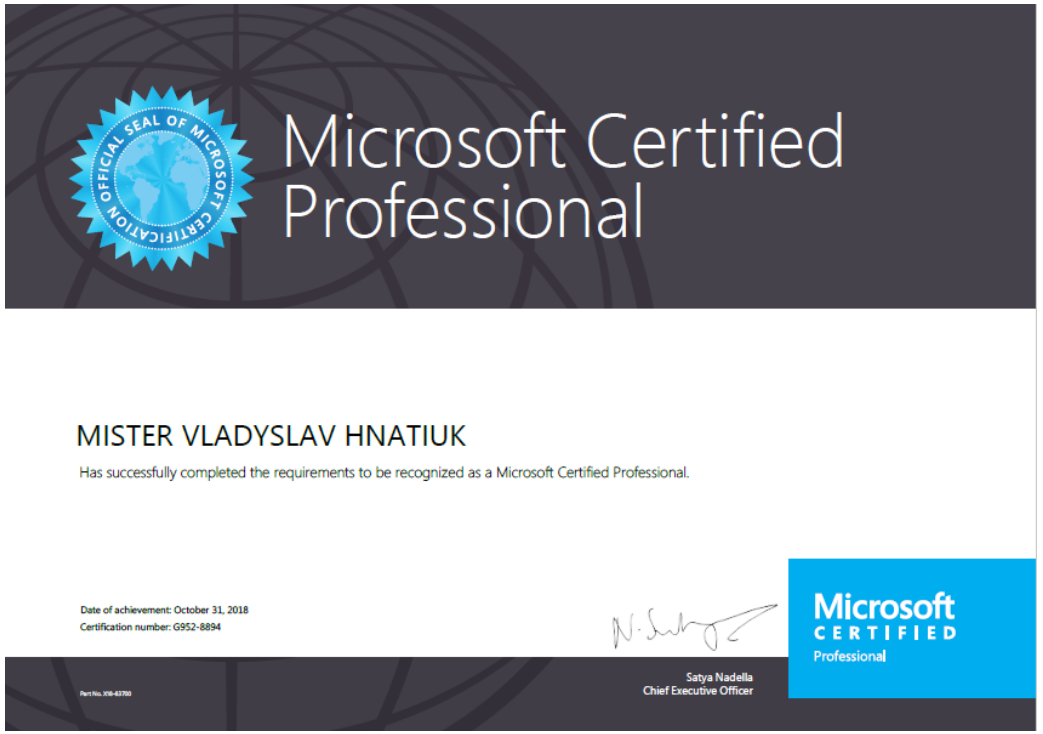
C# Асинхронное программирование

ConcurrentDictionary. PLINQ

C# Асинхронное программирование

Автор курса

Гнатюк Владислав



MCID:16354168

Асинхронное программирование

После урока обязательно

Повторите этот урок в видео формате на ITVDN.com

Проверьте как Вы усвоили данный материал на TestProvider.com

ConcurrentDictionary. PLINQ

План урока

1) Потокобезопасный словарь ConcurrentDictionary
2) Описание API ConcurrentDictionary
3) Особенности работы с ConcurrentDictionary
4) Параллельная обработка. PLINQ

ConcurrentDictionary

ConcurrentDictionary<TKey, TValue> - потокобезопасная коллекция, работающая по принципу «ключ-значение», доступ к которой могут одновременно получать несколько потоков.

Работа с элементами в ConcurrentDictionary

Для работы с данными в ConcurrencyDictionary есть две группы методов:

1. Методы шаблона **TryXXX**:
   - TryAdd – добавление элемента
   - TryGetValue – получение значения элемента
   - TryRemove – удаление элемента
   - TryUpdate – обновление значения элемента

2. Методы **«Всегда-выполняемые»**:
   - AddOrUpdate – изменение значения элемента или его добавление
   - GetOrAdd – получение значения элемента или его добавление

Методы шаблона TryXXX

Методы шаблона **TryXXX** – выполняются без генерации исключений. Для отображения корректности своего выполнения, методы шаблона TryXXX возвращают значение true, если у них получилось выполниться, false – если результат выполнения неудача:

- bool TryAdd(TKey key, TValue value) – метод пытается добавить элемент в коллекцию по ключу.
- bool TryGetValue(TKey key, out TValue value) – метод пытается получить значение через out параметр.
- bool TryRemove(TKey key, out TValue value) – метод пытается удалить элемент по ключу, значение удаленного элемента будет помещено в out параметр.
- bool TryUpdate(TKey key, TValue newValue, TValue comparisonValue) - метод пытается обновить значение элемента по ключу (первый параметр). Второй параметр – это новое значение. Третий параметр – значение для сравнения с элементом, который находится внутри словаря.

AddOrUpdate

Метод AddOrUpdate – используется для добавления или изменения значения в коллекции. Если ключ существует, то значение будет изменено, если нет – будет добавлено по ключу указанное значение. Метод возвращает новое значение, которое теперь находится по указанному ключу.

Для обновления элемента необходимо передать функцию, которая вычислит и вернет новое значение.

```
public TValue AddOrUpdate(TKey key, TValue addValue, Func<TKey, TValue, TValue> updateValueFactory)
```

Если для добавления нового значения требуются вычисления – есть специальные перегрузки, которые позволяют передать функцию. Также, если требуется, можно передать дополнительное значение.

```
public TValue AddOrUpdate(TKey key,
                          Func<TKey, TValue> addValueFactory,
                          Func<TKey, TValue, TValue> updateValueFactory)

public TValue AddOrUpdate(TKey key,
                          Func<TKey, TArg, TValue> addValueFactory,
                          Func<TKey, TValue, TArg, TValue> updateValueFactory,
                          TArg factoryArgument)
```

GetOrAdd

Метод GetOrAdd – используется для получения значения из коллекции по ключу. Но если такого ключа там не будет, метод добавит новое указанное значение, вернув его из метода.

```
public TValue GetOrAdd(TKey key, TValue value)
```

Существуют дополнительные перегрузки, когда для добавления нового значения требуется его вычислять с помощью функции.

```
public TValue GetOrAdd(TKey key, Func<TKey, TValue> valueFactory)
```

Функция может быть достаточно сложной, требуя дополнительного входящего параметра. Для этого существует еще одна дополнительная перегрузка.

```
public TValue GetOrAdd<TArg>(TKey key, Func<TKey, TArg, TValue> valueFactory, TArg factoryArgument)
```

Рекомендации по использованию «всегда-выполняемых» методов

Методы AddOrUpdate/GetOrAdd имеют в качестве параметров делегаты:

- Когда вы передаете метод или лямбда выражение в параметр на делегат, то тело метода или лямбда выражения должно быть как можно проще и короче. Ведь вы должны понимать, что там может произойти блокировка и чем дольше будет выполняться ваш делегат, тем дольше могут ждать другие потоки.

- Ни в коем случае методы или лямбда выражения, передаваемые в параметр делегата, не должны выбрасывать исключения. Потому что «всегда-выполняемые» методы выбросят исключение вам в точку их вызова.

Внутреннее устройство ConcurrentDictionary

Хранилище ConcurrentDictionary состоит из так называемых ведер (buckets). Каждый bucket в ConcurrentDictionary представлен экземпляром класса Node.

Класс Node представляет собой однонаправленный связный список. Он хранит в себе: ключ, значение, хэш-код ключа и ссылку на следующий экземпляр класса Node.

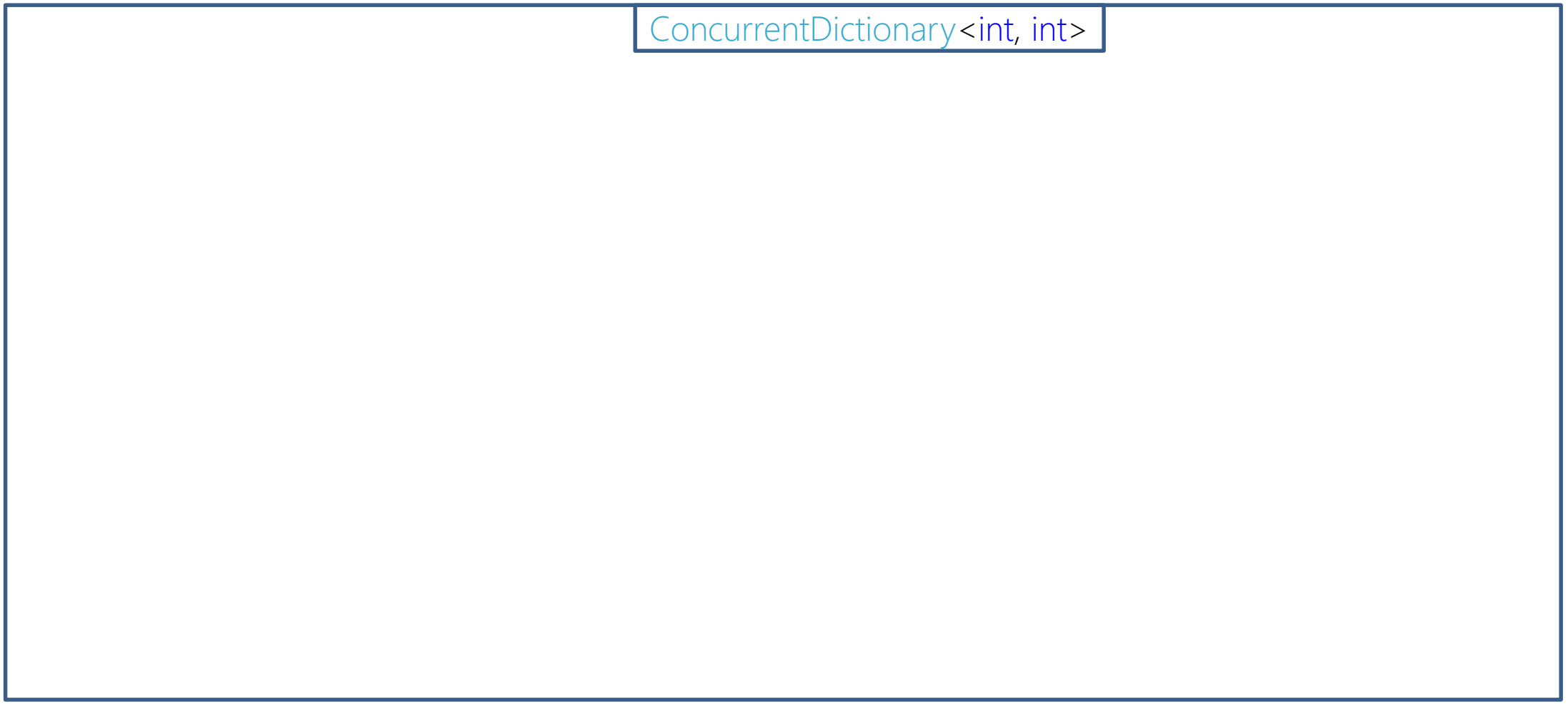
```
public class ConcurrentDictionary<TKey, TValue>
{
        Tables m_tables;

        private class Tables
        {
                Node[] m_buckets;
        }

        private class Node
        {
                TKey m_key;
                TValue m_value;
                Node m_next;
                int m_hashcode;
        }
}
```

Группировка элементов по bucket-ам и нахождение объекта блокировки

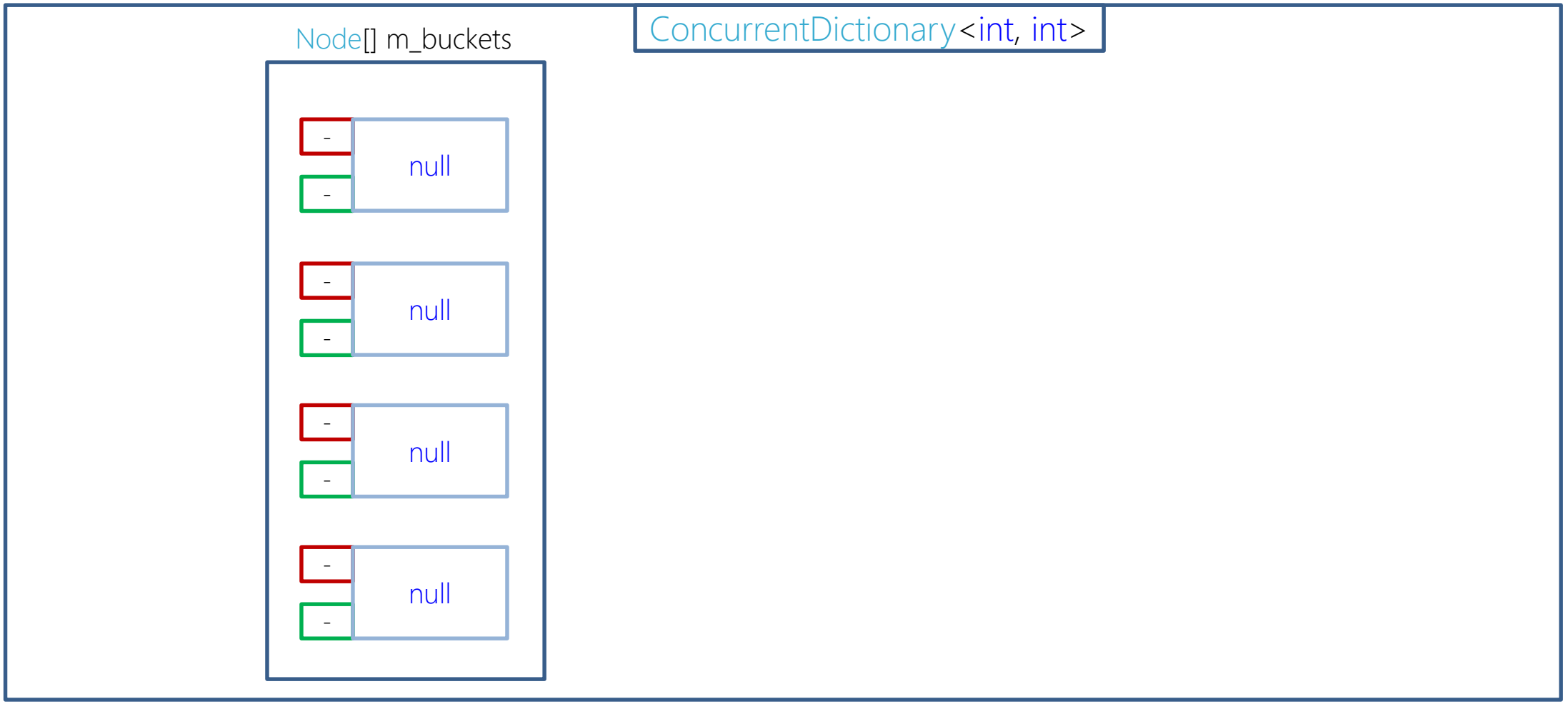
Элемент добавляется в ConcurrentDictionary на основе хэш-кода передаваемого ключа. Но, все элементы группируются в один bucket по специальному алгоритму метода GetBucketAndLockNo. Этот же метод указывает какой объект блокировки будет использоваться для работы с элементами группы.

```
private void GetBucketAndLockNo(int hashcode, out int bucketNo, out int lockNo, int bucketCount, int lockCount)
{
        bucketNo = (hashcode & int.MaxValue) % bucketCount;
        lockNo = bucketNo % lockCount;
}
```

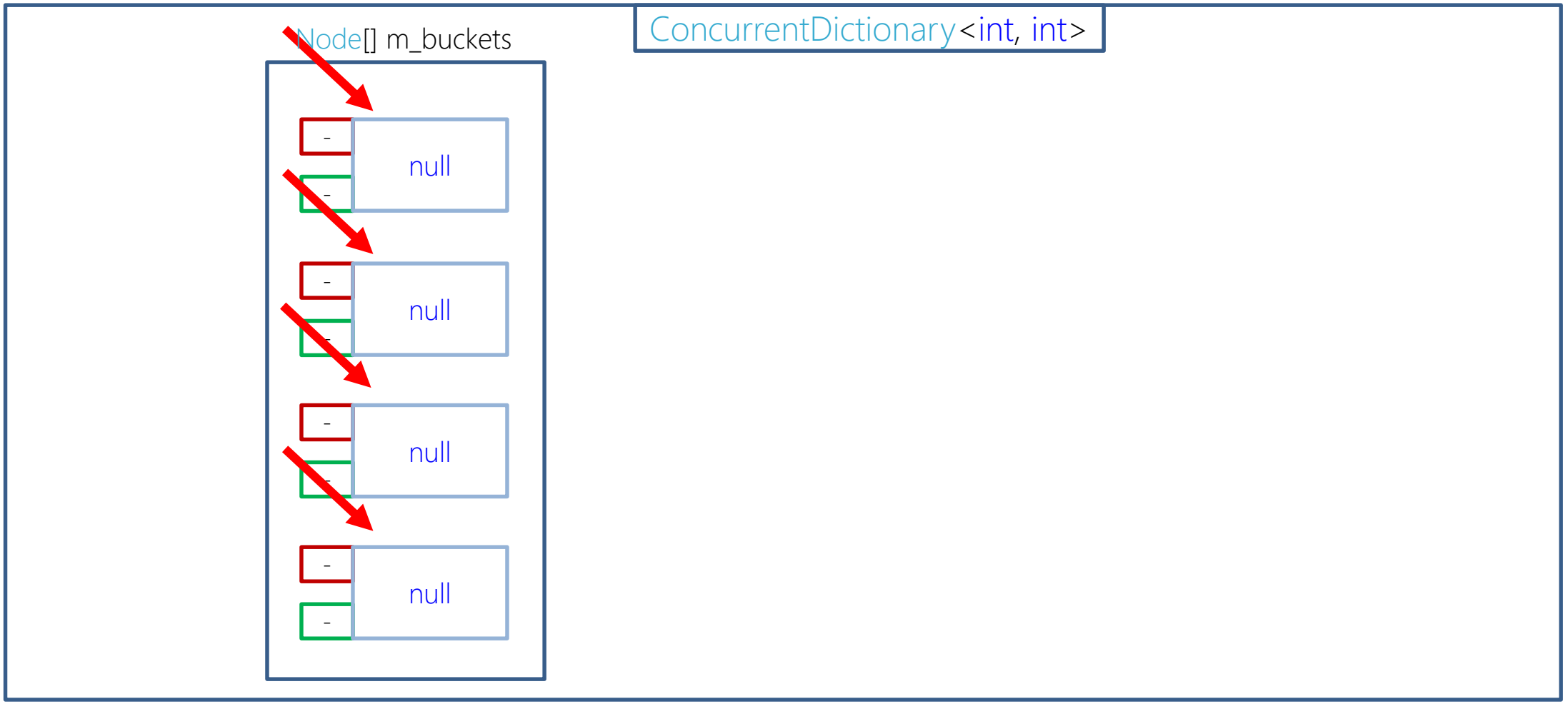
Хранение элементов в ConcurrentDictionary



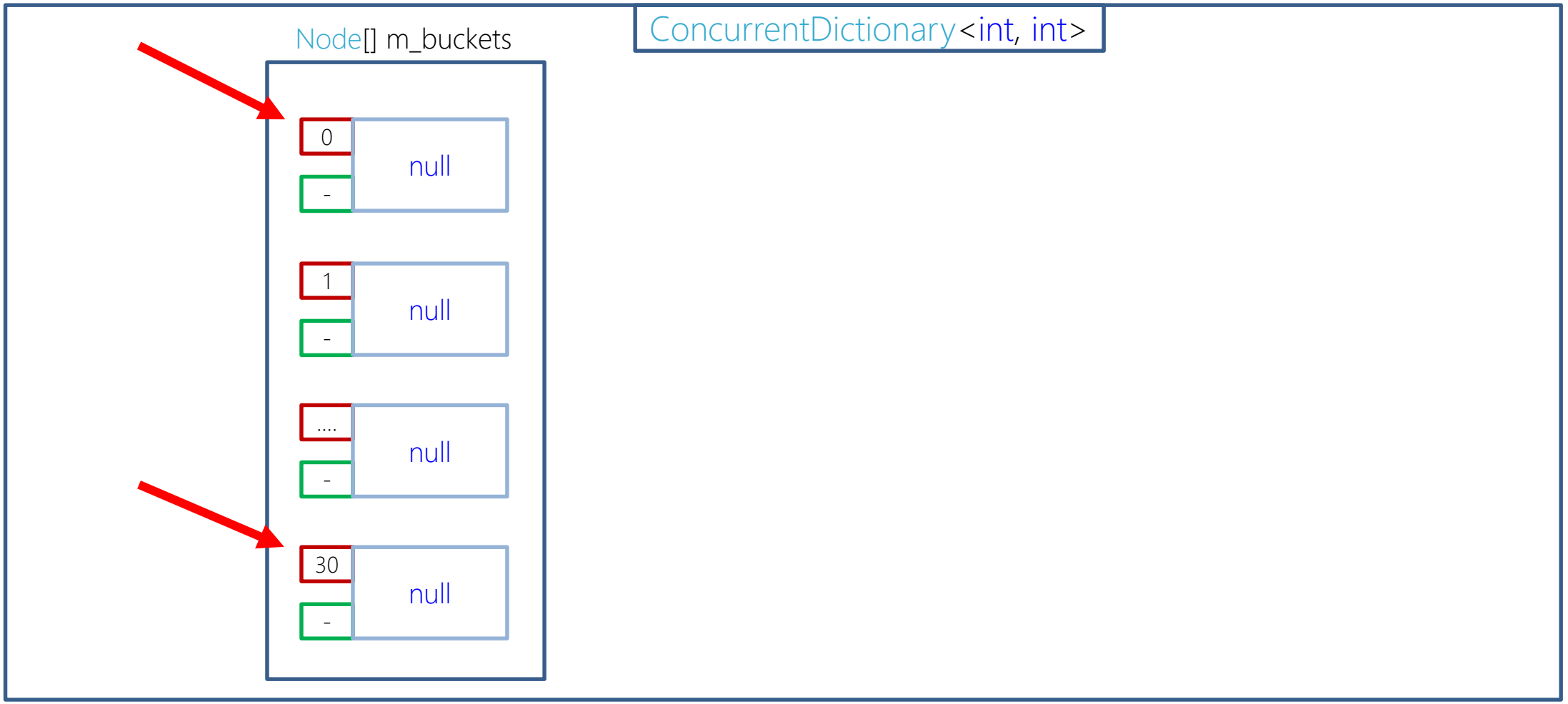
Хранение элементов в ConcurrentDictionary



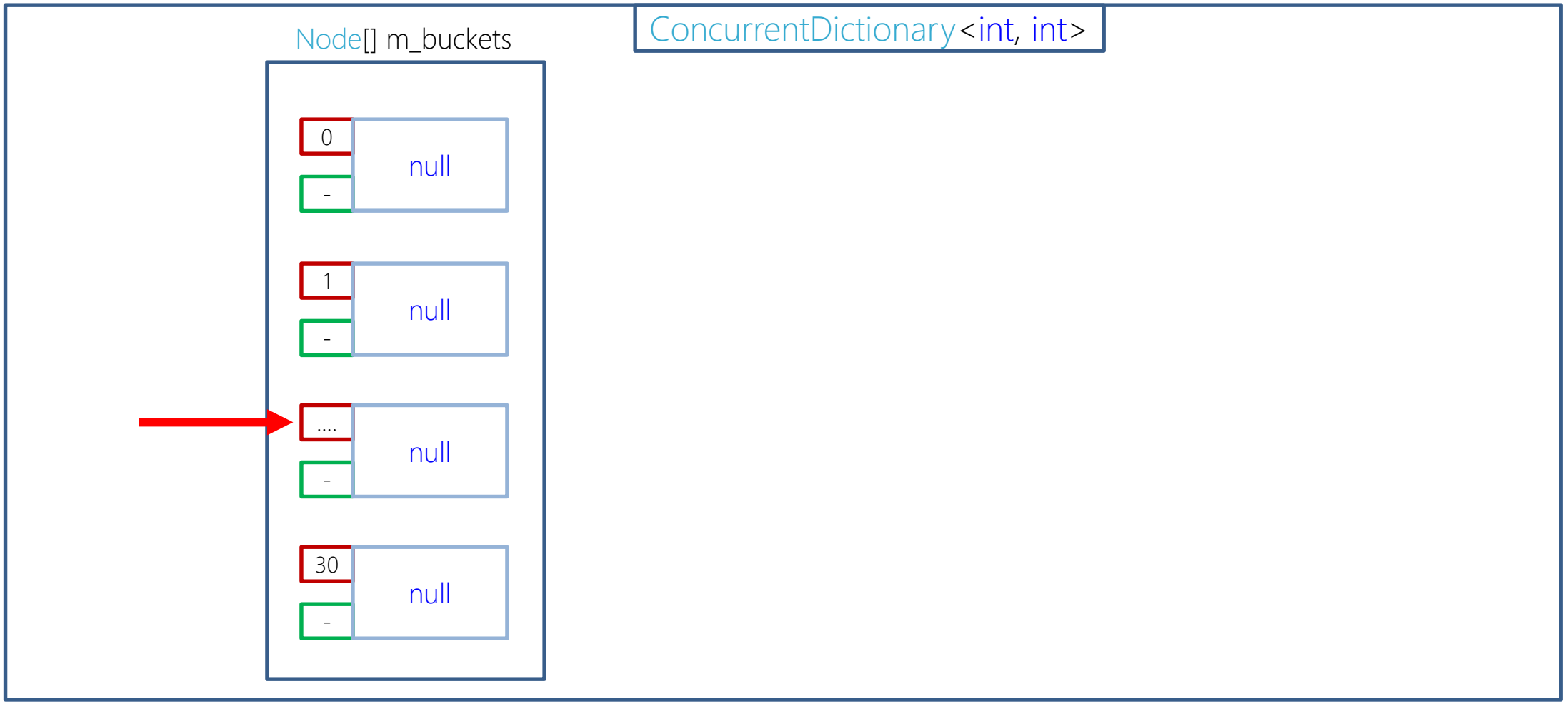
Хранение элементов в ConcurrentDictionary



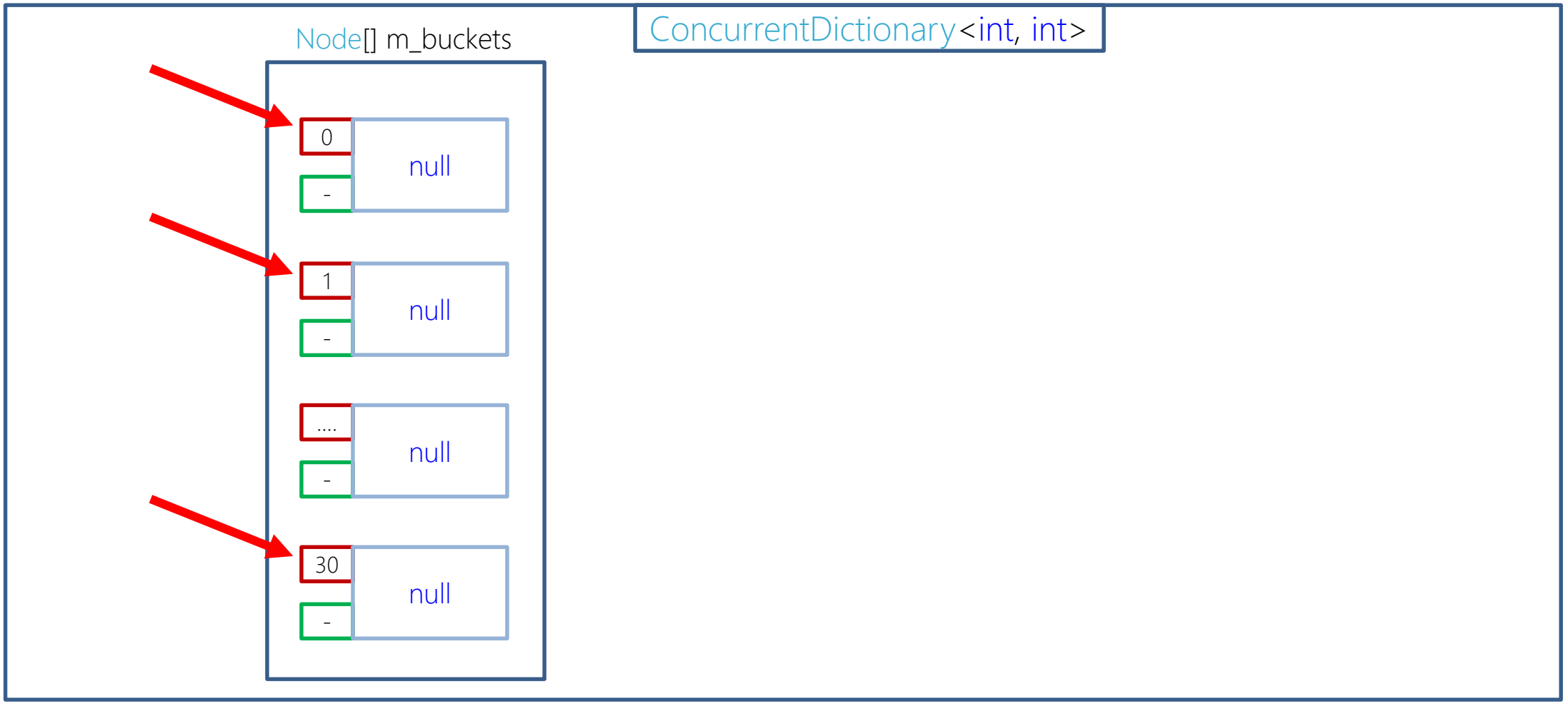
Хранение элементов в ConcurrentDictionary



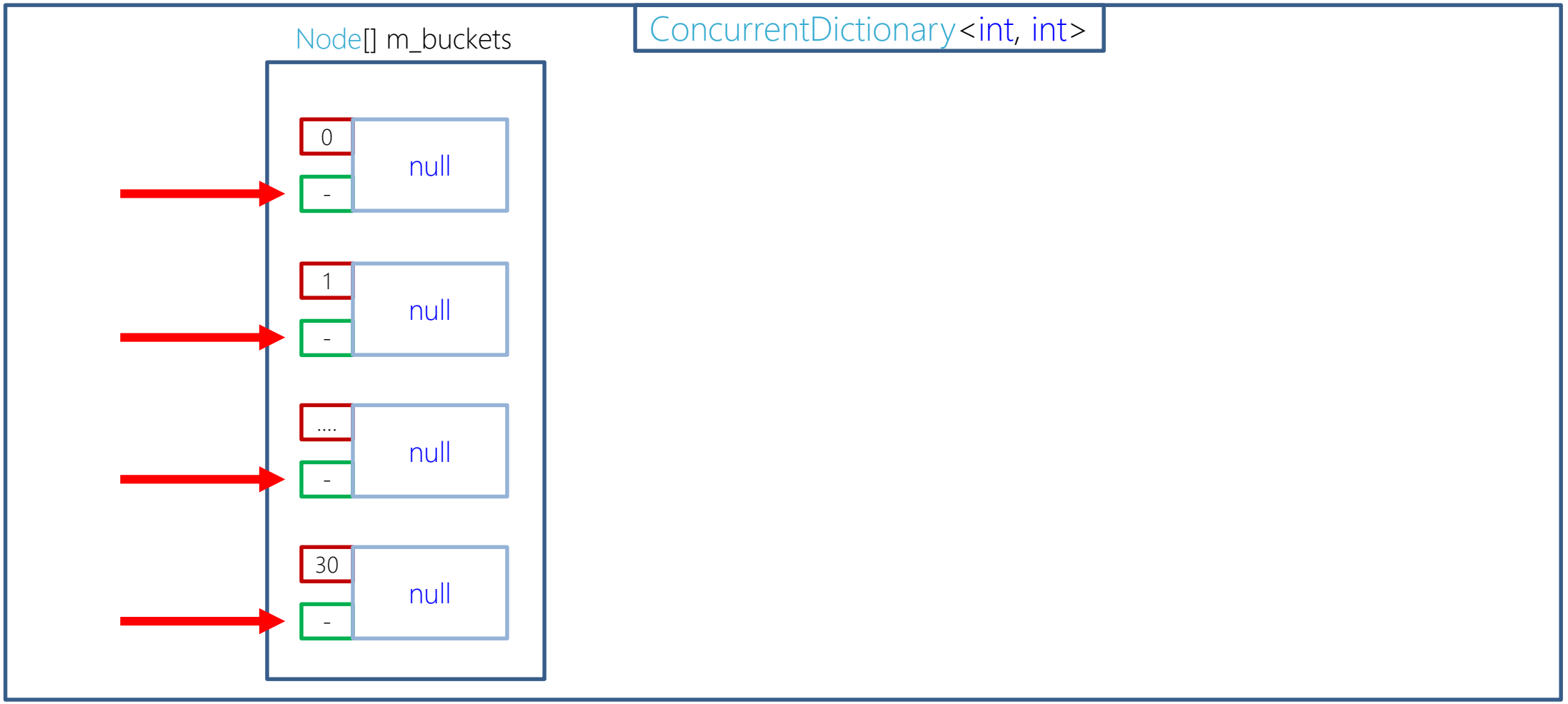
Хранение элементов в ConcurrentDictionary



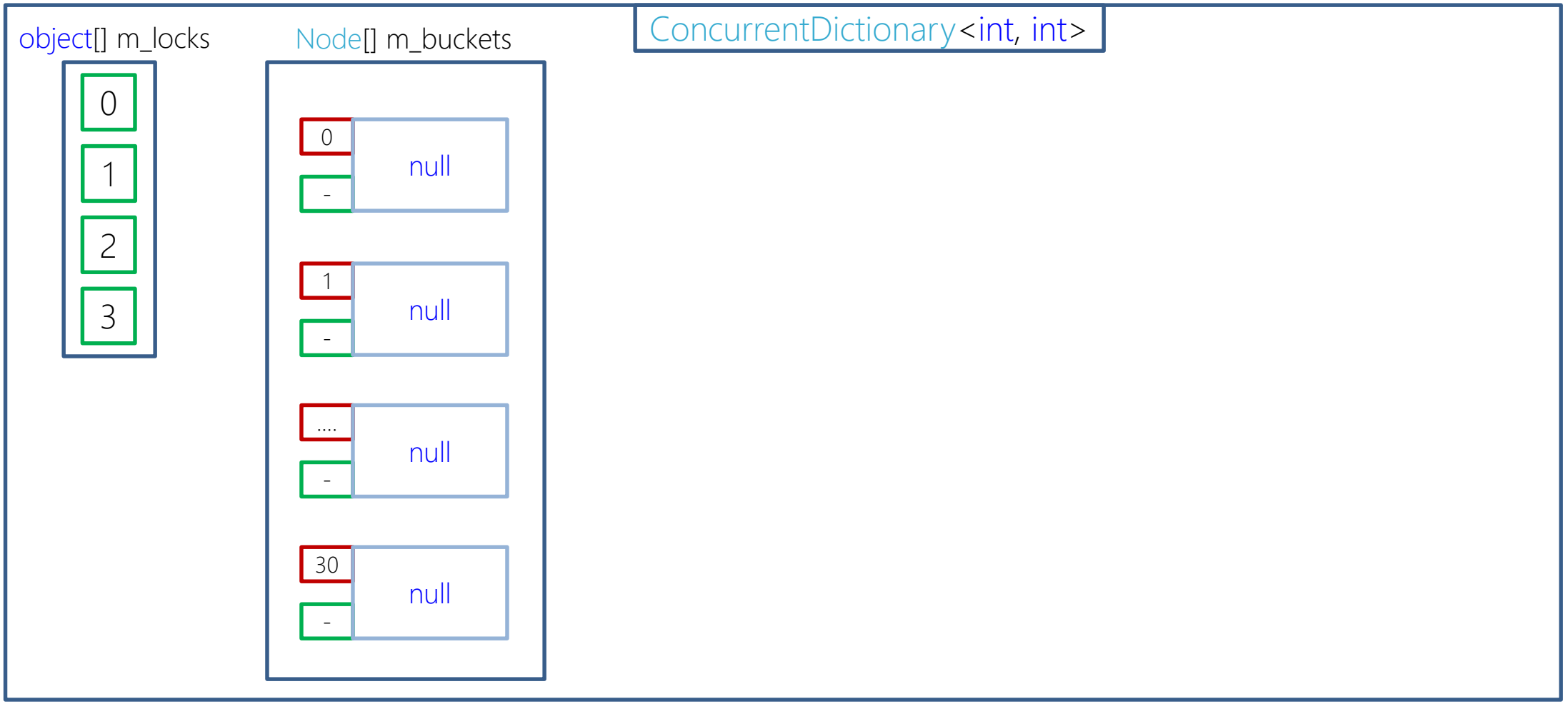
Хранение элементов в ConcurrentDictionary



