Functional Distributed Programming with Irmin

QCon NYC 2015, New York

Anil Madhavapeddy (speaker)
with Benjamin Farinier, Thomas Gazagnaire, Thomas Leonard
University of Cambridge Computer Laboratory

June 12, 2015

- Background
  - Git in the datacenter
  - Irmin, a large-scale, immutable, branch-consistent storage
- Weakly consistent data structures
  - Mergeable queues
  - Mergeable ropes
- Benchmarking Irmin
- Use Cases

Common features every distributed system needs

- **Persistence** for fault tolerance and scaling
- **Scheduling** of communication between nodes
- **Tracing** across nodes for debugging and profiling

Most distributed systems run over an operating system, and so are stuck with the OS kernel exerting control. We use *unikernels*, which are application VMs that have complete control over their resources.

What if we just used Git?

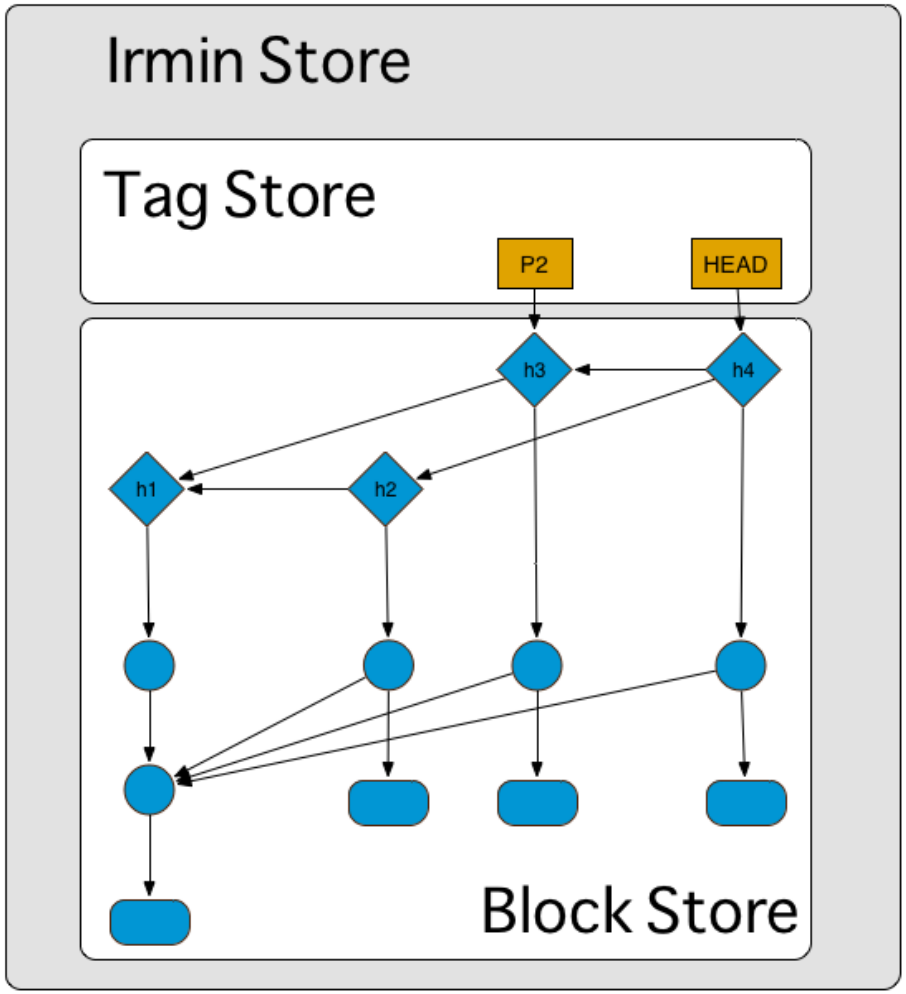
- **Persistence**
  - `git clone` of a shared repository across nodes
  - `git commit` of local operations in the node
- **Scheduling**
  - `git pull` to receive events from other nodes
  - `git push` to publish events to other nodes
- **Tracing and Debugging**
  - `git log` to see global operations
  - `git checkout` to roll back time to a snapshot
  - `git bisect` to locate problem messages

Problems with using Git?

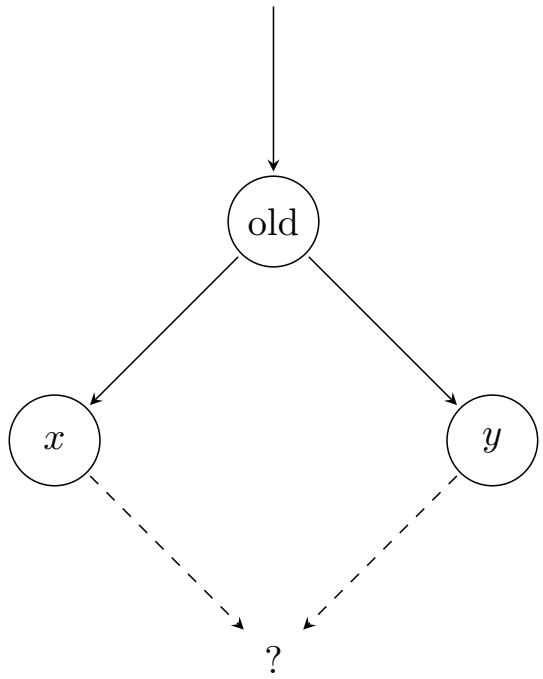
- **Garbage Collection**
  - Git records all operations permanently, so our database will grow permanently!
  - `git rebase` is needed to compact history.
- **Shell Control**
  - Calling the `git` command-line is slow and lacks fine control.
  - Makes it hard to extend the Git protocol for additional features.
- **Programming Model**
  - Git is designed for distributed source code manipulation.
  - Built-in merge functions designed around text files.
  - Let's use it for *distributed data structures* instead!

Irmin, large-scale, immutable, branch-consistent storage

- Irmin is a library to **persist** and **synchronize distributed data structures** both on-disk and in-memory
- It enables a style of programming very similar to the **Git workflow**, where distributed **nodes fork, fetch, merge and push** data between each other
- The general idea is that you want every active node to get a **local** (partial) **copy of a global database** and always be very explicit about how and when data is shared and migrated


Irmin Store
Tag Store
P2
HEAD
h3
h4
h1
h2
Block Store

```
type t = ...
(** User-defined contents. *)

type result = [
    `Ok of t
  | `Conflict of string
  ]

val merge:
    old:t → t → t → result
(** 3-way merge functions. *)
```



