Многопоточное Программирование: BSTs

Виталий Аксёнов, ИТМО, aksenov@itmo.ru
Роман Елизаров, JetBrains, elizarov@gmail.com
Никита Коваль, JetBrains, ndkoval@ya.ru

ИТМО 2019

Операции BST

- Insert(v)
  - True, если v не содержится
  - False, есть v содержится
- Remove(v)
  - True, если v содержится
  - False, если v не содержится
- Contains(v)
  - True, если v содержится
  - False, если v не содержится

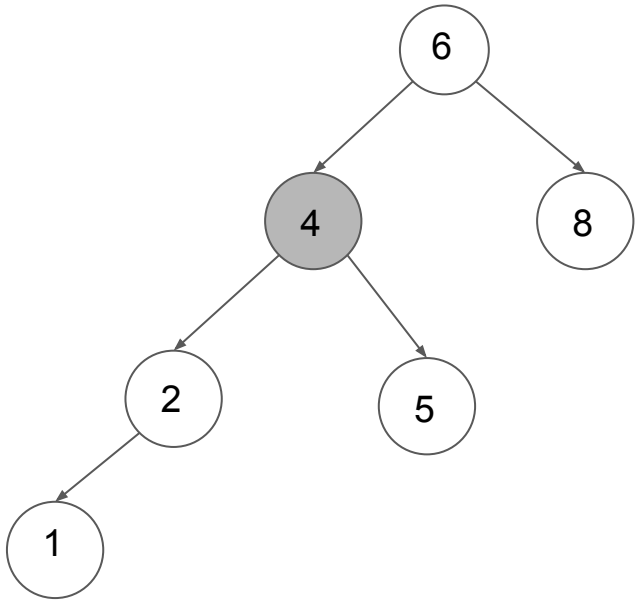
Виды BSTs

- External
- Partially External
- Internal

Partially-external BST

Свойства:

- Обычное свойство ключей бинарного дерева поиска.
- Два типа вершин: `DATA` (белые) или `ROUTING` (серые).
- Инвариант: `ROUTING` вершины всегда имеют два ребёнка.
- Множество состоит из вершин `DATA`.

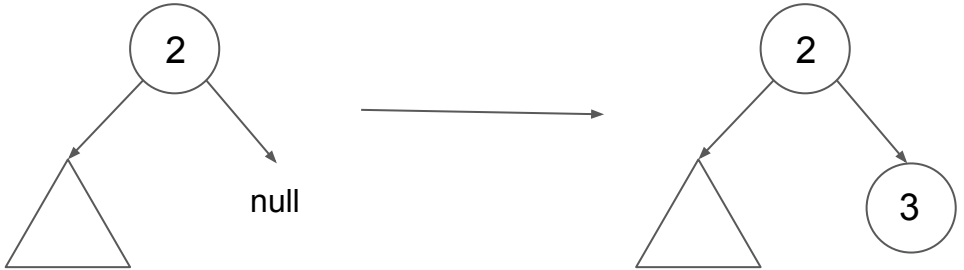


Partially-external BST. Traversal.

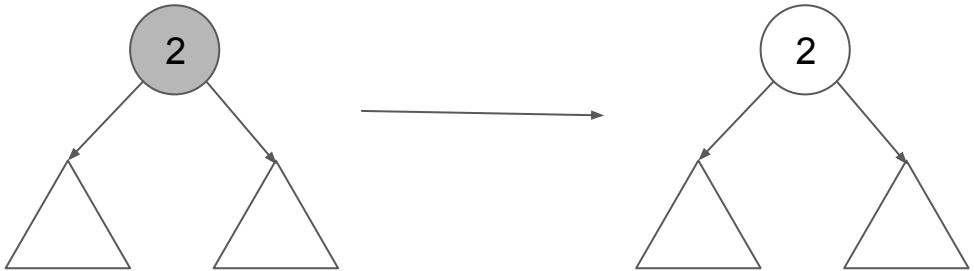
```
Window search(Node root, K key) {
    gprev = null, prev = null, curr = root
    while (curr.key != key && curr != null) {
        if (curr.key < key) {
            <gprev, prev, curr> = <prev, curr, curr.r>
        } else {
            <gprev, prev, curr> = <prev, curr, curr.l>
        }
    }
    return <gprev, prev, curr>
}
```

Partially-external BST. Insert.

