Verifying a Lazy Concurrent List-Based Set Algorithm in Iris

Daniel Nezamabadi
Supervisor: Isaac van Bakel, Prof. Dr. Ralf Jung

Research in Computer Science, ETH Zurich
February 3rd, 2025

We want a set algorithm that…

…is concurrent

…has good performance

…implementable

…**is correct**

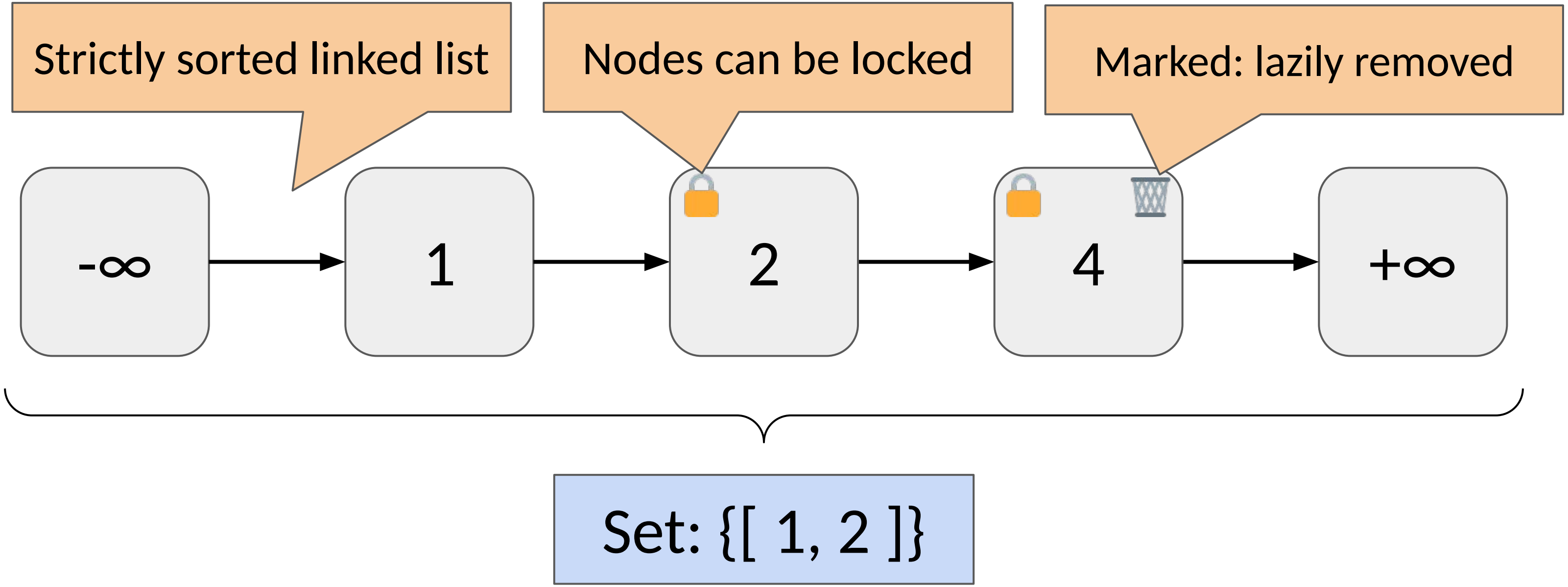
Our Candidate:

A Lazy Concurrent List-Based Set Algorithm

Steve Heller[1], Maurice Herlihy[2], Victor Luchangco[1], Mark Moir[1], William N. Scherer III[3], and Nir Shavit[1]

[1] Sun Microsystems Laboratories
[2] Brown University
[3] University of Rochester

Anatomy of a Lazy List-Based Set



High-Level Properties

Less lock contention,
good for performance

Optimistic locking:
Don't lock while walking the list

Common design
technique

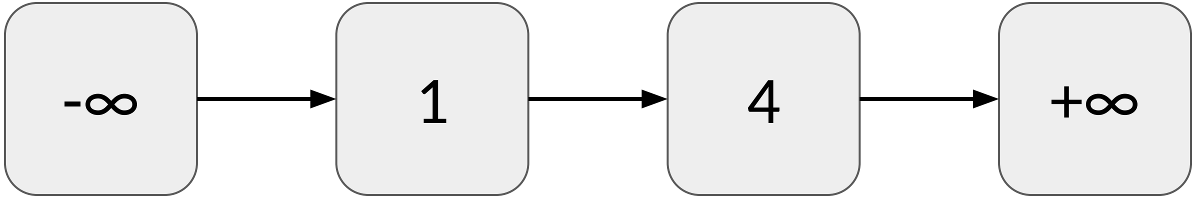
Lazy removal:
First mark a node, then unlink it

No lock (contention)

Wait-free contains:
Always finishes in finite #steps

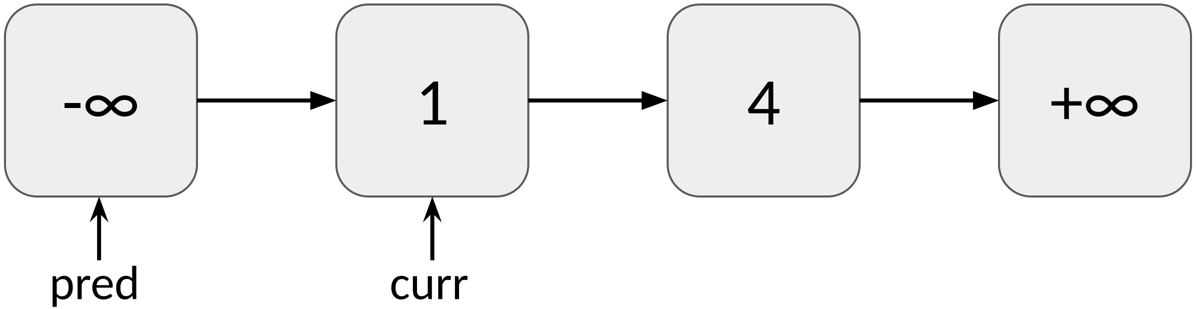
add(2)

Initial State



add(2)

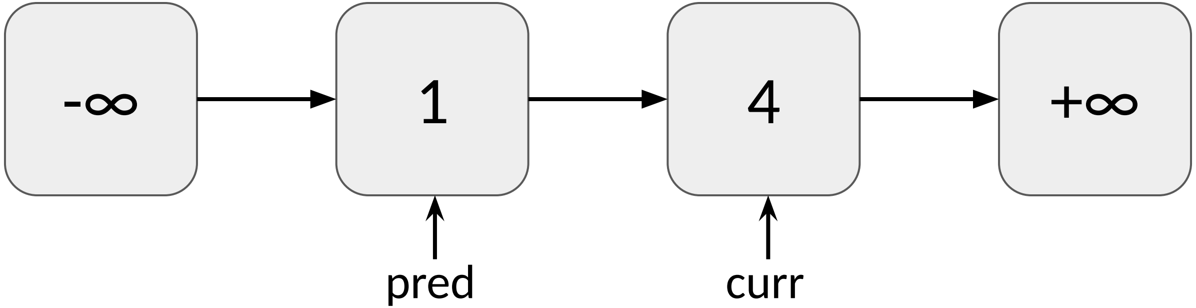
Thread 1 - add(2): walk



Optimistic: No locking yet

add(2)

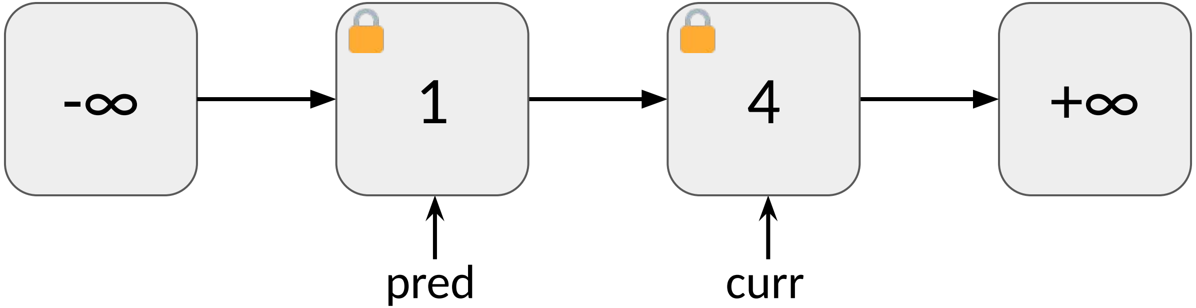
Thread 1 - add(2): walk is done



Optimistic: No locking yet

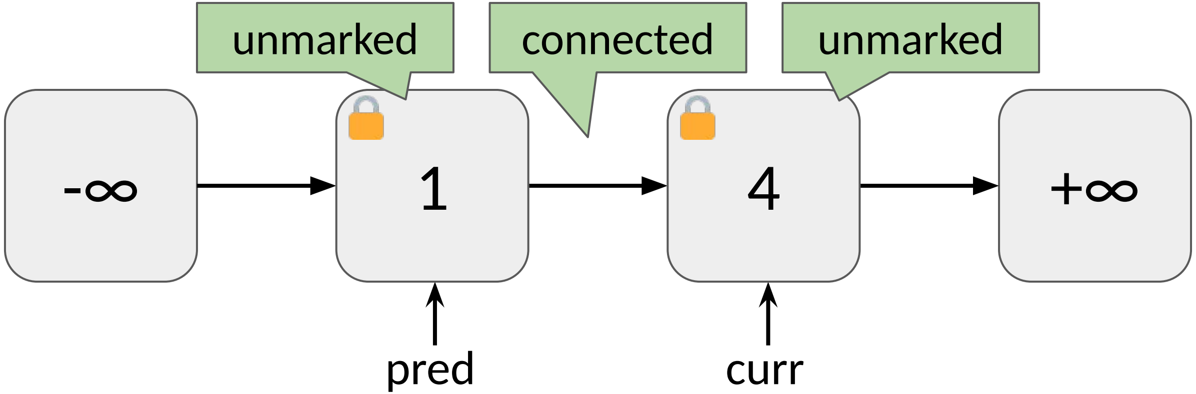
add(2)