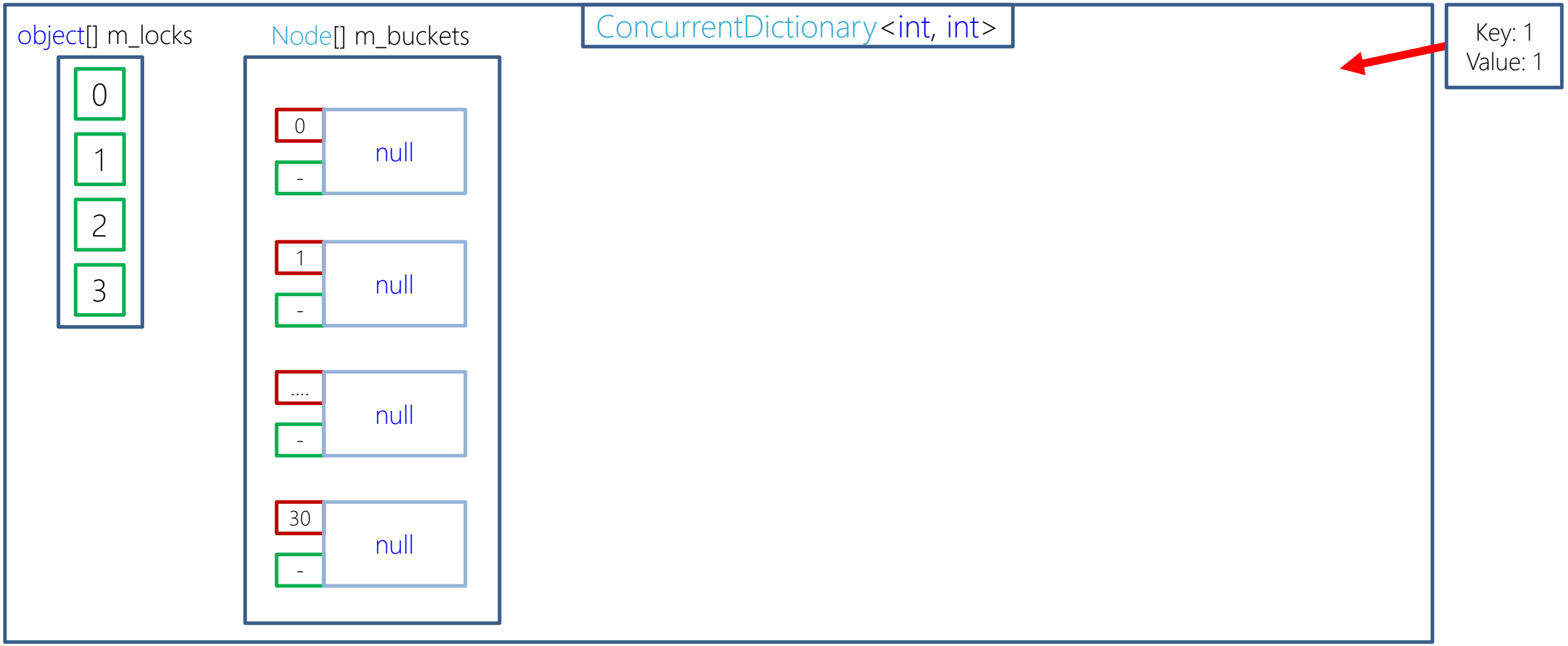
Хранение элементов в ConcurrentDictionary



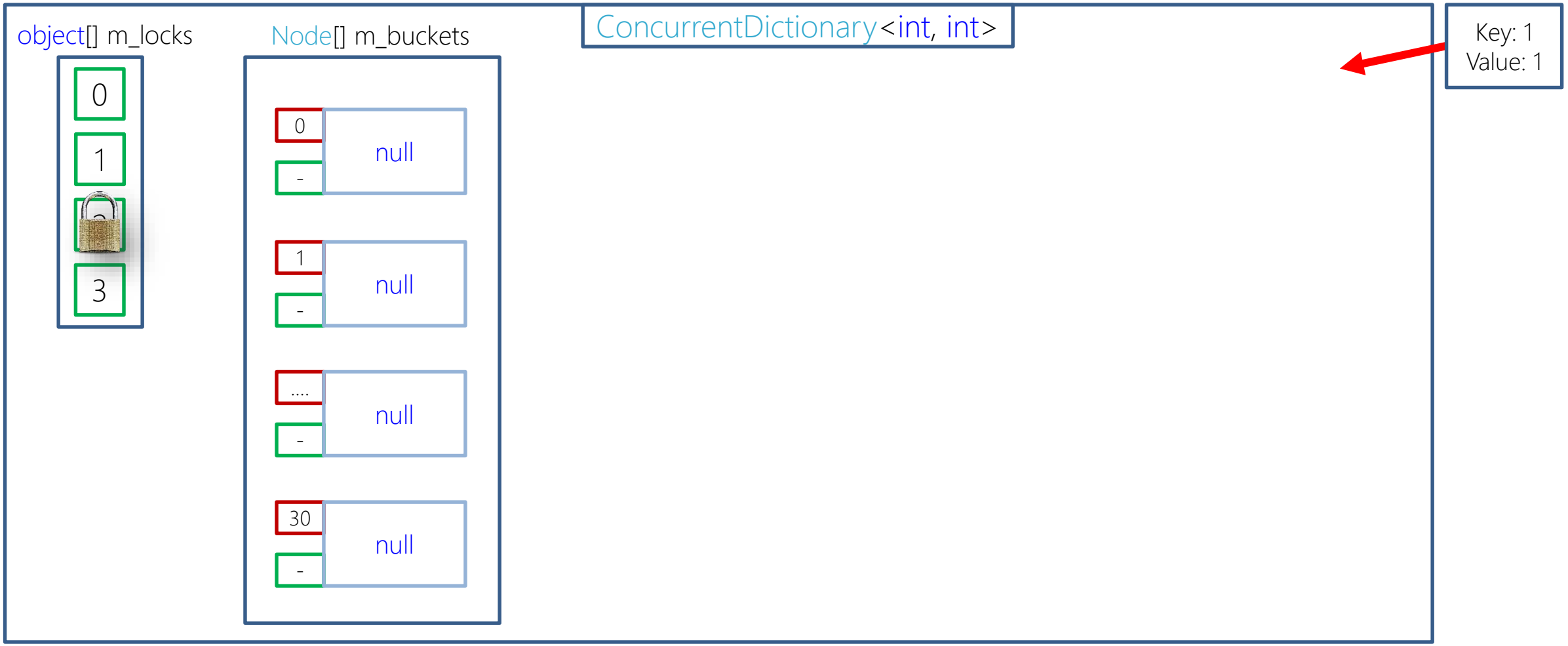
Хранение элементов в ConcurrentDictionary



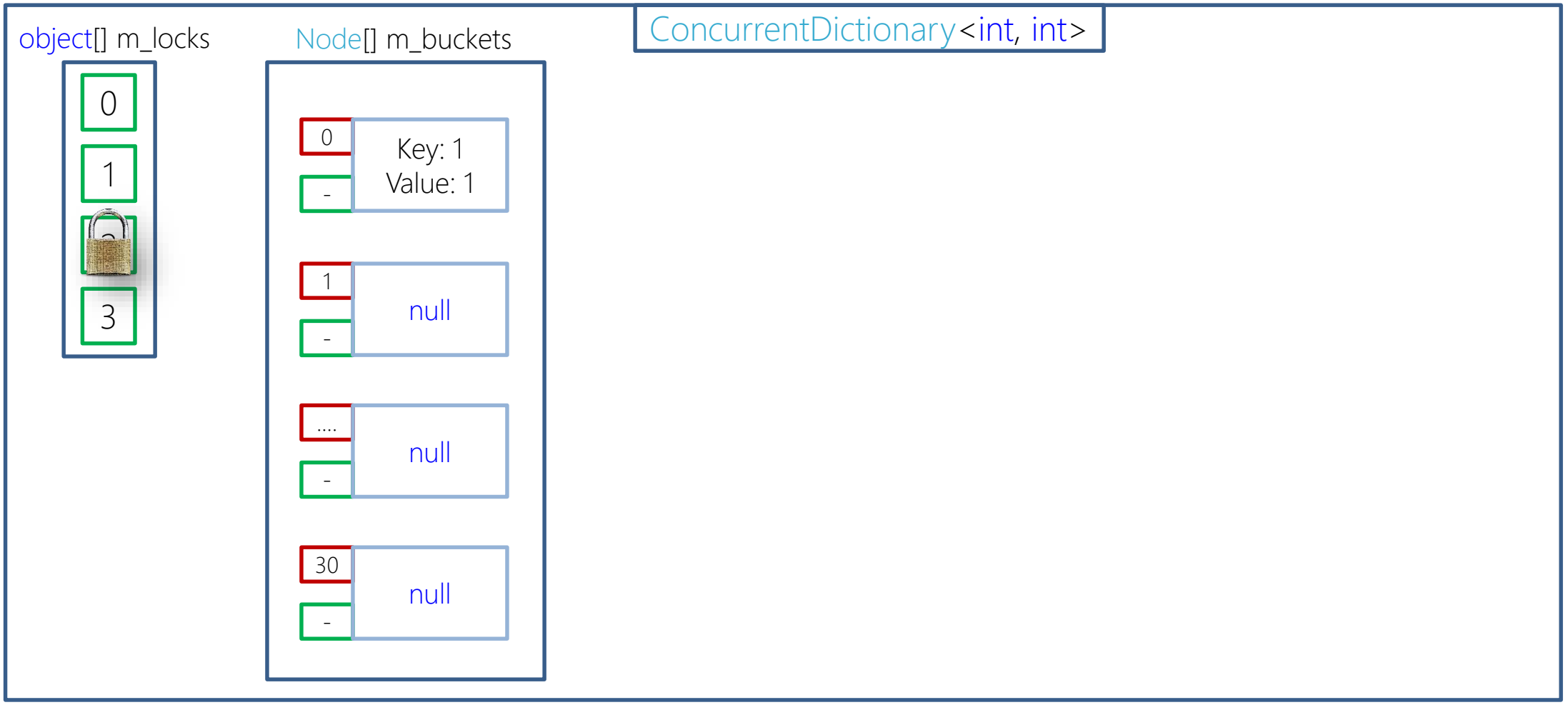
Хранение элементов в ConcurrentDictionary



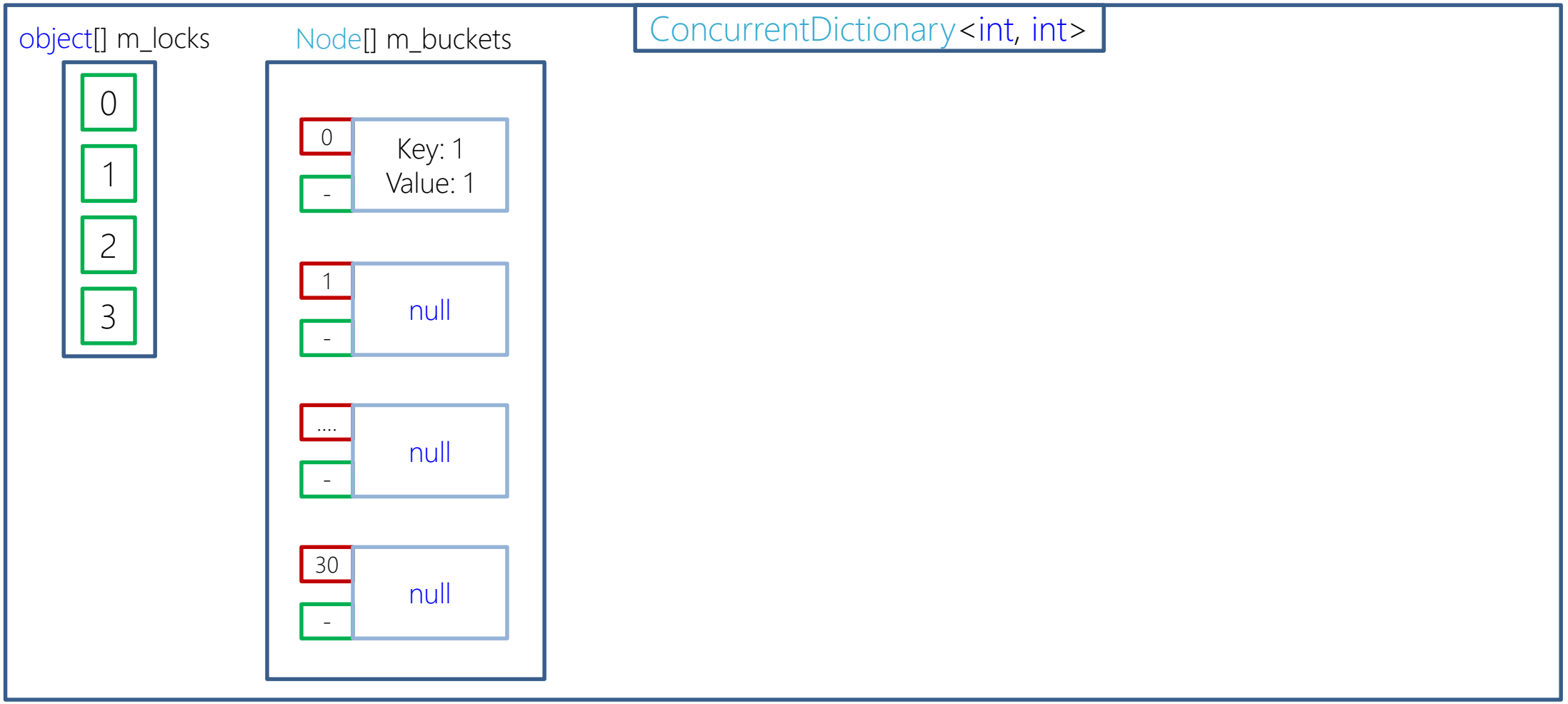
Хранение элементов в ConcurrentDictionary



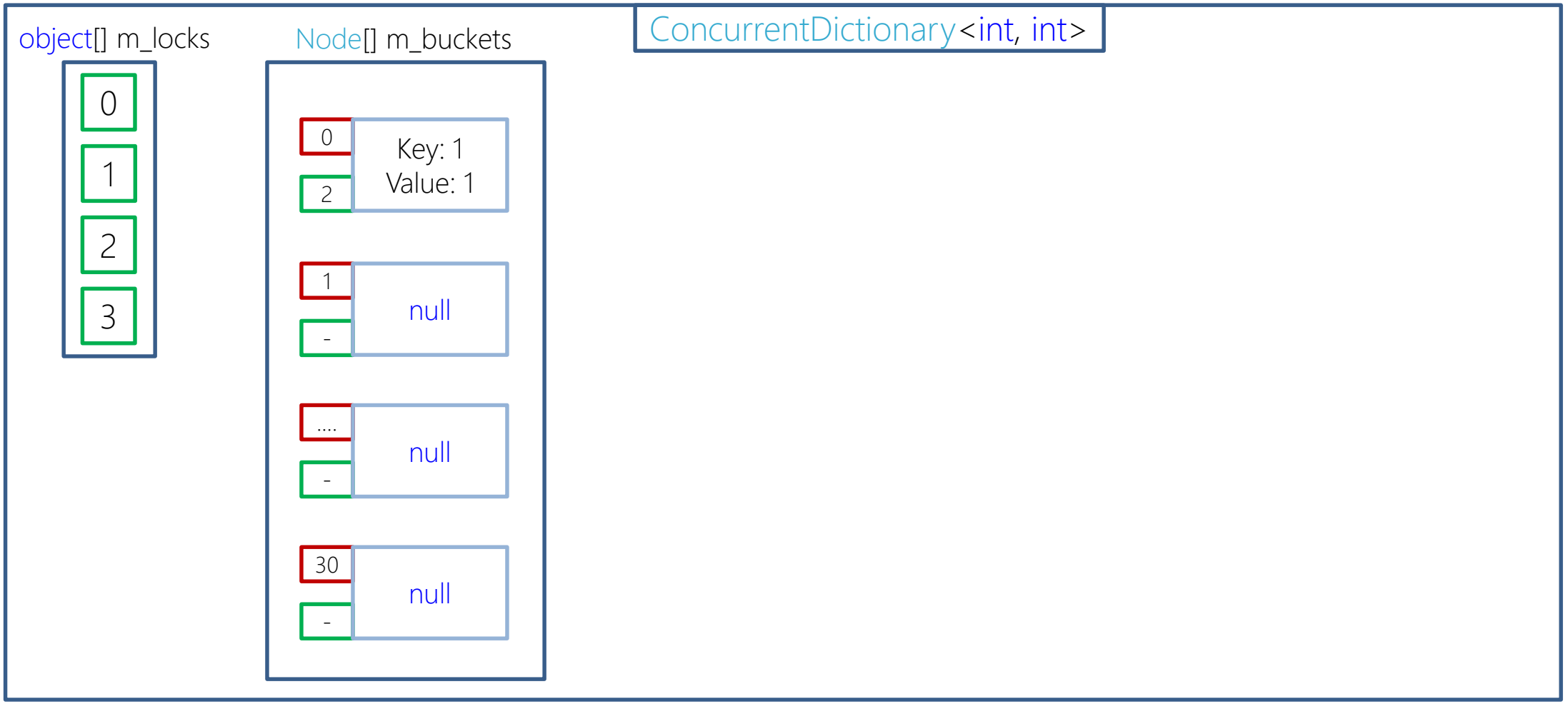
Хранение элементов в ConcurrentDictionary



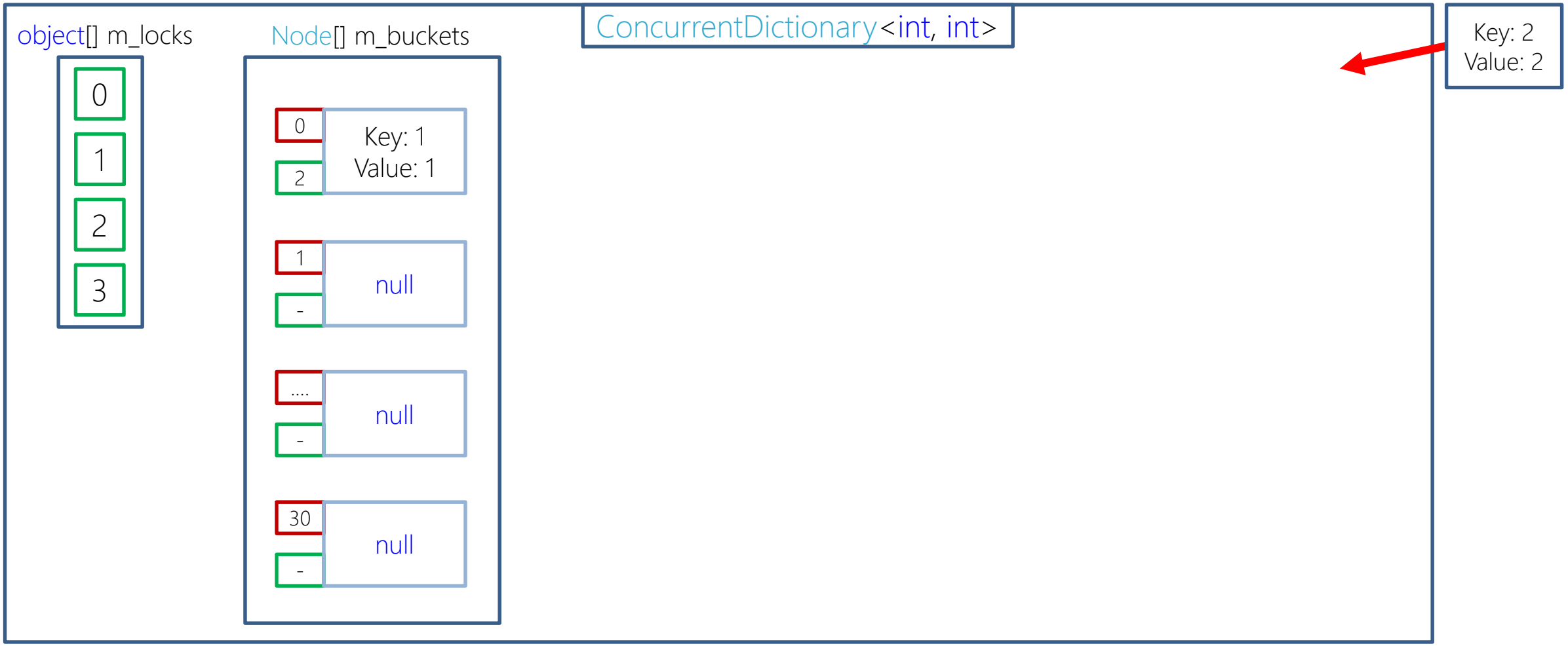
Хранение элементов в ConcurrentDictionary



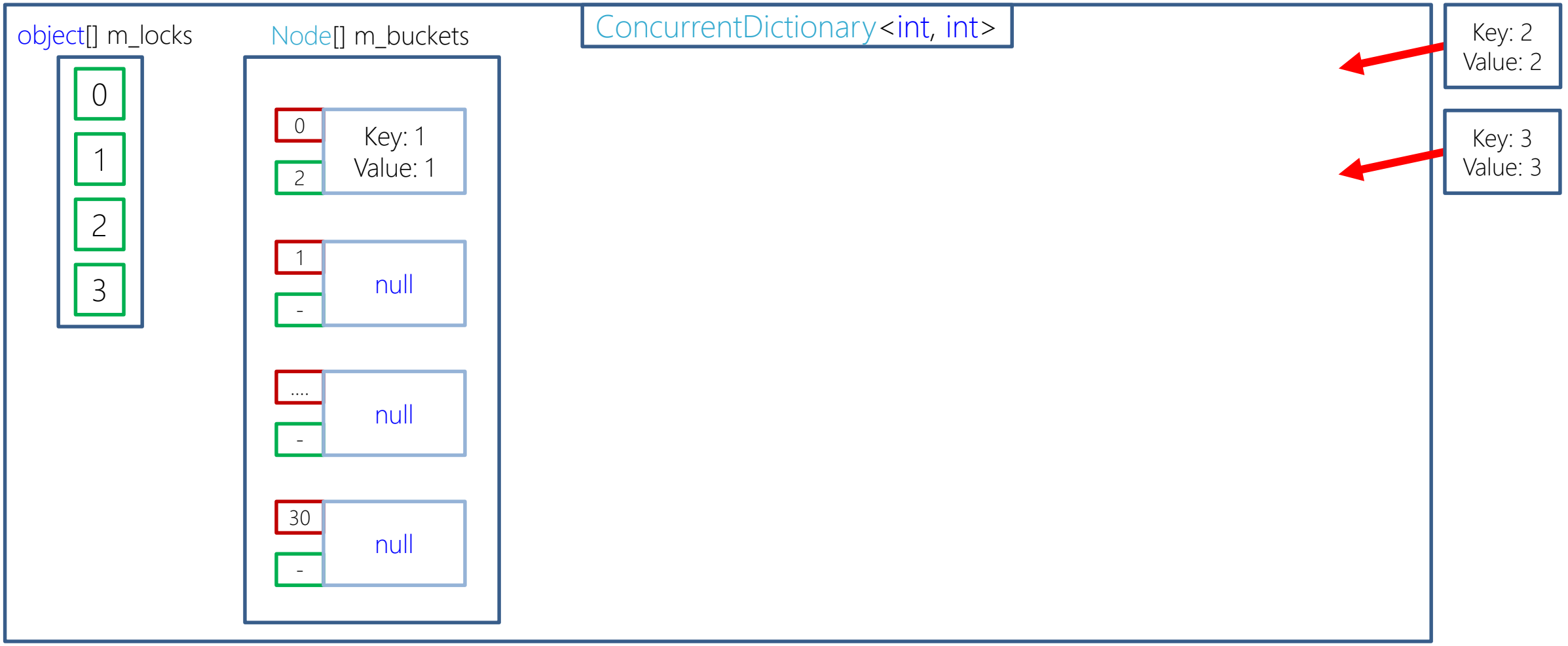
Хранение элементов в ConcurrentDictionary



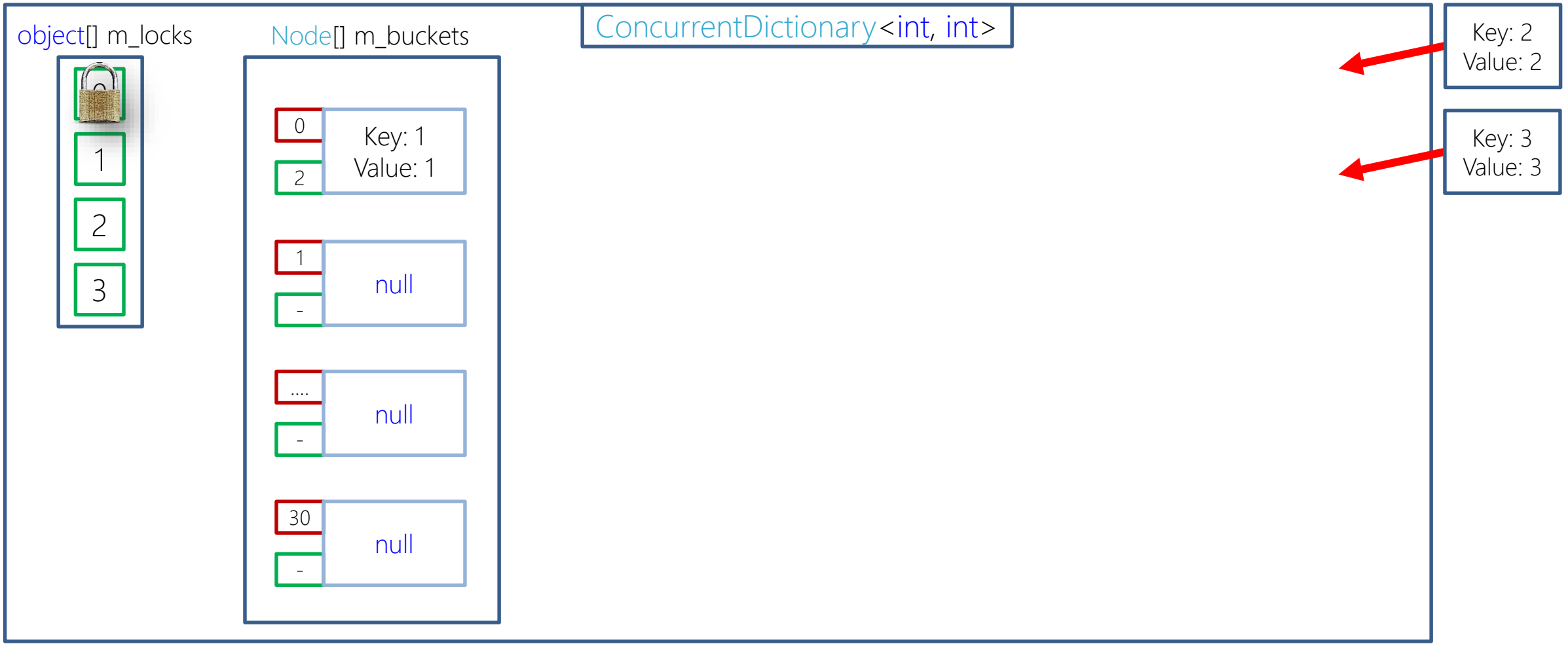
Хранение элементов в ConcurrentDictionary



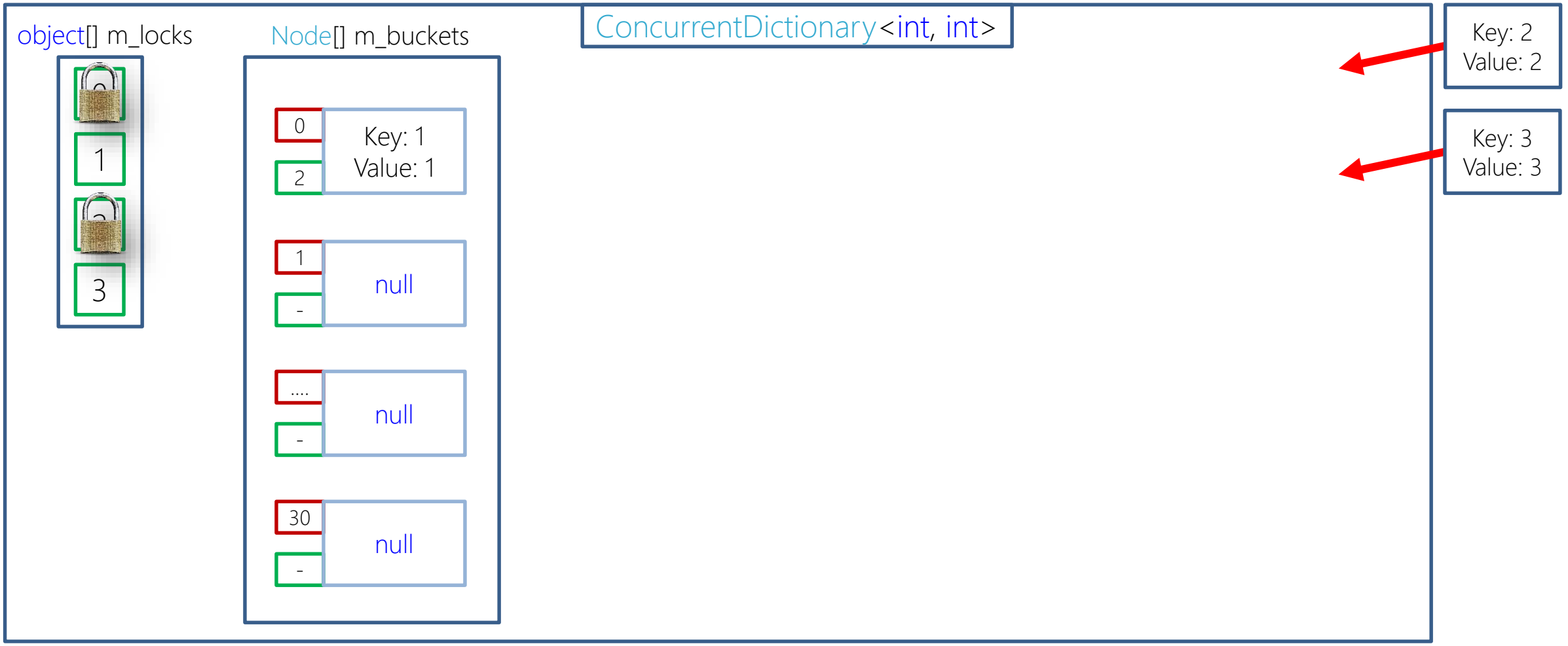
Хранение элементов в ConcurrentDictionary



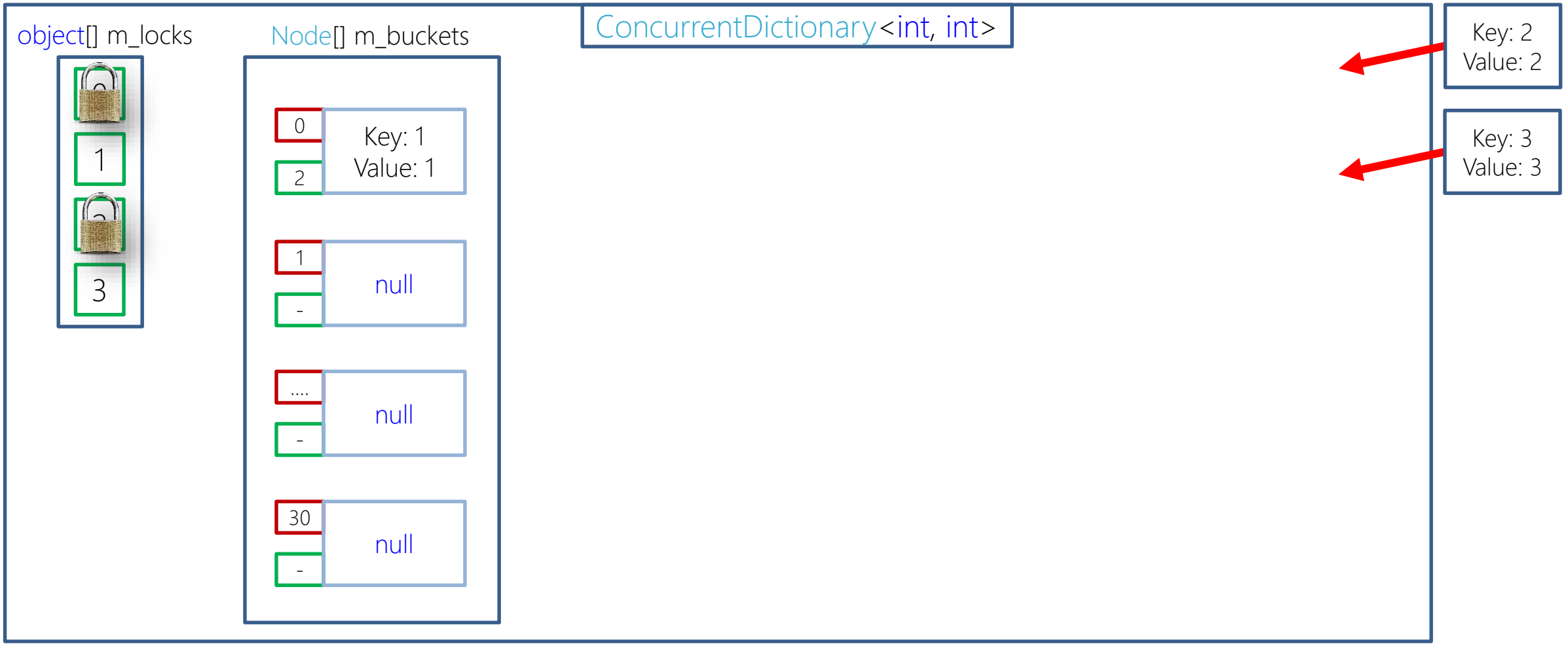
Хранение элементов в ConcurrentDictionary



