



NATIONAL RESEARCH
UNIVERSITY

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Параллельные структуры данных (части I и II из II) **Concurrent Data Structures**

Рамон Антонио Родригес Залепинос
arodriges@hse.ru

Структура модуля 2

Модуль № 2	№		Дата	Тема лекции	№	Домашние задания
	7	1	27 окт	Хеширование, хэш таблицы 2	2b	ДЗ-Каникулы
	8	2	03 ноя	Фильтры	3	Задание на C++
	9	3	10 ноя	СД для вторичной памяти	4	Задание на C++
	10	4	17 ноя	Пространственные СД	4	Продление дедлайна №4
	11	5	24 ноя	Параллельные СД	5	Задание на C++
	12	6	01 дек	Параллельные СД 2	6	Отмена №6 ввиду продления дедлайна по №4
	13	7	08 дек	Деревья в оперативной памяти	Примерно за 2 недели заканчиваются ДЗ	
	14	8	15 дек	Современные тренды		
СЕССИЯ с 21.12.2020						

Highlights:

- лекции, семинары и ДЗ синхронизированы
- некоторые представления об эффективности СД развеяны
 - на семинаре вы собственноручно, на практике сравните производительность красно-черных деревьев и хэш таблиц (если не нужны next & prev)
- пространственные СД – обширный класс: **помимо int & string ∃ и другие типы данных**
 - в современном мире, **около 80% всех данных содержат географическую привязку**: [ссылка 1](#) (Forbes), [ссылка 2](#) (Carto)
 - отдельные секции на значимых конференциях (e.g., VLDB: <https://vldb2020.org/program.html>)

Требование к студентам на лекции: слушайте внимательно!

Предмет
Geoapplications Development
Разработка геоприложений
3 курс, по выбору

План (резюме) лекции, часть I из II

Фрагмент I. Мотивация создания параллельных СД – короткий обзор

Что такое || СД? Почему полезно? Какие особенности?

1. Терминология
2. Предпосылки и причины разработки || СД
3. Ядра, процессы, потоки
4. Как разрабатываются || СД

Фрагмент II. Структура данных «Список с пропусками» (1989 г.)

SkipList, последовательная версия

1. Сложность операций
2. Идея построения
3. Объем занимаемой памяти
4. Алгоритм поиска элемента

* Алг. вставки, высота SkipList

Фрагмент III. Резюме лекции

Следующая лекция: Concurrent Skip List (2006 г.)

План (резюме) лекции, часть II из II

Продолжение: Структура данных «Список с пропусками» (1989 г.)

Последовательная версия SkipList

5. Алгоритм вставки
6. Алгоритм удаления (не сложный, похож на алгоритм вставки)
7. Алгоритм генерации № уровня для нового элемента (вероятность)
8. Алгоритм поиска i -го элемента

Фрагмент II. Структура данных «Concurrent Skip List» (2006 г.)

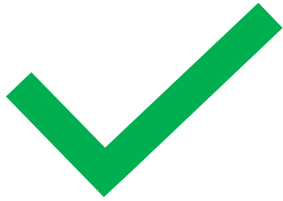
Параллельная версия SkipList

1. Блокировки: lock/unlock
2. Concurrent SkipList: структура узла
3. Параллельный алгоритм поиска узла
4. Параллельный алгоритм вставки узла
5. Особенности работы параллельного алгоритма вставки

* Фрагмент III. Анализ SkipList (приглашаю разобрать на консультации)

Фрагмент IV. Резюме лекции

Терминология: Concurrent Data Structures



- Параллельные структуры данных
- Многопоточные структуры данных
- Структуры данных для параллельного программирования
- «Потокобезопасные» (thread-safe) структуры данных



**В пределах
одной машины**



- Конкурентные структуры данных



- Распределенные структуры данных
- Distributed data structures

**Задействована
компьютерная сеть:
кластеры, GRID, ...**

Деревья – последовательные СД


Библиотечные реализации в языках программирования

std::map

<https://ru.cppreference.com/w/cpp/container/map>

Определён в заголовочном файле `<map>`

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class map;
```

`std::map` — отсортированный ассоциативный контейнер, который содержит пары ключ-значение с неповторяющимися ключами. Порядок ключей задаётся функцией сравнения `Compare`. Операции поиска, удаления и вставки имеют логарифмическую сложность. Данный тип, как правило, реализуется как **красно-чёрное дерево** .

Java SE 12 & JDK 12

