Лекция 13. Параллельные коллекции.

Цель разработки

- Проблема блокирующих алгоритмов не в их малой производительности.
- Проблема в непредсказуемости времени исполнения.
- Обращение к критической секции может занять 10 тактов, а может и 10000.
- Большее количество потоков - большая конкуренция (contention).
- Требуется выбирать правильный алгоритм под задачу.

Параллельные структуры данных

- Сама структура данных — каким-либо образом связанные участки памяти.
- Алгоритм обслуживания структуры данных.
- **Create** (Insert)
- **Read** (Select, Find)
- **Update**
- **Delete**

Параллельные структуры данных

- Для изолированных данных проблема конкуренции стоит менее остро.
- Параллельные структуры данных обслуживают много потоков.
  - Разделяемые счётчики.
  - Разделяемые контейнеры объектов.
  - Вспомогательные структуры данных.
- Сами алгоритмы — часто модификация последовательных алгоритмов с рядом особенностей.

Параллельные структуры данных

Два основных типа контейнеров:

- Интересующиеся значением объекта или ключа: put(key, item), item = get(key), find(key)
  - Множества (sets)
  - Отображения (maps)
  - Хеш-таблицы (hash-sets)
- Не интересующиеся значением объекта (Пулы): put(item), item = get()
  - Стеки
  - Деки
  - Очереди
- Смешанные: очереди с приоритетами: put(item, priority), get()

Использование структур данных: массивы

- Изменение размера - операция с динамической памятью. Долго.
- Удаление $O(N)$ элемента.
- Максимально быстрый поиск по ключу $O(1)$
- Длинный поиск по содержимому $O(N)$

- Правило 90/9/1: 90% только читателей, 9% иногда модифицируют, 1% создаёт
- Может подходить для множеств.
- Может быть вспомогательной структурой для пулов.

Использование структур данных: (упорядоченный) связный список

- CRUD = $O(N)$
- Нет `alloc`/`free` проблем.
- Есть `hash local` проблемы.
- Подходит для пулов.
- Принцип: объект имеется в множестве, если он достижим из головы списка.
- Хорошо подходит для `optimistic` и `lazy` стратегий.
- Опасен с точки зрения `ABA`.

Использование структур данных: деревья

- (Сбалансированный) деревья поиска. Бинарные кучи.
- `CRUD` $= O(\log N)$
- Хороши для множеств 90/9/1
- `Coarse-grained`