Thread 1 - add(2): acquire locks



add(2)

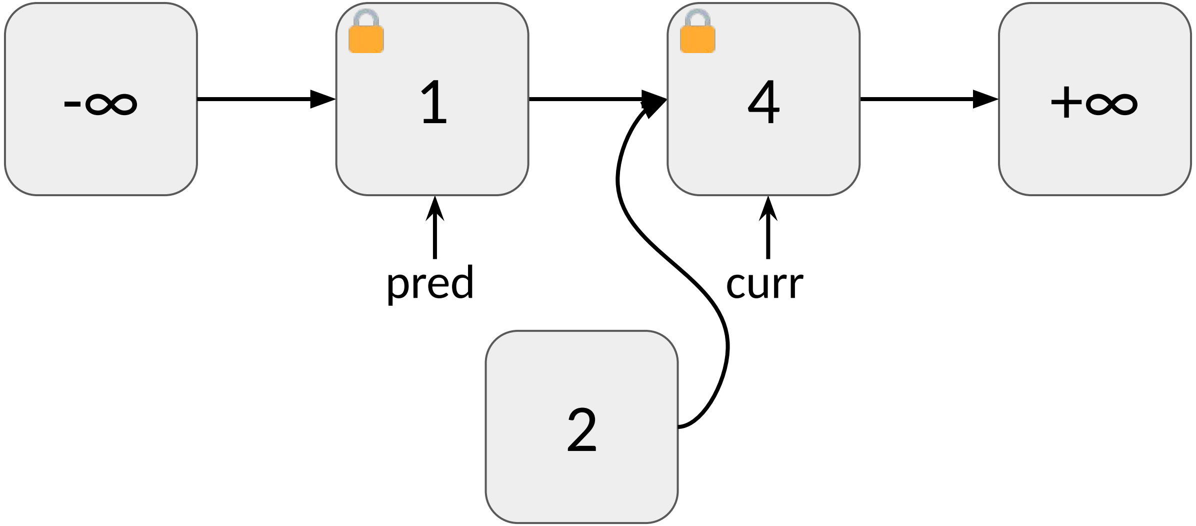
Thread 1 - add(2): validate state



Validation succeeds

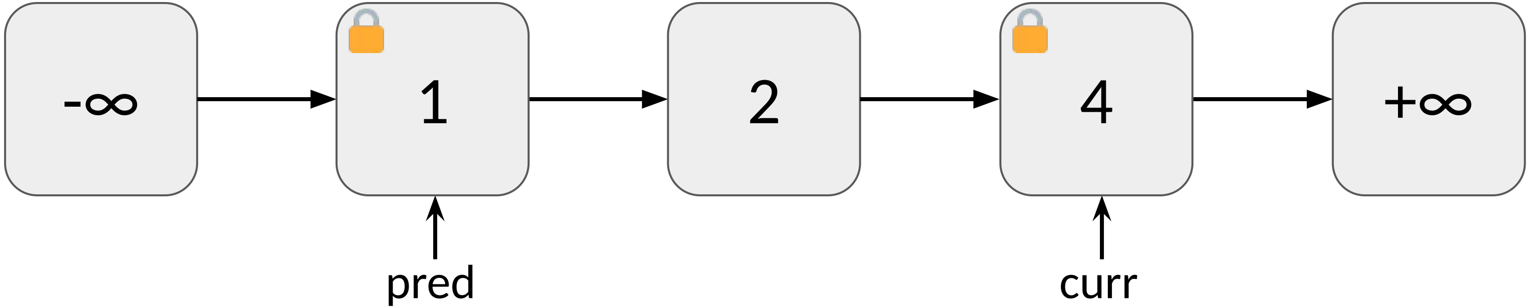
add(2)

Thread 1 - add(2): create new node



add(2)

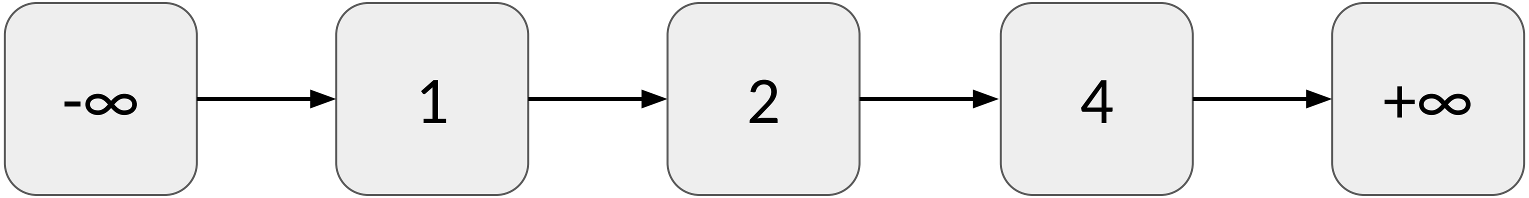
Thread 1: add(2) - **hook in new node**



Linearization Point: {[1, 4]} → {[1, 2, 4]}

add(2)

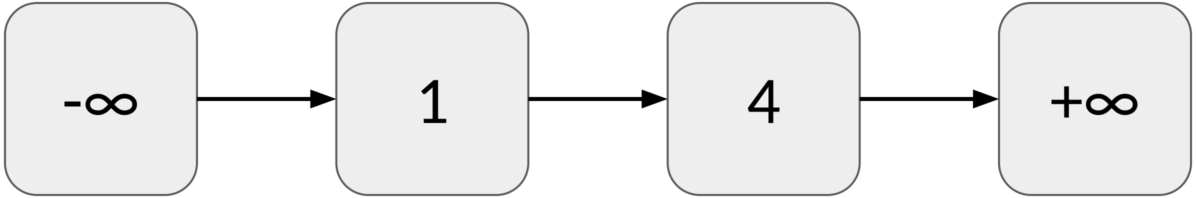
Thread 1: release locks



What if we are too optimistic?

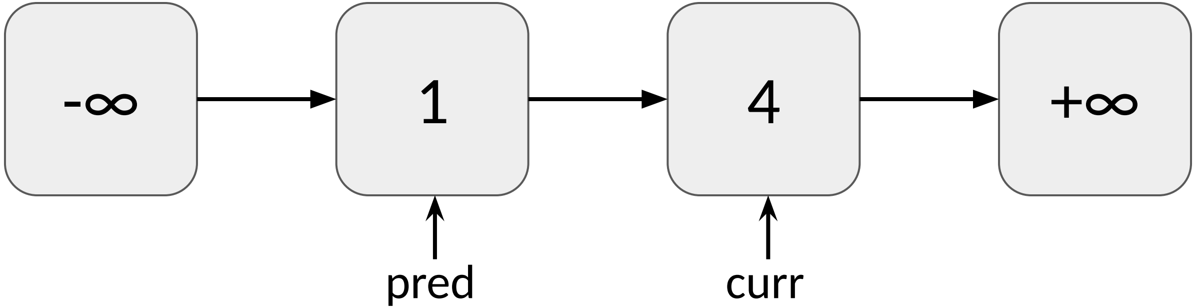
add(2) || add(2) ;; add(3)

Initial State



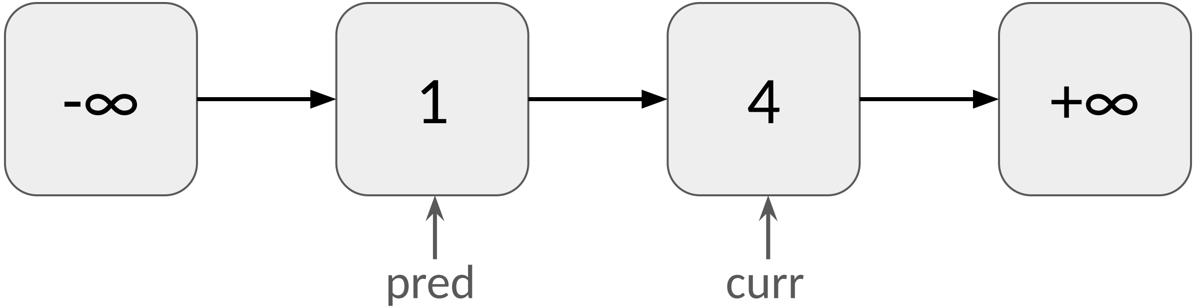
add(2) || add(2) ;; add(3)