```
public class TreeMap<K,V>
    extends AbstractMap<K,V>
    implements NavigableMap<K,V>, Cloneable, Serializable
```

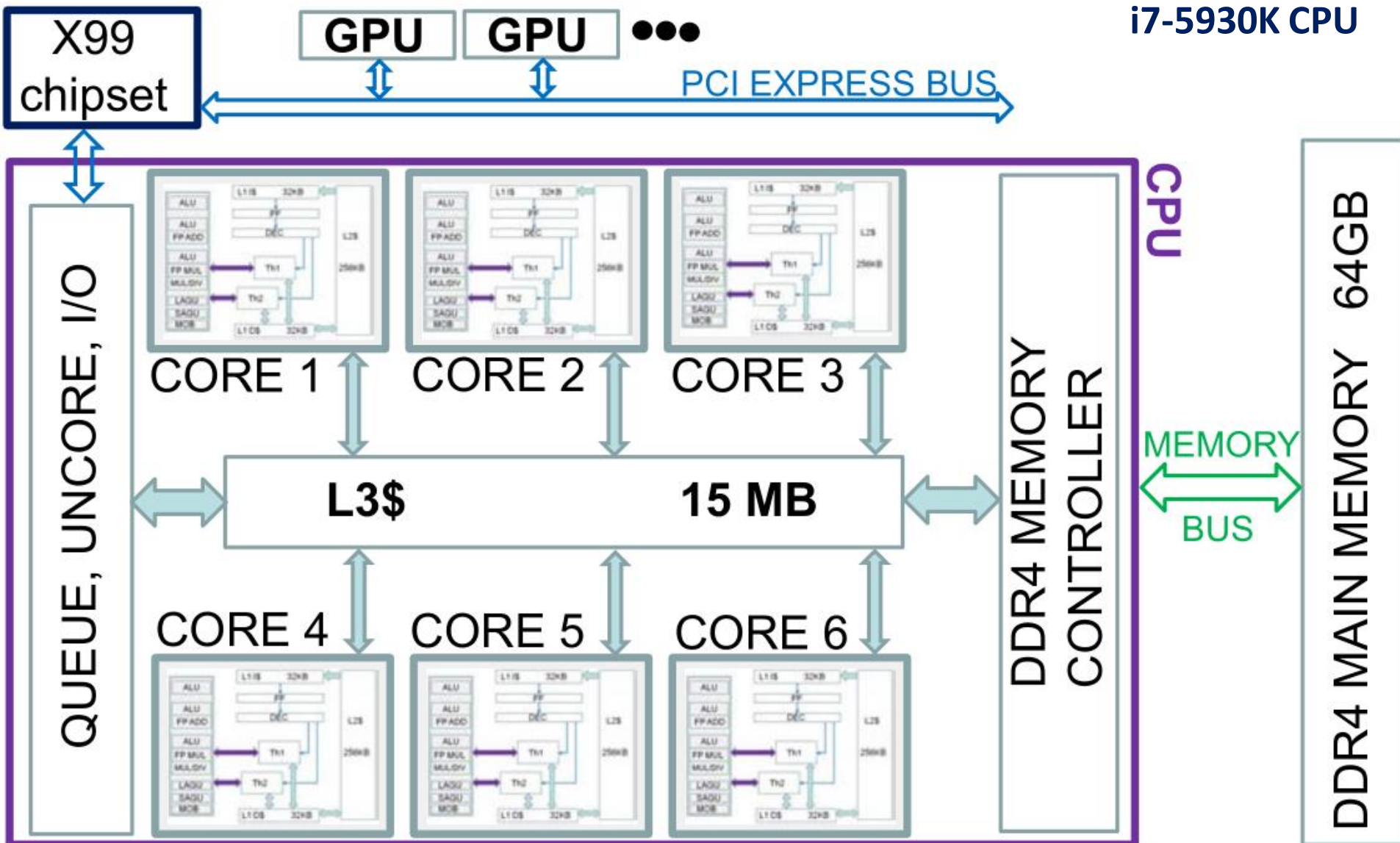
A Red-Black tree based `NavigableMap` implementation. The map is sorted according to the natural ordering of its keys, or by a `Comparator` provided at map creation time, depending on which constructor is used.

<https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/TreeMap.html>

Предпосылки разработки параллельных СД: multicore CPUs

Ядра и архитектура современных CPU

i7-5930K CPU



Современные CPU



Процессор Intel Core i9-10900K OEM

[LGA 1200, 10 x 3700 МГц, L2 - 2.5 МБ, L3 - 20 МБ, 2xDDR4-2933 МГц, Intel UHD Graphics 630, TDP 95 Вт]

Код товара: 1645612

50 299 ₺



Процессор Intel Core i9-10920X BOX

[LGA 2066, 12 x 3500 МГц, L2 - 12 МБ, L3 - 19.25 МБ, 4xDDR4-2933 МГц, TDP 165 Вт]

Код товара: 1605271

59 999 ₺



Процессор Intel Core i9-10900K BOX

[LGA 1200, 10 x 3700 МГц, L2 - 2.5 МБ, L3 - 20 МБ, 2xDDR4-2933 МГц, Intel UHD Graphics 630, TDP 125 Вт]

Код товара: 1689781

51 499 ₺



Процессор Intel Core i9-10940X OEM

[LGA 2066, 14 x 3300 МГц, L2 - 14 МБ, L3 - 19.25 МБ, 4xDDR4-2933 МГц, TDP 165 Вт]

Код товара: 1605279

66 999 ₺



Процессор Intel Core i9-10900K BOX

[LGA 1200, 10 x 3700 МГц, L2 - 2.5 МБ, L3 - 20 МБ, 2xDDR4-2933 МГц, Intel UHD Graphics 630, TDP 125 Вт]

Код товара: 1645625

51 499 ₺



Процессор Intel Core i9-10940X BOX

[LGA 2066, 14 x 3300 МГц, L2 - 14 МБ, L3 - 19.25 МБ, 4xDDR4-2933 МГц, TDP 165 Вт]

Код товара: 1605276

67 999 ₺



Процессор Intel Core i9-10900X OEM

[LGA 2066, 10 x 3700 МГц, L2 - 10 МБ, L3 - 19.25 МБ, 4xDDR4-2933 МГц, TDP 165 Вт]

Код товара: 1605267

52 199 ₺



Процессор Intel Core i9-10900X BOX

[LGA 2066, 10 x 3700 МГц, L2 - 10 МБ, L3 - 19.25 МБ, 4xDDR4-2933 МГц, TDP 165 Вт]

Код товара: 1605260

52 999 ₺



Процессор Intel Core i9-10980XE OEM

[LGA 2066, 18 x 3000 МГц, L2 - 18 МБ, L3 - 24.75 МБ, 4xDDR4-2933 МГц, TDP 165 Вт]

Код товара: 1605284

87 999 ₺



Процессор Intel Core i9-10920X OEM

[LGA 2066, 12 x 3500 МГц, L2 - 12 МБ, L3 - 19.25 МБ, 4xDDR4-2933 МГц, TDP 165 Вт]

Код товара: 1605274

58 999 ₺



Процессор Intel Core i9-10980XE BOX

[LGA 2066, 18 x 3000 МГц, L2 - 18 МБ, L3 - 24.75 МБ, 4xDDR4-2933 МГц, TDP 165 Вт]

Код товара: 1605281

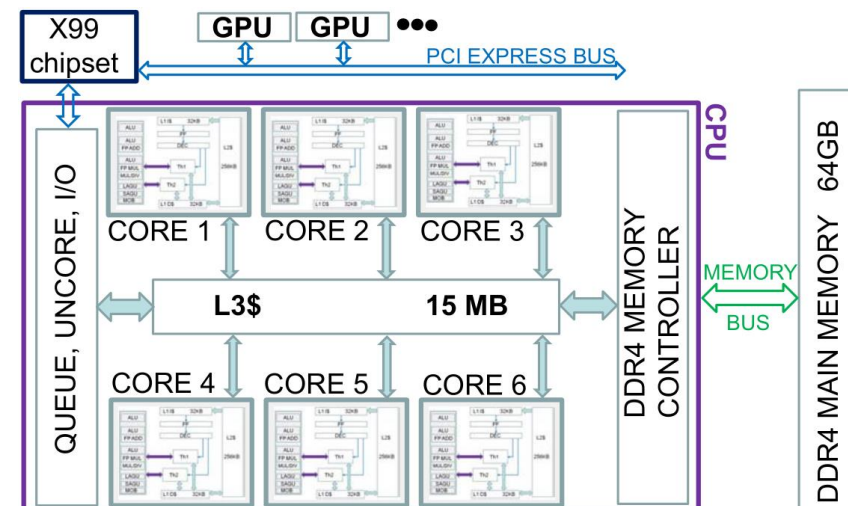
88 999 ₺

Асинхронные методы и параллельные программы

— Асинхронные методы и параллельные программы

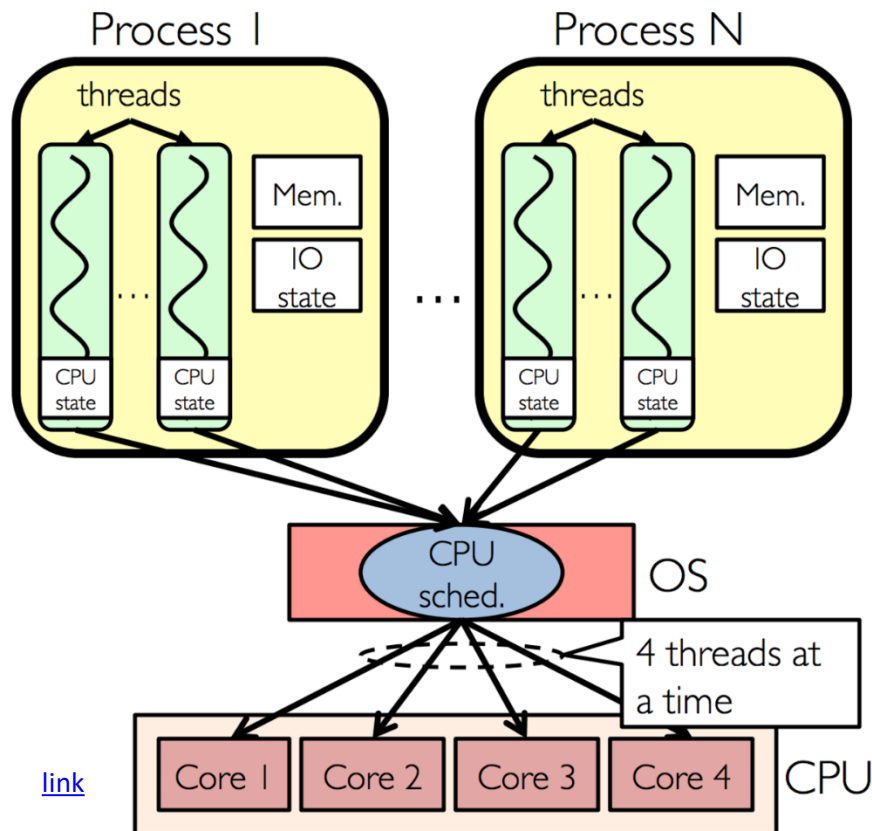
Процессы операционной системы и потоки исполнения. Многопоточность. Пул потоков исполнения. Класс Thread и его члены. Создание потоков. Синхронизация потоков. Передача данных в поток и из потока. Обмены между потоками. Патерны асинхронного программирования: паттерн опроса (polling), паттерн ожидания, паттерн ответного вызова. Механизм async/await. Возможности делегатов в параллельном программировании. Таймеры.

<https://www.hse.ru/edu/courses/292698280#sections>



Ядра (cores), процессы (processes), потоки (threads)

- Процессы – как правило отдельные программы, каждый процесс очень автономен, например имеют свою память и привилегии
- Потоки, *или легковесные процессы*, имеют доступ к общим ресурсам процесса, например, памяти и файловым дескрипторам
- Потоки имеют собственный стек и свои локальные переменные
- Современные операционные системы планируют потоки, не процессы



- Потоки/процессы повышают утилизацию оборудования
- Параллельные программы обычно строятся и использованием потоков (многопоточное программирование)
- Мы не будем говорить о GPU, FPGA, затронем только CPU

Причины разработки параллельных СД

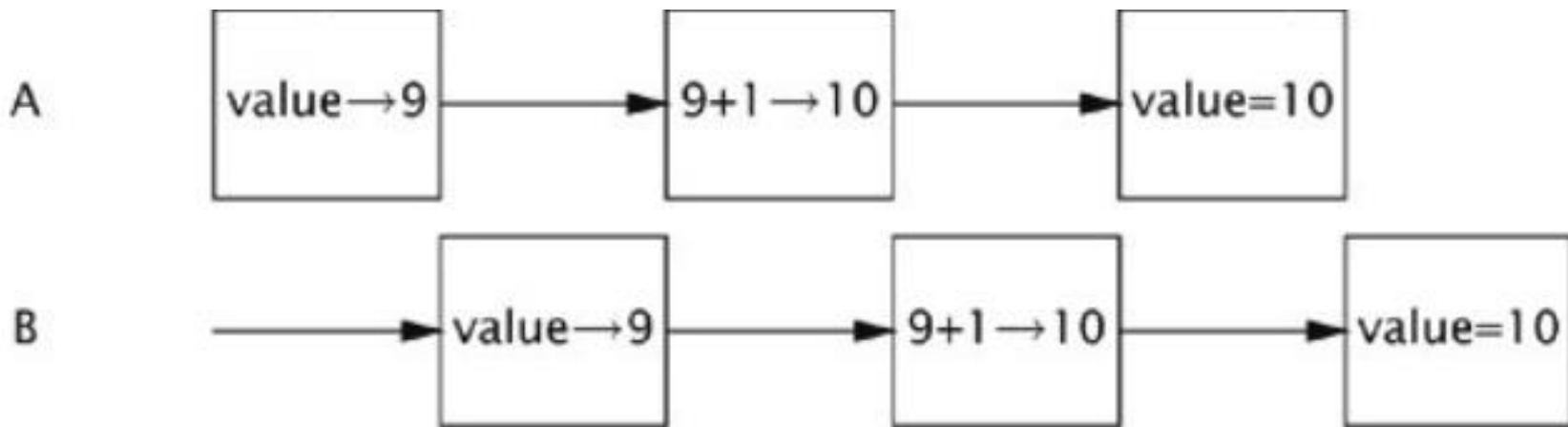
Q: у нас уже есть Структуры Данных (СД), почему нельзя использовать существующие СД?

A: При параллельном доступе из нескольких потоков на запись (write access), СД может перестать быть целостной.

