Concurrent Data Structures Made Easy

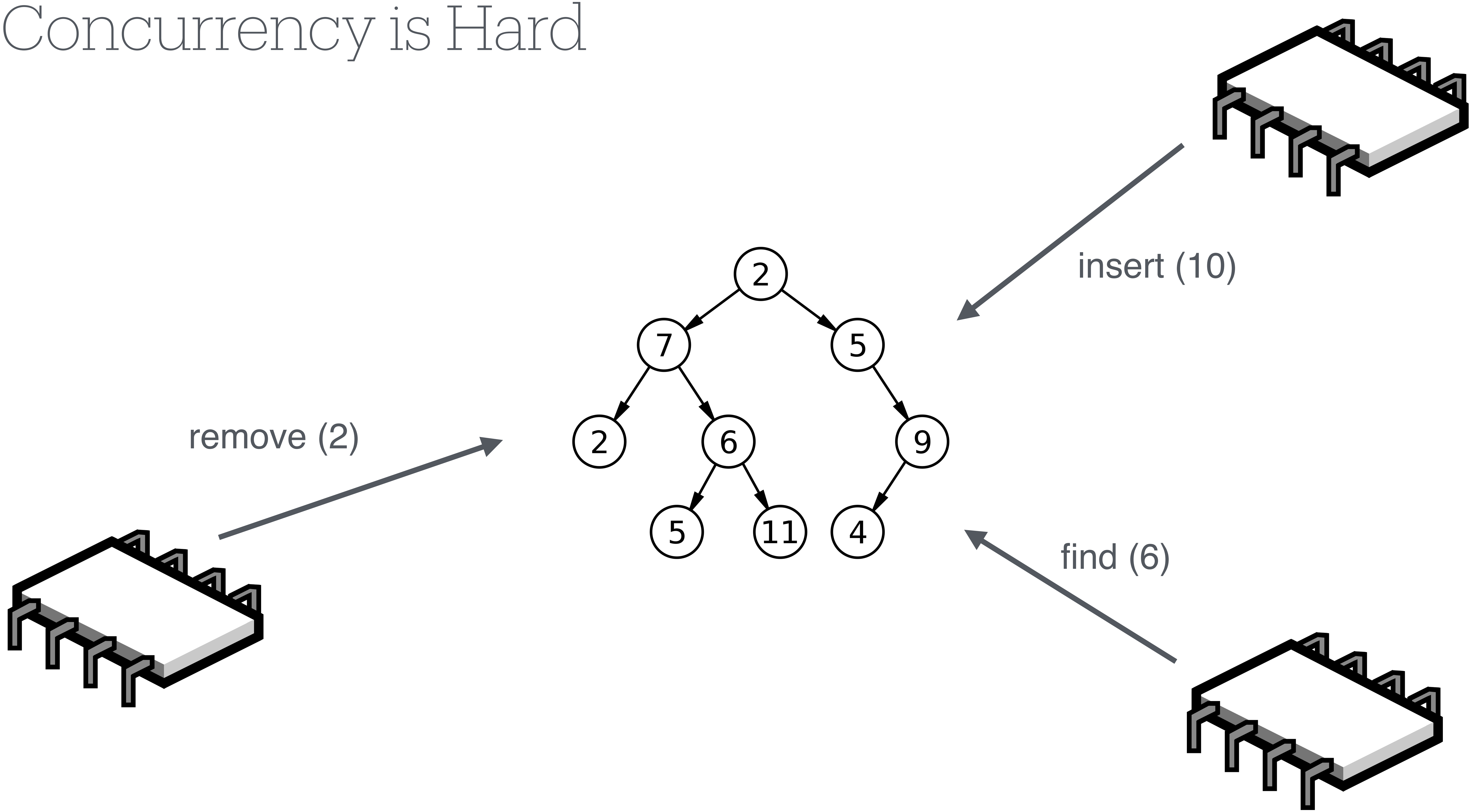
Ilya Sergey

ilyasergey.net

National University of Singapore

joint work with Callista Le, Koon Wen Lee, Kiran Gopinathan, and Seth Gilbert

Concurrency is Hard

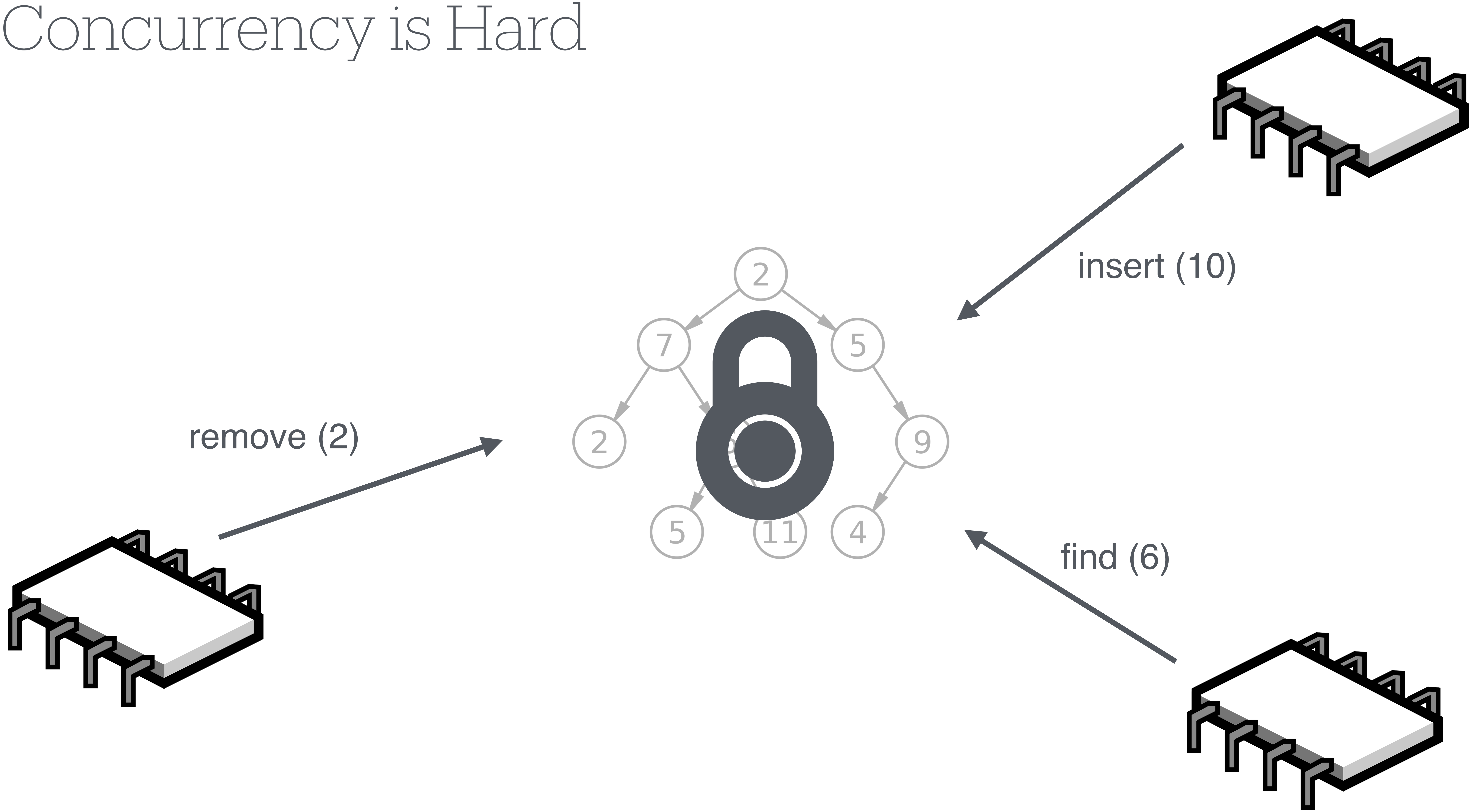


insert (10)

remove (2)

find (6)

Concurrency is Hard



insert (10)

remove (2)

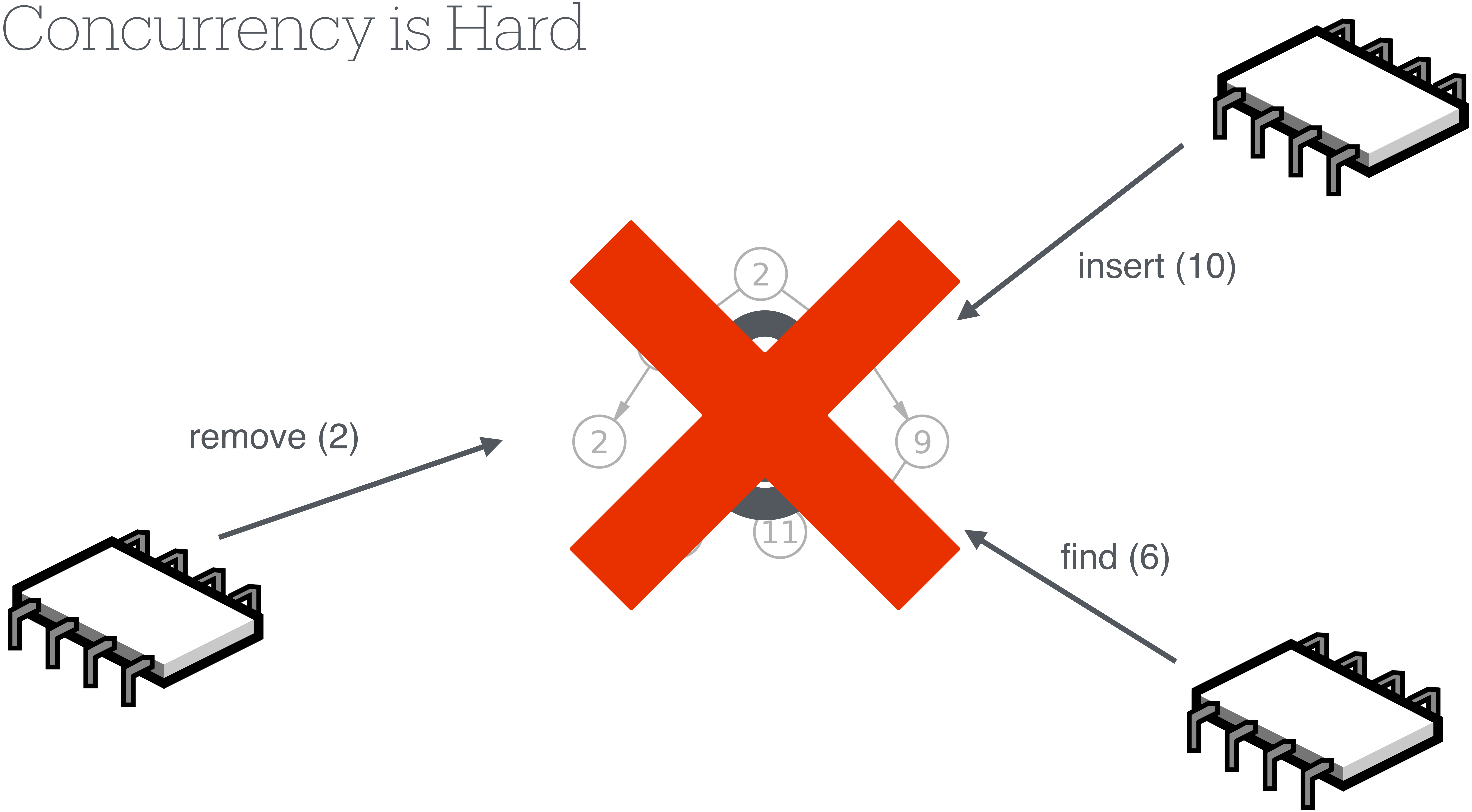
find (6)

Concurrency is Hard

Coarse-grained concurrency

- **Advantages**:
  - Easy to implement
  - Immediately thread-safe
- **Disadvantages**:
  - High lock contention
  - No parallelisation

Concurrency is Hard

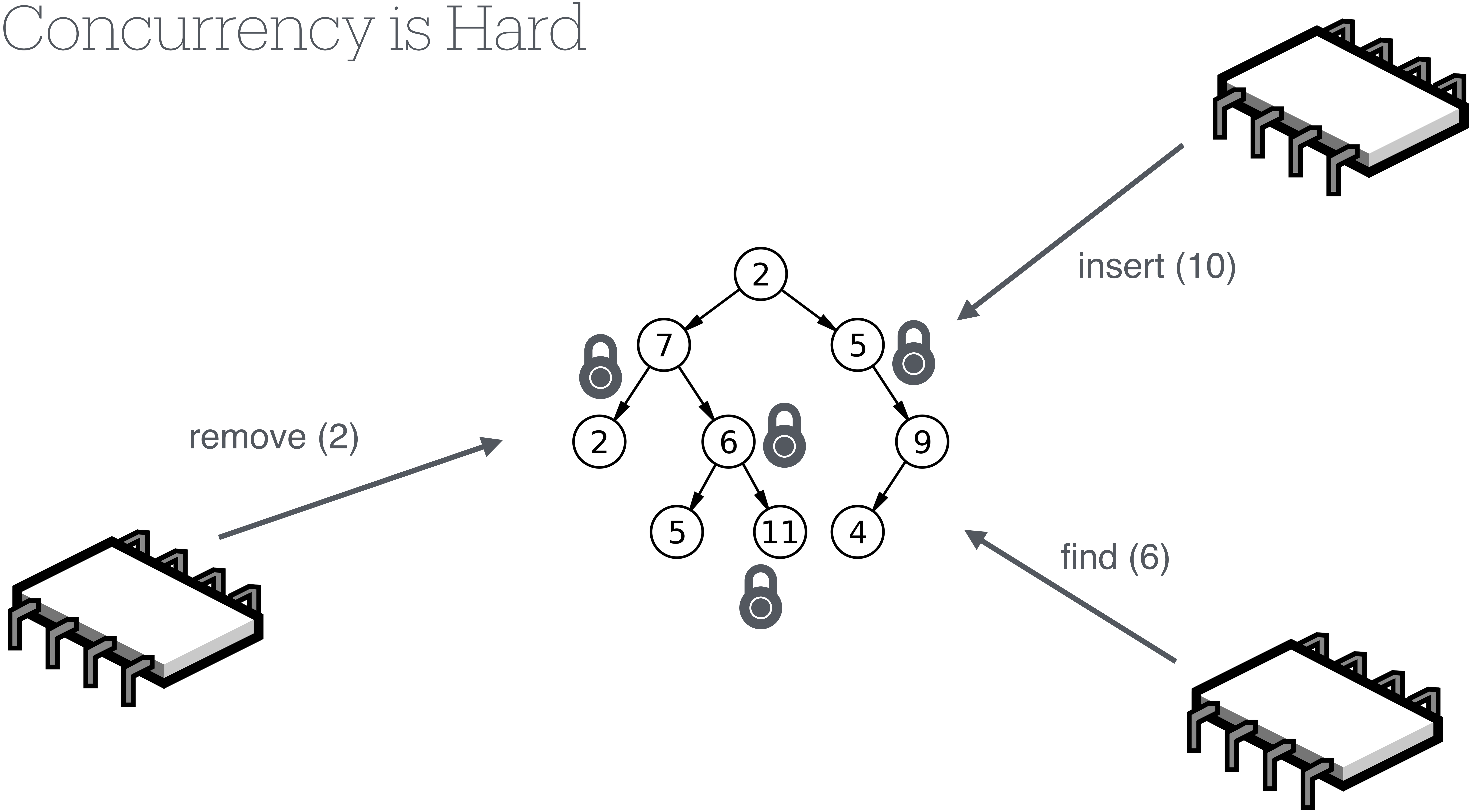


insert (10)

remove (2)

find (6)

