ==> 302 ==> 401 ==> 402 ==> 201 ==> 202 ==> 101 ==> 102 ==> end	успешно добавлены в списко. Такой случайный порядок нормальный в условяих параллельности, списко постоянно меняется. Также было 5 упешных параллельных find.
3.	testConcurrentInsert-Delete() 5 потоков. Каждый делает по 10 вставок в начало и сразу удаляет значение с начала списка.	==> end	Все вставки и удаления корректно отработали. Было сделано 50 вставок и затем 50 удалений. Получился пустой список.

Написание класса для пайплайн тестирования и замера времени

Реализуем класс, который будет тестировать lock-free, грубую или тонкую блокировку на случайных перациях и замерять время выполнения. В конструкторе задаются: объект списка с нужной блокировкой, количество читателей, кличество редакторов (50% insert, 50% delete) и начальный размер списка. Каждый поток будет выполнять 1 из 3 операций в случайном индексе. Потоки работают параллельно.

После запуска тестирования начинается замер времени. Окончание работы и вывод результата замера – признак, что читатели и редакторы корректно тработали над структурой данных.

Проводилось тестирование с усреднением на разных изначальных объемах списка (300, 3000, 5000). Анализировались разные соотношения редакторов и читателей (100 сумарных потоков в 1 тесте).

Получили следующие графики.