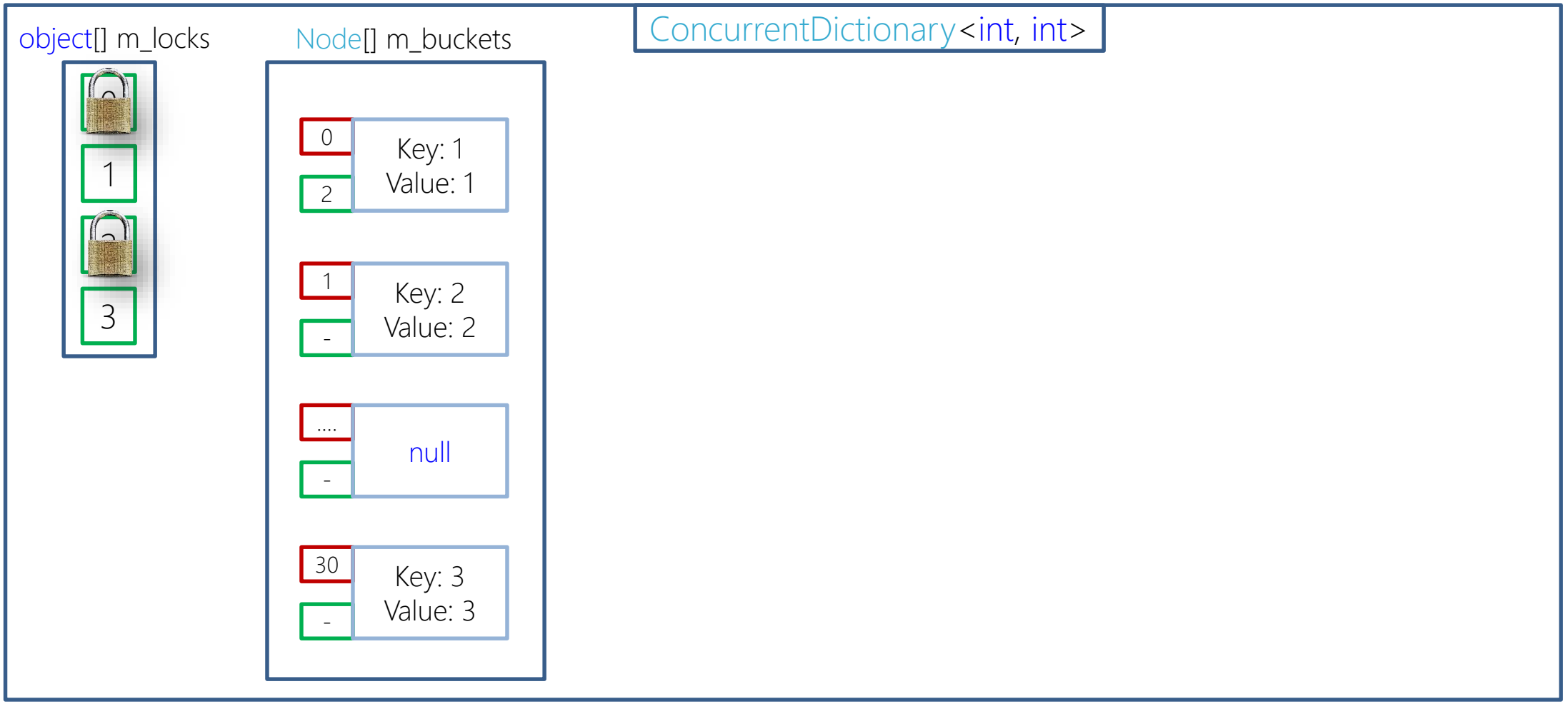
Хранение элементов в ConcurrentDictionary



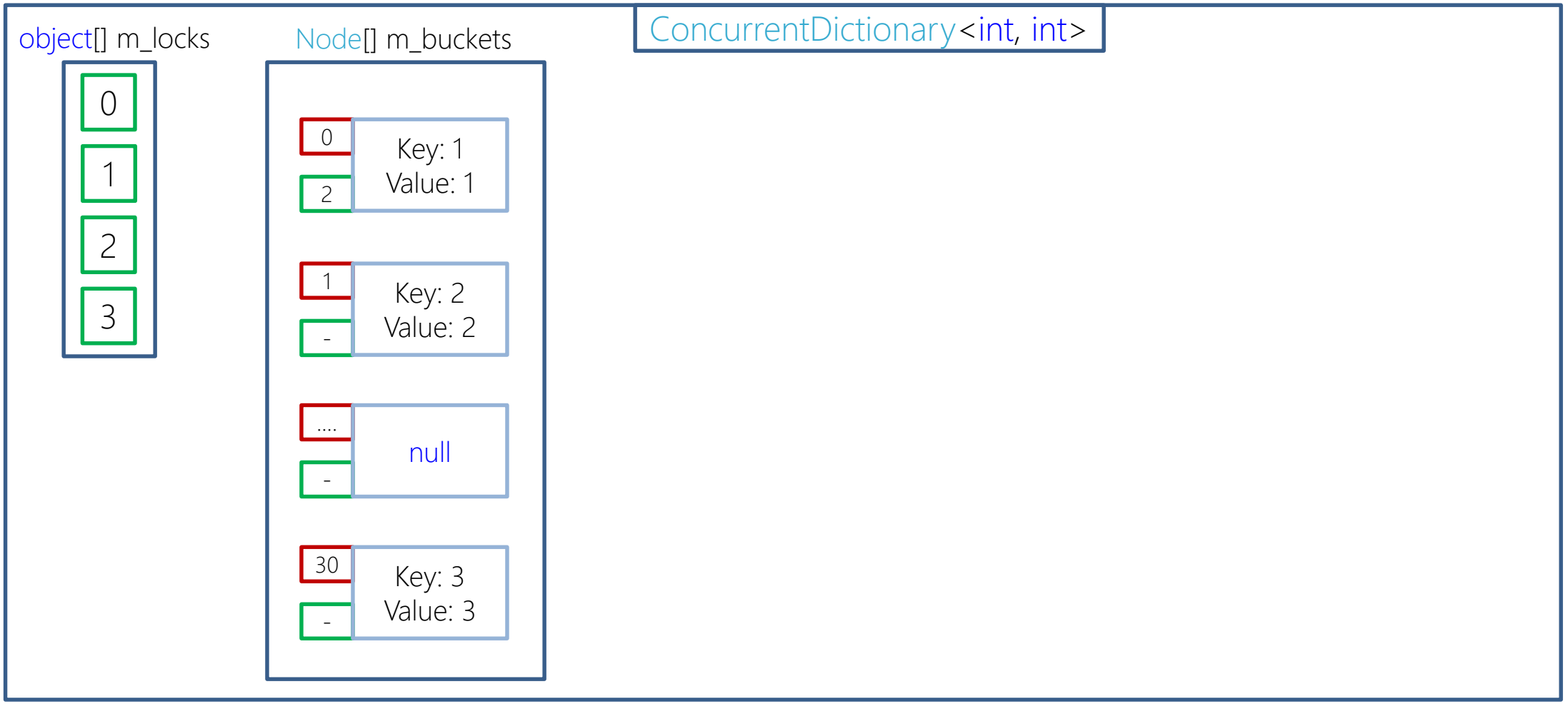
Хранение элементов в ConcurrentDictionary



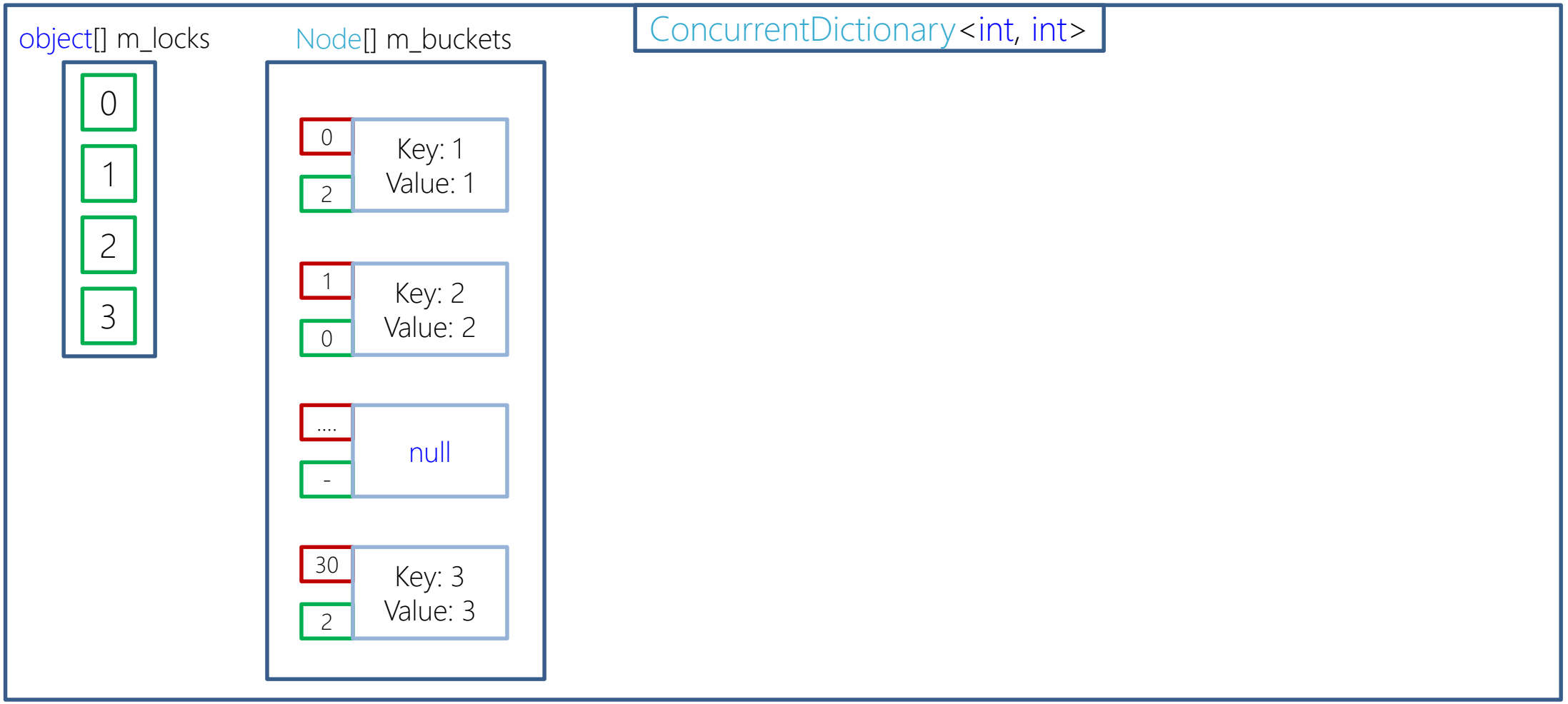
Хранение элементов в ConcurrentDictionary



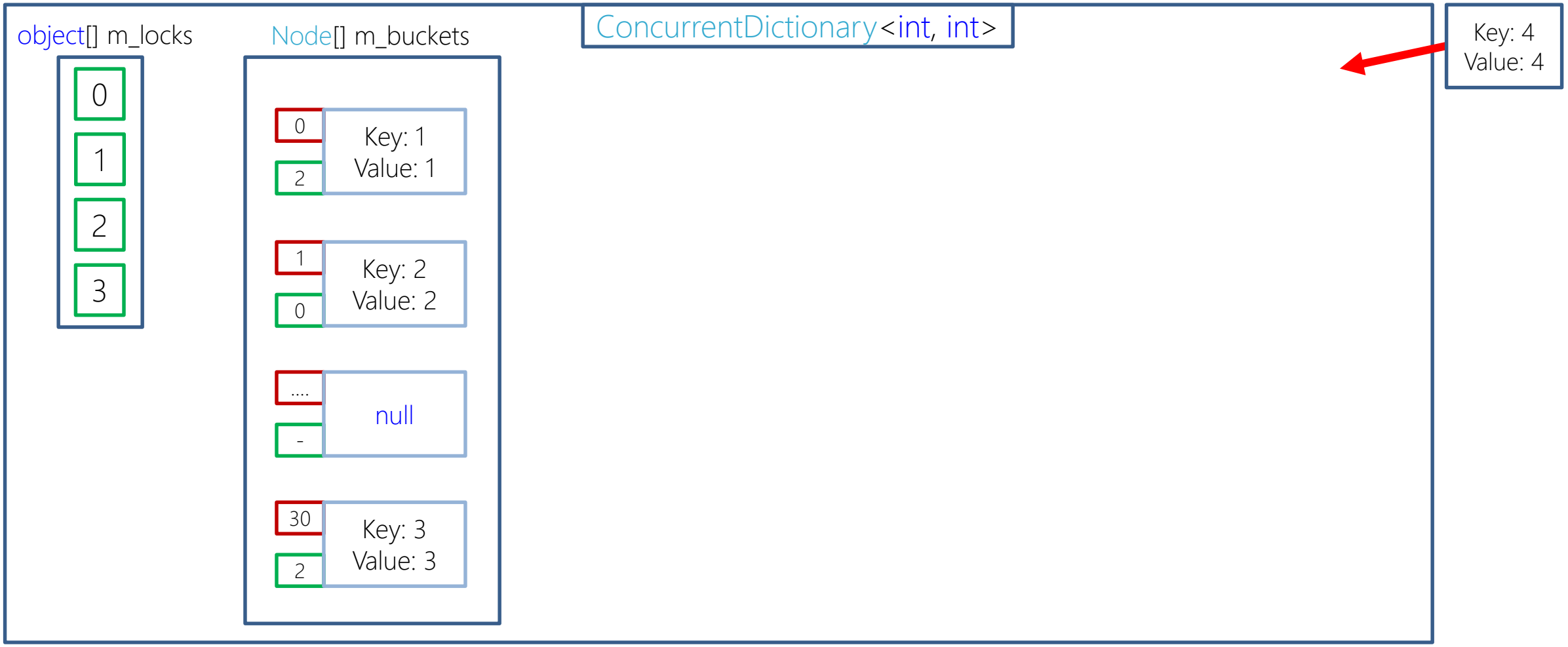
Хранение элементов в ConcurrentDictionary



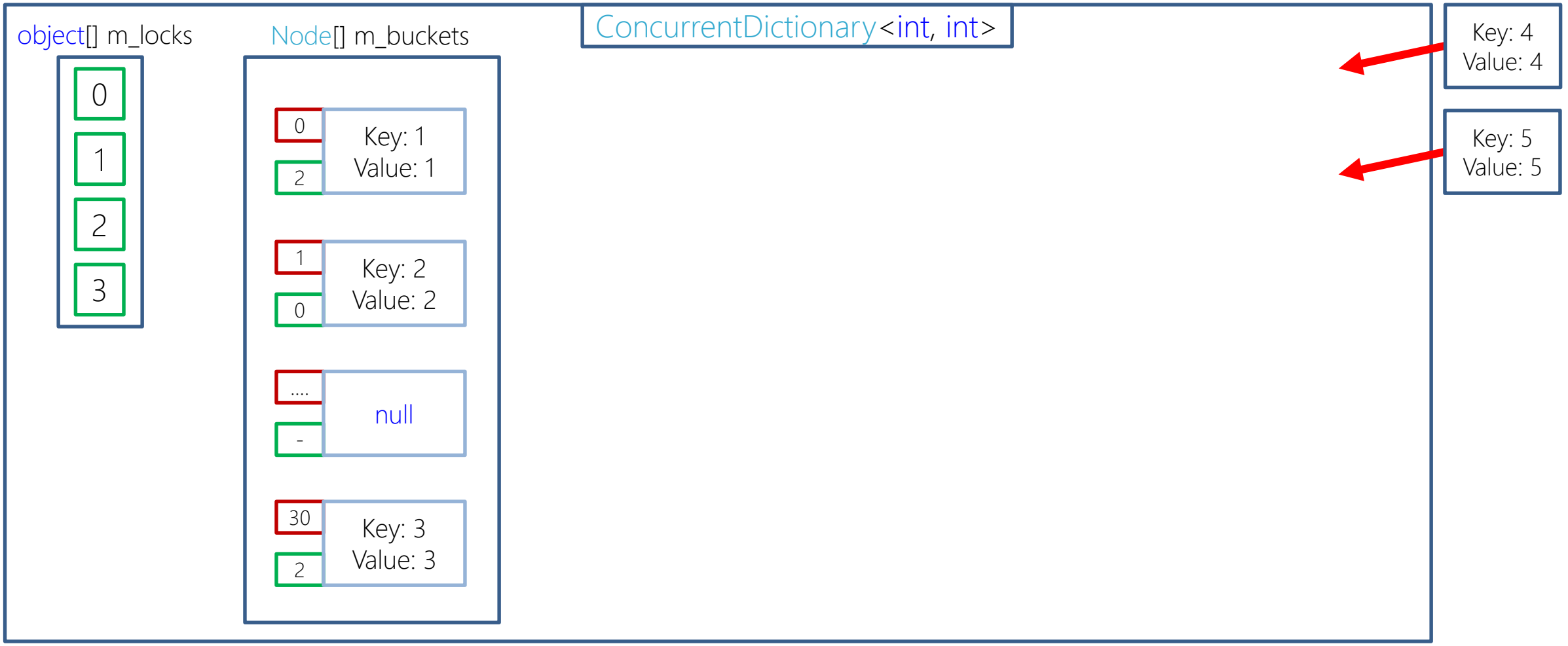
Хранение элементов в ConcurrentDictionary



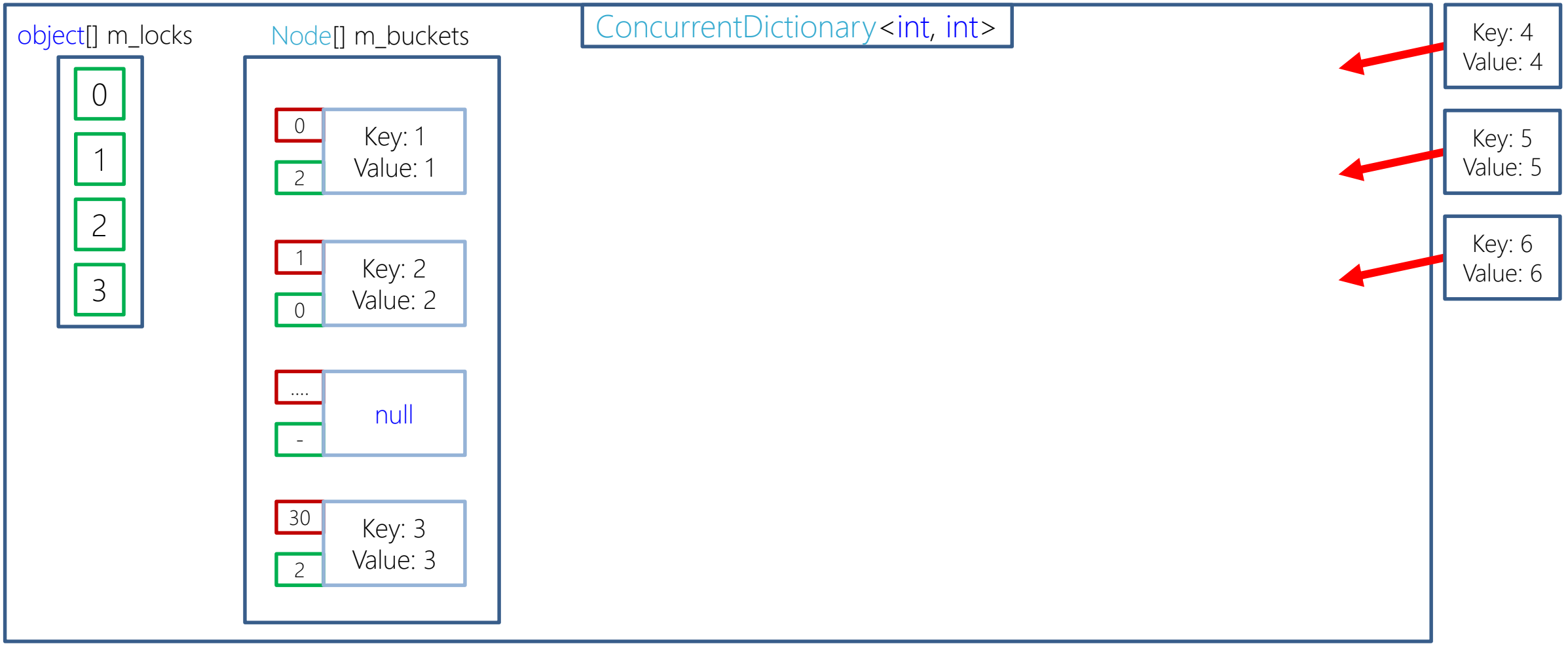
Хранение элементов в ConcurrentDictionary



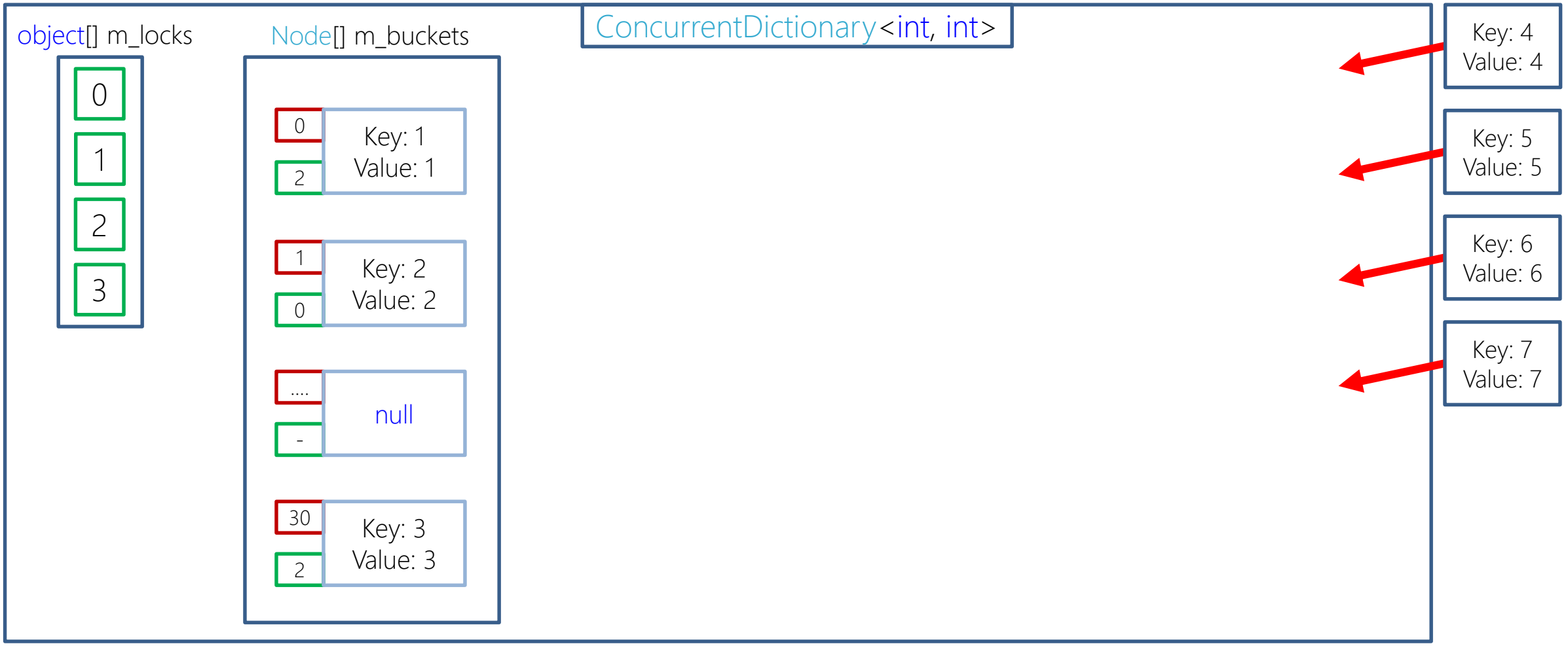
Хранение элементов в ConcurrentDictionary



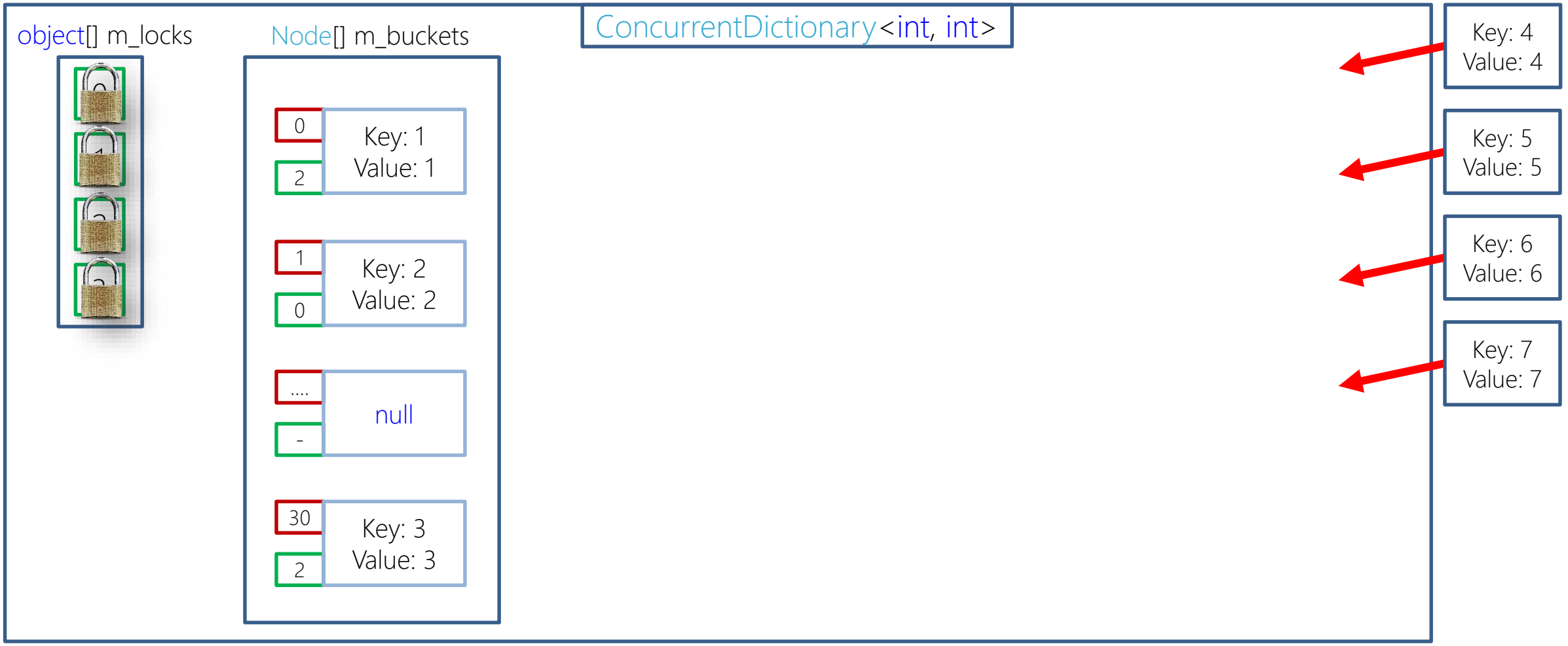
Хранение элементов в ConcurrentDictionary



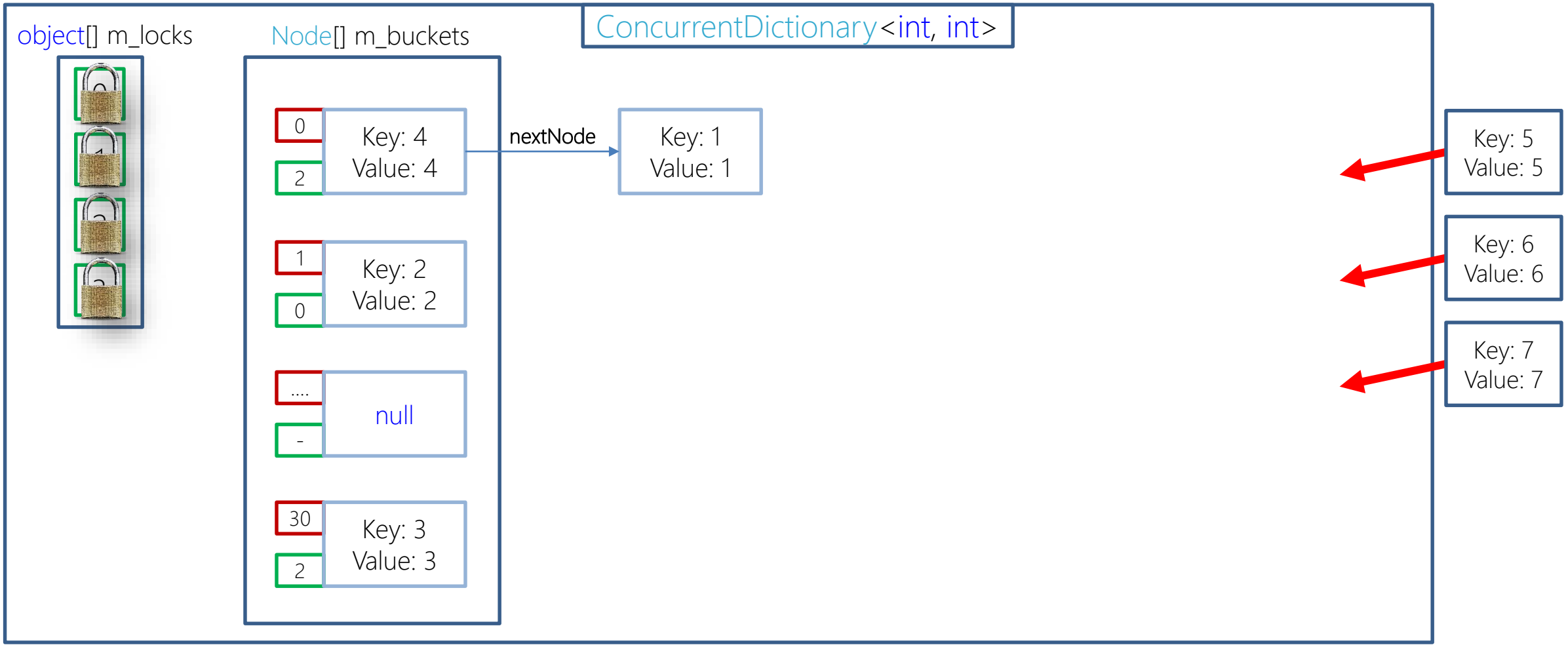
Хранение элементов в ConcurrentDictionary



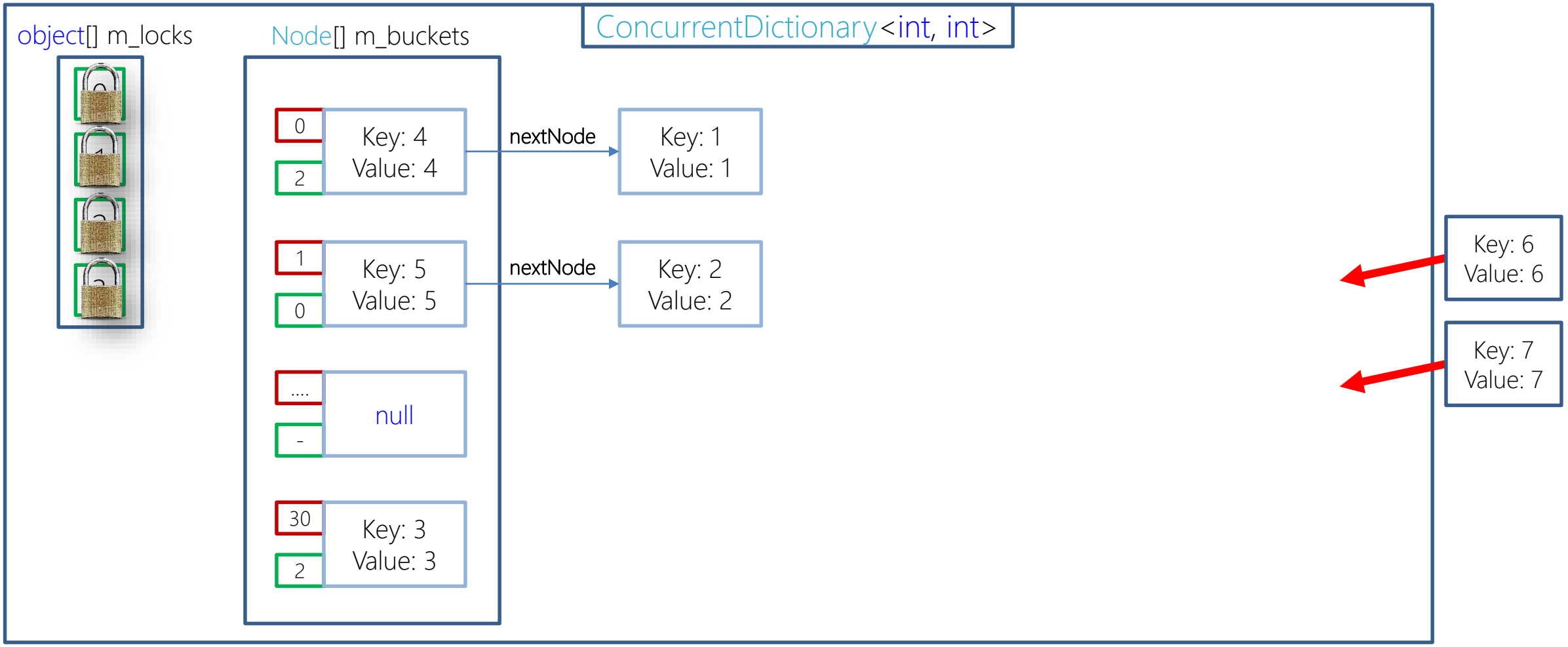
Хранение элементов в ConcurrentDictionary