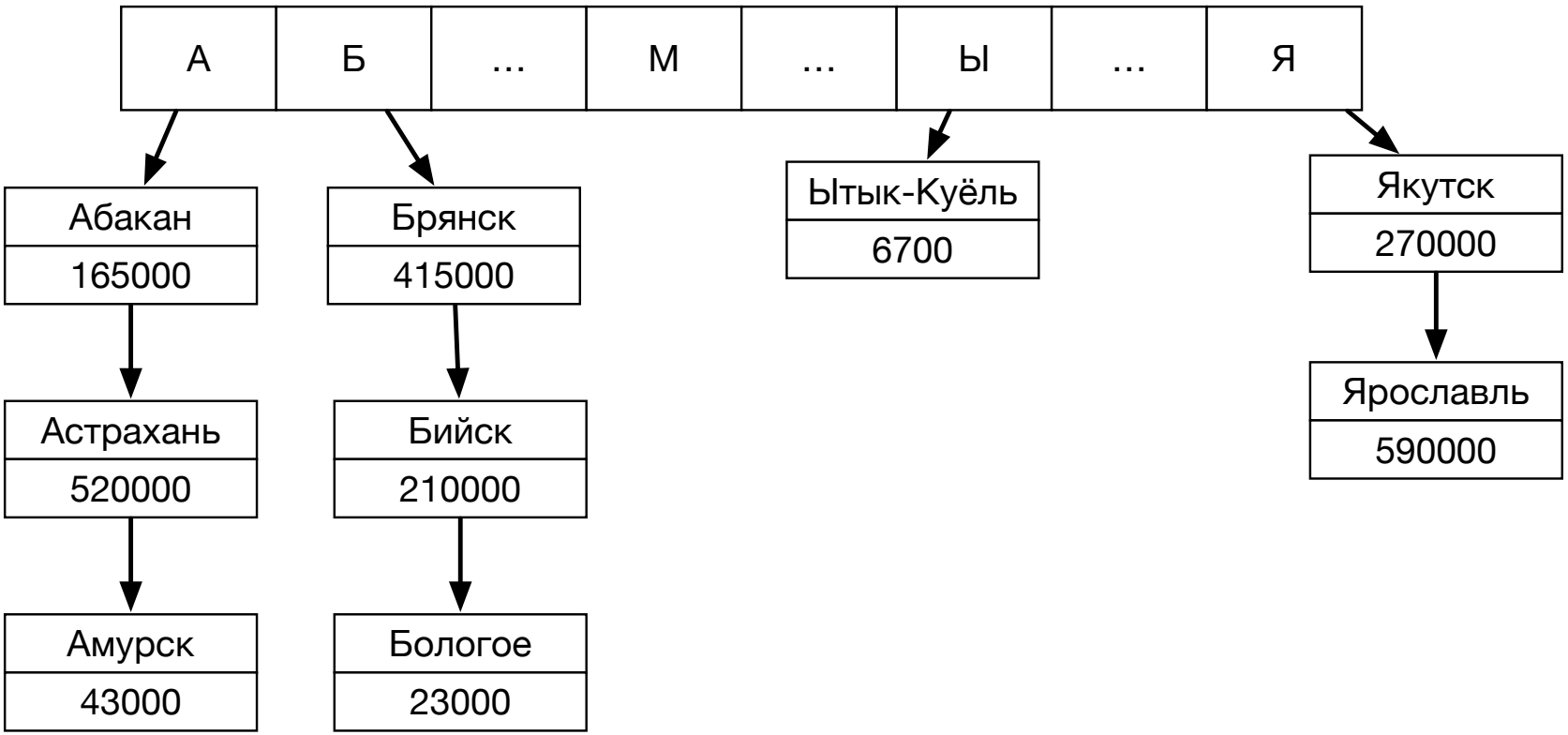
Хеш-таблицы

Принцип: память дешёвая, время дорогое.

- Требуется:
  - Уменьшить амортизационную стоимость поиска
  - Уменьшить функцию (например, $O(\log N) \to O(\log \log N)$)

Обобщённый быстрый поиск

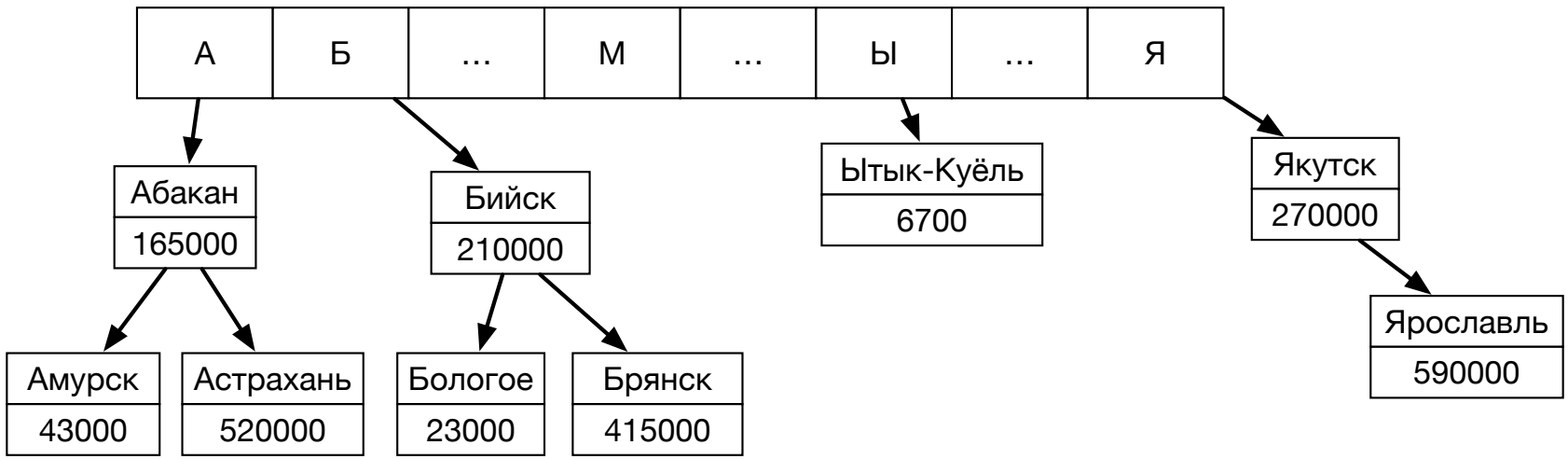
- База данных названий городов и их численности.