Concurrency is Hard



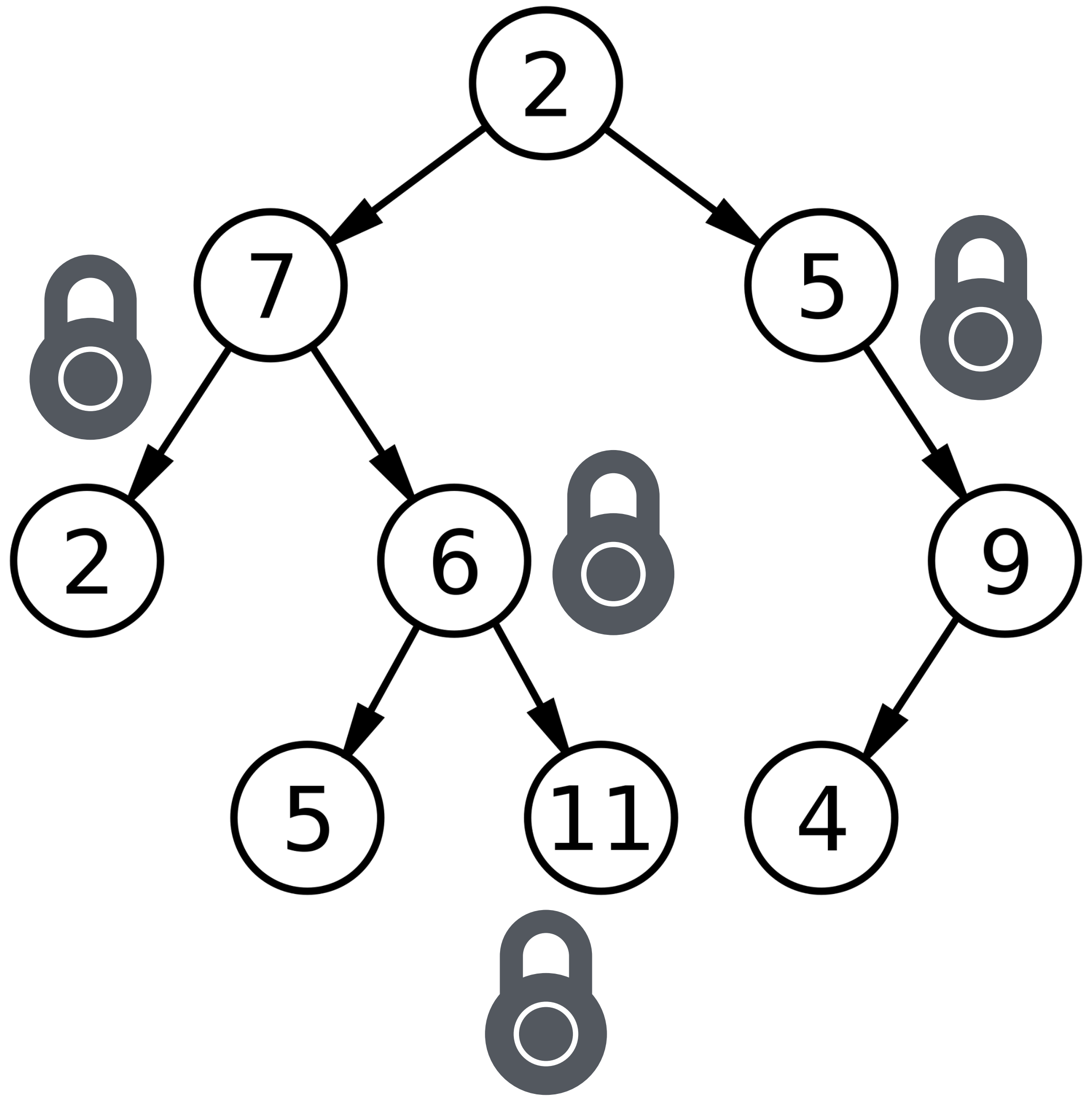
insert (10)

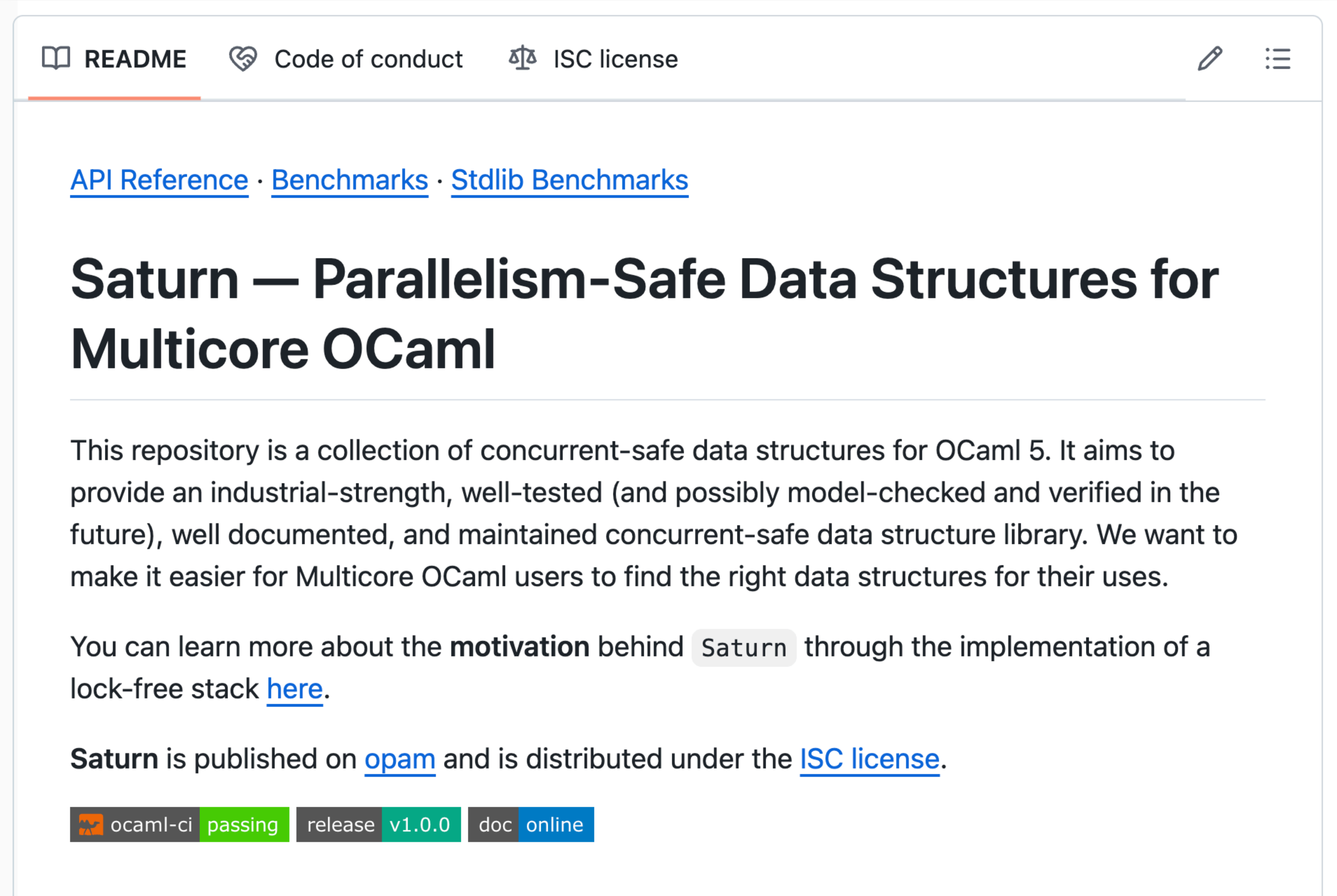
remove (2)

find (6)

Concurrency is Hard

Fine-grained Concurrency





API Reference · Benchmarks · Stdlib Benchmarks

Saturn — Parallelism-Safe Data Structures for Multicore OCaml

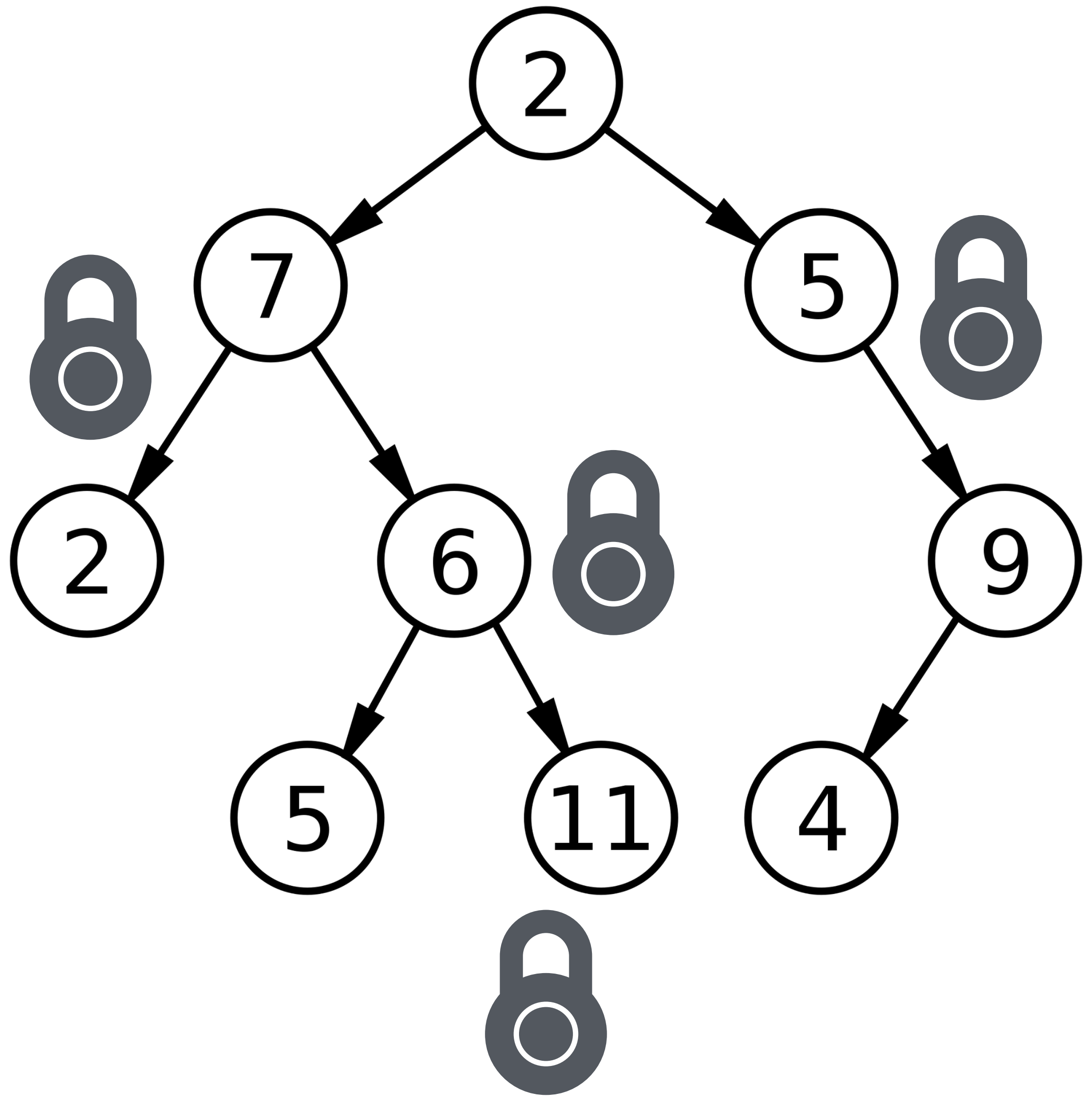
This repository is a collection of concurrent-safe data structures for OCaml 5. It aims to provide an industrial-strength, well-tested (and possibly model-checked and verified in the future), well documented, and maintained concurrent-safe data structure library. We want to make it easier for Multicore OCaml users to find the right data structures for their uses.

You can learn more about the **motivation** behind `Saturn` through the implementation of a lock-free stack here.

Saturn is published on opam and is distributed under the ISC license.

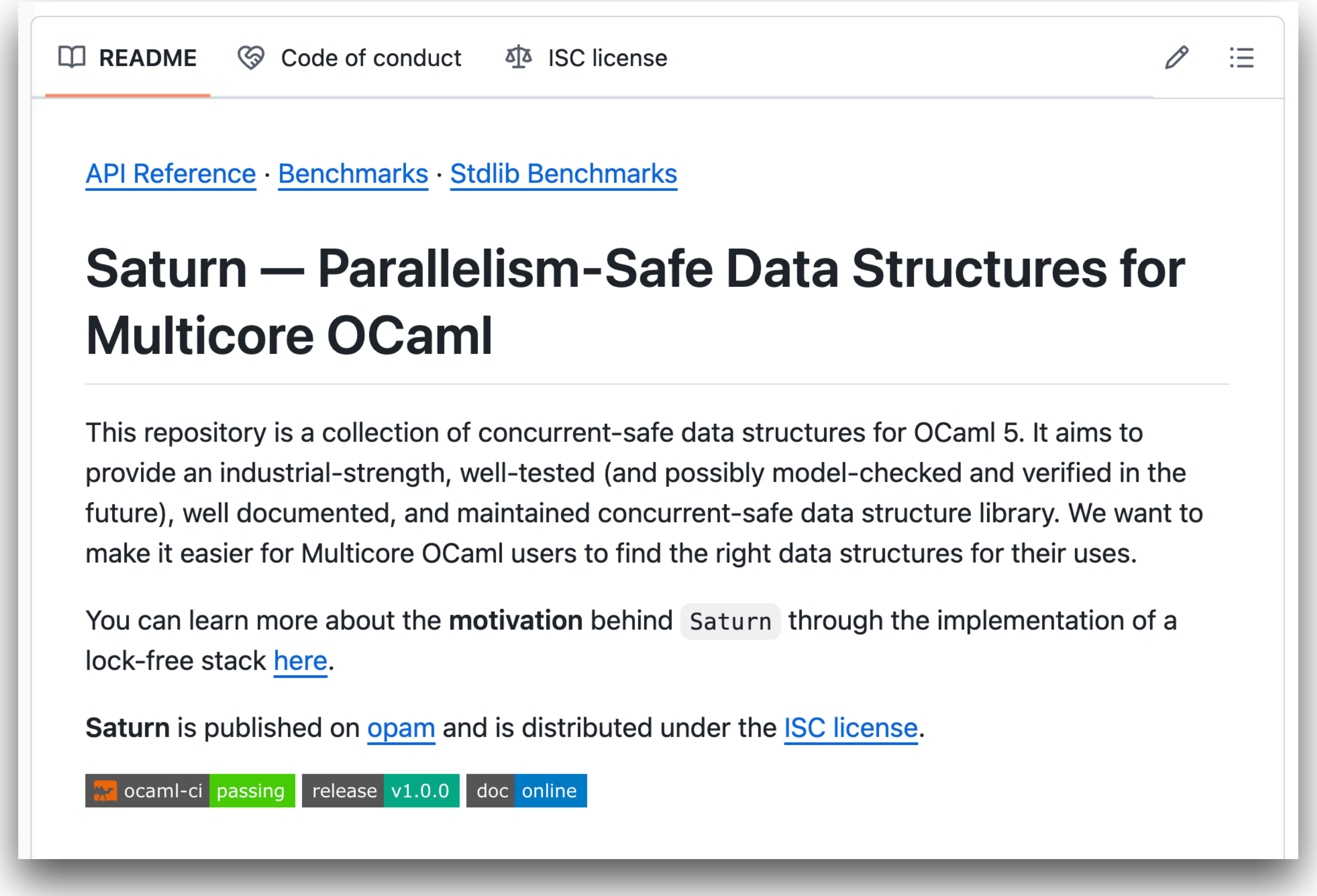
Concurrency is Hard

Fine-grained Concurrency



- **Advantages**:
  - High degree of parallelism
  - Little contention

- **Disadvantages**:
  - Hard to design
  - Hard to reason about
  - Hard to debug



API Reference · Benchmarks · Stdlib Benchmarks

Saturn — Parallelism-Safe Data Structures for Multicore OCaml

This repository is a collection of concurrent-safe data structures for OCaml 5. It aims to provide an industrial-strength, well-tested (and possibly model-checked and verified in the future), well documented, and maintained concurrent-safe data structure library. We want to make it easier for Multicore OCaml users to find the right data structures for their uses.

You can learn more about the **motivation** behind `Saturn` through the implementation of a lock-free stack here.

Saturn is published on opam and is distributed under the ISC license.

Concurrency is Hard

MUL
PRO
Mauric

Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris

Ike Mulder
Radboud University Nijmegen
The Netherlands
me@ikemulder.nl

Robbert Krebbers
Radboud University Nijmegen
The Netherlands
mail@robbertkrebbers.nl

Herman Geuvers
Radboud University Nijmegen and
Eindhoven University of Technology
The Netherlands
herman@cs.ru.nl

Abstract

Fine-grained concurrent programs are difficult to get right, yet play an important role in modern-day computers. We want to prove strong specifications of such programs, with minimal user effort, in a trustworthy way. In this paper, we present **Diaframe**—an *automated* and *foundational* verification tool for fine-grained concurrent programs.

Diaframe is built on top of the Iris framework for higher-order concurrent separation logic in Coq, which already has a foundational soundness proof and the ability to give strong specifications, but lacks automation. Diaframe equips Iris with strong automation using a novel, extendable, goal-directed proof search strategy, using ideas from linear logic programming and bi-abduction. A benchmark of 24 examples from the literature shows that the proof burden of Diaframe is competitive with existing non-foundational tools, while its expressivity and soundness guarantees are stronger.

CCS Concepts: • **Theory of computation** → **Separation logic**; **Automated reasoning**; **Program verification**.

Keywords: Separation logic, fine-grained concurrency, proof automation, Iris, Coq

ACM Reference Format:
Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523432

1 Introduction

Fine-grained concurrent programs, such as locks, reference counters, barriers, and queues, play a critical role in modern day programs and operating systems. Based on 15 years of research on concurrent separation logic [12, 13, 25, 29, 30, 32
35, 48, 67, 68, 74, 80, 81, 85–89], it has become possible t
verify increasingly complicated versions of such program
Yet, while several tools for verification of fine-grained co
current programs based on these logics exist, none of the
are both *automated* (the majority of the proof work is carrie
out by the tool) and *foundational* (a closed proof w.r.t. th
operational semantics is produced in a proof assistant).

Tools with good automation like Caper [31], Starling [9
and Voila [91], generally use SMT [27] or separation-log
solvers [65, 73] as trusted oracles. They are capable of pro
ing programs correct with relatively little help from th
user, allowing quick experimentation when designing alg
rithms. However, they have a large *trusted computing base*
one needs to trust their implementation, the used solvers, th
translation of the required side conditions to the used solve
and sometimes also the soundness of the underpinned log
In particular, the results of such tools do not come wit
closed proofs that can be checked independently.