Хранение элементов в ConcurrentDictionary

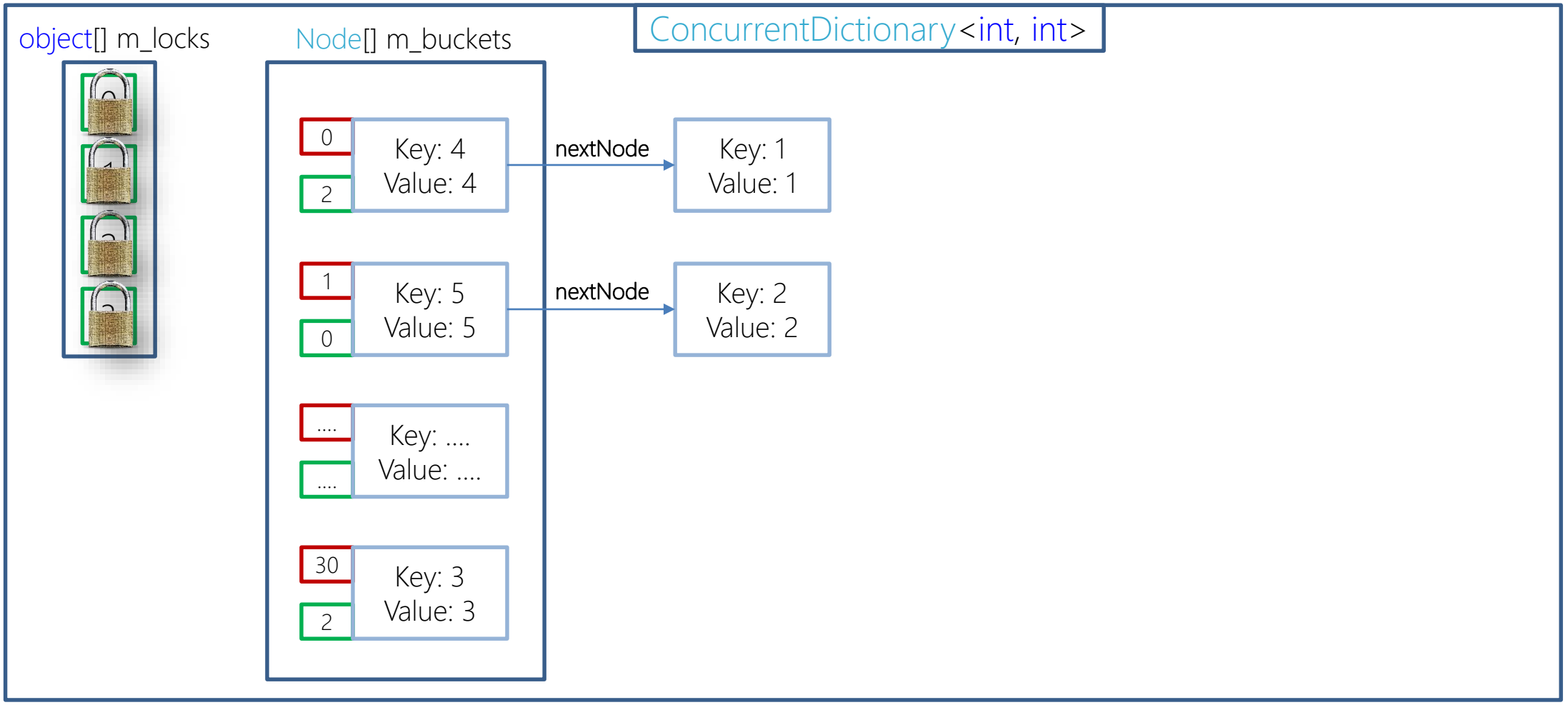


Хранение элементов в ConcurrentDictionary

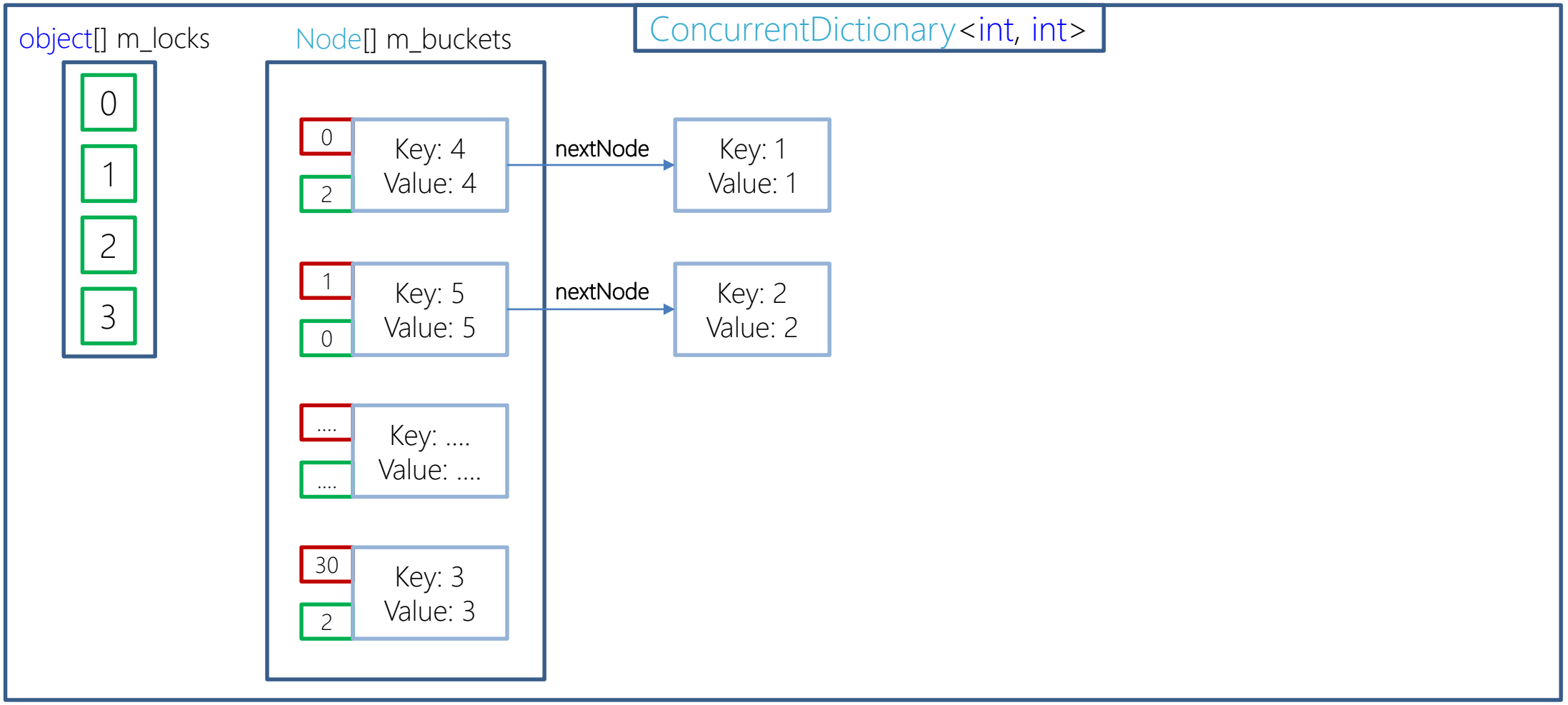


C# Асинхронное программирование

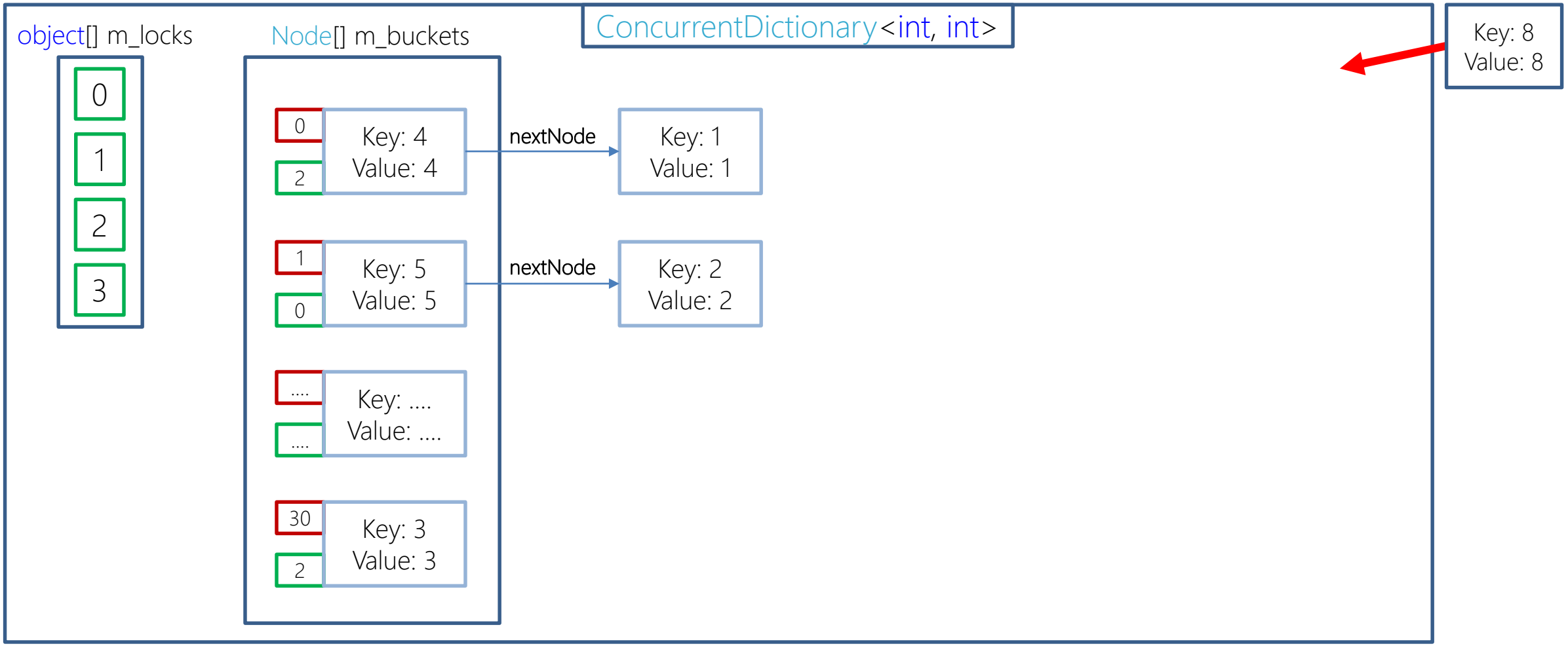
Хранение элементов в ConcurrentDictionary



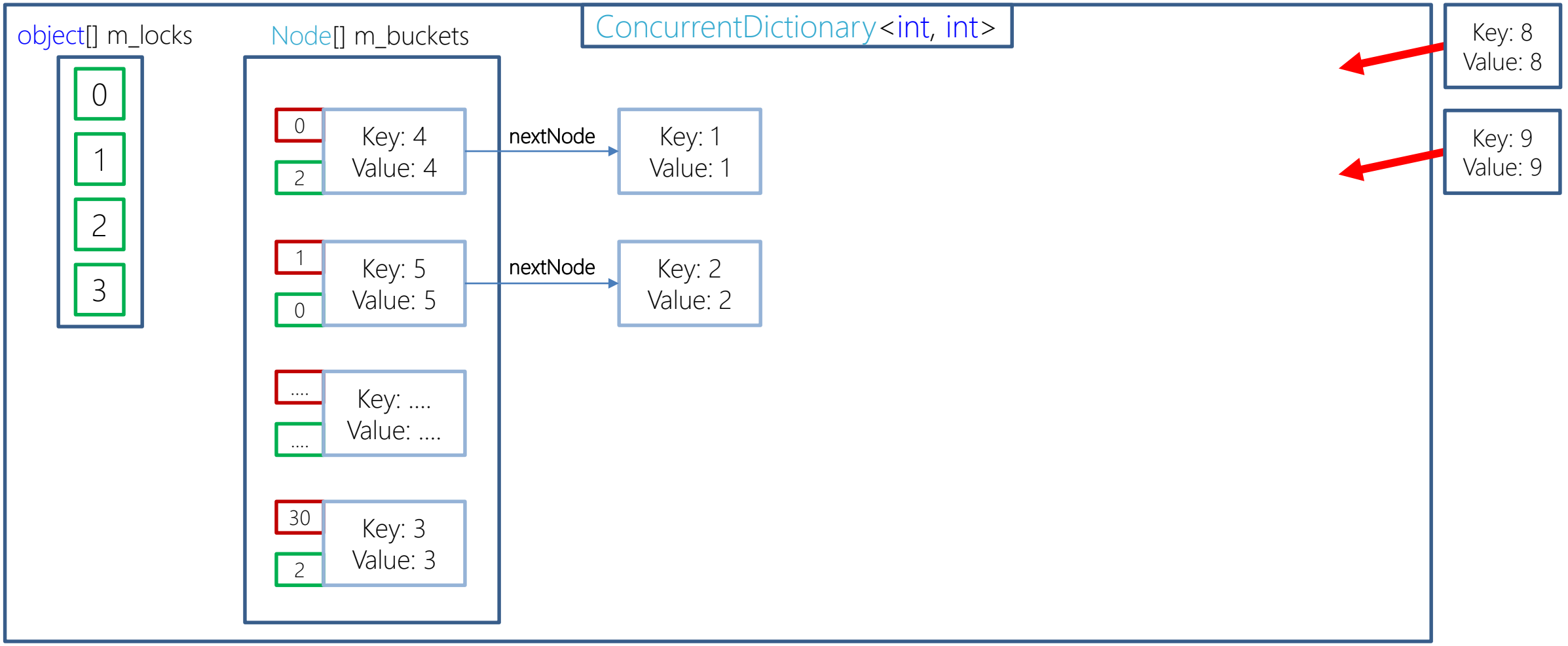
Хранение элементов в ConcurrentDictionary



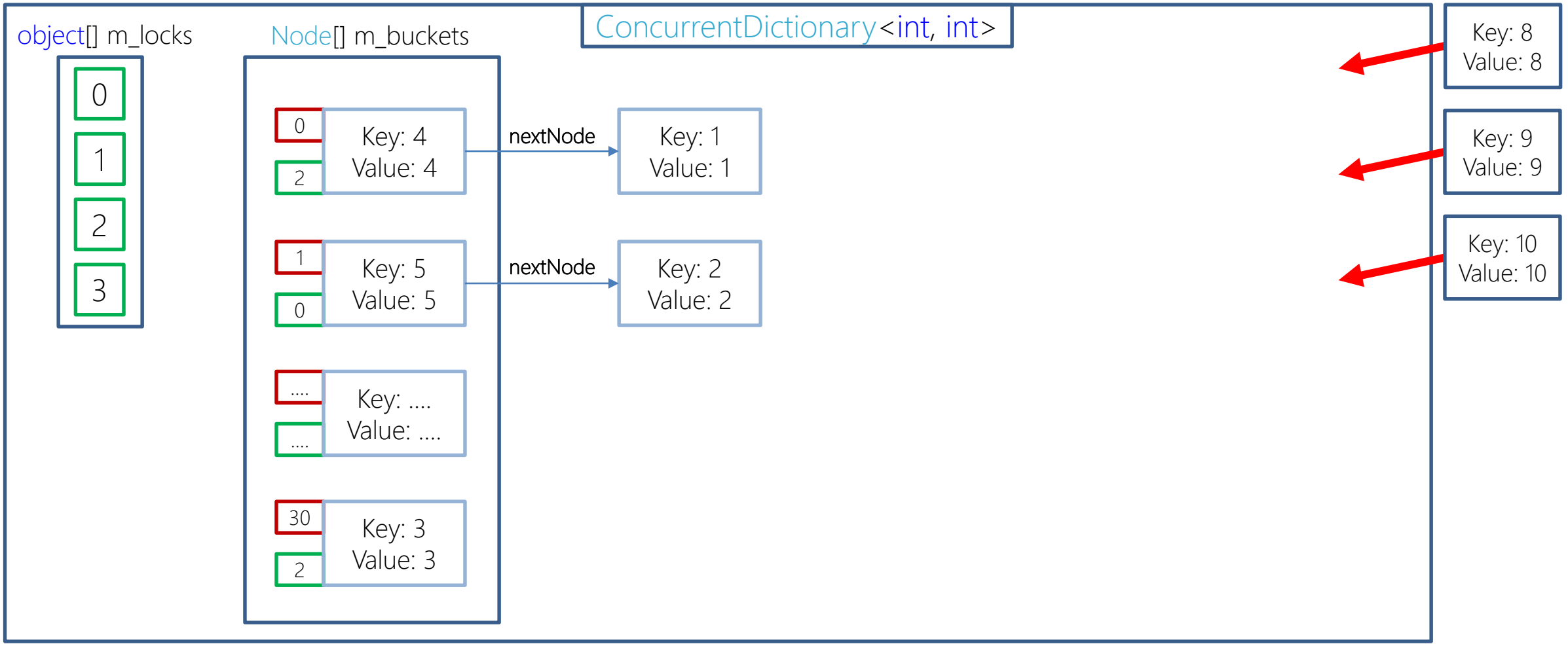
Хранение элементов в ConcurrentDictionary



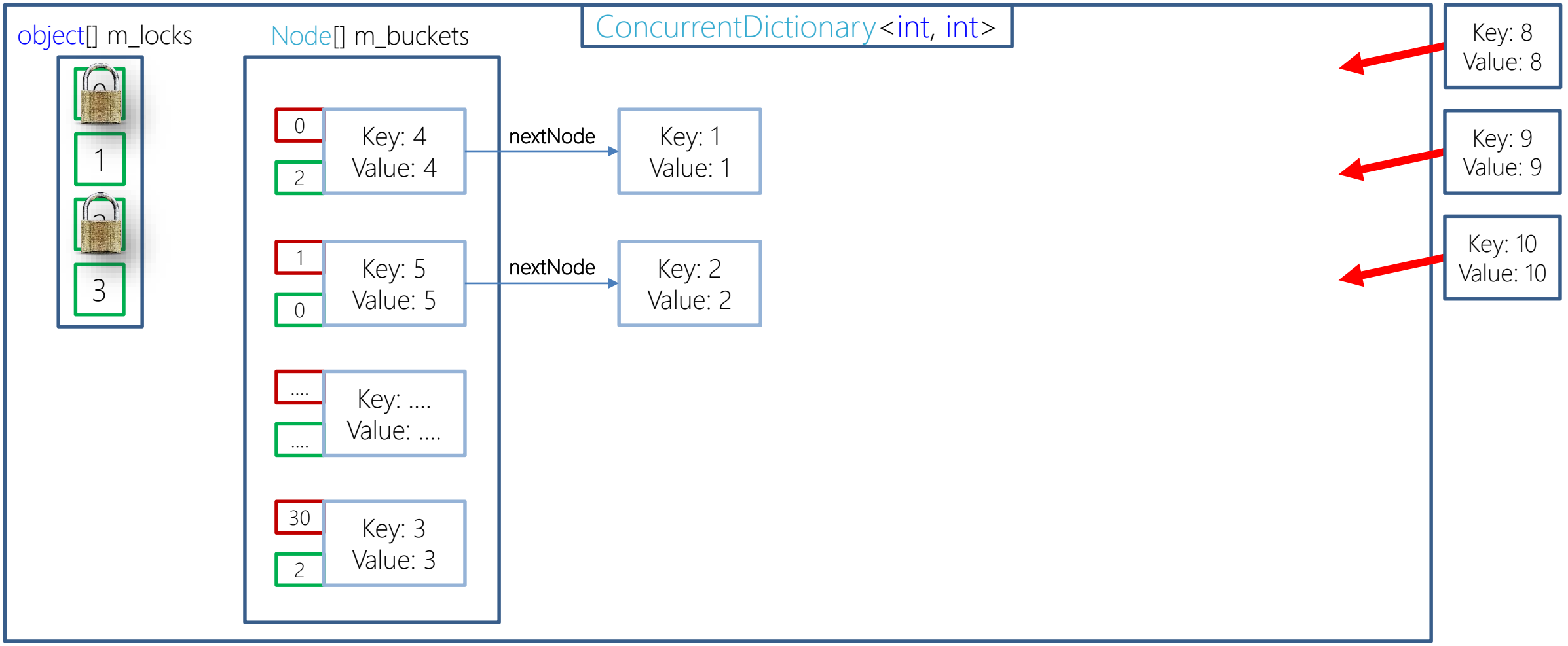
Хранение элементов в ConcurrentDictionary



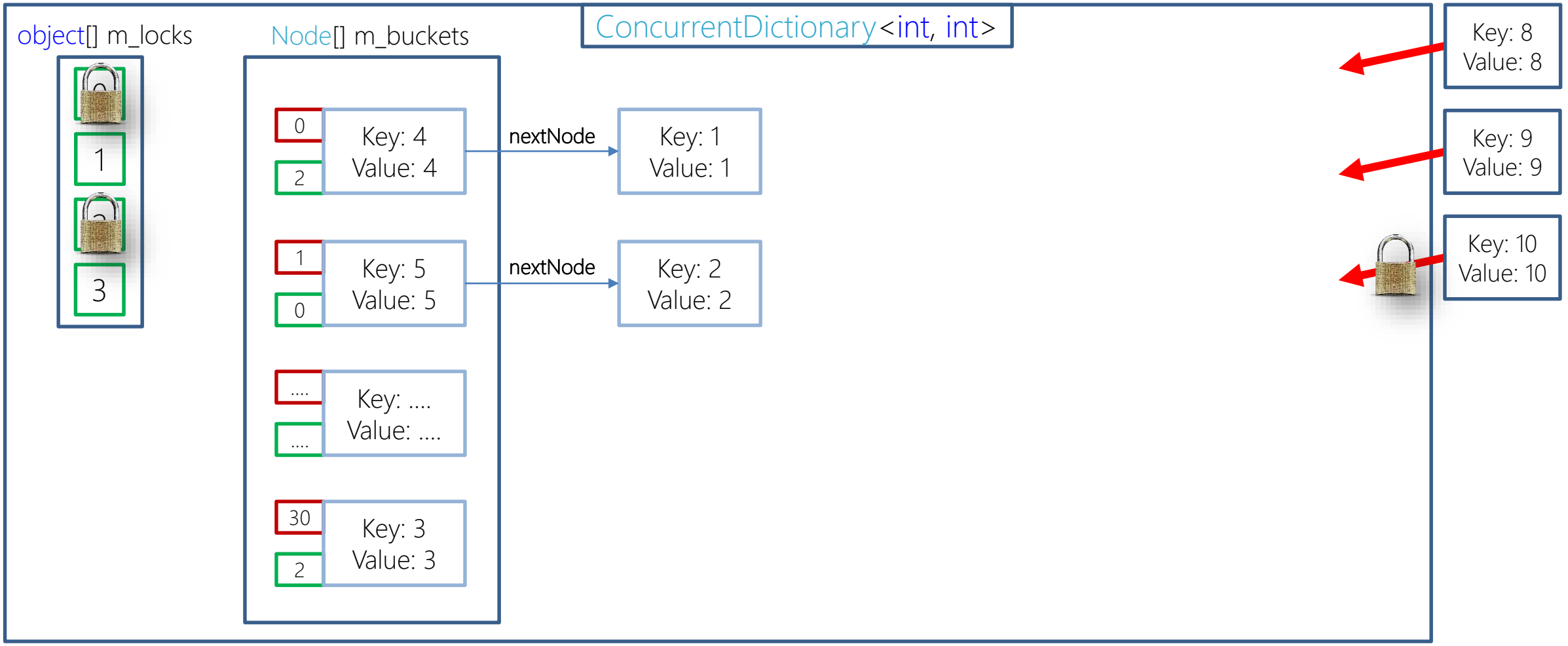
Хранение элементов в ConcurrentDictionary



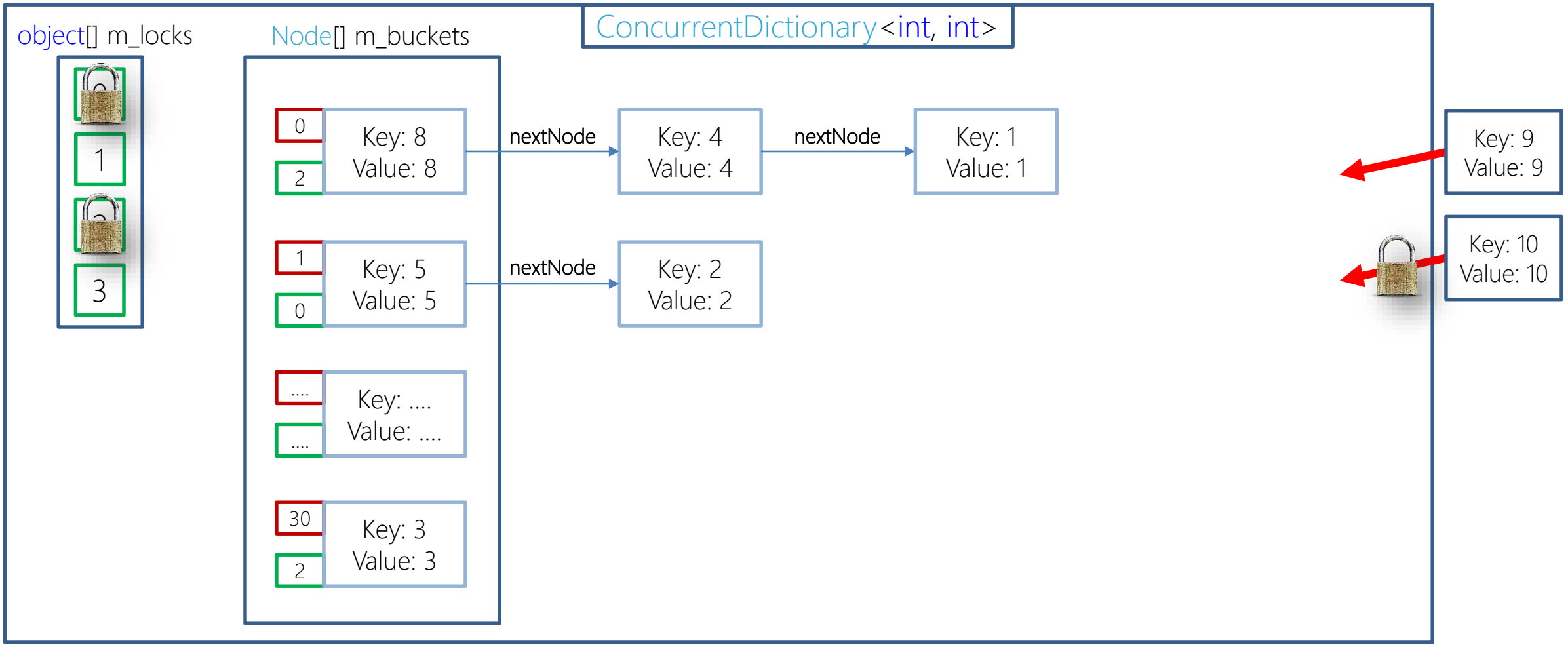
Хранение элементов в ConcurrentDictionary



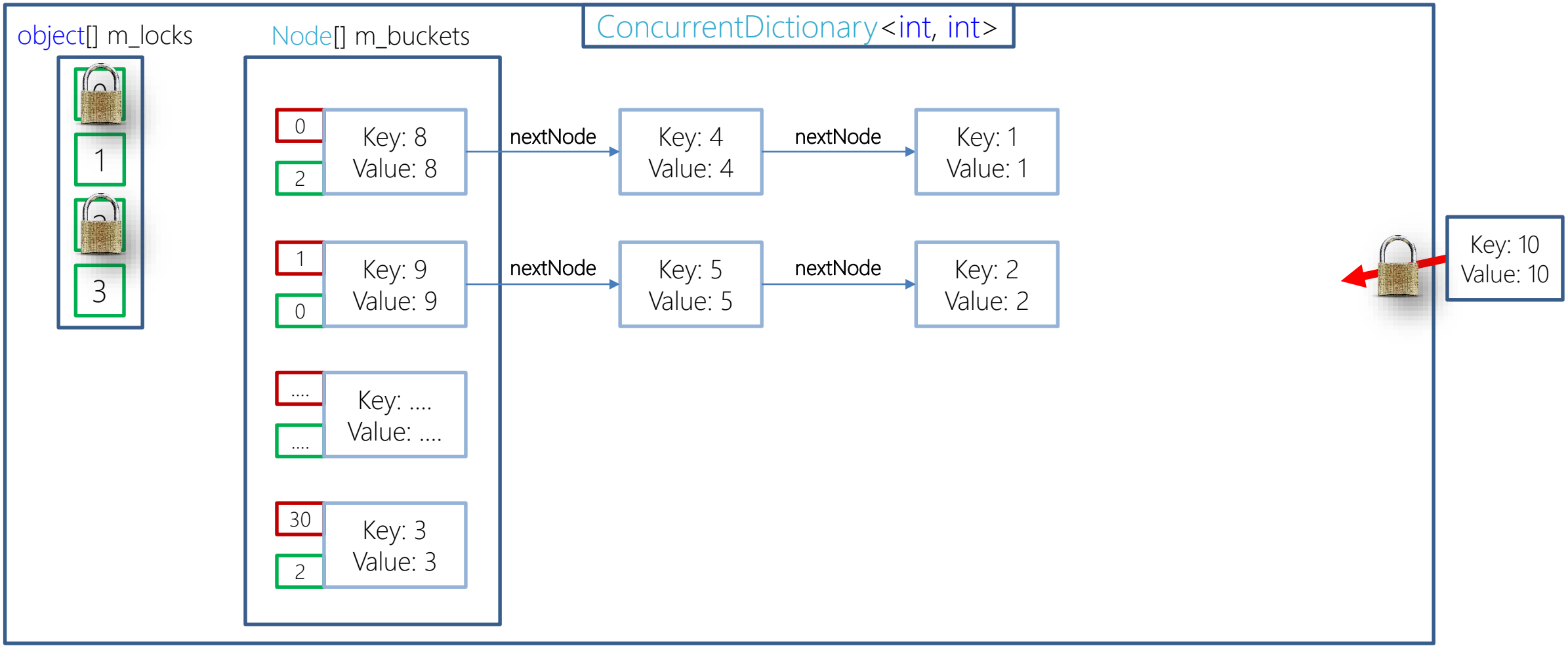
Хранение элементов в ConcurrentDictionary



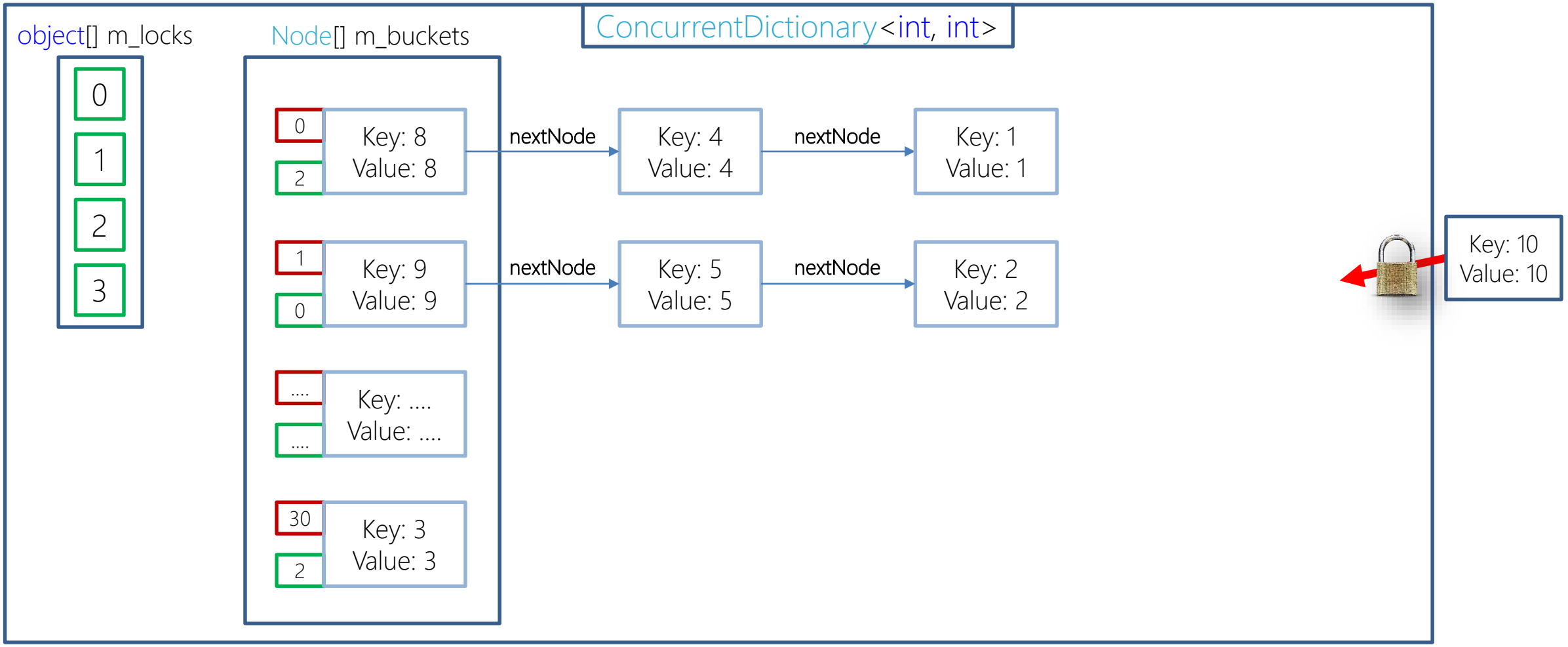
Хранение элементов в ConcurrentDictionary



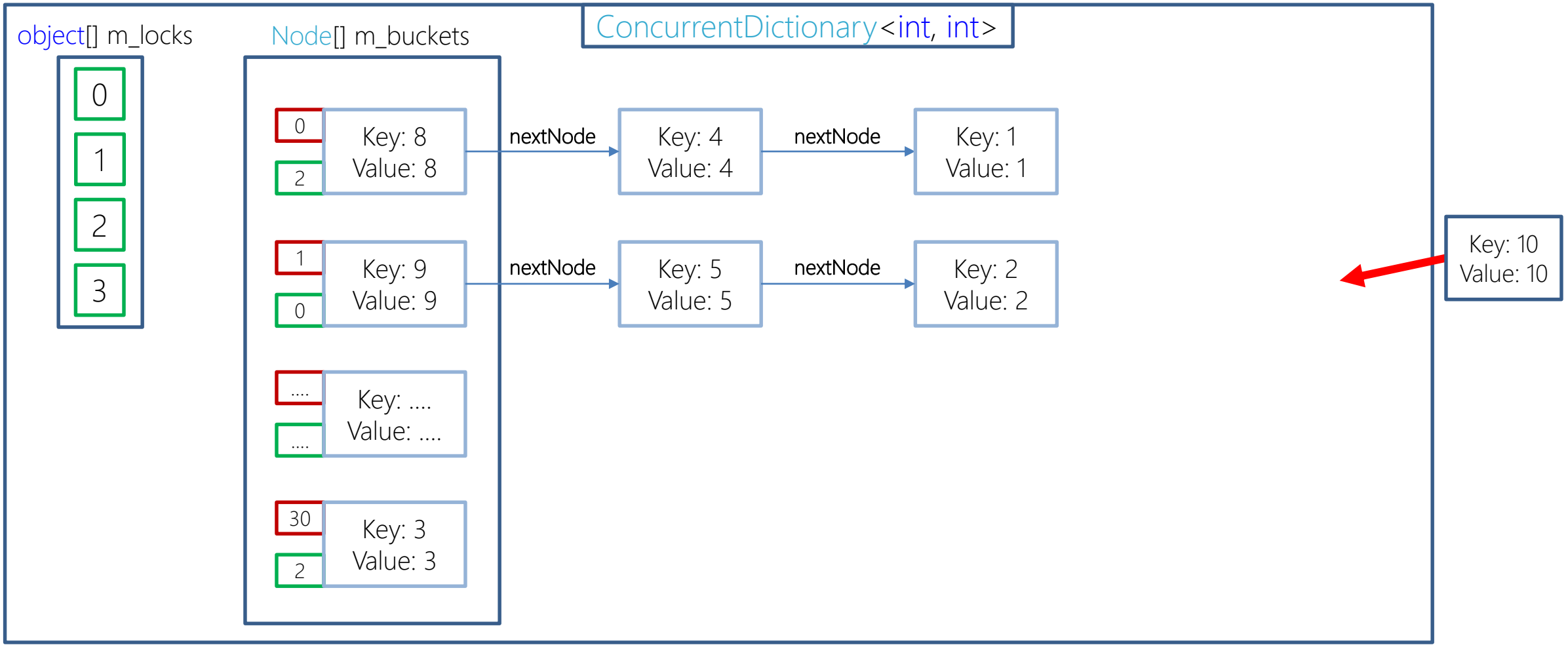
Хранение элементов в ConcurrentDictionary



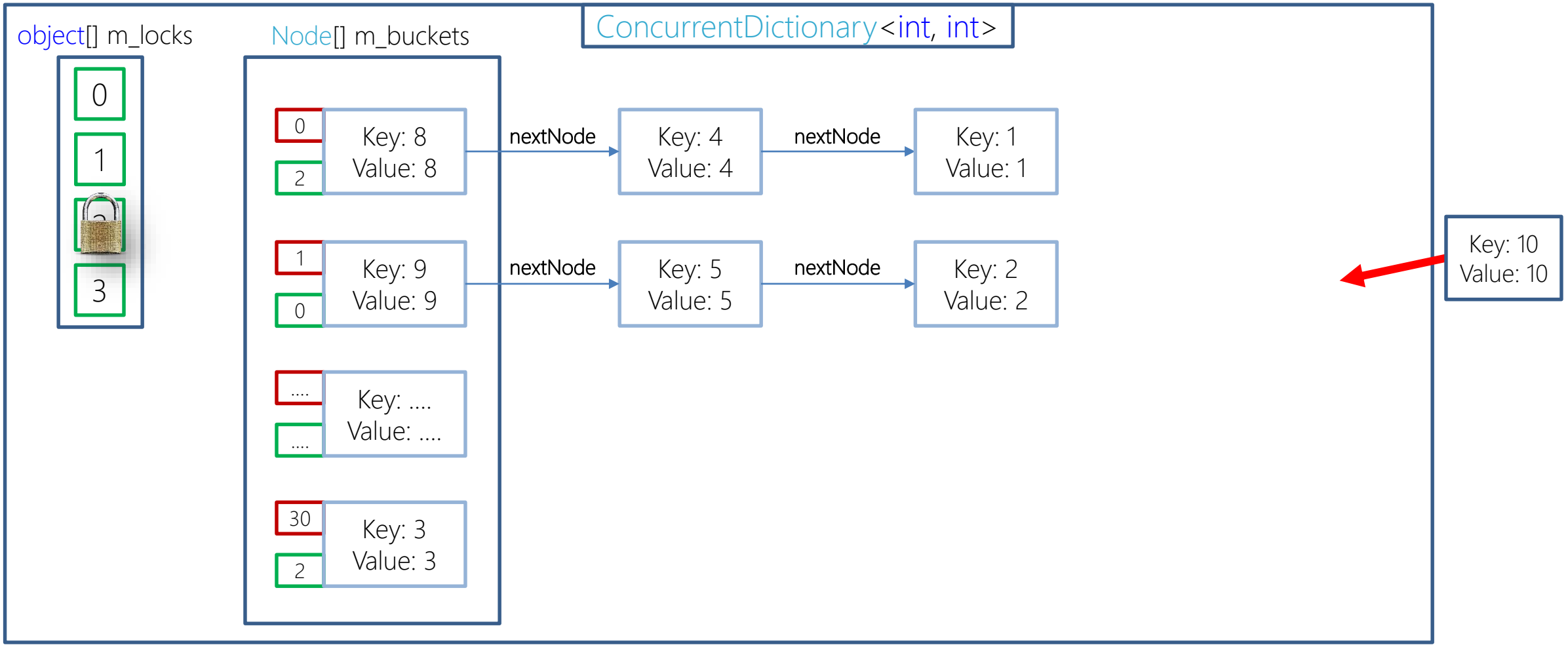
Хранение элементов в ConcurrentDictionary