Thread 1 - add(2): walk is done



add(2) || add(2) ;; add(3)

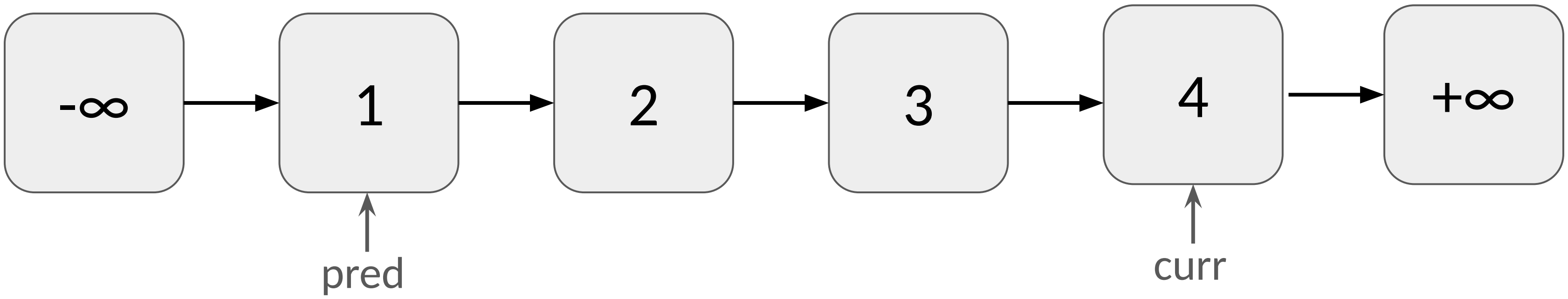
Thread 1 - add(2): preempted



add(2) || add(2) ;; add(3)

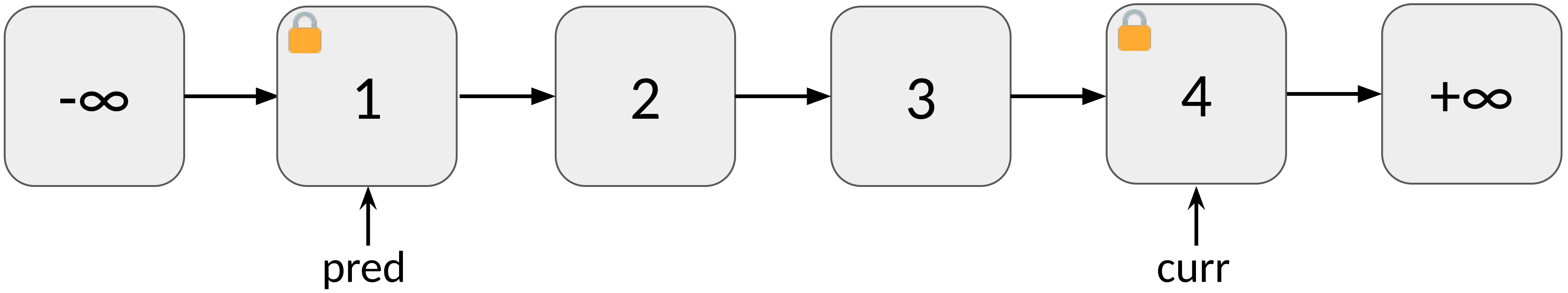
Thread 1 - add(2): preempted

Thread 2 - adds 2 and 3



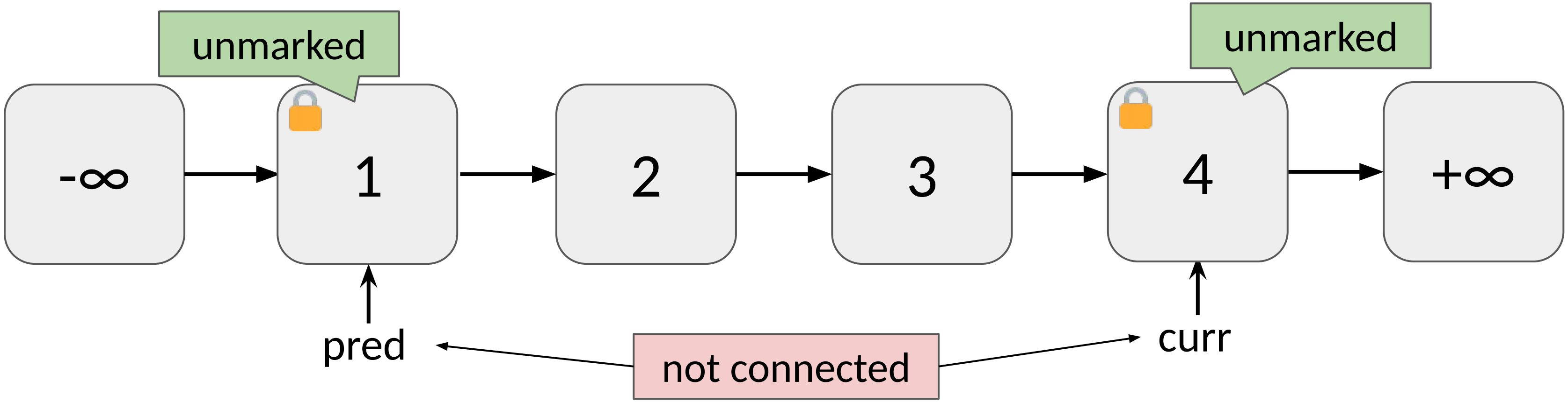
add(2) || add(2) ;; add(3)

Thread 1 - add(2): acquire locks



add(2) || add(2) ;; add(3)

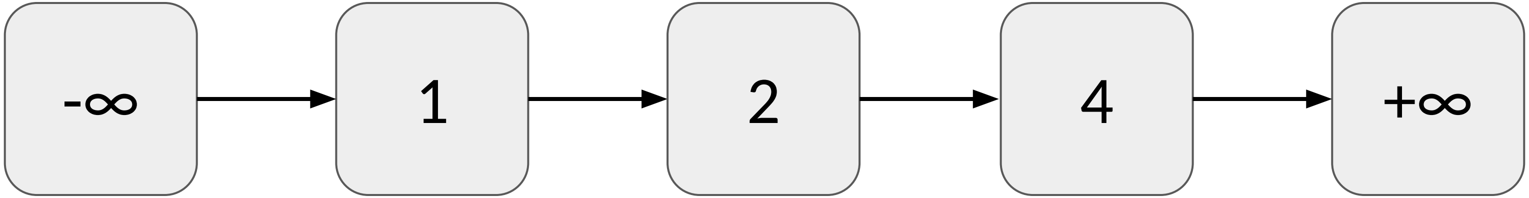
Thread 1 - add(2): validate state



Continuing would remove 3 ⇒ Abort & Retry

remove(4)