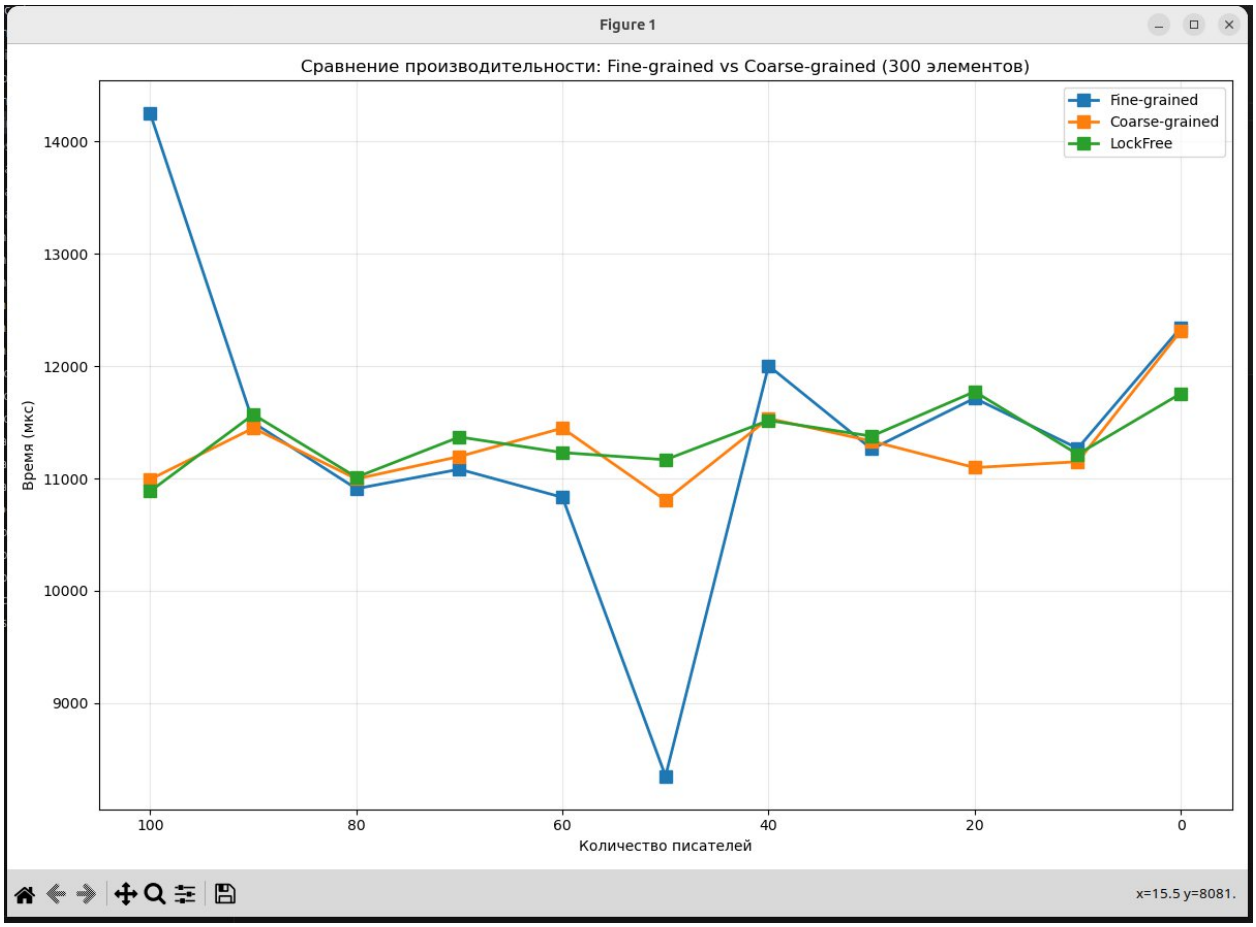

Рисунок 1 – график зависимости времени операций от соотношения редакторы/читатели на маленьком списке