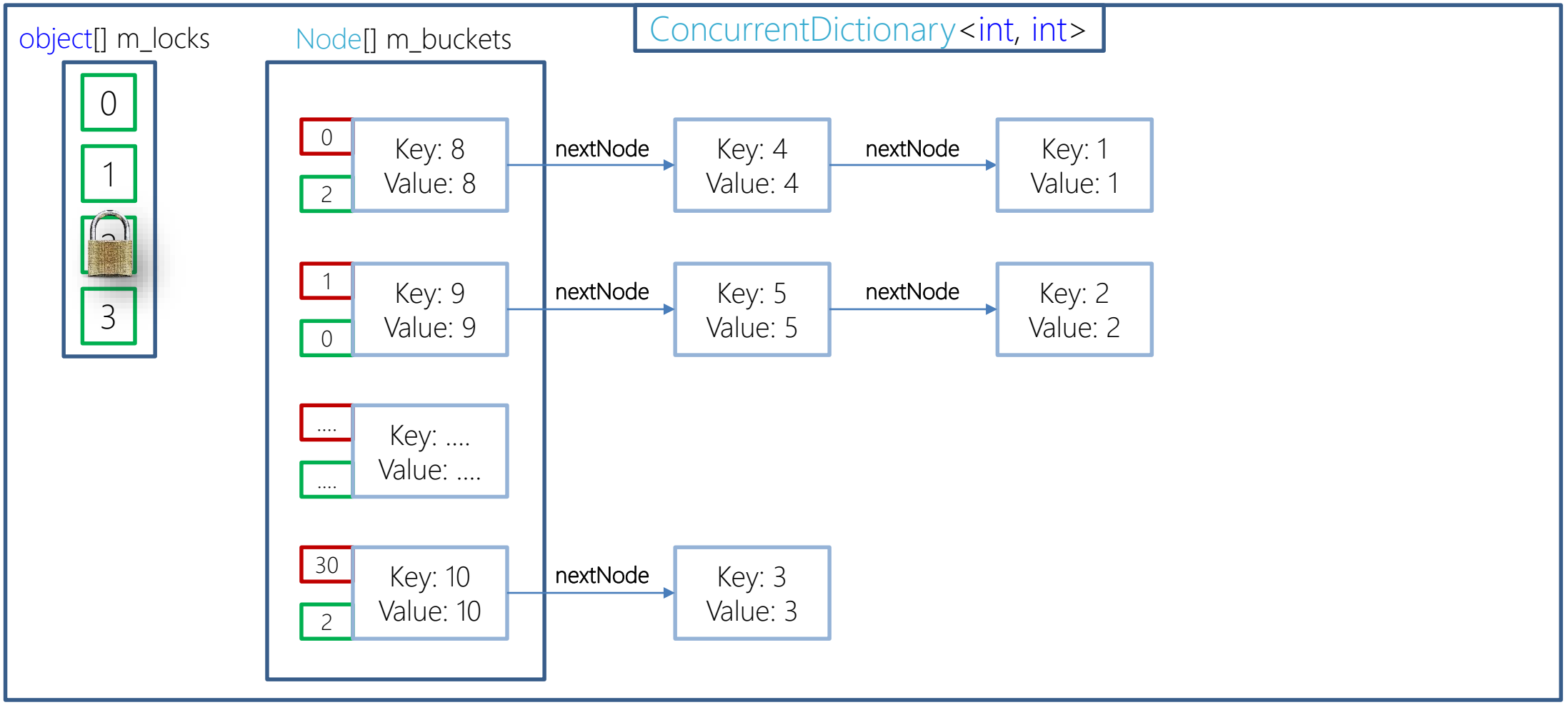
Хранение элементов в ConcurrentDictionary



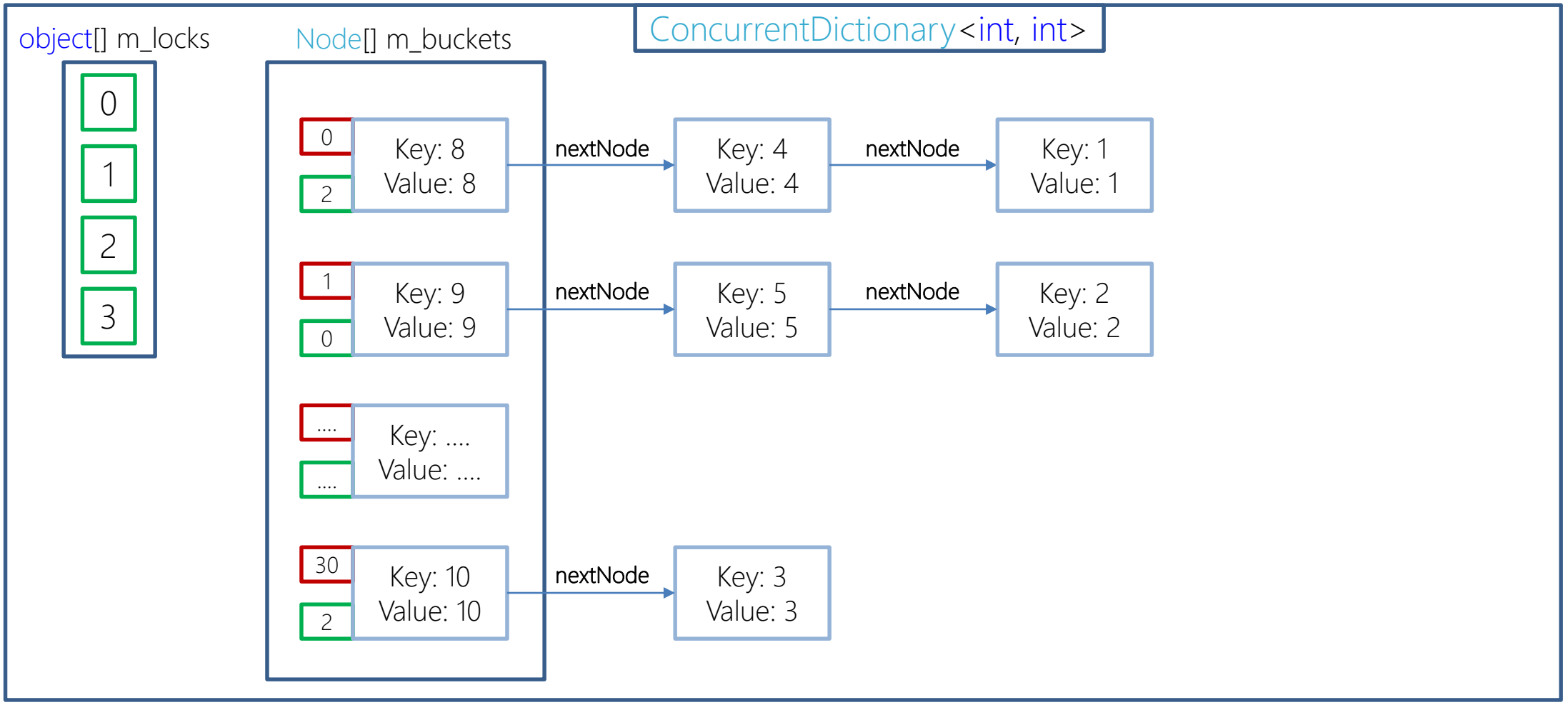
Хранение элементов в ConcurrentDictionary



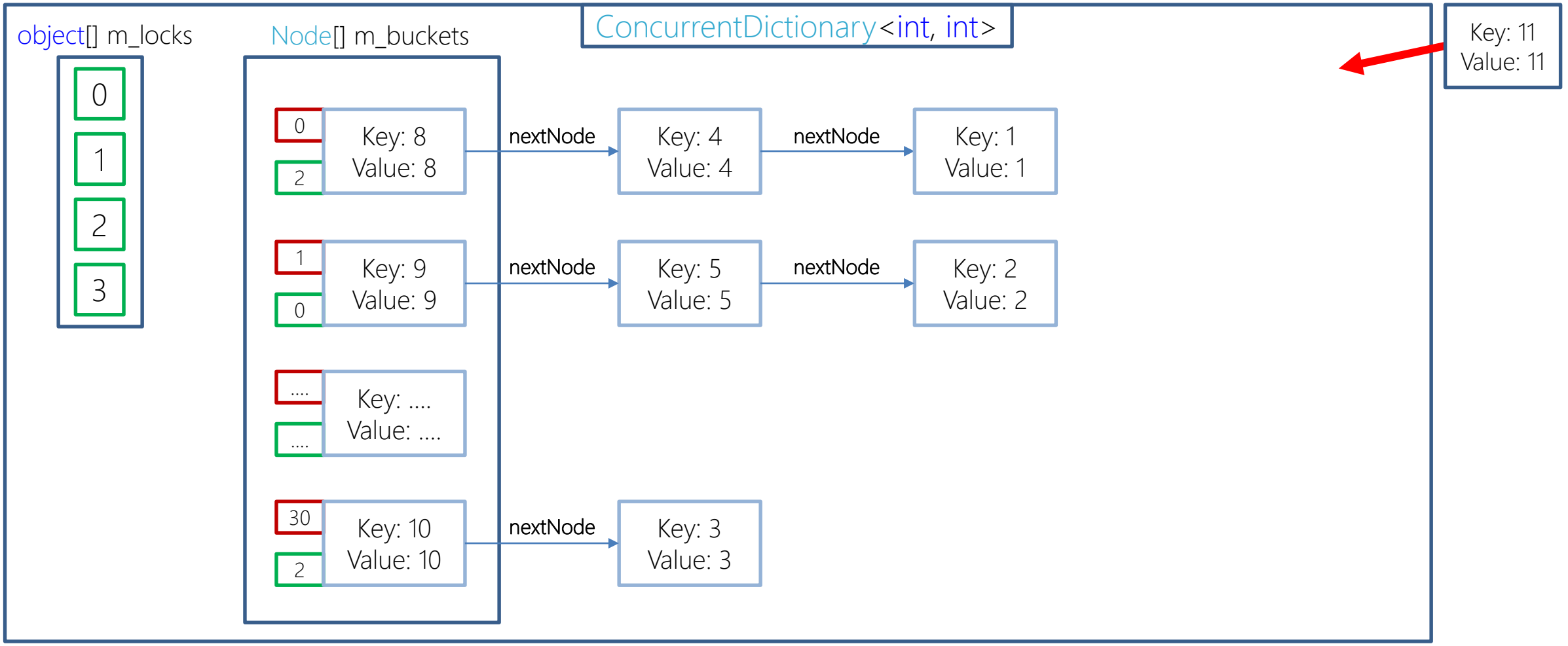
Хранение элементов в ConcurrentDictionary



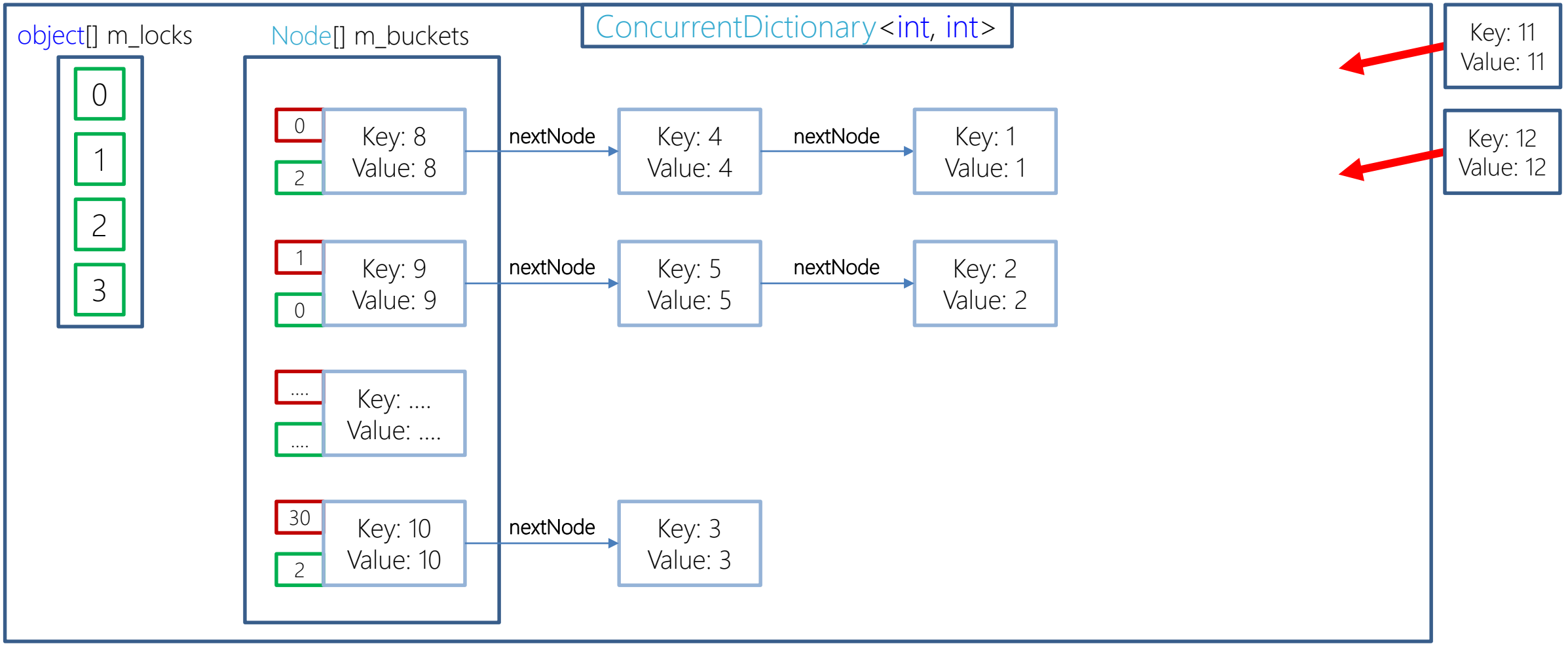
Хранение элементов в ConcurrentDictionary



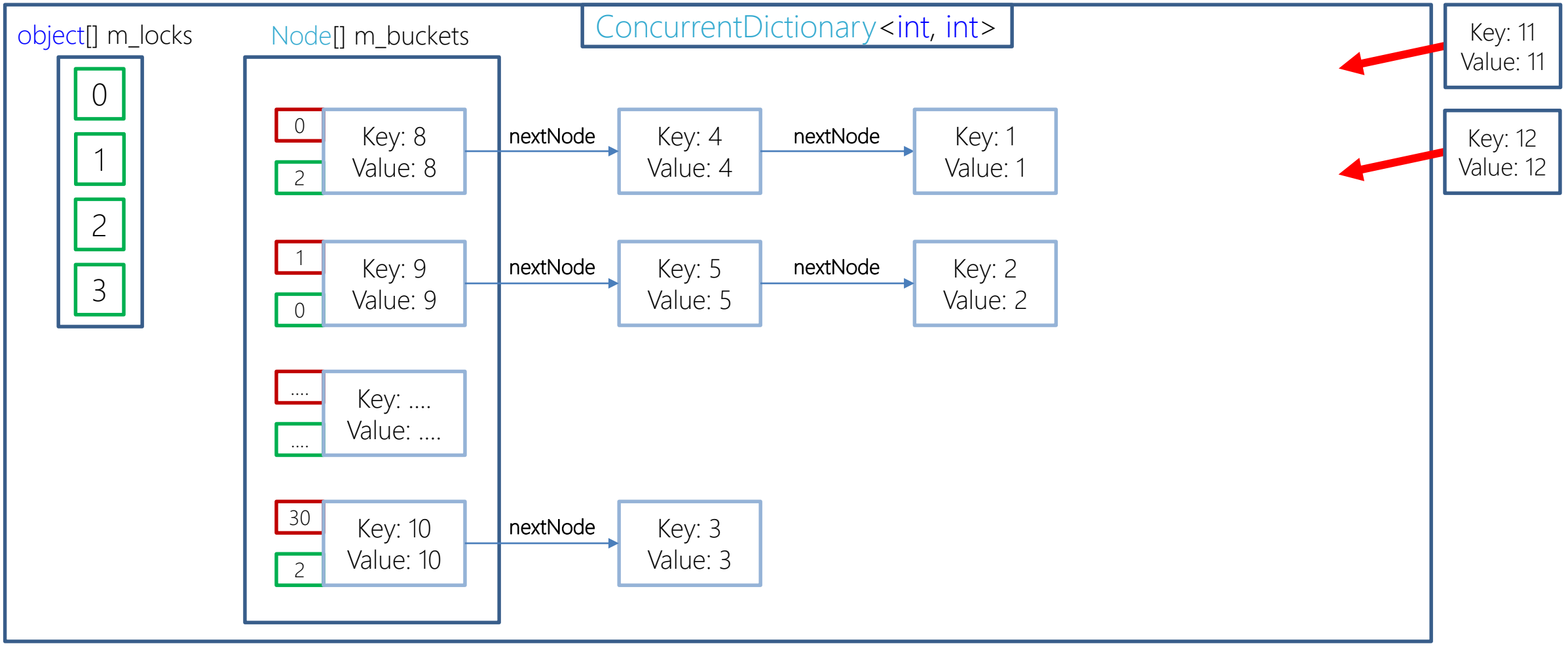
Хранение элементов в ConcurrentDictionary



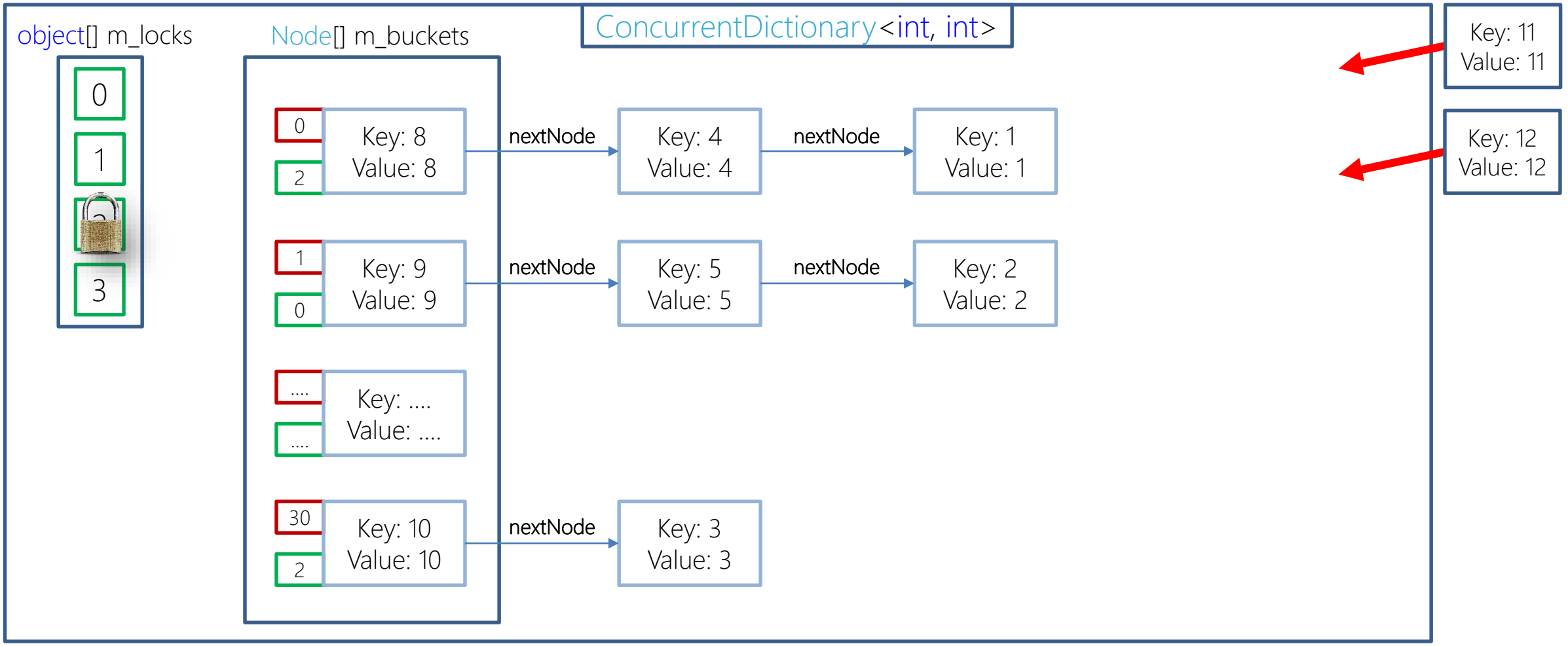
Хранение элементов в ConcurrentDictionary



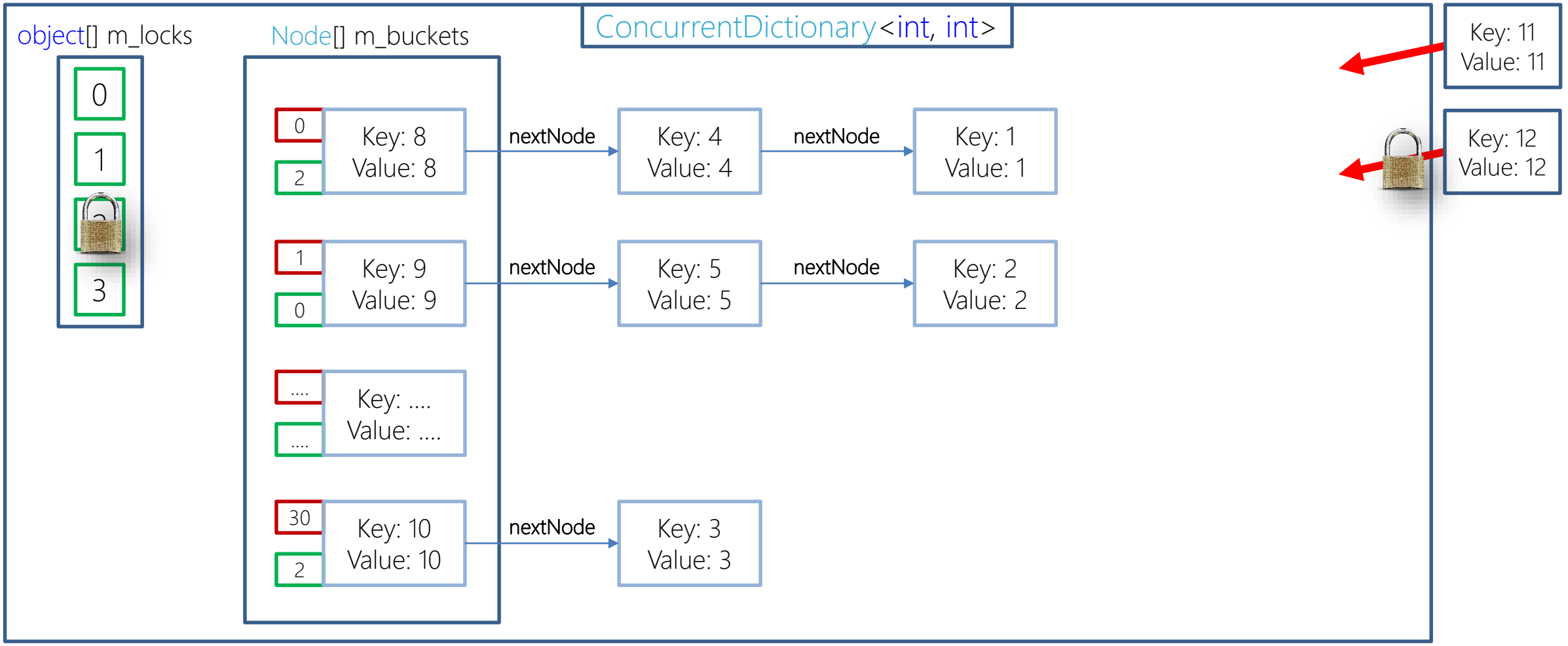
Хранение элементов в ConcurrentDictionary



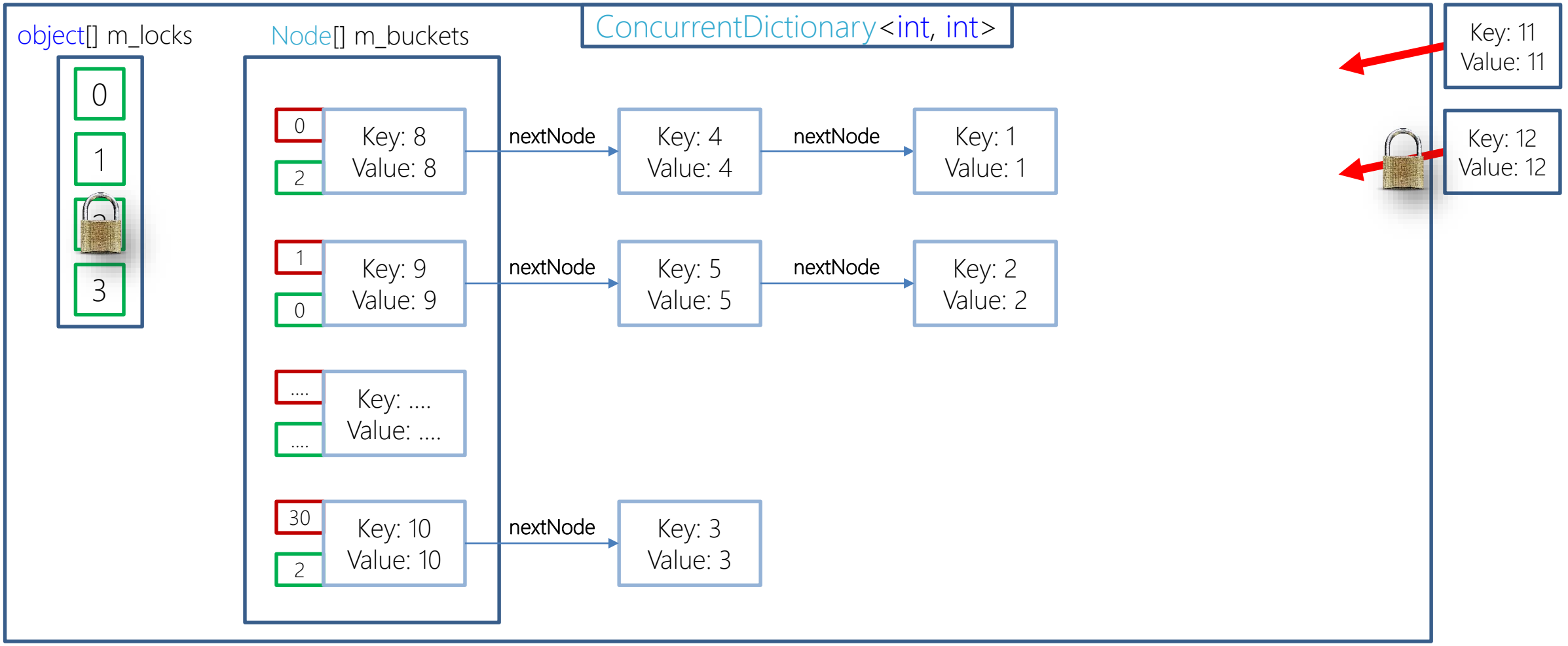
Хранение элементов в ConcurrentDictionary



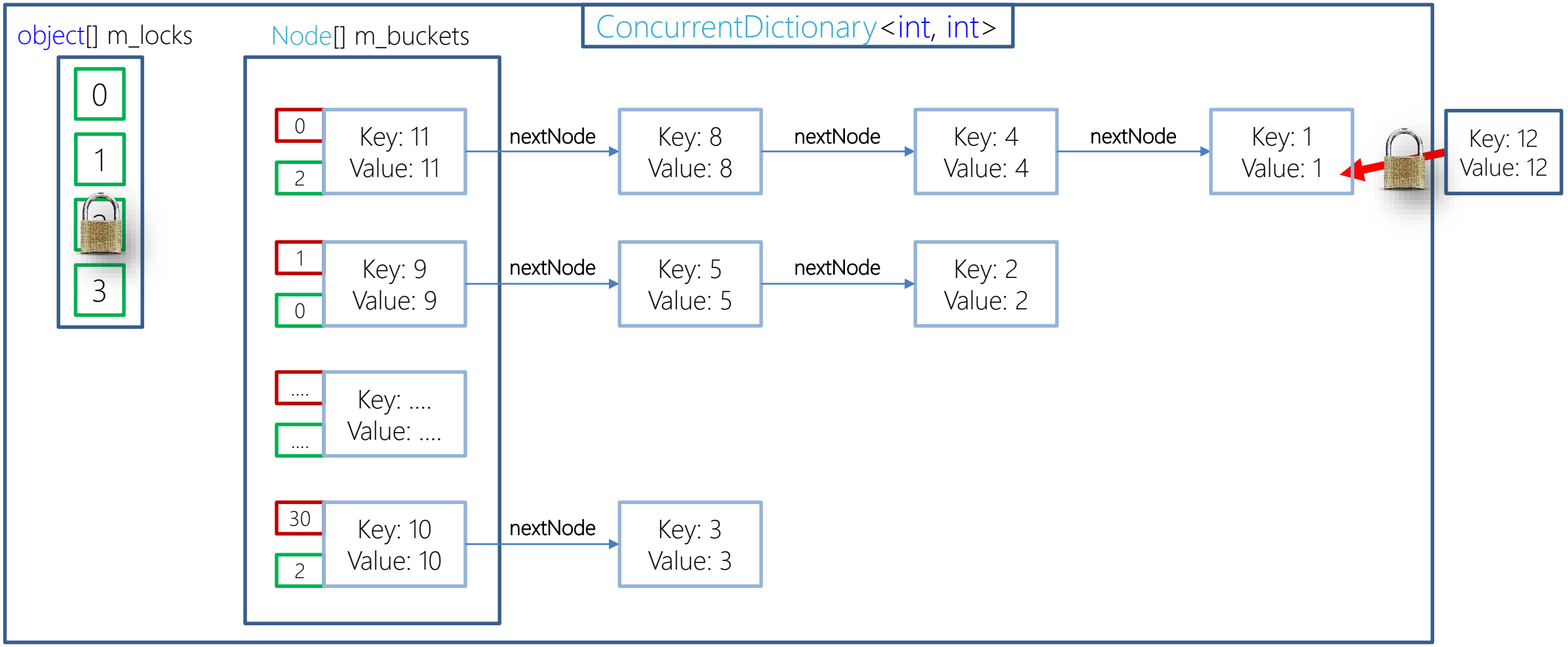
Хранение элементов в ConcurrentDictionary



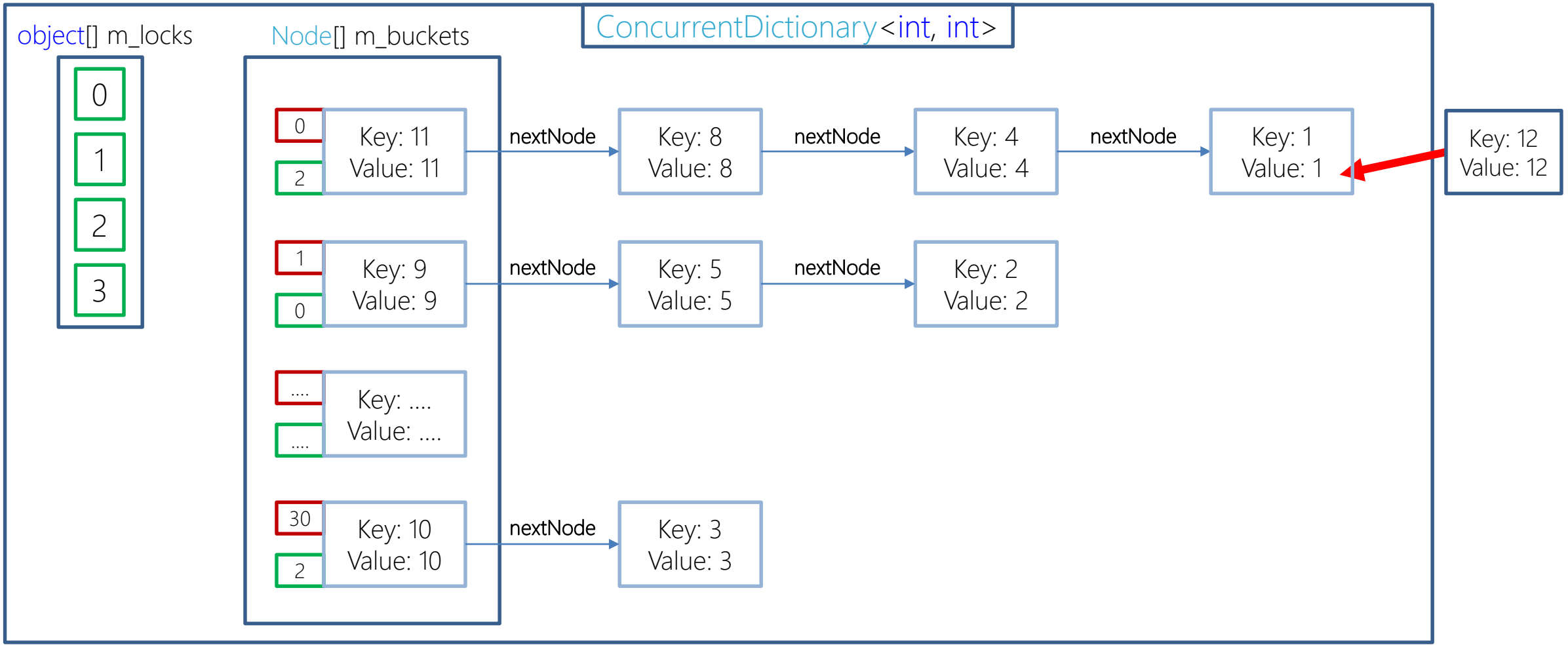
Хранение элементов в ConcurrentDictionary



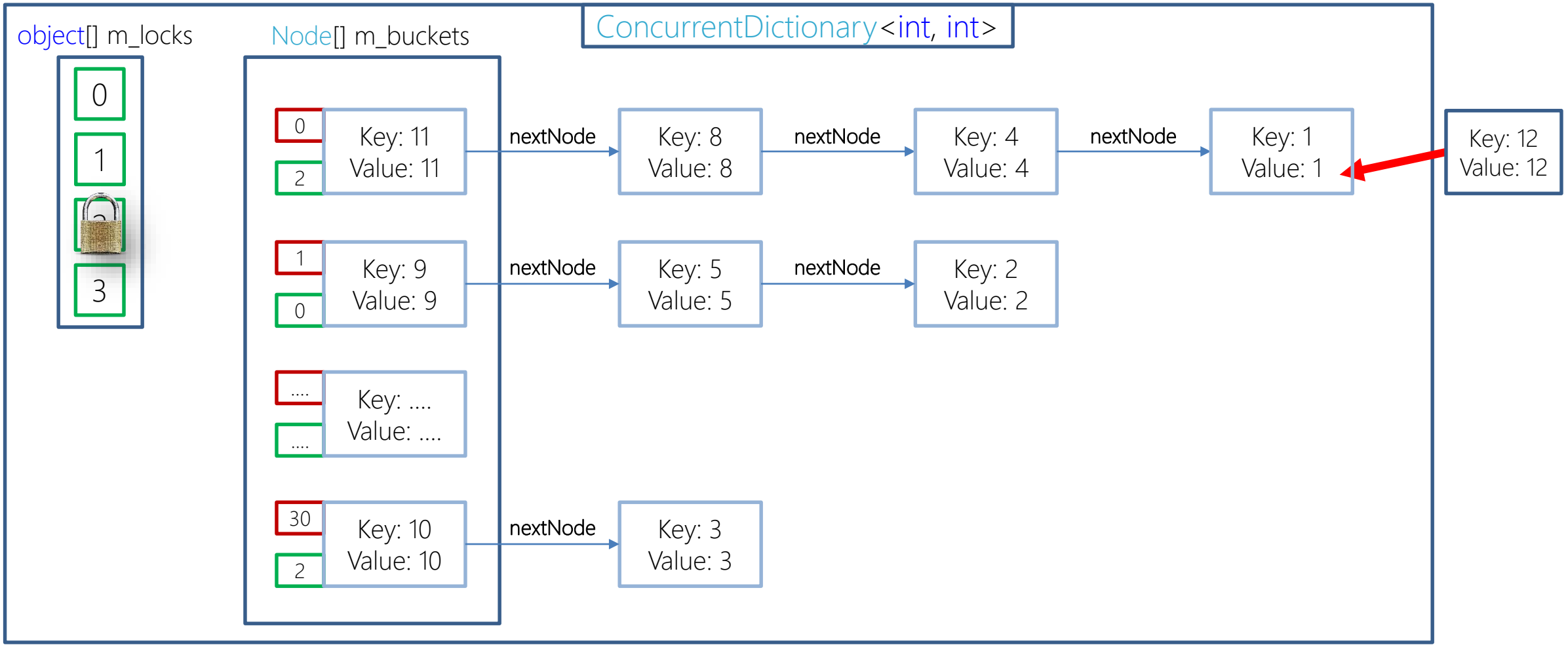
Хранение элементов в ConcurrentDictionary