```
SomeFunction() {  
    value++;  
}
```

Два потока несколько раз,
параллельно, вызывают
SomeFunction

race condition
результат зависит от
планирования выполнения
потоков, таймера



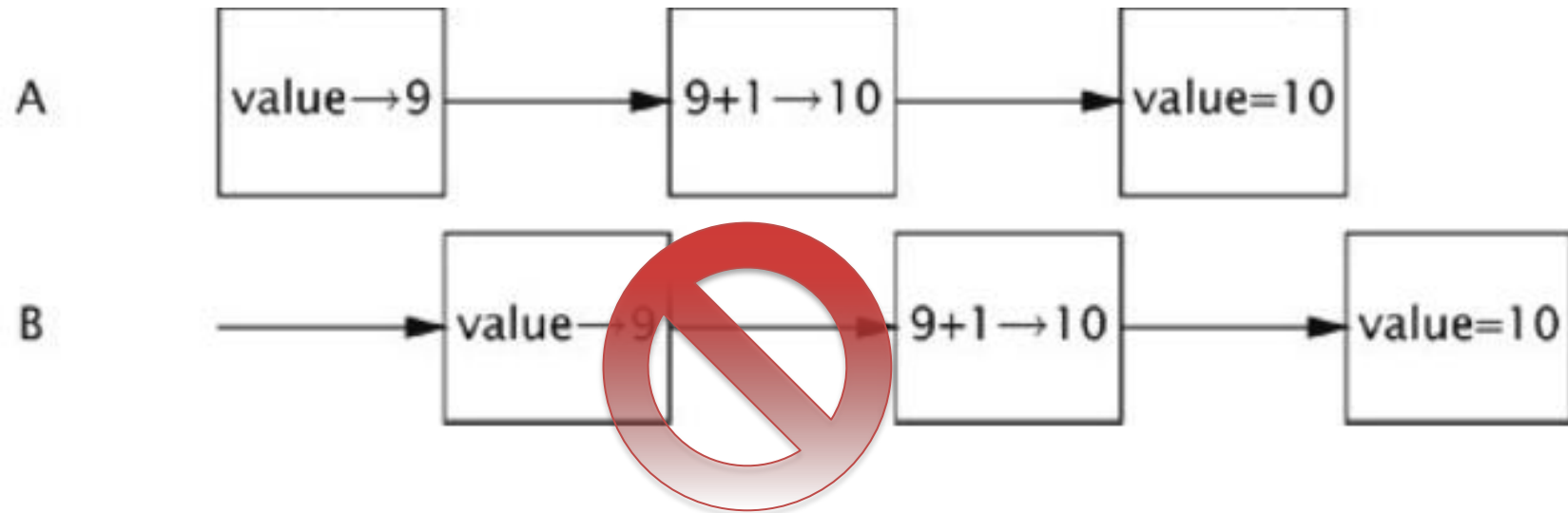
- При доступе на чтение (read access) обычно все в порядке
 - Q: почему?

Критическая секция

Может выполняться одновременно только одним потоком

```
SomeFunction() {  
    НАЧАЛО_КРИТИЧЕСКОЙ_СЕКЦИИ {  
        value++;  
    }  
}
```

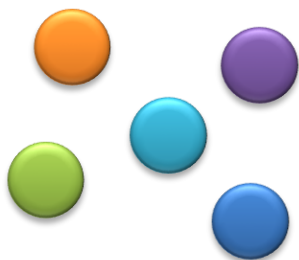
Два потока несколько раз,
параллельно, вызывают
SomeFunction



Причины разработки параллельных СД

Самый простой способ избежать race condition – каждая операция может выполняться только одним потоком

разные потоки

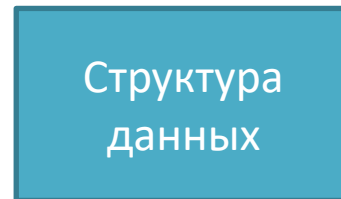


Только один поток
получает доступ к СД в
один момент времени



```
insert(int key) {  
    КРИТИЧЕСКАЯ_СЕКЦИЯ {  
        ВСЕ ТЕЛО функции  
    }  
}
```

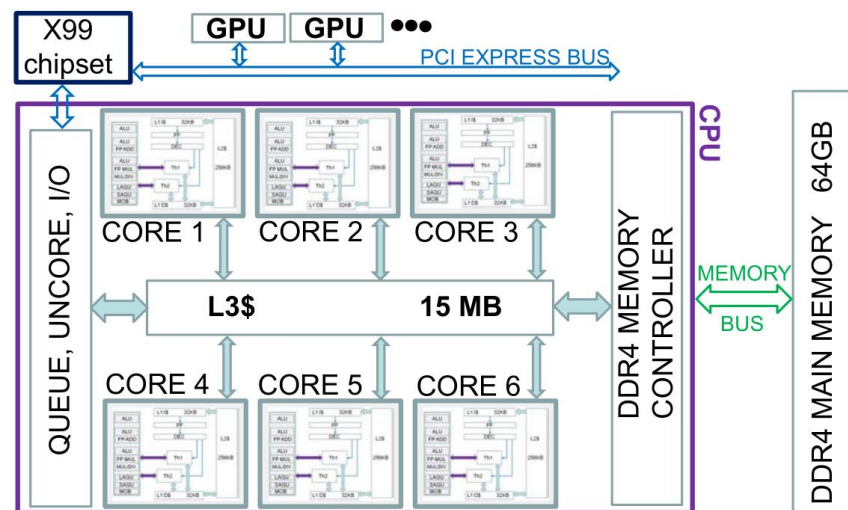
Структура
данных



эти потоки простаивают

Какой поток захватит
управление потом –
неизвестно

herd-effect, это не
очередь!



Как разрабатываются параллельные структуры данных

*распараллеливание алгоритмов
(операций) вставки, удаления, ...*

Существующая последовательная
структура данных *A*



A' параллельная (concurrent) версия
последовательной структуры данных *A*

- thread-safe
- scalable

- сразу очень тяжело мыслить параллельно
- тяжело сразу научиться || программированию
- процедурный стиль → ООП
- ≈ последовательное → параллельное программирование

Аналоги последовательным СД?

Последовательные СД

`std::map`

Определён в заголовочном файле `<map>`

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator
> class map;
```

`std::map` — отсортированный ассоциативный контейнер с неповторяющимися ключами. Порядок ключей и операций удаления и вставки имеют логарифмическую сложность. [\[1\]](#)

```
public class TreeMap<K,V>
    extends AbstractMap<K,V>
    implements NavigableMap<K,V>, Cloneable, Serializable
```

A Red-Black tree based `NavigableMap` implementation. The `map` constructor provides a `Comparator` provided at map creation time, depending on which

Параллельные СД (в оперативной памяти)

?