Foundational tools like Iris [45, 46, 48, 52], FCSL [77] an
VST [3, 17] are embedded in a proof assistant. Hence, o
only needs to trust the implementation of the proof assista
and the operational semantics of the programming langua
but not the solvers or underpinned logic. Foundational too
typically provide tactics [2, 6, 17, 51, 53, 60] to hide low-lev
proofs, but the bulk of the proof work needs to be spelle
out. There are two reasons for this status quo. First, found
tional tools cannot rely on trusted oracles, unless proofs a
reconstructed so that the proof assistant can verify them i
dependently. Second, foundational tools usually have a ri
logic that can prove strong specifications, *e.g.*, using impre
icative invariants [80], for which automation has receiv
little attention, even in a non-foundational setting.

In this paper, we present **Diaframe**—a foundational to
for automatic verification of fine-grained concurrent pr
grams. Diaframe extends Iris [45, 46, 48, 52]—a framewo
for interactive proofs in higher-order impredicative conc
rent separation logic in Coq—with powerful tactics to pe
form the bulk of the proof work automatically. This mea
we get the best of both worlds: closed proofs to underpin o
results, while needing relatively little help from the user.

An overview of the architecture of Diaframe is display
in Figure 1. Diaframe takes two inputs from the user (mark

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523432

Proving Highly-Concurrent Traversals Correct

YOTAM M. Y. FELDMAN, Tel Aviv University, Israel
KHYZHA, Tel Aviv University, Israel
NTIN ENEA, IRIF, Université de Paris, France
ORRISON, Tel Aviv University, Israel

Mechanized Verification of Fine-grained Concurrent Programs

Ilya Sergey
IMDEA Software Institute
ilya.sergey@imdea.org

Aleksandar Nanevski
IMDEA Software Institute
aleks.nanevski@imdea.org

Anindya Banerjee
IMDEA Software Institute
anindya.banerjee@imdea.org

Abstract

Efficient concurrent programs and data structures rarely employ coarse-grained synchronization mechanisms (*i.e.*, locks); instead, they implement custom synchronization patterns via fine-grained primitives, such as *compare-and-swap*. Due to sophisticated interference scenarios between threads, reasoning about such programs is challenging and error-prone, and can benefit from mechanization.

In this paper, we present the first completely formalized framework for mechanized verification of *full functional correctness* of fine-grained concurrent programs. Our tool is based on the recently proposed program logic FCSL. It is implemented as an embedded domain-specific language in the dependently-typed language of the Coq proof assistant, and is powerful enough to reason about programming features such as higher-order functions and local thread spawning. By incorporating a uniform concurrency model, based on *state-transition systems* and *partial commutative monoids*, FCSL makes it possible to build proofs about concurrent libraries in a thread-local, compositional way, thus facilitating scalability and reuse: libraries are verified *just once*, and their specifications are used ubiquitously in client-side reasoning. We illustrate the proof layout in FCSL by example, and report on our experience of using FCSL to verify a number of concurrent programs.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, Theory, Verification

Keywords Compositional program verification, concurrency, separation logic, mechanized proofs, dependent types.

1. Introduction

It has been long recognized that efficient concurrency is of crucial importance for high-performant software. Unfortunately, proving correctness of concurrent programs, in which several computations can be executed in parallel, is difficult due to the large number of possible interactions between concurrent processes/threads on shared data structures.

One way to deal with the complexity of verifying concurrent code is to employ the mechanisms of so-called *coarse-grained* synchronization, *i.e.*, locks. By making use of locks in the code, the programmer ensures mutually-exclusive thread access to critical resources, therefore, reducing the proof of correctness of concurrent code to the proof of correctness of *sequential* code. While sound, this approach to concurrency prevents one from taking full advantage of parallel computations. An alternative is to implement shared data structures in a *fine-grained* (*i.e.*, lock-free) manner, so the threads manipulating such structures would be reaching a consensus via the active use of non-blocking read-modify-write operations (*e.g.*, compare-and-swap) instead of locks.

Despite the clear practical advantages of the fine-grained approach to the implementation of concurrent data structures, it requires significant expertise to devise such structures and establish correctness of their behavior.

In this paper, we focus on *program logics* as a generic approach to specify a program and formally prove its correctness *wrt.* the given specification. In such logics, program specifications (or specs) are represented by Hoare triples $\{P\}\ c\ \{Q\}$, where c is a program being described, P is a precondition that constrains a state in which the program is safe to run, and Q is a postcondition, describing a state upon the program's termination. Modern logics are sufficiently *expressive*: they can reason about programs operating with first-class executable code, locally-spawned threads and other features omnipresent in modern programming. Verifying a program in a Hoare-style program logic can be done *structurally*, *i.e.*, by means of systematically applying syntax-directed inference rules, until the spec is proven.

Importantly, logic-based verification of fine-grained concurrency requires reasoning about a number of concepts that don't have direct analogues in reasoning about sequential or coarse-grained concurrent programs:

(1) **Custom resource protocols.** Each shared data structure (*i.e.*, a *resource*) that can be used by several threads concurrently, requires a specific "evolution protocol", in order to enforce preservation of the structure's consistency. In contrast to the coarse-grained case, where the protocol is fixed to be locking/unlocking, a fine-grained resource comes with its own notion of consistency and protocol.
(2) **Interference and stability.** Absent locking, local reasoning about a shared resource from a single thread's perspective should manifest the admissible changes that can be made by other threads that interfere with the current one. Every thread-local assertion about a fine-grained data structure's state should be *stable*, *i.e.*, invariant under possible concurrent modifications of the resource.
(3) **Helping.** This concurrent pattern appears in fine-grained programs due to relaxing the mutual exclusion policy; thus several threads can simultaneously operate with a single shared resource. The "helping" happens when a thread is scheduled for a task involving the resource, but the task is then accomplished by *another* thread; however, the result of the work, once the task is completed, is ascribed to the initially assigned thread.

In addition, Hoare-style reasoning about coarse- or fine-grained concurrency requires a form of (4) **auxiliary state** to partially expose the internal threads' behavior and relate local program asser-

PLDI'15, June 13–17, 2015, Portland, OR, USA.

http://dx.doi.org/10.1145/2737924.2737964

A Concurrent Program Logic with a Future and History

EYER, TU Braunschweig, Germany
IES, New York University, USA
WOLFF, New York University, USA

grained optimistic concurrent programs remains an open problem. Modern program logics
tion mechanisms and compositional reasoning principles to deal with the inherent complexity.
use is mostly confined to pencil-and-paper or mechanized proofs. We devise a new separation
vards the lacking automation. While local reasoning is known to be crucial for automation, we
show how to retain this locality for (i) reasoning about inductive properties without the need
and (ii) reasoning about computation histories in hindsight. We implemented our new logic
ed it to automatically verify challenging concurrent search structures that require inductive
hindsight reasoning, such as the Harris set.

• **Theory of computation** → **Separation logic**; **Hoare logic**; **Automated reasoning**;
ation; Programming logic.

Words and Phrases: Linearizability, Non-blocking Data Structures, Harris Set

ce Format:
Thomas Wies, and Sebastian Wolff. 2022. A Concurrent Program Logic with a Future and History.
ram. Lang. 6, OOPSLA2, Article 174 (October 2022), 30 pages. https://doi.org/10.1145/3563337

UCTION

comes at a cost, at least, in terms of increased effort when verifying program cor-
re has been a proliferation of concurrent program logics that provide an arsenal of
hniques to address this challenge [Bell et al. 2010; Delbianco et al. 2017; Elmas et al.
2010; Gotsman et al. 2013; Gu et al. 2018; Hemed et al. 2015; Jung et al. 2018; Liang
3; Manna and Pnueli 1995; Parkinson et al. 2007; Sergey et al. 2015; Vafeiadis and
7]. In addition, a number of general approaches have been developed to help structure
proof argument [Feldman et al. 2018, 2020; Kragl et al. 2020; O'Hearn et al. 2010;
oodman 1988]. However, the use of these techniques has been mostly confined to
s done on paper, or mechanized proofs constructed in interactive proof assistants.
these works a concurrent separation logic suitable for automating the construction
ctness proofs for highly concurrent data structures. We focus on concurrent search
s and maps indexed by keys), but the developed techniques apply more broadly. Our
ple is to perform all inductive reasoning, both in time and space, in lock-step with the
ution. The reasoning about inductive properties of graph structures and computation
legated to the meta-theory of the logic by choosing appropriate semantic models.

mple. We motivate our work using the Harris non-blocking set data structure [Harris
we will also use as a running example throughout the paper.

s: Roland Meyer, TU Braunschweig, Germany, roland.meyer@tu-bs.de; Thomas Wies, New York University,
u.edu; Sebastian Wolff, New York University, USA, sebastian.wolff@cs.nyu.edu.

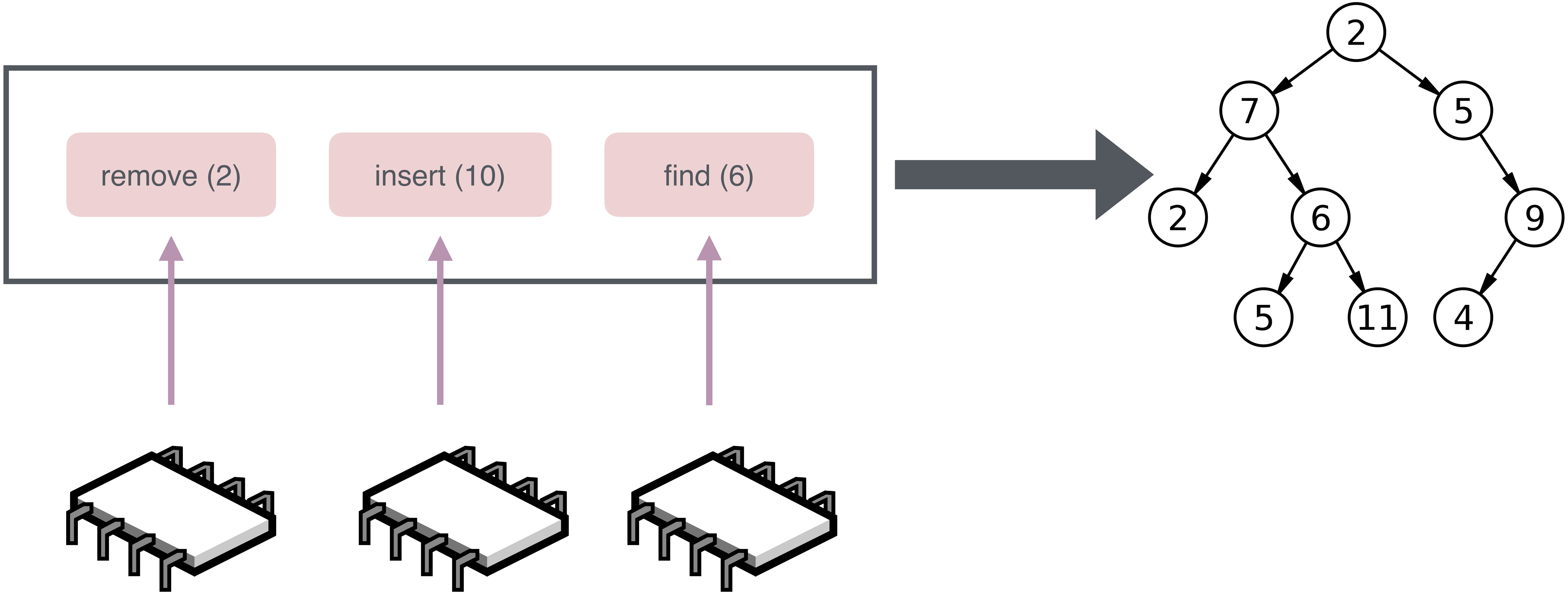
0-ART174
.1145/3563337

Batch Parallelism

Batch Parallelism

Insight:
Handling a batch of ***known operations***
is ***easier*** than
handling a stream of ***arbitrary operations***

Batch Parallelism

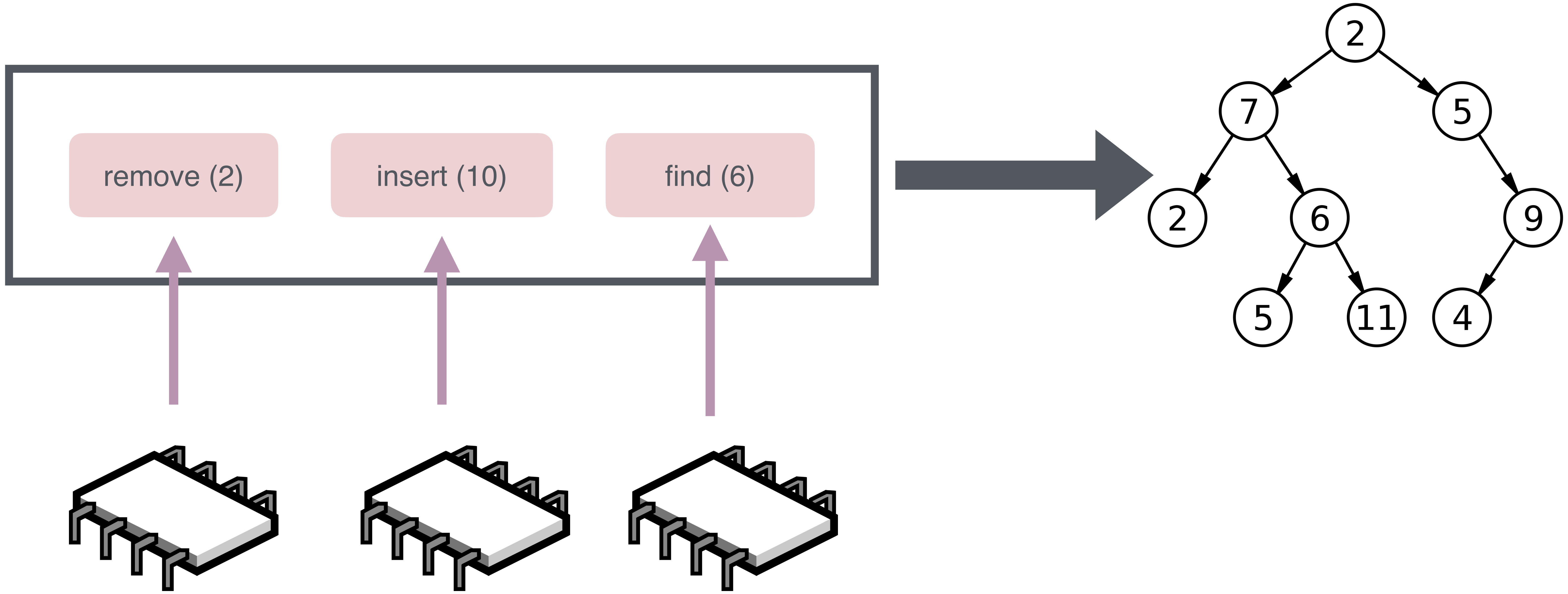


remove (2)

insert (10)

find (6)

Batch Parallelism



remove (2)

insert (10)

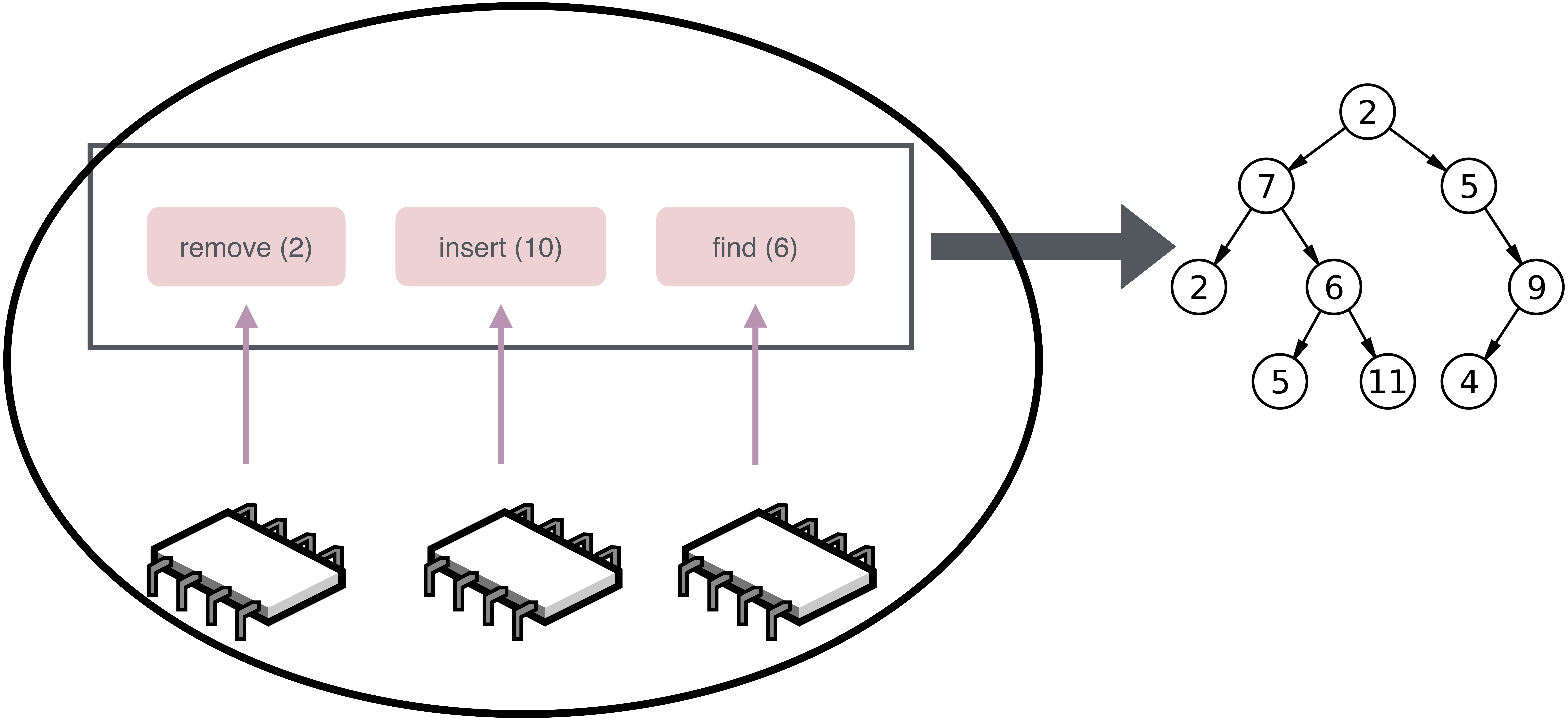
find (6)