33 связных списка.

Обобщённый быстрый поиск

- База данных названий городов и их численности.



33 сбалансированных дерева.

Обобщённый быстрый поиск

- Основная идея — разбиение пространства ключей на независимые подпространства (*partitioning*).
- При независимом разбиении на M подпространств сложность уменьшается.

Для разбиения множества N ключей на примерно равные M подмножеств сложность вычисляется по главной теореме о рекурсии при числе подзадач M, коэффициенте размножения 1 и консолидации $O(1)$.

$$C \cdot O(N) \to \frac{C}{M} O(N)$$

$$C \cdot O(N \log N) \to \frac{C}{M} O(N \log N)$$

Обобщённый быстрый поиск

- При увеличении M

$$\lim_{K \to \infty} T(N, M) = O(1)$$

$$\lim_{K \to \infty} Mem(N, M) = \infty$$

- Имеется зона оптимальности при $M \approx N$

Обобщённый быстрый поиск

- Требуется иметь детерминированный способ разбиения пространства ключей на M независимых подпространств.
- Условия разбиения:

$$|K_1| \approx |K_2| \approx \cdots \approx |K_M|$$

$$\sum_{i=1}^{M} |K_i| = |K|$$

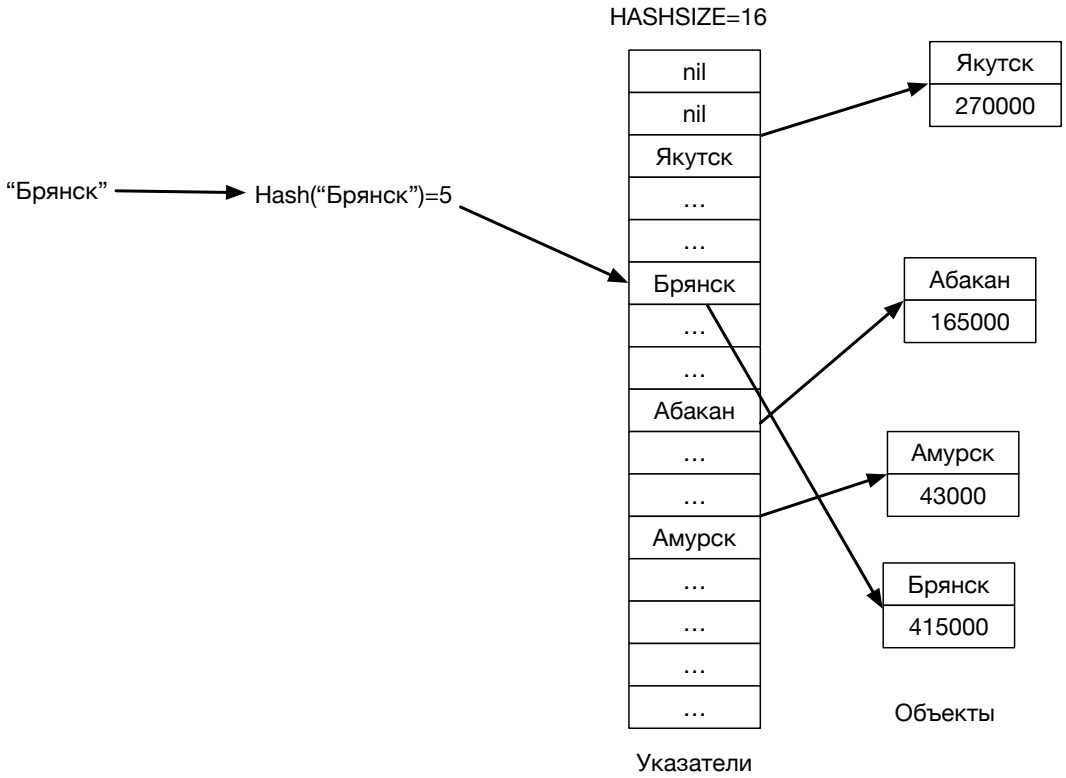
- Эврика! Создаём функцию $H(K)$, удовлетворяющую некоторым условиям.