- Точно не деревья
- Должны быть настолько простыми, что их легко распараллелить
- Легко (относительно) проверить корректность || версии

Список с пропусками – SkipList

Предложен в 1989 г. by William Pugh

- **Вероятностная** структура данных
- Реализация операций SkipList **быстрее и проще** операций над многими видами сбалансированных деревьев (для большинства видов нагрузок)
- SkipList выполняет **неявную балансировку** своей структуры путем рандомизации операций вставки (нет амортизированной стоимости)
- Напоминает по природе QuickSort, потому что **в среднем** время выполнения многих операций очень мало, напр., $O(\log n)$
- **Параллельная реализация** SkipList очень проста

Сложность операций SkipList

- Хранит n элементов
- Каждый элемент имеет ключ key
- Для типа ключа определены операции сравнения, напр., $<$

Алгоритм/ Свойство	В среднем w.h.p. \equiv всегда*	Худший случай w.l.p. \equiv никогда*
Занимаемое место	$O(n)$	$O(n \times \log n)$
Поиск по ключу	$O(\log n)$	$O(n)$
Вставка по ключу	$O(\log n)$	$O(n)$
Удаление по ключу	$O(\log n)$	$O(n)$
Найти минимум по ключу	$O(1)$	$O(1)$
Найти максимум по ключу	$O(1)^{**}$	$O(1)^{**}$
Найти следующий элемент по ключу	$O(\log n)$	$O(n)$
Найти i -й элемент по ключу	$O(\log n)^{***}$	$O(n)^{***}$
Слить два отдельных SkipList	$O(\log n)$	$O(n \times \log n)$
Разбить SkipList на два SkipList	$O(\log n)$	$O(n \times \log n)$

w.h.p = with high probability
w.l.p. = with low probability

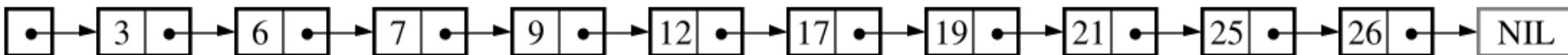
* почти

** при небольшой модификации

*** при индексированном SkipList

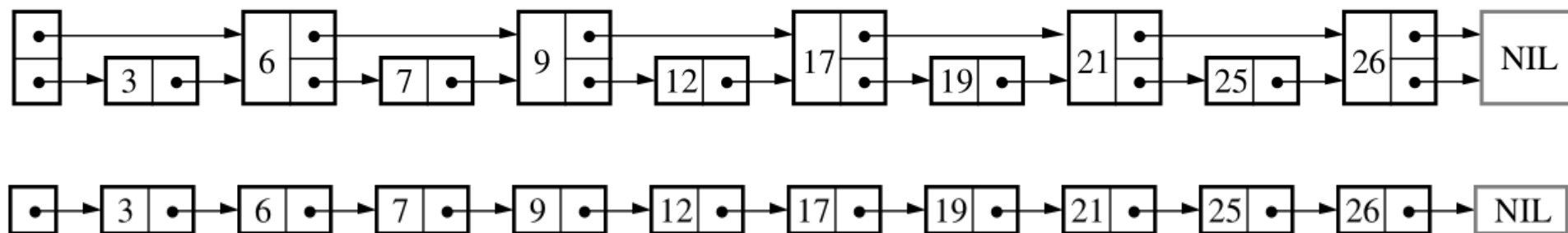
SkipList: идея

- Построим многоуровневый список из обычного односвязного списка с ограничителем
- Вначале отсортируем список (нас интересует быстрый поиск)



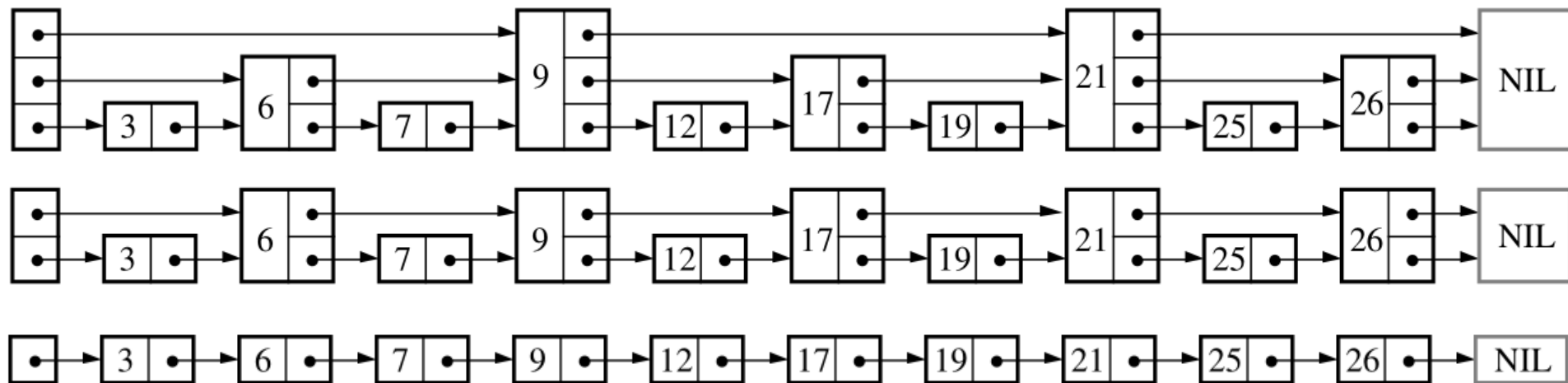
SkipList: идея

- Построим многоуровневый список из обычного односвязного списка с ограничителем
- Вначале отсортируем список (нас интересует быстрый поиск)
- Возьмем каждый второй элемент и построим второй список
- Добавим еще один указатель на два элемента вперед (forward), пропустив каждый второй элемент
- Поиск в таком списке займет не более $\lceil n/2 \rceil + 1$ шагов



SkipList: идея

- Также можно поступить с каждым четвертым элементом
- Поиск займет не более $\lceil n/4 \rceil + 2$ шагов
- Если взять каждый 2^i -й узел, то он будет пропускать $2^i - 1$ узлов самого нижнего уровня и иметь $i + 1$ указателей вперед (forward)
- Поиск займет примерно $\lceil \log n \rceil$ шагов

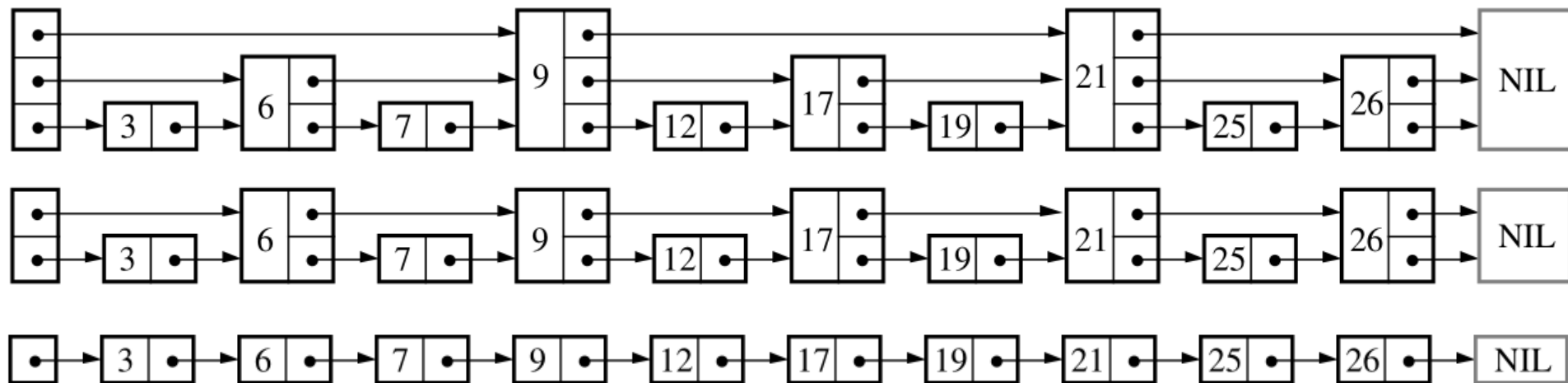


SkipList: идея

- Если взять каждый 2^i -й узел, то он будет пропускать $2^i - 1$ узлов самого нижнего уровня и иметь $i + 1$ указателей вперед (forward)
- Поиск займет примерно $\lceil \log n \rceil$ шагов

ВСТАВКА, УДАЛЕНИЕ, ... - НЕПРАКТИЧНЫ

ИДЕЯ: вставлять случайно, но с той же пропорцией



SkipList: объем занимаемой оперативной памяти

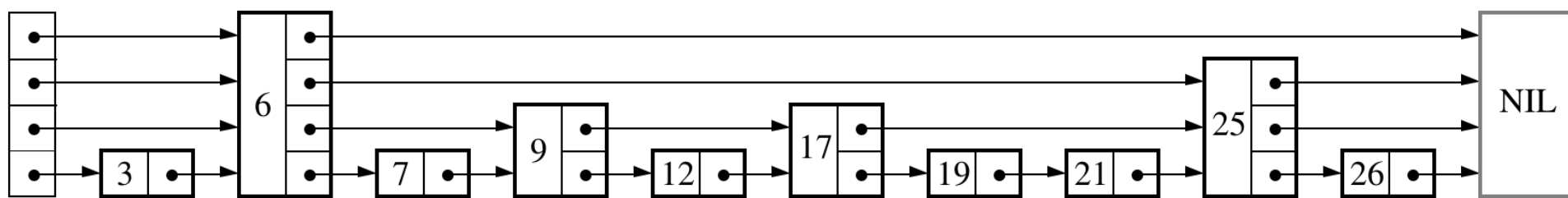
Самый нижний уровень требует $O(n)$ оперативной памяти

- Какой объем оперативной памяти требуется для остальных уровней?
- Observation: полезная нагрузка (ключ и данные) не дублируются на все уровни

