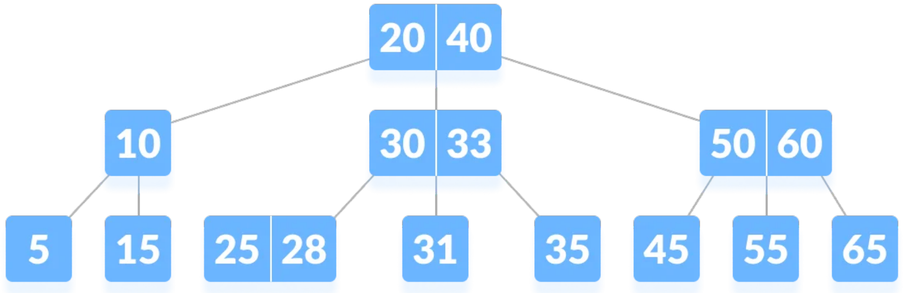
Verification of Concurrent Search Structures

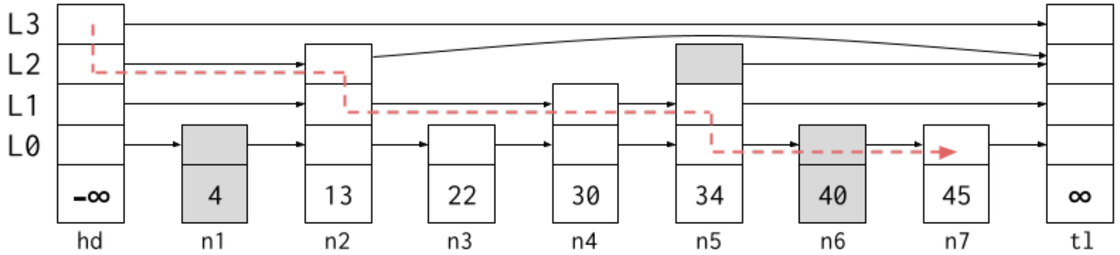
Nisarg Patel

Dissertation Defense
Aug 6, 2024

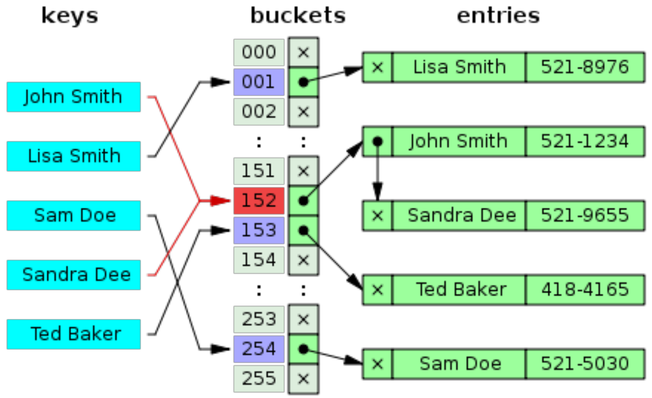
Concurrent Search Structures



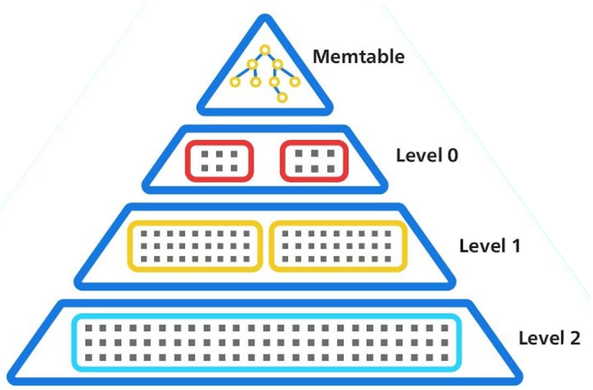
B-trees



Skiplists



Hash-tables



Log-Structured Merge (LSM) Trees

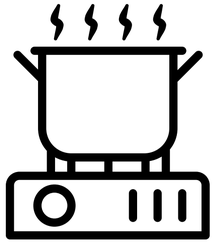
Recipe for modular verification

Step 1:
Find a class of structures with common correctness reasoning

- B-trees, Hash-tables, lock-based linked lists
- LSM Trees, Differential Files
- Lock-free linked lists and skiplists

Step 2:
Develop enabling technology

- Template Algorithms
- Edgeset Framework
- Flow Framework



Step 3:
Formalize the proof

- Resource Algebras
- Supports proof modularity

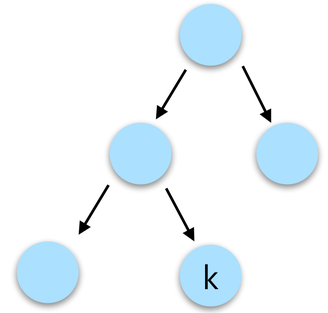
- Siddharth Krishna et al. *Verifying concurrent search structure templates*. [PLDI 2020]

Contributions

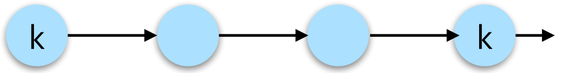
	Single-copy/In-place	Multicopy/Out-of-place
Lock-based	B+-trees B-link trees Hash tables Lock-coupling Lists	LSM Trees Differential Files
Lock-free	Michael's Set Harris List Skiplists	Lock-free LSM Trees BW-trees

Contributions

	Single-copy/In-place	Multicopy/Out-of-place
Lock-based	B+-trees B-link trees Hash tables Lock-coupling Lists	LSM Trees Differential Files
Lock-free	Michael's Set Harris List Skiplists	Lock-free LSM Trees BW-trees



Single-copy



Multicopy

Contributions

	Single-copy/In-place	Multicopy/Out-of-place
Lock-based	B+-trees B-link trees Hash tables Lock-coupling Lists	LSM Trees Differential Files
Lock-free	Michael's Set Harris List Skiplists	Lock-free LSM Trees BW-trees

- Siddharth Krishna et al. *Verifying concurrent search structure templates*. [PLDI 2020]
- Nisarg Patel et al. *Verifying concurrent multicopy search structures*. [OOPSLA 2021]
- Nisarg Patel, Dennis E. Shasha and Thomas Wies. *Verifying lock-free search structure templates*. [ECOOP 2024]

Contributions

	Single-copy/In-place	Multicopy/Out-of-place
Lock-based	B+-trees B-link trees Hash tables Lock-coupling Lists  PLDI20	LSM Trees Differential Files
Lock-free	Michael's Set Harris List Skiplists	Lock-free LSM Trees BW-trees

- Siddharth Krishna et al. *Verifying concurrent search structure templates*. [PLDI 2020]
- Nisarg Patel et al. *Verifying concurrent multicopy search structures*. [OOPSLA 2021]
- Nisarg Patel, Dennis E. Shasha and Thomas Wies. *Verifying lock-free search structure templates*. [ECOOP 2024]

Contributions

	Single-copy/In-place	Multicopy/Out-of-place
Lock-based	B+-trees B-link trees Hash tables Lock-coupling Lists  PLDI20	LSM Trees Differential Files  OOPSLA21
Lock-free	Michael's Set Harris List Skiplists	Lock-free LSM Trees BW-trees

- Siddharth Krishna et al. *Verifying concurrent search structure templates*. [PLDI 2020]
- Nisarg Patel et al. *Verifying concurrent multicopy search structures*. [OOPSLA 2021]
- Nisarg Patel, Dennis E. Shasha and Thomas Wies. *Verifying lock-free search structure templates*. [ECOOP 2024]

Contributions

	Single-copy/In-place	Multicopy/Out-of-place
Lock-based	B+-trees B-link trees Hash tables Lock-coupling Lists  PLDI20	LSM Trees Differential Files  OOPSLA21
Lock-free	Michael's Set Harris List Skiplists  ECOOP24	Lock-free LSM Trees BW-trees

- Siddharth Krishna et al. *Verifying concurrent search structure templates*. [PLDI 2020]
- Nisarg Patel et al. *Verifying concurrent multicopy search structures*. [OOPSLA 2021]
- Nisarg Patel, Dennis E. Shasha and Thomas Wies. *Verifying lock-free search structure templates*. [ECOOP 2024]

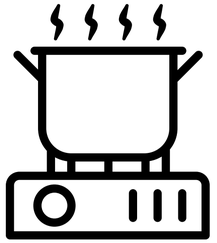
Outline

Step 1:
Find a class of structures with common correctness reasoning

- ECOOP24 : (Lock-free, single-copy) linked lists and skiplists
- OOPSLA21 : (Lock-based, multicopy) Log-Structured Merge (LSM) Trees

Step 2:
Develop enabling technology

- Template Algorithms
- Hindsight Framework
- Keyset Reasoning



Step 3:
Formalize the proof

- Evaluation

- Nisarg Patel, Dennis E. Shasha and Thomas Wies. *Verifying lock-free search structure templates*. [ECOOP 2024]
- Nisarg Patel et al. *Verifying concurrent multicopy search structures*. [OOPSLA 2021]

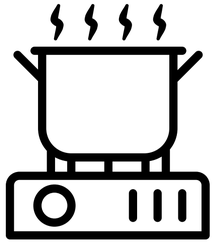
Outline

Step 1:
Find a class of structures with common correctness reasoning

- ECOOP24 : (Lock-free, single-copy) linked lists and skiplists
- OOPSLA21 : (Lock-based, multicopy) Log-Structured Merge (LSM) Trees

Step 2:
Develop enabling technology

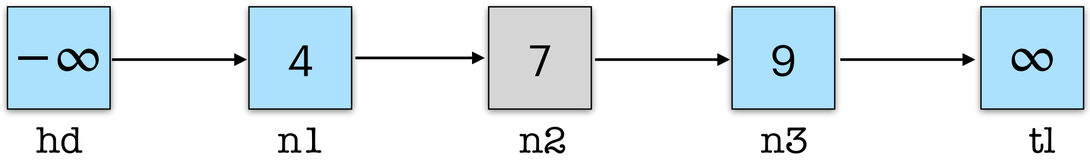
- Template Algorithms
- Hindsight Framework
- Keyset Reasoning



Step 3:
Formalize the proof

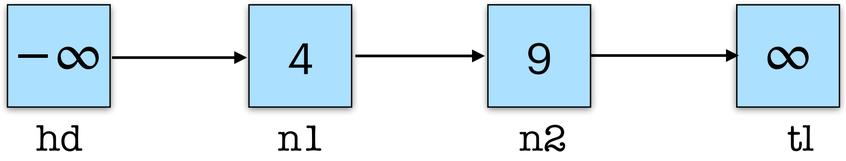
- Evaluation

Michael's Set



Michael's Set

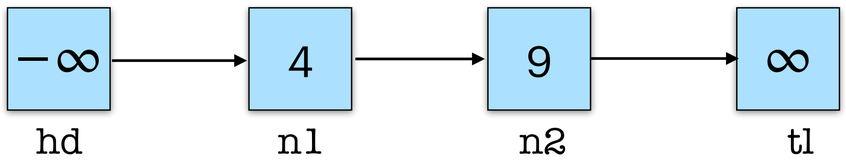
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```



Michael's Set

```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

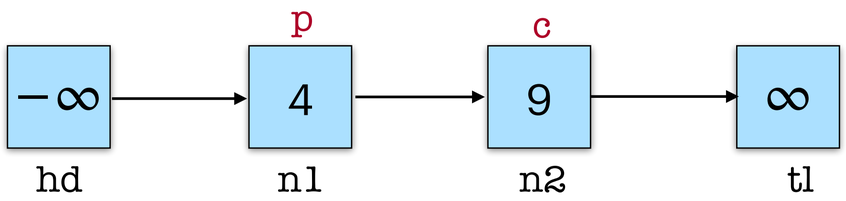
insert(7)



Michael's Set

```
insert(k) =
  p, c, res = find(k);
  if res then false
  else
    n = new_node(k, c);
    if CAS(p.next, (c, 0), (n, 0))
    then true else insert(k)
```

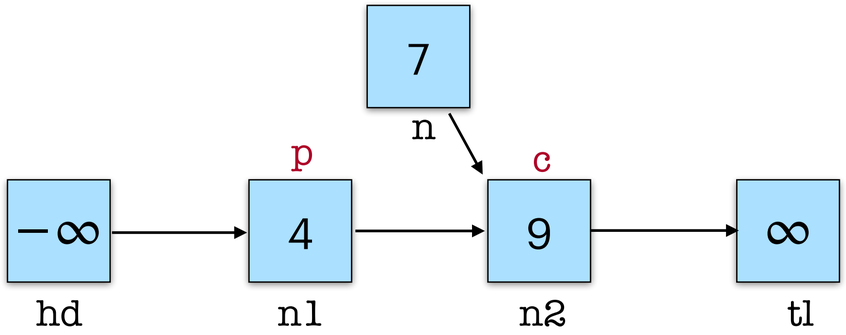
insert(7)



Michael's Set

```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
    ➡ n = new_node(k, c);
      if CAS(p.next, (c, 0), (n, 0))
      then true else insert(k)
```

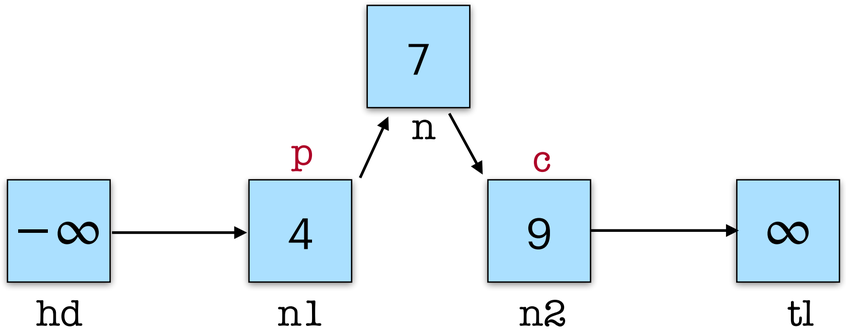
insert(7)



Michael's Set

```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
    ➔ if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

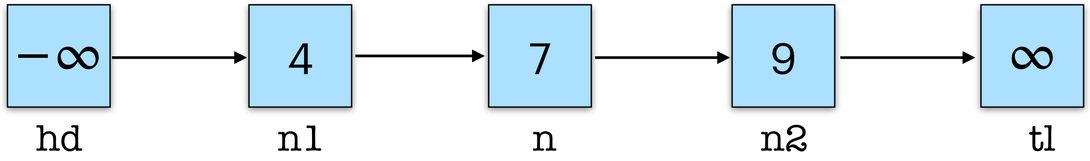
insert(7)



Michael's Set

```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

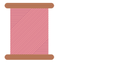


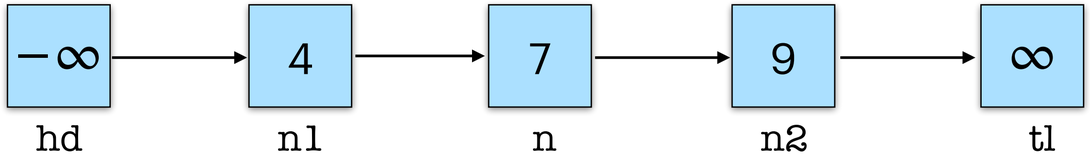
Michael's Set

```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

delete(7)



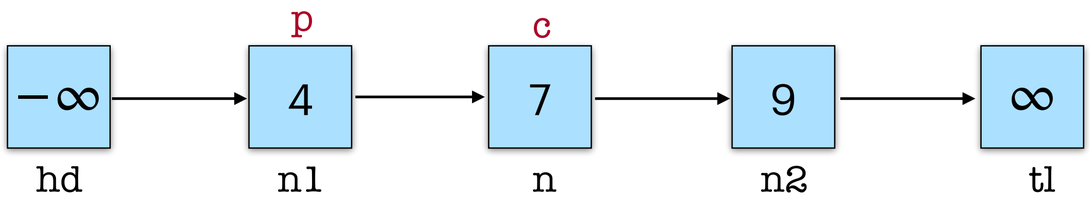


Michael's Set

```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

delete(7)

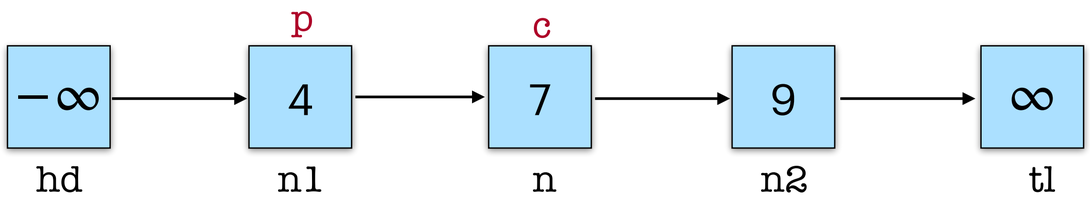


Michael's Set

```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
    ➔ if MARK(c)
        then true else false
```

delete(7)

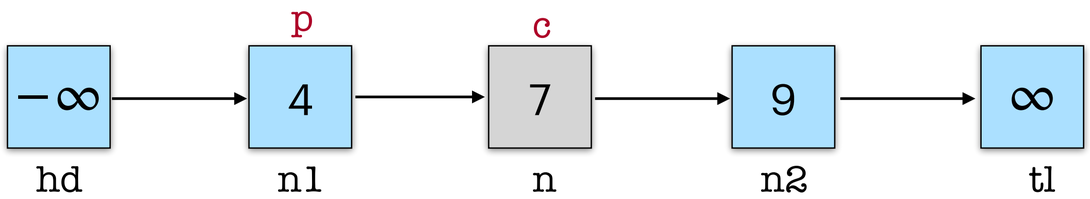


Michael's Set