Advantages:

- Less lock contention
- Parallel operations
- Simpler design

But… some problems

Explicit Batch Parallelism

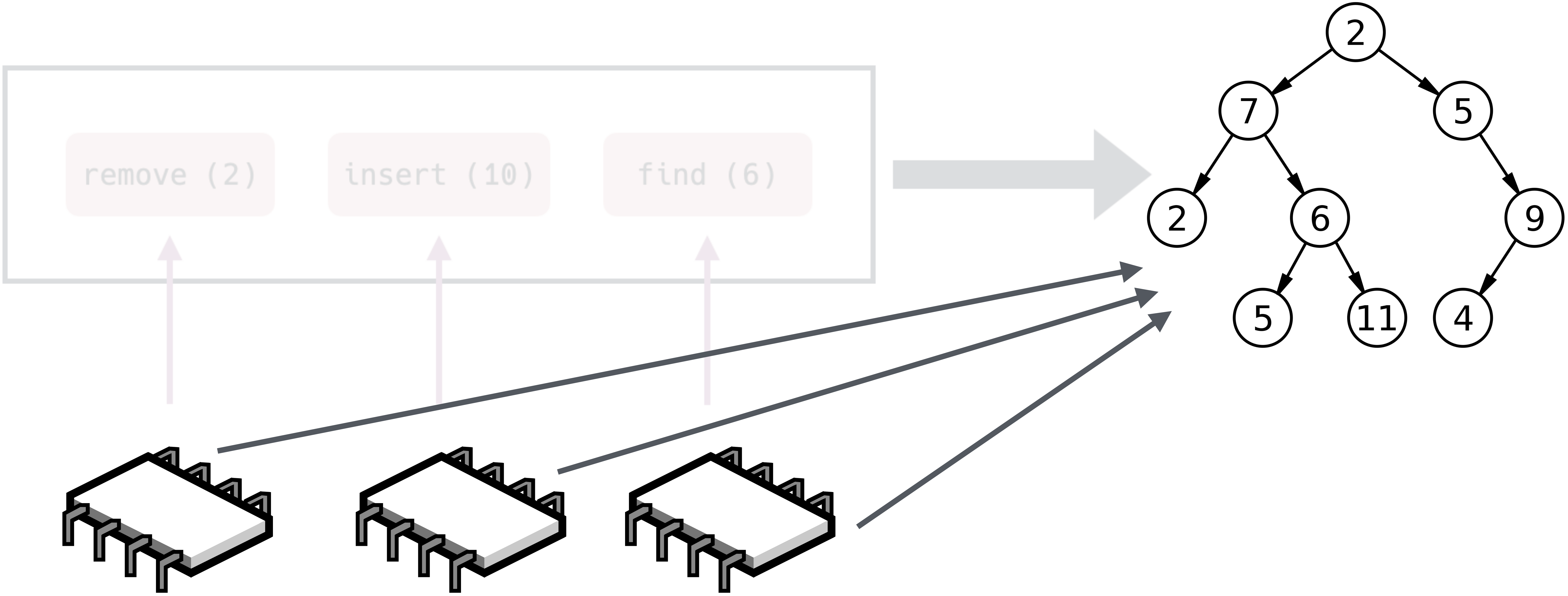


remove (2)

insert (10)

find (6)

Implicit Batch Parallelism



`remove (2)` `insert (10)` `find (6)`

This Work

1. **How to implement implicit batching**

2. **How to parallelise operations within a batch**

This Work

1. **How to implement implicit batching**

2. **How to parallelise operations within a batch**

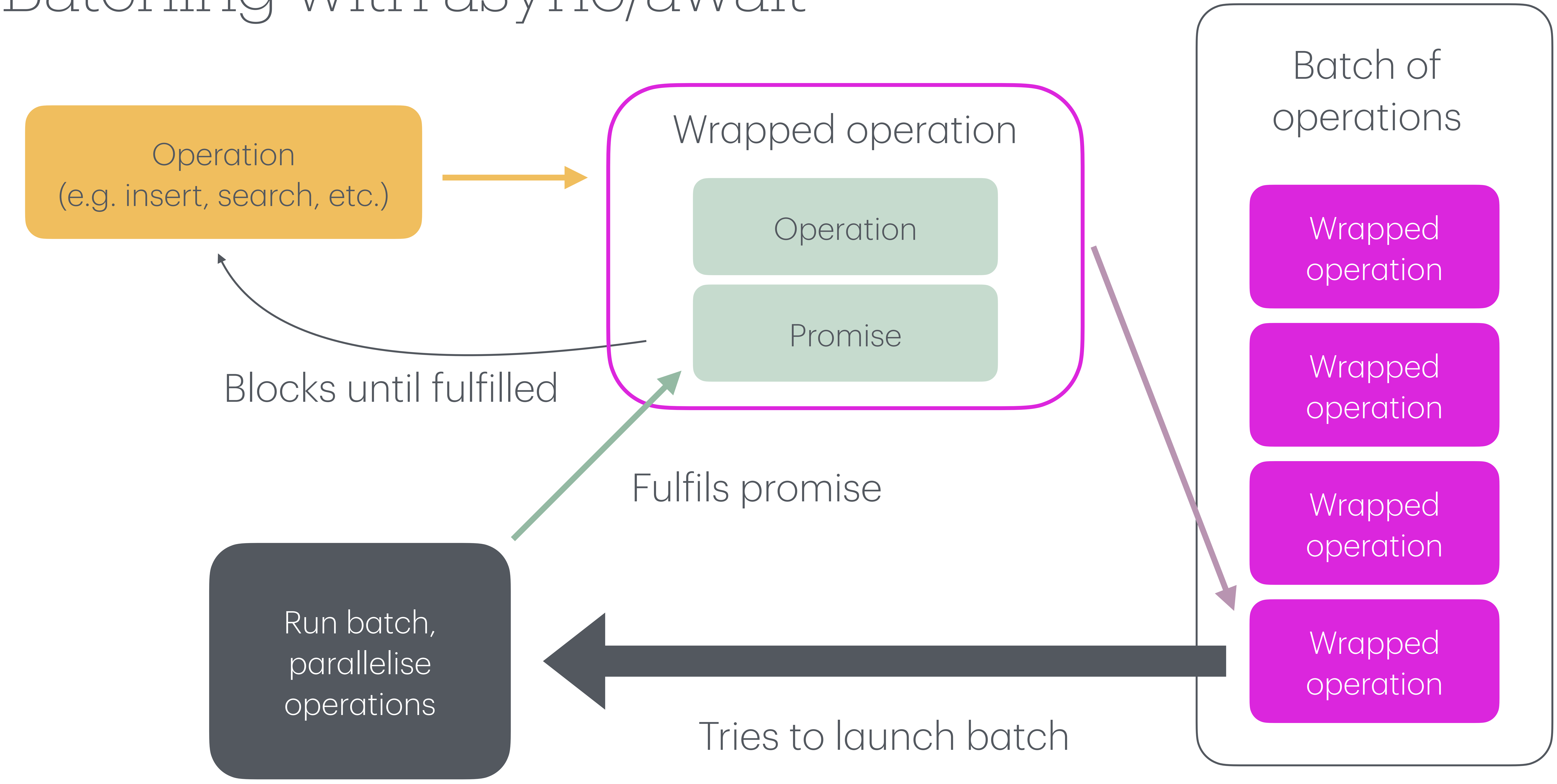
This Work

1. How to implement implicit batching

Key idea: You only need async/await for this

2. How to parallelise operations within a batch

Batching with async/await



Operation
(e.g. insert, search, etc.)

Wrapped operation

Operation

Promise

Blocks until fulfilled

Fulfils promise

Batch of
operations

Wrapped
operation

Wrapped
operation

Wrapped
operation

Wrapped
operation

Run batch,
parallelise
operations

Tries to launch batch

This Work

1. How to implement implicit batching ✅

Key idea: You only need async/await for this

2. How to parallelise operations within a batch

Outline

1. How to implement implicit batching ✅

Key idea: You only need async/await for this

2. How to parallelise operations within a batch

Outline

1. **How to implement implicit batching** ✅

   Key idea: You only need async/await for this

2. **How to parallelise operations within a batch**

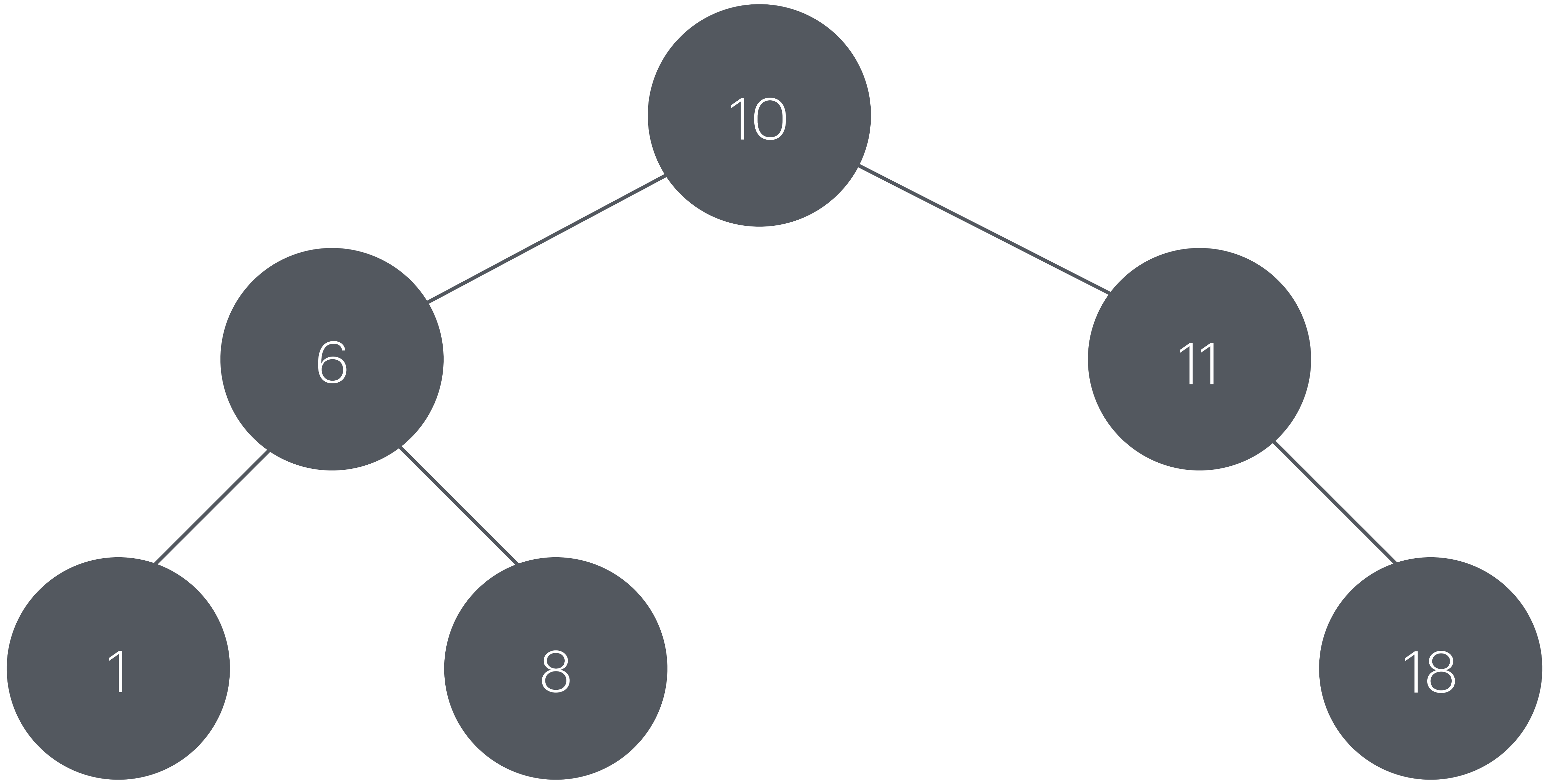
   Key idea: Sequential strategies for batch-parallelism

A Sequential Strategy: Split-Join

1. Split data structure into independent (sub-)data structures.
2. Modify each split data structure in parallel.
3. Rejoin modified data structures together.

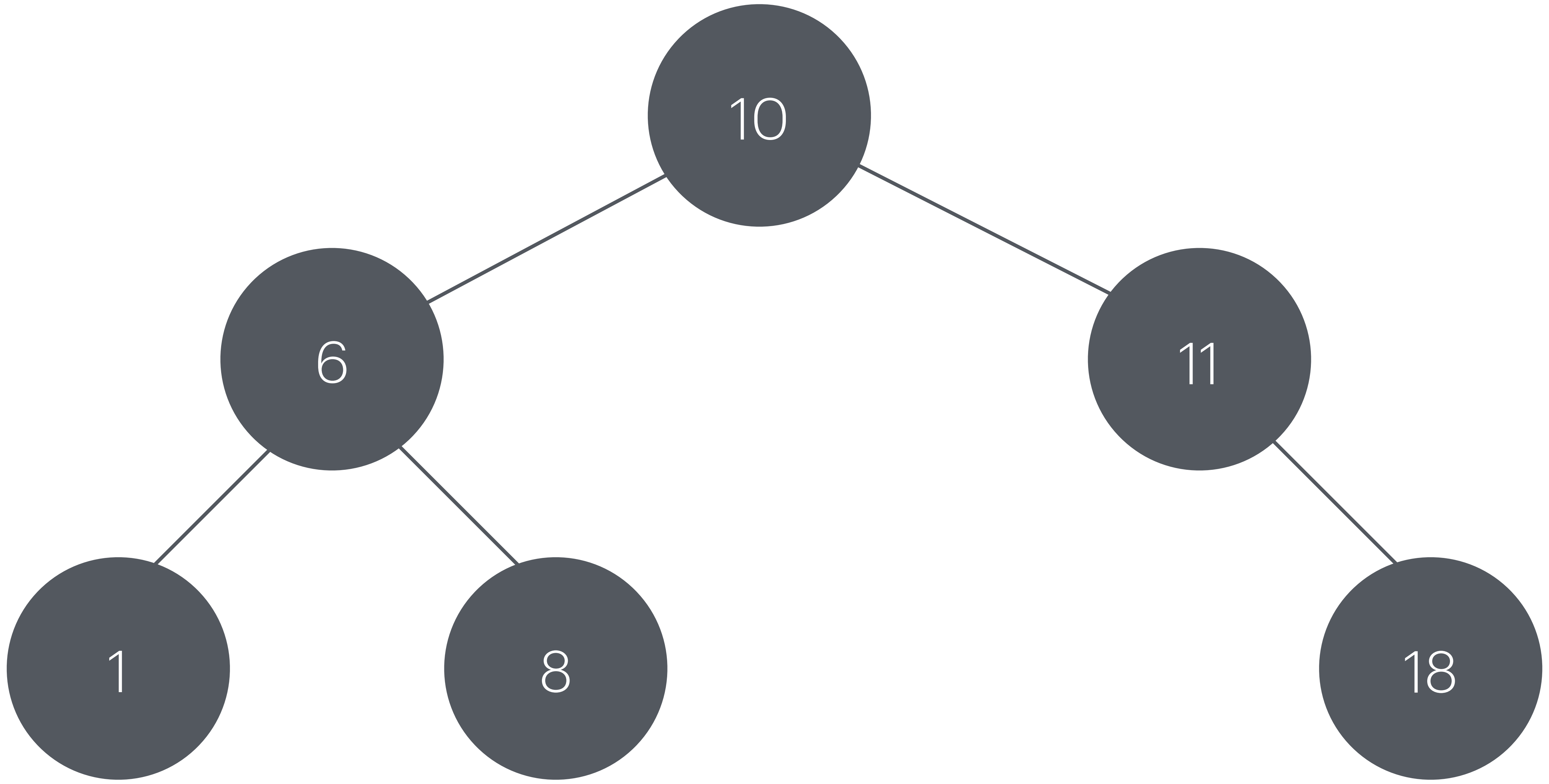
Let’s look at an example!