Геометрическая прогрессия

Поиск элемента в SkipList

- (1) идти с головы списка самого верхнего уровня, **пропуская (skip)** как можно больше элементов предыдущего уровня
- (2) если ключ следующего элемента $>$ искомого, то спуститься на уровень ниже



Search(list, searchKey)

$x := \text{list} \rightarrow \text{header}$

-- *loop invariant*: $x \rightarrow \text{key} < \text{searchKey}$

for $i := \text{list} \rightarrow \text{level}$ **downto** 1 **do**

while $x \rightarrow \text{forward}[i] \rightarrow \text{key} < \text{searchKey}$ **do**

$x := x \rightarrow \text{forward}[i]$

-- $x \rightarrow \text{key} < \text{searchKey} \leq x \rightarrow \text{forward}[1] \rightarrow \text{key}$

$x := x \rightarrow \text{forward}[1]$

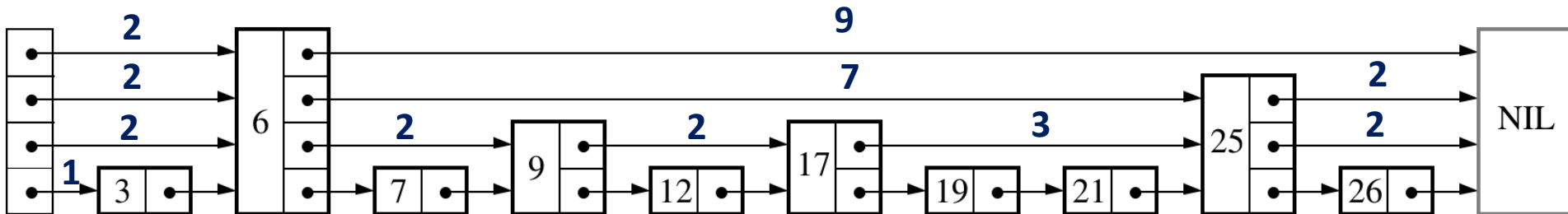
if $x \rightarrow \text{key} = \text{searchKey}$ **then return** $x \rightarrow \text{value}$

else return *failure*

- Давайте найдем элемент с ключом 25
- Теперь давайте найдем элемент с ключом 21
- А если искать 26?
- Всегда идем до уровня 1
- Всегда идем только вперед и вниз: получается «лесенка»
- Мы будем запоминать для add()/delete() ВСЕ предыдущие узлы (поэтому важно спуститься до низшего уровня)

Поиск i -го элемента в SkipList

Для каждой ссылки укажем ее длину – сколько ссылок она пропускает на самом первом уровне



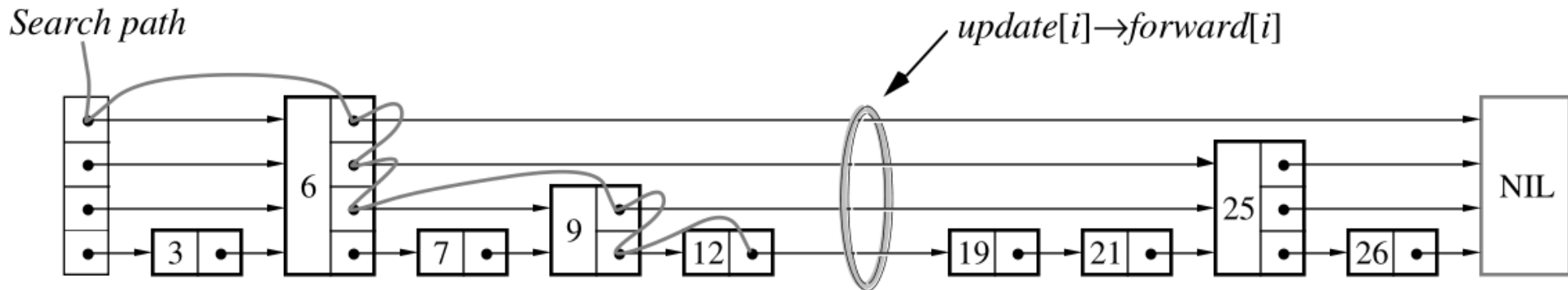
Вставка элемента в SkipList

- (1) **Горизонтально:** найти место для вставки нового элемента \equiv **операция поиска**
- (2) **Вертикально:** сгенерировать случайным образом (равномерное распределение) уровень, на который будет вставлен новый элемент (**именно здесь в структуру данных заложена вероятность**)
- (3) **Обновить** указатели у предыдущих элементов – во время поиска запоминать элемент перед спуском в массиве **update**

Примечание: элементы на более высоких уровнях не обновляются

Пример: вставляем элемент с ключом 17 на уровень 2

До вставки:



После вставки:



Insert(list, searchKey, newValue)

```
local update[1..MaxLevel]
x := list→header
for i := list→level downto 1 do
    while x→forward[i]→key < searchKey do
        x := x→forward[i]
    --  $x \rightarrow \text{key} < \text{searchKey} \leq x \rightarrow \text{forward}[i] \rightarrow \text{key}$ 
    update[i] := x
x := x→forward[1]
if x→key = searchKey then x→value := newValue
else
    lvl := randomLevel()
    if lvl > list→level then
        for i := list→level + 1 to lvl do
            update[i] := list→header
        list→level := lvl
    x := makeNode(lvl, searchKey, value)
    for i := 1 to level do
        x→forward[i] := update[i]→forward[i]
        update[i]→forward[i] := x
```

Вставка элемента в SkipList

- (1) **Горизонтально:** найти место для вставки нового элемента \equiv **операция поиска**
- (2) **Вертикально:** сгенерировать случайным образом (равномерное распределение) уровень, на который будет вставлен новый элемент (**именно здесь в структуру данных заложена вероятность**)
- (3) **Обновить** указатели у предыдущих элементов – во время поиска запоминать элемент перед спуском в массиве **update**

После вставки:



Удаление элемента из SkipList

Внимание: проще вставки; **нет** никакой вероятности; **нет** никакой балансировки

(1) **Найти** элемент

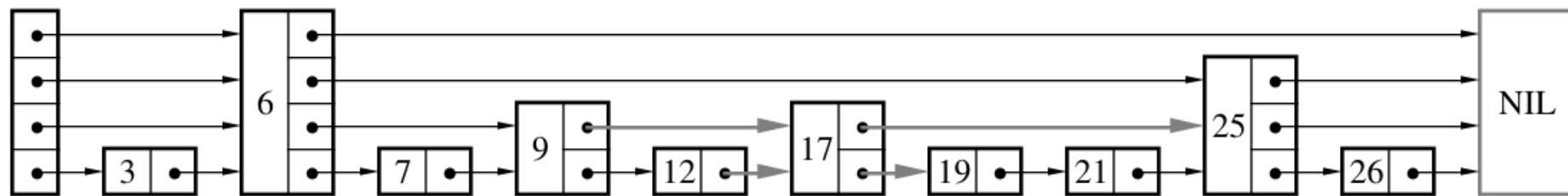
(2) Во время поиска запомнить указатели на предыдущие элементы (на каждом уровне на элемент **ссылается только один предыдущий элемент**)

(3) **Обновить** указатели у предыдущих элементов

Примечание: элементы на более высоких уровнях не обновляются

Пример: удаляем элемент с ключом **17**

После удаления:



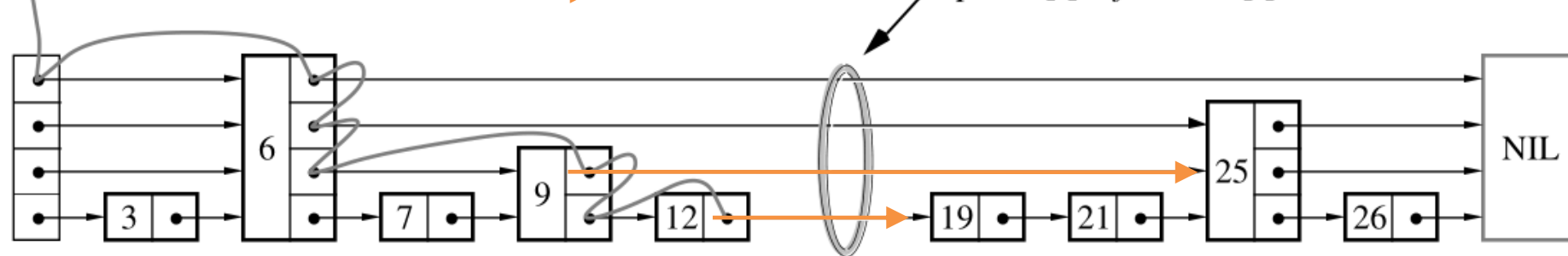
До удаления:

Два указателя обновлены

Search path



$update[i] \rightarrow forward[i]$



Генерация номера уровня для нового элемента при вставке в SkipList

- В дискретном равномерном распределении величина принимает конечное число значений с равными вероятностями.
- Вначале за p мы обозначили долю элементов, переходящих с уровня i на уровень $i + 1$. Тогда величина p не была вероятностью.
- Теперь p будет вероятностью.