```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
    ➔ if MARK(c)
        then true else false
```

delete(7)



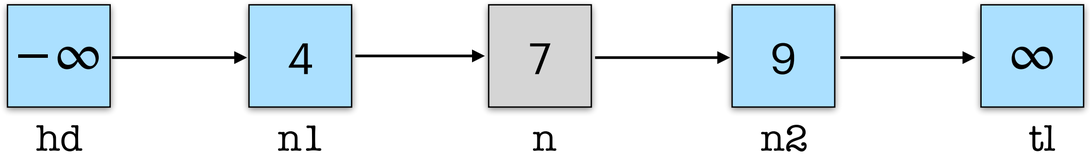
Michael's Set

```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```



Michael's Set

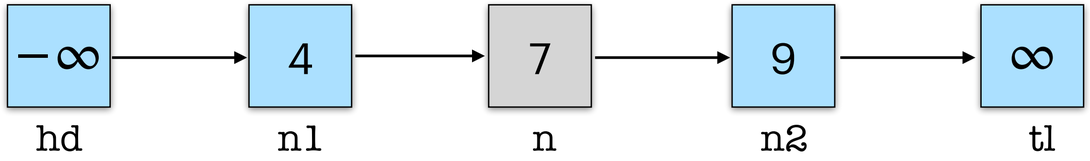
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

search(9)



Michael's Set

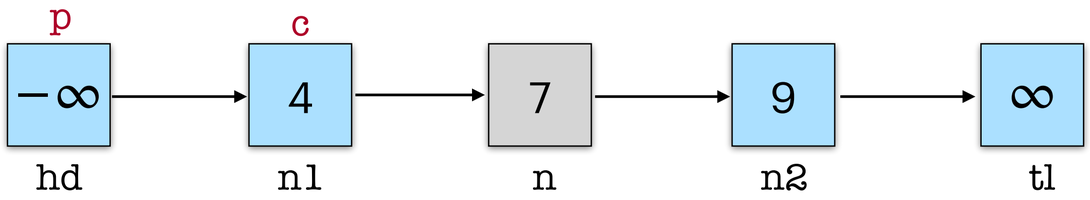
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
➔   (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

search(9)



Michael's Set

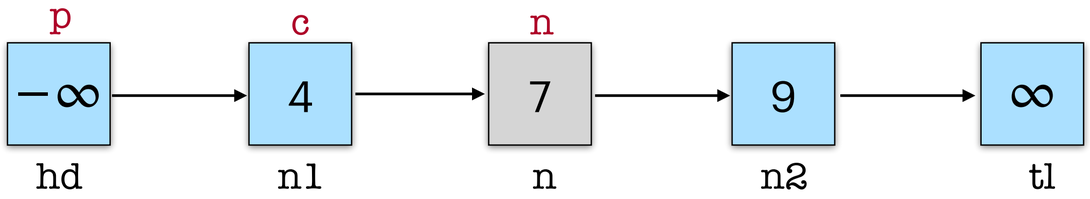
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
➔   (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

search(9)



Michael's Set

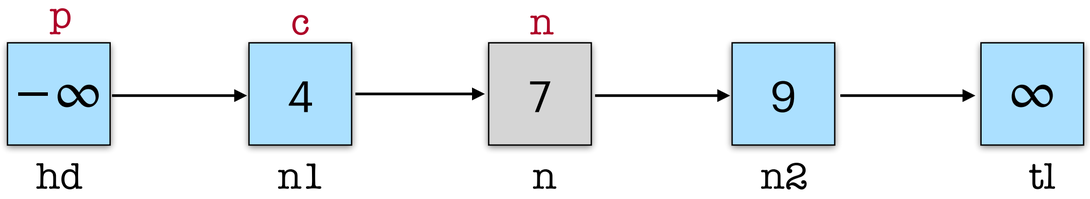
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
    ➜ if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

search(9)



Michael's Set

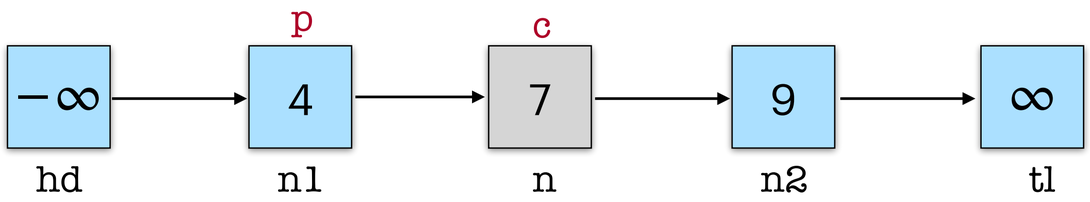
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
    ➜ if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

search(9)



Michael's Set

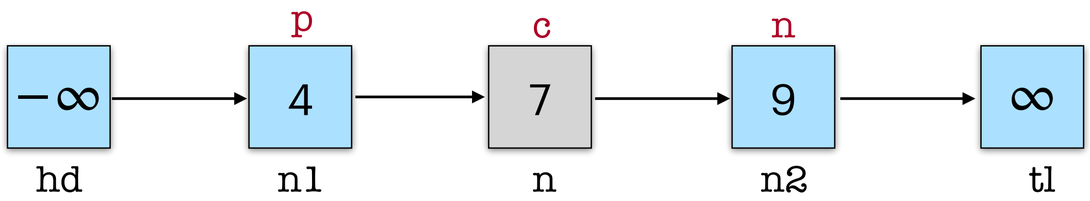
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
➔   (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

search(9)



Michael's Set

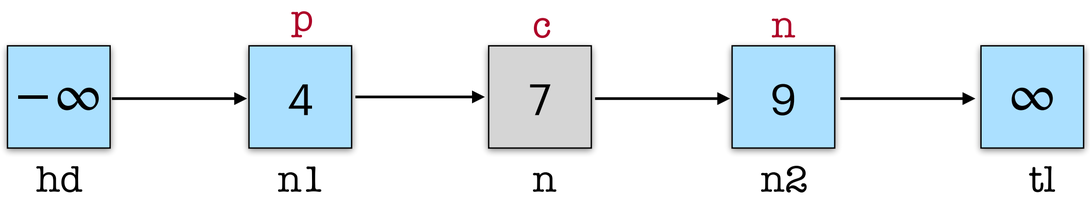
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
    ➜ if CAS(p.next, (c,0), (n,0))
      then traverse(k, p, n) else find(k)
    else
      if c.key < k then traverse(k, c, n)
      else
        res = c.key == k;
        (p, c, res)
```

search(9)



Michael's Set

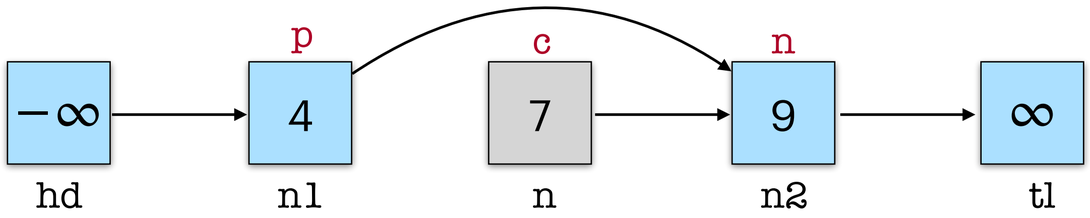
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
    ➜ if CAS(p.next, (c,0), (n,0))
      then traverse(k, p, n) else find(k)
    else
      if c.key < k then traverse(k, c, n)
      else
        res = c.key == k;
        (p, c, res)
```

search(9)



Michael's Set

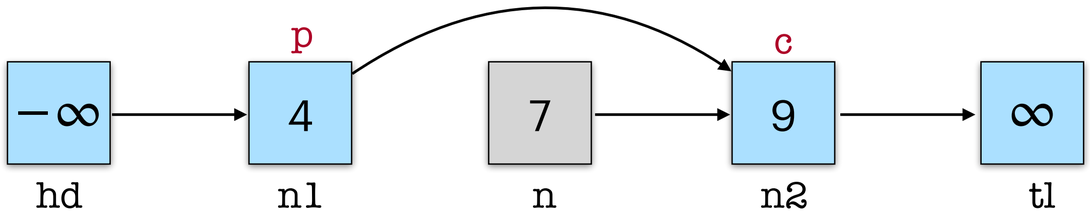
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

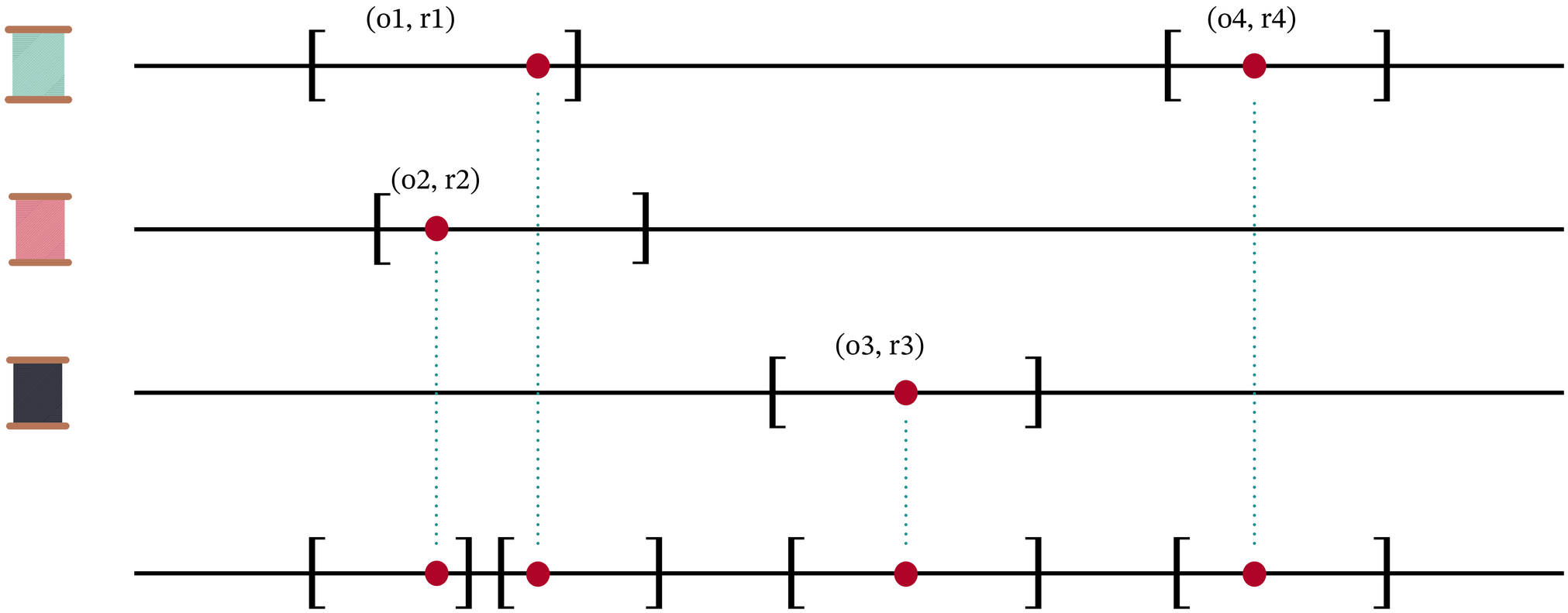
```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
        ➡ res = c.key == k;
          (p, c, res)
```

search(9)



Linearizability

"for each concurrent execution, there exists an equivalent order-preserving sequential execution"



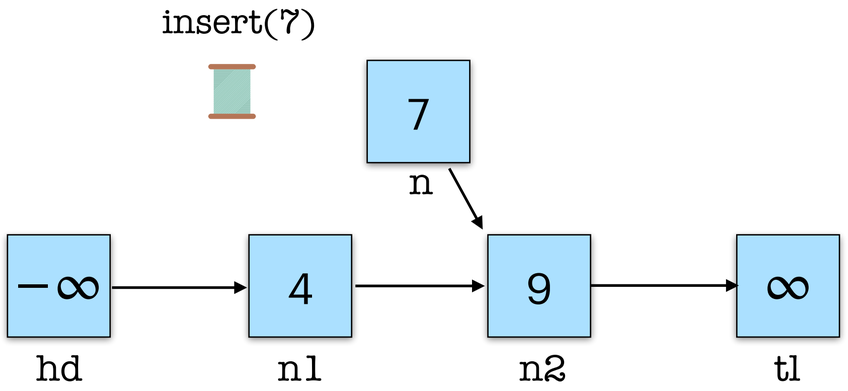
Linearization Points

```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```



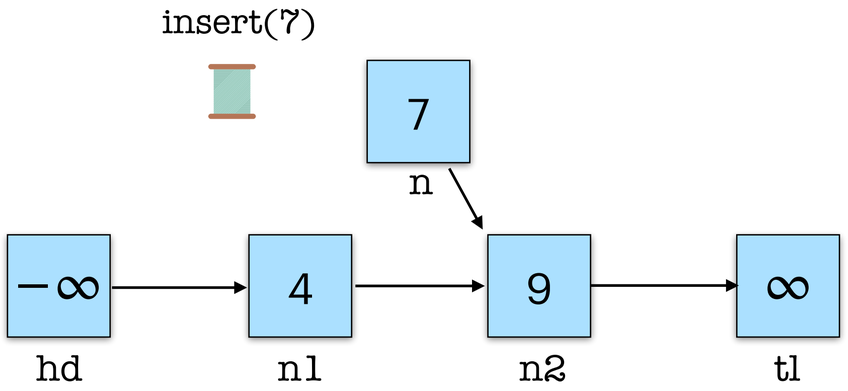
Linearization Points

```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```



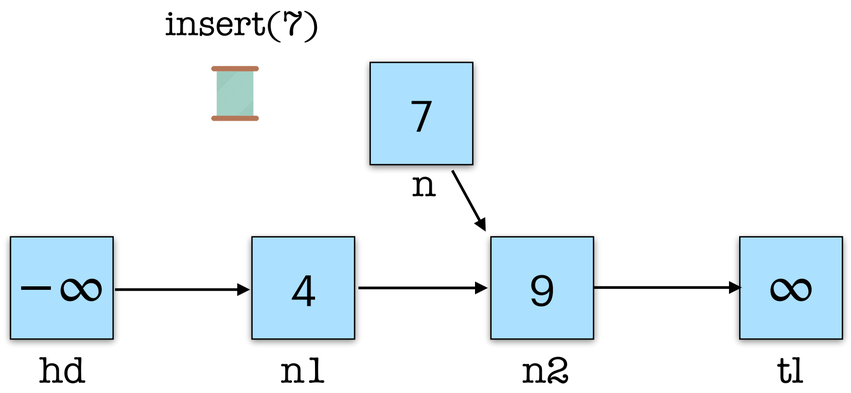
Linearization Points

```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
    ➔   if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```



Modifying Linearization Points

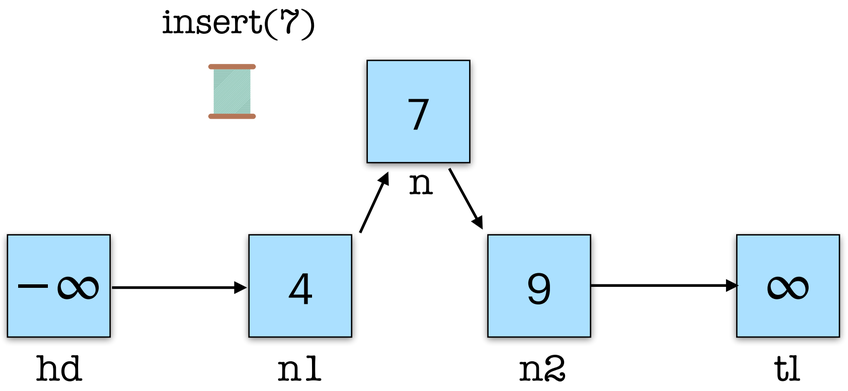
Linearization Points

```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```



Modifying Linearization Points

Linearization Points

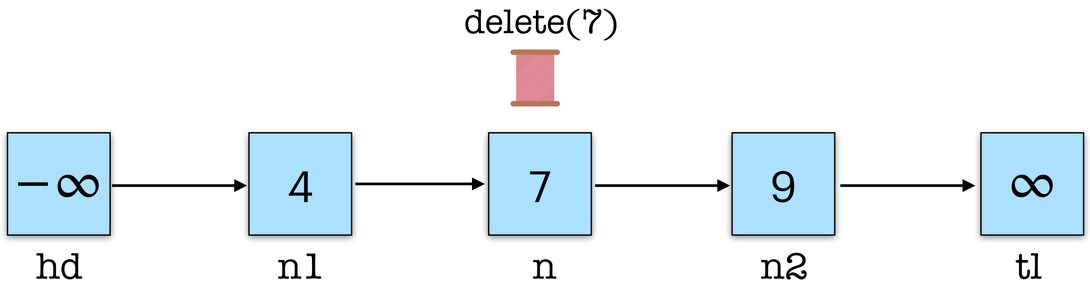
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

Modifying Linearization Points



Linearization Points

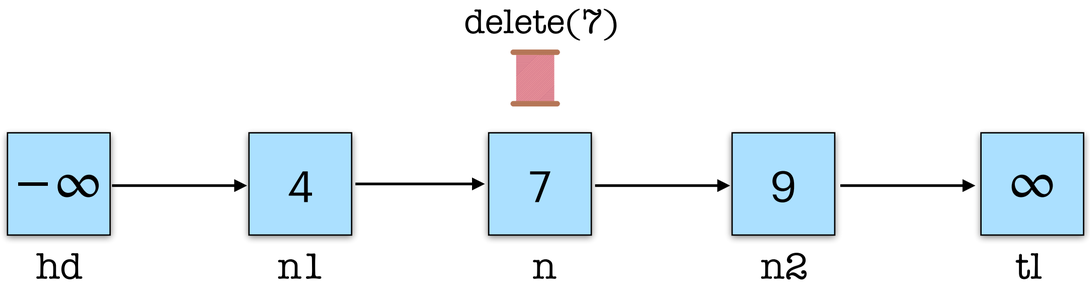
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
    ➜ if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

Modifying Linearization Points



Linearization Points

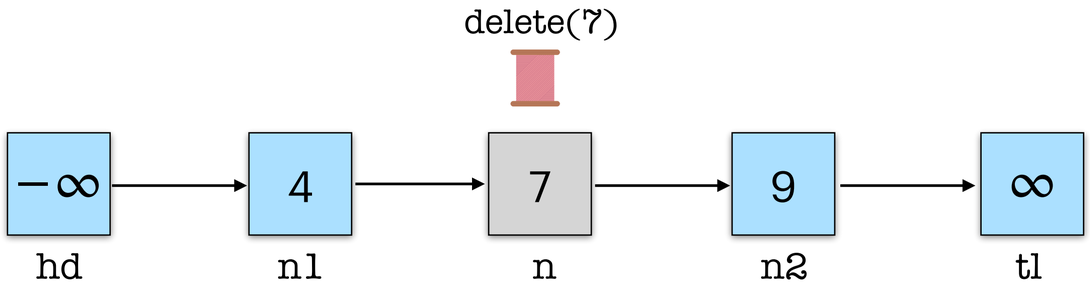
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
    ➜ if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

Modifying Linearization Points



Linearization Points

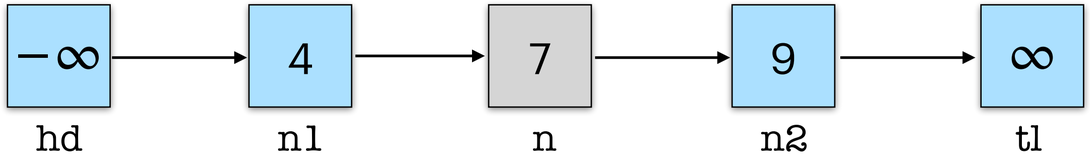
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

Unmodifying Linearization Points?