Вставить лист.
`insert(3)`



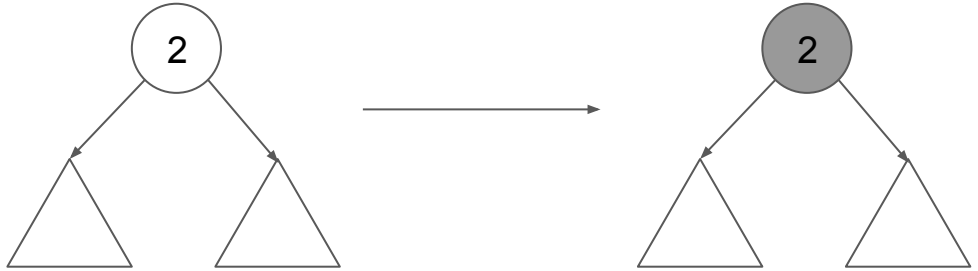
Вставка в ROUTING.
`insert(2)`



Partially-external BST. Remove. (I)

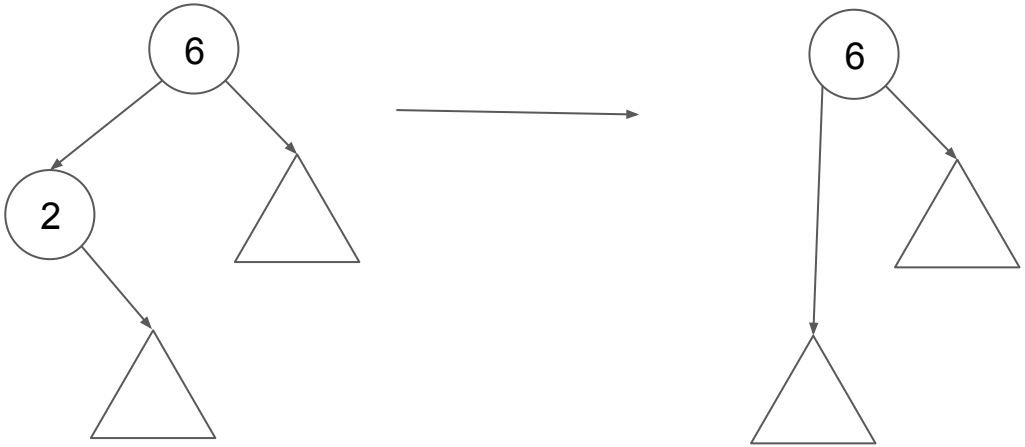
Удаление вершины с двумя детьми.

`remove(2)`



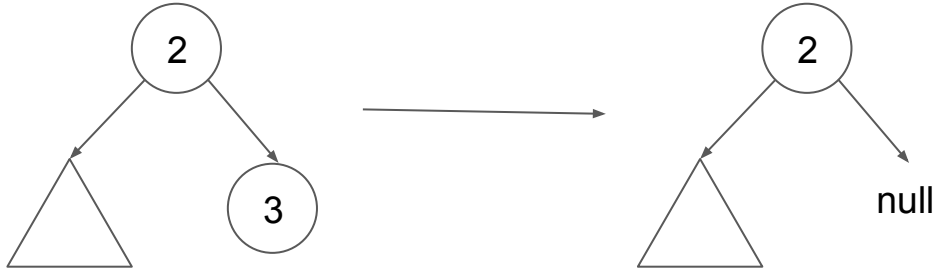
Удаление вершины с одним ребёнком.

`remove(2)`

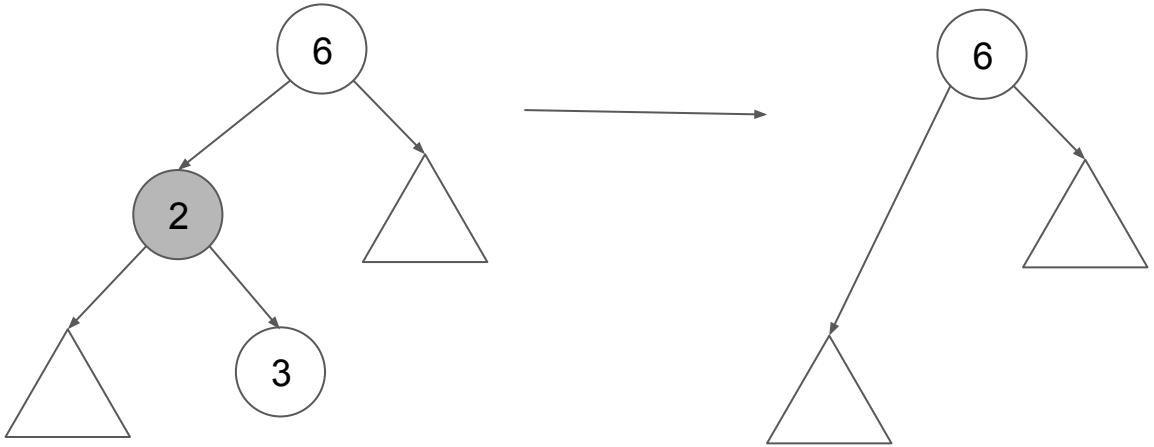


Partially-external BST. Remove. (II)