randomLevel()

lvl := 1

-- *random() that returns a random value in [0...1)*

while random() < p **and** lvl < MaxLevel **do**

 lvl := lvl + 1

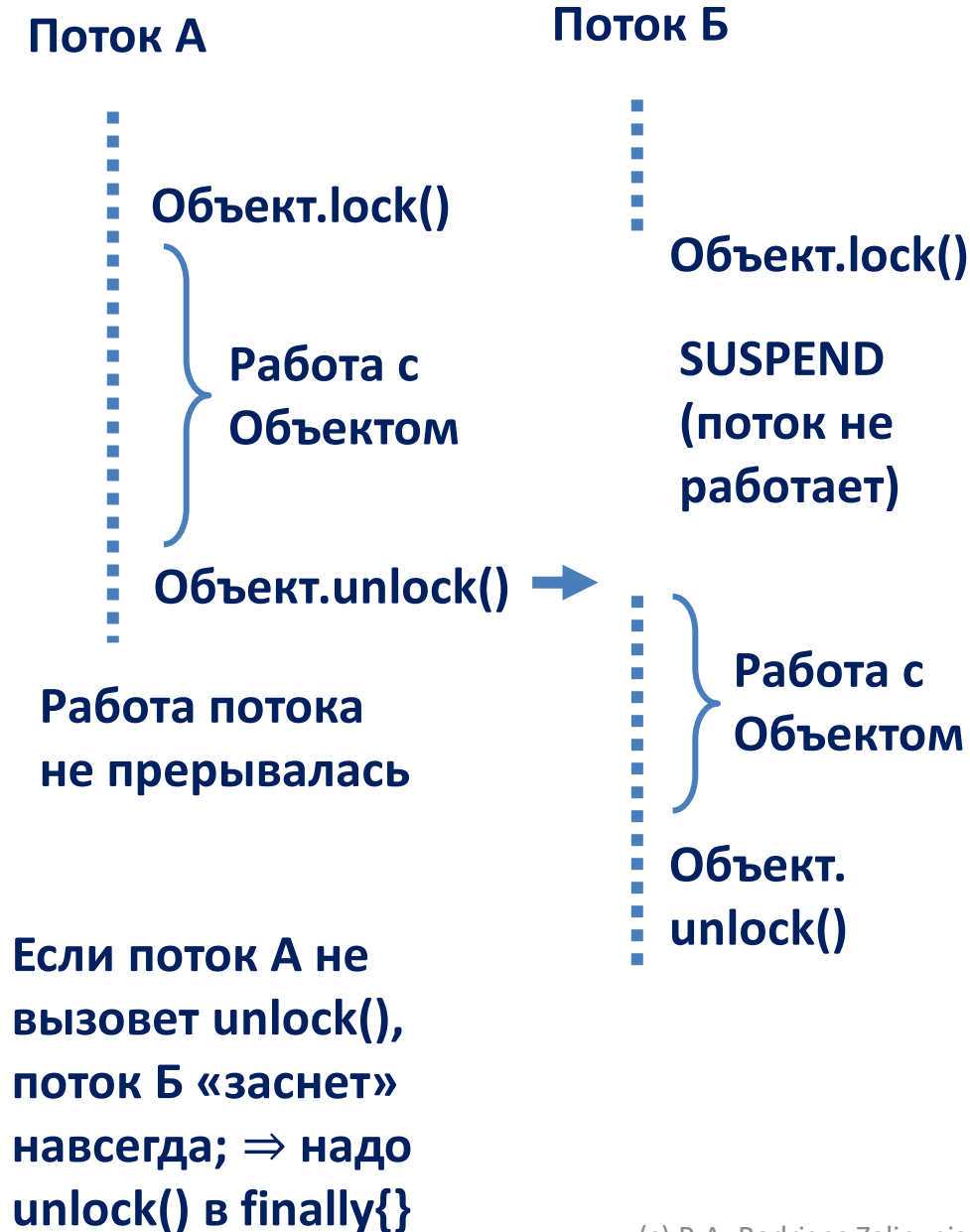
return lvl

Простой перебор уровней,
начиная с первого

Concurrent SkipList

lock / блокировка

- строительный блок параллельного программирования
- предотвращает одновременную модификацию состояния объекта разными потоками
- синтаксис различный в различных языках
- **lock** может «захватить» только один поток
- работа остальных потоков будет приостановлена (тех потоков, которые попытались захватить **lock** и не успели это сделать)
- работа приостановленных потоков возобновиться только тогда, когда захвативший **lock** поток освободит этот **lock**
- типичная ситуация (логическая ошибка алгоритма) – **deadlock**: потоки ждут друг друга и останавливаются навсегда; программа «зависает»



Concurrent SkipList (2006 г.)

Herlihy M. et al. A provably correct scalable concurrent skip list // Conference On Principles of Distributed Systems (OPODIS). – 2006.

- Простые алгоритмы
- Не нарушают структуру SkipList
- Не приводят к deadlocks
- Оптимистические алгоритмы: find() не вызывает lock(), т.е. он выполняется параллельно при выполнении add()/delete()
- add()/delete():
 - в цикле lock() все предшествующие узлы
 - поскольку время прошло (несколько узлов блокировать – неатомарная операция), то надо проверить после, что они не изменились
 - вставить/удалить узел изменив ссылки на следующие узлы у своих предшественников
 - потом блокировки снимаются \Rightarrow add()/delete() атомарные

Concurrent SkipList: структура узла

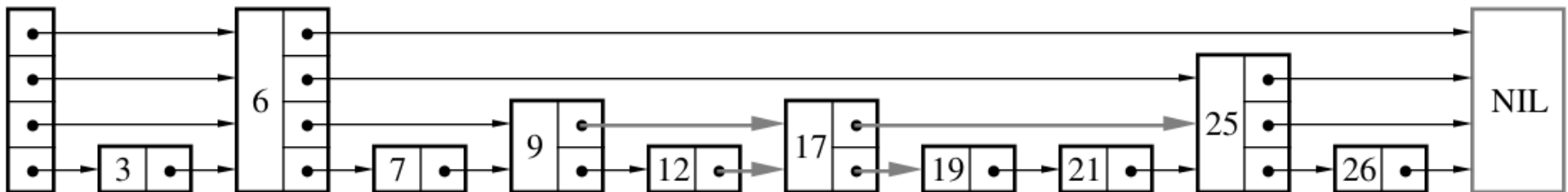
```
class Node {  
    int key;           — Ключ  
    int topLayer;      — Уровень, на котором расположен узел  
    Node** nexts;      — Указатели на следующие узлы  
    bool marked;       — Удаляется ли сейчас этот узел?  
    bool fullyLinked;  — Завершена ли вставка узла?  
    Lock lock;         — Блокировка  
};
```

- Ключ находится в SkipList **if and only if**
 ключ не удаляется и вставка завершена
- Нужен сборщик мусора... см. далее

Concurrent SkipList: findNode()

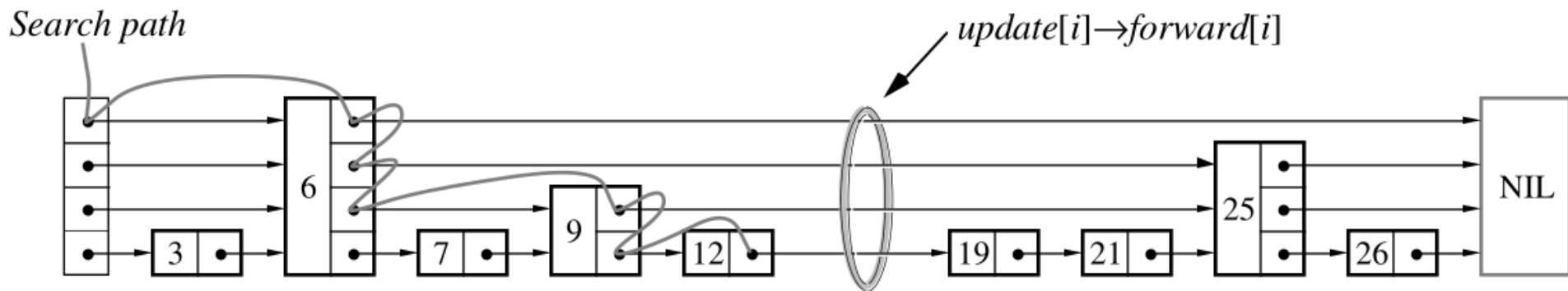
```
int findNode(int v,  
             Node* preds[],  
             Node* succs[]) {  
    int lFound = -1;  
    Node* pred = &LSentinel;  
    for (int layer = MaxHeight - 1;  
         layer ≥ 0;  
         layer--) {  
        Node* curr = pred->nexts[layer];  
        while (v > curr->key) {  
            pred = curr; curr = pred->nexts[layer];  
        }  
        if (lFound == -1 && v == curr->key) {  
            lFound = layer;  
        }  
        preds[layer] = pred;  
        succs[layer] = curr;  
    }  
    return lFound;  
}
```