Хеш - параллелизм

- При партиционировании имеется натуральный параллелизм.
- После поиска по ключу и определении партиции вероятность конфликта уменьшается.

Хеш-таблицы

- Простая хеш-таблица, реализация в виде массива указателей

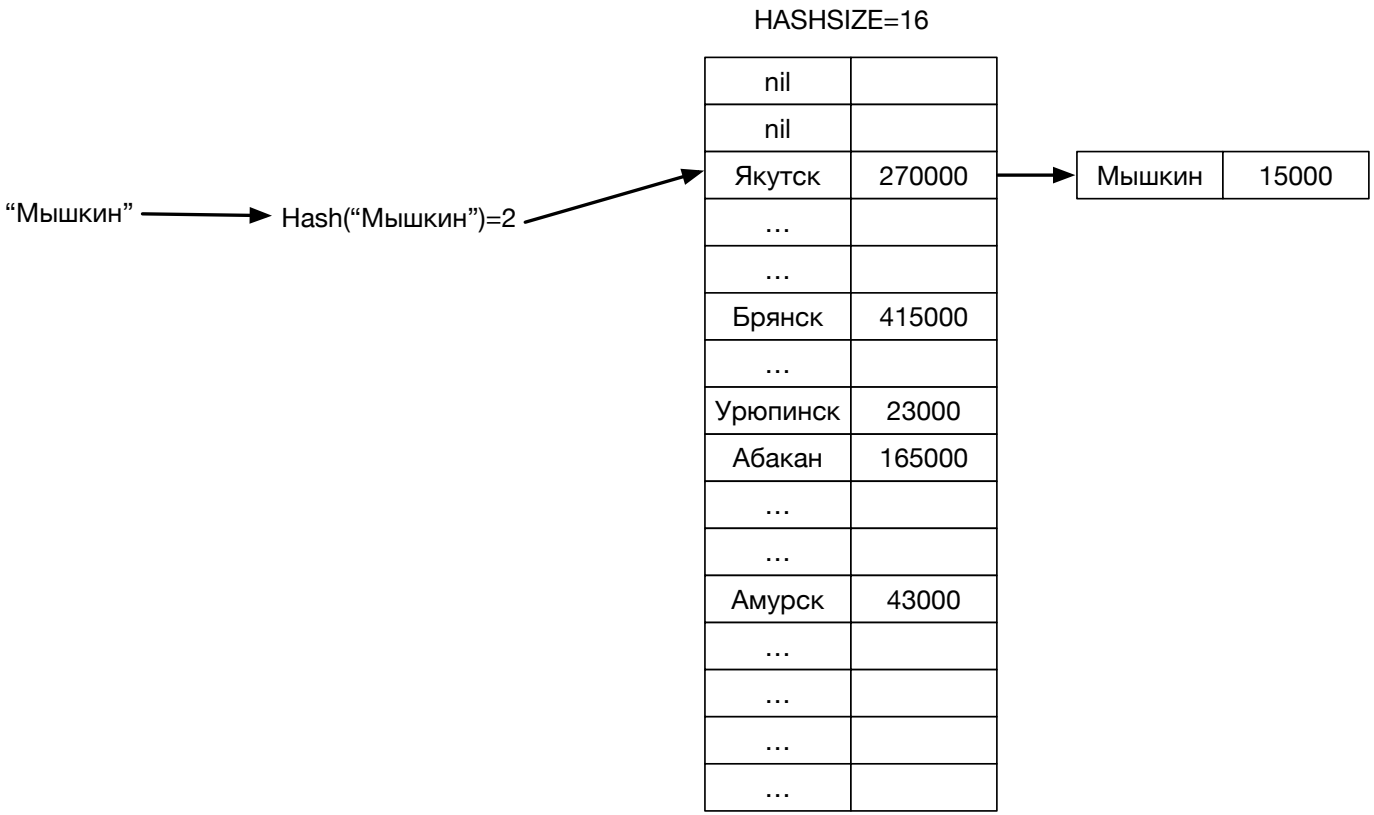


Хеш-таблицы

- Известно количество элементов в контейнере C
- Известен размер массива M
- $\alpha = \frac{C}{M}$ — коэффициент заполнения, *fill-factor*, *load-factor*.
- α — главный показатель хеш-таблицы.