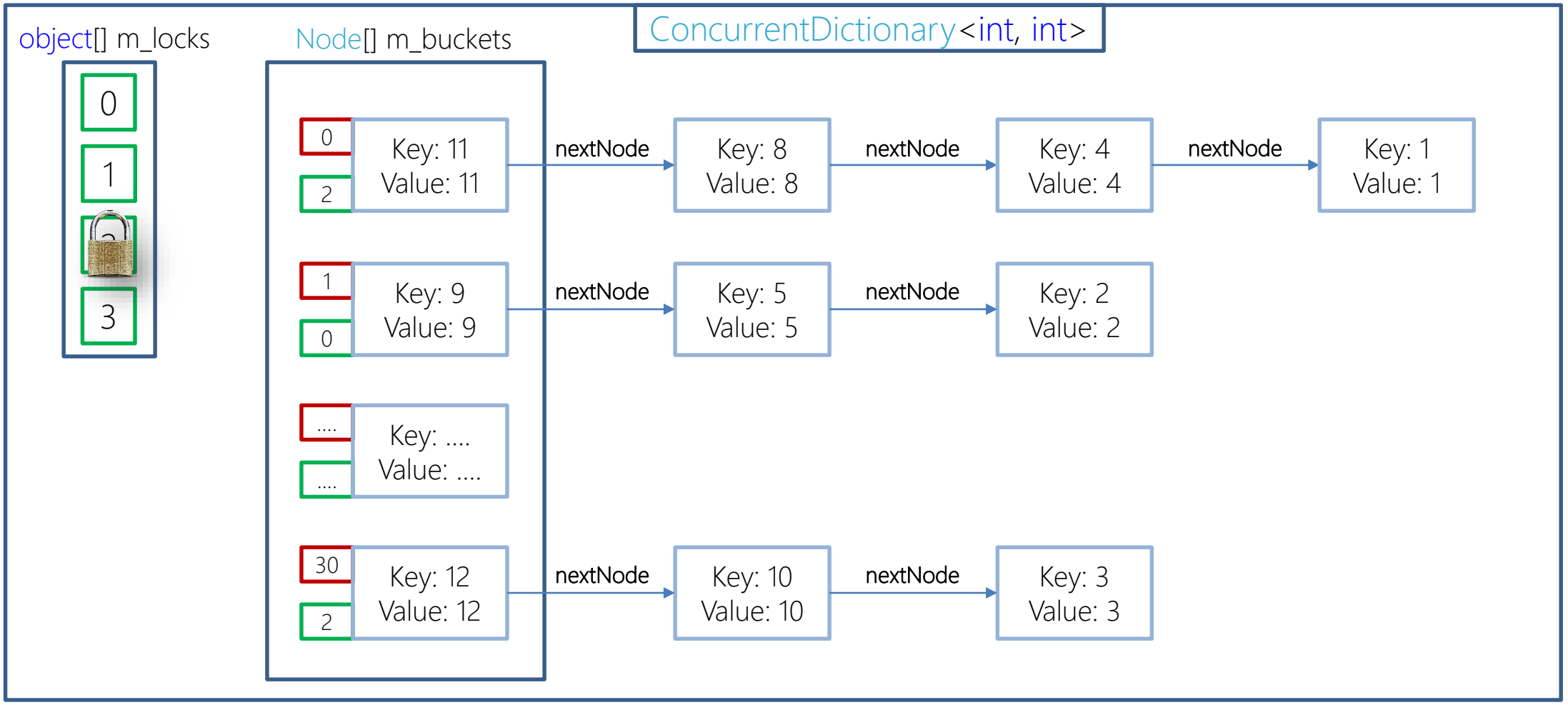
Хранение элементов в ConcurrentDictionary



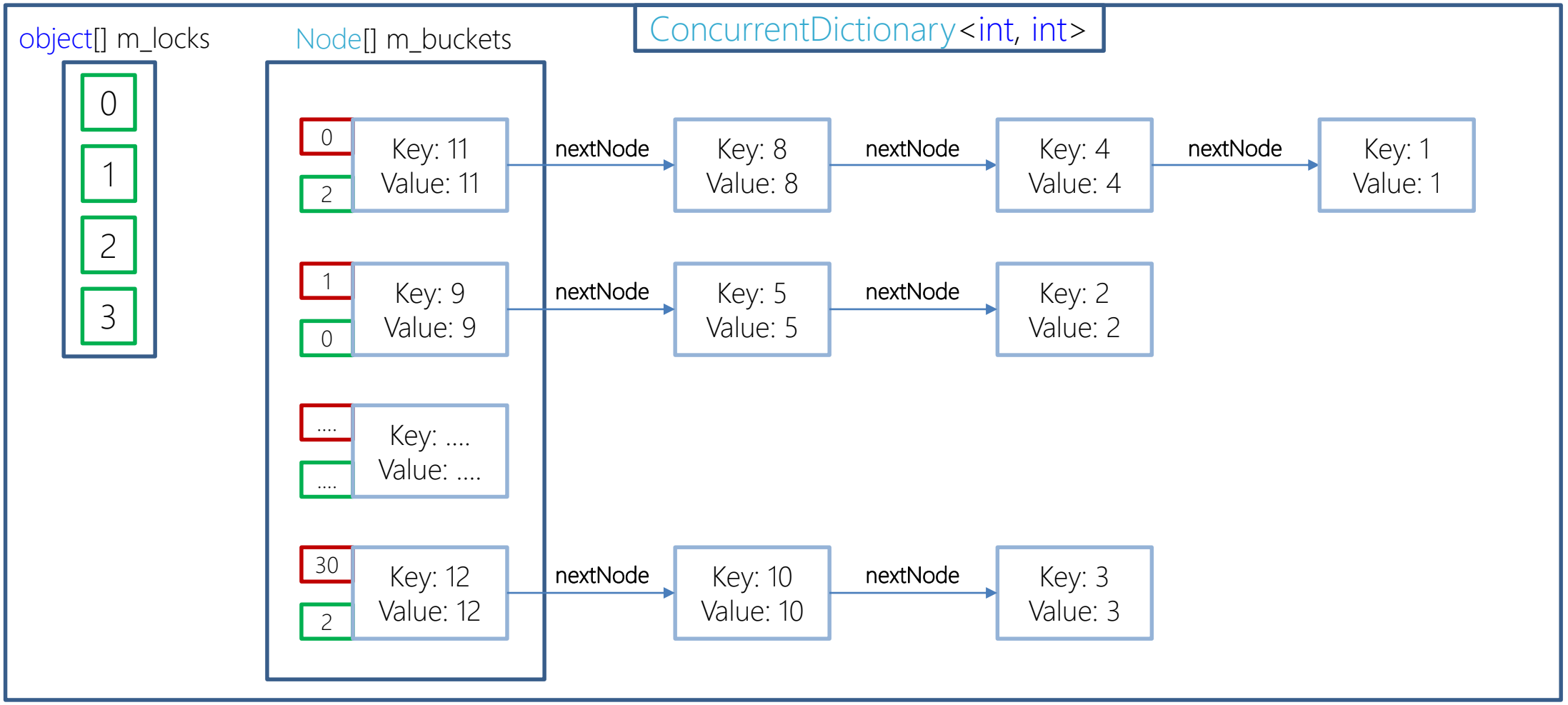
Хранение элементов в ConcurrentDictionary



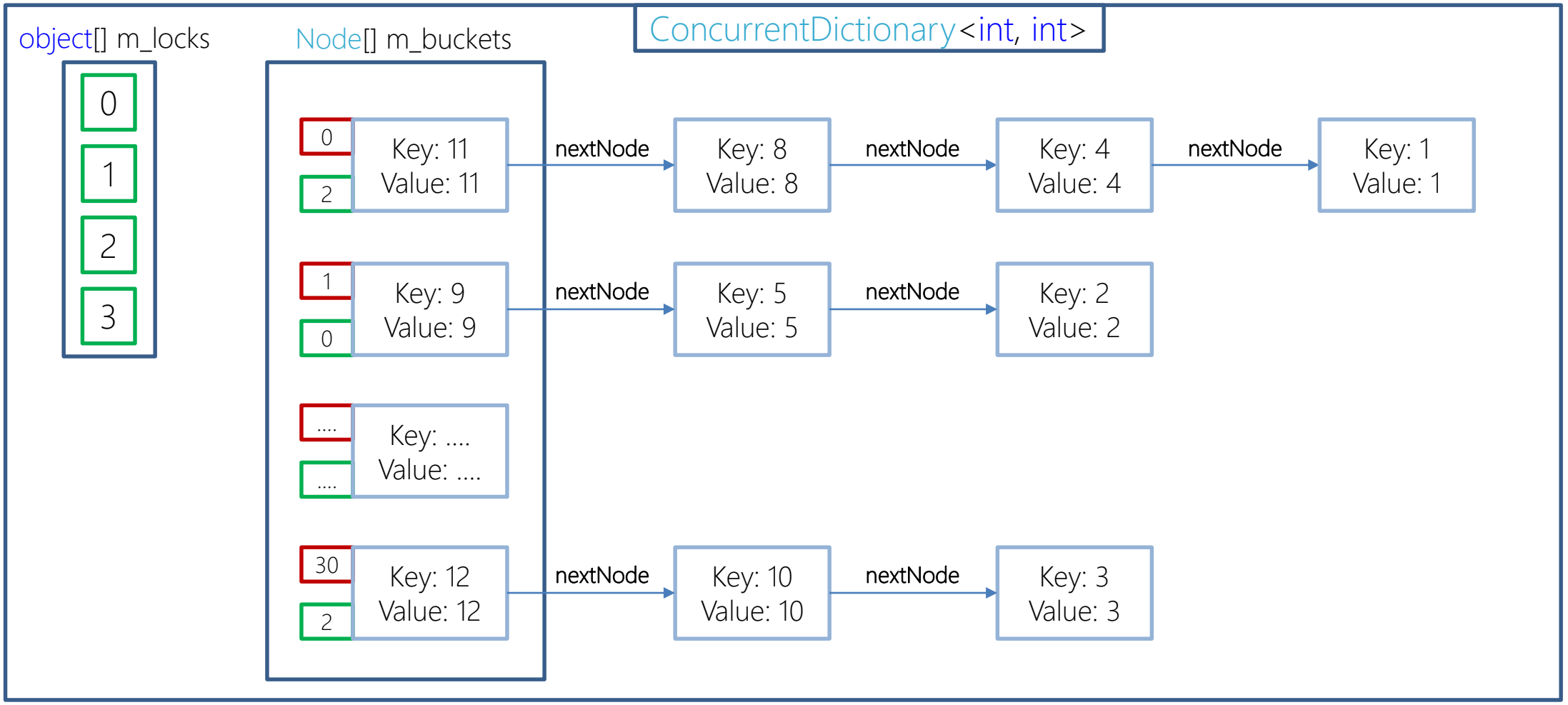
Хранение элементов в ConcurrentDictionary



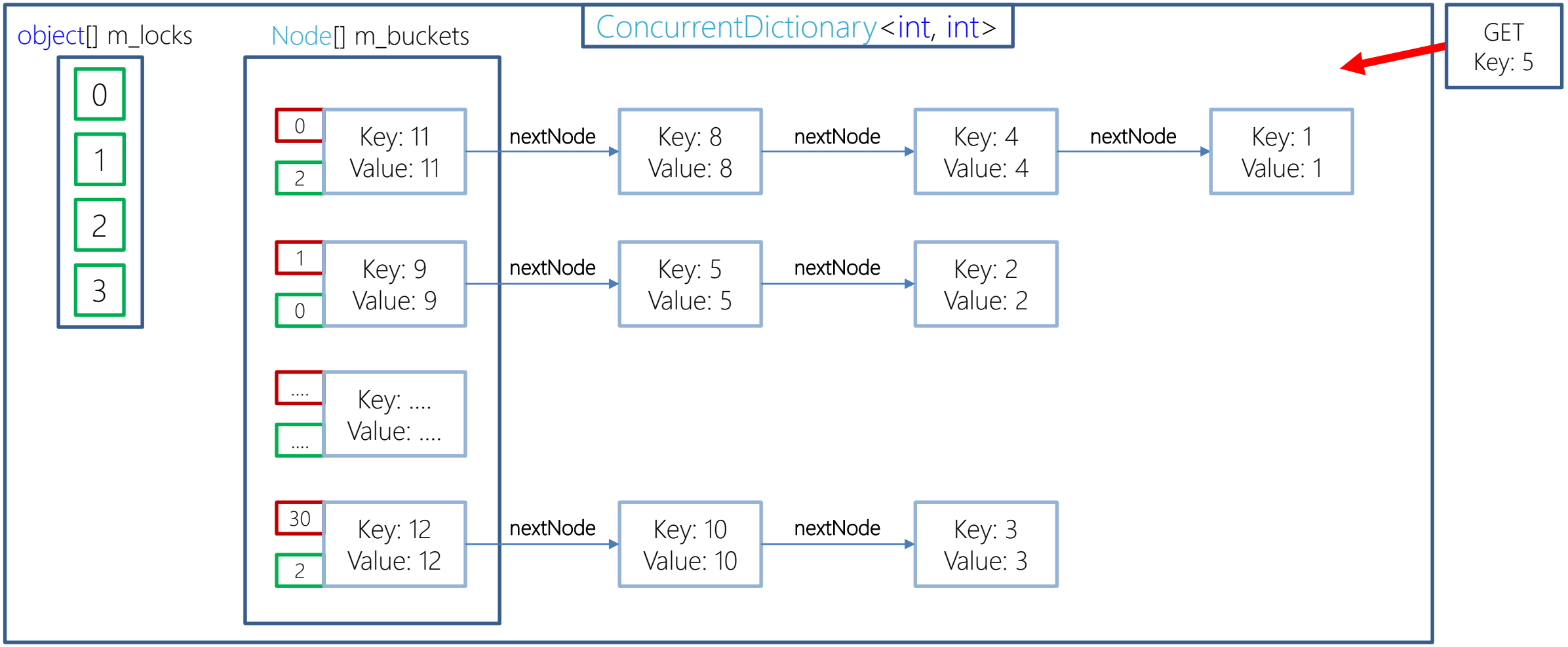
Хранение элементов в ConcurrentDictionary



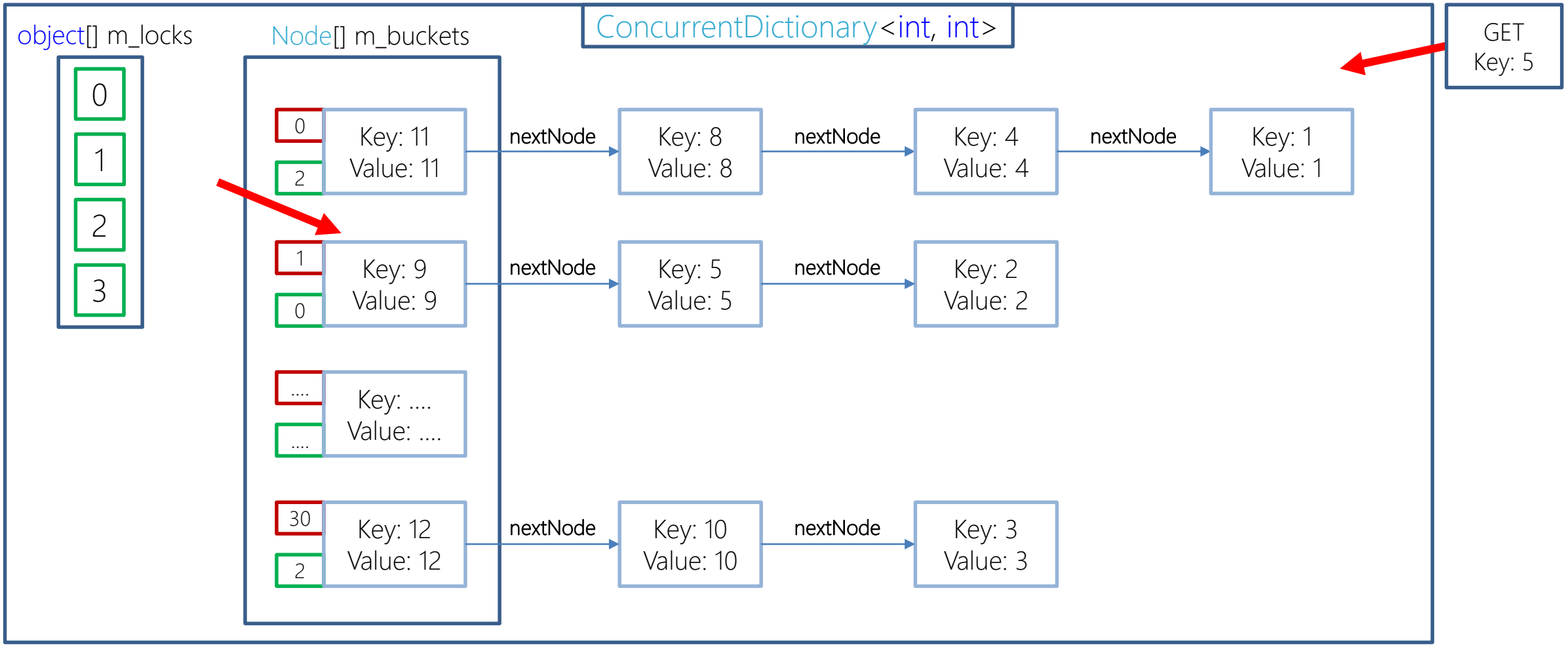
Получение значения из ConcurrentDictionary



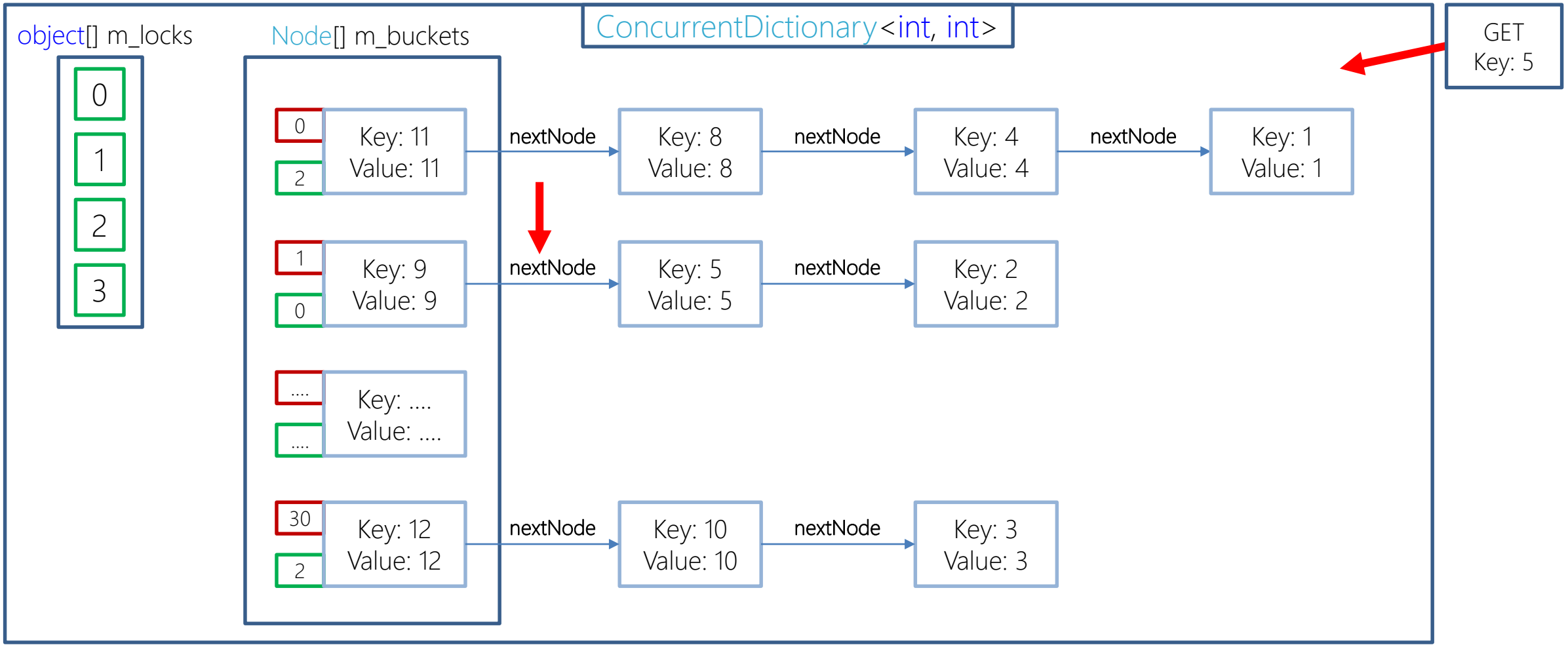
Получение значения из ConcurrentDictionary



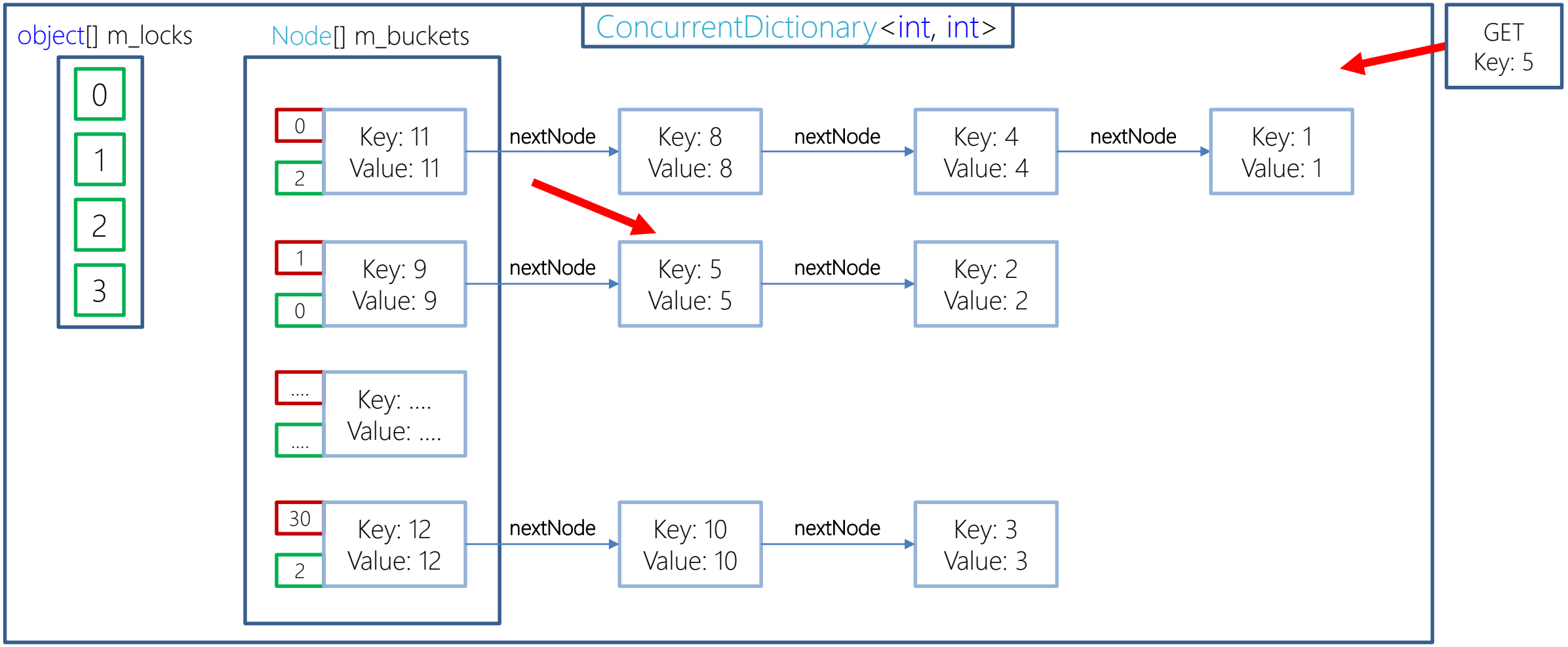
Получение значения из ConcurrentDictionary



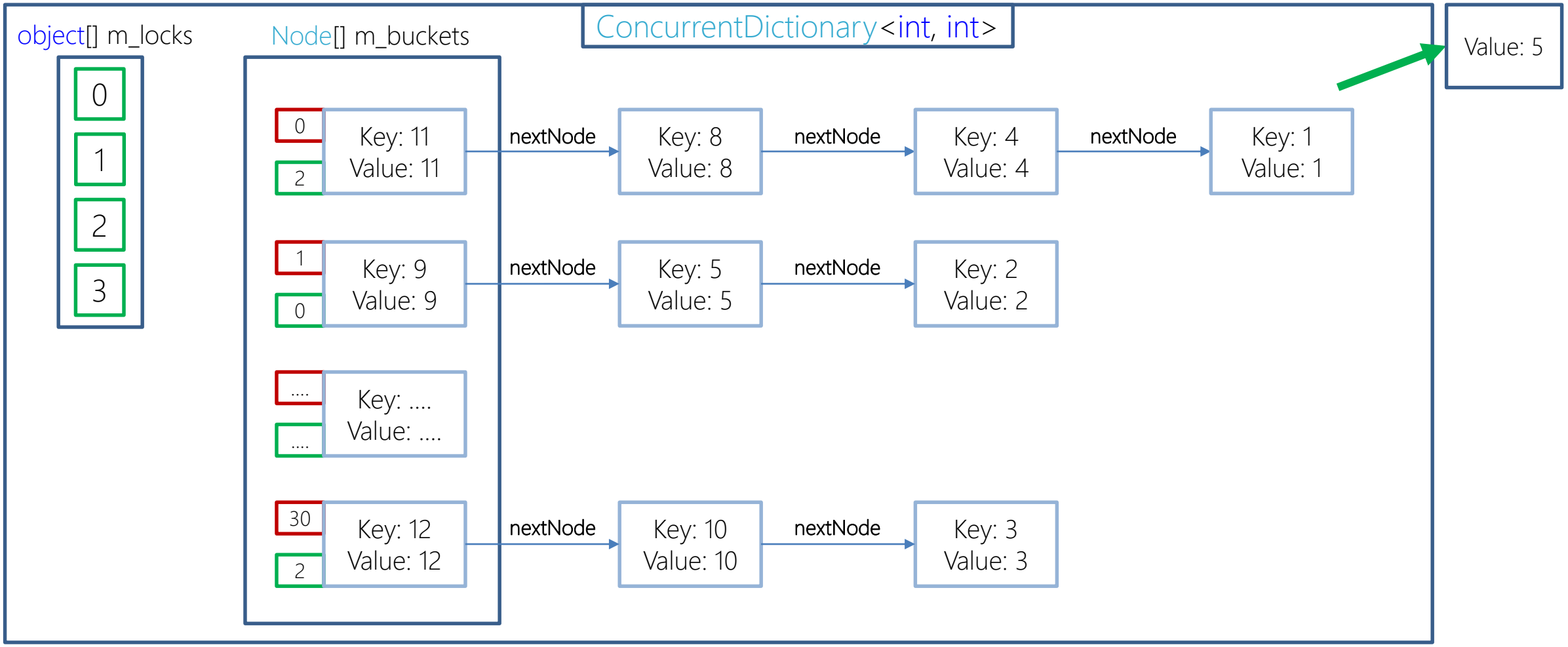
Получение значения из ConcurrentDictionary



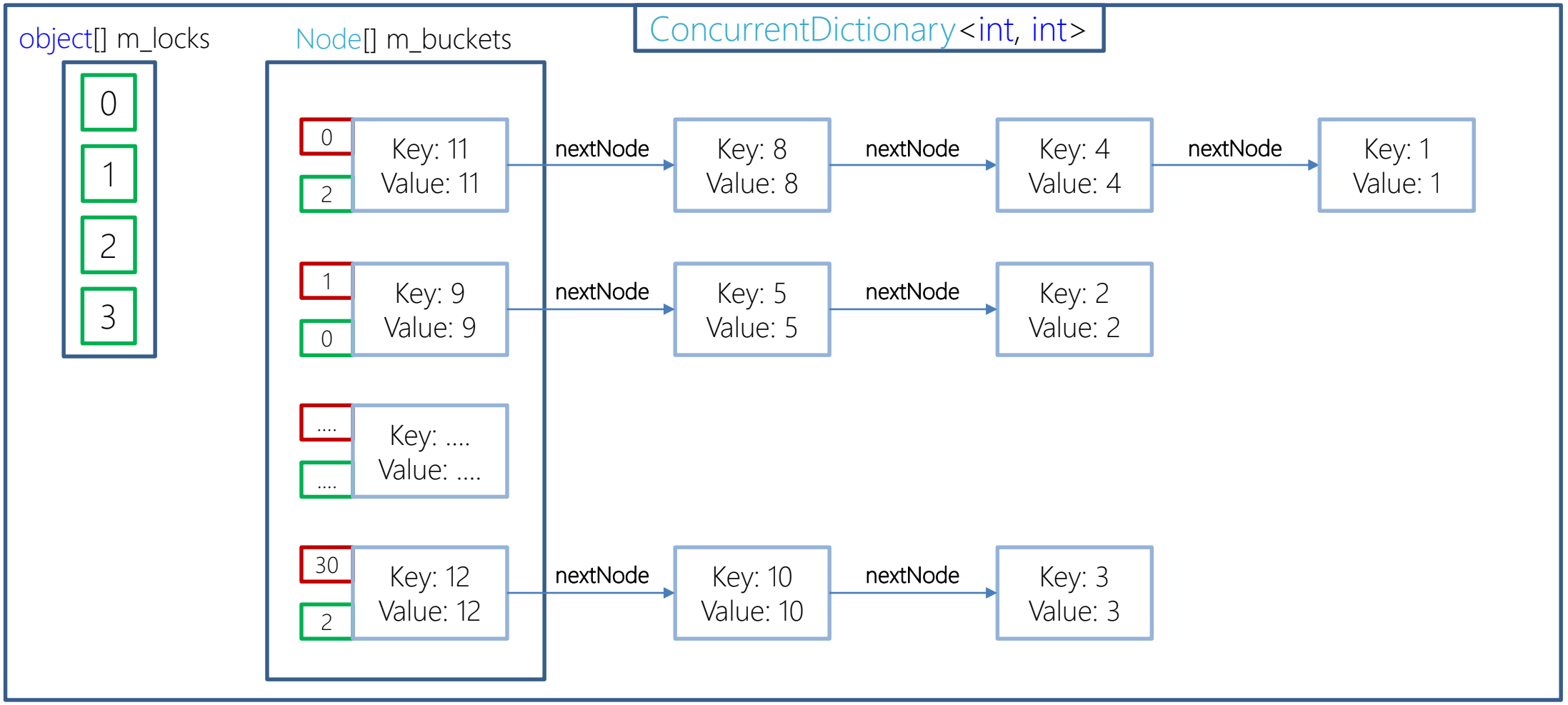
Получение значения из ConcurrentDictionary



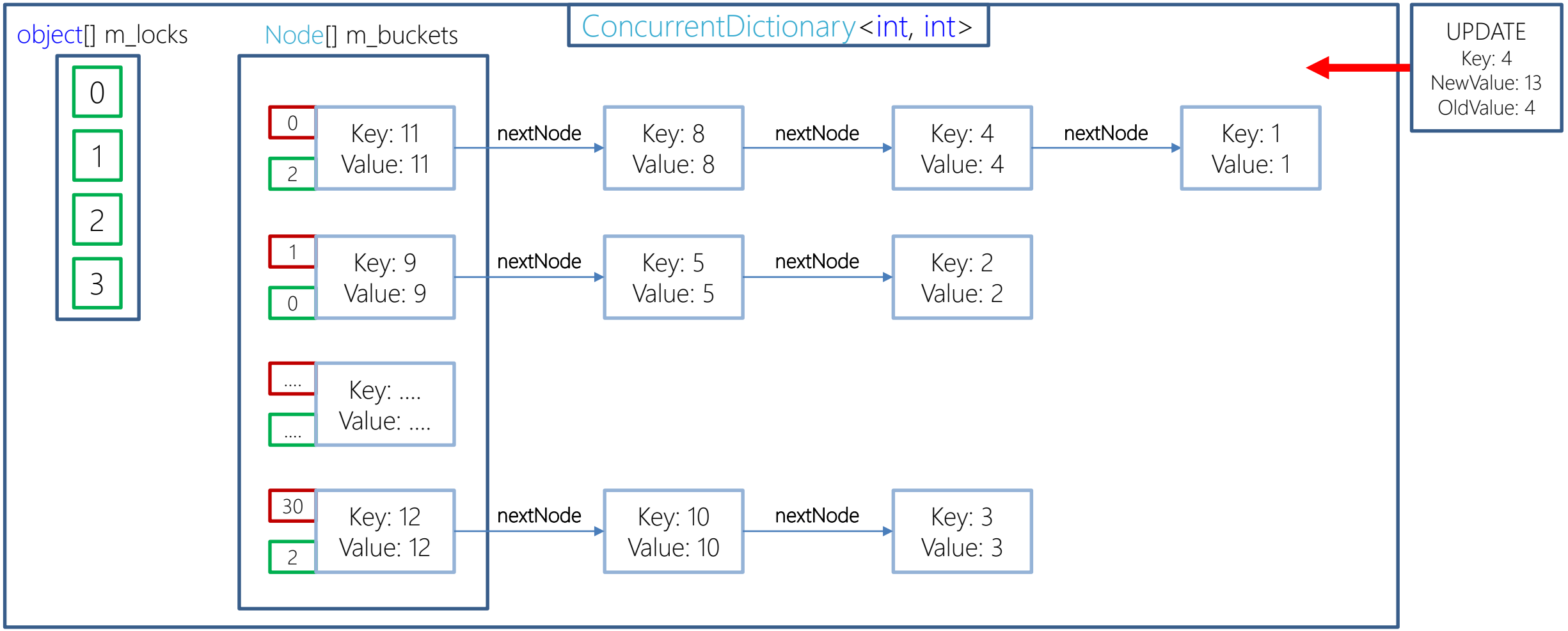
Получение значения из ConcurrentDictionary