Удаление листа с DATA отцом.
`remove(3)`



Удаление листа с ROUTING отцом.
`remove(3)`



Concurrent partially-external BST. First try.

- Самый простой вариант написать всё на блокировках.
- Придётся иметь `deleted` поле для логического удаления.
- Две политики взятия блокировок:
  - Локи только на вершинах. Рёбра блокируются тоже.
  - Раздельные локи на состояние вершины и на рёбра.
- Какой вариант предпочтительнее?
- Ещё надо не забывать о порядке взятия блокировок, а то случится взаимная блокировка (deadlock). Мы же этого не хотим? Поэтому берём их сверху вниз.

Concurrent partially-external BST. First try.

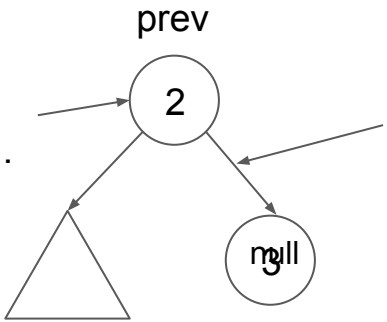
Как правильно делать блокировку? Вспоминаем беду со spin-lock: помимо постоянной попытки сделать CAS, лучше поспиниться.

Следует сделать приблизительно так:

```
void lock(lock, check) {
     if (!check()) {
          restart
     }
     lock.lock()
     if (!check()) {
          lock.unlock()
          restart
     }
}
```

Concurrent partially-external BST. First try. Example 1.

1. Блокируем состояние.
Проверяем `not deleted`.



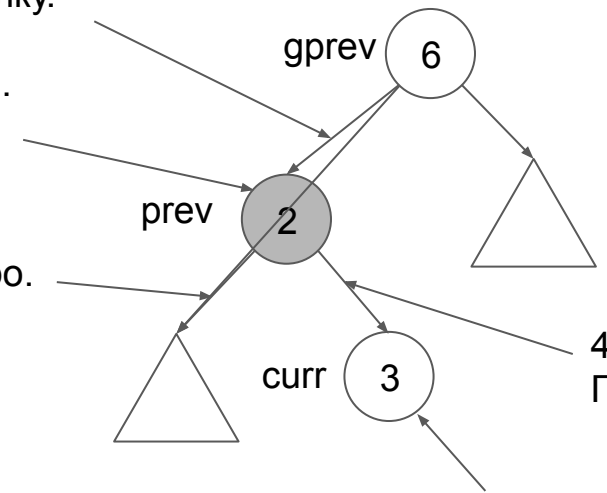
2. Блокируем правое ребро.
Сравниваем с `null`.

Concurrent partially-external BST. First try. Example 2.

1. Блокируем левое ребро.
Проверяем ссылку.

2. Блокируем состояние.
`state == ROUTING`
`not deleted`

3. Блокируем левое ребро.
Проверяем ссылку.



4. Блокируем правое ребро.
Проверяем ссылку.

5. Блокируем состояние.
`not deleted`
Проверяем, что лист.

But what about asymptotics?

- Предложенный алгоритм работает долго, если нагрузка не сбалансирована.
- Нужно добавить какую-то балансировку.
- Relaxed AVL-balancing [Bouge et al., Height-relaxed AVL rebalancing: …, 1998]
  - Храним в каждой вершине высоту левого и правого ребёнка.
  - Сравниваем их и поворачиваем, если надо.
  - Корректность: самый последний поток, который будет проходить снизу вверх подправит все высоты и корректно всё повернёт.

Partially-external balanced BST. Attempt 1.

[Bronson et al., A Practical Concurrent Binary Search Tree, 2010] сделали достаточно хитро и сложно, но не очень элегантно. Зато первые!

- `get(v)` работает как hand-over-hand locking, но optimistic. Мы двигаемся в нужном направлении, а затем проверяем, не испортил ли (повернул/удалил) кто-нибудь дерево между этими двумя вершинами.
- Удаление и повороты изменяют версию вершины. Повороты при этом работают хитро: битовой маской изменяют, в какую сторону повернулось.