Linearization Points

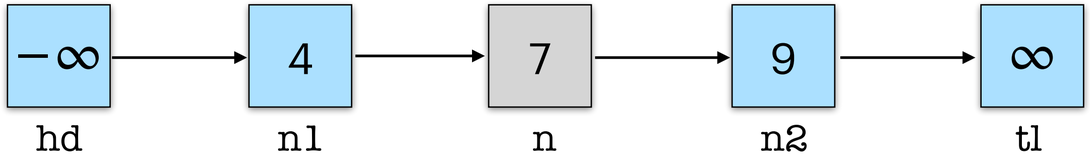
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

Unmodifying Linearization Points?
Future-dependent, external!



Linearization Points

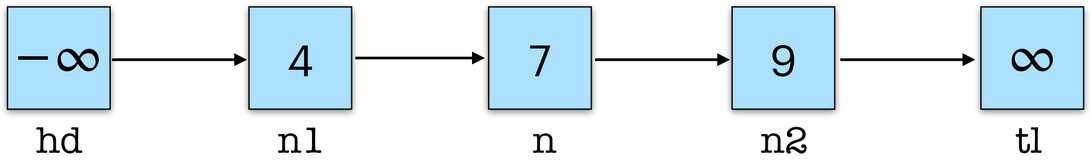
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

search(7)



Linearization Points

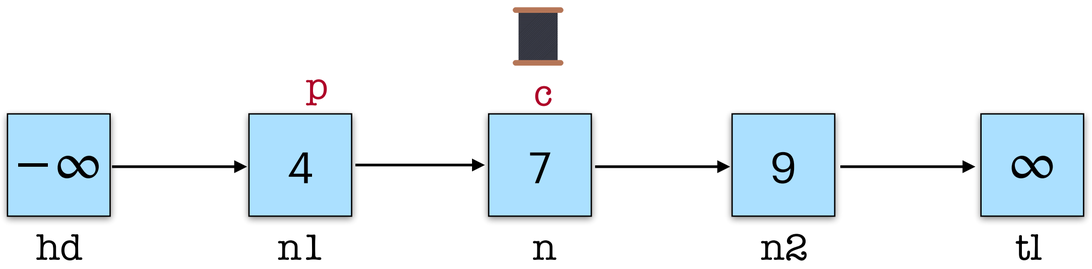
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

search(7)



Linearization Points

```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

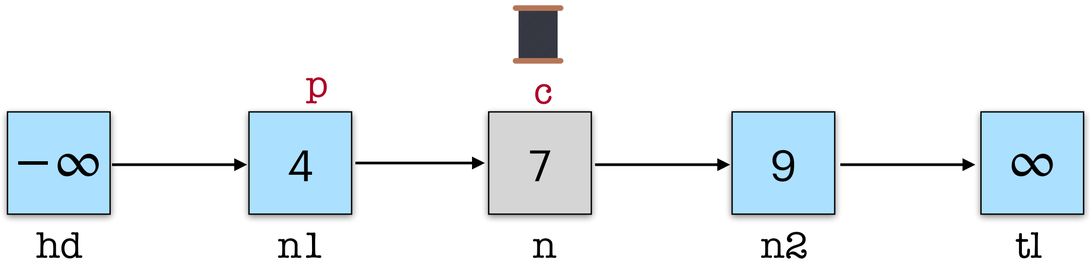
```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

search(7)

delete(7)



Linearization Points

```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

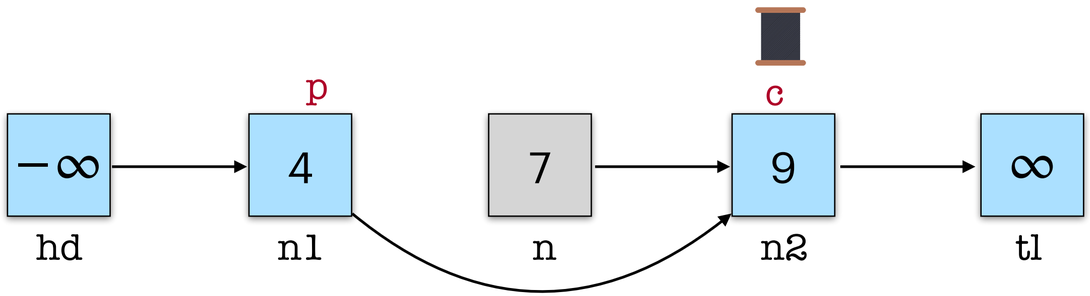
```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

search(7)

delete(7)



Linearization Points

```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

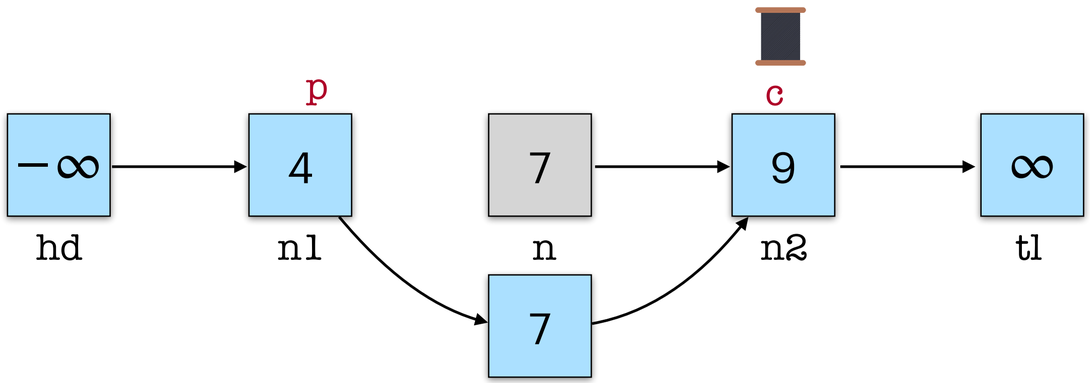
```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

search(7)

delete(7)

insert(7)



Intuitive Proof

```
find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)

traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

- find(k) returns true → at some point, k was in the structure.
- find(k) returns false → at some point, k was not in the structure.

Intuitive Proof

```
find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)

traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

- find(k) returns true → at some point, k was in the structure.
- find(k) returns false → at some point, k was not in the structure.

traversal invariant := key(p) < k && next(p) = c && mark(p) = false

Intuitive Proof

```
find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)

traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

- find(k) returns true → at some point, k was in the structure.
- find(k) returns false → at some point, k was not in the structure.

traversal invariant := key(p) < k && next(p) = c && mark(p) = false ❌

Intuitive Proof

```
find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)

traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

- find(k) returns true → at some point, k was in the structure.
- find(k) returns false → at some point, k was not in the structure.

traversal invariant := *at some point*, key(p) < k && next(p) = c && mark(p) = false

Intuitive Proof

```
find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)

traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

- find(k) returns true → at some point, k was in the structure.
- find(k) returns false → at some point, k was not in the structure.

traversal invariant := *at some point*, key(p) < k && next(p) = c && mark(p) = false

Intuitive Proof

```
find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)

traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

- find(k) returns true → at some point, k was in the structure.
- find(k) returns false → at some point, k was not in the structure.

traversal invariant := *at some point*, key(p) < k && next(p) = c && mark(p) = false

Hindsight Reasoning

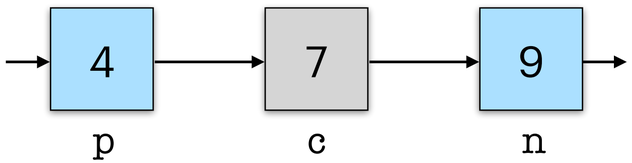
- Peter W. O'Hearn et al. *Verifying linearizability with hindsight*. [PODC 2010]
- Yotam M. Y. Feldman et al. *Order out of chaos: Proving linearizability using local views*. [DISC 2018]
- Yotam M. Y. Feldman et al. *Proving highly-concurrent traversals correct*. [OOPSLA 2020]
- Roland Meyer, Thomas Wies and Sebastien Wolff. *A concurrent program logic with a future and history*. [OOPSLA 2022]
- Roland Meyer, Thomas Wies and Sebastien Wolff. *Embedding hindsight reasoning in separation logic*. [PLDI 2023]

Intuitive Proof

```
find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)

traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
    ➡ if CAS(p.next, (c,0), (n,0))
      then traverse(k, p, n) else find(k)
    else
      if c.key < k then traverse(k, c, n)
      else
        res = c.key == k;
        (p, c, res)
```

- find(k) returns true → at some point, k was in the structure.
- find(k) returns false → at some point, k was not in the structure.

traversal invariant := *at some point*, key(p) < k && next(p) = c && mark(p) = false



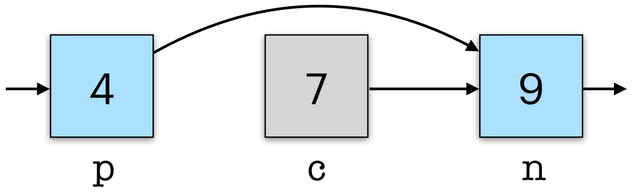
search(7)

Intuitive Proof

```
find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)

traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
    ➡ if CAS(p.next, (c,0), (n,0))
      then traverse(k, p, n) else find(k)
    else
      if c.key < k then traverse(k, c, n)
      else
        res = c.key == k;
        (p, c, res)
```

- find(k) returns true → at some point, k was in the structure.
- find(k) returns false → at some point, k was not in the structure.

traversal invariant := *at some point*, key(p) < k && next(p) = c && mark(p) = false



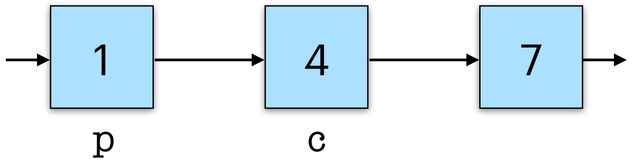
search(7)

Intuitive Proof

```
find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)

traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
    ➡ if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

- find(k) returns true → at some point, k was in the structure.
- find(k) returns false → at some point, k was not in the structure.

traversal invariant := *at some point*, key(p) < k && next(p) = c && mark(p) = false



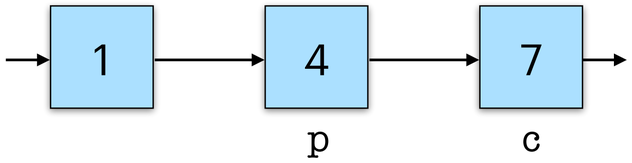
search(7)

Intuitive Proof

```
find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)

traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
    ➡ if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

- find(k) returns true → at some point, k was in the structure.
- find(k) returns false → at some point, k was not in the structure.

traversal invariant := *at some point*, key(p) < k && next(p) = c && mark(p) = false



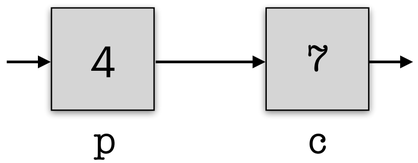
search(7)

Intuitive Proof

```
find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)

traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
        ➡ res = c.key == k;
          (p, c, res)
```

- find(k) returns true → at some point, k was in the structure.
- find(k) returns false → at some point, k was not in the structure.

traversal invariant := *at some point*, key(p) < k && next(p) = c && mark(p) = false



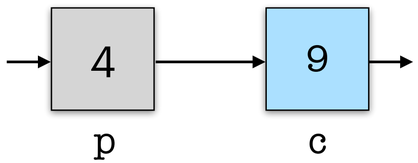
search(7)

Intuitive Proof

```
find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)

traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
        ➡ res = c.key == k;
          (p, c, res)
```

- find(k) returns true → at some point, k was in the structure.
- find(k) returns false → at some point, k was not in the structure.

traversal invariant := *at some point*, key(p) < k && next(p) = c && mark(p) = false



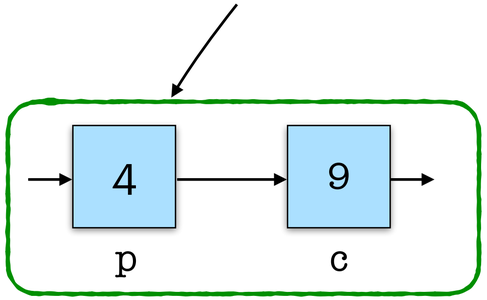
search(7)

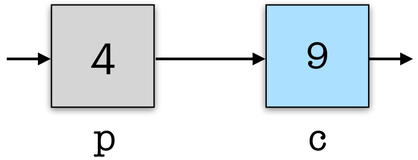
Intuitive Proof

```
find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)

traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
        ➡ res = c.key == k;
            (p, c, res)
```

- find(k) returns true → at some point, k was in the structure.
- find(k) returns false → at some point, k was not in the structure.

traversal invariant := *at some point*, key(p) < k && next(p) = c && mark(p) = false





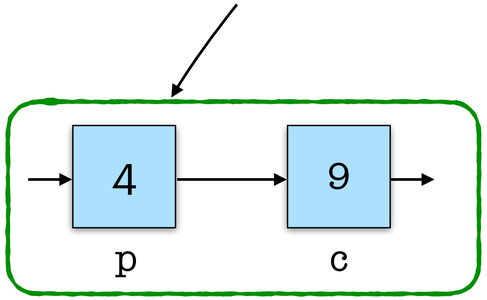
search(7)

Intuitive Proof

```
find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)

traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
        ➡ res = c.key == k;
          (p, c, res)
```

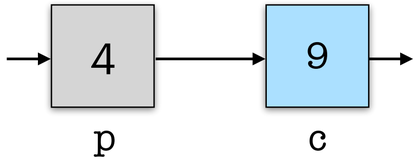
- find(k) returns true → at some point, k was in the structure.
- find(k) returns false → at some point, k was not in the structure.

traversal invariant := *at some point*, key(p) < k && next(p) = c && mark(p) = false



1. A node once marked remains marked.
2. The key of a node key never changes.
3. hd-list is sorted.

....



search(7)

Michael's Set

```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

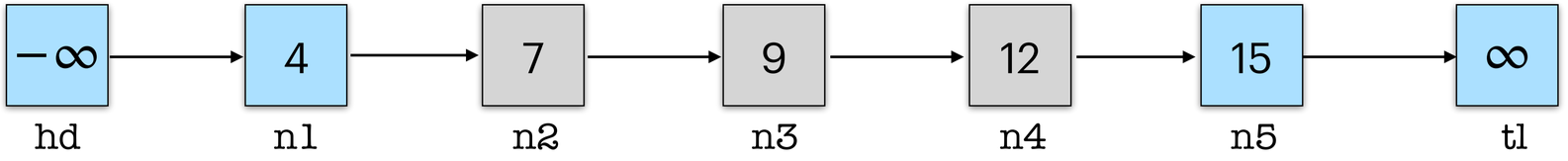
Harris List

```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, pn, c) =
    (n, b) = c.next;
    if b == 0 then
        traverse(k, p, pn, n)
    else
        if CAS(p.next, (pn, 0), (c, 0)) then
            if c.key < k then traverse(c, n, n)
            else
                res = c.key == k;
                (p, c, res)
        else find(k)
```



Harris List

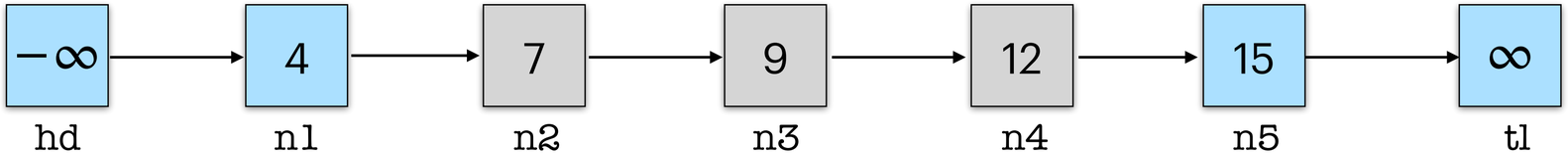
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, pn, c) =
    (n, b) = c.next;
    if b == 0 then
        traverse(k, p, pn, n)
    else
        if CAS(p.next, (pn, 0), (c, 0)) then
            if c.key < k then traverse(c, n, n)
            else
                res = c.key == k;
                (p, c, res)
        else find(k)
```

search(15)