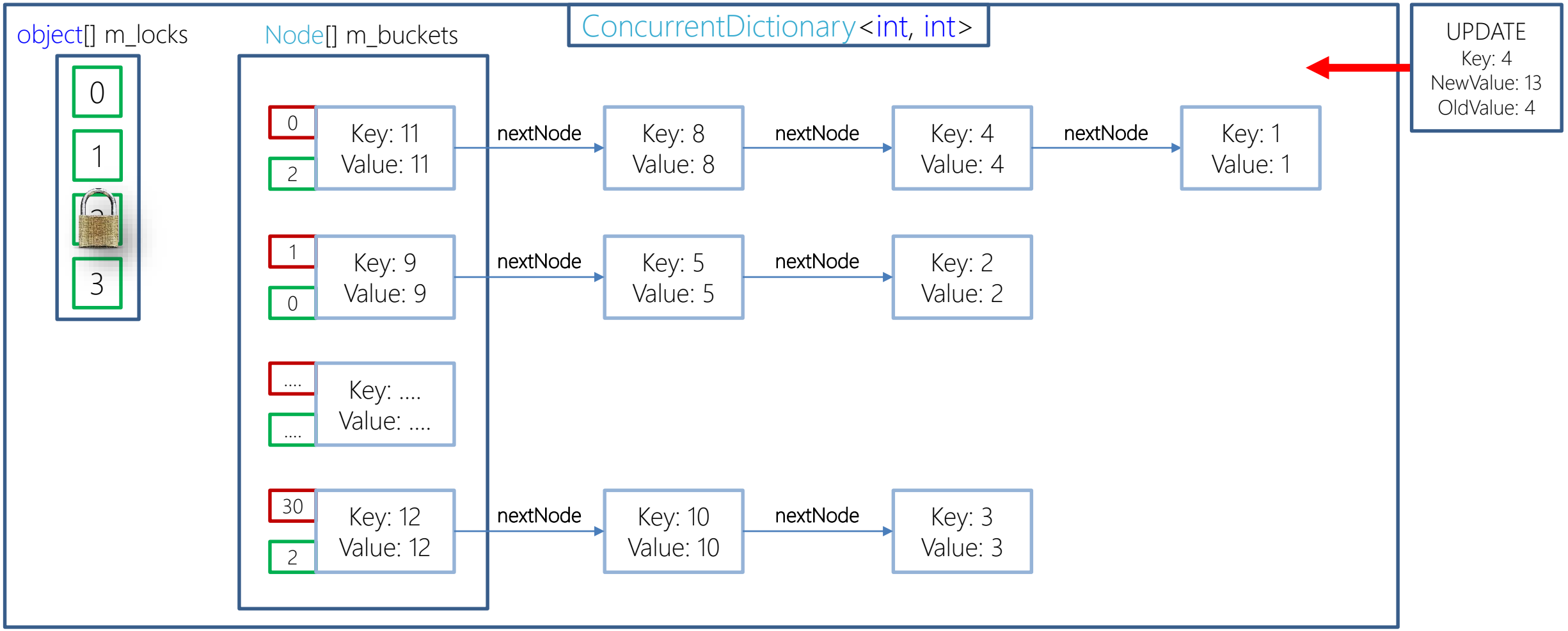
Изменение значения в ConcurrentDictionary



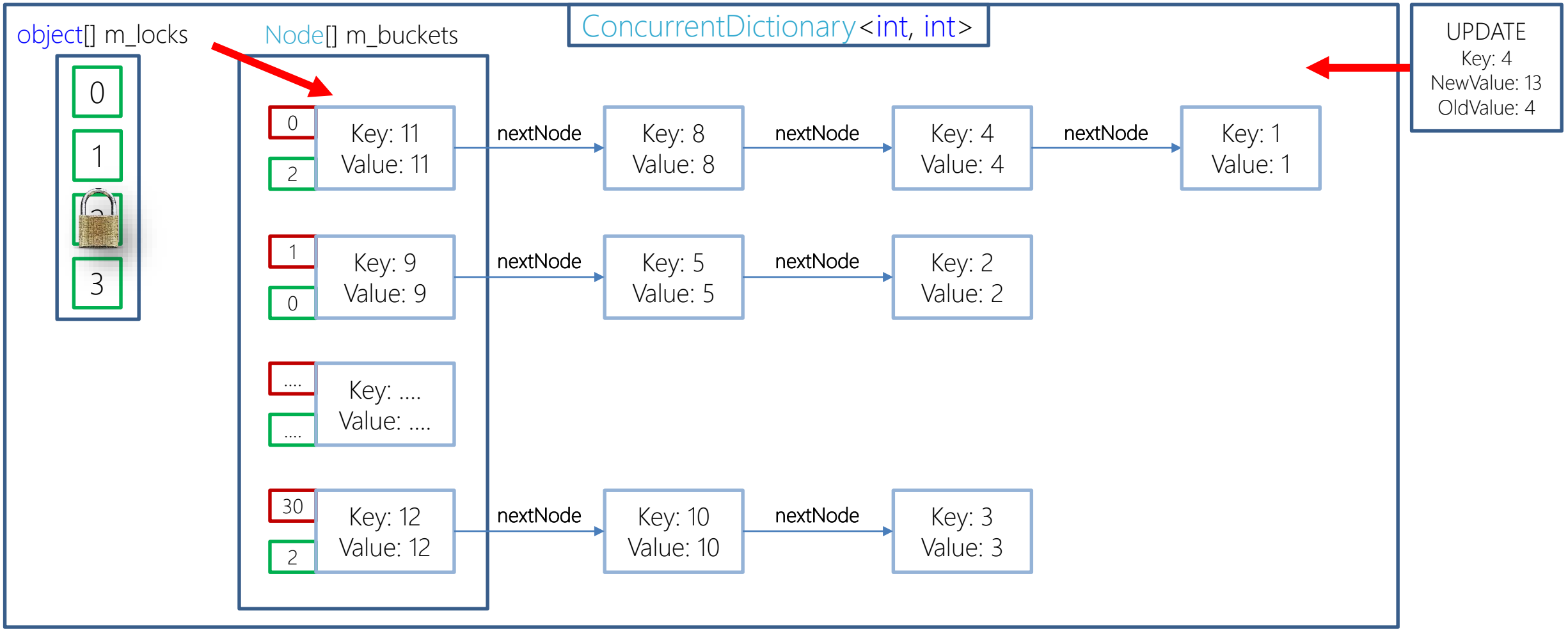
Изменение значения в ConcurrentDictionary



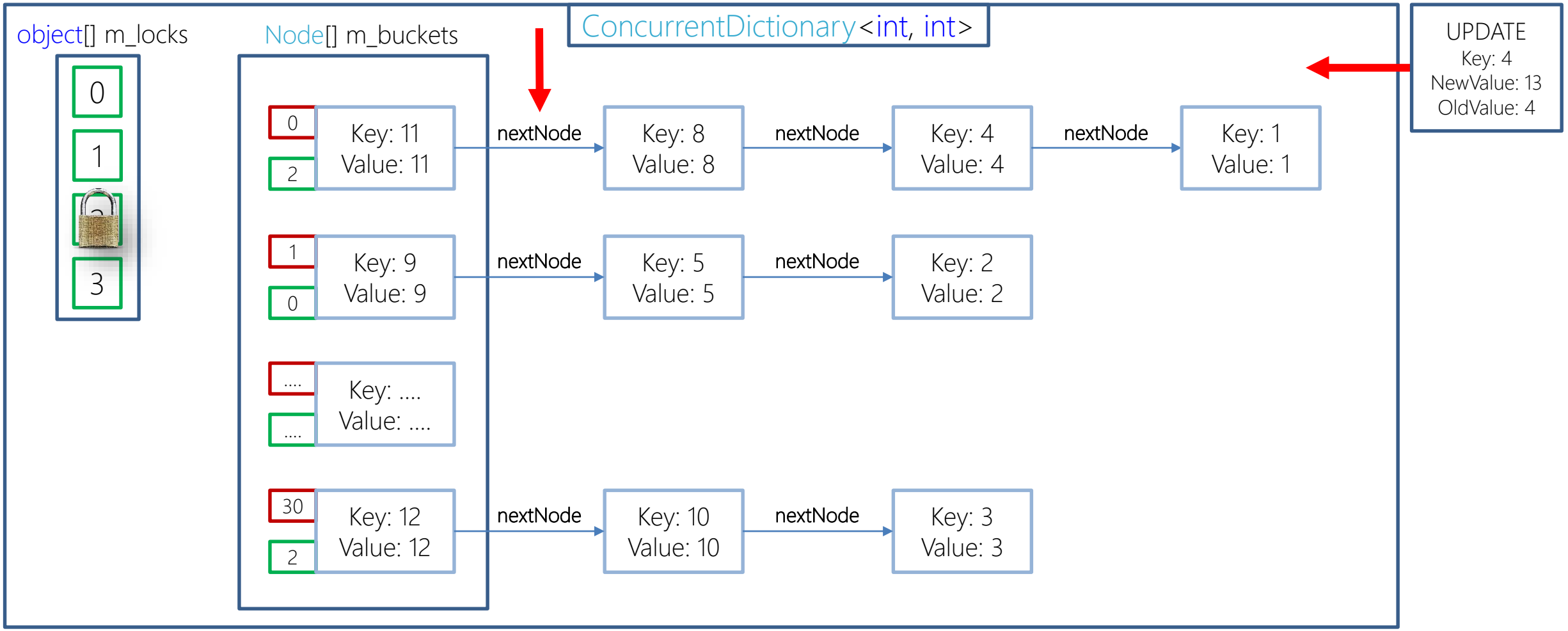
Изменение значения в ConcurrentDictionary



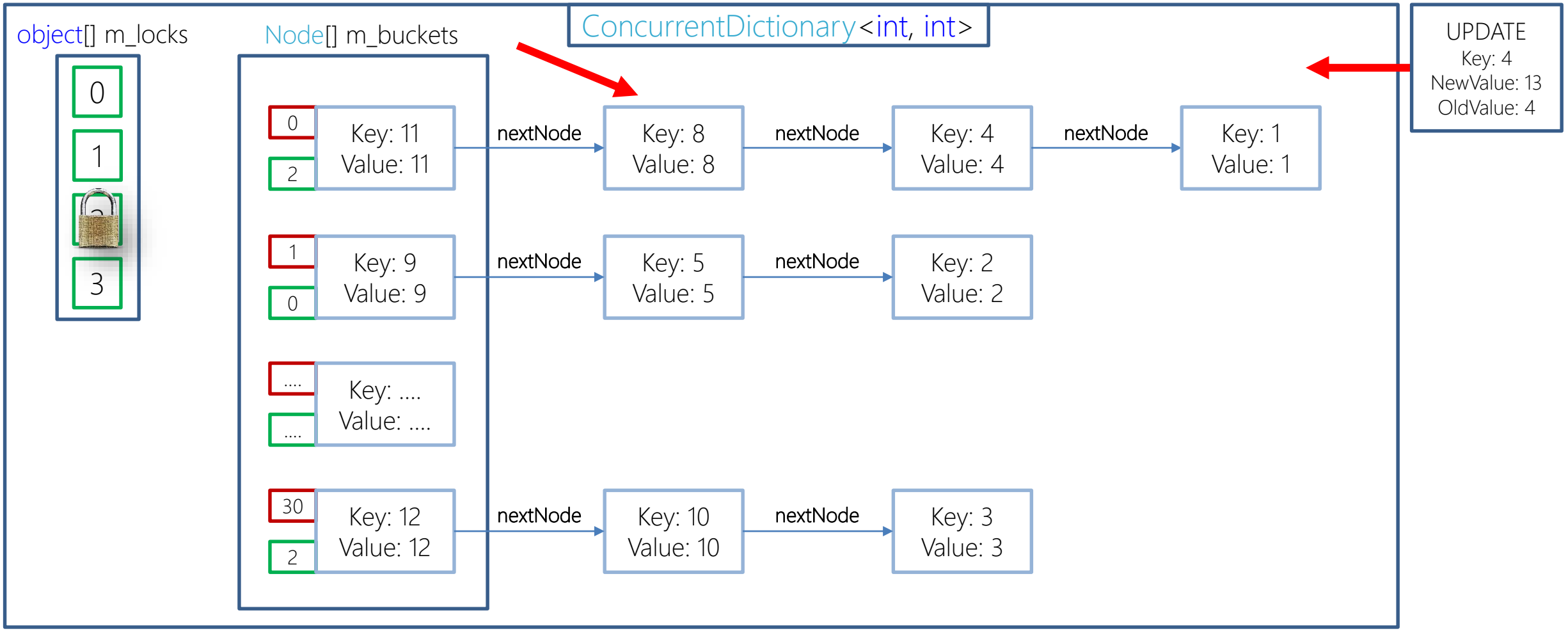
Изменение значения в ConcurrentDictionary



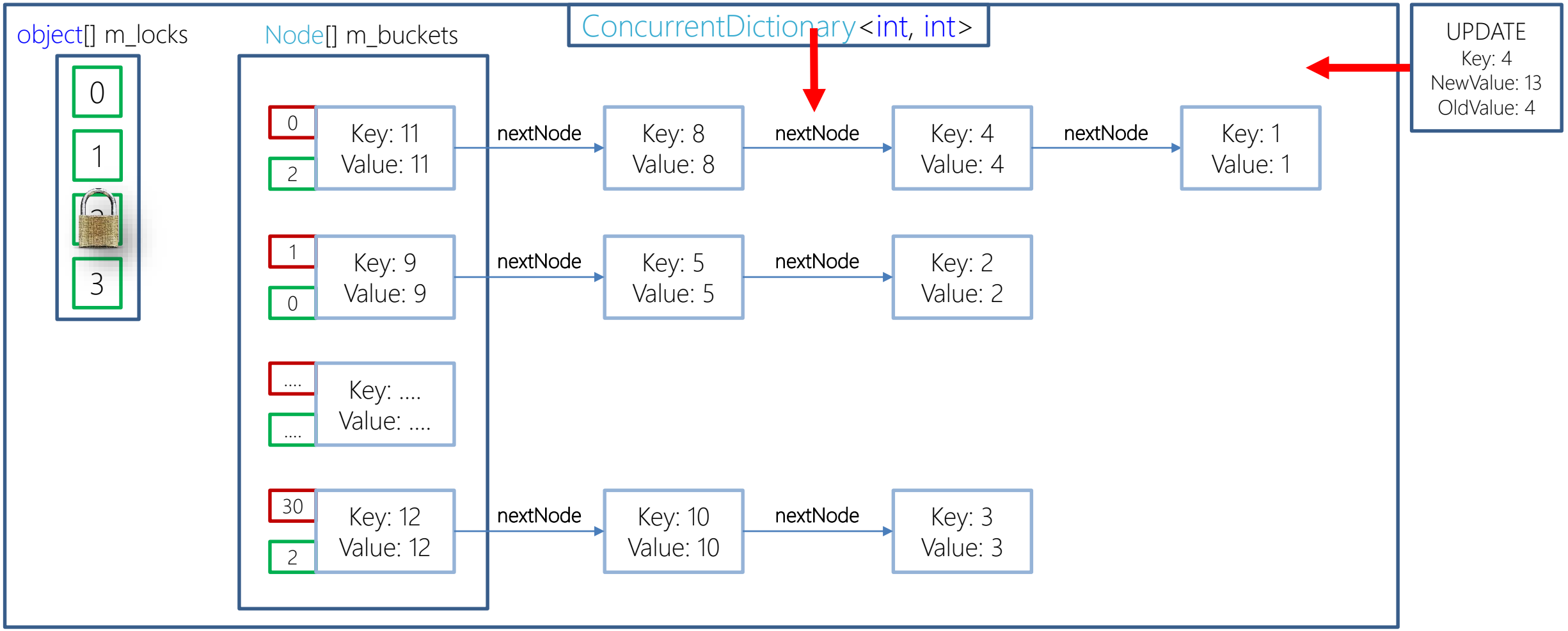
Изменение значения в ConcurrentDictionary



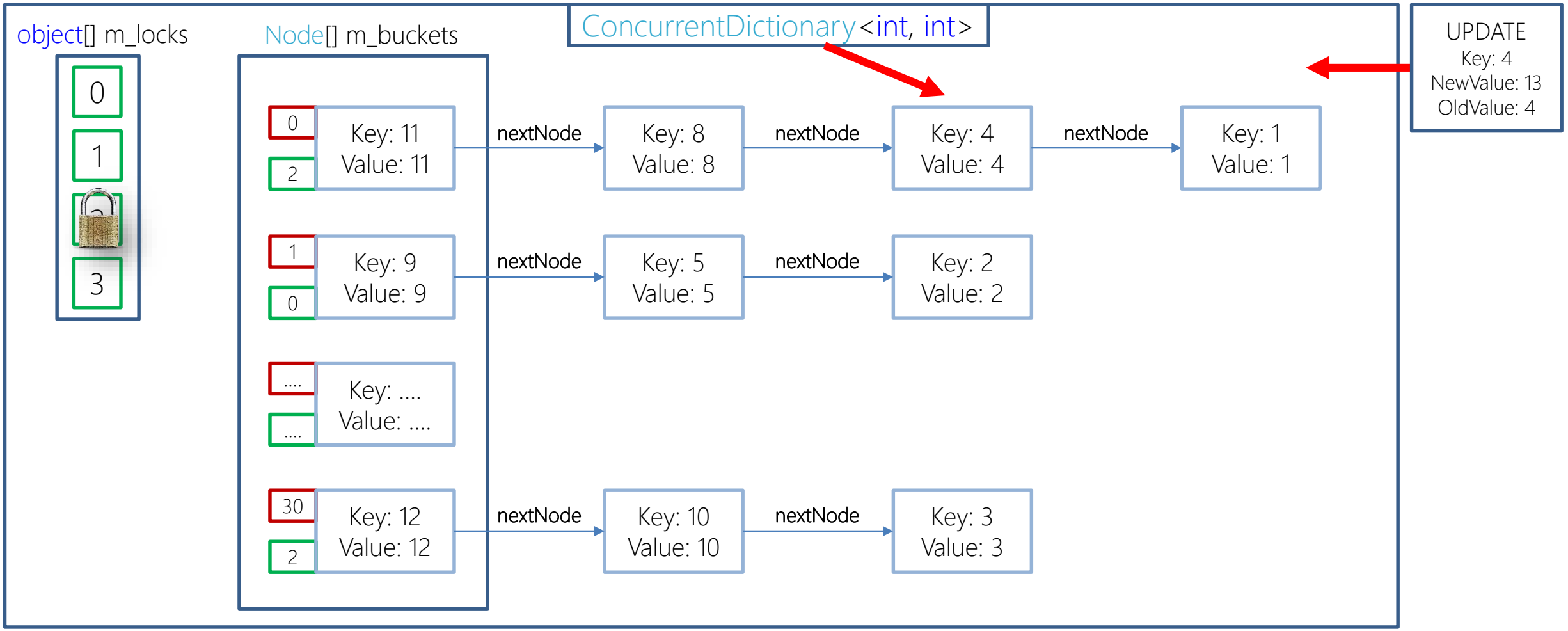
Изменение значения в ConcurrentDictionary



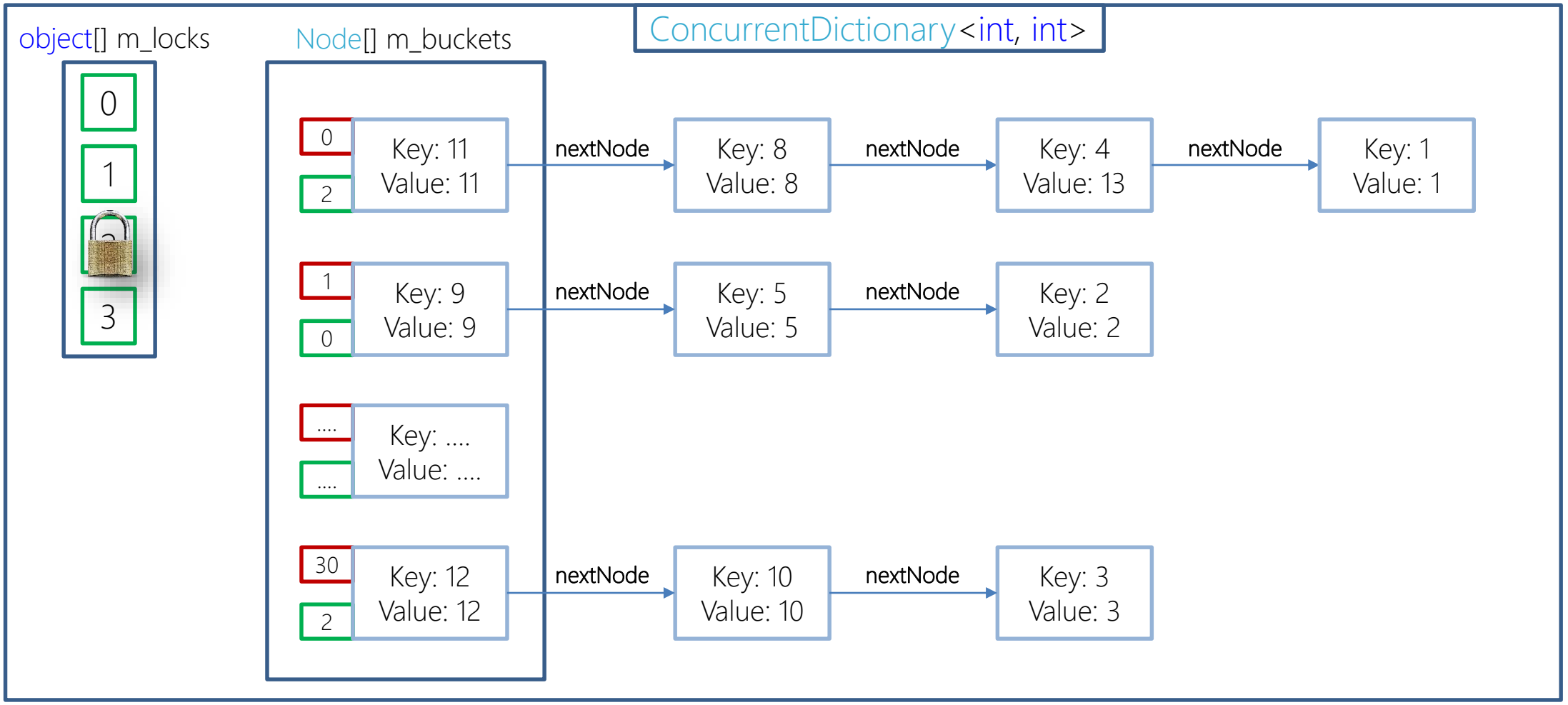
Изменение значения в ConcurrentDictionary



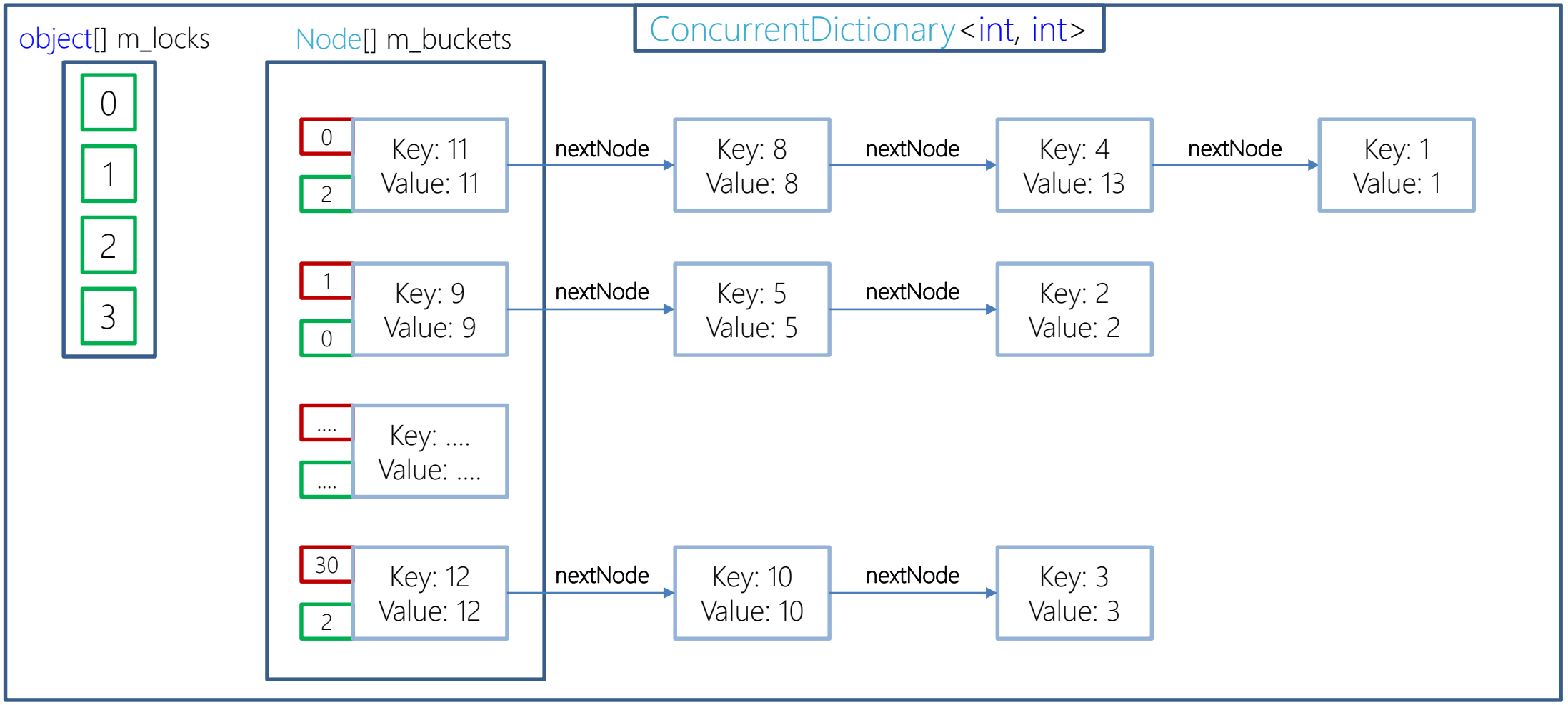
Изменение значения в ConcurrentDictionary



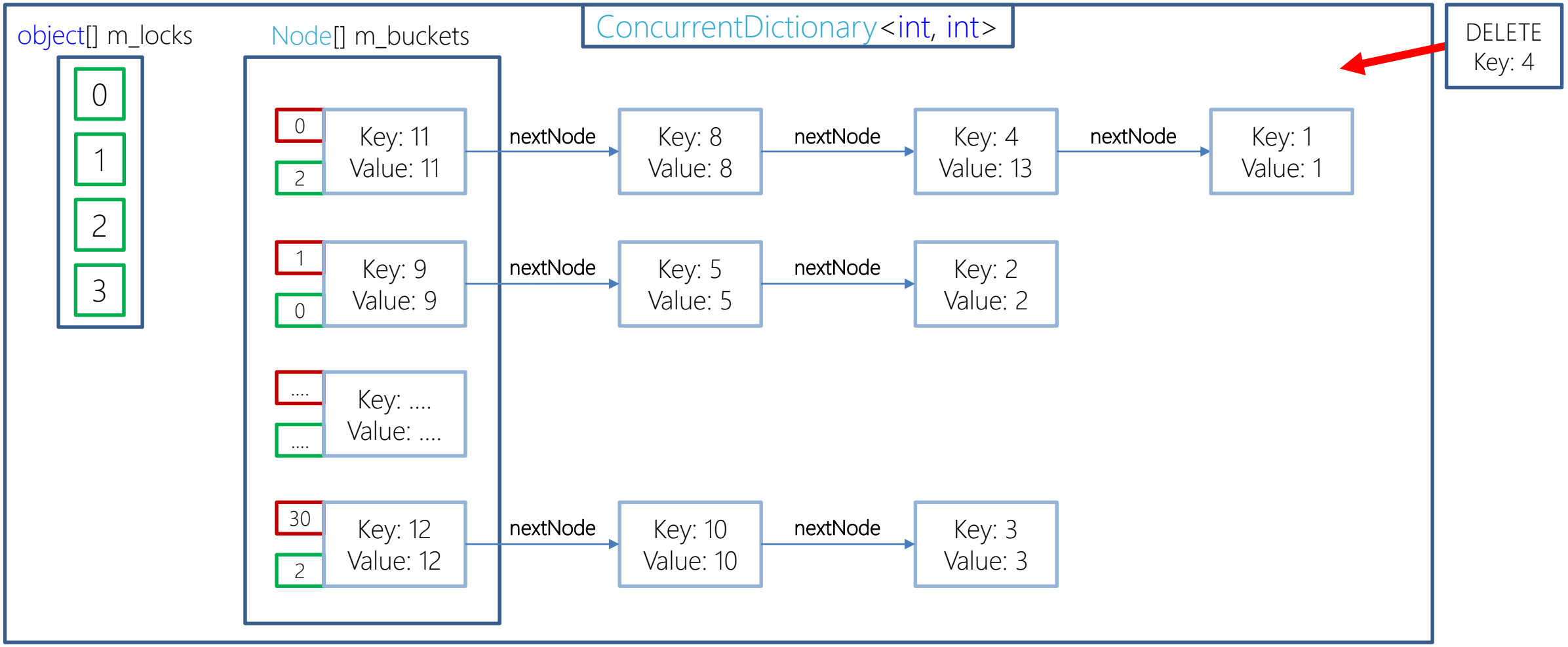
Изменение значения в ConcurrentDictionary



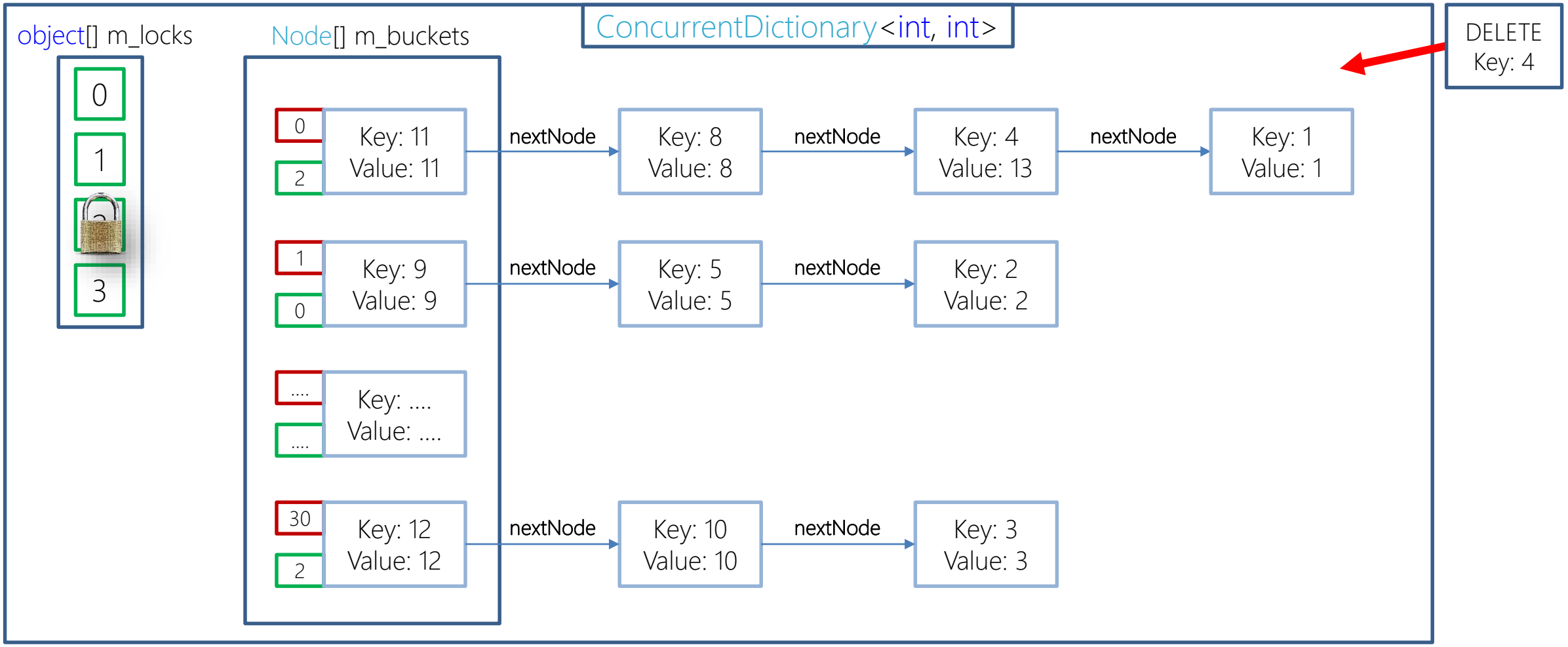
Изменение значения в ConcurrentDictionary