Meet the Red-Black Tree



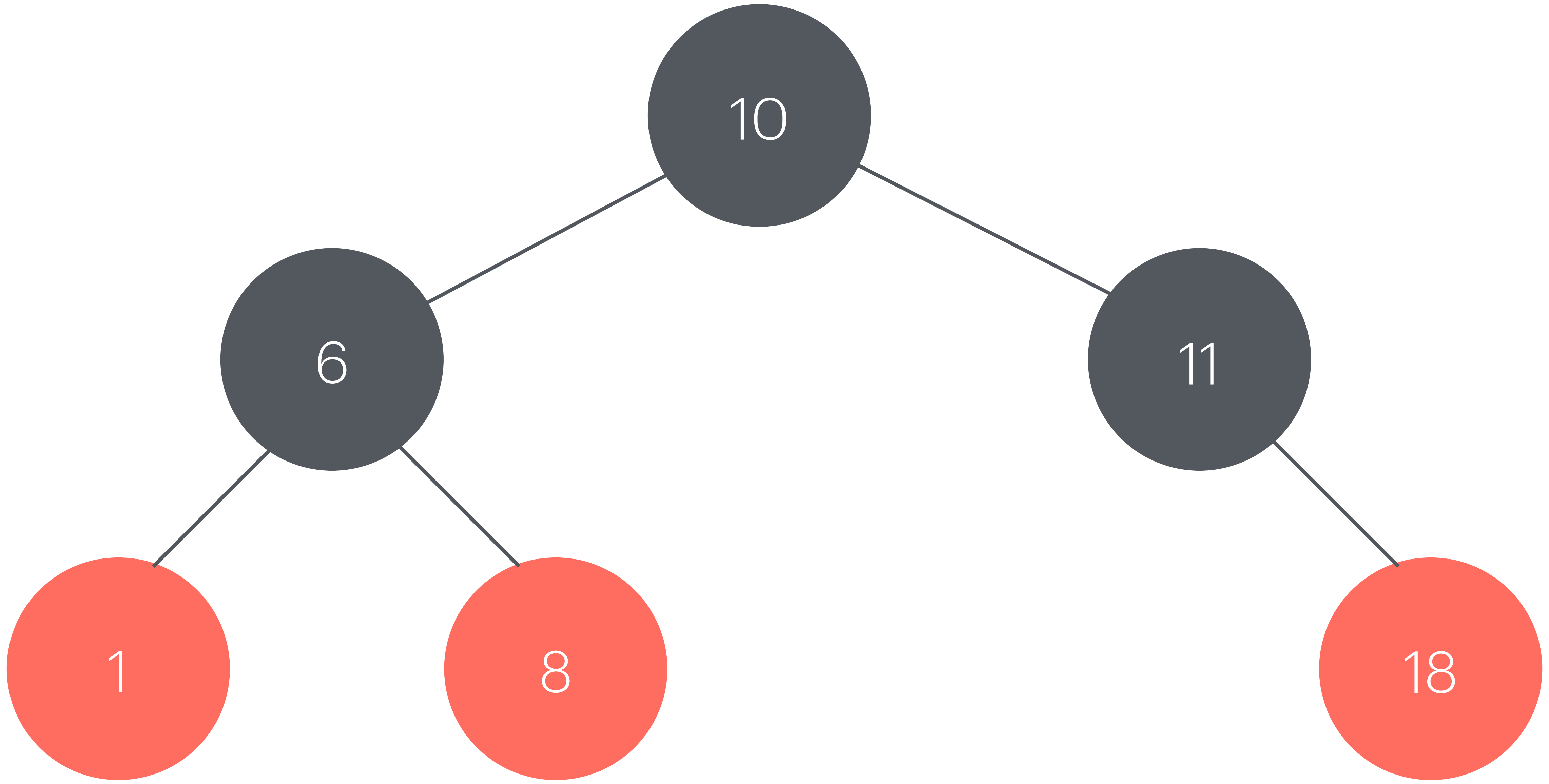
Meet the Red-Black Tree

- An approximately balanced binary tree.



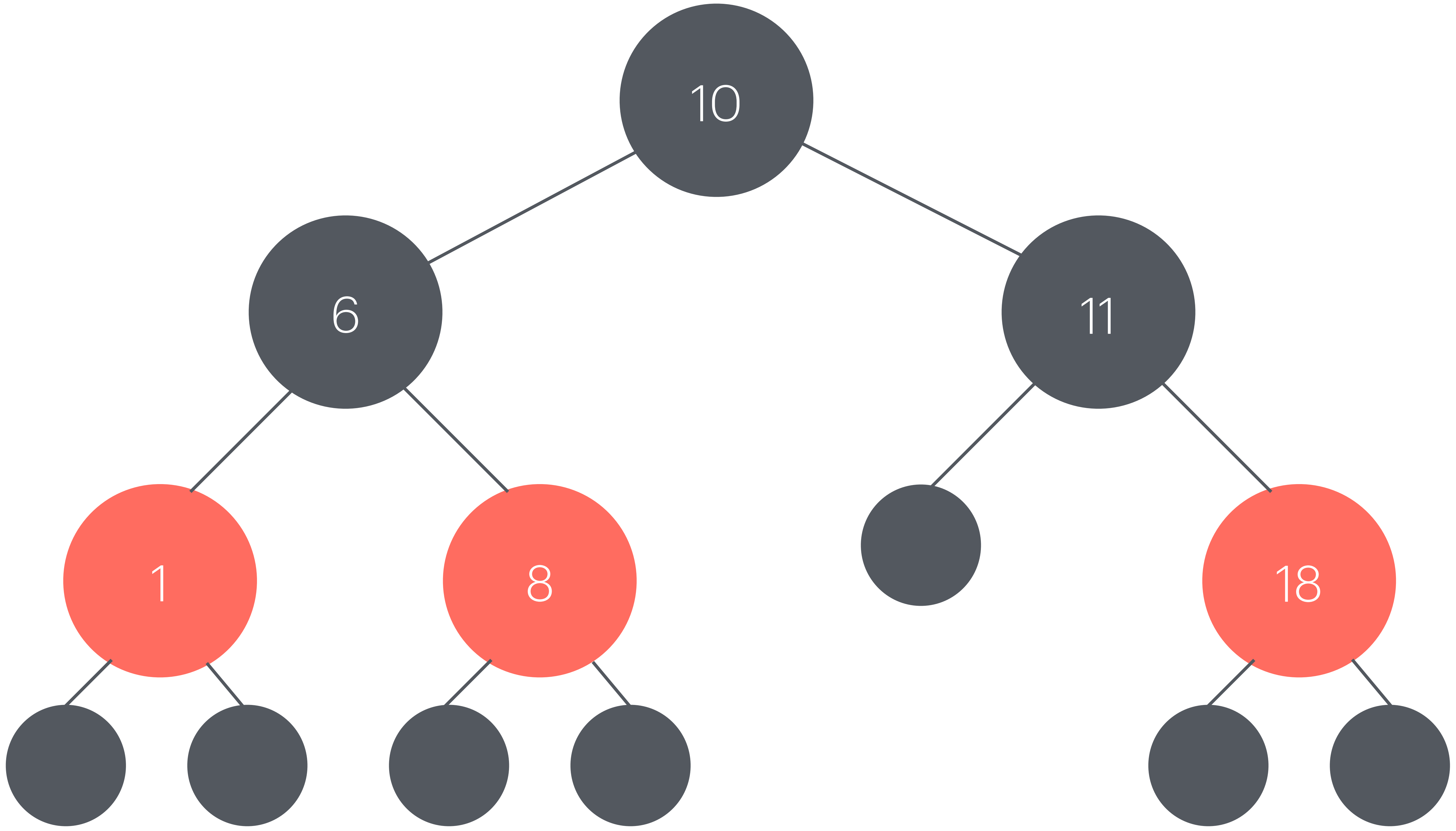
Meet the Red-Black Tree

- An approximately balanced binary tree.
- Each node is either red or black.



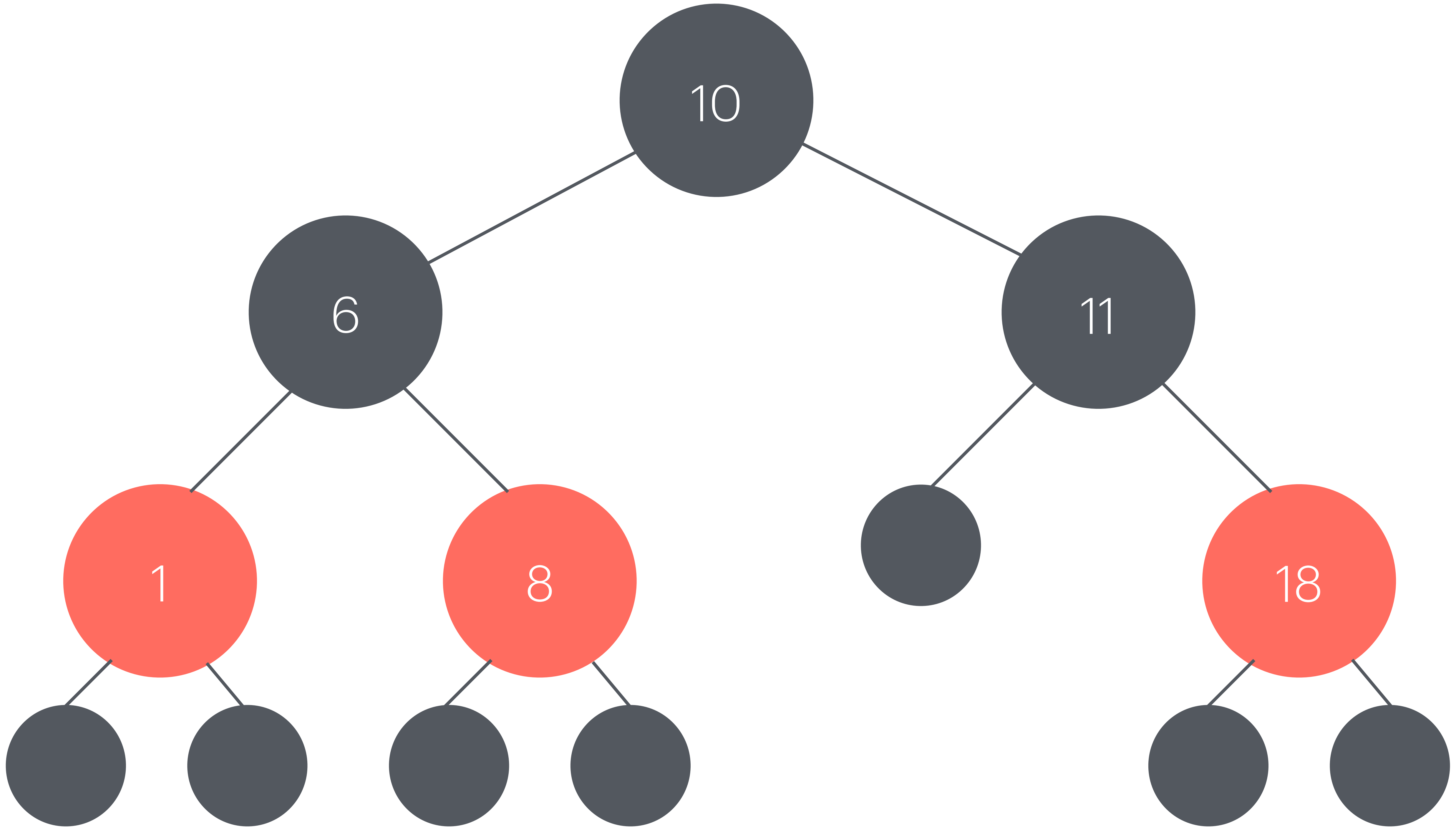
Meet the Red-Black Tree

- An approximately balanced binary tree.
- Each node is either red or black.
- Has empty leaves at the bottom layer.



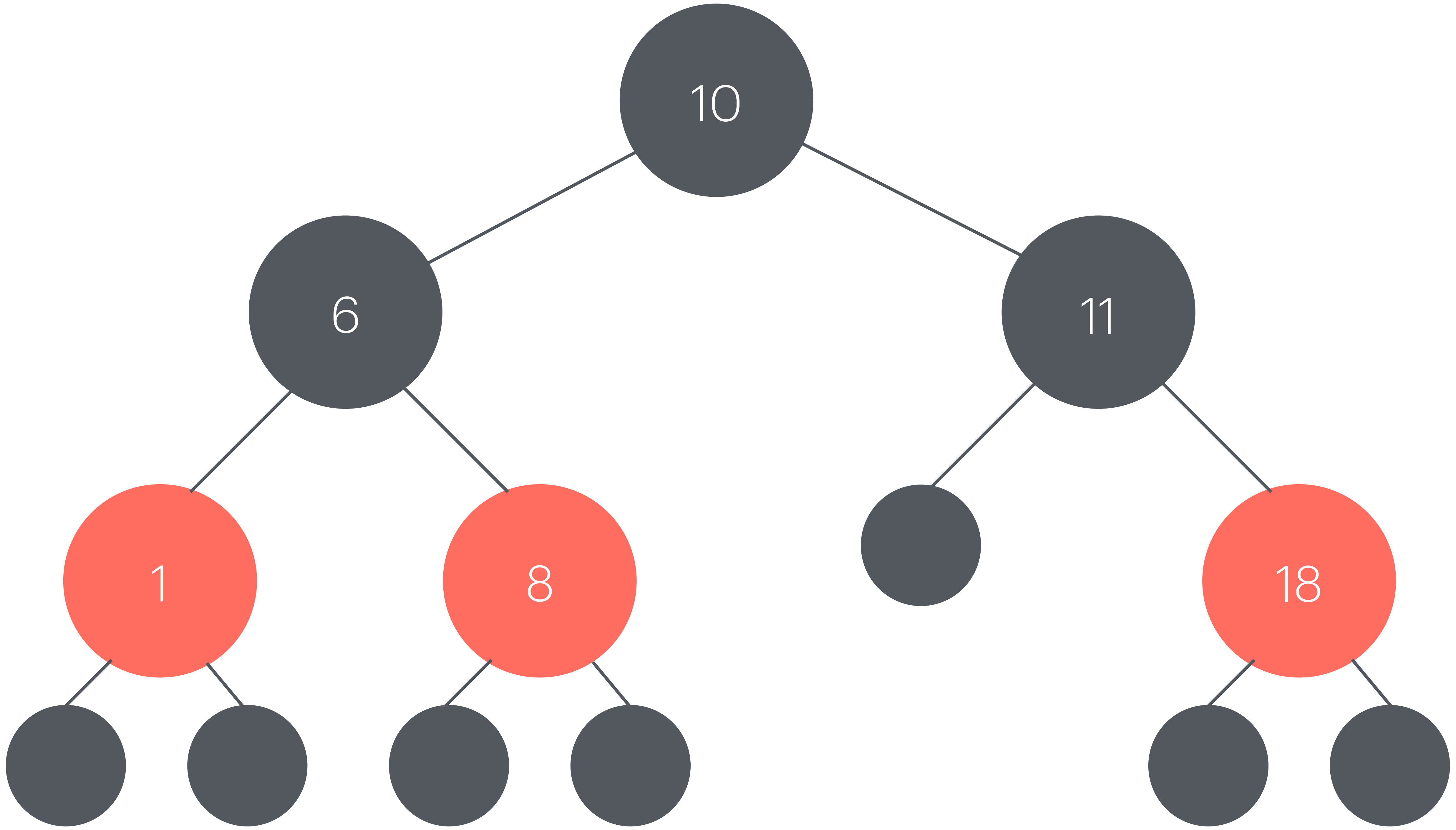
Meet the Red-Black Tree

- An approximately balanced binary tree.
- Each node is either red or black.
- Has empty leaves at the bottom layer.
- Supports search, insert, delete operations in *O(log n)* time complexity.