Demo: Distributed Logging

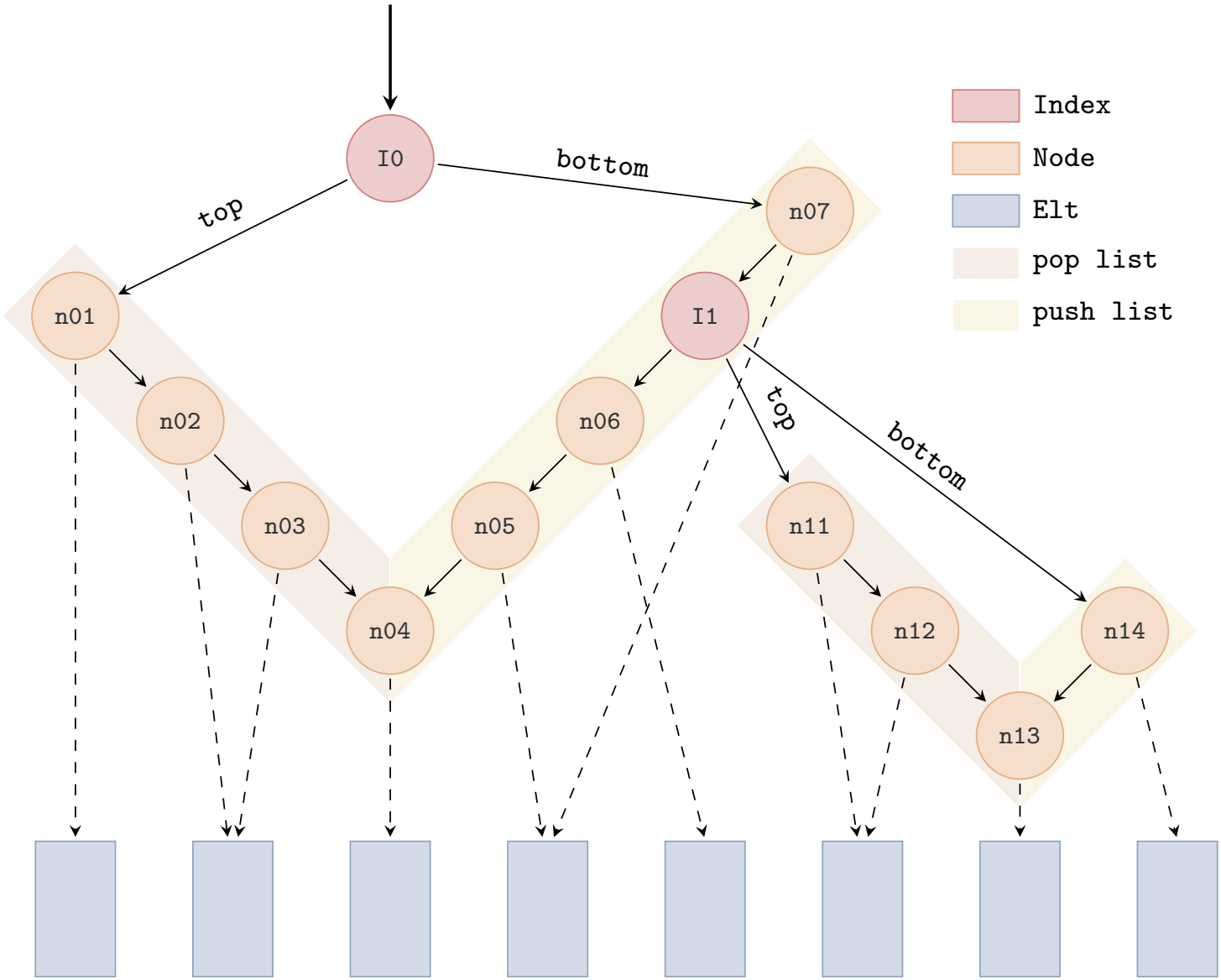
Multiple nodes all logging to a central store:

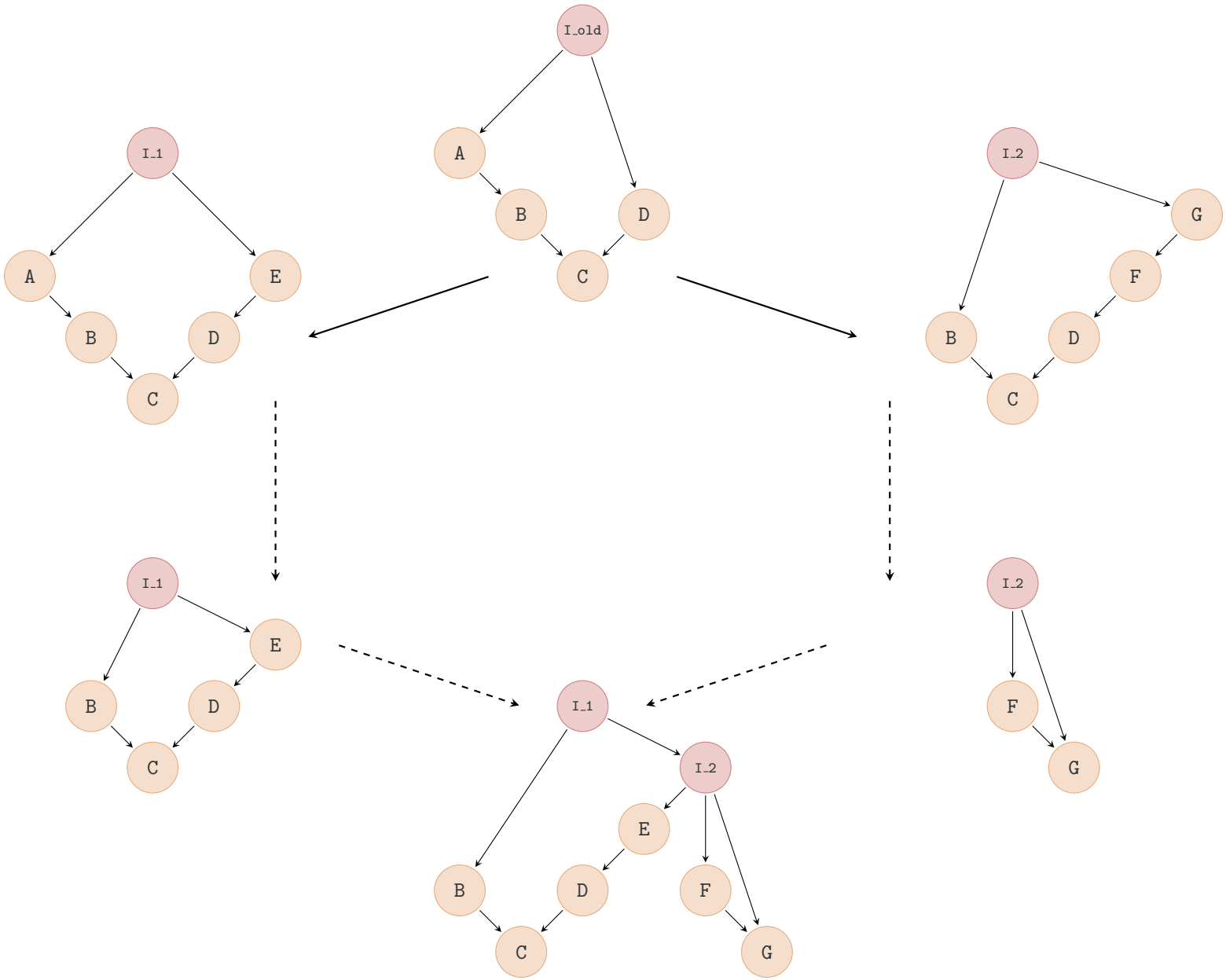
1. Design the logging data structure.
   - A log is a list of (string + timestamp)
   - When merging, the timestamps must be in increasing order
   - Equal timestamps can be in any order
   - With this logic, merge conflicts are impossible
2. Every node clones the log repository
3. A log is recorded locally, then pushed centrally.