Удаление значения из ConcurrentDictionary

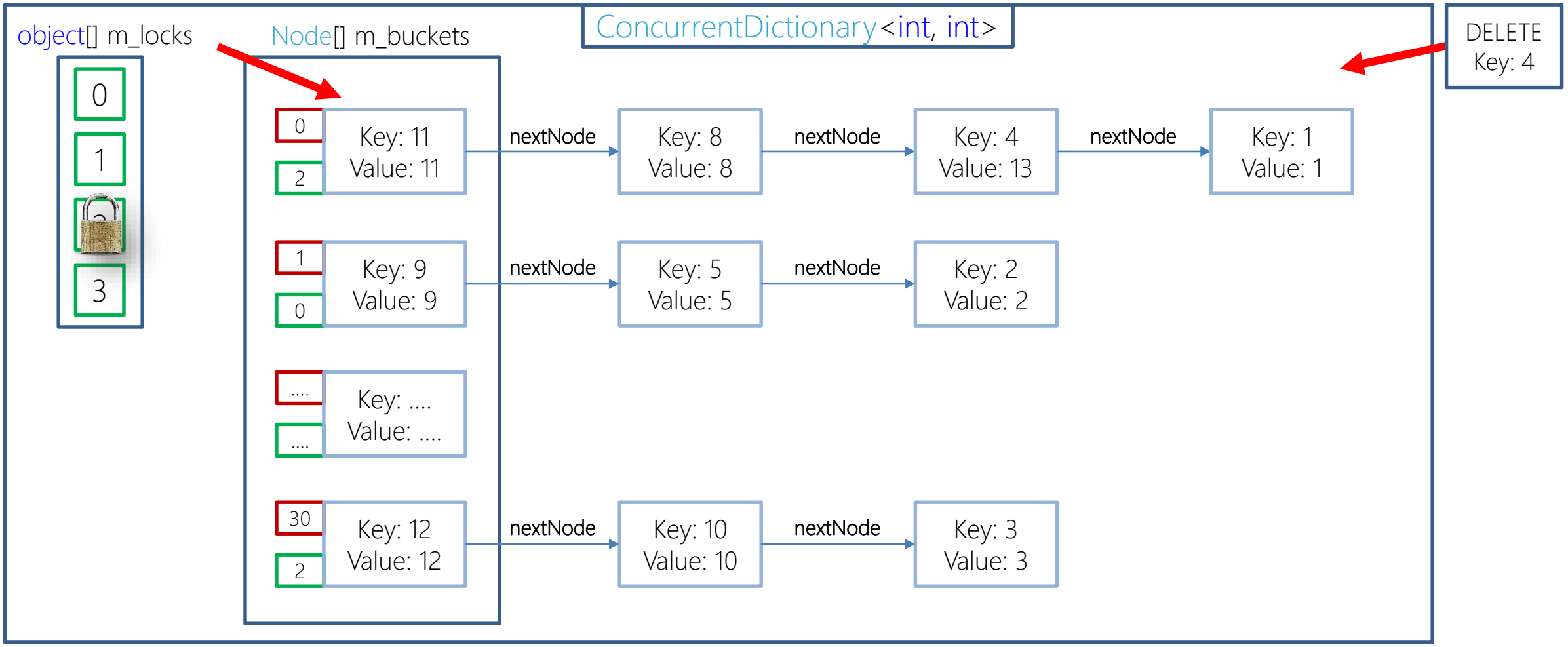


Удаление значения из ConcurrentDictionary

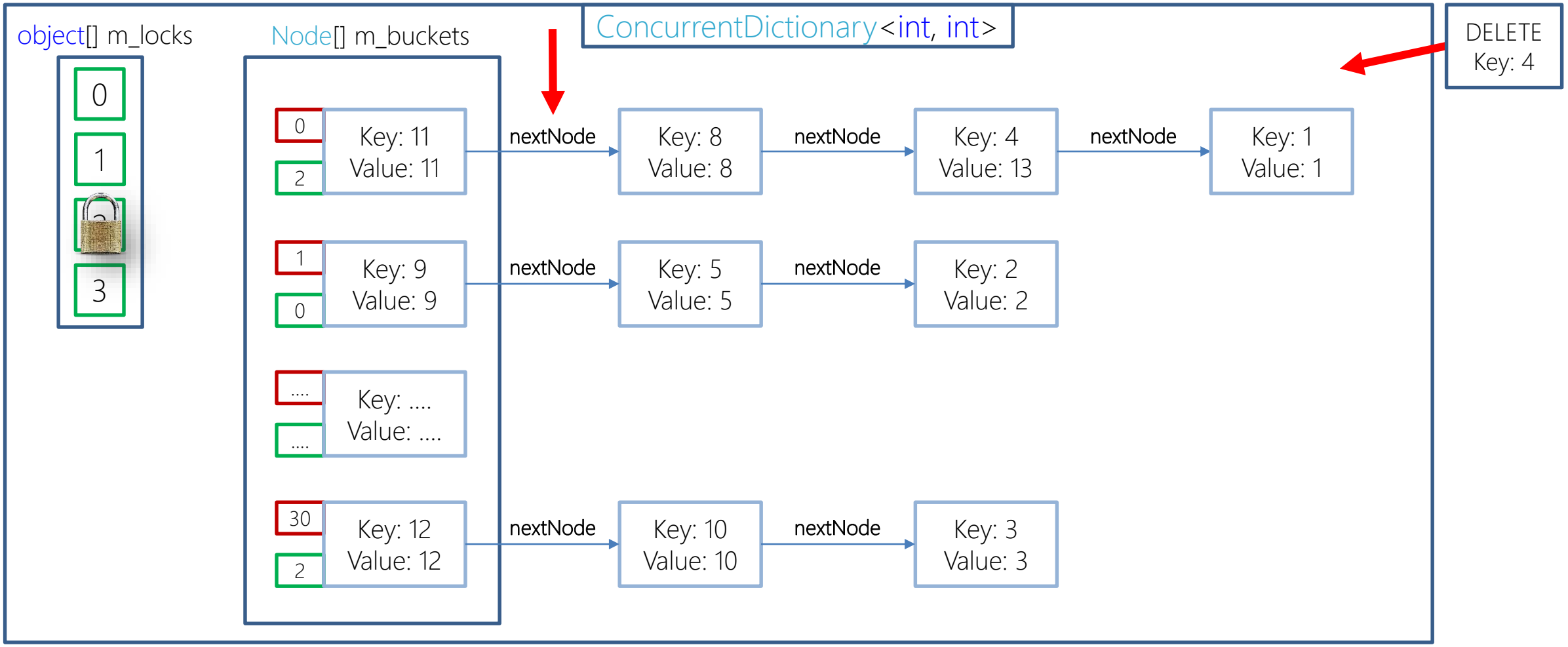


C# Асинхронное программирование

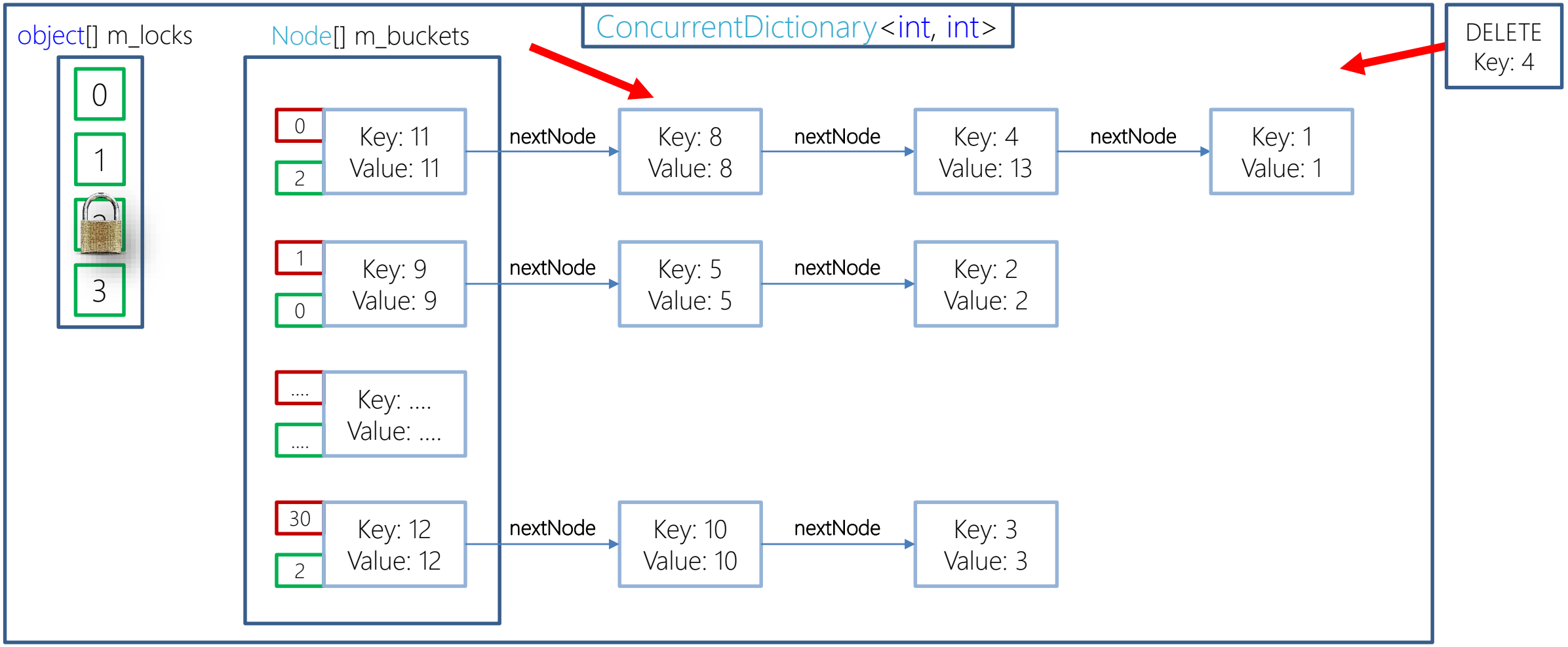
Удаление значения из ConcurrentDictionary



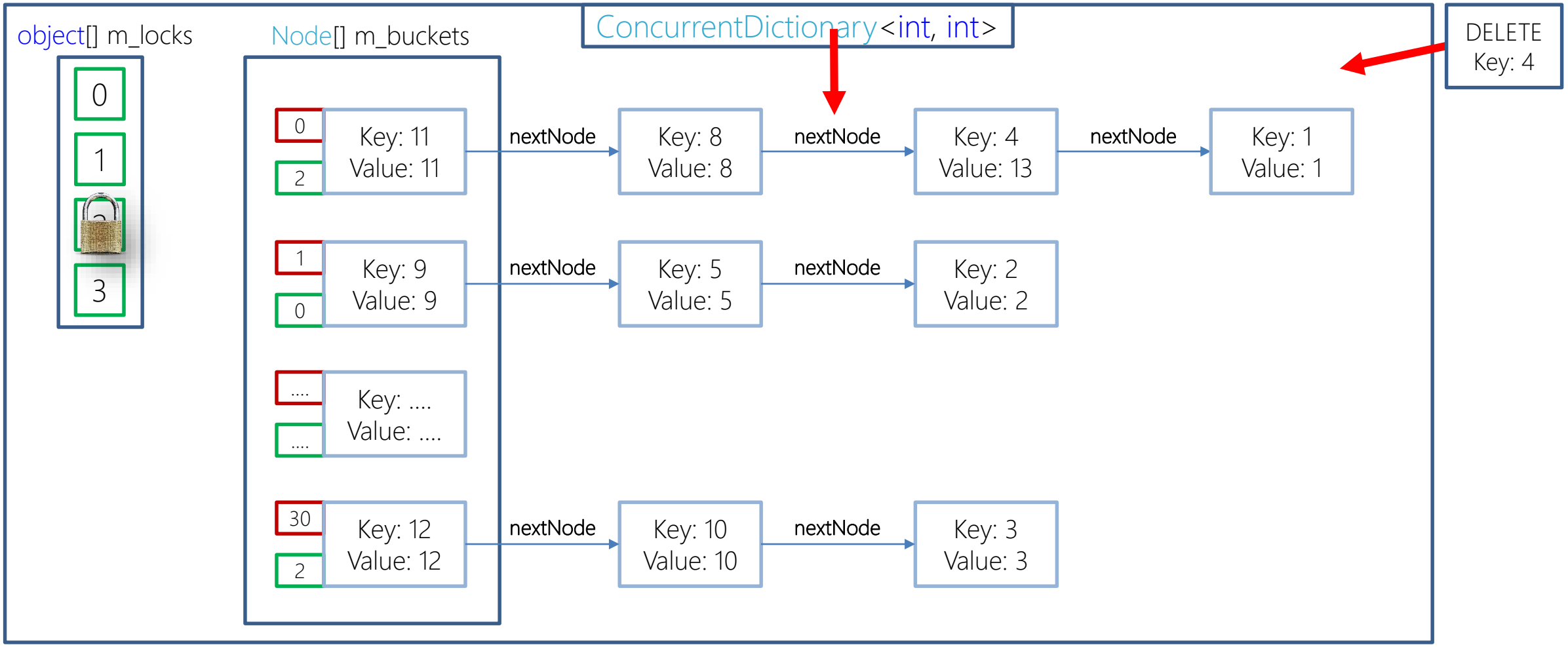
Удаление значения из ConcurrentDictionary



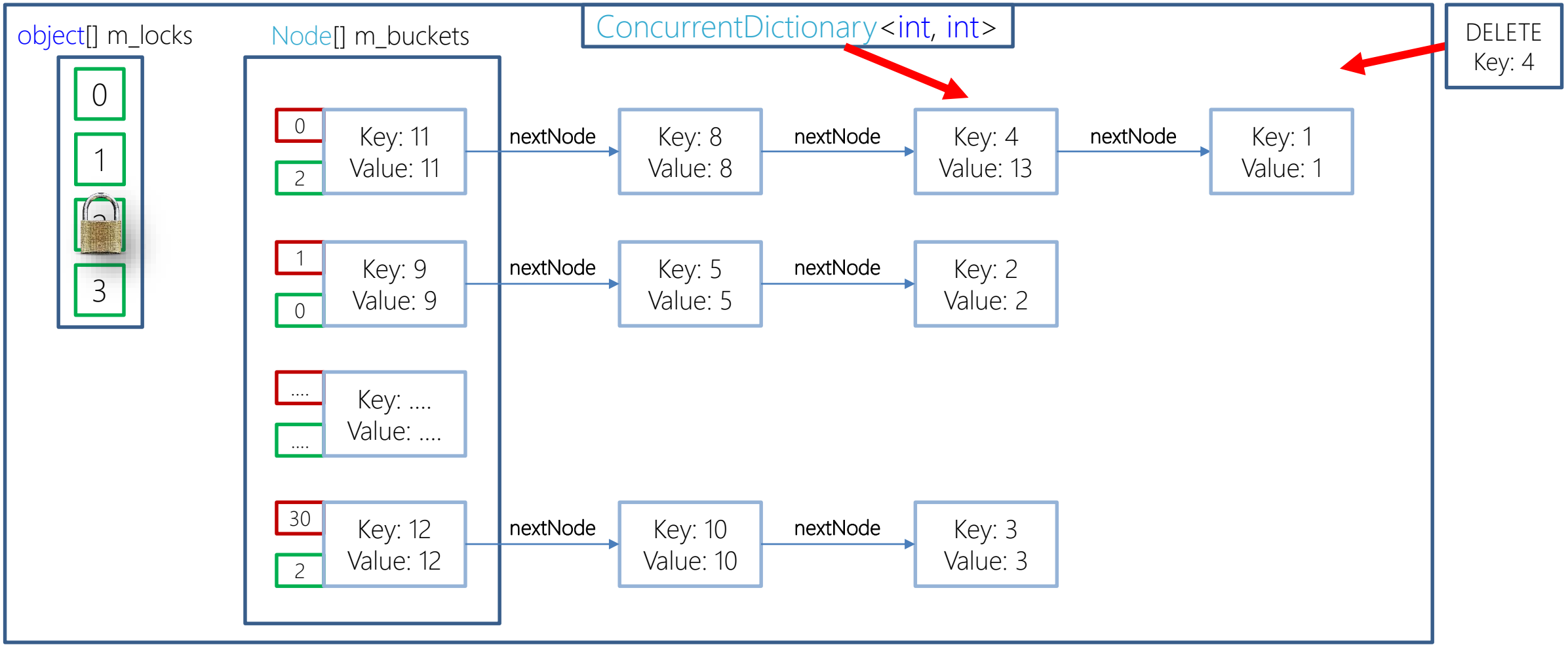
Удаление значения из ConcurrentDictionary



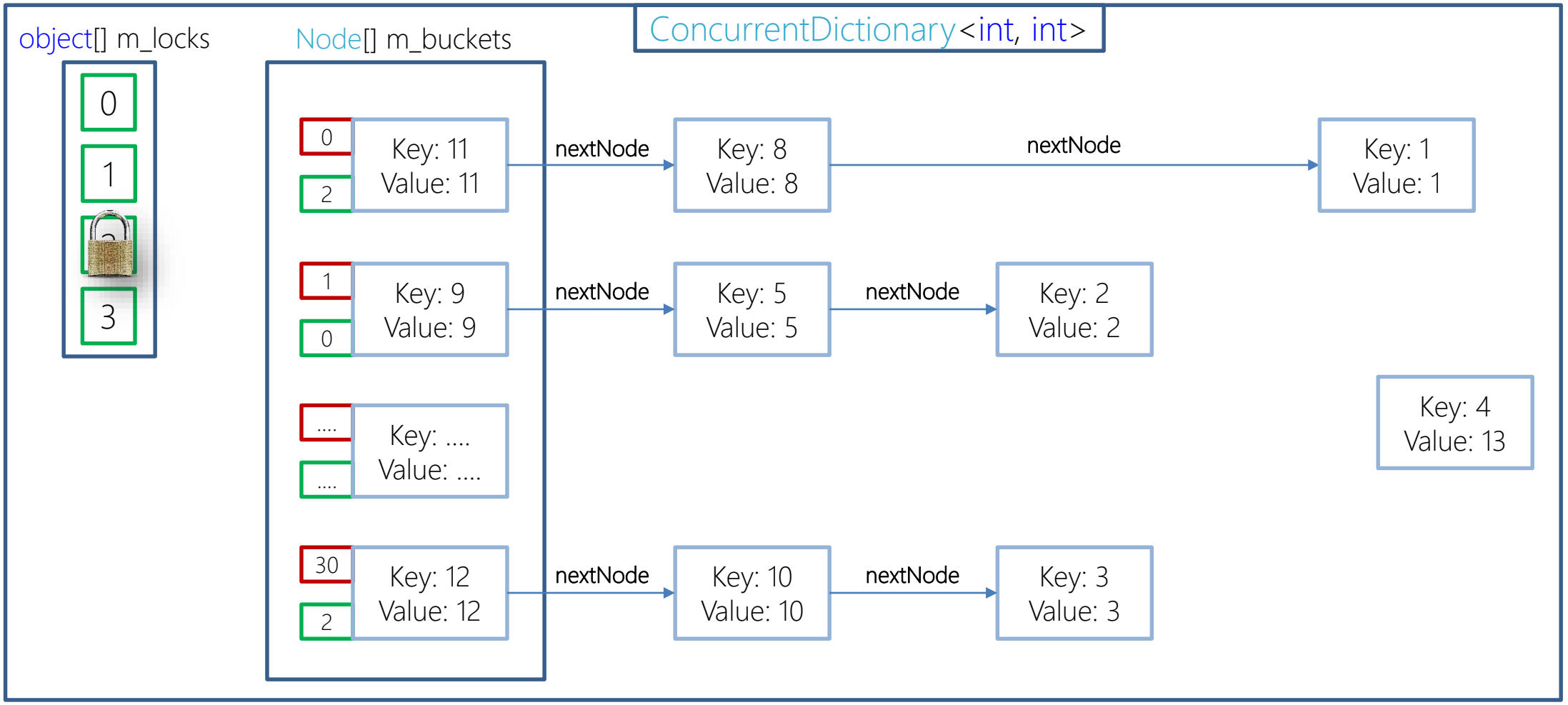
Удаление значения из ConcurrentDictionary



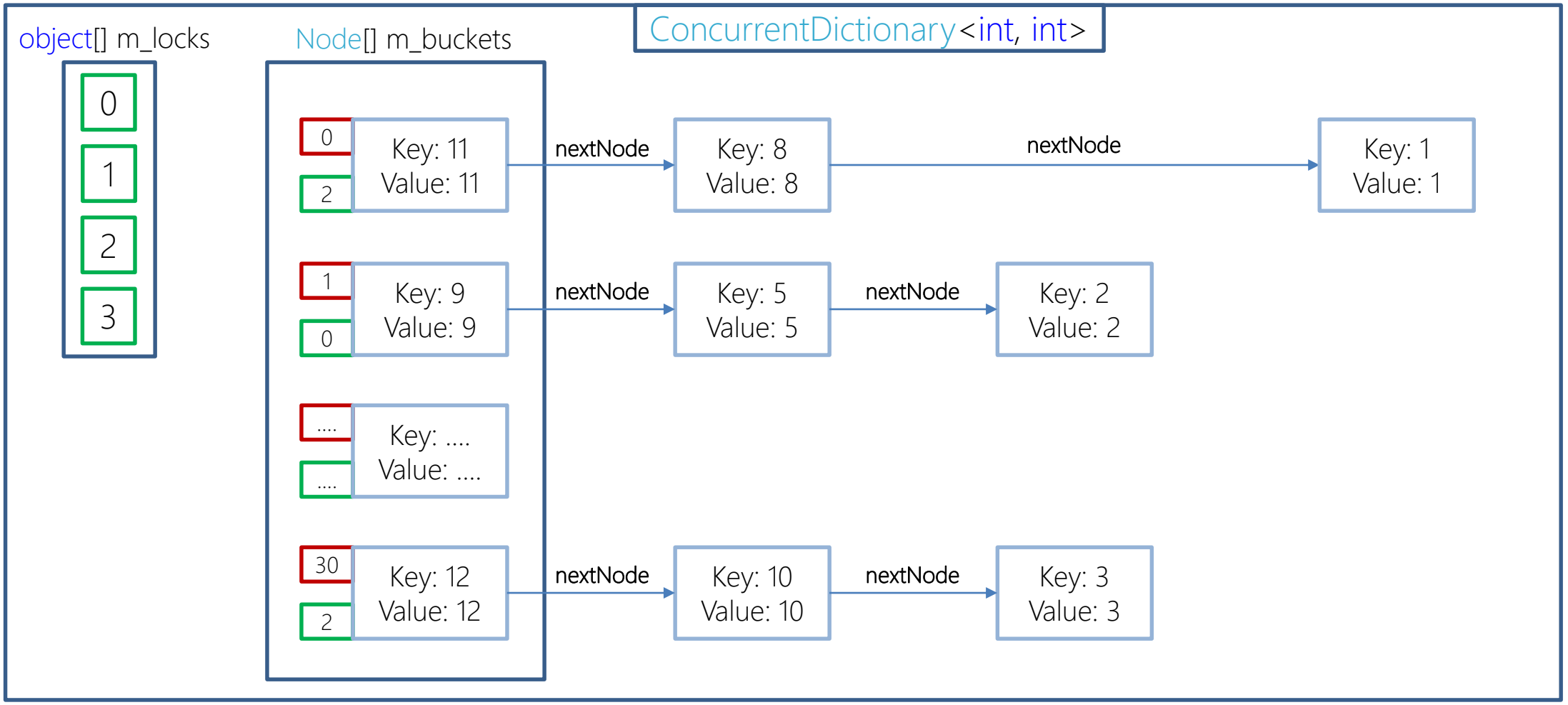
Удаление значения из ConcurrentDictionary



Удаление значения из ConcurrentDictionary



Удаление значения из ConcurrentDictionary



Настройка экземпляра ConcurrentDictionary

При создании, вы можете сконфигурировать экземпляр ConcurrentDictionary с помощью выбора необходимой перегрузки конструктора.

Перегрузки:

```
public ConcurrentDictionary();
public ConcurrentDictionary(IEnumerable<KeyValuePair<TKey, TValue>> collection);
public ConcurrentDictionary(IEqualityComparer<TKey> comparer);
public ConcurrentDictionary(int concurrencyLevel, int capacity);
public ConcurrentDictionary(IEnumerable<KeyValuePair<TKey, TValue>> collection, IEqualityComparer<TKey> comparer);
public ConcurrentDictionary(int concurrencyLevel, IEnumerable<KeyValuePair<TKey, TValue>> collection, IEqualityComparer<TKey> comparer);
public ConcurrentDictionary(int concurrencyLevel, int capacity, IEqualityComparer<TKey> comparer);
```

Параметры:

- IEnumerable<KeyValuePair<TKey, TValue>> – позволяет создать коллекцию ConcurrentDictionary на основе другого словаря, передав из него элементы.
- IEqualityComparer<TKey> – позволяет указать свой тип, который сравнивает ключи необходимым вам образом.
- int concurrencyLevel – позволяет указать уровень параллелизма. Он влияет на начальное количество объектов для блокировки.
- int capacity – позволяет указать начальное количество элементов словаря. Он влияет на начальное количество создаваемых bucket-ов.

Способы получения значения из ConcurrentDictionary

- **Индексатор** – если есть 100% уверенность, что элемент есть в коллекции.

```
double value = movieRating[ “The Dark Knight“ ];
```

- Метод **TryGetValue** – если нет уверенности, что элемент есть в коллекции, при этом вы не хотите добавлять его при отсутствии.

```
bool isSuccess = movieRating.TruGetValue(“The Dark Knight”, out double value);
```

- Метод **GetOrAdd** – если нет уверенности, что элемент есть в коллекции, при этом вы хотите добавлять его при отсутствии.

```
double value = movieRating.GetOrAdd( “The Dark Knight” , 0);
```

Параллельное программирование

.NET предлагает вам несколько стандартных решений для распараллеливания кода:

- Типы System.Threading.Tasks.Task и System.Threading.Tasks.Task<TResult>
- Тип System.Threading.Tasks.Parallel
- Параллельный LINQ (PLINQ) в виде типов System.Linq.ParallelEnumerable и System.Linq.ParallelQuery<T>.

PLINQ

Parallel LINQ (PLINQ) – это параллельная реализация LINQ to Objects. Он имеет полный набор стандартных операторов запроса LINQ в виде методов расширений, а также дополнительные операторы для параллельных операций. Операции PLINQ представляет класс ParallelEnumerable

PLINQ прост в использовании как обычный LINQ, при этом дополняясь мощью параллельного программирования. Он может значительно увеличить скорость запросов LINQ to Objects за счет более эффективного использования всех доступных ядер на компьютере.

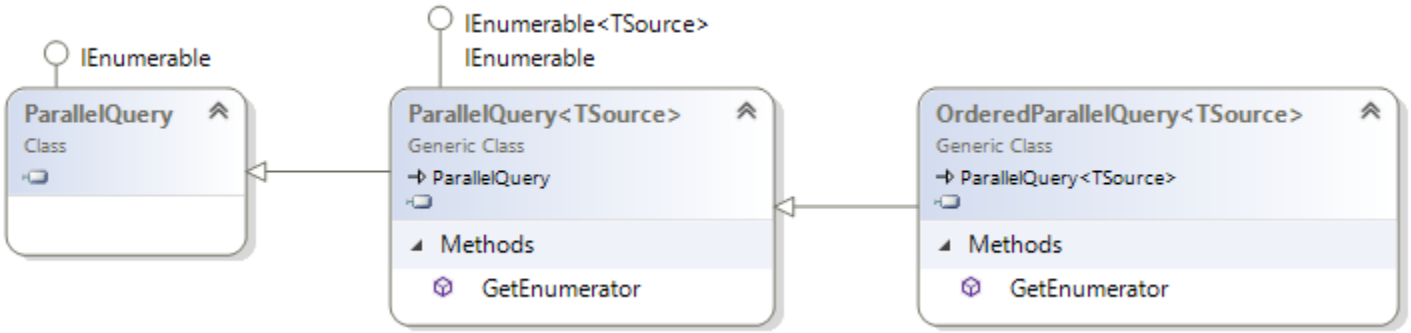
PLINQ можно использовать:

- В синтаксическом варианте
- Расширяющими методами
- Статическими методами

PLINQ

Все расширяющие методы PLINQ находятся в классе ParallelEnumerable. Большая часть расширяющих методов вам будет знакома в виду их наличия в статическом классе Enumerable, который представляет операции LINQ to Objects.

Чтобы вы могли использовать методы PLINQ, ваш источник должен быть типа ParallelQuery<TSource>. Поэтому практически все расширяющие методы класса ParallelEnumerable возвращают именно этот тип.



Операторы PLINQ

- **AsParallel** – метод вызывается на последовательности IEnumerable<T>, чтобы превратить LINQ в PLINQ. Он возвращает ParallelQuery<T>, что позволяет использовать операторы ParallelEnumerable. Чтобы начать работать с PLINQ необходимо вызвать этот метод.

Список операторов, которые присутствуют только в PLINQ:

- **AsOrdered** – указывает, что PLINQ должен сохранять порядок элементов исходной последовательности для остальной части запроса.
- **AsUnordered** – указывает, что PLINQ для остальной части запроса не должен сохранять исходный порядок элементов.
- **AsSequential** – указывает, что вся последующая часть запроса должна вновь выполняться последовательно. Превращает ParallelQuery<T> в IEnumerable<T>.
- **ForAll** – многопоточный метод, который позволяет параллельно обработать результаты запроса указанным вами делегатом, без слияния элементов. Метод ничего не возвращает.

Как работает PLINQ

Работа каждого оператора разделяется на несколько частей, количество которых зависит от указания максимального уровня параллелизма. Каждая часть представляет собой PartitionerStream, который будет вытягивать данные из перебираемой последовательности. Все PartitionerStream-ы будут запущены в контексте задач (Task), что и дает параллельное выполнение.

Во время построения запроса каждый оператор запоминает, кто будет выполняться после него. Таким образом, после завершения параллельной работы одного оператора – работа переходит к другому, который повторяет те же самые действия.

Операторы настройки PLINQ

Список операторов для корректировки работы PLINQ:

- WithCancellation – позволяет передать токен отмены (CancellationToken), чтобы у вас была возможность отменить выполнение параллельной обработки запроса.
- WithDegreeOfParallelism – позволяет указать максимальное число процессоров, которое PLINQ может использовать для параллельной обработки. Но, PLINQ сам выбирает, сколько ему использовать процессоров для обработки. Вы можете указать только граничное значение.
- WithExecutionMode – позволяет заставить PLINQ выполняться параллельно в 100% случаев, в отличие от поведения по умолчанию, где PLINQ пытается понять, стоит ему выполнять запрос параллельно или последовательно.
- WithMergeOptions – указывает, как PLINQ должен объединить результаты из различных разделов обратно в одну последовательность.

ParallelExecutionMode

Метод **WithExecutionMode**(ParallelExecutionMode **executionMode**) позволяет вам указать режим выполнения запроса для PLINQ.

Перечисление ParallelExecutionMode имеет следующие константы:

Default (Значение по умолчанию) – PLINQ проверит структуру запроса и выберет распараллеливание только в том случае, если ему покажется, что это приведет к ускорению обработки.

ForceParallelism – явное указание, чтобы PLINQ распараллелил запрос. Необходимо указывать этот флаг, если вы уверены, что параллельное выполнение приведет к ускорению работы.

ParallelMergeOptions

Метод **WithMergeOptions**(MergeOptions mergeOptions) – устанавливает параметры слияния параллельных вычислений запроса.

Перечисление MergeOptions имеет следующие константы:

NotBuffered – требует немедленно возвращать каждый обработанный элемент из каждого потока сразу же после его создания.

AutoBuffered – запрос собирает элементы в буфер, после чего периодически выдает все содержимое буфера потоку-потребителю. При выборе этого флага может понадобиться больше времени, чем при использовании NotBuffered, чтобы передать первый элемент в поток-потребитель.

FullyBuffered – требует собирать все выходные данные от запроса в единый буфер, после чего возвращать его элементы. При выборе этого флага может понадобиться больше всего времени до передачи первого элемента в поток-потребитель, но при этом полные результаты могут быть получены быстрее.

Следует замерять работу после указания флага, чтобы понять действительно ли он ускоряет вам выборку в конкретном запросе.

Исключения в PLINQ

Исключение в одном из параллельных потоков обработчиков приводит к завершению выполнения запроса PLINQ. Все исключения из параллельных обработчиков собираются и помещаются в исключение AggregateException. Оно будет выброшено через участок кода, который инициирует выполнение запроса PLINQ (Например: цикл foreach, ForAll(), ToList(), ToArray(), ToDictionary(), ToLookup()…).

Для обработки исключений PLINQ необходимо помещать вызов инициатора запроса в тело блока try конструкции try-catch.

Блокирование вызывающего потока PLINQ

Если блокирование вызывающего потока при использовании запросов PLINQ для вас неудобно, то вы можете использовать класс Task для решения.

Поместите вызов запроса PLINQ в задачу (например: через метод Task.Run) и воспользуйтесь ключевыми словами async await для неблокирующего ожидания завершения выполнения задачи.

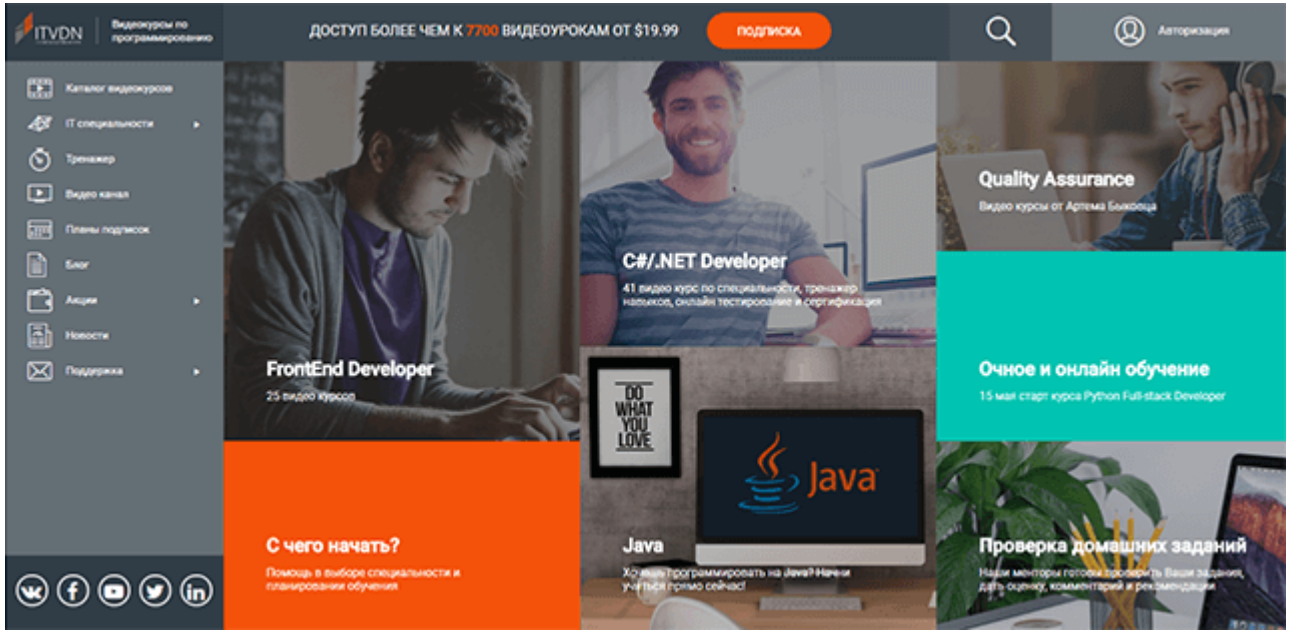
Пример:

```
public async Task OperationAsync<T>(IEnumerable<int> sequence)
{
        await Task.Run(() =>
        {
                var query = from n in sequence
                                .AsParallel()
                            where n % 2 == 0
                            select n;

                var list = query.ToList();
        });
}
```

Q&A

Смотрите наши уроки в видео формате

ITVDN.com



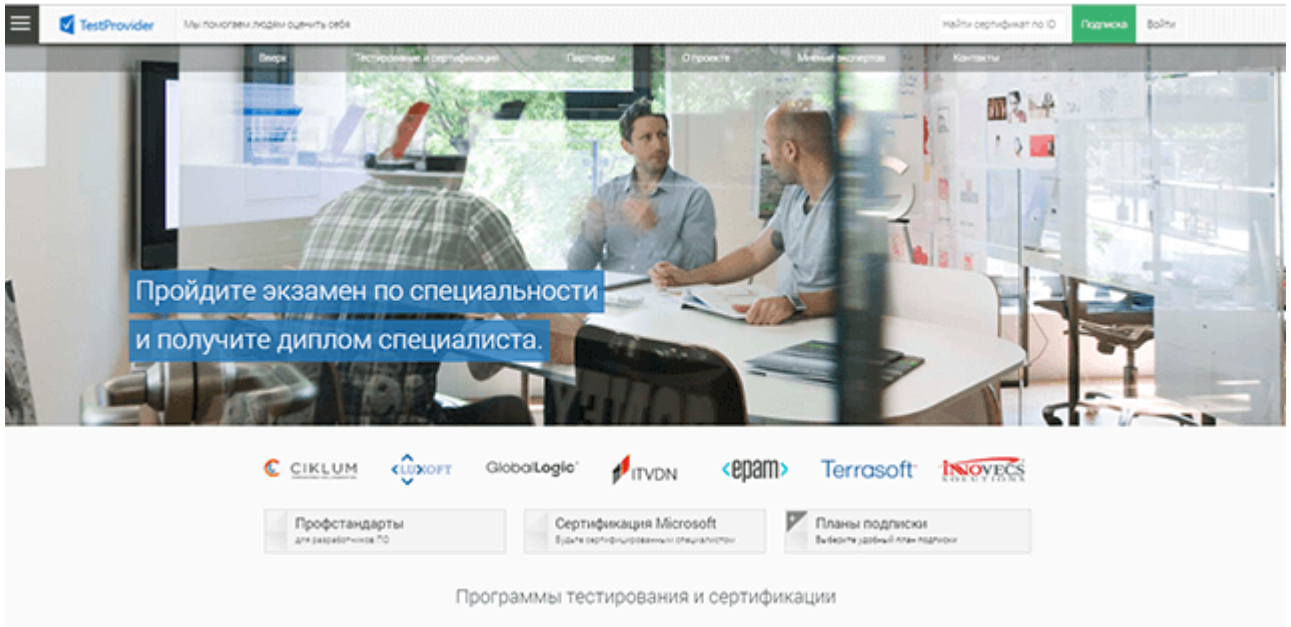
Посмотрите этот урок в видео формате на образовательном портале ITVDN.com для закрепления пройденного материала.

Курсы записаны сертифицированными тренерами, которые работают в учебном центре CyberBionic Systematics и другими высококвалифицированными разработчиками.

ITVDN
IT VIDEO DEVELOPERS NETWORK

Проверка знаний

TestProvider.com



TestProvider – это online сервис проверки знаний по информационным технологиям. С его помощью Вы можете оценить Ваш уровень и выявить слабые места. Он будет полезен как в процессе изучения технологии, так и для общей оценки знаний IT специалиста.

После каждого урока проходите тестирование для проверки знаний на TestProvider.com

Успешное прохождение финального тестирования позволит Вам получить соответствующий Сертификат.

TestProvider

C# Асинхронное программирование

Спасибо за внимание! До новых встреч!

Гнатюк Владислав

MCID:16354168

Информационный видеосервис для разработчиков программного обеспечения

ITVDN

IT VIDEO DEVELOPERS NETWORK

TestProvider PROMETRIC TEST CENTER CyberBionic systematics Microsoft Partner Silver Learning Windows Azure Cloud Partner EBA European Business Association