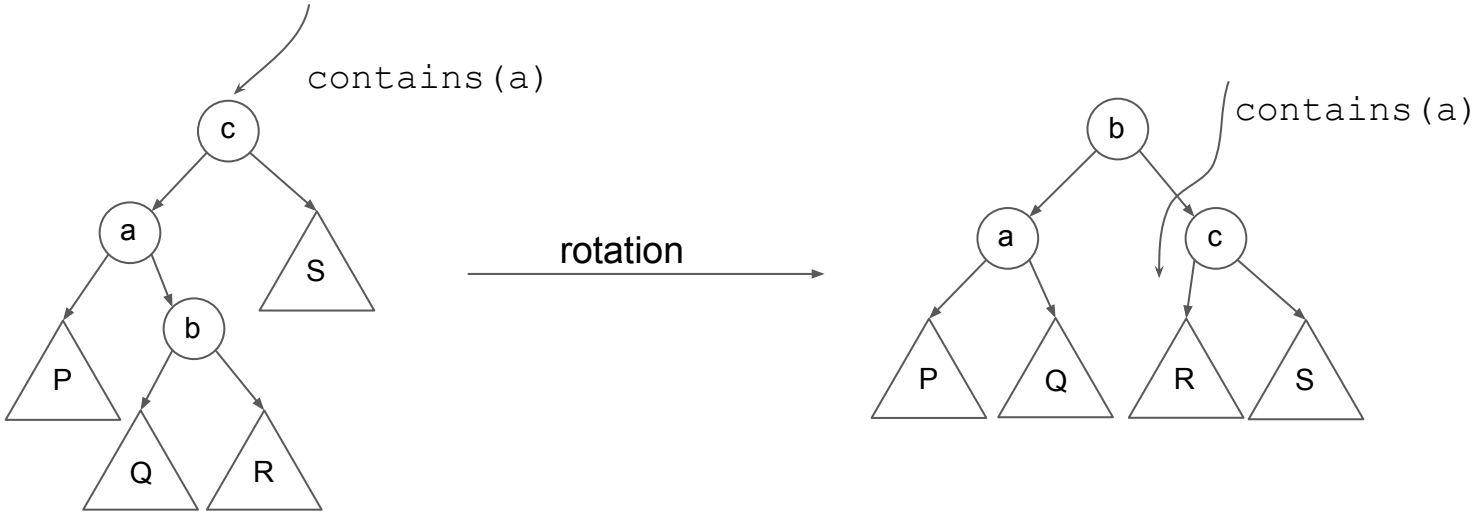
Partially-external balanced BST. Attempt 2.

Проблема предыдущего дерева?

- У нас `get` был не wait-free. А хочется…

Что мешало?

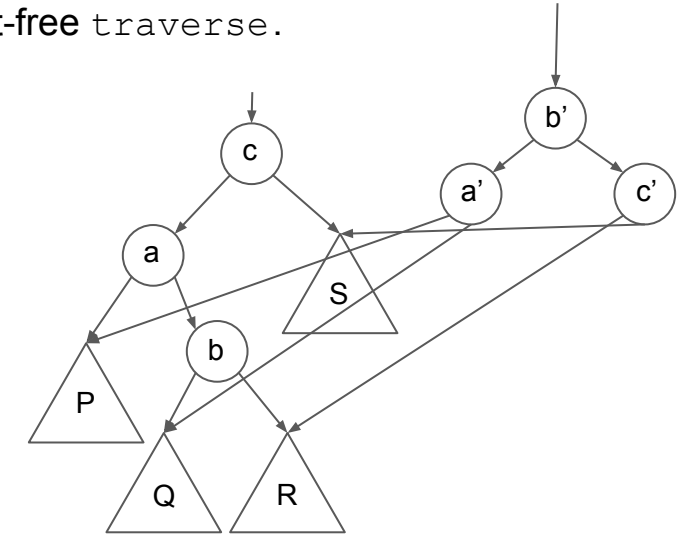
- Когда мы идём wait-free кто-то мог повернуть дерево, и мы пошли не туда, и можем не найти ключ, хотя он есть.



Partially-external balanced BST. Attempt 2.

[Crain et al., A Contention-Friendly Binary Search Tree, 2013] для разрешения этой проблемы предложили простую идеи: давайте при rotation брать блокировки на всё, а потом просто создавать новые вершины и их влинковывать.