Хеш-таблицы с прямой адресацией

- При коллизии во время создания элемента создаётся вторичная структура данных конфликтующих.



Хеш-таблицы с прямой адресацией

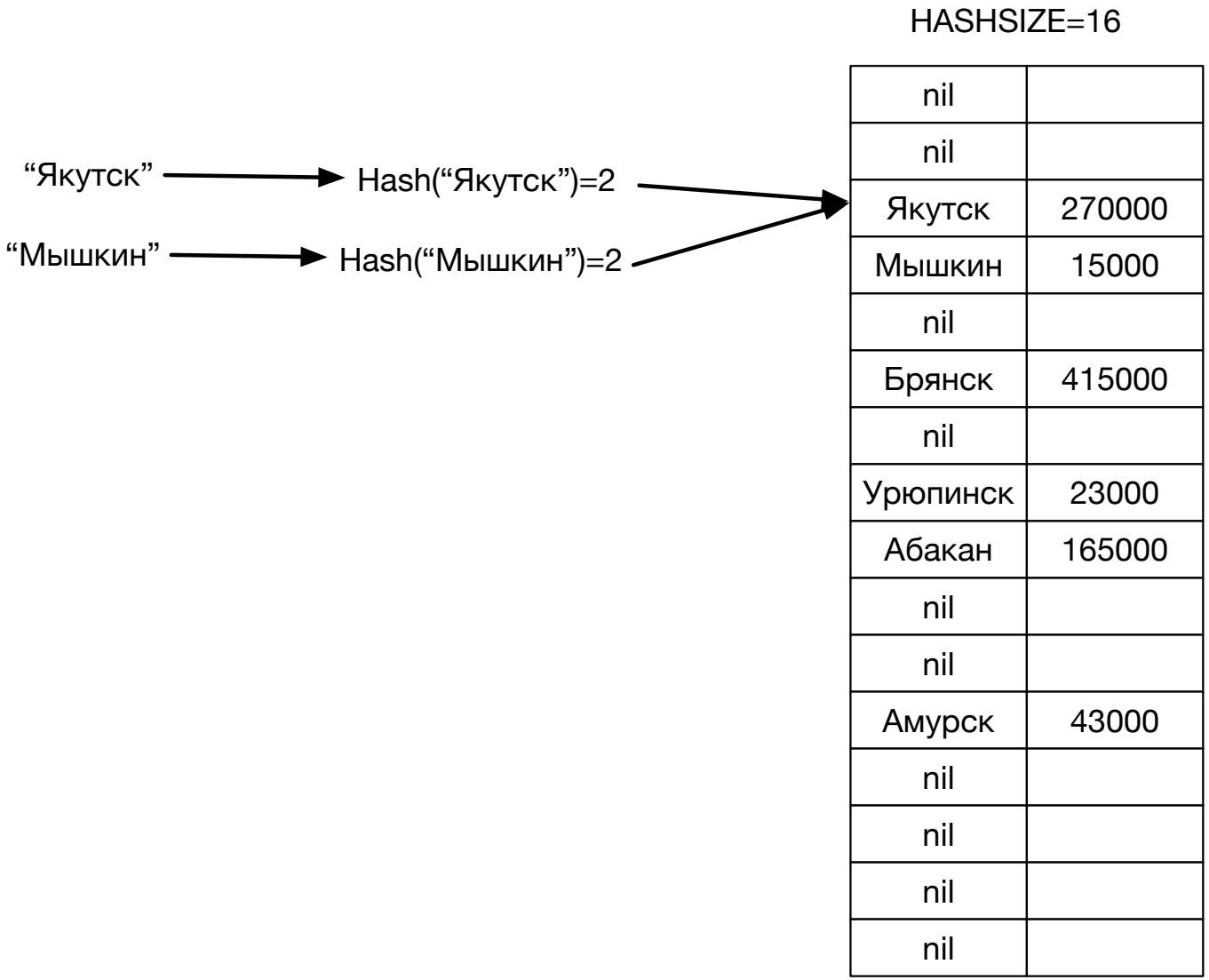
1. При поиске вычисляется хеш-функция.
2. Определяется место поиска — вторичная поисковая структуре данных.
3. Если вторичной структуры нет, то нет и элемента.
4. Иначе элемент ищется во вторичной структуре.