- Возвращаем уровень узла
- Запоминаем предшествующие и последующие узлы (могут быть ограничителями)



Concurrent SkipList: add()

— Идея: локально подготовить условия, lock все предыдущие элементы



```

bool add(int v) {
    int topLayer = randomLevel(MaxHeight);
    Node* preds[MaxHeight], succs[MaxHeight];
    while (true) {
        int lFound = findNode(v, preds, succs);
        if (lFound != -1) {
            Node* nodeFound = succs[lFound];
            if (!nodeFound->marked) {
                while (!nodeFound->fullyLinked) {};
                return false;
            }
            continue;
        }
        int highestLocked = -1;
        try {
            Node *pred, *succ, *prevPred = null;
            bool valid = true;
            for (int layer = 0;
                valid && (layer <= topLayer);
                layer++) {
                pred = preds[layer];
                succ = succs[layer];
                if (pred != prevPred) {
                    pred->lock.lock();
                    highestLocked = layer;
                    prevPred = pred;
                }
                valid = !pred->marked && !succ->marked &&
                    pred->nexts[layer] == succ;
            }
            if (!valid) continue;

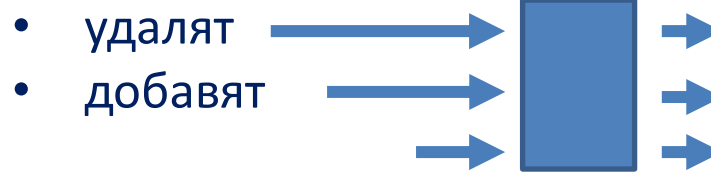
            Node* newNode = new Node(v, topLayer);
            for (int layer = 0;
                layer <= topLayer;
                layer++) {
                newNode->nexts[layer] = succs[layer];
                preds[layer]->nexts[layer] = newNode;
            }

            newNode->fullyLinked = true;
            return true;
        }
        finally { unlock(preds, highestLocked); }
    }
}

```

Все дело в связях: ждем,
пока связи установят, т.е.

пока предшествующие узлы



Concurrent SkipList: add()

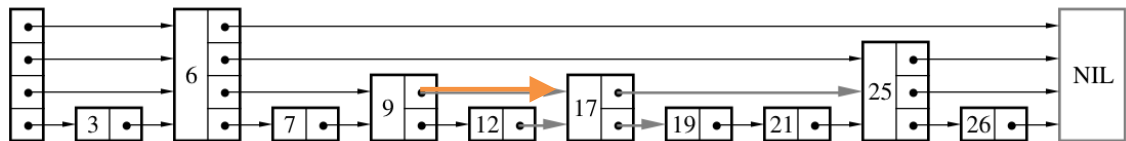
- удалят
- добавят
- Найти место для вставки узла \approx findNode()
- Спускаемся к месту вставки, запоминаем предыдущие и следующие узлы.
- По пути lock() предыдущие узлы (их будем модифицировать, другие узлы – нет)
- По пути проверяем, может предыдущие/ следующие узлы начали удалять? Между ними ничего нового не появилось (предыдущий указывает на следующий)?
- Если так, unlock() в finally и все повторим; к тому времени, возможно, delete() завершатся
- Наконец-то! Мы получили lock на все предыдущие узлы, а значит никто их не изменяет (напр., не удаляет ни их, ни следующие, не вставляет после них: для всего этого надо получать lock)
- Теперь можно спокойно вставить новый узел: снизу вверх, справа налево
- Узел могут найти до установления всех связей
- Внимание: finally находится внутри while

Concurrent SkipList: add()

```
bool add(int v) {  
    int topLayer = randomLevel(MaxHeight);  
    Node* preds[MaxHeight], succs[MaxHeight];  
    while (true) {  
        int lFound = findNode(v, preds, succs);  
        if (lFound  $\neq$  -1) {  
            Node* nodeFound = succs[lFound];  
            if (!nodeFound->marked) {  
                while (!nodeFound->fullyLinked) {}  
                return false;  
            }  
            continue;  
        }  
        int highestLocked = -1;  
        try {  
            Node *pred, *succ, *prevPred = null;  
            bool valid = true;  
            for (int layer = 0;  
                valid && (layer  $\leq$  topLayer);  
                layer++) {  
                pred = preds[layer];  
                succ = succs[layer];  
                if (pred  $\neq$  prevPred) {  
                    pred->lock.lock();  
                    highestLocked = layer;  
                    prevPred = pred;  
                }  
                valid = !pred->marked && !succ->marked &&  
                    pred->nexts[layer] == succ;  
            }  
            if (!valid) continue;  
  
            Node* newNode = new Node(v, topLayer);  
            for (int layer = 0;  
                layer  $\leq$  topLayer;  
                layer++) {  
                newNode->nexts[layer] = succs[layer];  
                preds[layer]->nexts[layer] = newNode;  
            }  
  
            newNode->fullyLinked = true;  
            return true;  
        }  
        finally { unlock(preds, highestLocked); }  
    }  
}
```

Поток А

Поток Б

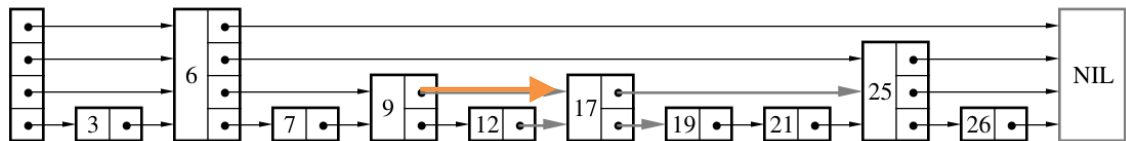


Concurrent SkipList: add()

```
bool add(int v) {  
    int topLayer = randomLevel(MaxHeight);  
    Node* preds[MaxHeight], succs[MaxHeight];  
    while (true) {  
        int lFound = findNode(v, preds, succs);  
        if (lFound  $\neq$  -1) {  
            Node* nodeFound = succs[lFound];  
            if (!nodeFound->marked) {  
                while (!nodeFound->fullyLinked) {}  
                return false;  
            }  
        }  
        continue;  
    }  
    int highestLocked = -1;  
    try {  
        Node *pred, *succ, *prevPred = null;  
        bool valid = true;  
        for (int layer = 0;  
            valid && (layer  $\leq$  topLayer);  
            layer++) {  
            pred = preds[layer];  
            succ = succs[layer];  
            if (pred  $\neq$  prevPred) {  
                pred->lock.lock();  
                highestLocked = layer;  
                prevPred = pred;  
            }  
            valid = !pred->marked && !succ->marked &&  
                pred->nexts[layer] == succ;  
        }  
        if (!valid) continue;  
  
        Node* newNode = new Node(v, topLayer);  
        for (int layer = 0;  
            layer  $\leq$  topLayer;  
            layer++) {  
            newNode->nexts[layer] = succs[layer];  
            preds[layer]->nexts[layer] = newNode;  
        }  
  
        newNode->fullyLinked = true;  
        return true;  
    }  
    finally { unlock(preds, highestLocked); }  
}
```

Поток А

Поток Б: remove()

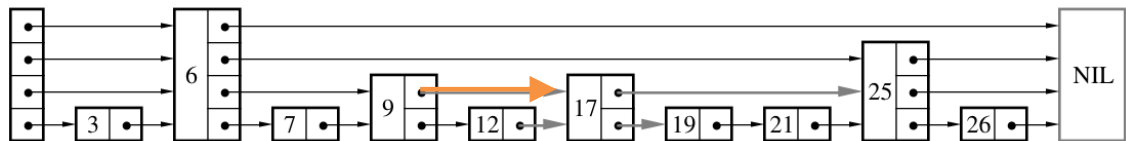


Concurrent SkipList: add()

```
bool add(int v) {  
    int topLayer = randomLevel(MaxHeight);  
    Node* preds[MaxHeight], succs[MaxHeight];  
    while (true) {  
        int lFound = findNode(v, preds, succs);  
        if (lFound != -1) {  
            Node* nodeFound = succs[lFound];  
            if (!nodeFound->marked) {  
                while (!nodeFound->fullyLinked) {}  
                return false;  
            }  
        }  
        continue;  
    }  
    int highestLocked = -1;  
    try {  
        Node *pred, *succ, *prevPred = null;  
        bool valid = true;  
        for (int layer = 0;  
            valid && (layer <= topLayer);  
            layer++) {  
            pred = preds[layer];  
            succ = succs[layer];  
            if (pred != prevPred) {  
                pred->lock.lock();  
                highestLocked = layer;  
                prevPred = pred;  
            }  
            valid = !pred->marked && !succ->marked &&  
                pred->nexts[layer] == succ;  
        }  
        if (!valid) continue;  
  
        Node* newNode = new Node(v, topLayer);  
        for (int layer = 0;  
            layer <= topLayer;  
            layer++) {  
            newNode->nexts[layer] = succs[layer];  
            preds[layer]->nexts[layer] = newNode;  
        }  
  
        newNode->fullyLinked = true;  
        return true;  
    }  
    finally { unlock(preds, highestLocked); }  
}
```