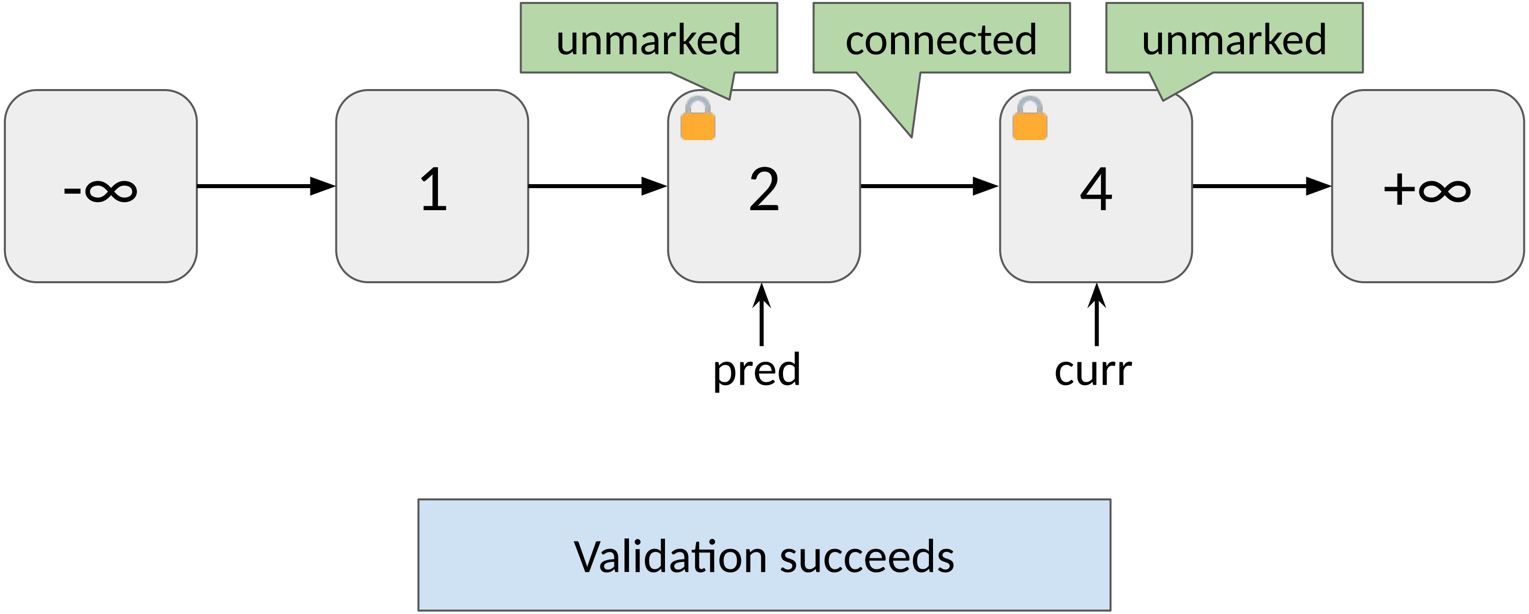
Initial State



walk, lock, and validate are the same as add

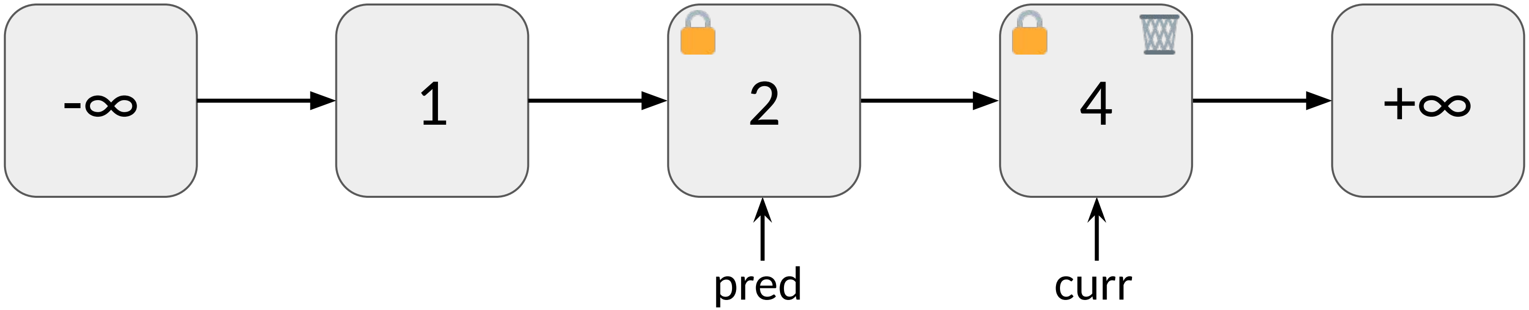
remove(4)

Thread 1 - remove(4): validate state



remove(4)

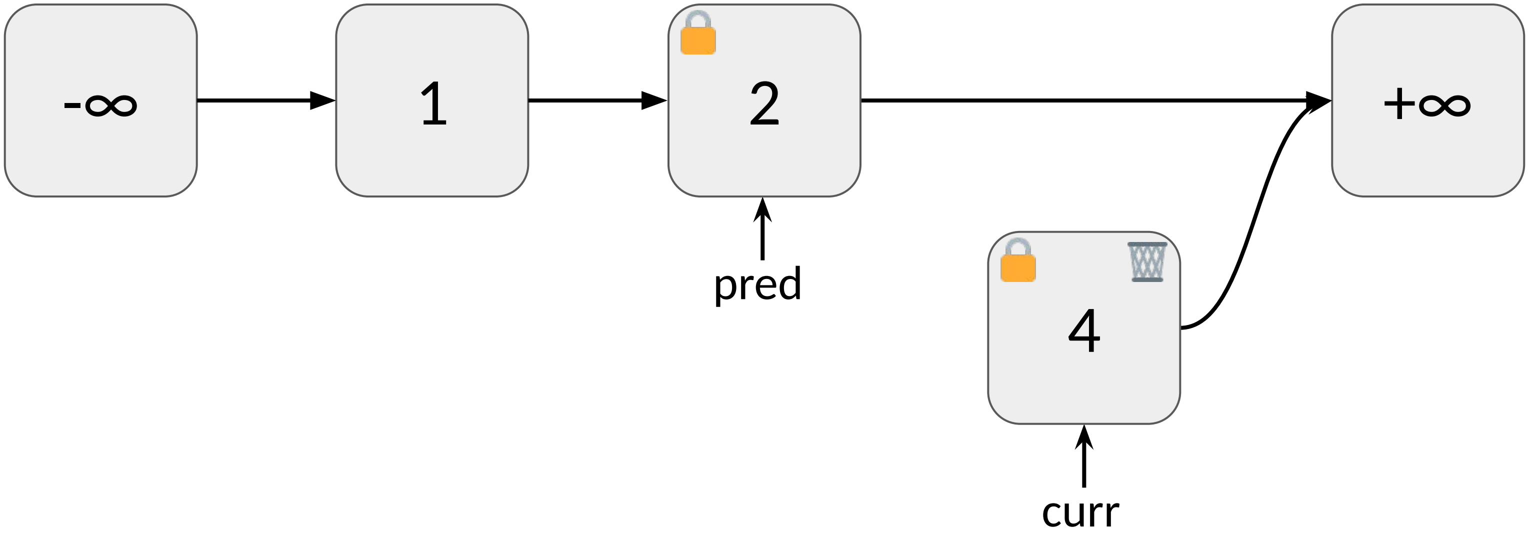
Thread 1 - remove(4): **lazily remove 4**



Linearization Point: {[1, 2, 4]} → {[1, 2]}

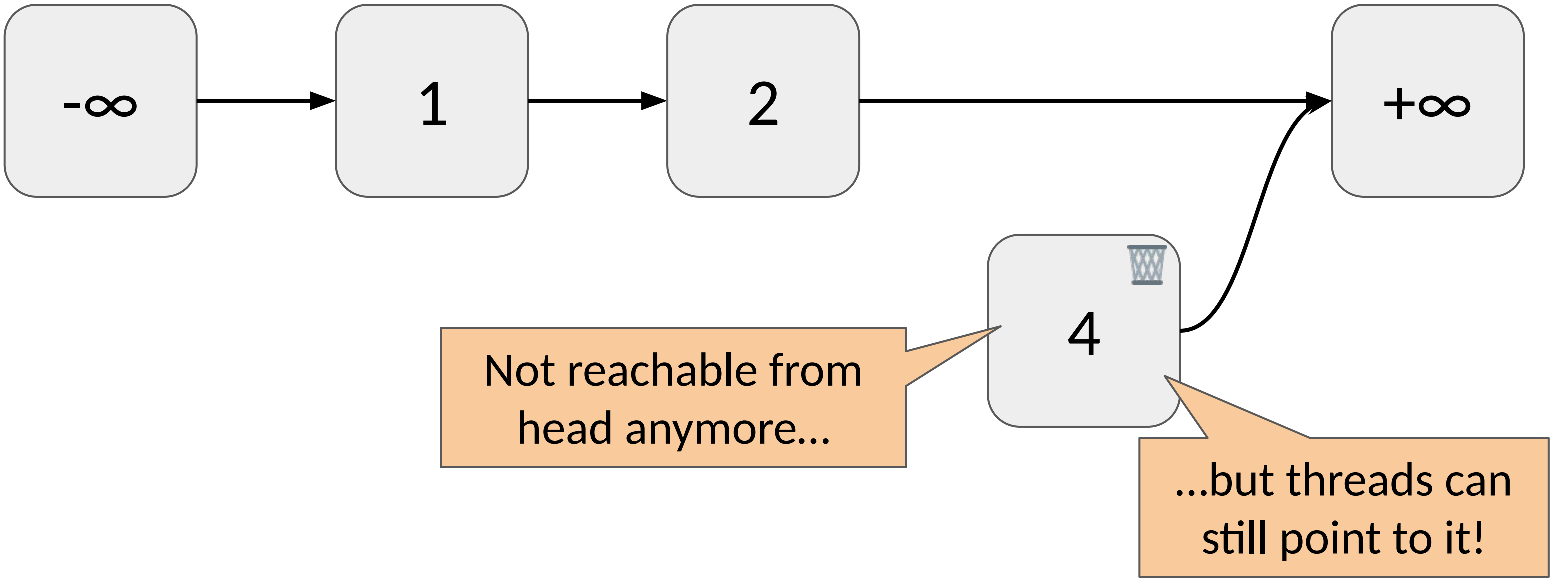
remove(4)