Таким образом у нас получается wait-free `traverse`.



Partially-external balanced BST. Attempt 2.

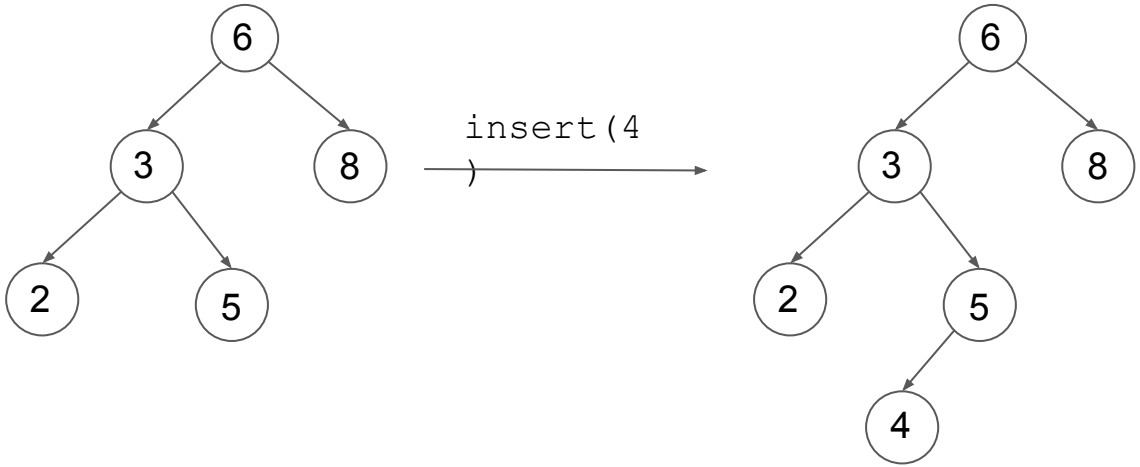
Что ещё было предложено?

- Разделение между основными процессами и фоновым процессом (daemon-ом).
- Основные процессы выполняют операции и просто помечают вершины.
- Daemon делает структурные изменения: периодически проходит по дереву, поддерживает ROUTING-инвариант и балансирует.
- Так как структурные изменения отделены, писать оказывается проще.

Internal BST

Самое обычное, стандартное дерево поиска.

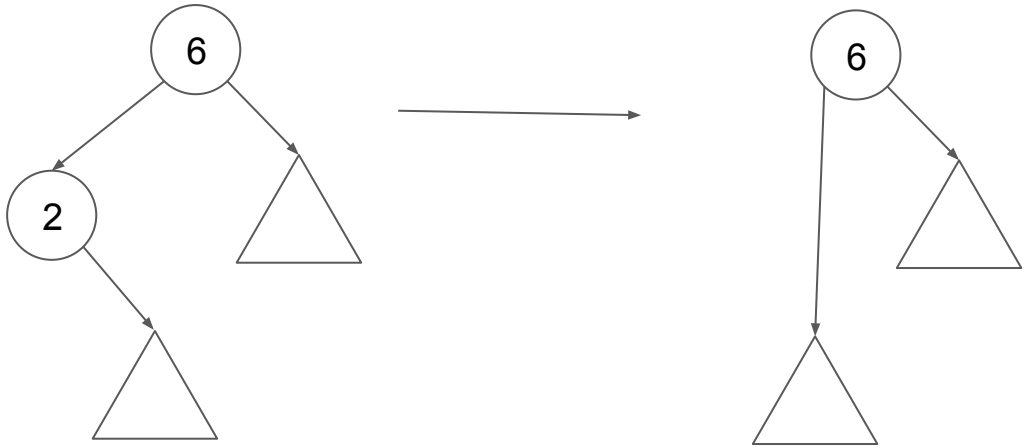
`insert` просто проходит путь и добавляет новый лист.



Internal BST. Remove. (I)

`remove` уже не такой локальный как в случае partially-external BST.