Harris List

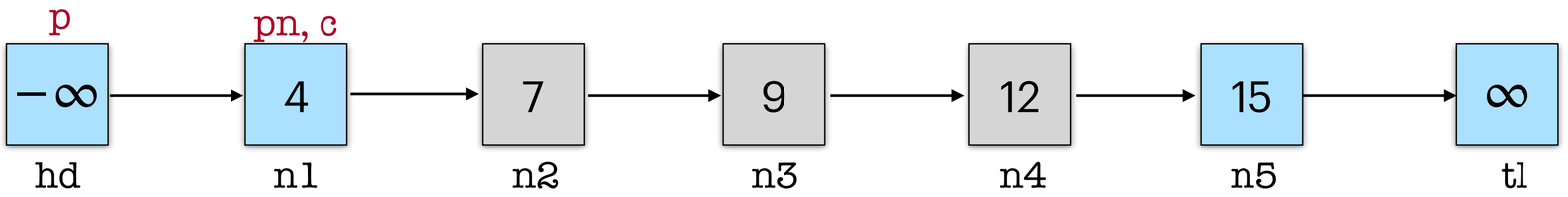
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, pn, c) =
    (n, b) = c.next;
    if b == 0 then
        traverse(k, p, pn, n)
    else
        if CAS(p.next, (pn, 0), (c, 0)) then
            if c.key < k then traverse(c, n, n)
            else
                res = c.key == k;
                (p, c, res)
        else find(k)
```

search(15)



Harris List

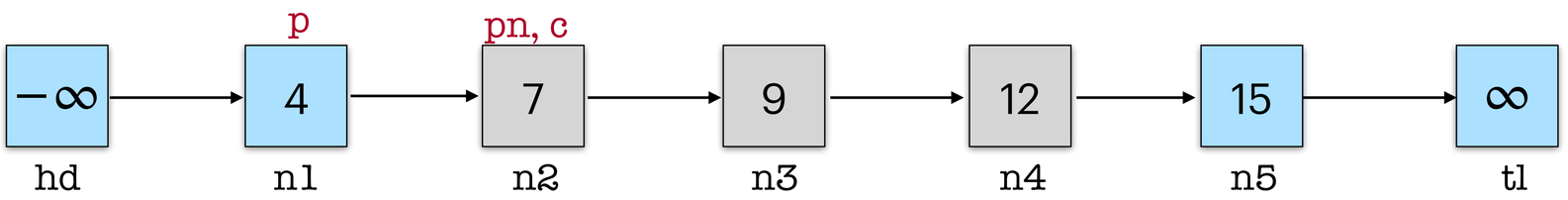
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, pn, c) =
    (n, b) = c.next;
    if b == 0 then
        traverse(k, p, pn, n)
    else
        if CAS(p.next, (pn, 0), (c, 0)) then
            if c.key < k then traverse(c, n, n)
            else
                res = c.key == k;
                (p, c, res)
        else find(k)
```

search(15)



Harris List

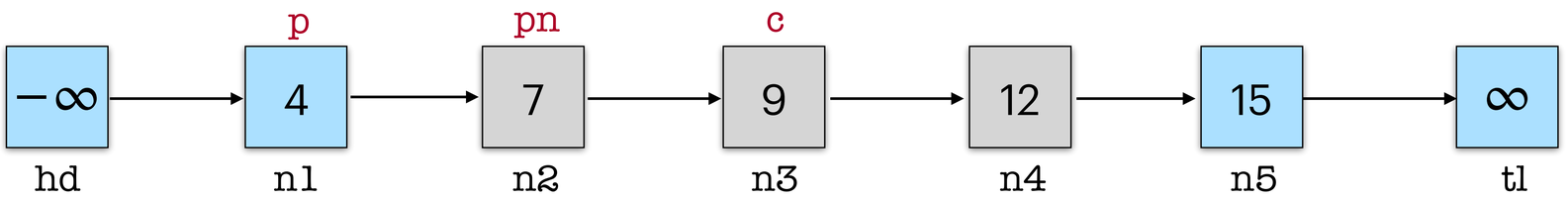
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, pn, c) =
    (n, b) = c.next;
    if b == 0 then
        traverse(k, p, pn, n)
    else
        if CAS(p.next, (pn, 0), (c, 0)) then
            if c.key < k then traverse(c, n, n)
            else
                res = c.key == k;
                (p, c, res)
        else find(k)
```

search(15)



Harris List

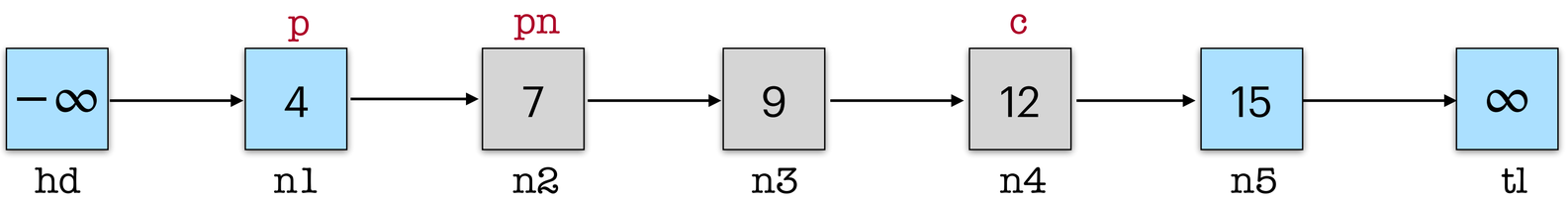
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, pn, c) =
    (n, b) = c.next;
    if b == 0 then
        traverse(k, p, pn, n)
    else
        if CAS(p.next, (pn, 0), (c, 0)) then
            if c.key < k then traverse(c, n, n)
            else
                res = c.key == k;
                (p, c, res)
        else find(k)
```

search(15)



Harris List

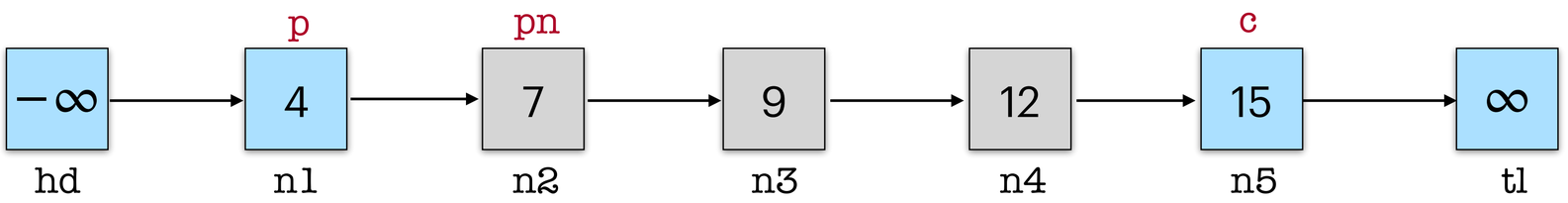
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

```
traverse(k, p, pn, c) =
    (n, b) = c.next;
    if b == 0 then
        traverse(k, p, pn, n)
    else
        if CAS(p.next, (pn, 0), (c, 0)) then
            if c.key < k then traverse(c, n, n)
            else
                res = c.key == k;
                (p, c, res)
        else find(k)
```

search(15)



Harris List

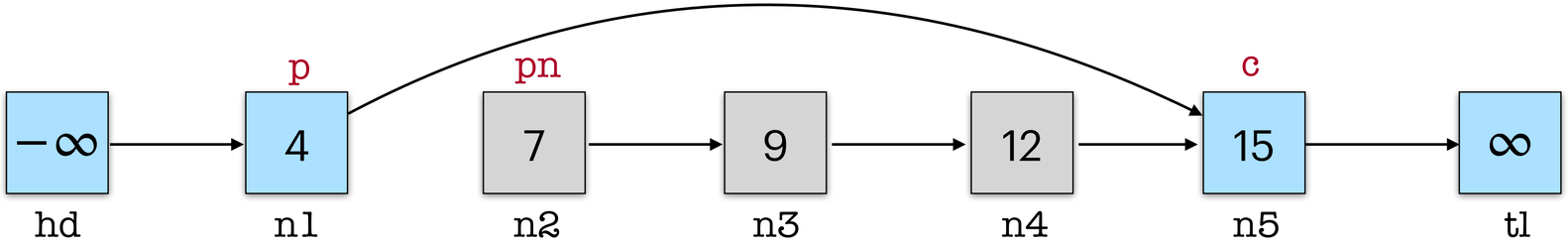
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

```
search(k) =
    _, _, res = find(k);
    res

find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)
```

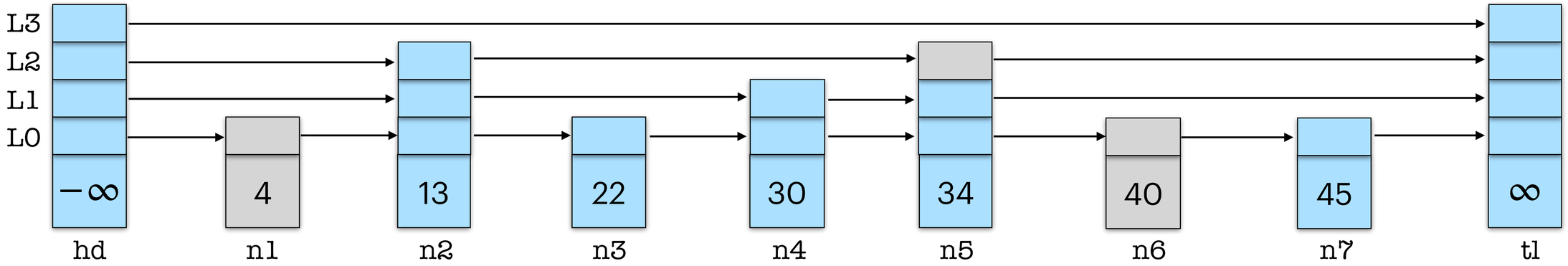
```
traverse(k, p, pn, c) =
    (n, b) = c.next;
    if b == 0 then
        traverse(k, p, pn, n)
    else
        if CAS(p.next, (pn, 0), (c, 0)) then
            if c.key < k then traverse(c, n, n)
            else
                res = c.key == k;
                (p, c, res)
        else find(k)
```

search(15)



Skiplists

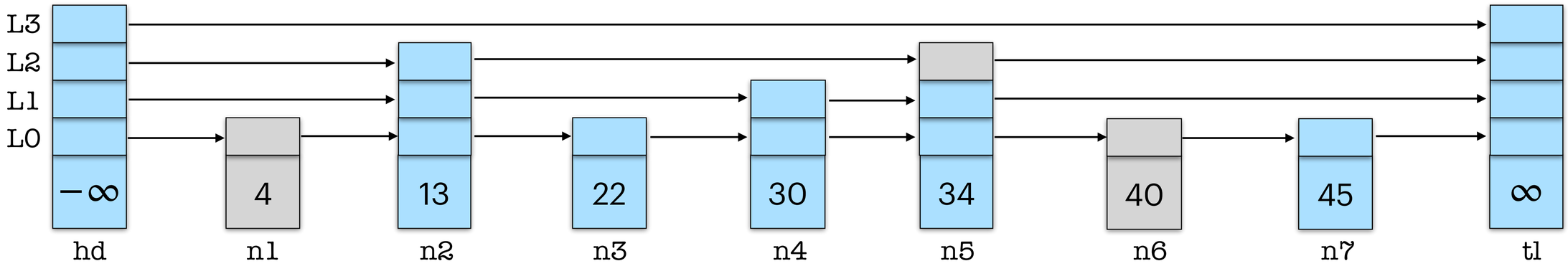
Michael's Set + Levels ≈ Herlihy-Shavit Skiplist



Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

search(42)

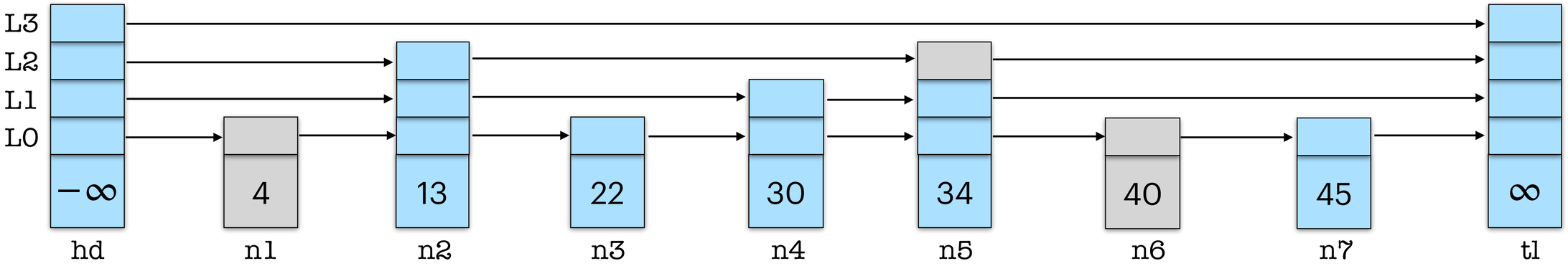


Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

search(42)

	pred	succ
L3		
L2		
L1		
L0		

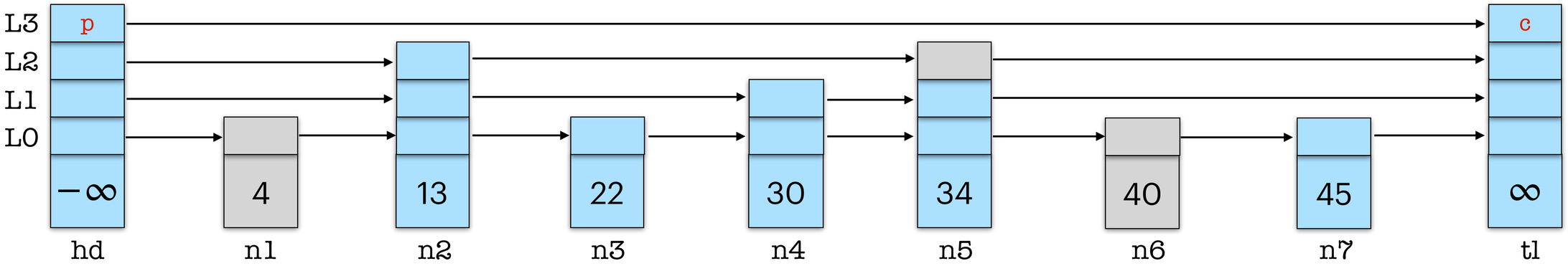


Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

search(42)

	pred	succ
L3	hd	tl
L2		
L1		
L0		

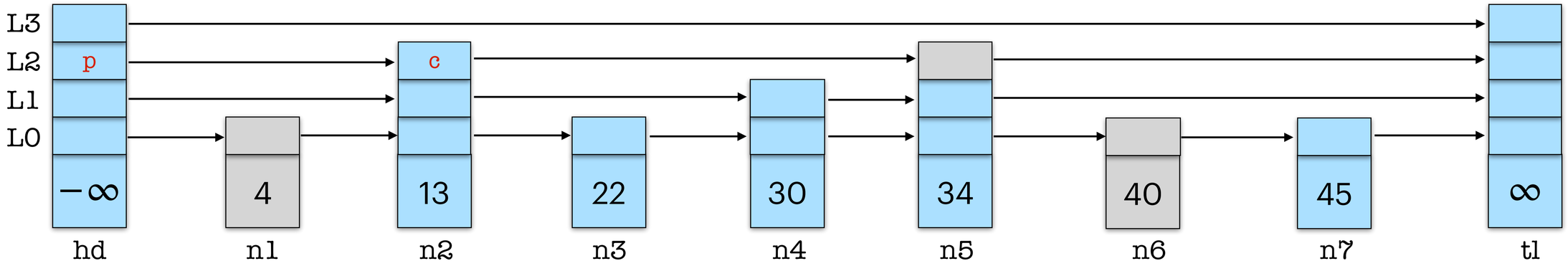


Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

search(42)

	pred	succ
L3	hd	tl
L2		
L1		
L0		

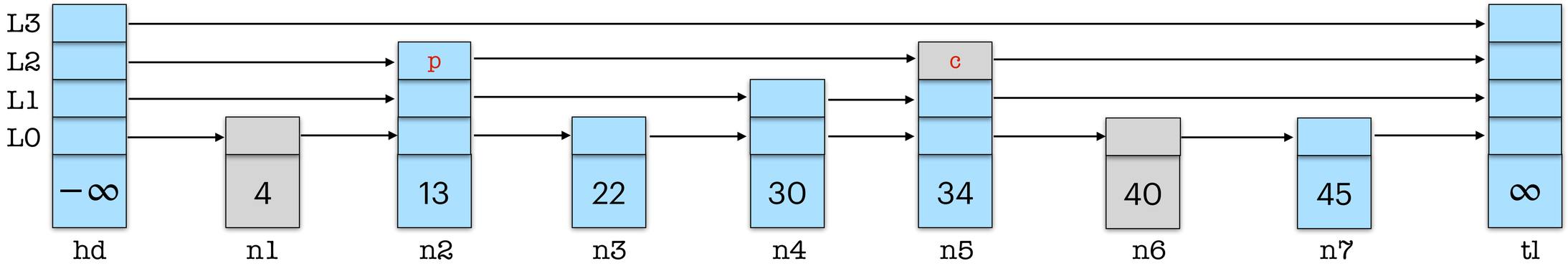


Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

search(42)