Поток А – потенциально: suspend

Поток Б – add тем временем другой элемент
либо delete следующий по списку

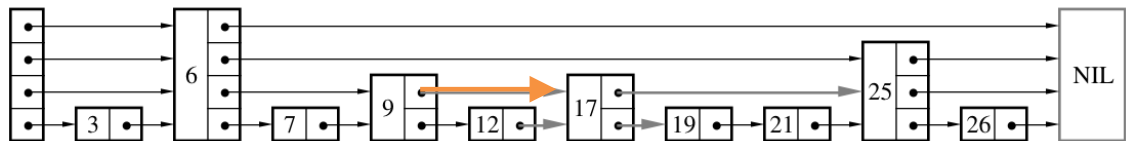


Concurrent SkipList: add()

```
bool add(int v) {  
    int topLayer = randomLevel(MaxHeight);  
    Node* preds[MaxHeight], succs[MaxHeight];  
    while (true) {  
        int lFound = findNode(v, preds, succs);  
        if (lFound != -1) {  
            Node* nodeFound = succs[lFound];  
            if (!nodeFound->marked) {  
                while (!nodeFound->fullyLinked) {}  
                return false;  
            }  
        }  
        continue;  
    }  
    int highestLocked = -1;  
    try {  
        Node *pred, *succ, *prevPred = null;  
        bool valid = true;  
        for (int layer = 0;  
            valid && (layer <= topLayer);  
            layer++) {  
            pred = preds[layer];  
            succ = succs[layer];  
            if (pred != prevPred) {  
                pred->lock.lock();  
                highestLocked = layer;  
                prevPred = pred;  
            }  
            valid = !pred->marked && !succ->marked &&  
                pred->nexts[layer] == succ;  
        }  
        if (!valid) continue;  
  
        Node* newNode = new Node(v, topLayer);  
        for (int layer = 0;  
            layer <= topLayer;  
            layer++) {  
            newNode->nexts[layer] = succs[layer];  
            preds[layer]->nexts[layer] = newNode;  
        }  
  
        newNode->fullyLinked = true;  
        return true;  
    }  
    finally { unlock(preds, highestLocked); }  
}
```

Поток А – допустим точно suspend

Теперь Поток А «проснулся». Проверяем состояния только после того, как получили lock, иначе мы можем проверить до lock, но потом состояние может измениться

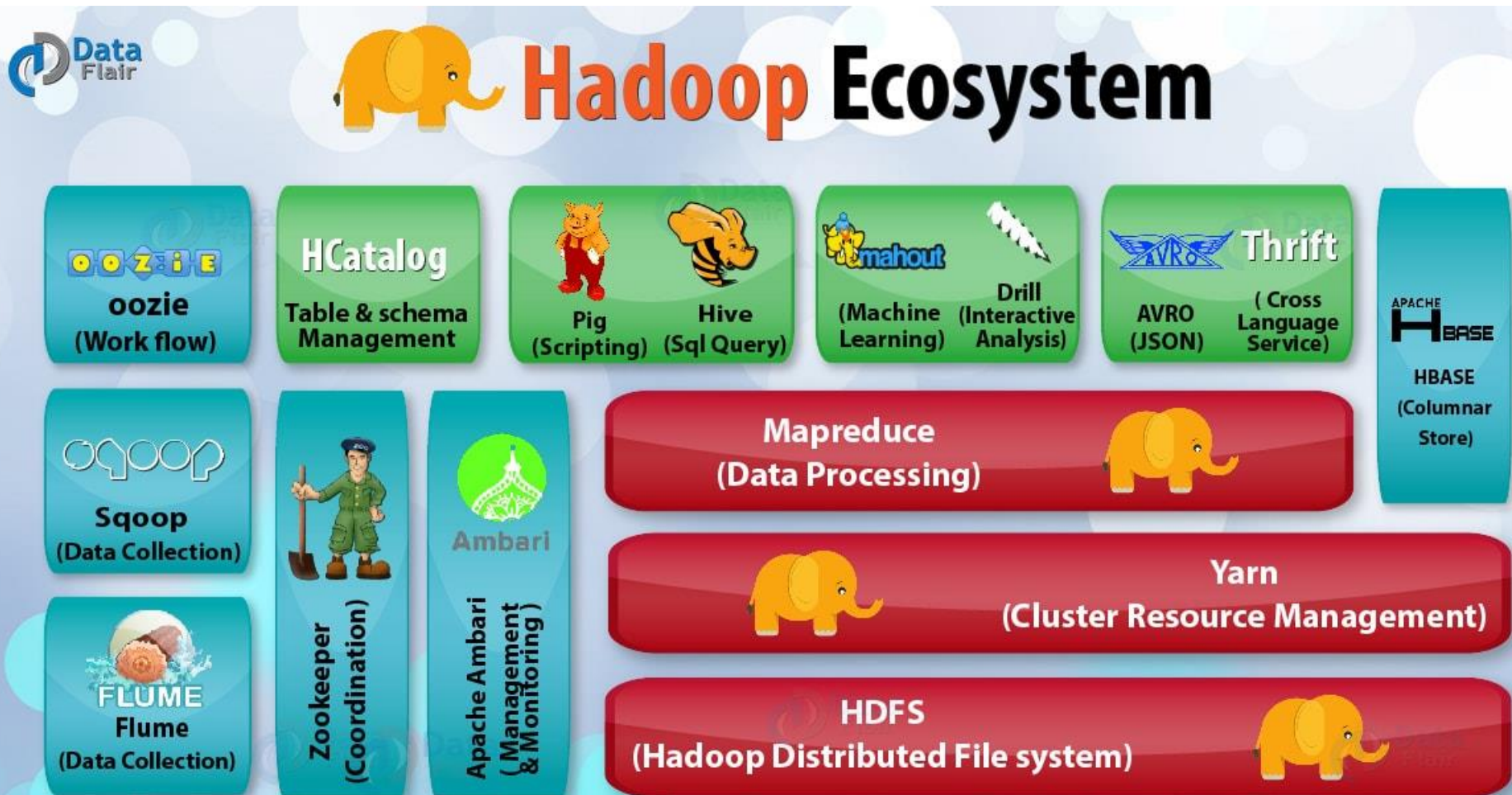


Анализ SkipList

на консултации

Промышленное использование SkipList

[ConcurrentSkipListSet](#) и [ConcurrentSkipListMap](#) в Java API
используется [Apache Hbase](#) – из экосистемы Hadoop



Список литературы

1. Pugh, W. (1990). "Skip lists: A probabilistic alternative to balanced trees" (PDF). Communications of the ACM. 33 (6): 668–676. doi:10.1145/78973.78977.
2. Pugh W. A skip list cookbook. – 1998.
3. Pugh, William (April 1989). Concurrent Maintenance of Skip Lists (PS, PDF) (Technical report). Dept. of Computer Science, U. Maryland. CS-TR-2222.
4. Herlihy M. et al. A provably correct scalable concurrent skip list // Conference On Principles of Distributed Systems (OPODIS). – 2006.

Некоторые рисунки в данной презентации из [1]

План (резюме) лекции, часть I из II

Фрагмент I. Мотивация создания параллельных СД – короткий обзор

Что такое || СД? Почему полезно? Какие особенности?

1. Терминология
2. Предпосылки и причины разработки || СД
3. Ядра, процессы, потоки
4. Как разрабатываются || СД

Фрагмент II. Структура данных «Список с пропусками» (1989 г.)

SkipList, последовательная версия

1. Сложность операций
2. Идея построения
3. Объем занимаемой памяти
4. Алгоритм поиска элемента

* Алг. вставки, высота SkipList

Фрагмент III. Резюме лекции

Следующая лекция: Concurrent Skip List (2006 г.)

План (резюме) лекции, часть II из II

Продолжение: Структура данных «Список с пропусками» (1989 г.)

Последовательная версия SkipList

5. Алгоритм вставки
6. Алгоритм удаления (не сложный, похож на алгоритм вставки)
7. Алгоритм генерации № уровня для нового элемента (вероятность)
8. Алгоритм поиска i -го элемента

Фрагмент II. Структура данных «Concurrent Skip List» (2006 г.)

Параллельная версия SkipList

1. Блокировки: lock/unlock
2. Concurrent SkipList: структура узла
3. Параллельный алгоритм поиска узла
4. Параллельный алгоритм вставки узла
5. Особенности работы параллельного алгоритма вставки

* Фрагмент III. Анализ SkipList (приглашаю разобрать на консультации)

Фрагмент IV. Резюме лекции



NATIONAL RESEARCH
UNIVERSITY

Благодарю за внимание!

Рамон Антонио Родригес Залепинос
arodriges@hse.ru