Хеш-таблицы с прямой адресацией

1. При удалении вычисляется хеш-функция.
2. Определяется место поиска — вторичная поисковая структуре данных.
3. Если вторичной структуры нет, то нет и элемента.
4. Иначе элемент удаляется из вторичной структуре.
5. Если вторичная структура пуста, удаляет точку входа.

Хеш-таблицы с открытой адресацией

- Другой способ поиска — искать в той же таблице повторно.



Хеш-таблицы с открытой адресацией

1. При поиске существующего вычисляется хеш-функция.
2. Определяется место поиска — индекс в хеш-таблице.
3. Если по индексу ничего нет, то нет и элемента.
4. Иначе по индексу — элемент с нашим ключом — элемент найден.
5. Если по индексу — элемент с другим ключом или элемент помечен удалённым, индекс изменяется по какому-либо правилу (например, увеличиваем на единицу) и переходим к пункту 3.
6. Следующий индекс вычисляется по тому же правилу.

Хеш-таблицы с открытой адресацией

1. При вставке вычисляется хеш-функция.
2. Определяется место поиска — индекс в хеш-таблице.
3. Если по индексу ничего нет или элемент помечен удалённым, то вставляем по индексу и выходим.
4. Если по индексу элемент с нашим ключом — меняем данные и выходим.
5. Если по индексу имеется элемент с другим ключом то индекс изменяем по правилу и переходим к пункту 3.
6. Следующий индекс вычисляется по тому же правилу.

Подходы к синхронизации хеш-таблиц: `coarse hash set`

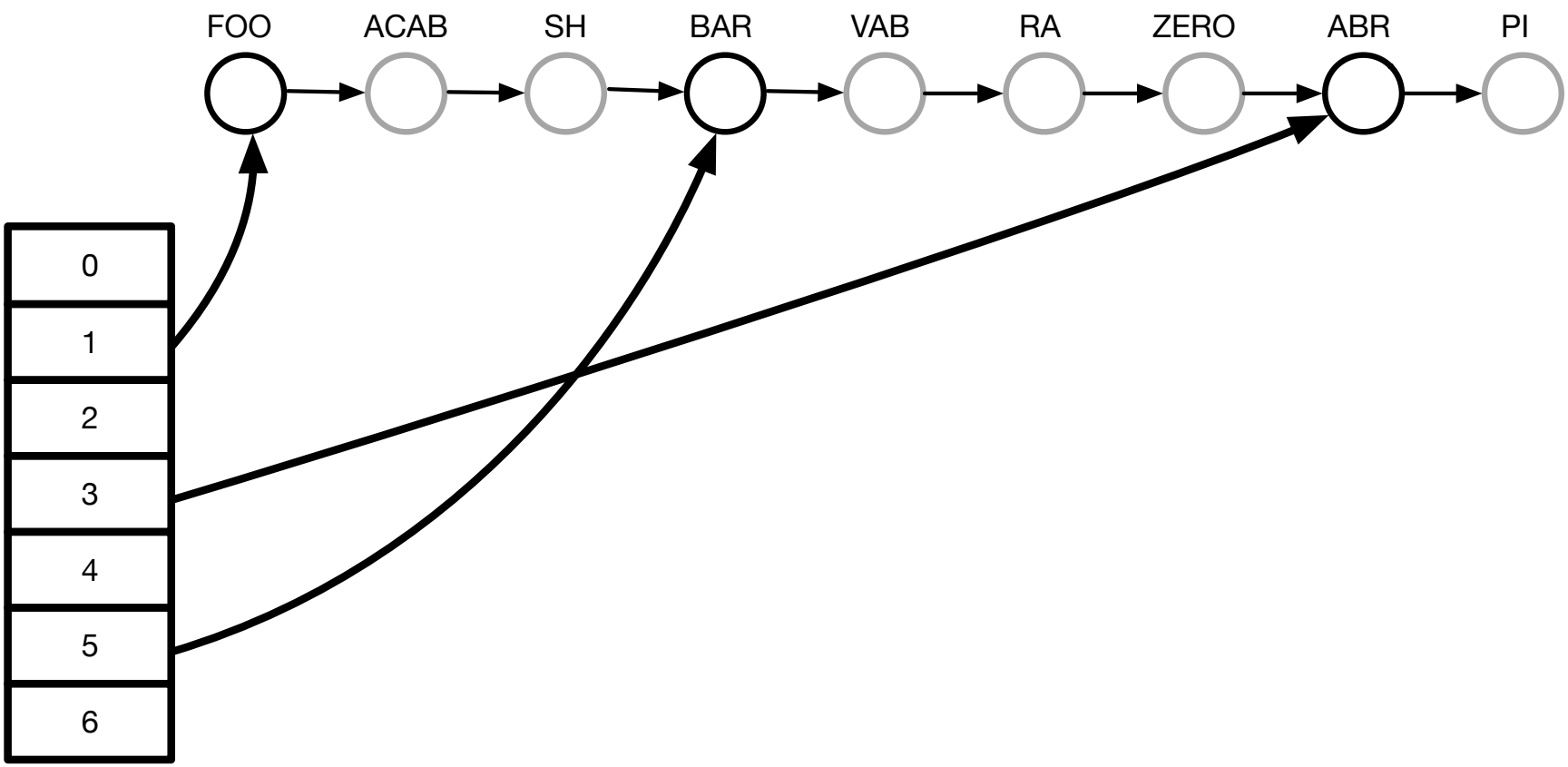
- Имеется глобальный на таблицу замок.
- Операции `create`/`read`/`delete` блокируются на нём на время $O(1)$.
- Операция `resize` блокирует на $O(N)$.