Weakly consistent data structures

Mergeable queues

```
module type IrminQueue.S = sig
  type t
  type elt

  val create : unit → t
  val length : t → int
  val is_empty : t → bool

  val push : t → elt → t
  val pop : t → (elt * t)
  val peek : t → (elt * t)

  val merge : IrminMerge.t
end
```


I0
top
bottom
n07
n01
I1
n02
n06
top
bottom
n03
n05
n11
n04
n12
n14
n13
Index
Node
Elt
pop list
push list


I_old
A
B
D
C
I_1
A
E
B
D
C
I_2
G
F
B
D
C
I_1
E
B
D
C
I_2
F
G
I_1
I_2
E
B
D
F
C
G

Current state

Operation	Read	Write	
Push	0	2	$O(1)$
Pop	2 on average	1 on average	$O(1)$
Merge	n	1	$O(n)$

Current state

Operation	Read	Write	
Push	0	2	$O(1)$
Pop	2 on average	1 on average	$O(1)$
Merge	n	1	$O(n)$

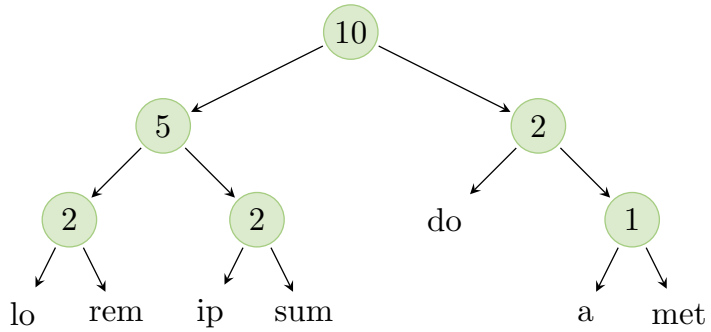
With a little more work

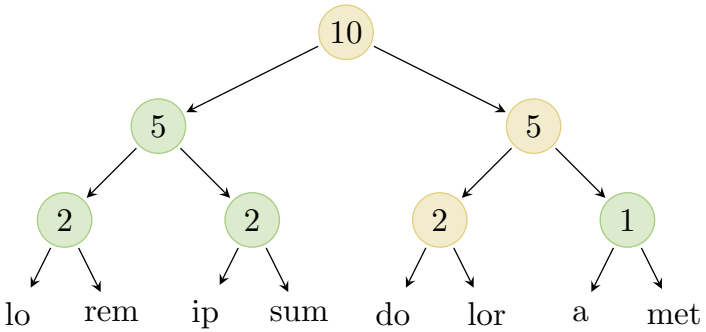
Operation	Read	Write	
Push	0	2	$O(1)$
Pop	2 on average	1 on average	$O(1)$
Merge	$\log n$	1	$O(\log n)$

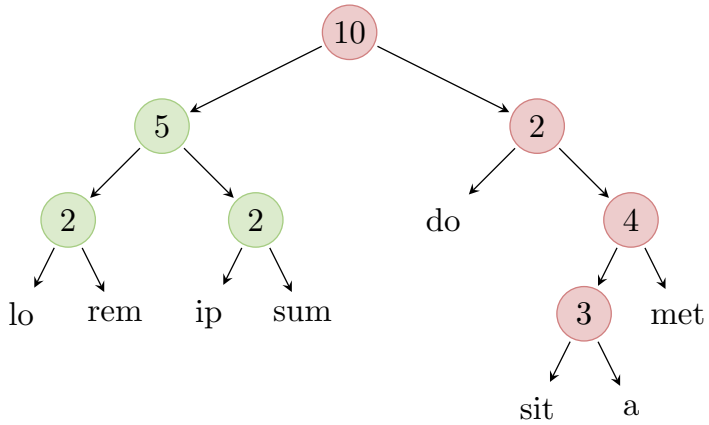
Mergeable ropes

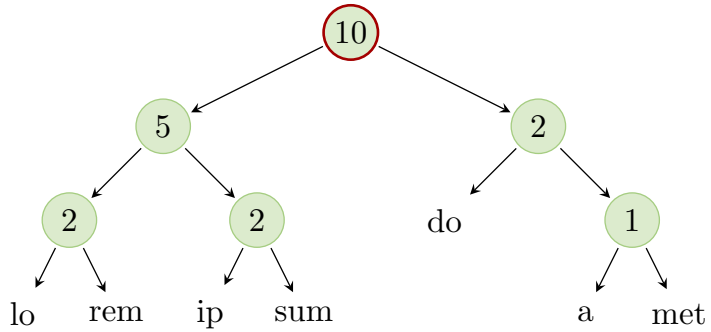
```
module type IrminRope.S = sig
  type t
  type value (* e.g char *)
  type cont  (* e.g string *)

  val create : unit → t
  val make : cont → t
  ...
  val set : t → int → value → t
  val get : t → int → value
  val insert : t → int → cont → t
  val delete : t → int → int → t
  val append : t → t → t
  val split : t → int → (t * t)

  val merge : IrminMerge.t
end
```

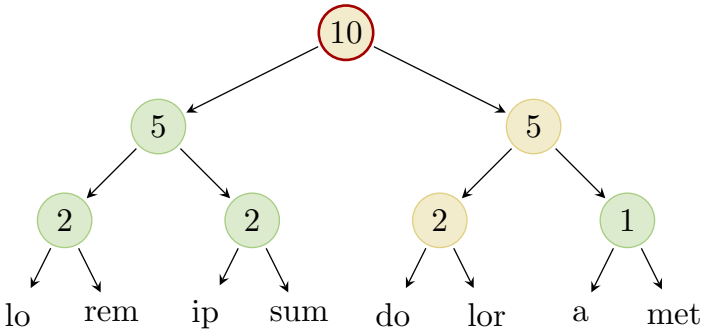
Operation	Rope	String
Set/Get	$O(\log n)$	$O(1)$
Split	$O(\log n)$	$O(1)$
Concatenate	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$
Merge	$\log(f(n))$	$f(n)$