Удаление вершины с одним или нулём детей
`remove(2)`

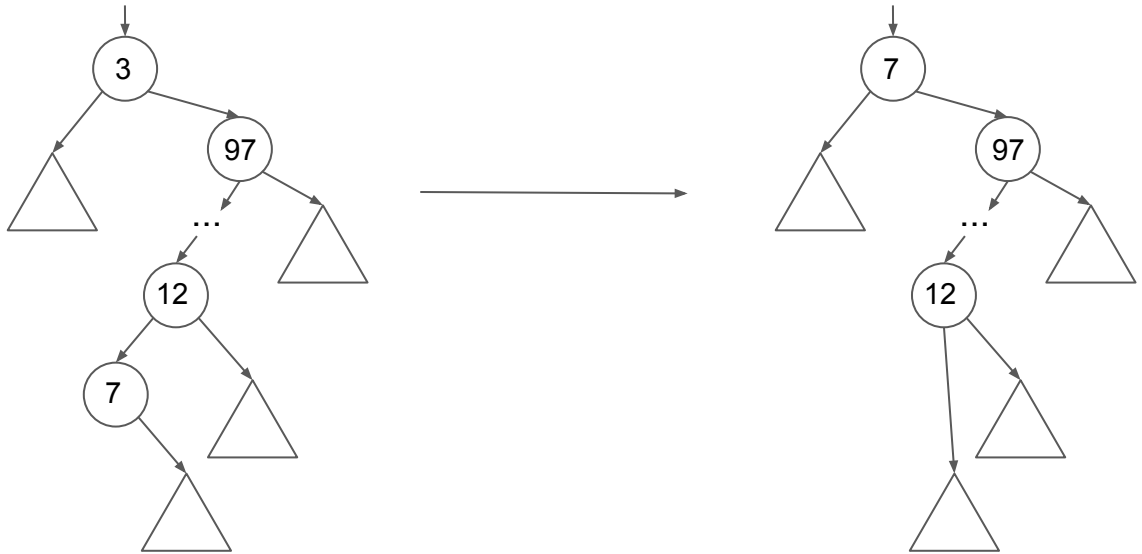


Internal BST. Remove. (II)

При удалении вершины с двумя детьми мы будем искать следующий ключ в правом поддереве и заменим на него.

Удаление вершины с 2 детьми

`remove(3)`



Internal BST. Attempt 1.

Опять будем пытаться сделать lock-based версию.

- Ищем спуском следующую вершину. Звучит неплохо, но:
  - пока искали, кто-то что-то поменял и вставилась вершинка с меньшим ключом.
  - wait-free `traverse` не сделать, потому что вершины умеют “улетать наверх”.
- Наивный способ:
  - заблокировать все вершины на пути.
  - делать hand-over-hand locking при `traverse`.
- Всё это медленно и неэффективно.
- А хорошо бы ещё сделать дерево сбалансированным...

Internal BST. Attempt 2.

[Drachsler et al., Practical Concurrent Binary Search Trees via Logical Ordering, 2014]

Что можно сделать с деревом, чтобы искать следующий ключ за O(1)?

- Его можно прошить связным списком!

Вау! Так это ещё помогает делать wait-free `traverse`. Но как?!

- Спускаемся вниз в поисках вершины. Дойдя до листа, двигаемся влево или вправо по списку.

Internal BST. Attempt 2.

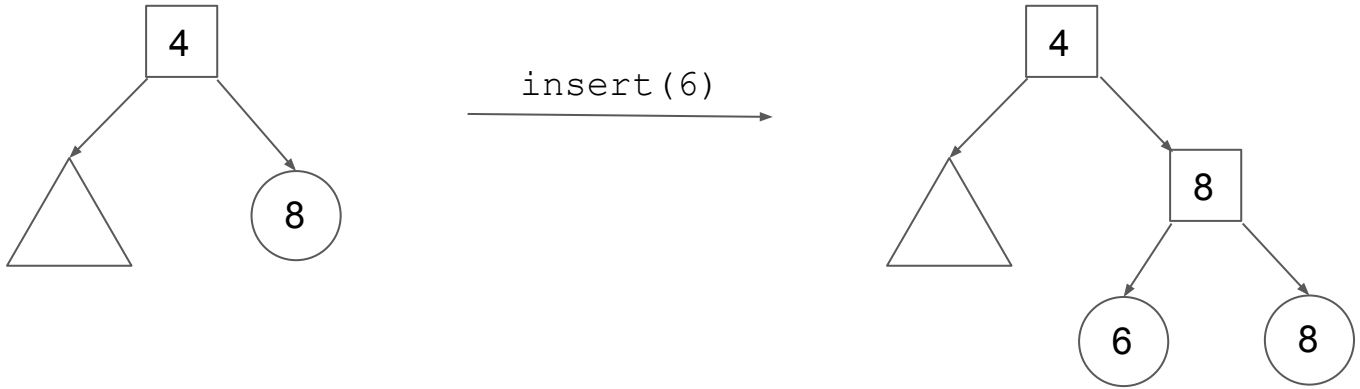
- Что ещё нужно? Да, в-общем-то ничего.
- `insert` и `remove` делается in a straightforward manner. Если что, `traverse` по списку сделает своё дело.
- Тоже самое и с балансировкой.
- Как элегантно! Но работает медленно, потому что надо ещё работать со списком.
- На удивление, представленный в статье алгоритм ещё и не линеаризуется… Ошибка найдена в 2019 и поправлена. Не верьте статьям на слово!