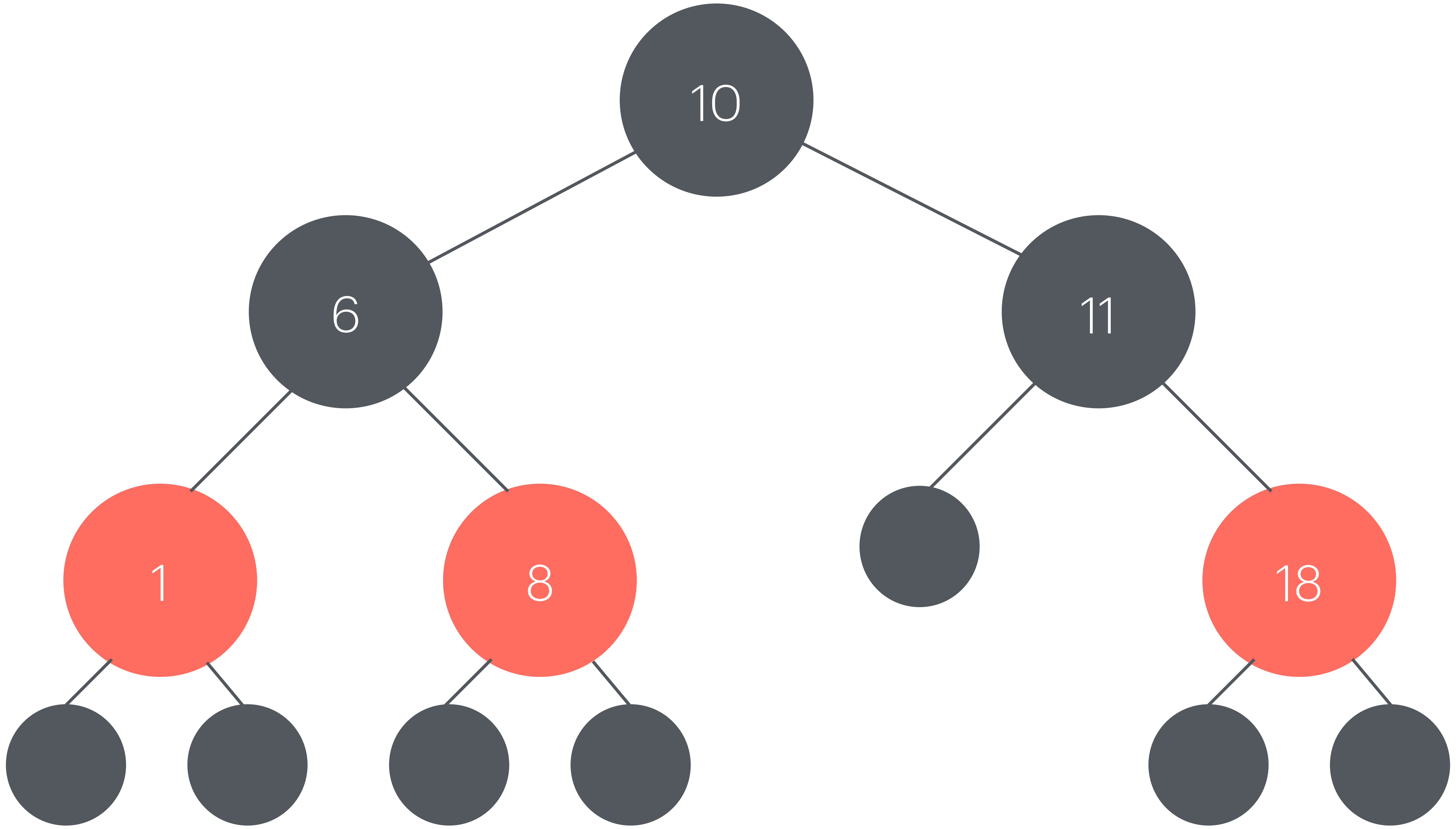
Meet the Red-Black Tree

- Invariants:
  - Every leaf is black.
  - If a node is red, both its children must be black.
  - Paths from a given node to any of its descendant leaves must have the same number of black nodes.
- **Must rebalance after each update.**



A Batch-Parallel Red-Black Tree

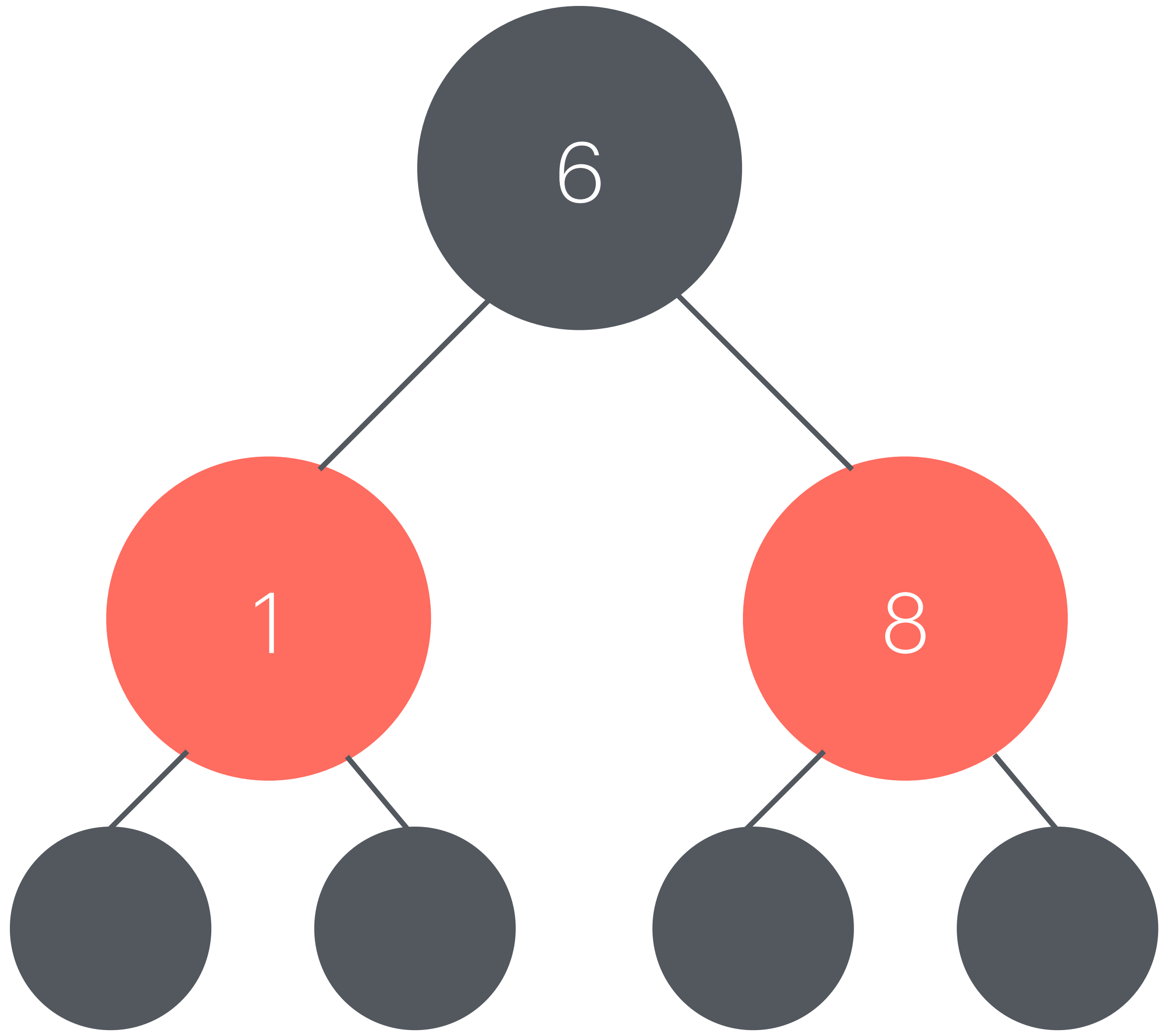
2	3	4	5	7	9	12	13	15	20	22	24

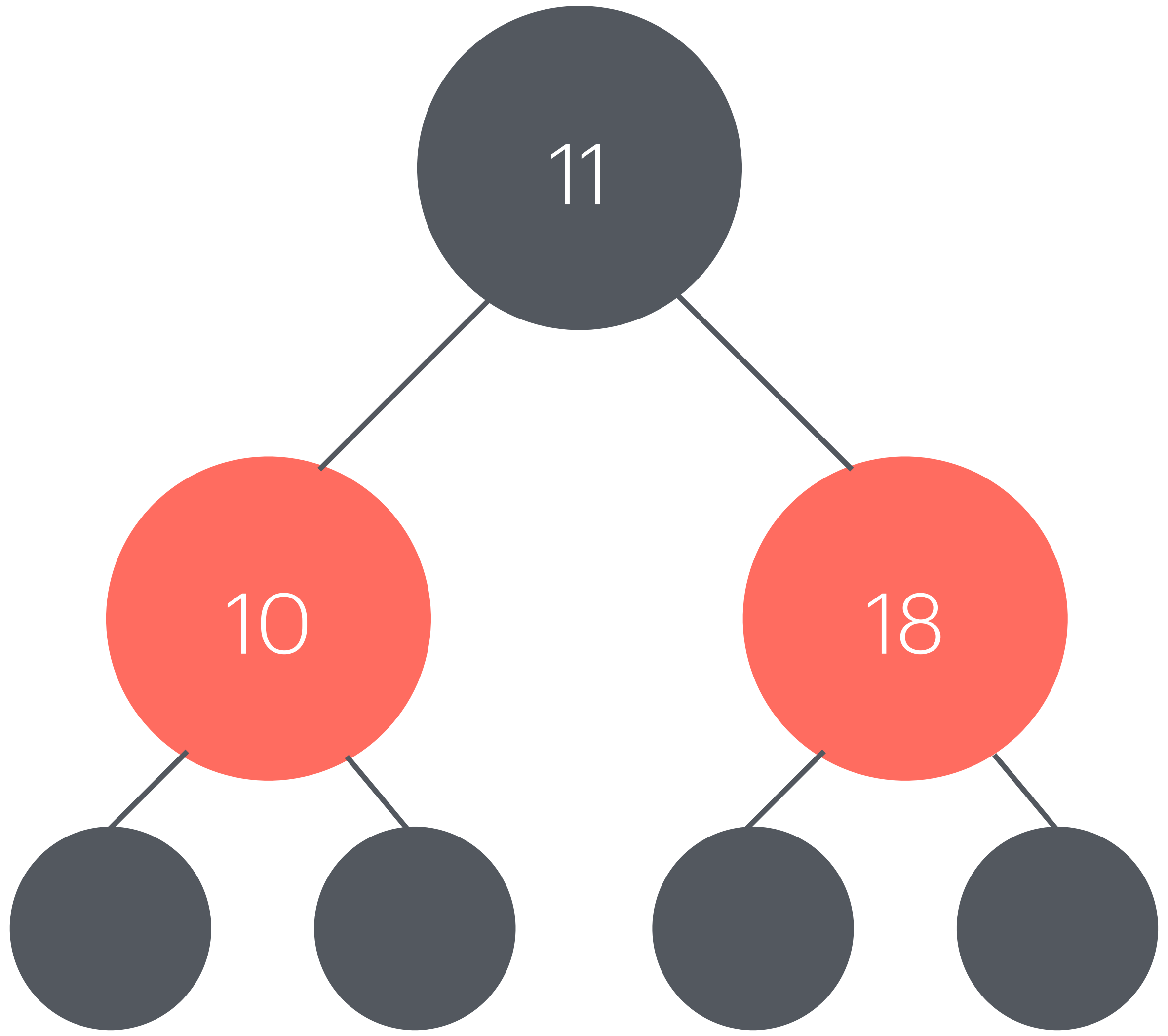


A Batch-Parallel Red-Black Tree

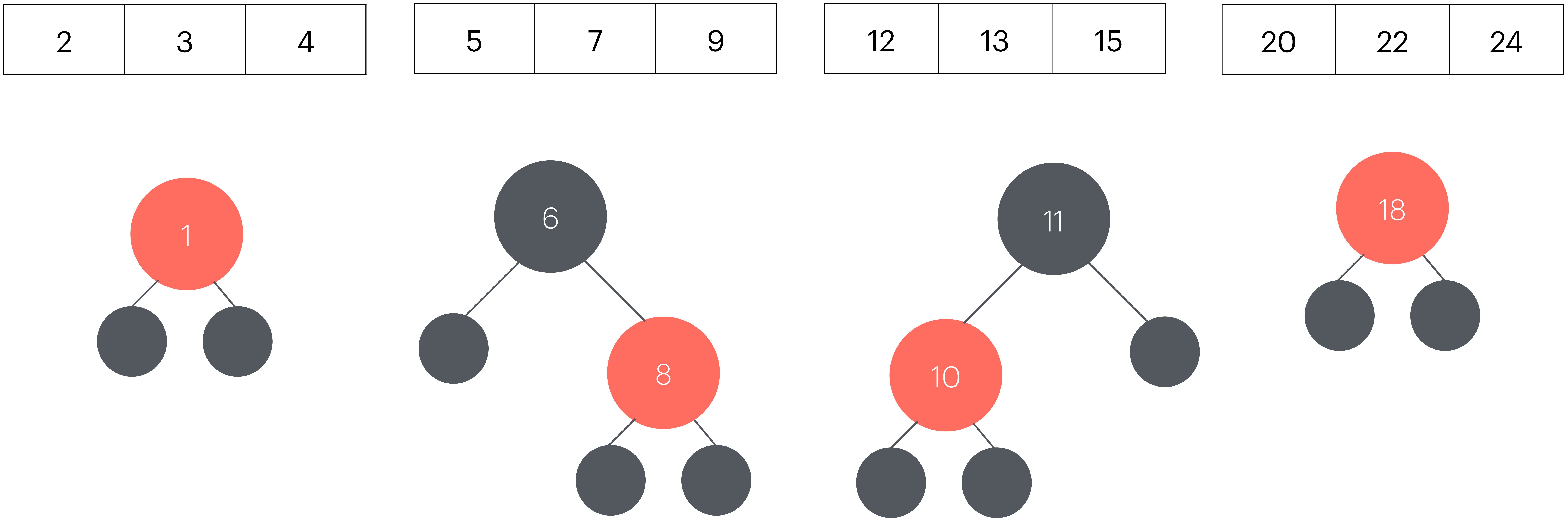
2	3	4	5	7	9

12	13	15	20	22	24





A Batch-Parallel Red-Black Tree



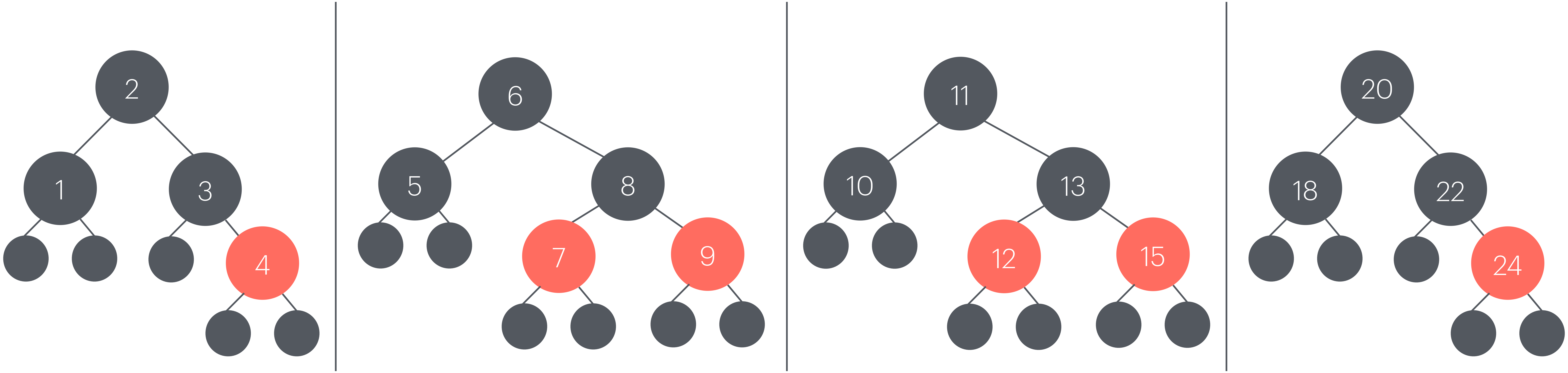
2	3	4

5	7	9

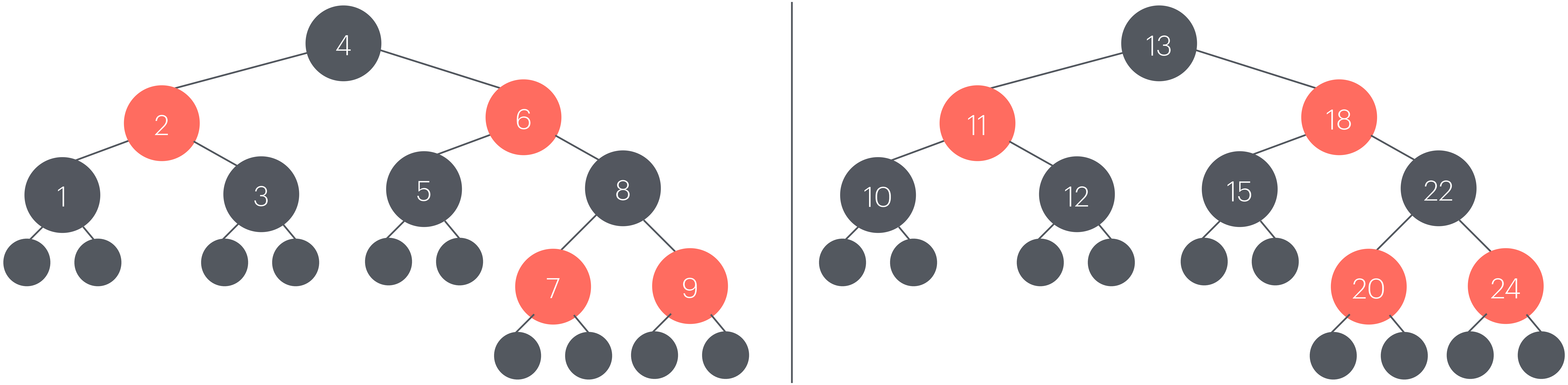
12	13	15

20	22	24

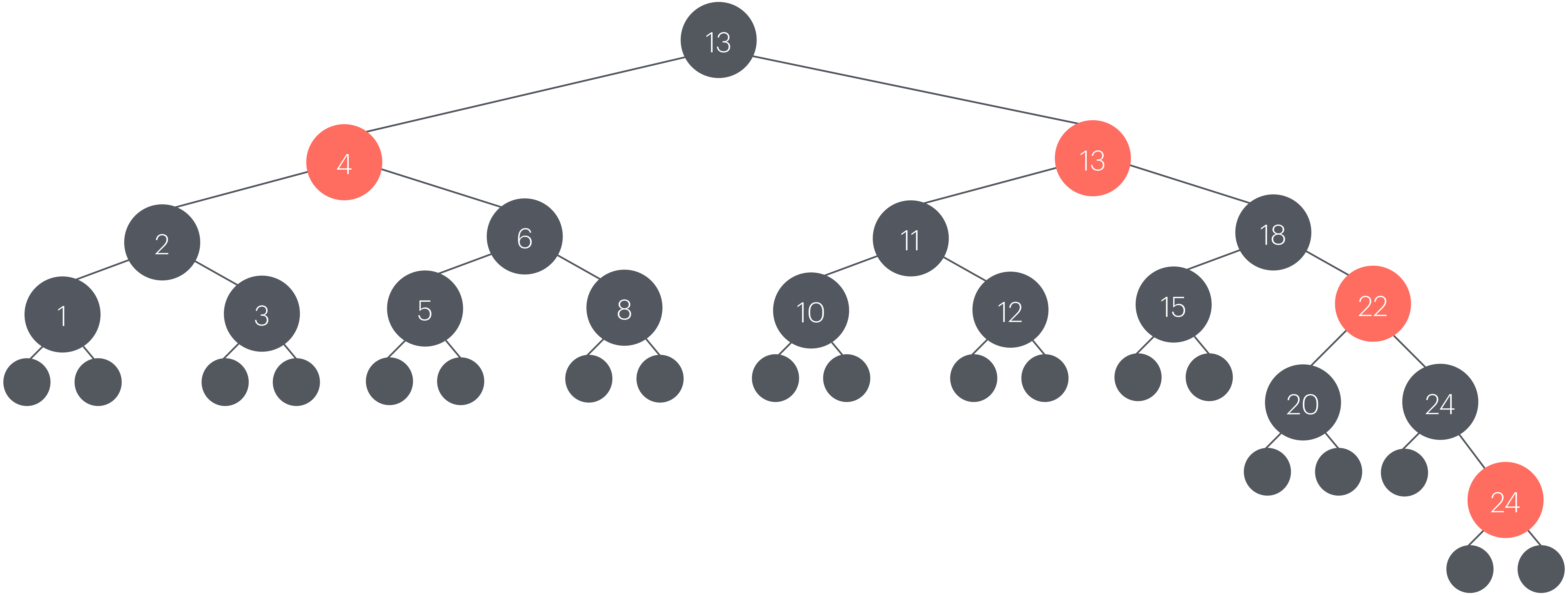
A Batch-Parallel Red-Black Tree



A Batch-Parallel Red-Black Tree



A Batch-Parallel Red-Black Tree



Batch Parallelisation Strategy: Split-Join

To sum up:

- Just implement **split** and **join**.
- No concurrent programming!
- Can be made generic, e.g. via functors (OCaml) or traits (Rust)

Other Data Structures and Strategies

Split-Join

- AVL Tree
- Red-Black Tree
- Treap

Expose-Repair

- van Emde Boas Tree
- X-Fast Trie
- Y-Fast Trie

Ad-Hoc

- Skiplist
- B-tree
- Datalog

Our Work

1. How to implement implicit batching ✅

Key idea: You only need async/await

2. How to parallelise operations within a batch ✅

Key idea: Sequential strategies for batch-parallelism

Implementation Details

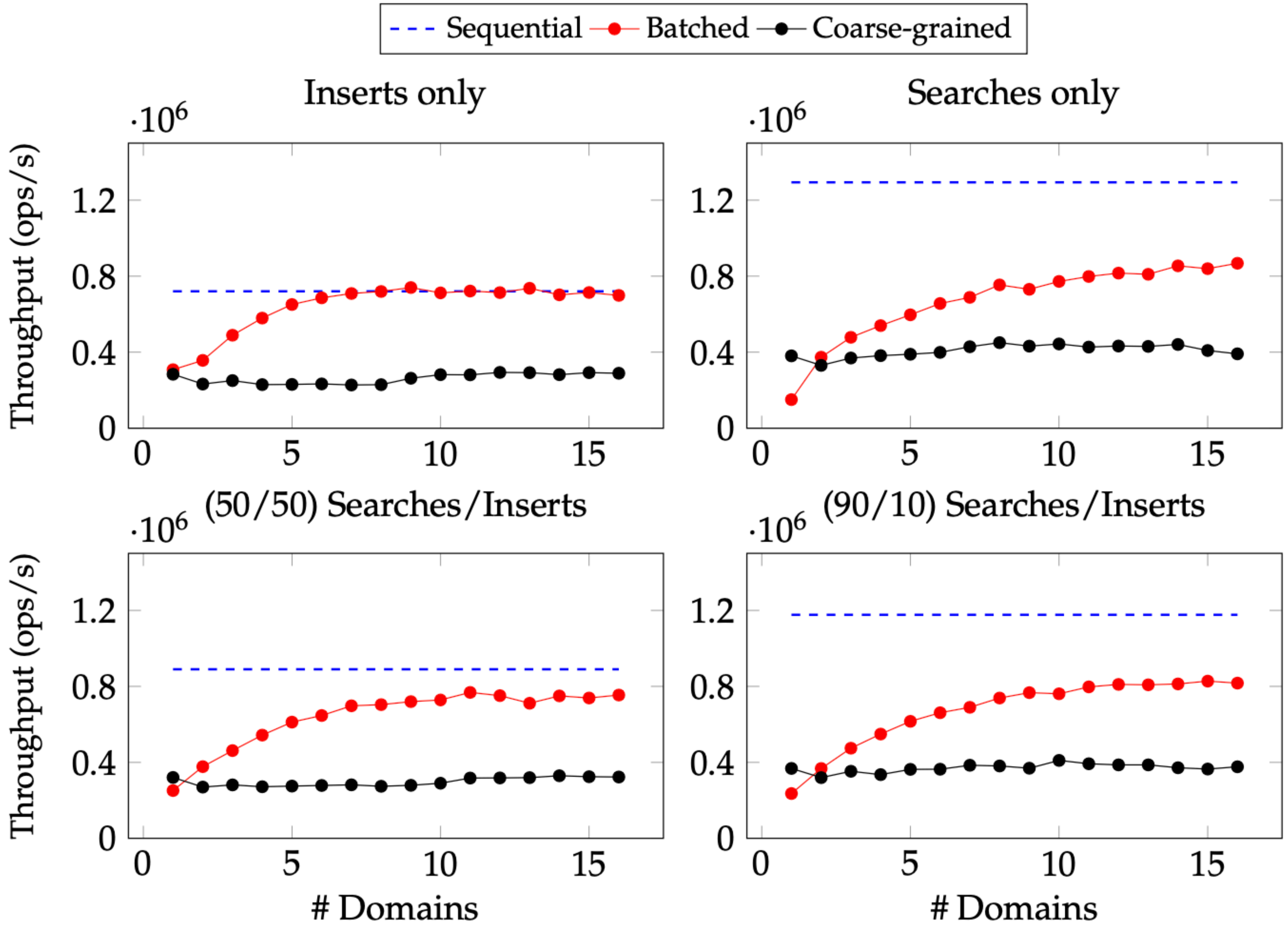
- Implemented in OCaml 5:
  - ~230 LOC for implicit batching.
  - ~150 to ~200 LOC for generic part of each batching pattern.
  - Using async/await from Domainslib library.
- Implemented in Rust:
  - Approx. 150 LOC for implicit batching

Performance Evaluation

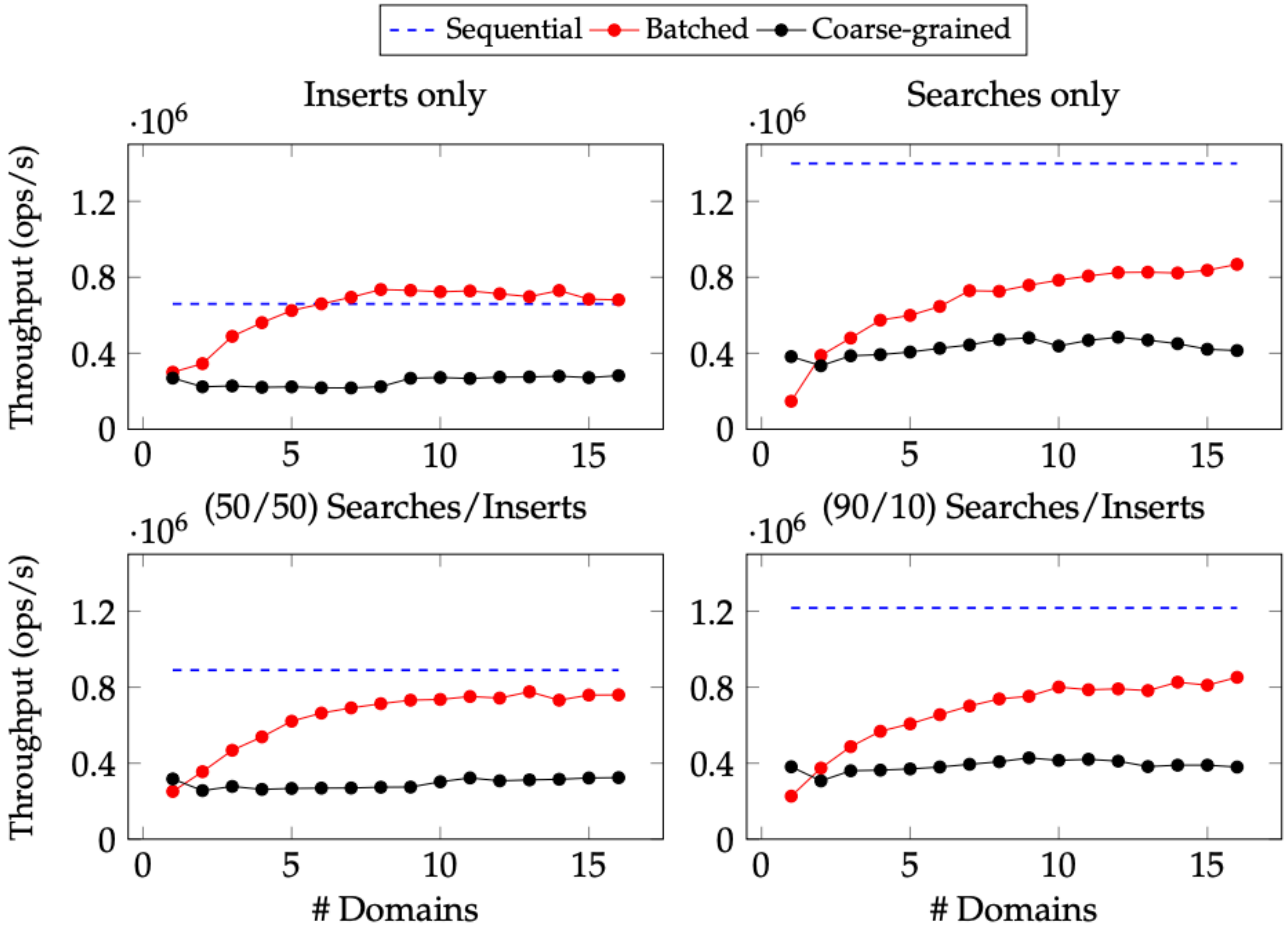
Test setup:

- OCaml implementation.
- Setup: 2M initial elements, 1M benchmarked operations.
- One operation = one concurrent task.
- Machine: AWS EC2, Intel Core Xeon Processor, 24 cores, 96 GB of RAM, Ubuntu 22.02.