Thread 1 - remove(4): physically remove 4



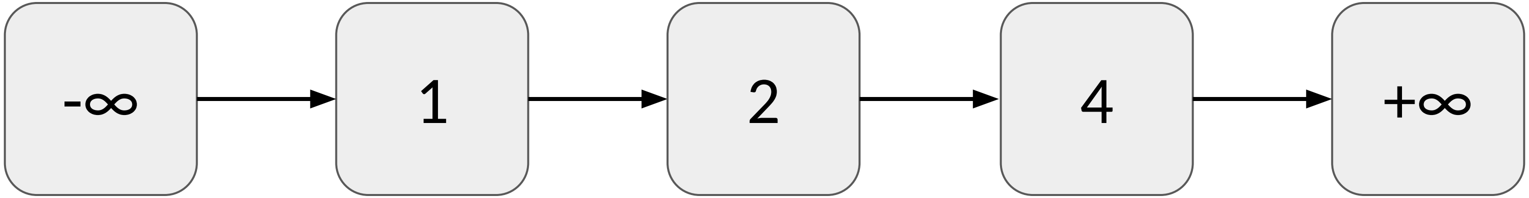
remove(4)

Thread 1 - remove(4): release locks



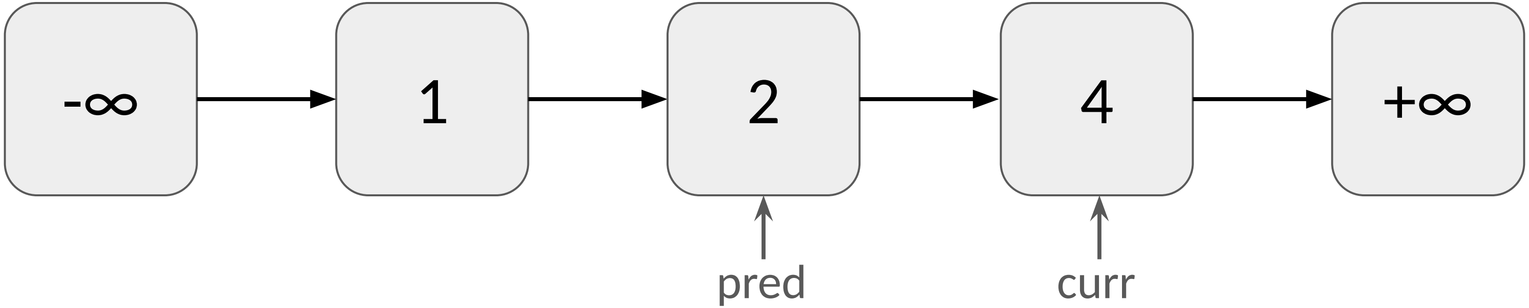
add(5) || remove(4)

Initial State



add(5) || remove(4)

Thread 1 - add(5): preempted during walk

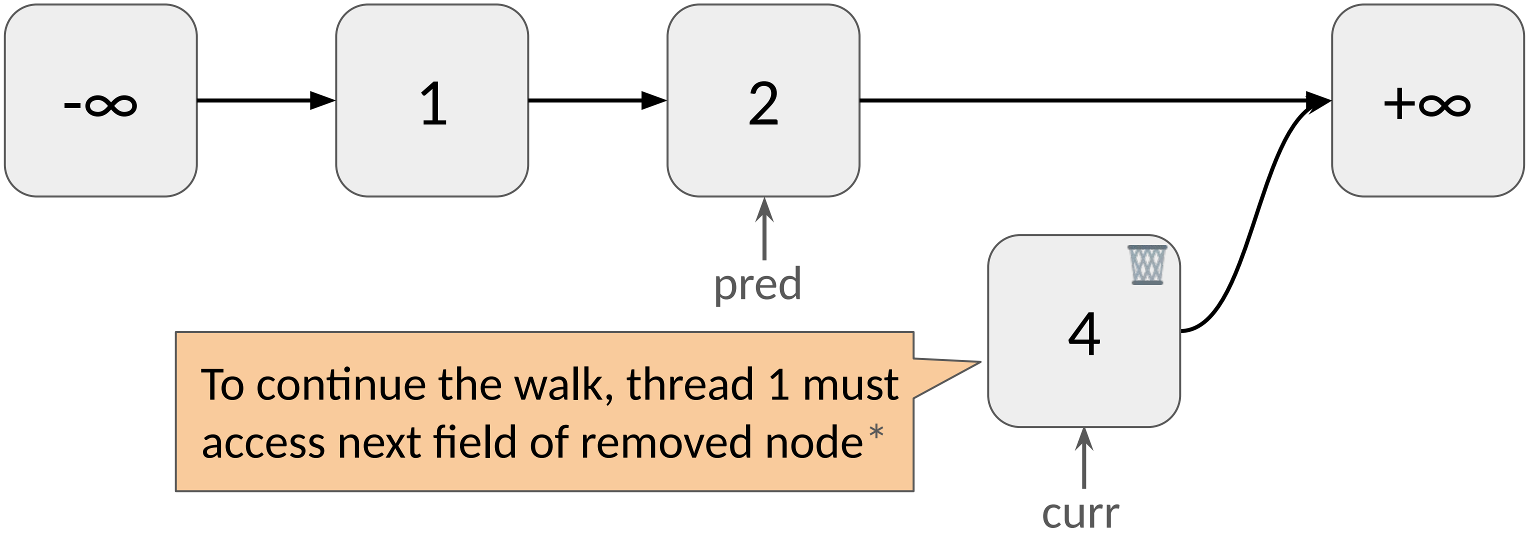


add(5) || remove(4)

Thread 1 - add(5): preempted during walk

Thread 2 - removes 4

Possible, because no locking during walk



To continue the walk, thread 1 must access next field of removed node*

* Figured this out the hard way.

What about contains?

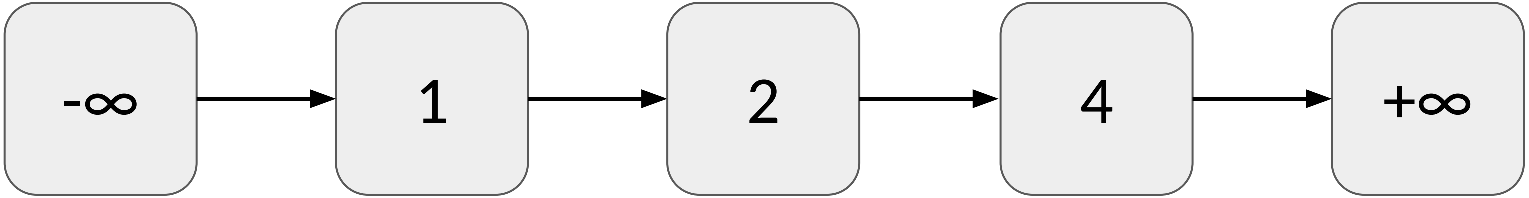
```
public boolean contains(int key) {
  Entry curr = this.head;
  while (curr.key < key)
    curr = curr.next;
  return curr.key == key && !curr.marked;
}
```

wait-free ⇒ lock-free

Simple implementation, tricky linearization point.

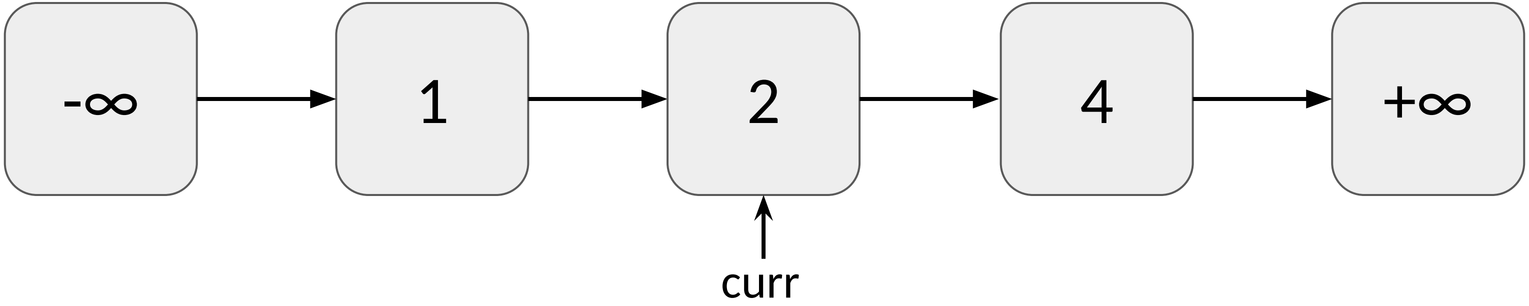
contains(2) || remove(2)