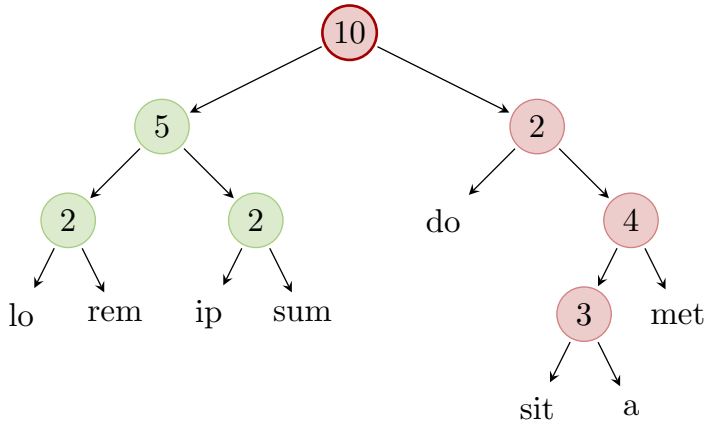

10
5
2
2
2
do
1
lo
rem
ip
sum
a
met

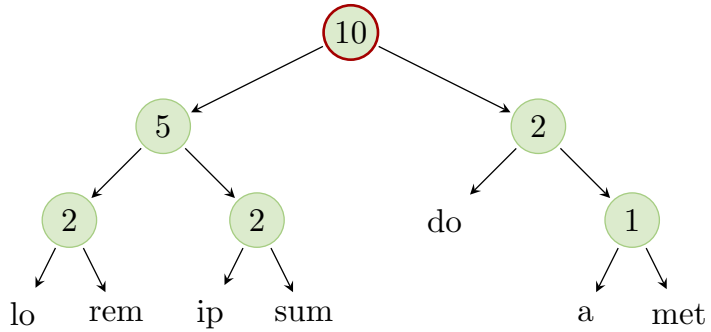

10
5
5
2
2
2
1
lo
rem
ip
sum
do
lor
a
met

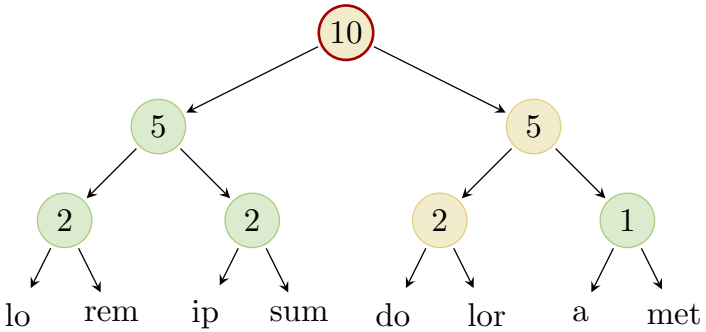

10
5
2
2
2
do
4
3
met
lo
rem
ip
sum
sit
a

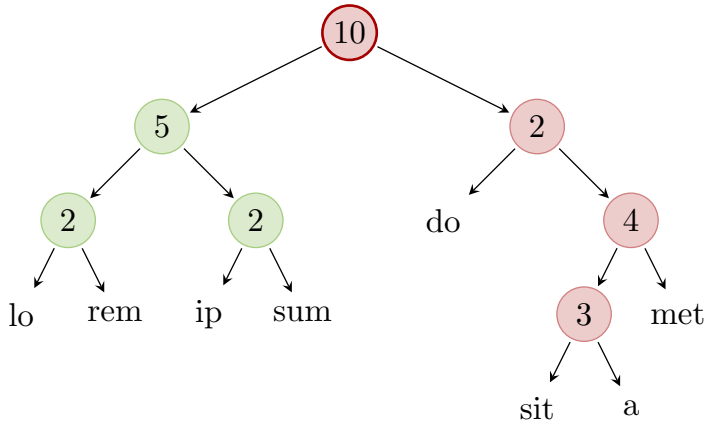

10
5
2
2
2
do
1
lo
rem
ip
sum
a
met

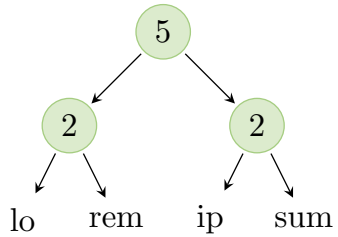

10
5
5
2
2
2
1
lo
rem
ip
sum
do
lor
a
met

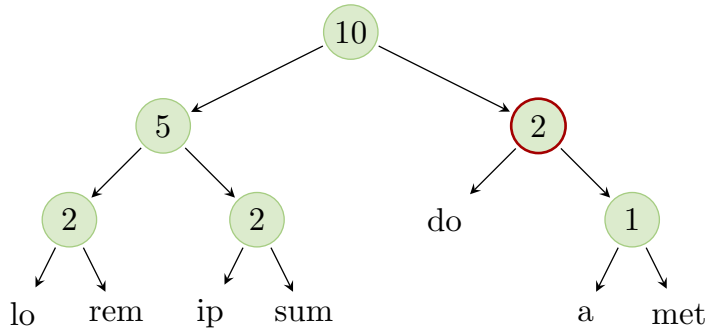

10
5
2
2
2
do
4
3
met
lo
rem
ip
sum
sit
a

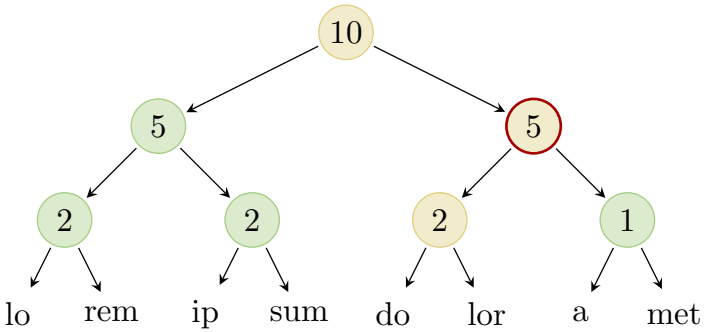

10
5
2
2
2
do
1
lo
rem
ip
sum
a
met

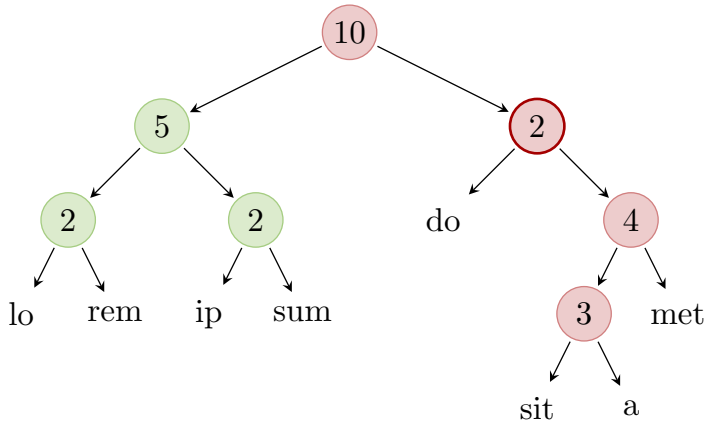

10
5
5
2
2
2
1
lo
rem
ip
sum
do
lor
a
met

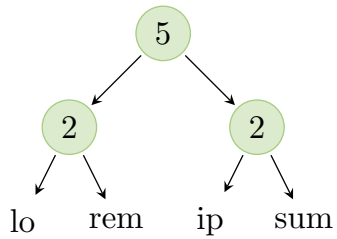

10
5
2
2
2
do
4
3
met
lo
rem
ip
sum
sit
a

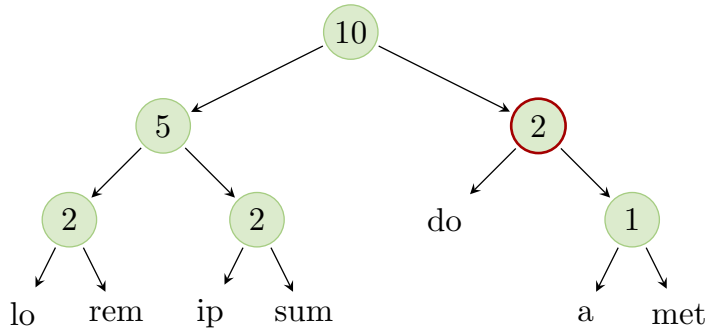

5
2
2
lo
rem
ip
sum

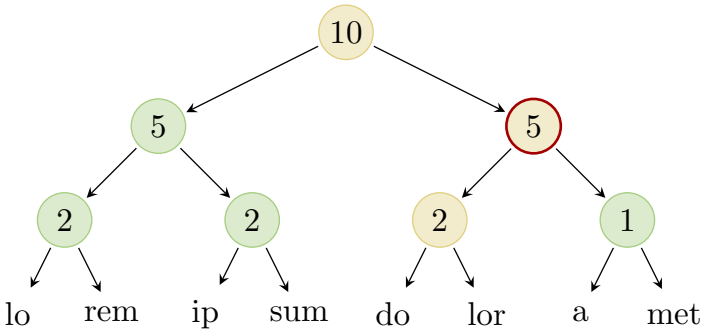

10
5
2
2
2
do
1
lo
rem
ip
sum
a
met

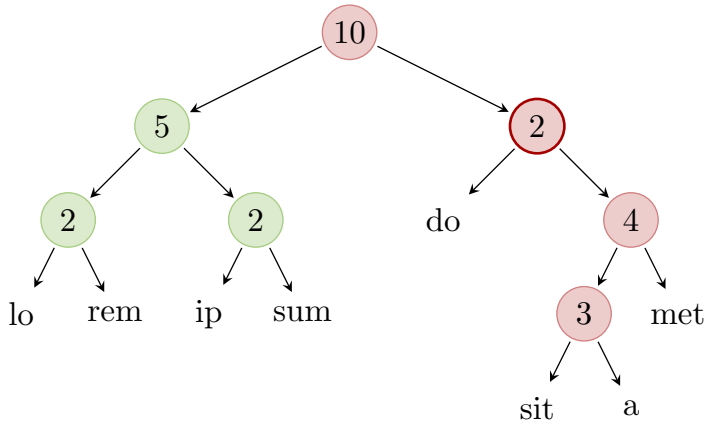

10
5
5
2
2
2
1
lo
rem
ip
sum
do
lor
a
met

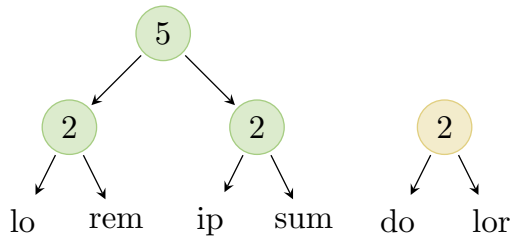

10
5
2
2
2
do
4
3
met
lo
rem
ip
sum
sit
a

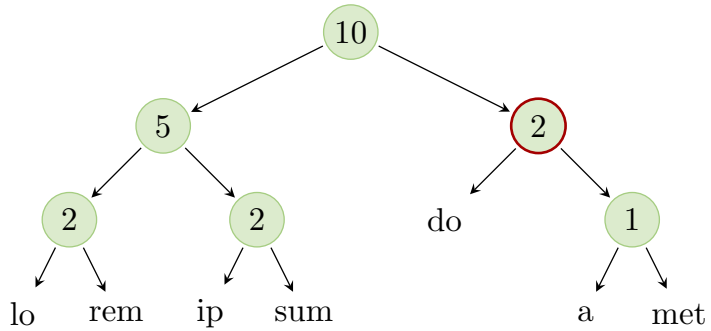