Lock-free BSTs

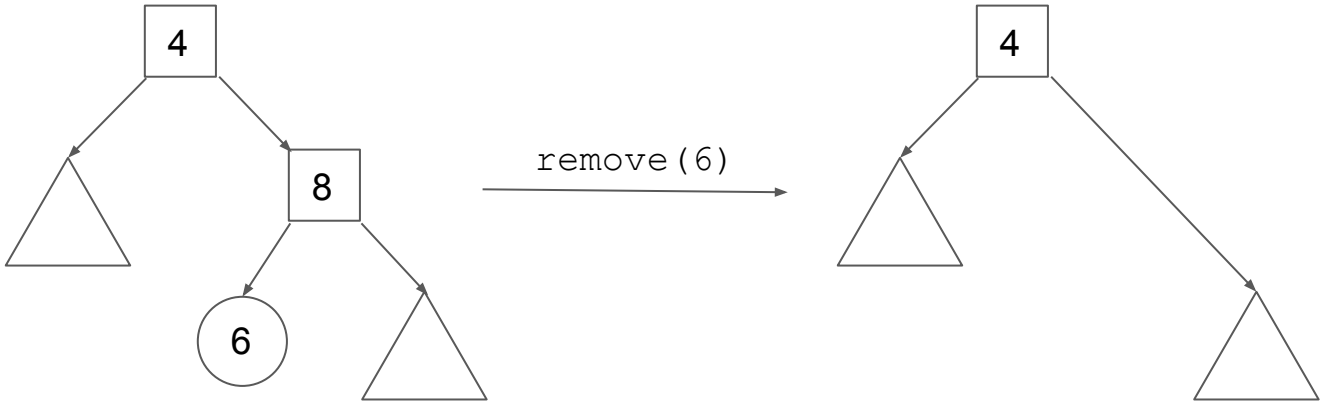
- Пока были разговоры только про lock-based деревья. Неужели нет lock-free?
- Конечно же есть, но их мало и они на порядок сложнее…
- Partially-external и internal деревья поиска выглядят как-то сложновато для того, чтобы сделать их lock-free.
- Благо для этого есть external деревья поиска, которые на порядок проще...

External BST

Ключи хранятся только в листах. Все внутренние вершины хранят только routing информацию для `traverse`.



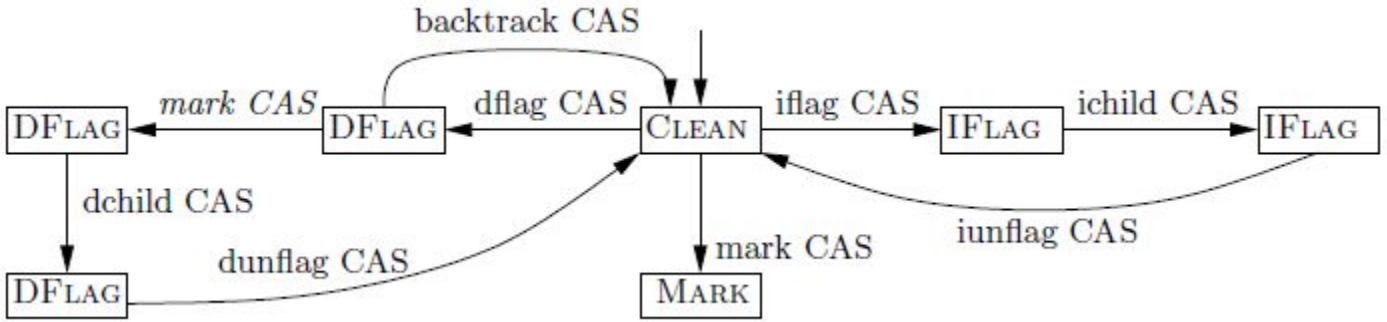
External BST. Remove.



Lock-free external BST. Attempt 1.

Первая попытка: [Ellen et al., Non-blocking Binary Search Trees, 2010].

- На самом деле ничего хитрого. Просто используем дескрипторы, которые храним в вершинах. Когда видим дескриптор - помогаем операции завершиться, и только потом выполняем свою.
- Но блок-схема состояний не то, чтобы шибко простая.
- Хорошо, хоть wait-free `traverse` в external деревьях спокойно выполняется, и `remove` на него никак не влияет.