Initial State



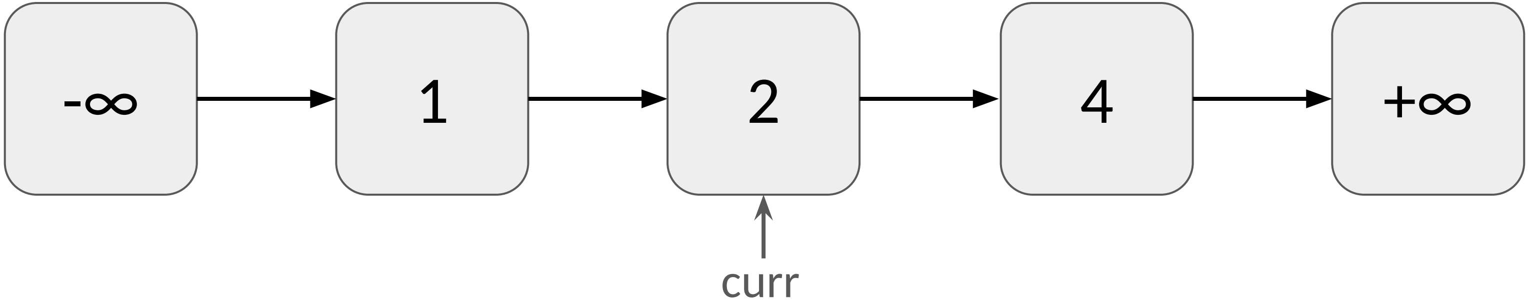
contains(2) || remove(2)

Thread 1 - contains(2): walk is done



contains(2) || remove(2)

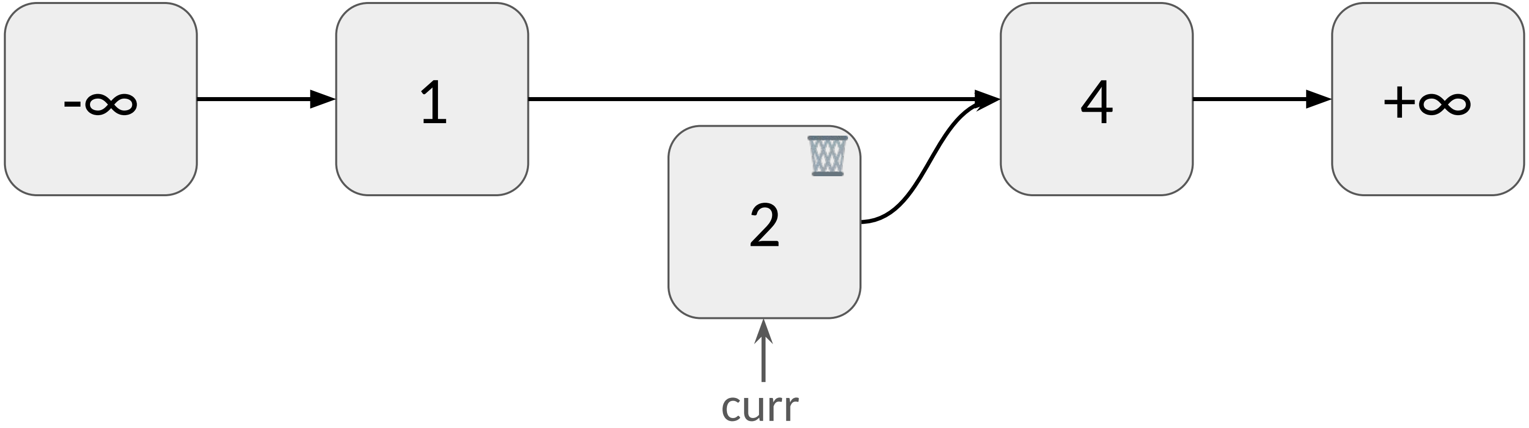
Thread 1 - contains(2): preempted



contains(2) || remove(2)

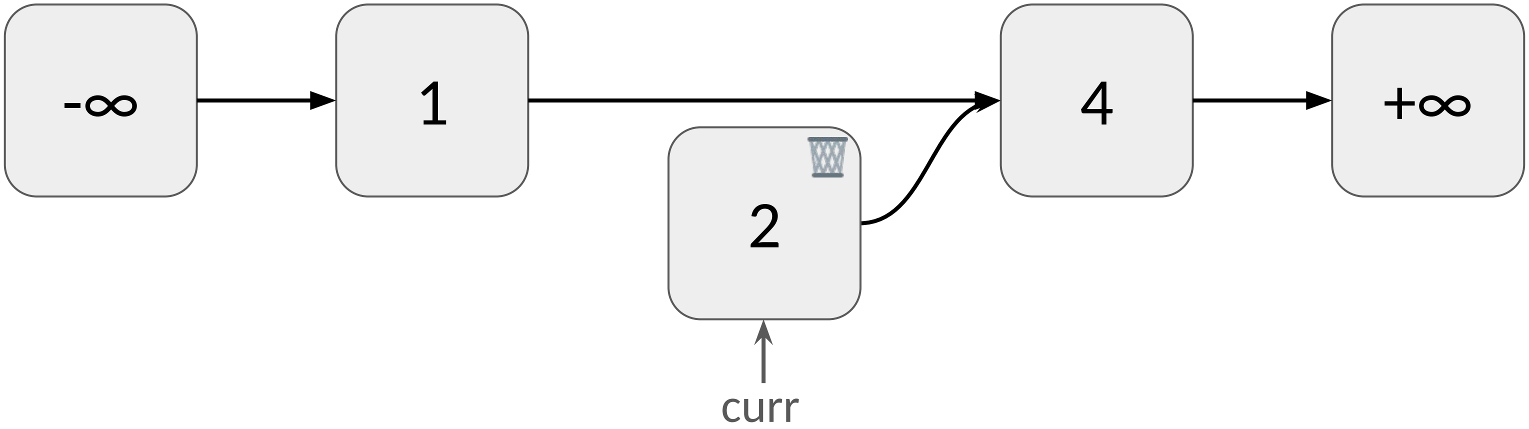
Thread 1 - contains(2): preempted

Thread 2 removes 2



contains(2) || remove(2)

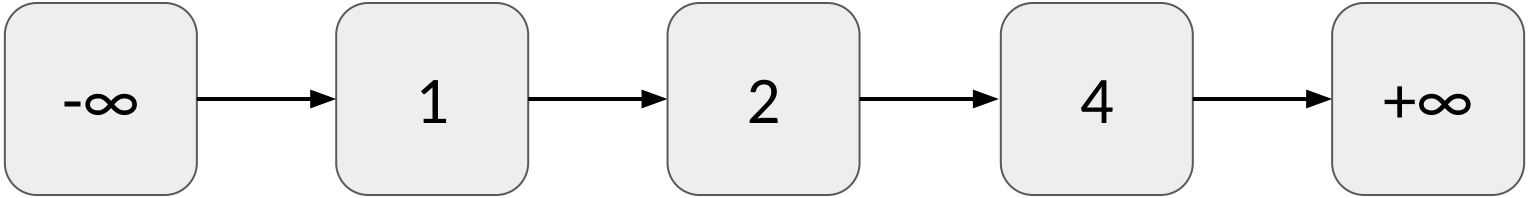
Thread 1 - contains(2): **reads marked is true**



Linearization Point: 2 is not in the set

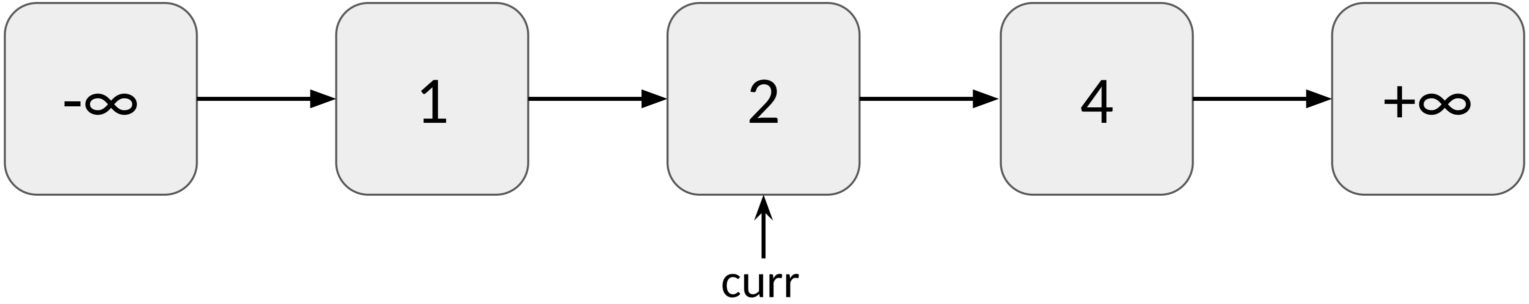
contains(2) || remove(2) ;; add(2)