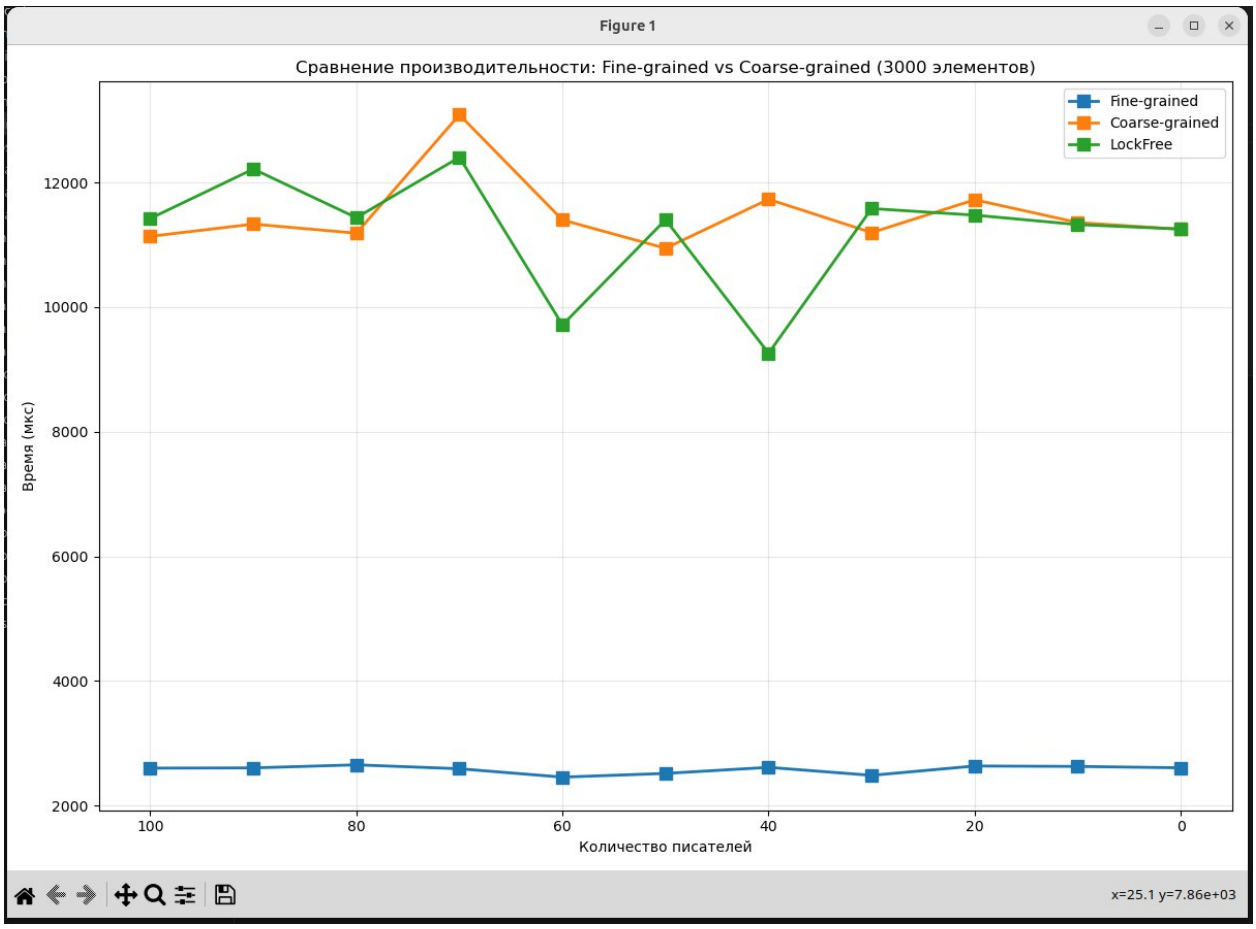

Рисунок 2 – график зависимости времени операций от соотношения редакторы/читатели на среднем списке

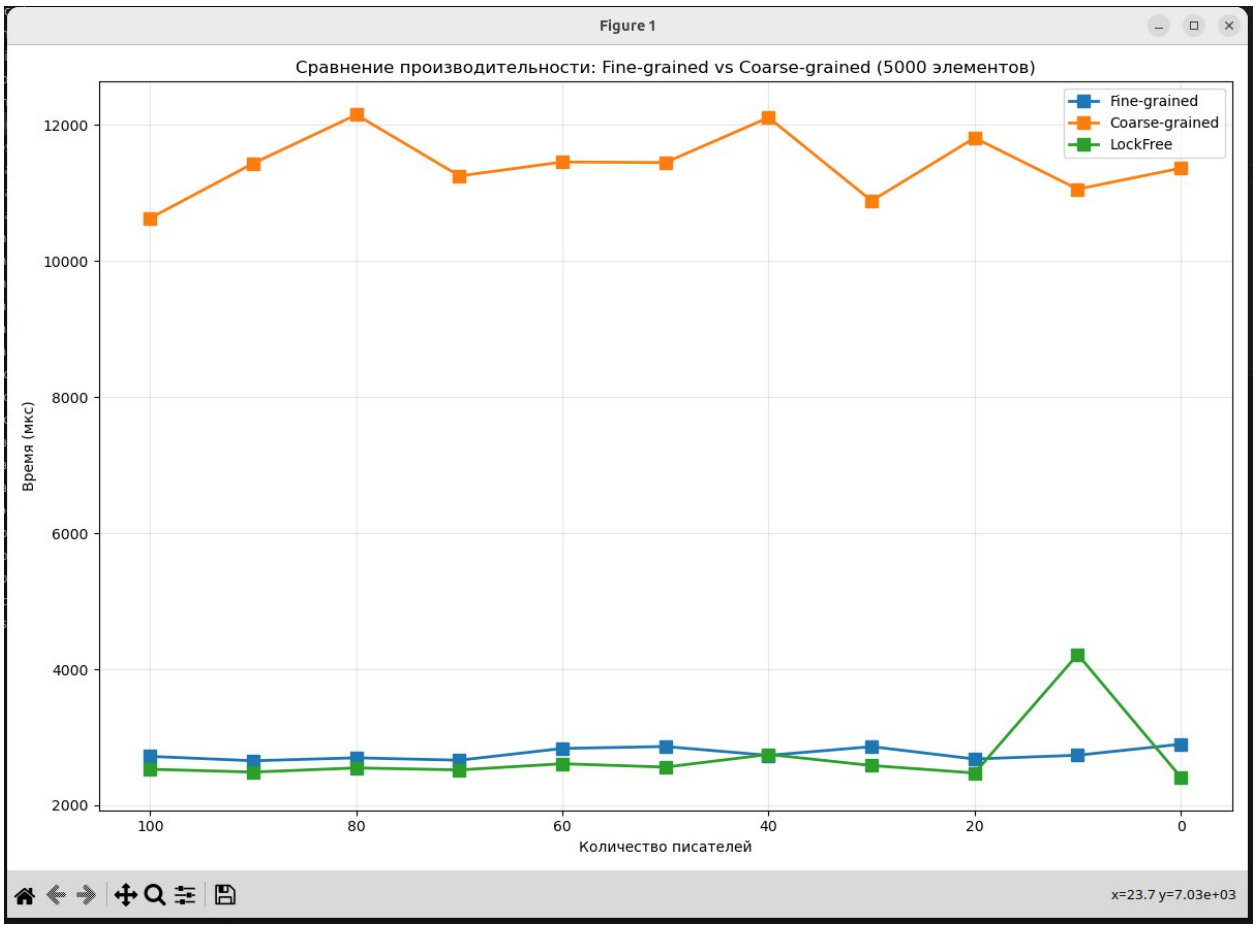

Рисунок 3 – график зависимости времени операций от соотношения редакторы/читатели на большом списке