	pred	succ
L3	hd	tl
L2		
L1		
L0		

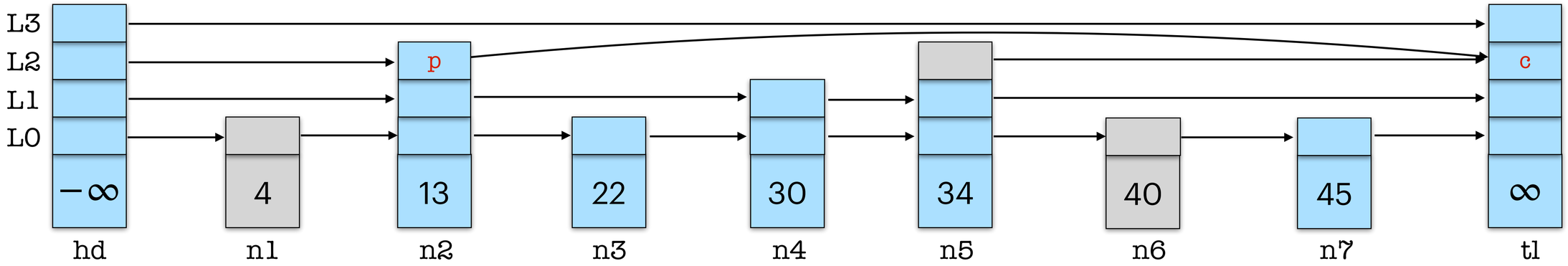


Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

search(42)

	pred	succ
L3	hd	tl
L2	n2	tl
L1		
L0		

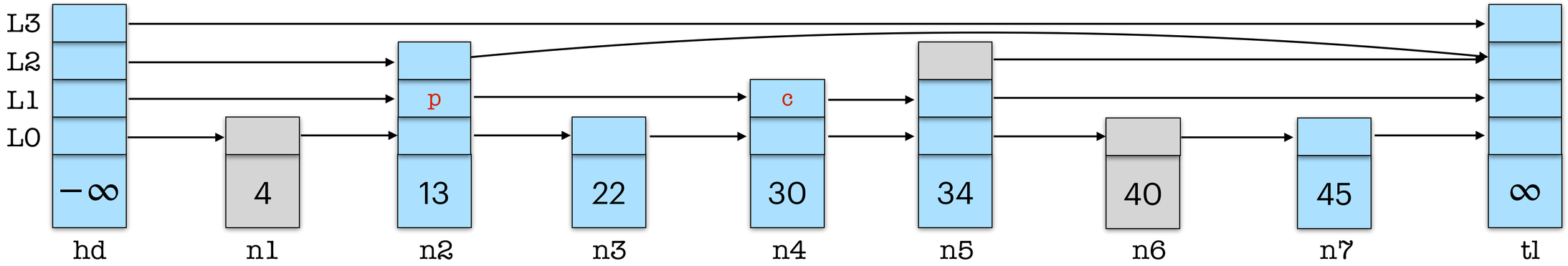


Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

search(42)

	pred	succ
L3	hd	tl
L2	n2	tl
L1		
L0		

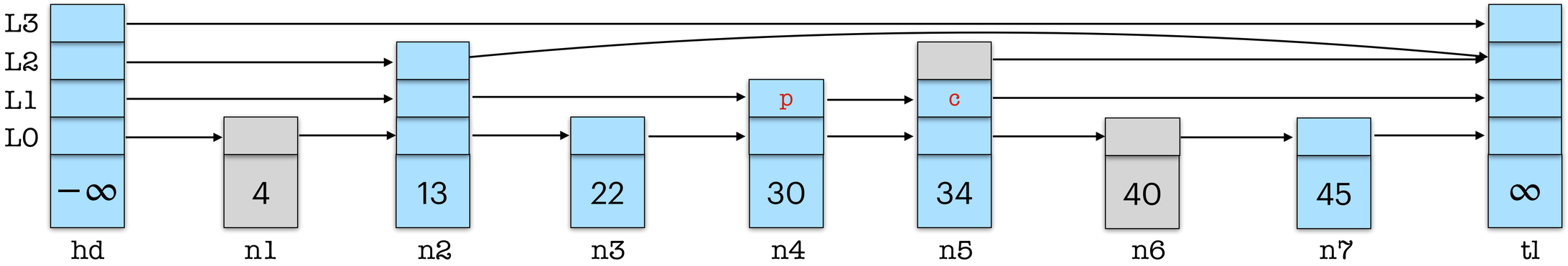


Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

search(42)

	pred	succ
L3	hd	tl
L2	n2	tl
L1		
L0		

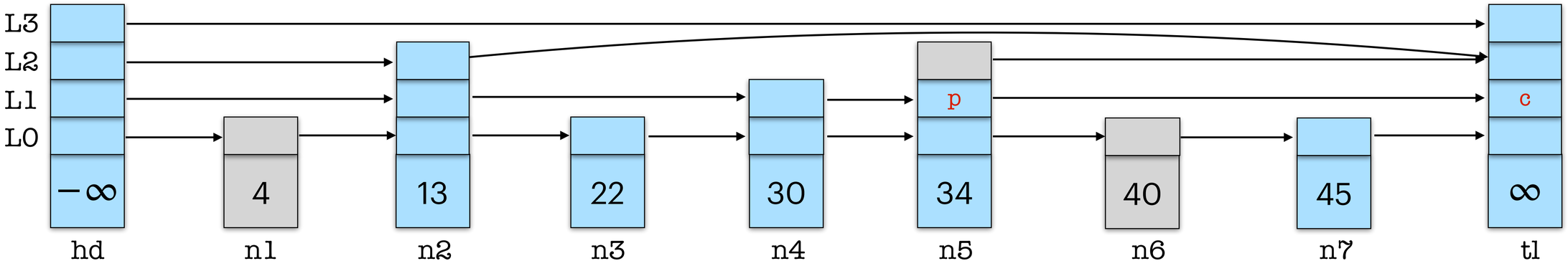


Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

search(42)

	pred	succ
L3	hd	tl
L2	n2	tl
L1	n5	tl
L0		

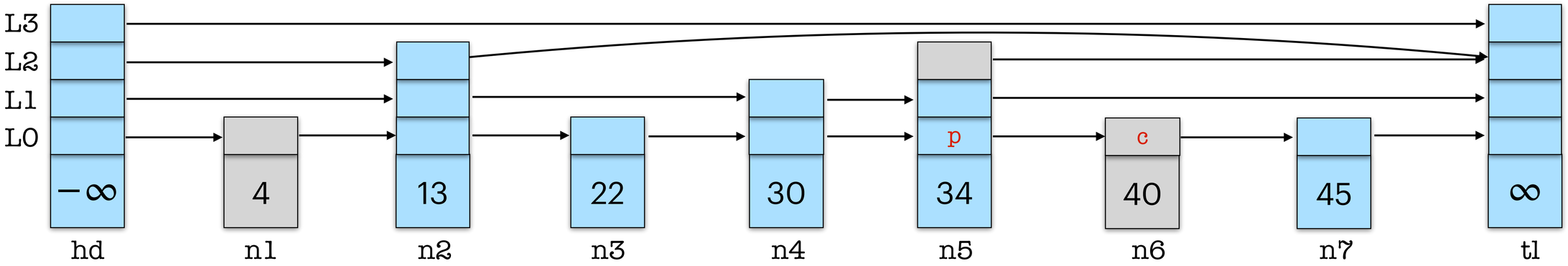


Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

search(42)

	pred	succ
L3	hd	tl
L2	n2	tl
L1	n5	tl
L0		

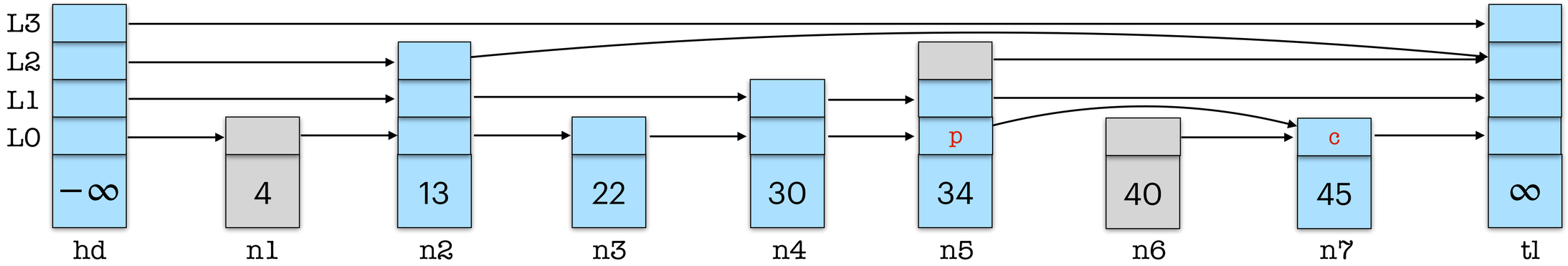


Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

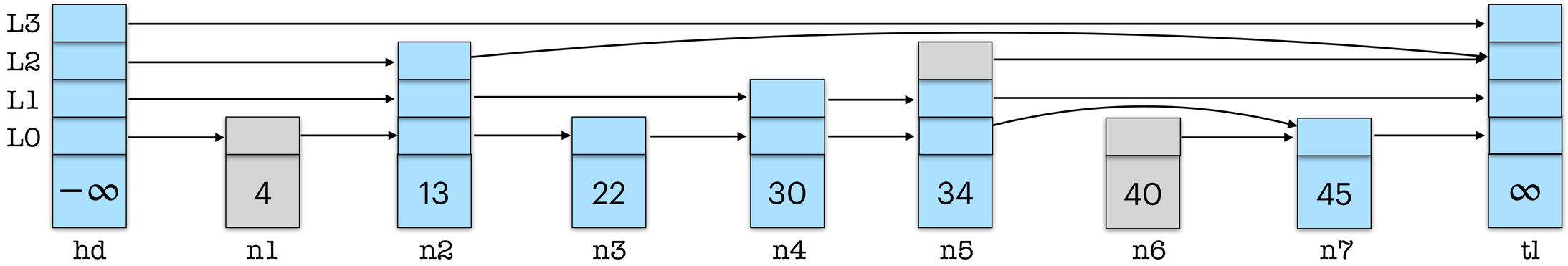
search(42)

	pred	succ
L3	hd	tl
L2	n2	tl
L1	n5	tl
L0	n5	n7



Skiplists

Harris List + Levels ≈ ??



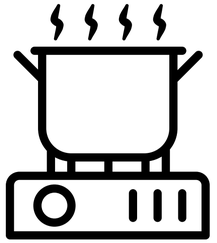
Outline

Step 1:
Find a class of structures with common correctness reasoning

- ECOOP24 : (Lock-free, single-copy) linked lists and skiplists
- OOPSLA21 : (Lock-based, multicopy) Log-Structured Merge (LSM) Trees

Step 2:
Develop enabling technology

- Template Algorithms
- Hindsight Framework
- Keyset Reasoning



Step 3:
Formalize the proof

- Evaluation

Skiplist Templates

```
let search k =
  let ps = allocArr L hd in
  let cs = allocArr L tl in
  let _, _, res = traverse ps cs k in
  res

let delete k =
  let ps = allocArr L hd in
  let cs = allocArr L tl in
  let p, c, res = traverse ps cs k in
  if not res then
    false
  else
    maintainanceOp_del c;
    match markNode 0 c with
    | Success -> traverse ps cs k; true
    | Failure -> false
```

```
let insert k =
  let ps = allocArr L hd in
  let cs = allocArr L tl in
  let p, c, res = traverse ps cs k in
  if res then
    false
  else
    let h = randomNum L in
    let e = createNode k h cs in
    match changeNext 0 p c e with
    | Success ->
      maintainanceOp_ins k ps cs e; true
    | Failure -> insert k
```

Skiplist Templates

```
let search k =
  let ps = allocArr L hd in
  let cs = allocArr L tl in
  let _, _, res = traverse ps cs k in
  res

let delete k =
  let ps = allocArr L hd in
  let cs = allocArr L tl in
  let p, c, res = traverse ps cs k in
  if not res then
    false
  else
    maintainanceOp_del c;
    match markNode 0 c with
    | Success -> traverse ps cs k; true
    | Failure -> false
```

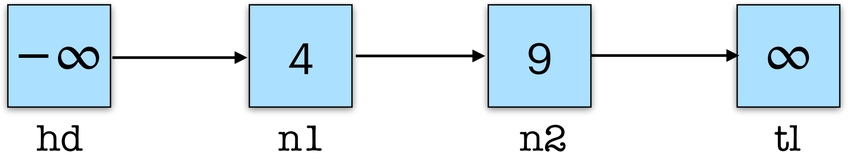
```
let insert k =
  let ps = allocArr L hd in
  let cs = allocArr L tl in
  let p, c, res = traverse ps cs k in
  if res then
    false
  else
    let h = randomNum L in
    let e = createNode k h cs in
    match changeNext 0 p c e with
    | Success ->
      maintainanceOp_ins k ps cs e; true
    | Failure -> insert k
```

Skiplist Templates

```
1  let search k =
2    let ps = allocArr L hd in
3    let cs = allocArr L tl in
4    let _, _, res = traverse ps cs k in
5    res
6
7  let delete k =
8    let ps = allocArr L hd in
9    let cs = allocArr L tl in
10   let p, c, res = traverse ps cs k in
11   if not res then
12     false
13   else
14     maintainanceOp_del c;
15     match markNode 0 c with
16     | Success -> traverse ps cs k; true
17     | Failure -> false
```

```
18 let insert k =
19   let ps = allocArr L hd in
20   let cs = allocArr L tl in
21   let p, c, res = traverse ps cs k in
22   if res then
23     false
24   else
25     let h = randomNum L in
26     let e = createNode k h cs in
27     match changeNext 0 p c e with
28     | Success ->
29       maintainanceOp_ins k ps cs e; true
30     | Failure -> insert k
```

insert(7)

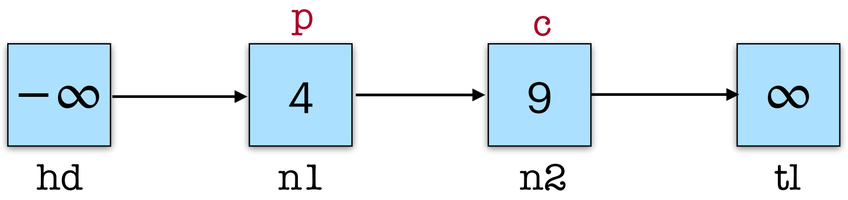


Skiplist Templates

```
1  let search k =
2    let ps = allocArr L hd in
3    let cs = allocArr L tl in
4    let _, _, res = traverse ps cs k in
5    res
6
7  let delete k =
8    let ps = allocArr L hd in
9    let cs = allocArr L tl in
10   let p, c, res = traverse ps cs k in
11   if not res then
12     false
13   else
14     maintainanceOp_del c;
15     match markNode 0 c with
16     | Success -> traverse ps cs k; true
17     | Failure -> false
```

```
18 let insert k =
19   let ps = allocArr L hd in
20   let cs = allocArr L tl in
21   let p, c, res = traverse ps cs k in
22   if res then
23     false
24   else
25     let h = randomNum L in
26     let e = createNode k h cs in
27     match changeNext 0 p c e with
28     | Success ->
29       maintainanceOp_ins k ps cs e; true
30     | Failure -> insert k
```

insert(7)

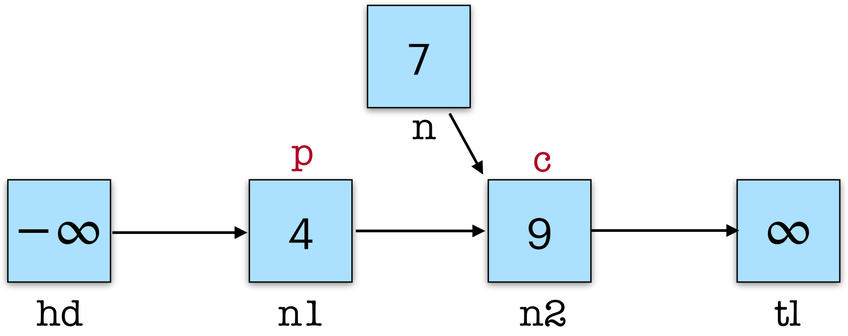


Skiplist Templates

```
1  let search k =
2    let ps = allocArr L hd in
3    let cs = allocArr L tl in
4    let _, _, res = traverse ps cs k in
5    res
6
7  let delete k =
8    let ps = allocArr L hd in
9    let cs = allocArr L tl in
10   let p, c, res = traverse ps cs k in
11   if not res then
12     false
13   else
14     maintainanceOp_del c;
15     match markNode 0 c with
16     | Success -> traverse ps cs k; true
17     | Failure -> false
```

```
18 let insert k =
19   let ps = allocArr L hd in
20   let cs = allocArr L tl in
21   let p, c, res = traverse ps cs k in
22   if res then
23     false
24   else
25     let h = randomNum L in
26     let e = createNode k h cs in
27     match changeNext 0 p c e with
28     | Success ->
29       maintainanceOp_ins k ps cs e; true
30     | Failure -> insert k
```

insert(7)