5
2
2
lo
rem
ip
sum

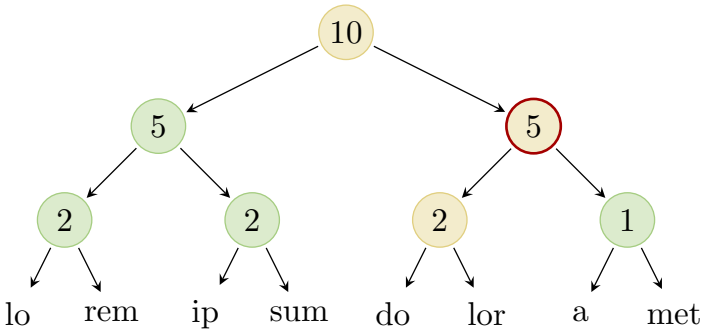

10
5
2
2
2
do
1
lo
rem
ip
sum
a
met

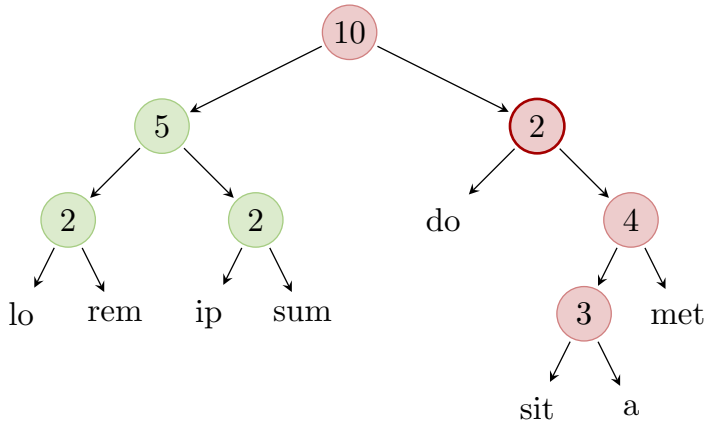

10
5
5
2
2
2
1
lo
rem
ip
sum
do
lor
a
met

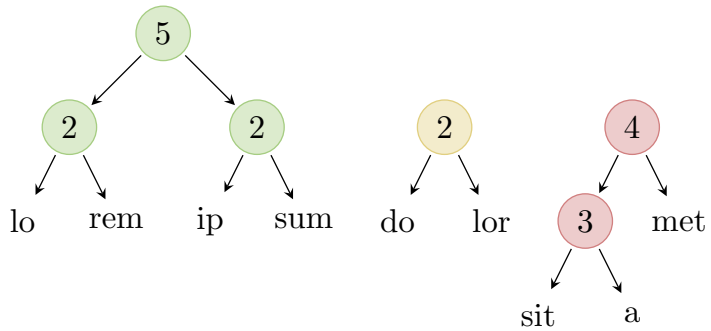

10
5
2
2
2
do
4
3
met
lo
rem
ip
sum
sit
a

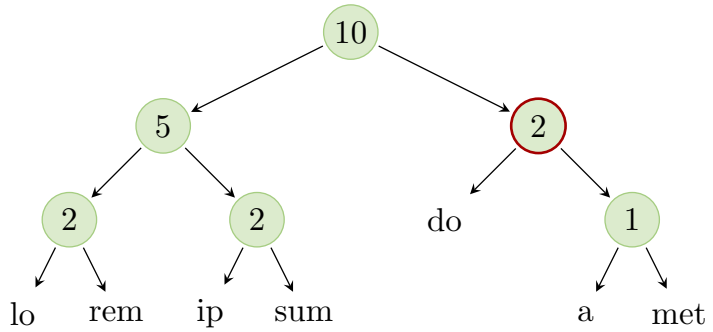

5
2
2
2
lo
rem
ip
sum
do
lor

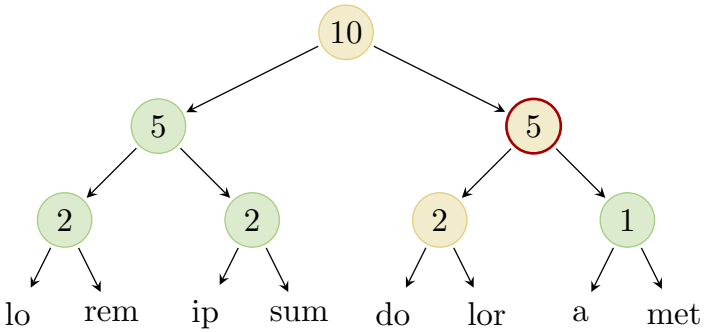

10
5
2
2
2
do
1
lo
rem
ip
sum
a
met

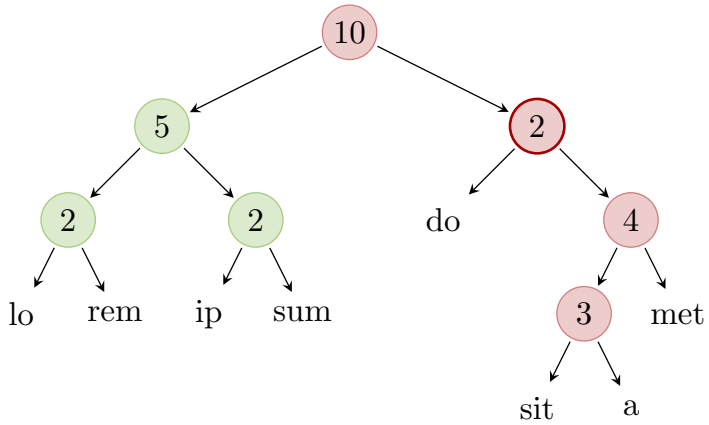

10
5
5
2
2
2
1
lo
rem
ip
sum
do
lor
a
met

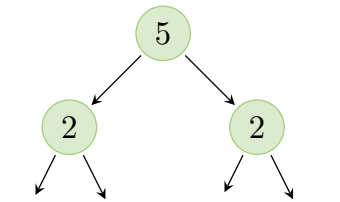

10
5
2
2
2
do
4
3
met
lo
rem
ip
sum
sit
a

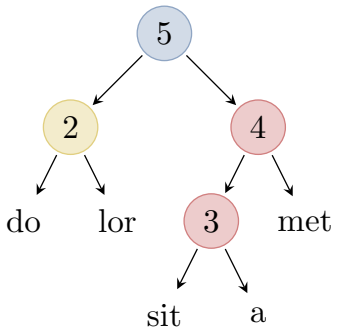

5
2
2
2
4
3
met
lo
rem
ip
sum
do
lor
sit
a

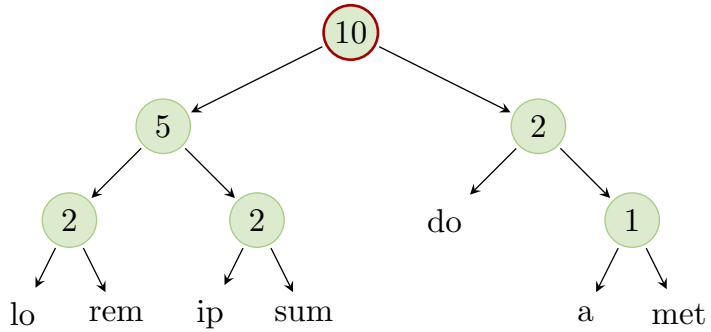

10
5
2
2
2
do
1
lo
rem
ip
sum
a
met

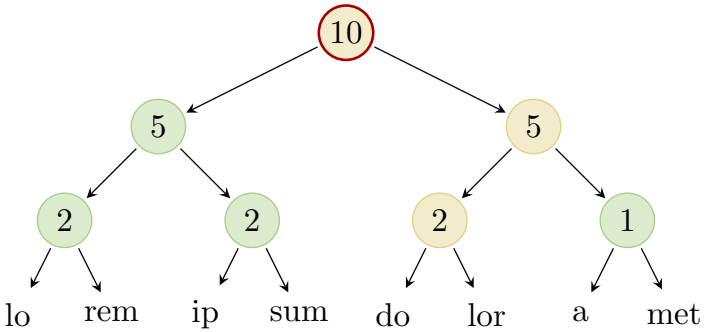

10
5
5
2
2
2
1
lo
rem
ip
sum
do
lor
a
met

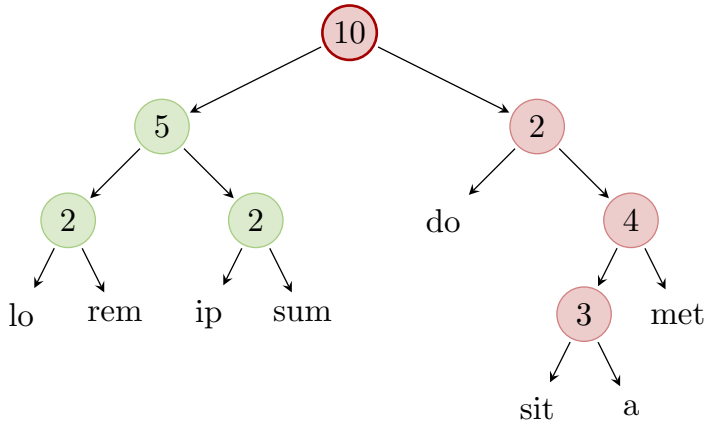

10
5
2
2
2
do
4
3
met
lo
rem
ip
sum
sit
a

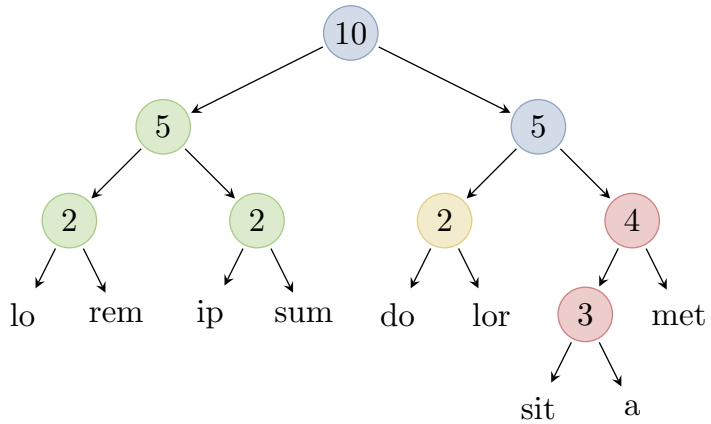

5
2
2
lo
rem
ip
sum

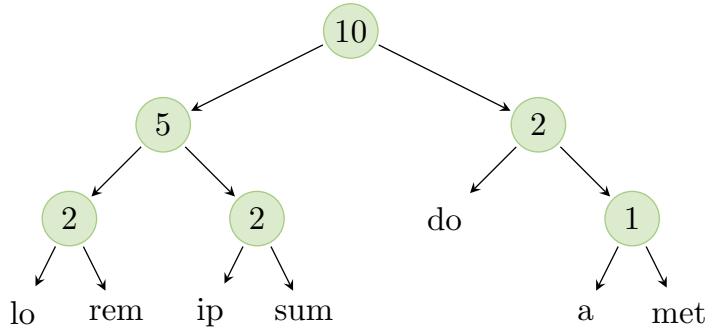

5
2
4
do
lor
3
met
sit
a

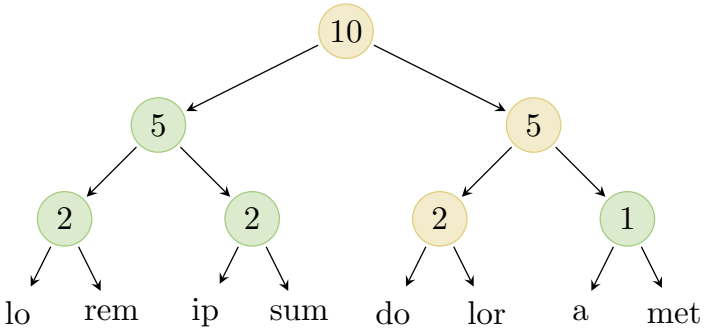