Вывод

В ходе лабораторной работы были реализованы тонкая, грубая блокировке и lock-free для такой структуры, как односвязный список. Тонкая блокировка и lock-free прошли успешную проверку на работоспособность.

Также было проведено исследование по сравнению производительности этих блокировок. По графикам можно сказать, что на маленьких списках время выполнения параллельных операций примерно одинаквая у блокировок. На списках среднего размера тонкая блокирвка значительно превосходит другие в скорости, lock-free страдает из-за конкуренции и частых перезапусках программы при проверке CAS. А вот на больших размерах лучшую скорость показывает lock-free, потоки равномерно распределяются и возникает меньше конфликтов, тонкая блокировка тратит время на закрытие/открытие доступа, это, как мы видим, проигрывает на больших данных.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: FineGrainedLock.cpp

```
#include "FineGrainedLock.h"

void FineGrained::insert(int value, int index){
    if (index < 0 || index > size){
        return;
    }

    NodeMutex* new_node = new NodeMutex(value);

    if (index == 0){
        std::lock_guard<std::mutex> head_lock(head_mutex);
        new_node->next = head;
        head = new_node;
        size++;
        return;
    }

    std::unique_lock<std::mutex> head_lock(head_mutex);

    int i = 1;
    NodeMutex* current = head;
    current->mutex.lock();
    head_lock.unlock();
    if (current->next){
        current->next->mutex.lock();
    }

    while (current->next){
        if (i == index){
            break;
        }
        i++;

        NodeMutex* next_node = current->next;
        if (next_node->next){
            next_node->next->mutex.lock();
        }

        current->mutex.unlock();
        current = next_node;
    }

    new_node->next = current->next;
    current->next = new_node;

    if (new_node->next){
        new_node->next->mutex.unlock();
    }
    current->mutex.unlock();
    size++;
```

```
}

int FineGrained::find(int index){
    if (index < 0 || index >= size){
        return -1;
    }

    int i = 0;

    std::unique_lock<std::mutex> head_lock(head_mutex);
    NodeMutex* current = head;
    current->mutex.lock();
    head_lock.unlock();

    while (current){
        if (i == index){
            int val= current->value;
            current->mutex.unlock();
            return val;
        }
        i++;
        if (current->next){
            current->next->mutex.lock();
        }
        current->mutex.unlock();
        current = current->next;
    }
    return -1;
}

void FineGrained::del(int index){
    if (index < 0 || index >= size){
        return;
    }
    if (index == 0){
        std::lock_guard<std::mutex> head_lock(head_mutex);
        NodeMutex* to_delete = head;
        head = head->next;
        delete to_delete;
        size--;
        return;
    }

    std::unique_lock<std::mutex> head_lock(head_mutex);

    int i = 1;
    NodeMutex* current = head;
    current->mutex.lock();
    head_lock.unlock();

    if (current->next){
        current->next->mutex.lock();
    }
    while (current->next){
        if (index == i){
            NodeMutex* to_delete = current->next;
            if (to_delete->next){
                to_delete->next->mutex.lock();
```

```
            }
            current->next = to_delete->next;
            if (current->next){
                current->next->mutex.unlock();
            }
            to_delete->mutex.unlock();
            current->mutex.unlock();
            size--;
            delete to_delete;
            return;
        }
        i++;
        NodeMutex* next_node = current->next;
        if (next_node->next){
            next_node->next->mutex.lock();
        }

        current->mutex.unlock();
        current = next_node;
    }
    current->mutex.unlock();
}

int FineGrained::getSize(){ return size; }

void FineGrained::printList(){
    std::unique_lock<std::mutex> head_lock(head_mutex);
    if (!head){
        std::cout << "empty" << std::endl;
        return;
    }
    NodeMutex* current = head;
    current->mutex.lock();
    head_lock.unlock();
    while (current)
    {
        std::cout << current->value << " ==> ";
        if (current->next){
            current->next->mutex.lock();
        }
        current->mutex.unlock();
        current = current->next;
    }
    std::cout << "end" << std::endl;

}
```