Initial State



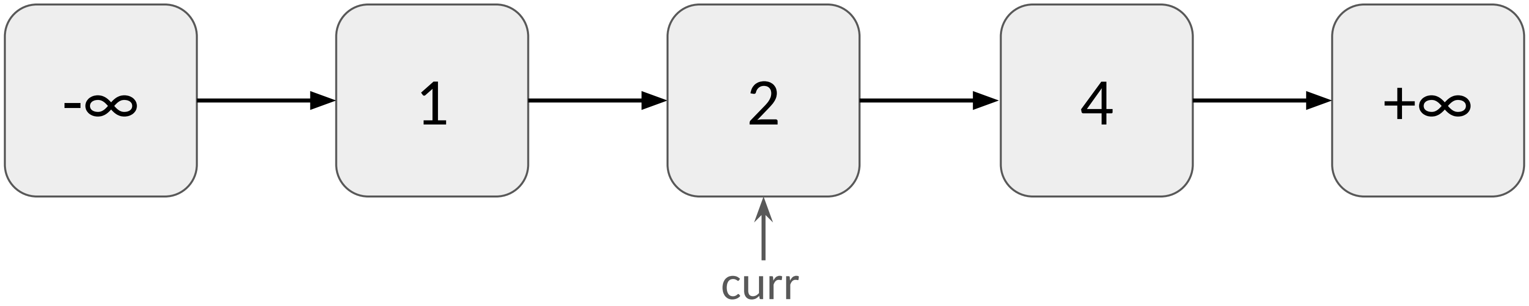
contains(2) || remove(2) ;; add(2)

Thread 1 - contains(2): walk is done



contains(2) || remove(2) ;; add(2)

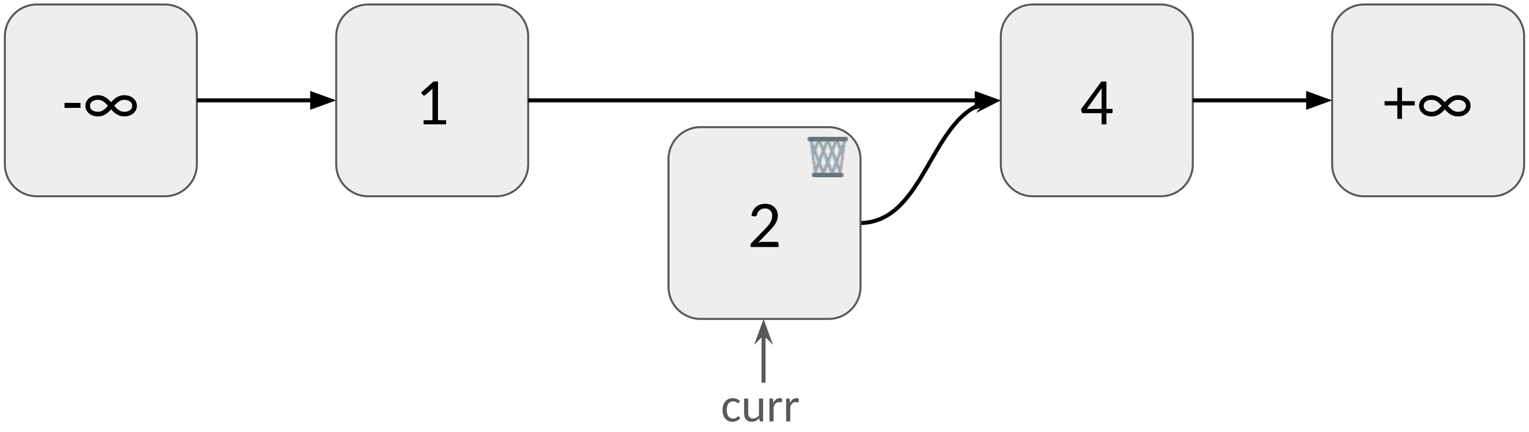
Thread 1 - contains(2): preempted



contains(2) || remove(2) ;; add(2)

Thread 1 - contains(2): preempted

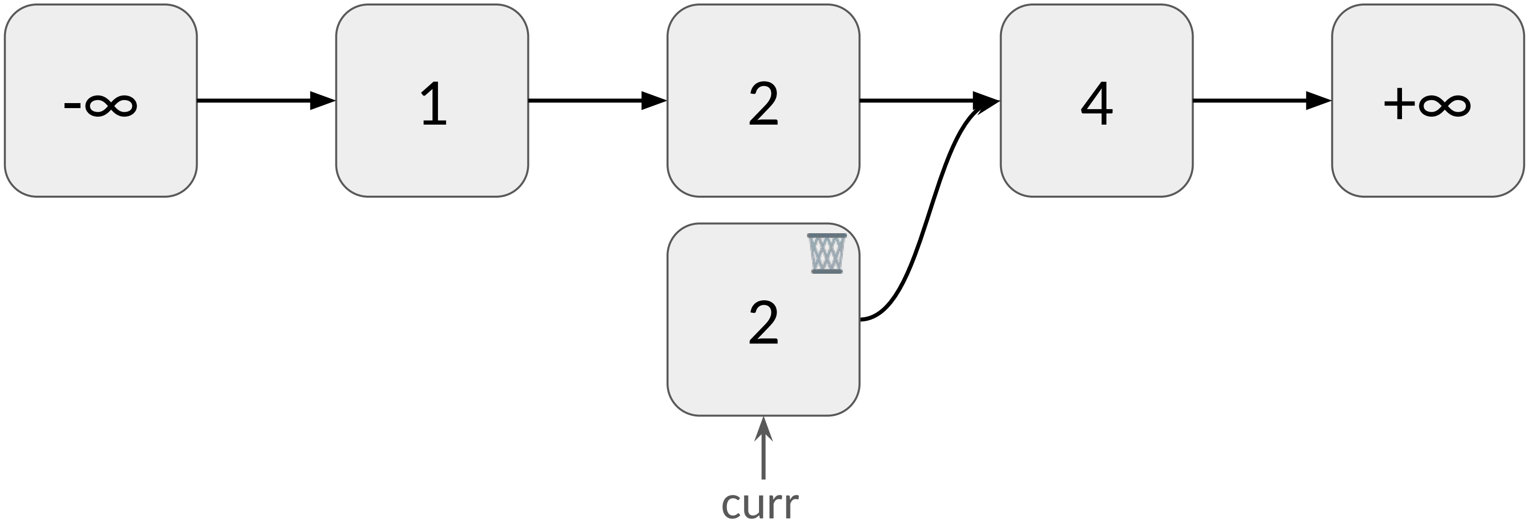
Thread 2 removes 2



contains(2) || remove(2) ;; add(2)

Thread 1 - contains(2): preempted

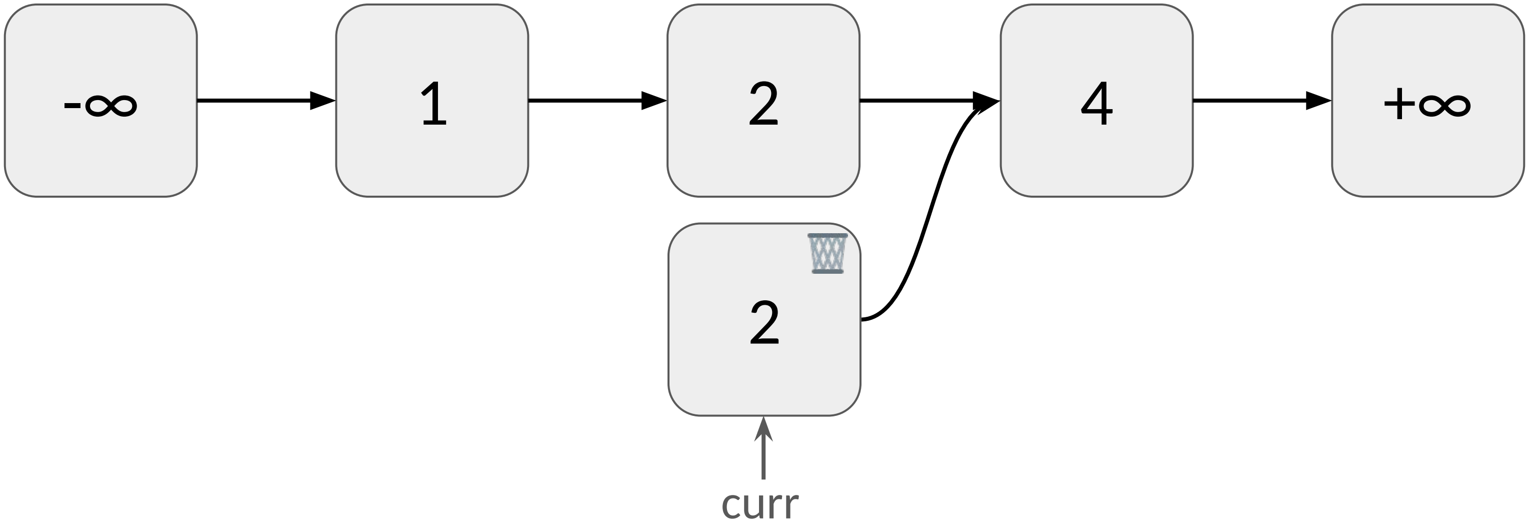
Thread 2 adds 2



contains(2) || remove(2) ;; add(2)

Thread 1 - contains(2): **reads marked is true**

This cannot be the linearization point, as 2 is in the list!