10
5
2
2
2
do
1
lo
rem
ip
sum
a
met

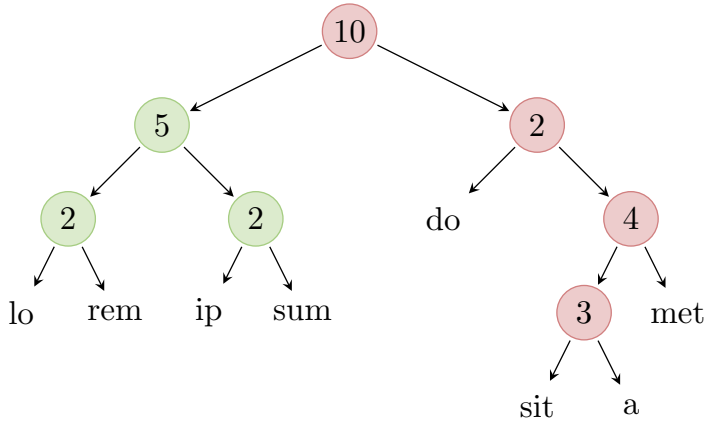

10
5
5
2
2
2
1
lo
rem
ip
sum
do
lor
a
met

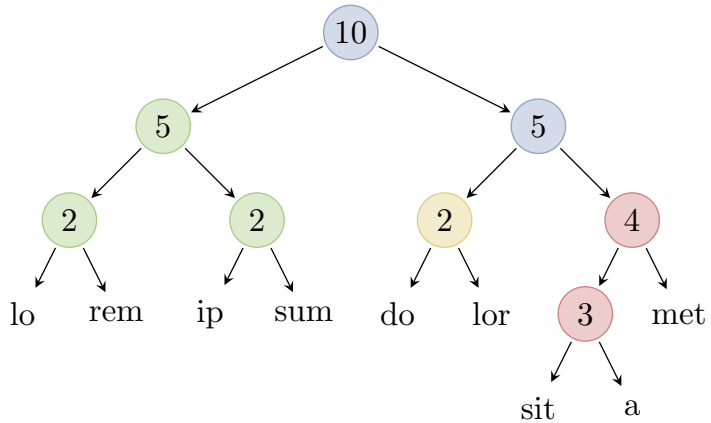

10
5
2
2
2
do
4
3
met
lo
rem
ip
sum
sit
a


10
5
5
2
2
2
4
3
met
lo
rem
ip
sum
do
lor
sit
a

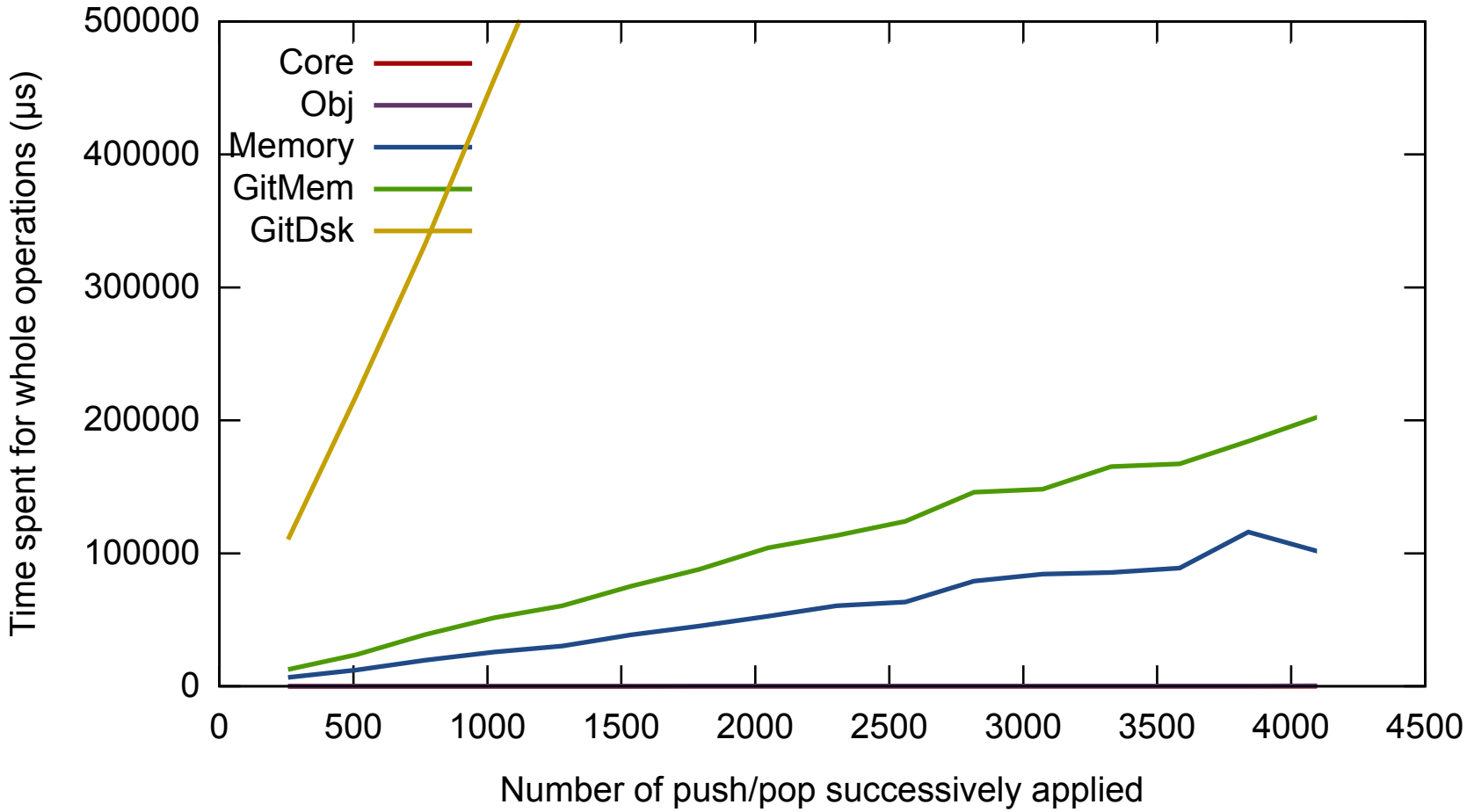

10
5
2
2
2
do
1
lo
rem
ip
sum
a
met


10
5
5
2
2
2
1
lo
rem
ip
sum
do
lor
a
met

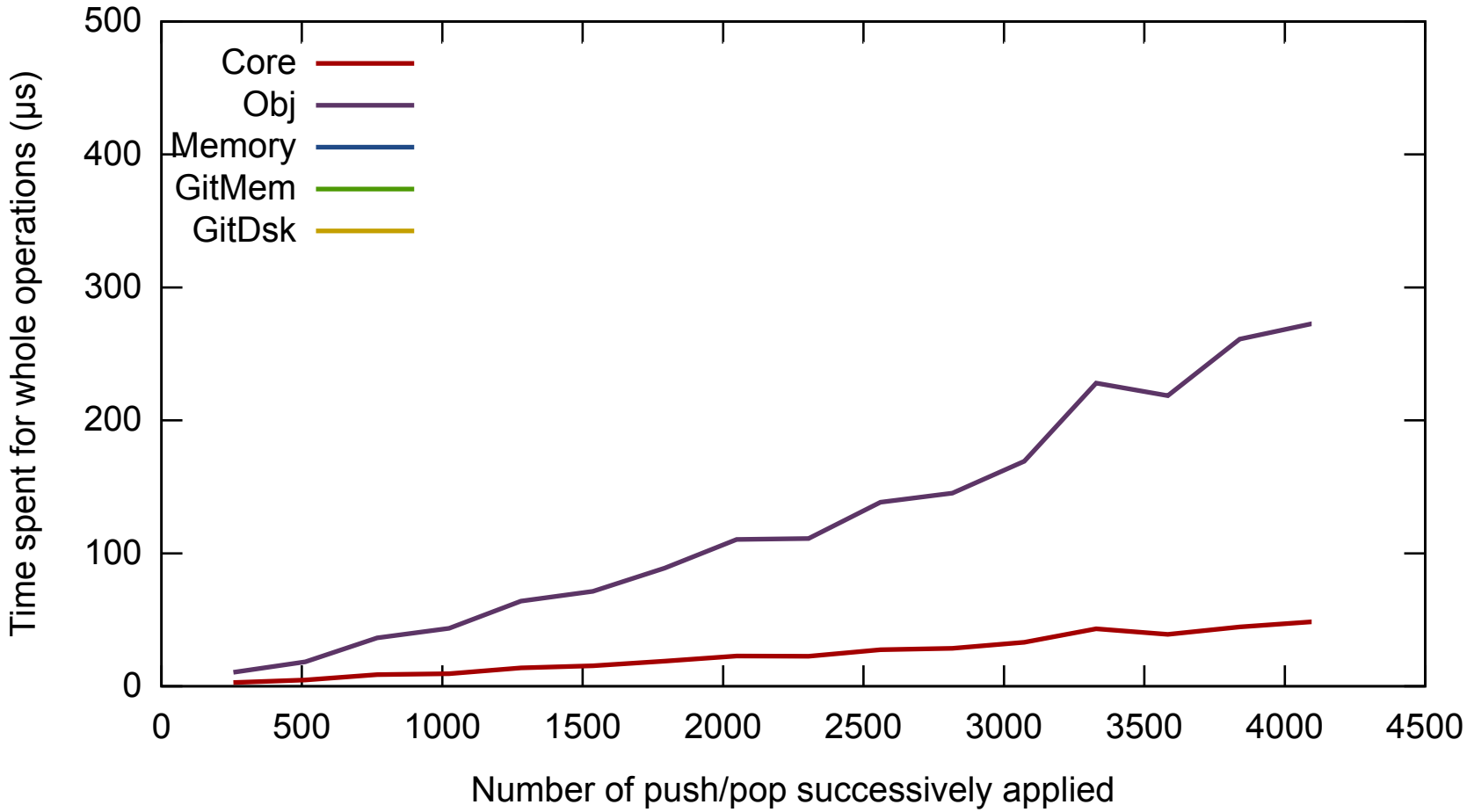

10
5
2
2
2
do
4
3
met
lo
rem
ip
sum
sit
a


10
5
5
2
2
2
4
3
met
lo
rem
ip
sum
do
lor
sit
a

Benchmarking Irmin


Time spent for whole operations (µs)
500000
400000
300000
200000
100000
0
Core
Obj
Memory
GitMem
GitDsk
0
500
1000
1500
2000
2500
3000
3500
4000
4500
Number of push/pop successively applied

```
module ObjBackend ...  = struct
  type t = unit
  type key = K.t
  type value = V.t

  let create () = return ()
  let clear () = return ()

  let add t value =
    return (Obj.magic (Obj.repr value))

  let read t key =
    return (Obj.obj (Obj.magic key))

  let mem t key = return true
  ...
end
```


Core
Obj
Memory
GitMem
GitDsk
Time spent for whole operations (µs)