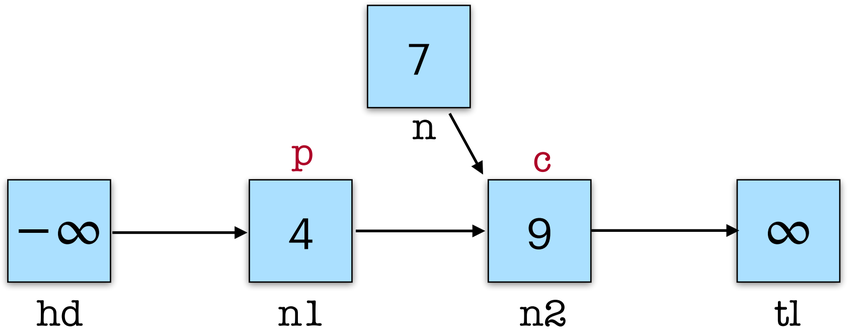


Skiplist Templates

```
1  let search k =
2    let ps = allocArr L hd in
3    let cs = allocArr L tl in
4    let _, _, res = traverse ps cs k in
5    res
6
7  let delete k =
8    let ps = allocArr L hd in
9    let cs = allocArr L tl in
10   let p, c, res = traverse ps cs k in
11   if not res then
12     false
13   else
14     maintainanceOp_del c;
15     match markNode 0 c with
16     | Success -> traverse ps cs k; true
17     | Failure -> false
```

```
18 let insert k =
19   let ps = allocArr L hd in
20   let cs = allocArr L tl in
21   let p, c, res = traverse ps cs k in
22   if res then
23     false
24   else
25     let h = randomNum L in
26     let e = createNode k h cs in
27     match changeNext 0 p c e with
28     | Success ->
29       maintainanceOp_ins k ps cs e; true
30     | Failure -> insert k
```

insert(7)

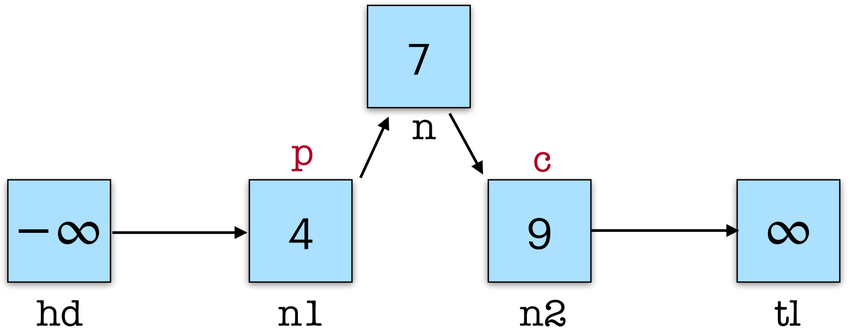


Skiplist Templates

```
1  let search k =
2    let ps = allocArr L hd in
3    let cs = allocArr L tl in
4    let _, _, res = traverse ps cs k in
5    res
6
7  let delete k =
8    let ps = allocArr L hd in
9    let cs = allocArr L tl in
10   let p, c, res = traverse ps cs k in
11   if not res then
12     false
13   else
14     maintainanceOp_del c;
15     match markNode 0 c with
16     | Success -> traverse ps cs k; true
17     | Failure -> false
```

```
18 let insert k =
19   let ps = allocArr L hd in
20   let cs = allocArr L tl in
21   let p, c, res = traverse ps cs k in
22   if res then
23     false
24   else
25     let h = randomNum L in
26     let e = createNode k h cs in
27     match changeNext 0 p c e with
28     | Success ->
29       maintainanceOp_ins k ps cs e; true
30     | Failure -> insert k
```

insert(7)



Skiplist Templates

```
let search k =
  let ps = allocArr L hd in
  let cs = allocArr L tl in
  let _, _, res = traverse ps cs k in
  res

let delete k =
  let ps = allocArr L hd in
  let cs = allocArr L tl in
  let p, c, res = traverse ps cs k in
  if not res then
    false
  else
    maintainanceOp_del c;
    match markNode 0 c with
    | Success -> traverse ps cs k; true
    | Failure -> false
```

```
let insert k =
  let ps = allocArr L hd in
  let cs = allocArr L tl in
  let p, c, res = traverse ps cs k in
  if res then
    false
  else
    let h = randomNum L in
    let e = createNode k h cs in
    match changeNext 0 p c e with
    | Success ->
      maintainanceOp_ins k ps cs e; true
    | Failure -> insert k
```

Skiplist Templates

```
let eager_i i k p c =
  match findNext i c with
  | cn, true ->
    match changeNext i p c cn with
    | Success -> eager_i i k p cn
    | Failure -> traverse ps cs k
  | cn, false ->
    let kc = getKey c in
    if kc < k then
      eager_i i k c cn
    else
      let res = (kc = k ? true : false) in
      (p, c, res)
```

```
let eager_rec i ps cs k =
  let p = ps[i+1] in
  let c, _ = findNext i p in
  let p', c', res = eager_i i k p c in
  ps[i] <- p';
  cs[i] <- c';
  if i = 0 then
    (p', c', res)
  else
  eager_rec (i-1) ps cs k

let traverse ps cs k =
  eager_rec (L - 2) ps cs k
```

Skiplist Templates

```
let search k =
  let ps = allocArr L hd in
  let cs = allocArr L tl in
  let _, _, res = traverse ps cs k in
  res

let delete k =
  let ps = allocArr L hd in
  let cs = allocArr L tl in
  let p, c, res = traverse ps cs k in
  if not res then
    false
  else
    maintainanceOp_del c;
    match markNode 0 c with
    | Success -> traverse ps cs k; true
    | Failure -> false
```

```
let insert k =
  let ps = allocArr L hd in
  let cs = allocArr L tl in
  let p, c, res = traverse ps cs k in
  if res then
    false
  else
    let h = randomNum L in
    let e = createNode k h cs in
    match changeNext 0 p c e with
    | Success ->
      maintainanceOp_ins k ps cs e; true
    | Failure -> insert k
```

Node-level Specs:

{ Node(n, k, m, n') } markNode i n { Node(n, k, m[i ↦ true], n') }

....

traverse Spec: ....

maintenanceOp Spec : ...

$\Longrightarrow$

Proof of the Skiplist Template

Atomic Triples

$$\langle\ C.\ \mathtt{CSS}(r, C)\ \rangle\ \ \mathtt{op}(r, k)\ \ \langle\ res.\ \exists\ C',\ \mathtt{CSS}(r, C') * \Psi(\mathtt{op}, k, C, C', res)\ \rangle$$

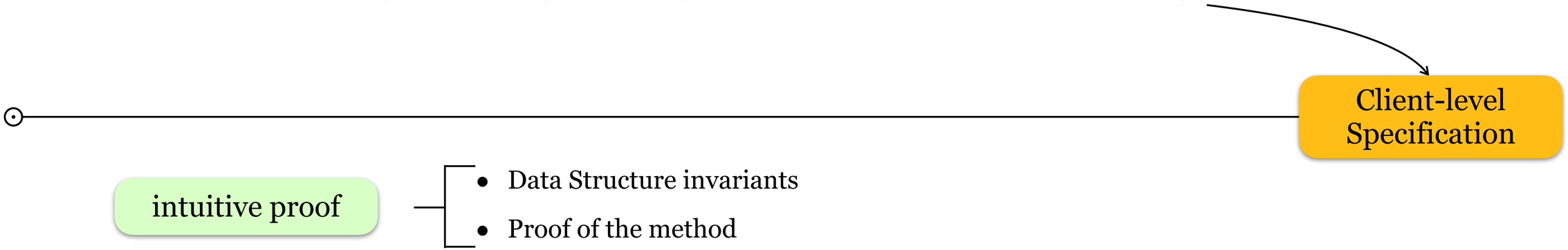
Atomic Triples

$$\langle\ C.\ \mathrm{CSS}(r, C)\ \rangle\ \ \mathrm{op}(r, k)\ \ \langle\ res.\ \exists\ C',\ \mathrm{CSS}(r, C') * \Psi(\mathrm{op}, k, C, C', res)\ \rangle$$

$$\Psi(\mathrm{op}, k, C, C', res) = \begin{cases} C = C' \ \&\&\ (res \leftrightarrow k \in C) & \mathrm{op} = \mathrm{search} \\ C = C' \cup \{k\} \ \&\&\ (res \leftrightarrow k \notin C) & \mathrm{op} = \mathrm{insert} \\ C = C' \setminus \{k\} \ \&\&\ (res \leftrightarrow k \in C) & \mathrm{op} = \mathrm{delete} \end{cases}$$

Atomic Triples

$\langle\ C.\ \mathtt{CSS}(r, C)\ \rangle\ \mathtt{op}(r, k)\ \langle\ res.\ \exists\ C',\ \mathtt{CSS}(r, C') * \Psi(op, k, C, C', res)\ \rangle$



Atomic Triples

$\langle\ C.\ \mathrm{CSS}(r, C)\ \rangle\ \mathrm{op}(r, k)\ \langle\ res.\ \exists\ C',\ \mathrm{CSS}(r, C') * \Psi(op, k, C, C', res)\ \rangle$

intuitive proof

- Data Structure invariants
- Proof of the method
- Prophecy variables:
  - What to predict?
- Helping Protocol:
  - Which threads require helping?
  - Who does the helping?
  - When is helping required?

Atomic Triples

$\langle\ C.\ \mathtt{CSS}(r, C)\ \rangle\ \mathtt{op}(r, k)\ \langle\ res.\ \exists\ C',\ \mathtt{CSS}(r, C') * \Psi(op, k, C, C', res)\ \rangle$

Client-level Specification

intuitive proof

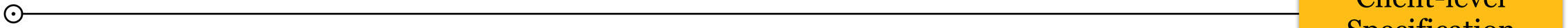
- Data Structure invariants
- Proof of the method

Is there a uniform answer?

- Prophecy variables:
  - What to predict?
- Helping Protocol:
  - Which threads require helping?
  - Who does the helping?
  - When is helping required?

Atomic Triples

$\langle\, C.\ \mathrm{CSS}(r, C) \,\rangle\ \mathrm{op}(r, k)\ \langle\, res.\ \exists\, C',\ \mathrm{CSS}(r, C') * \Psi(\mathrm{op}, k, C, C', res) \,\rangle$



intuitive proof

- Data Structure invariants
- Proof of the method

Is there a uniform answer?

Yes!

- Prophecy variables:
  - What to predict?
- Helping Protocol:
  - Which threads require helping?
  - Who does the helping?
  - When is helping required?

Hindsight Framework

- Data Structure invariants
- Proof of the method

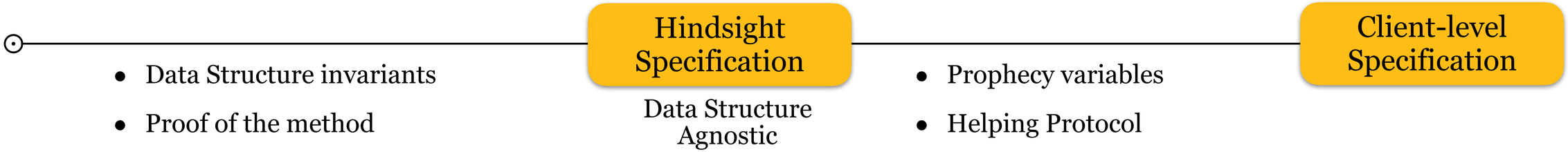
- Prophecy variables
- Helping Protocol

Hindsight Specification :

- Precondition : Modifying LP ⟶ Postcondition : Receipt of linearization
- Precondition : Unmodifying LP ⟶ Postcondition : at some point during the execution, $\Psi(\texttt{op}, \texttt{k}, \texttt{C}, \texttt{C}', \texttt{res})$ was true

- Applicable to structures that exhibit ***unmodifying future-dependent*** LPs.

Hindsight Framework

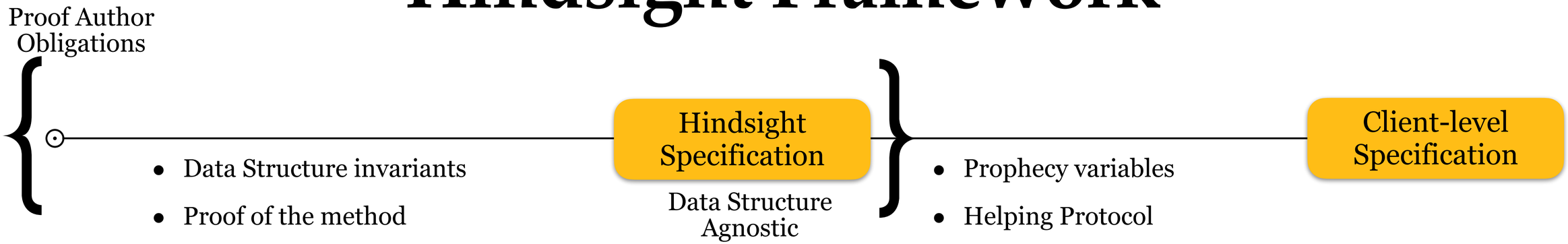


Hindsight Specification :

- Precondition : Modifying LP ⟶ Postcondition : Receipt of linearization
- Precondition : Unmodifying LP ⟶ Postcondition : at some point during the execution, $\Psi(\texttt{op}, \texttt{k}, \texttt{C}, \texttt{C}', \texttt{res})$ was true

- Applicable to structures that exhibit ***unmodifying future-dependent*** LPs.

Hindsight Framework



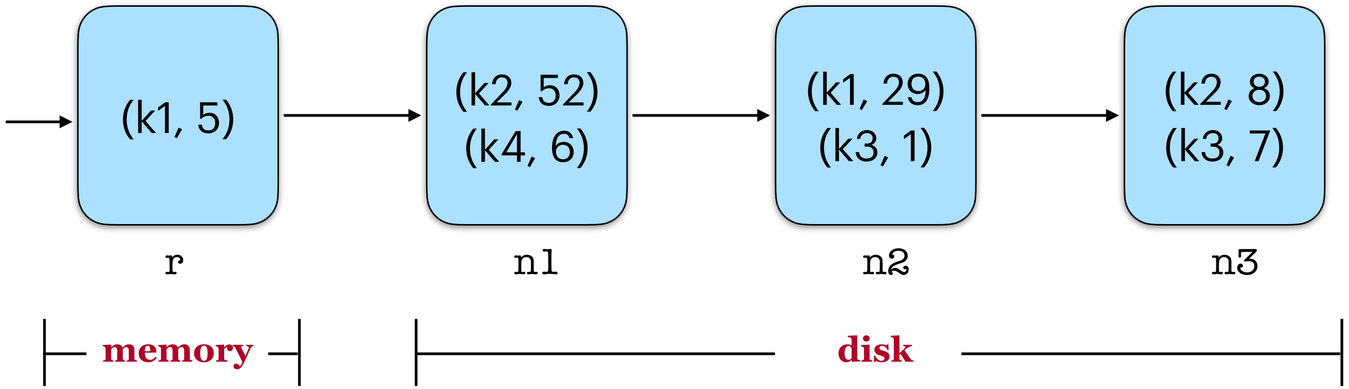
Hindsight Specification :

- Precondition : Modifying LP ⟶ Postcondition : Receipt of linearization
- Precondition : Unmodifying LP ⟶ Postcondition : at some point during the execution, $\Psi(\texttt{op}, \texttt{k}, \texttt{C}, \texttt{C}', \texttt{res})$ was true

- Applicable to structures that exhibit ***unmodifying future-dependent*** LPs.