Lock-free BST. Attempt 2.

Предыдущий алгоритм работает достаточно долго:

- слишком много нужно сделать операций с дескрипторами.
- дескрипторы хранятся только в вершинах, поэтому естественный contention.

[Natarajan and Mittal, Fast Concurrent Lock-Free Binary Search Trees, 2014] придумали, как можно реализовать дескрипторы более эффективно используя только два булевых флага на рёбрах.

More Lock-free BSTs

Я немного обманул. Есть и lock-free internal деревья. Но они достаточно сложны идейно, т.е. используют чуть более нетривиальные факты про internal деревья.

- [Howley and Jones, A Non-Blocking Internal Binary Search Tree, 2012]
- [Ramachandran and Mittal, A Fast Lock-Free Internal Binary Search Tree, 2015]

Lock-free Balanced BSTs

- Погодите-ка! Пока все деревья какие-то несбалансированные. Как же добавить балансировку?! Вот тут-то и начинаются самые сложности.
- Самый очевидный способ сделать всё и сразу lock-free структуру дерево - это CASN. Но вот известно, что масштабируется не очень хорошо.
- А кто-нибудь сделал хорошо работающее сбалансированное lock-free дерево?
- Да! [Brown et al., A General Technique for Non-blocking Trees, 2014] На деле External Chromatic BST.
- А как сделал? Почти CASN только чуть более легковесно - LLX и SCX.
    - Легковесно, например, потому что есть способ переиспользовать регистры. [Arbel-Raviv and Brown, Reuse, don’t Recycle: …, 2017]

LL and SC

- Я не буду рассказывать, что такое LLX и SCX.
- Но стоит знать, что такое однорегистровые операции `LL` (load-linked) and `SC` (store-conditional).
- `LL` - считывает ячейку регистра.
- `SC` - сохраняет новое значение в регистре, если ничего в ней не поменялось с момента вызова `LL`.
- Кажется, что операция чуть более мощная, чем CAS, например, исключает ABA проблему.
- Реализована на некоторых архитектурах, например, PowerPC (IBM) и ARM. Но реализована weak-версия: `SC` может упасть, даже если всё хорошо.

CAS through LL/SC

Представлено в статье [Anderson and Moir, Universal Constructions for Multi-Object Operations, 1995].

```
shared X
boolean CAS(old, new) {
    if (LL(X) != old) return false
    if (old == new) return true
    return SC(X, new)
}
```

LL/SC through CAS

```
llsctype = {value: Type; tag, pid: int}

shared X: llsctype
       A: array [1..N] of llsctype
private old, chk: llsctype
        j, newtag: int

Type LL() {
     old = X
     A[p] = old
     chk = X
     return old.value
}
```

```
boolean SC(val: Type) {
     if (chk != old) return false
     read A[j].tag
     if (j = N) { j = 1 } else { j = j + 1 }
     select newtag ∉ {last N tags read,
                     last N tags selected,
                     last tag succ. CAS'd}
     return CAS(X, old, {val, newtag, p})
}
```

Java-ists should suffer!

Основная беда большинства lock-free алгоритмов в том, что они используют стандартную технику: давайте сожрём битик там, сожрём битик сям.

На Java это превращается в огромную боль.

Например, я видел только реализацию [Ellen et al.] на Java. (может плохо искал?)

Wait-free BSTs

С wait-free деревьями всё ещё хуже, чем с lock-free. Но оно существует!

[Natarajan et al., Concurrent wait-free red black trees, 2013].

Основано на базе top-down красно-чёрного дерева.

Есть окно, которое спускается вниз. Если окна пересекаются, то мы помогаем нижнему окну.

Мы копируем окно, всё меняем и копируем назад.

What is with performance?

Если смотреть только на Java, то без балансировки выигрывает самое первое простое дерево, а с балансировкой выигрывает [Crain et al.] (дерево с daemon-потоком).

Что касается всех, то без балансировки выигрывает всех [Natarajan et al.], а с балансировкой у нас есть только одно.

Есть неполное сравнение алгоритмов [Arbel-Raviv et al., Getting to the Root of Concurrent Binary Search Tree Performance, 2018].

На самом деле все BST достаточно тухлые. Кто же лидер?

Lock-free relaxed (a, b)-tree [Brown, Techniques for Constructing Efficient Lock-free Data Structures, Section 8, 2017]