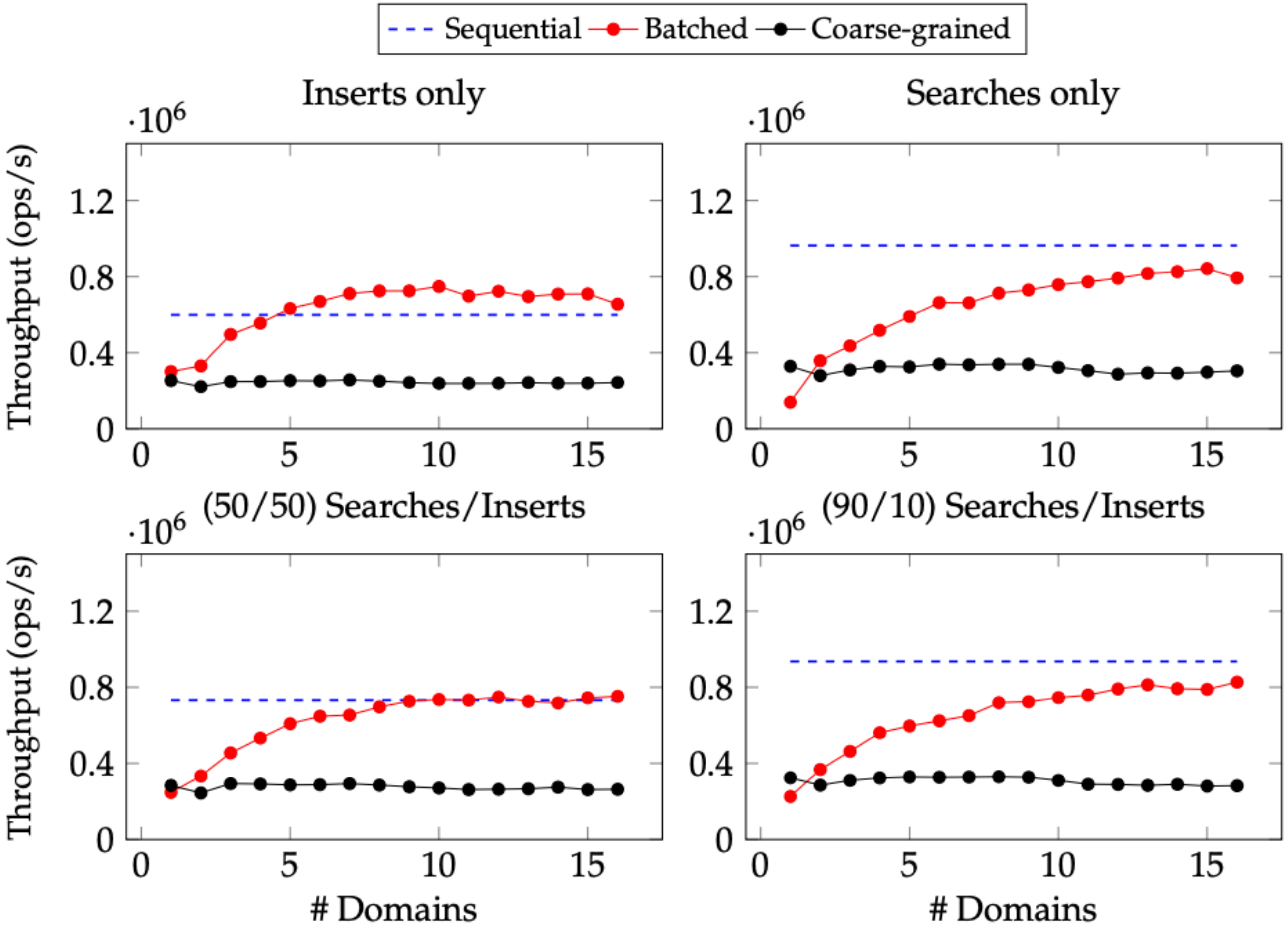
Performance Evaluation: Red-Black Tree



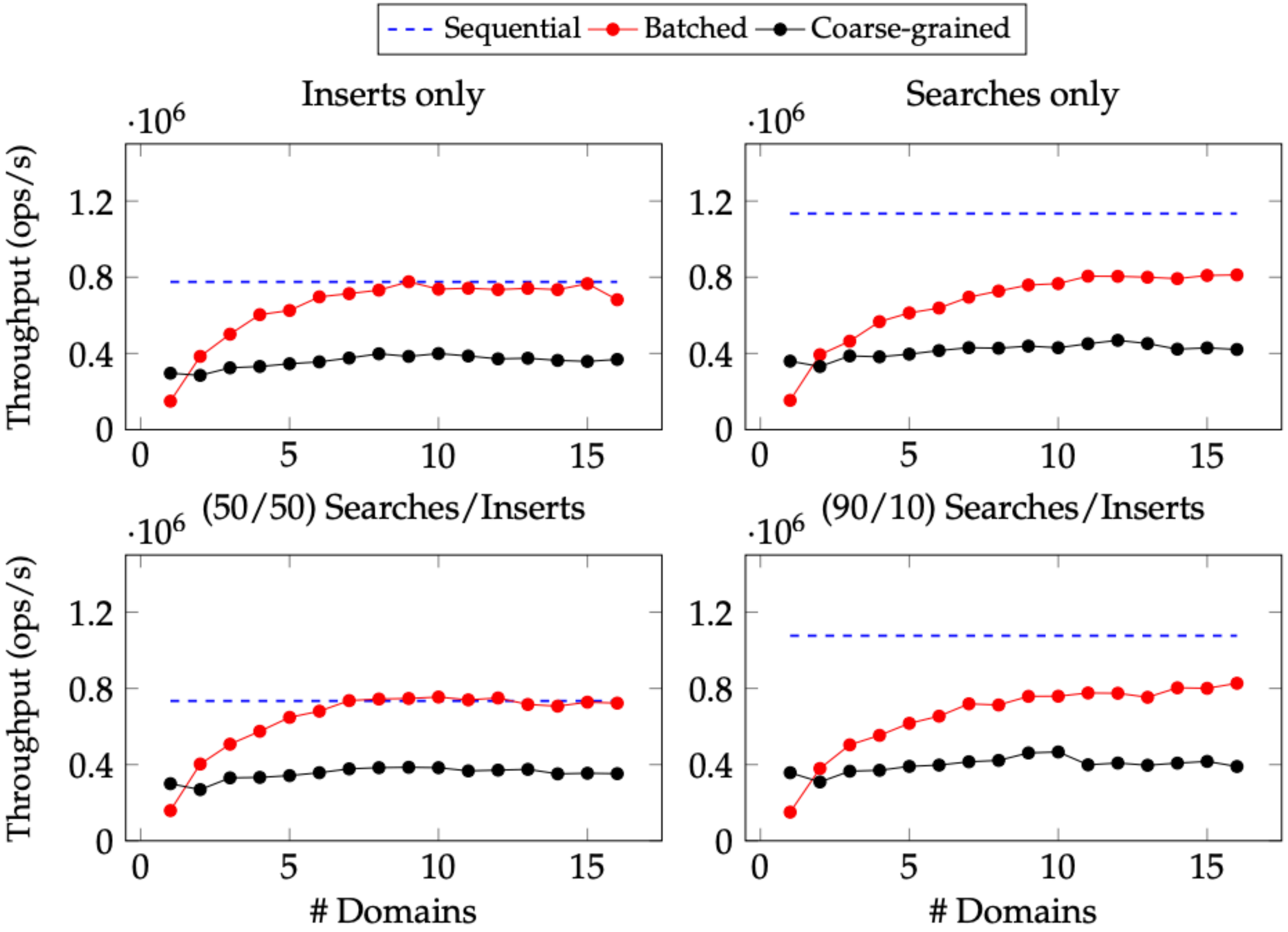
Performance Evaluation: AVL Tree



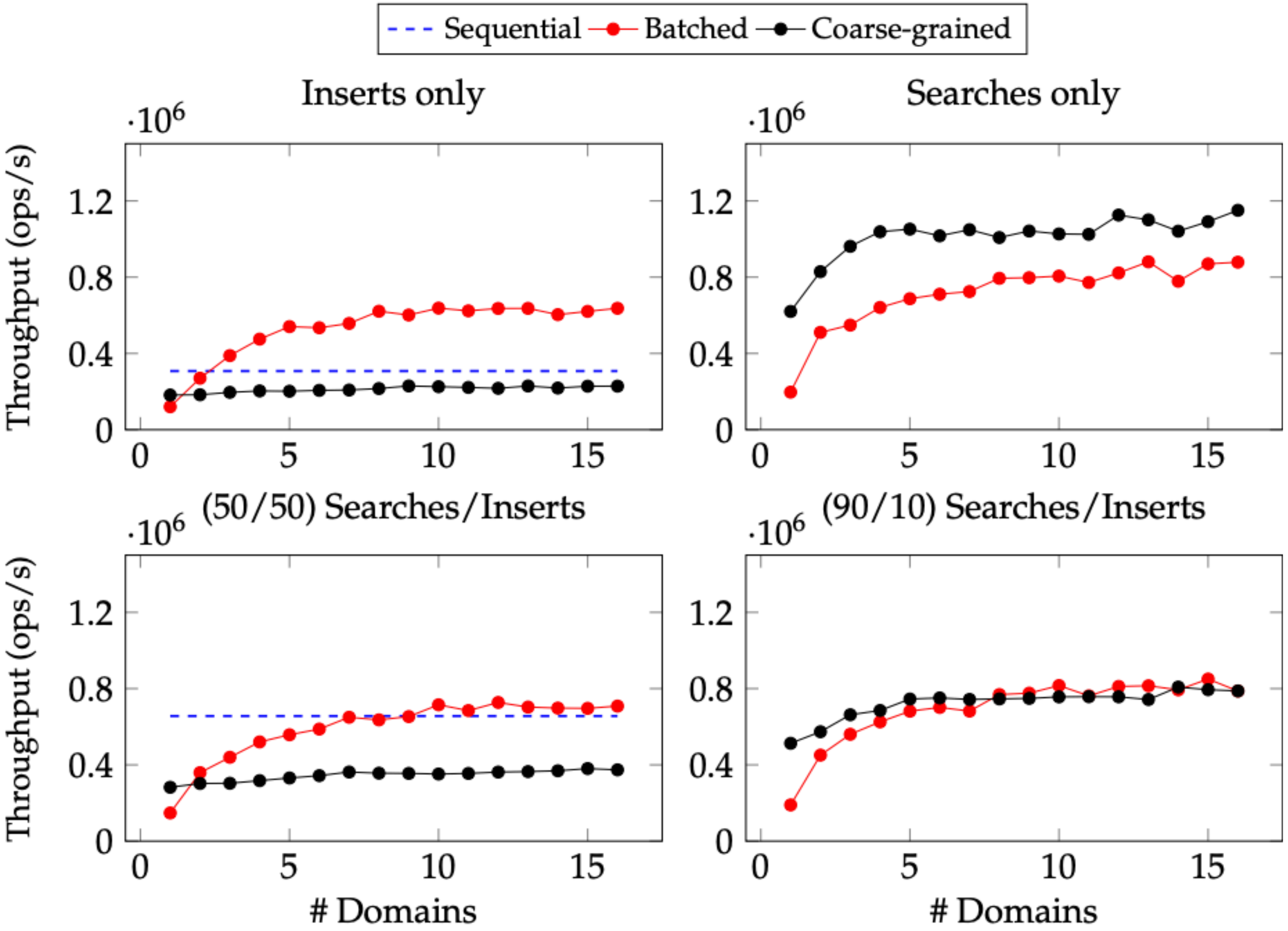
Performance Evaluation: Treap



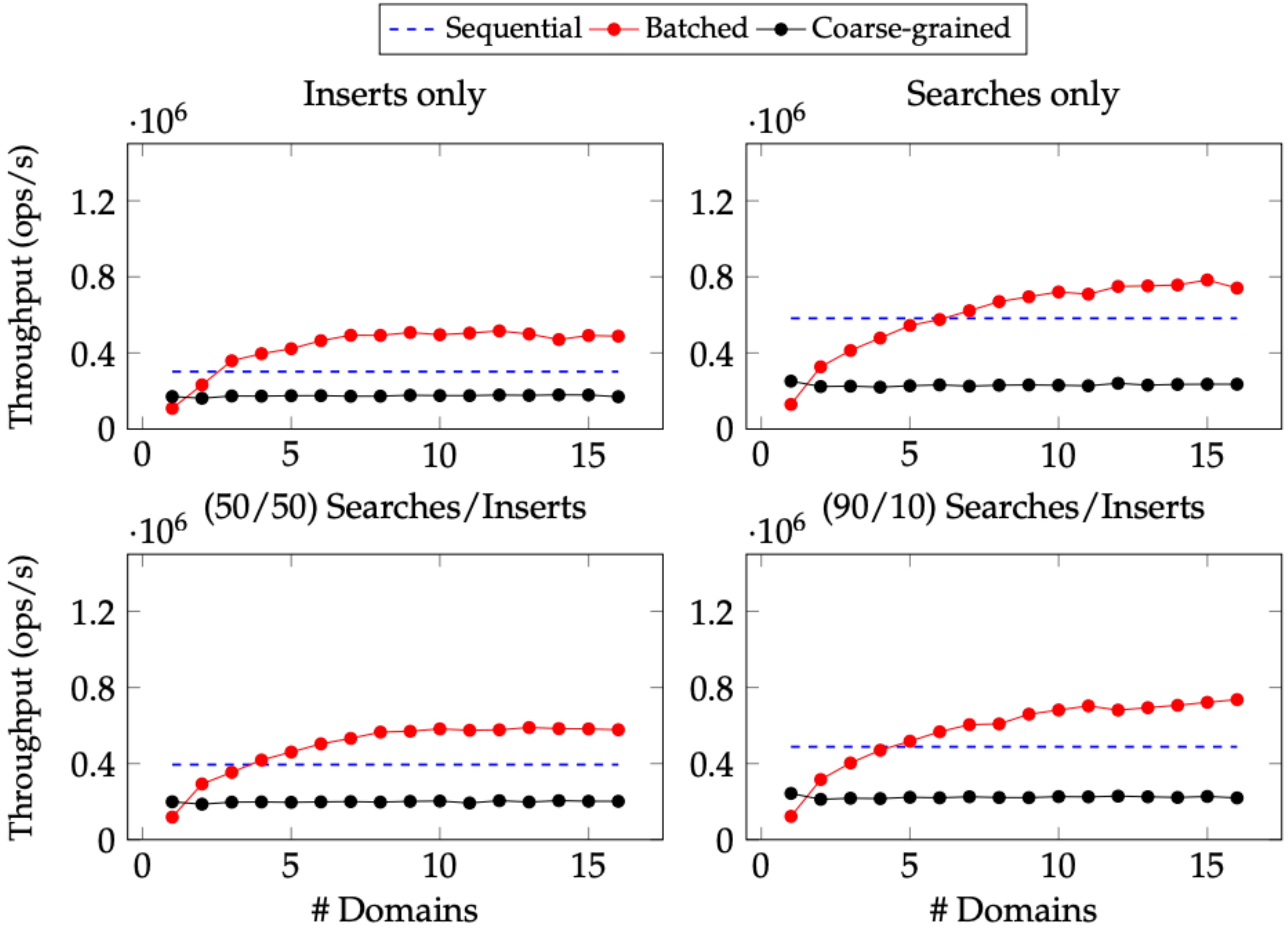
Performance Evaluation: van Emde Boas Tree



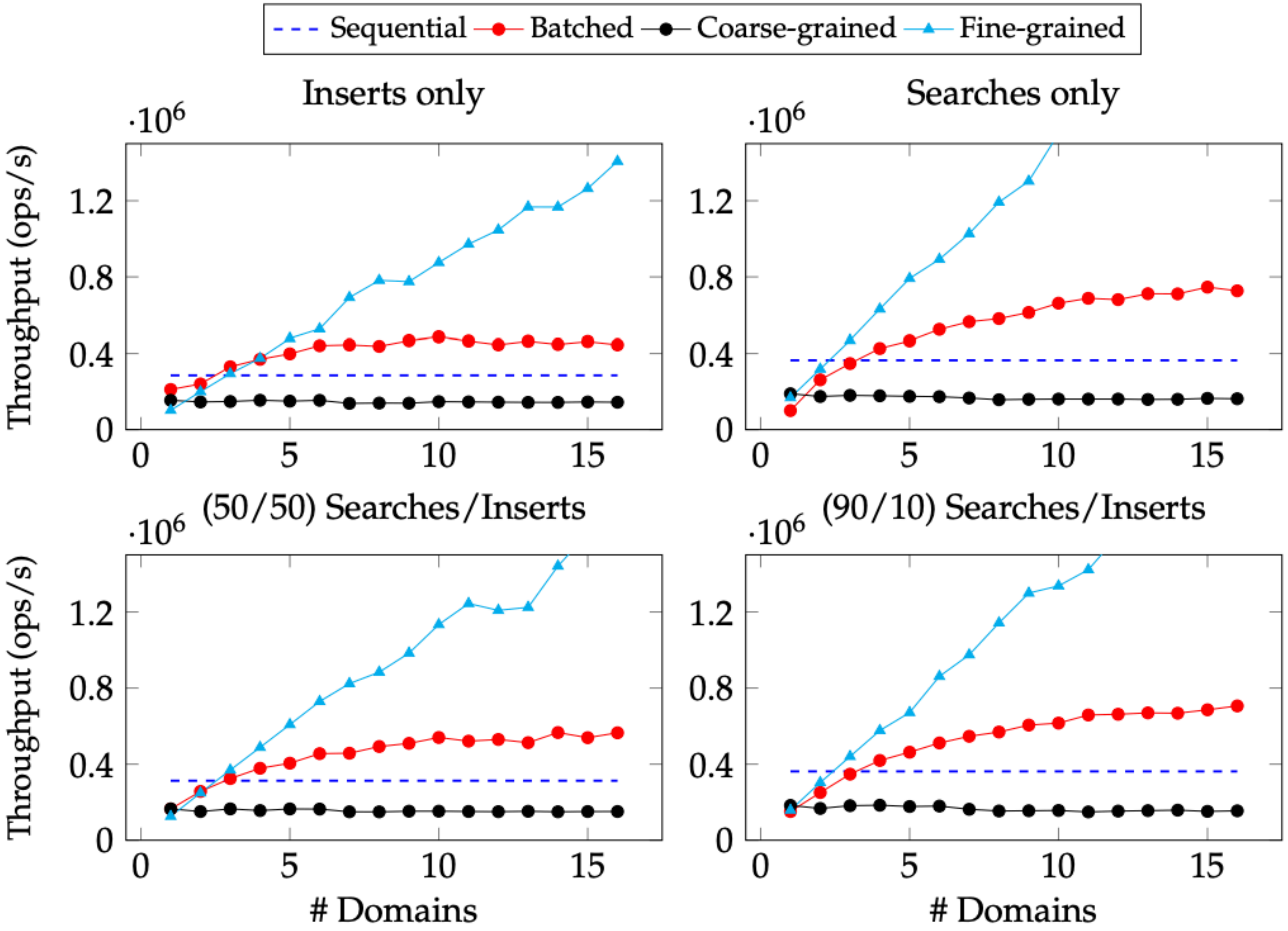
Performance Evaluation: X-Fast Trie



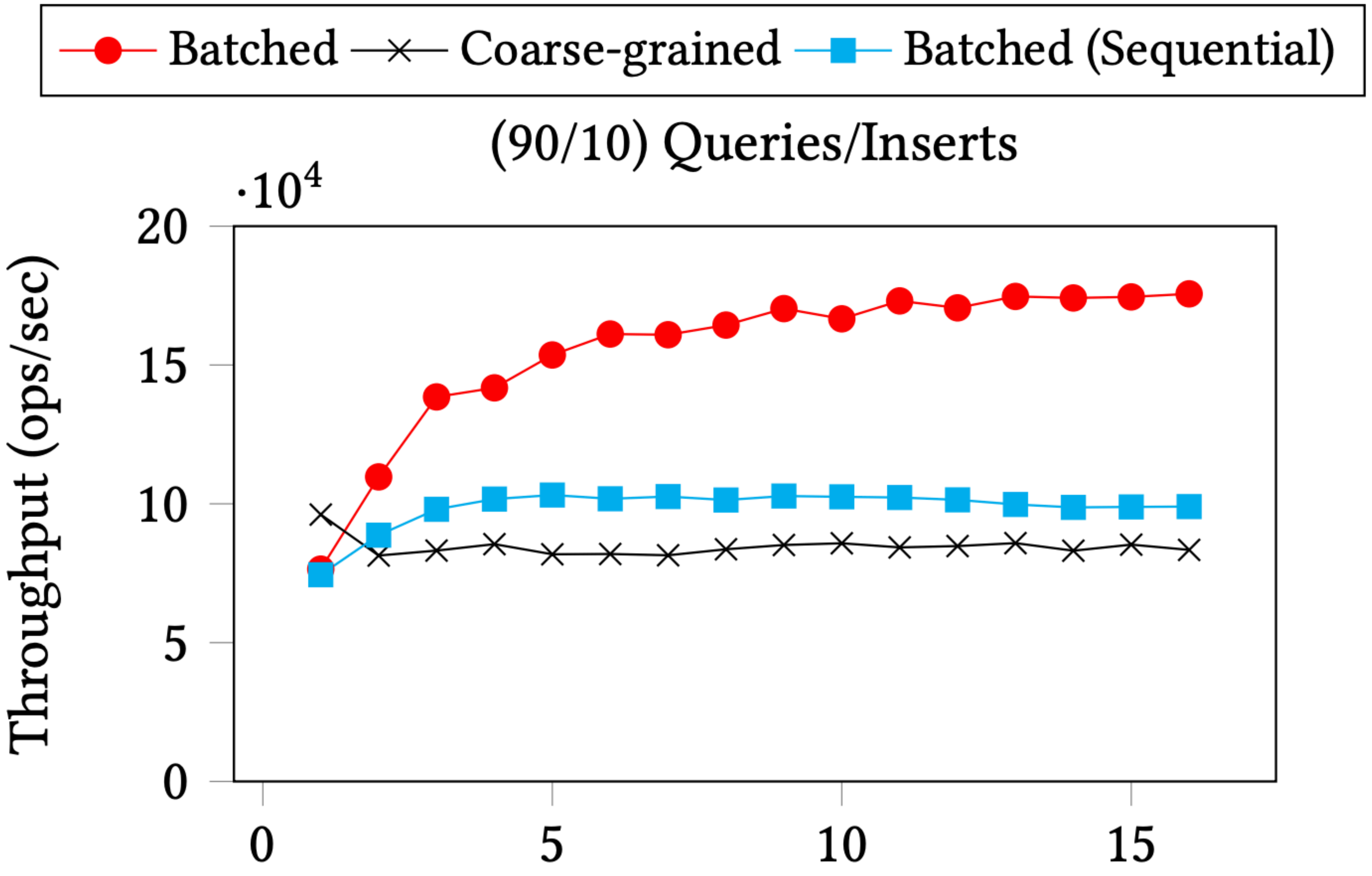
Performance Evaluation: Y-Fast Trie



Performance Evaluation: Skiplist



Performance Evaluation: Datalog Solver



To Take Away

- **Batching** — easy way to implement parallel processing
- **Implicit** batching can be implemented using **async/await**
- **Split/Join** and other strategies: concurrent data structures **without** concurrent code!

OCaml library

The paper