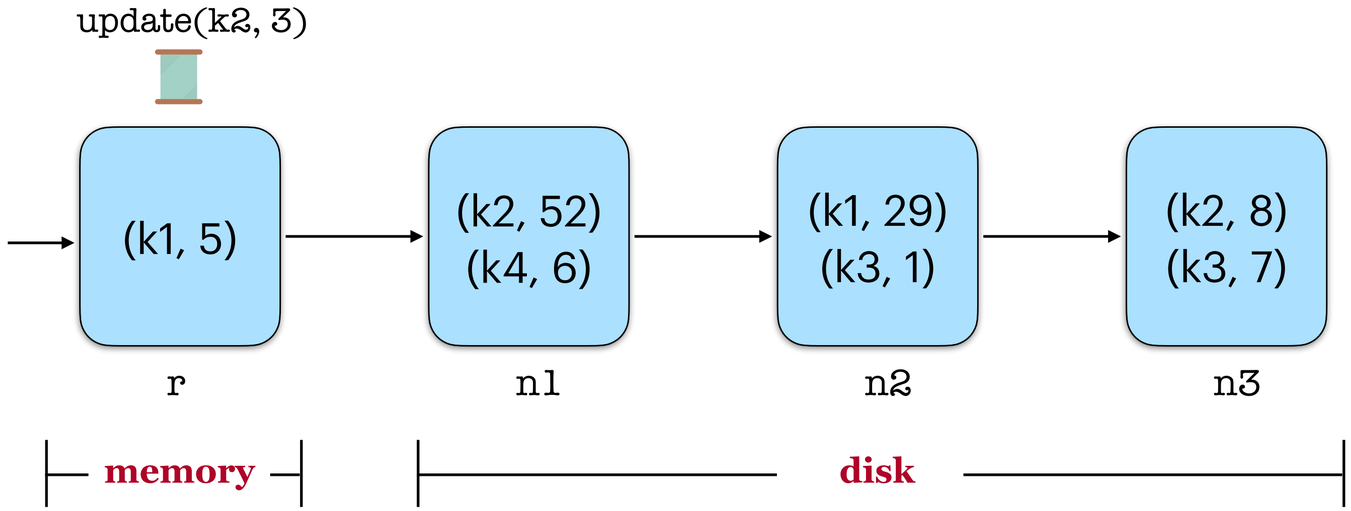
Multicopy Search Structures

Optimized for write-heavy workloads



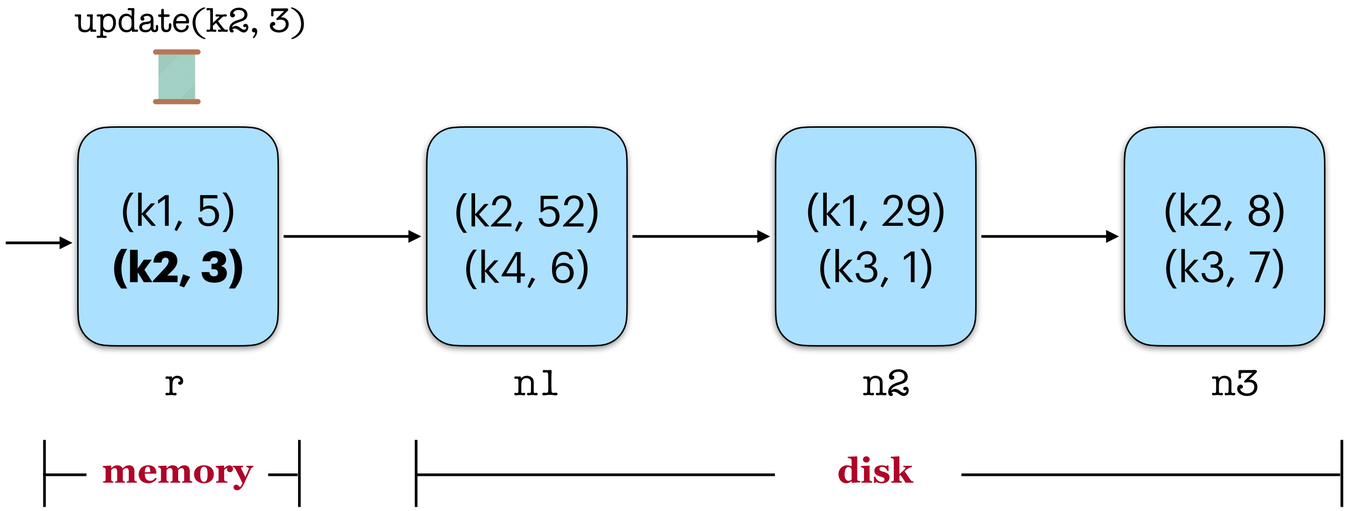
Multicopy Search Structures

Optimized for write-heavy workloads



Multicopy Search Structures

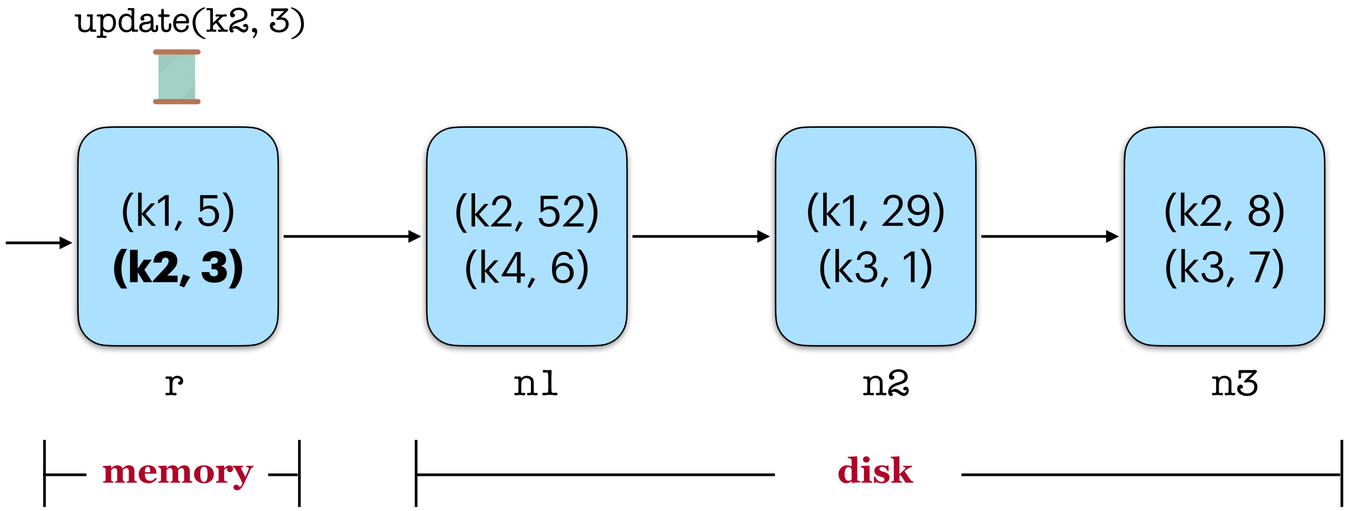
Optimized for write-heavy workloads



Multicopy Search Structures

Optimized for write-heavy workloads

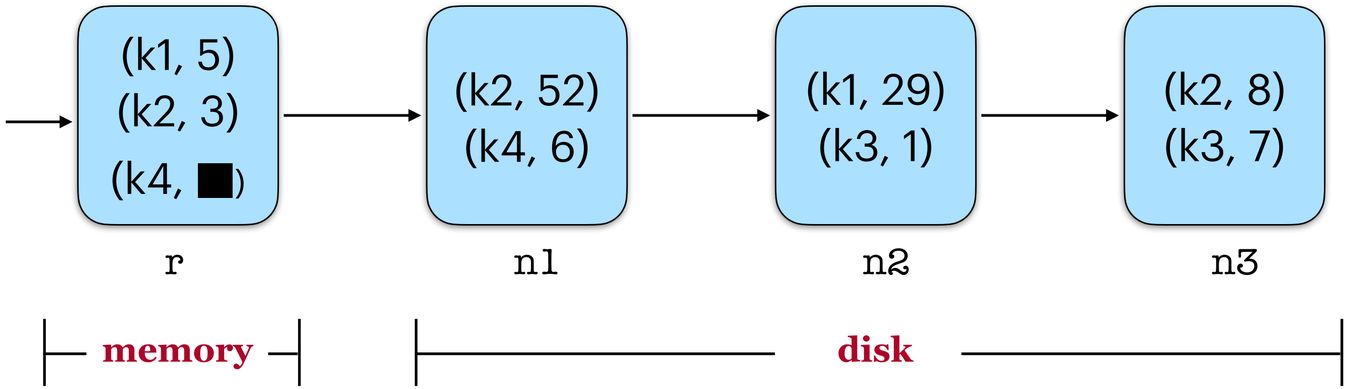
delete(k) ~ upsert(k, ■)



Multicopy Search Structures

Optimized for write-heavy workloads

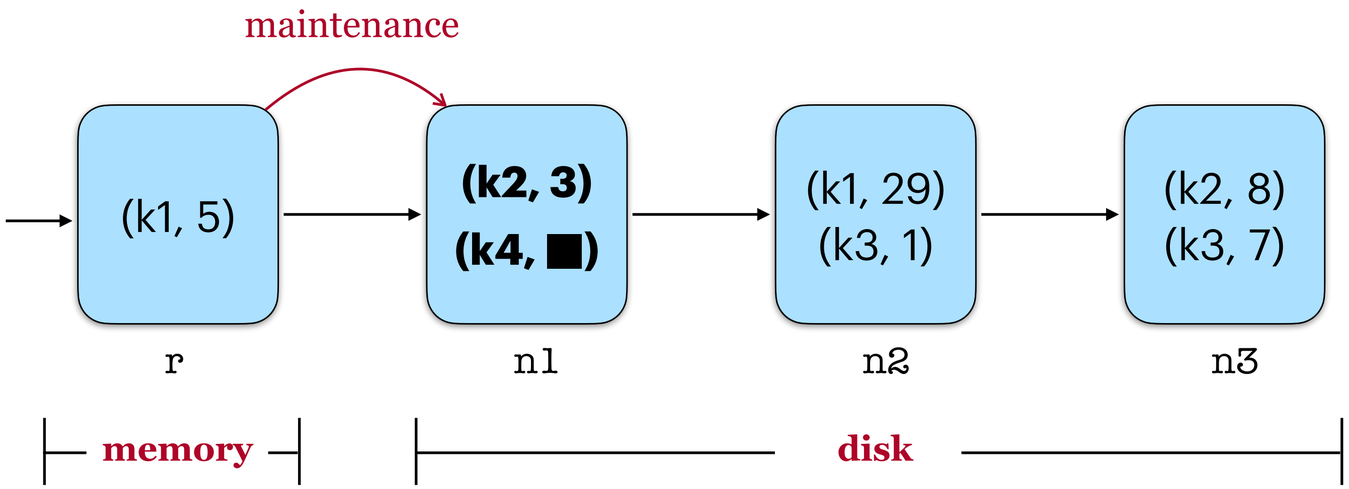
delete(k) ~ upsert(k, ■)



Multicopy Search Structures

Optimized for write-heavy workloads

delete(k) ~ upsert(k, ■)

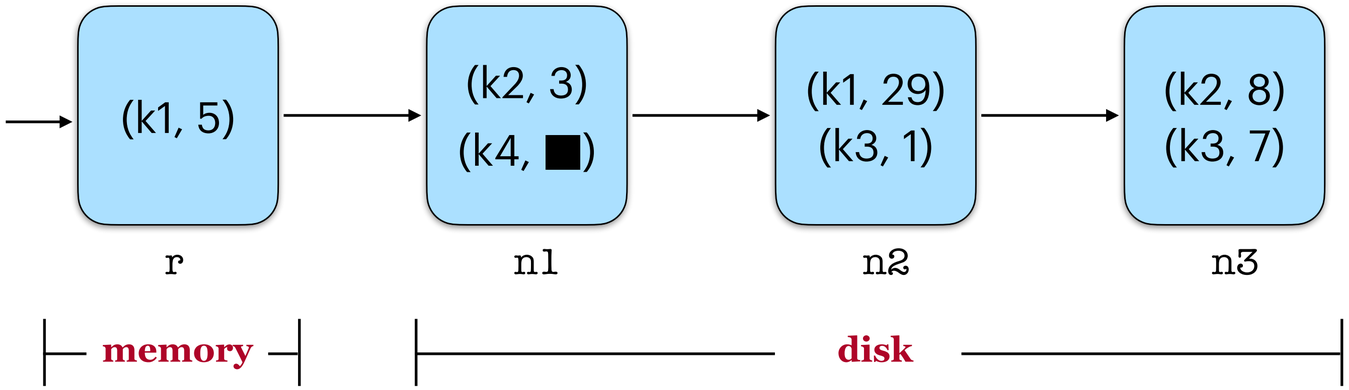


Multicopy Search Structures

Optimized for write-heavy workloads

delete(k) ~ upsert(k, ■)

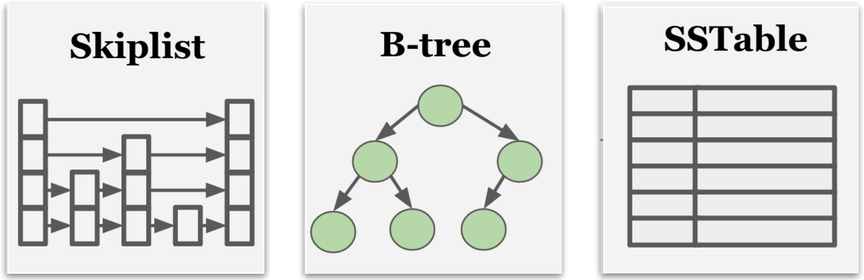
search(k3) returns (k3, 1)



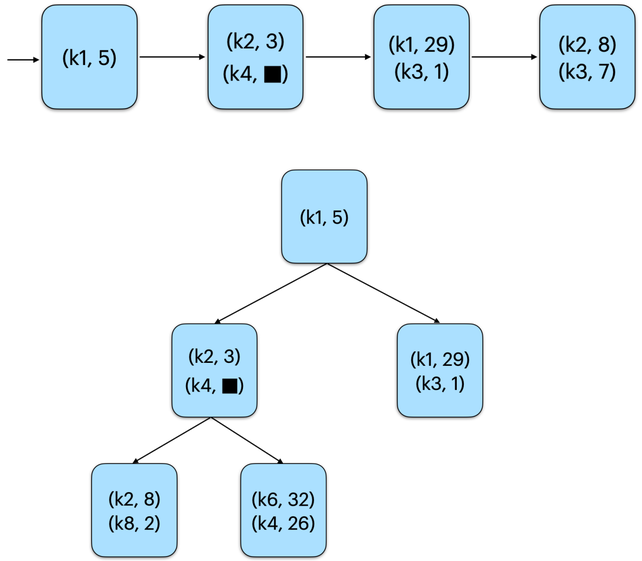
Multicopy Search Structures

LSM trees are used in data stores such as Bigtable, HBase, LevelDB, SQLite4,[5] Tarantool,[6] RocksDB, WiredTiger,[7] Apache Cassandra, InfluxDB[8] and ScyllaDB.

Inner Structure



Outer Structure

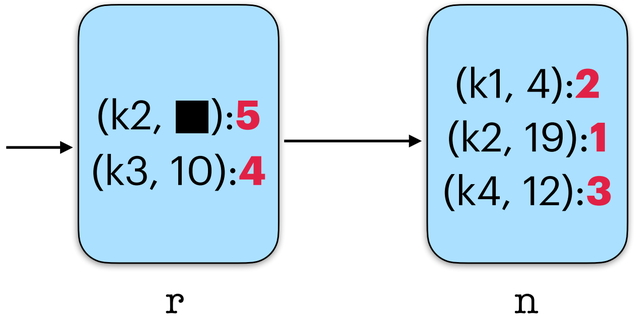


LSM-DAG Templates

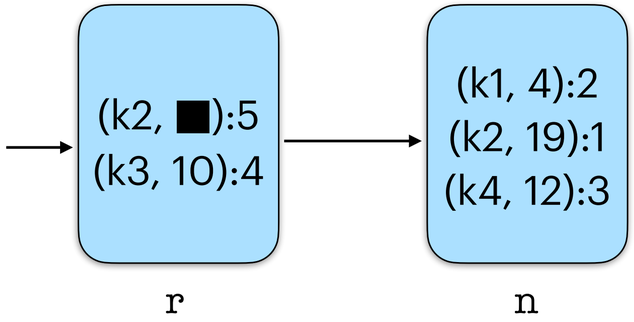
```
let rec traverse r n k =
  lockNode n;
  match inContents r n k with
  | Some v -> unlockNode n; v
  | None ->
    match findNext r n k with
    | Some n' ->
      unlockNode n;
      traverse r n' k
    | None -> unlockNode n; □

let search r k = traverse r r k
```

```
let rec upsert r k v =
  lockNode r;
  let res = addContents r k v in
  if res then
    unlockNode r
  else begin
    unlockNode r;
    upsert r k v
  end
```