Linearization Point for contains if Element Is Not in the Set

Earlier of the following points:

Point at which a removed matching entry is found	Point immediately before a new matching entry is added to the list

Proving Correctness in Iris

Implement methods in HeapLang

Define specifications in Iris

Prove specifications in Iris

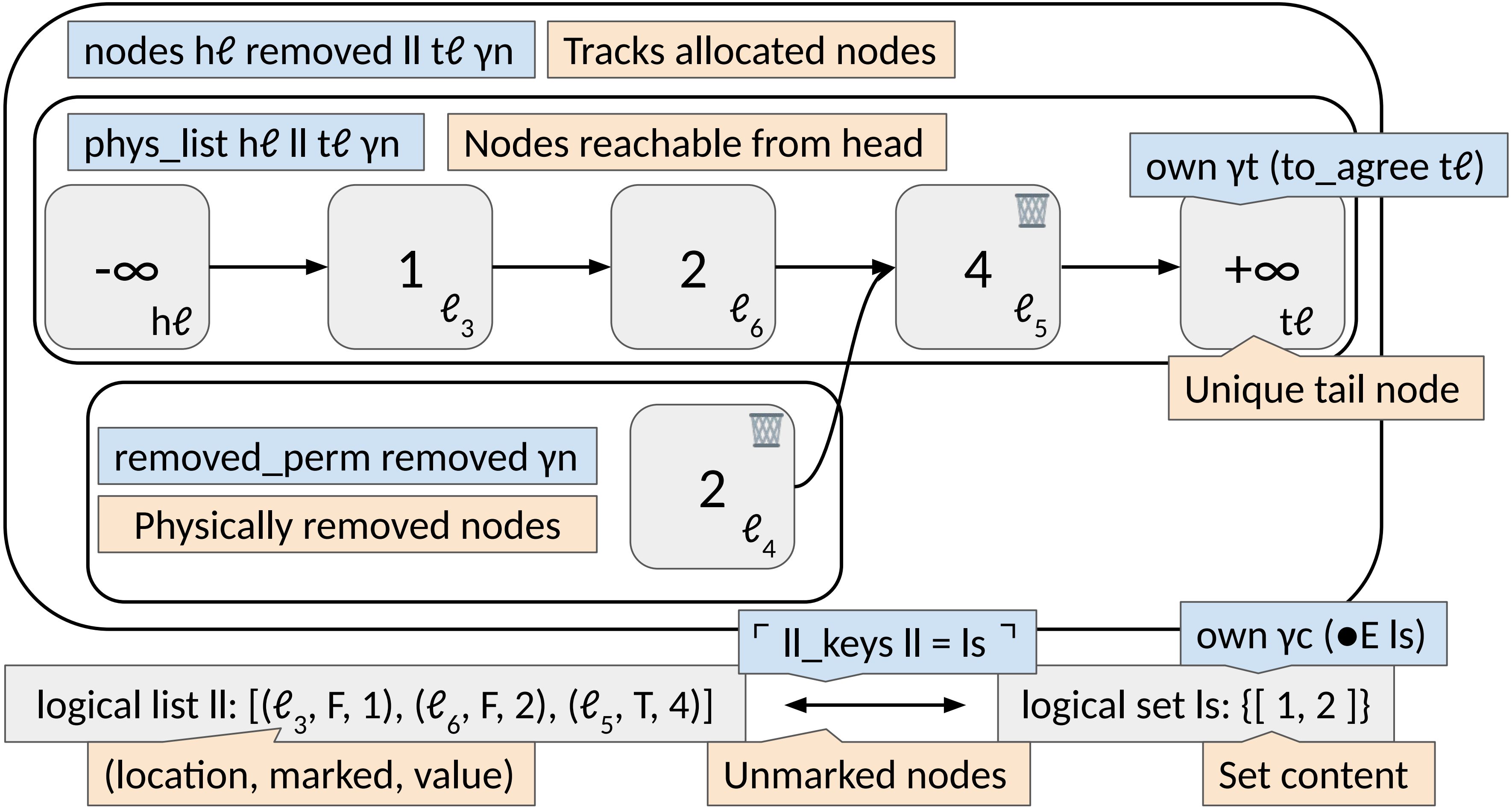
Need to define a set invariant!

Set Invariant

Pointer to set

Ghost State

```
Definition set_inv (hℓ : loc) (γn : gname) (γc : gname) (γt : gname) : iProp :=
  ∃ (removed : gset loc) (ls : gset Z) (ll : llist) (tℓ : loc),
    nodes hℓ removed ll tℓ γn ∗ own γc (●E ls) ∗ own γt (to_agree tℓ) ∗
    ⌜ ll_keys ll = ls ⌝ ∗ phys_list hℓ ll tℓ γn ∗ removed_perm removed γn.
```


nodes hℓ removed ll tℓ γn
Tracks allocated nodes
phys_list hℓ ll tℓ γn
Nodes reachable from head
own γt (to_agree tℓ)
-∞
hℓ
1
ℓ_3
2
ℓ_6
4
ℓ_5
+∞
tℓ
Unique tail node
removed_perm removed γn
Physically removed nodes
2
ℓ_4
⌜ ll_keys ll = ls ⌝
own γc (●E ls)
logical list ll: [(ℓ_3, F, 1), (ℓ_6, F, 2), (ℓ_5, T, 4)]
logical set ls: {[1, 2]}
(location, marked, value)
Unmarked nodes
Set content

Modeling Nodes

ℓ is a node

own γn (◯ {[ℓ]})

implies

read-only pointer

mutable pointer to next node

$\ell \mapsto_{\square}$ SOMEV (extZ#k, $\#\ell\ell_n$, $\#\ell_m$, lk)

mutable pointer to mark

lock

+

ℓ is the head node
$\lor$ ℓ is the tail node
$\lor$ ℓ is reachable from the head
$\lor$ ℓ has been physically removed

Often used in proofs

Specification for add

Key to be added

```
Lemma add_spec (hℓ : loc) (γn : gname) (γc : gname) (γt : gname) (key : Z) :
  is_set hℓ γn γc γt -∗
  <<{ ∀∀ (ls : gset Z), set_content γc ls }>>
    add #hℓ extZ#(Fin key) @ ↑N
  <<{ set_content γc (ls ∪ {[ key ]}) | RET #(bool_decide (key ∉ ls)) }>>.
```

Set Invariant

Logical Atomicity

Key added to set

Whether key was already in the set

Conclusion and Future Work

What We Did

Implemented *all* methods in HeapLang

Defined specifications for *all* methods in Iris

Proved specifications for *add*, and *remove* in Iris

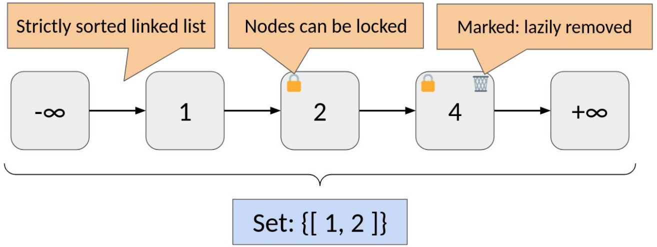
What Could Be Next

Prove *contains* using the helper pattern

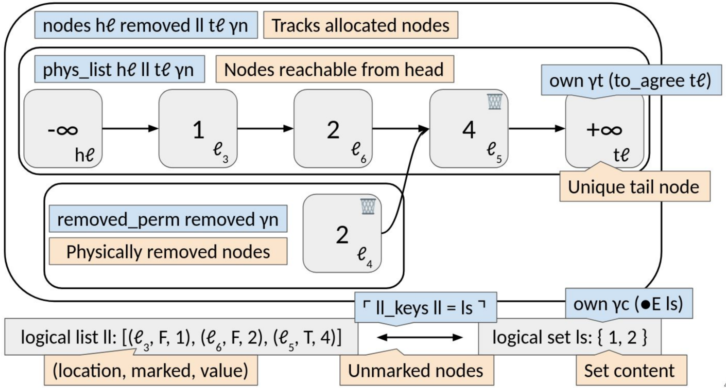
Nested invariants instead of tracking the set of all nodes?

Use more local instead of global information

Anatomy of a Lazy List-Based Set

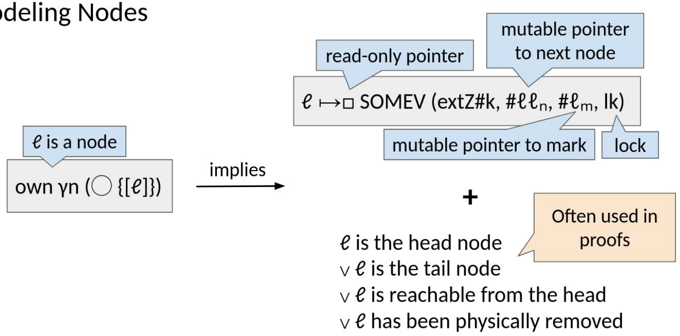


3



42

Modeling Nodes



43

Conclusion and Future Work

What We Did

- Implemented *all* methods in HeapLang
- Defined specifications for *all* methods in Iris
- Proved specifications for *add*, and *remove* in Iris

What Could Be Next

- Prove *contains* using the helper pattern
- Nested invariants instead of tracking the set of all nodes?
- Use more local instead of global information

45