Подходы к синхронизации хеш-таблиц: `lock-free hash set`

- Имеется глобальный связный список всех элементов.
- Хеш-таблица содержит указатели на элементы списка.
- Операция `create` всегда вставляет в голову партиции.
- Голова партиции помечается специальным образом.
- Операции `read` и `delete` пробегают по партиции до результата.

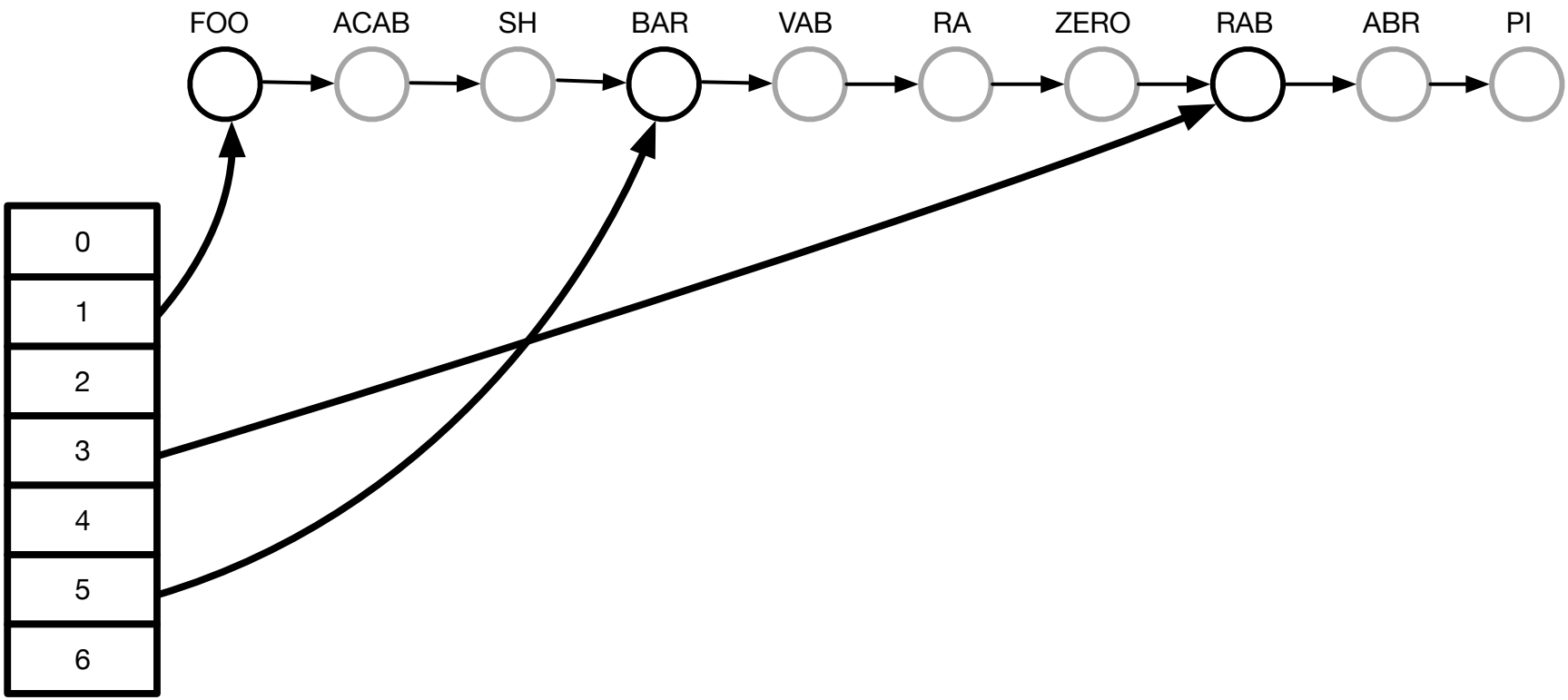
Подходы к синхронизации хеш-таблиц: lock-free hash set