Название файла: LockFreeList.cpp

```
#include "LockFreeList.h"

std::pair<NodeLockFree*,  NodeLockFree*>  LockFreeList::find_nodes(int
index) {
    NodeLockFree* prev = nullptr;
    NodeLockFree* curr = head.load(std::memory_order_acquire);

    for (int i = 0; i < index && curr != nullptr; i++) {
```

```
        prev = curr;
        curr = curr->next.load(std::memory_order_acquire);
    }

    return {prev, curr};
}

void LockFreeList::insert(int value, int index) {
    if (index < 0 || index > size.load(std::memory_order_relaxed)) {
        return;
    }

    NodeLockFree* new_node = new NodeLockFree(value);

    if (index == 0) {
        new_node->next.store(head.load(std::memory_order_acquire),
std::memory_order_release);

        NodeLockFree* expected = head.load(std::memory_order_acquire);
        while (!head.compare_exchange_weak(expected, new_node,
                                           std::memory_order_release,
                                           std::memory_order_re-
laxed)) {
            new_node->next.store(expected, std::memory_order_release);
        }
    } else {
        while (true) {
            auto [prev, curr] = find_nodes(index);

            if (!prev) {
                delete new_node;
                continue;
            }

            new_node->next.store(curr, std::memory_order_release);

            NodeLockFree* expected = curr;
            if (prev->next.compare_exchange_weak(expected, new_node,
                                                 std::memory_order_re-
lease,
                                                 std::memory_order_re-
laxed)) {
                break;
            }
        }
    }

    size.fetch_add(1, std::memory_order_relaxed);
}

int LockFreeList::find(int index) {
    if (index < 0 || index >= size.load(std::memory_order_acquire)) {
        return -1;
    }

    auto [prev, curr] = find_nodes(index);

    if (curr) {
```

```
        return curr->value;
    }
    return -1;
}

void LockFreeList::del(int index) {
    if (index < 0 || index >= size.load(std::memory_order_acquire)) {
        return;
    }

    if (index == 0) {
        NodeLockFree* old_head = head.load(std::memory_order_acquire);
        while (old_head &&
                !head.compare_exchange_weak(old_head,

old_head->next.load(std::memory_order_acquire),
                                            std::memory_order_release,
                                            std::memory_order_re-
laxed)) {
        }

        if (old_head) {
            delete old_head;
            size.fetch_sub(1, std::memory_order_relaxed);
            return;
        }
    } else {
        while (true) {
            auto [prev, curr] = find_nodes(index);

            if (!prev || !curr) {
                return;
            }

            NodeLockFree*  next_node  =  curr->next.load(std::memory_or-
der_acquire);

            NodeLockFree* expected = curr;
            if (prev->next.compare_exchange_weak(expected, next_node,
                                                 std::memory_order_re-
lease,
                                                 std::memory_order_re-
laxed)) {
                delete curr;
                size.fetch_sub(1, std::memory_order_relaxed);
                return;
            }
        }
    }

    return;
}

int LockFreeList::getSize() {
    return size.load(std::memory_order_acquire);
}

void LockFreeList::printList() {
```

```
    NodeLockFree* current = head.load(std::memory_order_acquire);
    while (current != nullptr) {
        std::cout << current->value;
        if (current->next.load(std::memory_order_acquire) != nullptr)
{
            std::cout << " ==> ";
        }
        current = current->next.load(std::memory_order_acquire);
    }
    std::cout << " ==> end" << std::endl;
}

LockFreeList::~LockFreeList(){
    NodeLockFree* current = head.load(std::memory_order_relaxed);
    while (current != nullptr) {
        NodeLockFree* next = current->next.load(std::memory_order_re-
laxed);
        delete current;
        current = next;
    }
}
```