Number of push/pop successively applied
0
100
200
300
400
500
0
500
1000
1500
2000
2500
3000
3500
4000
4500

Use Cases

Demo: Dog, a loyal synchronization tool

Command line interface to logging clients at `https://github.com/samoht/dog`

1. `dog listen` to setup the server listener
   - Server maintains list of clients in a subtree
   - It regularly merges all clients in parallel to master branch
2. `dog init` starts up a client logger
3. `dog push` syncs the client with the server

Demo: CueKeeper, an Irmin TODO manager

Do Git programming in the browser

`https://github.com/talex5/cuekeeper`

`http://test.roscidus.com/CueKeeper/`

1. Irmin is written in OCaml, and compiles to efficient JavaScript
   - Git objects are mapped into IndexedDB
   - Uses LocalStorage to sync between tabs
2. DOM elements are computed from the Git store (a React-like model)
3. Client has full history, snapshotting and custom merge logic.

Demo: XenStore TNG

The Xen hypervisor toolstack

`https://www.youtube.com/watch?v=DSzvFwIVm5s`

1. Xen is a widely deployed hypervisor (Amazon EC2, Rackspace Cloud, ...)
   - Every VM boot needs a lot of communication
   - Tracing when something goes wrong is hard
   - Programming model is quite reactive
2. Dave Scott from Citrix ported the core toolstack to use Irmin, and made it faster!

Why OCaml?

- Let us prototype complex functional datastructures very quickly
- Efficient compilation to native code (x86, ARM, PowerPC, Sparc, ...), unikernels (MirageOS), JavaScript and Java
- Execution model is strict and predictable, important for systems programming
- Native code compilation is statically linked, or can be used as a normal shared library

Irmin Status ("Not Entirely Insane")

- Still pre 1.0, but several useful datastructures such as distributed queues and efficient ropes.
- HTTP REST for remote clients, library via OCaml, or command-line interface.
- Bidirectional operation, so `git` commits map to Irmin commits from any direction.
- Open source at `https://irmin.io`, installable via the OPAM package manager at `https://opam.ocaml.org`
- Feedback welcome at `mirageos-devel@lists.xenproject.org` or `https://github.com/mirage/irmin/issues`