lock-free hash set с несколькими элементами



Подходы к синхронизации хеш-таблиц: lock-free hash set

lock-free hash set после вставки элемента RAB



Подходы к синхронизации хеш-таблиц: `stripped hash set`

- Разбиение на K полос. Для каждой из них — свой замок.
- После хеширования вычисляется нужная полоса и блокируется только она.
- Операции `create`/`read`/`delete` блокируются на нём на время $O(1)$ с вероятностью $\frac{1}{K}$.
- Операция `resize` блокирует на $O(N)$. Важен порядок при блокировке замков!

Подходы к синхронизации хеш-таблиц: `sequential cuckoo hash set`

- Используются две хеш-функции одинаковой мощности.
- Вставка (`create`) без коллизий элементарна.
- Вставка с коллизией всё равно вставляет элемент на нужное место и выкидывает старый элемент на альтернативную позицию и это происходит до тех пор, пока имеются коллизии.
- Количество «выкидываний» в цепочке ограничено и если не найдено место, расширяется таблица.

Подходы к синхронизации хеш-таблиц: `concurrent cuckoo hash set`

- Каждый метод разбивается на фазы.
- Организуется сдвоенная таблица пробных множеств. Каждое множество в свою очередь содержит фиксированное количество элементов.
- Операции `create` и `remove` захватывают множество, содержащее нужный хеш, в обоих таблицах. Операция `remove` далее совпадает с sequential алгоритмом. Операция `create` пытается добавить элемент в одно из множеств. При переполнении множества происходит вытеснение по какой-либо стратегии методом `sequential` сразу всех элементов сверх порога.

Неблокирующая хеш-таблица

- Без учёта операции resize хеш-таблицы наиболее естественно допускают параллельную реализацию.
- Одна из продуктивных идей — кооперативное выполнение.
- Если поток производит модификацию таблицы (вставку/удаление), он оставляет за собой следы и одновременно с этим может исправлять следы функционирования другого потока.
- При упрощении задачи (реализация `set`, а не `map`) и отказа от `resize` возможна относительно простая реализация.

Неблокирующая хеш-таблица

- Каждая позиция есть атомарная пара <состояние,ключ>.
- Возможны состояния {`empty`, `busy`, `inserting`, `member`}
- `Insert`:
  1. Резервируется пустая позиция {`inserting`}
  2. Проверяются другие возможные в соисполнении позиции «в процессе»
  3. Если произошла коллизия, то текущая операция неуспешна и позиция становится {`empty`}
  4. В процессе вставки позиции помечаются {`busy`} и это блокирует другие потоки.
  5. При успехе позиция помечается {`member`}

Неблокирующая хеш-таблица

- При одновременном удалении ключа только один поток удалит, остальные получат неуспех «ключа нет».
- При одновременном добавлении только один поток успешно завершит операцию, остальные получат неуспех «ключ есть»
- При любой, даже однопоточной операции имеются вызовы `CAS`.
- Нельзя хранить значения по ключу.

Спасибо за внимание.

До встречи на зачёте.