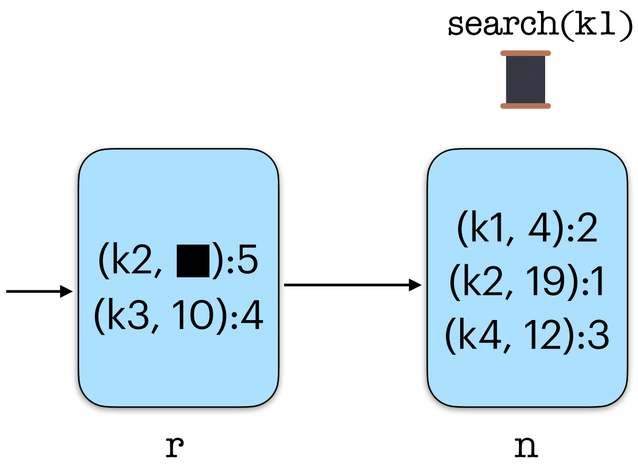
Problem Execution



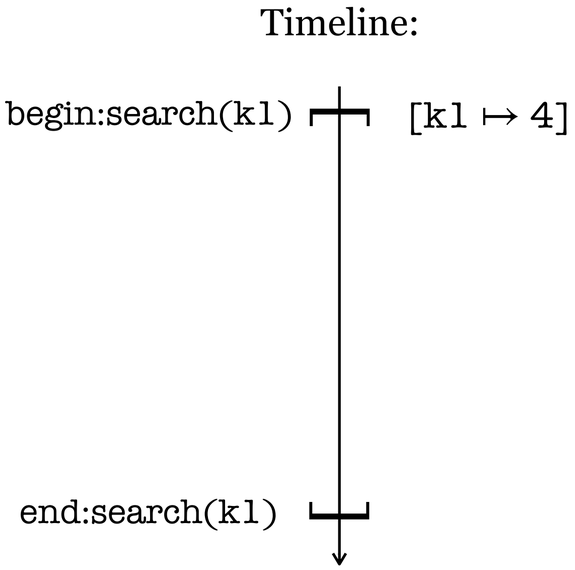
Problem Execution

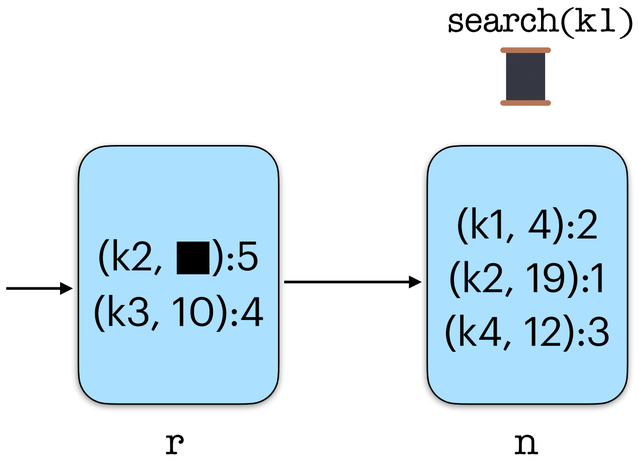


Problem Execution

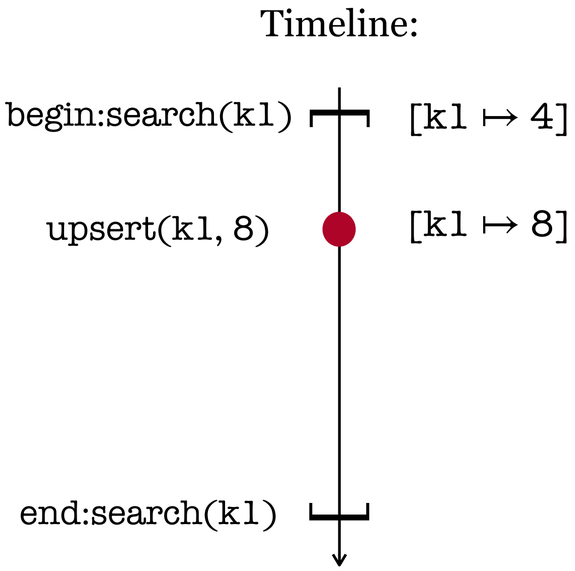


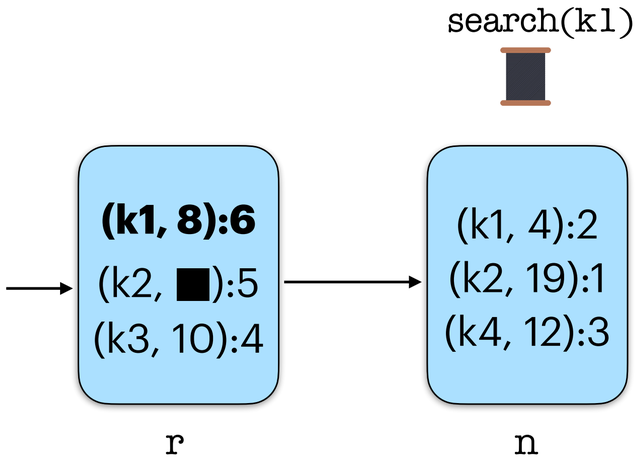
Problem Execution



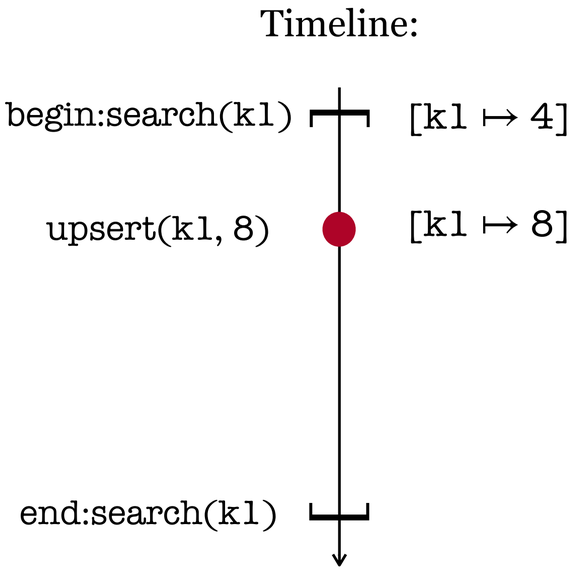


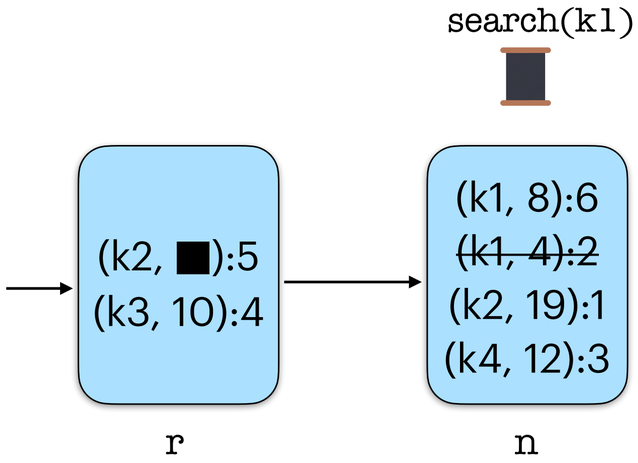
Problem Execution





Problem Execution

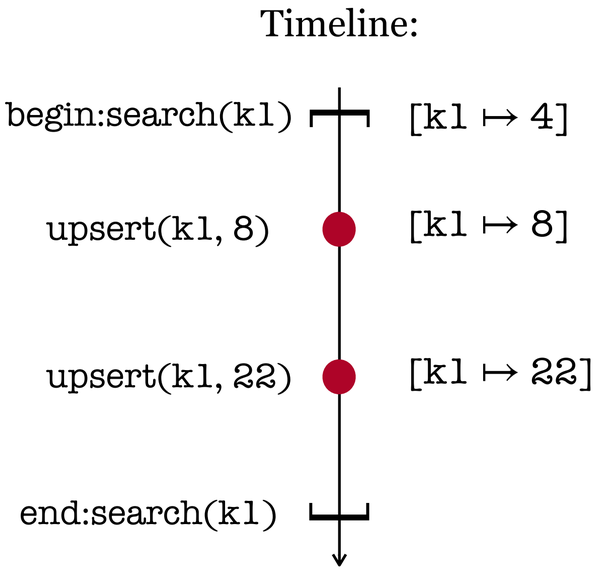


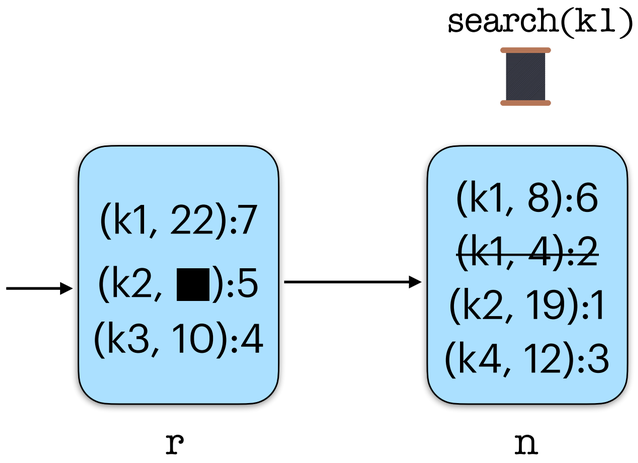


Problem Execution

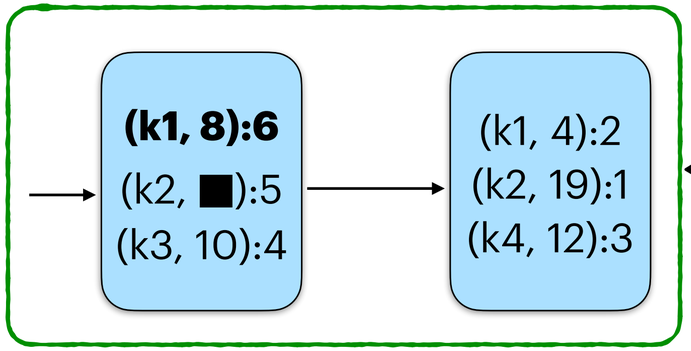
search(k1) finds (k1, 8) but [k1 ↦ 22]

When is the linearization point?



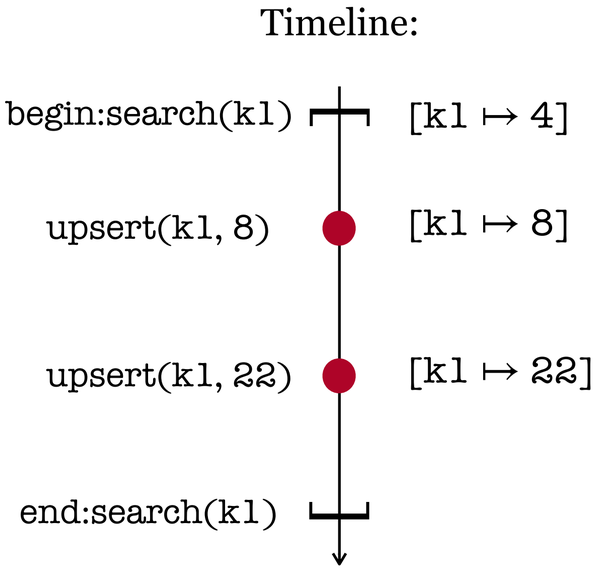


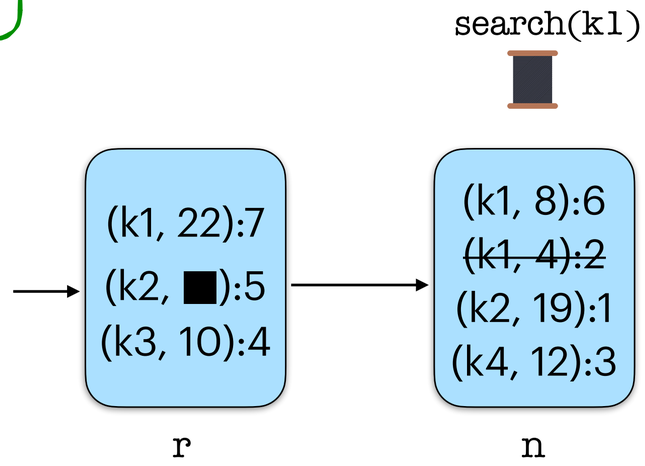
Problem Execution



search(k1) finds (k1, 8) but [k1 ↦ 22]

When is the linearization point?



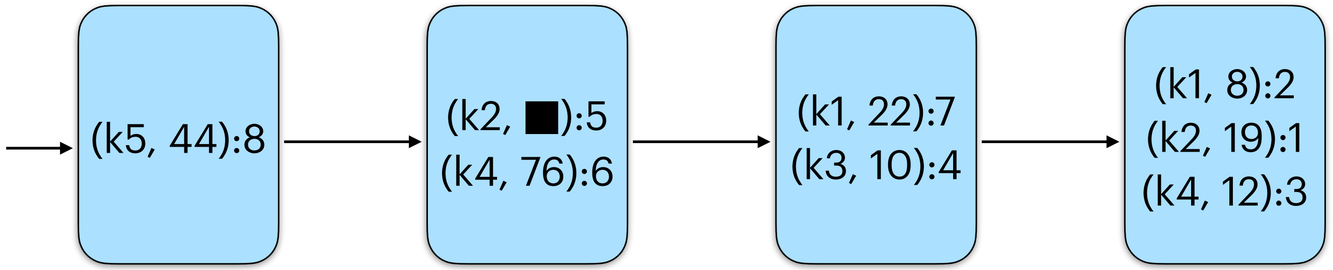


Search Recency

If $(k, v_0) : t_0$ is the latest copy of k when `search(k)` starts, then it returns either:

(i) $(k, v_0) : t_0$ or

(ii) $(k, v) : t$ for some v, t with $t > t_0$.



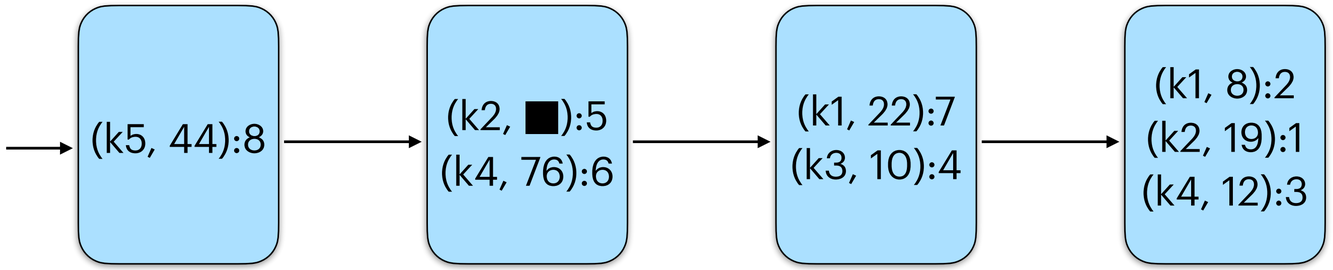
Search Recency

If $(k, v_0) : t_0$ is the latest copy of k when `search(k)` starts, then it returns either:

(i) $(k, v_0) : t_0$ or

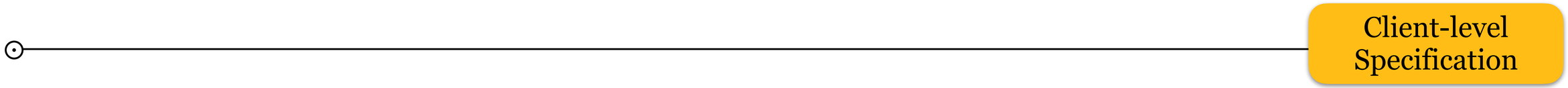
(ii) $(k, v) : t$ for some v, t with $t > t_0$.

Invariant := "timestamp of *reachable content* never decreases"



Multicopy with Hindsight

Framework provides:



Proof author obligations:

Multicopy with Hindsight

Framework provides:

Proof author obligations:

Multicopy with Hindsight

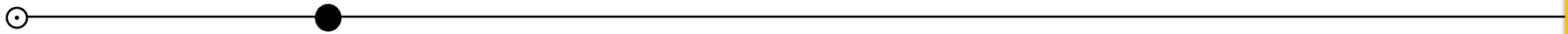
Framework provides:

Proof author obligations:

Multicopy with Hindsight

Framework provides:

0: Shared state invariant for storing history



Client-level Specification

1: Determine steps that may change the abstract state

Proof author obligations:

```
1  let rec traverse r n k =
2    lockNode n;
3    match inContents r n k with
4    | Some v -> unlockNode n; v
5    | None ->
6      match findNext r n k with
7      | Some n' ->
8        unlockNode n;
9        traverse r n' k
10     | None -> unlockNode n; □
11
12 let search r k = traverse r r k
```

```
13 let rec upsert r k v =
14   lockNode r;
15   let res = addContents r k v in
16   if res then
17     unlockNode r
18   else begin
19     unlockNode r;
20     upsert r k v
21   end
```

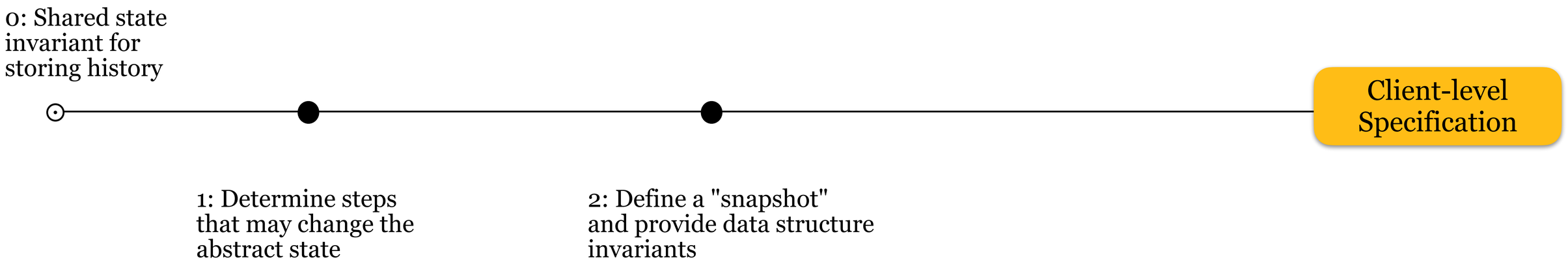
Multicopy with Hindsight

Framework provides:

Proof author obligations:

Multicopy with Hindsight

Framework provides:



Proof author obligations:

Multicopy with Hindsight

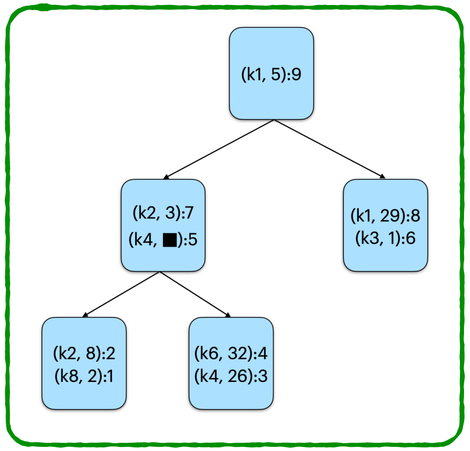
Framework provides:

0: Shared state invariant for storing history

Client-level Specification

1: Determine steps that may change the abstract state

2: Define a "snapshot" and provide data structure invariants

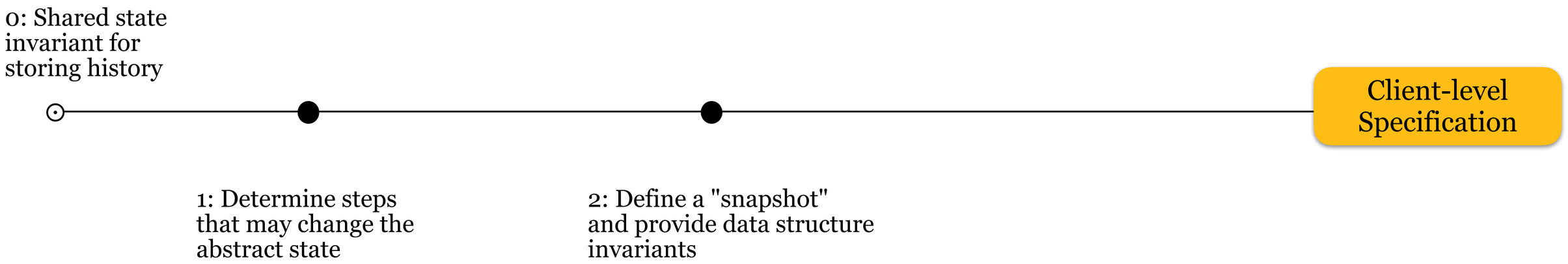


1. reachable content never decreases.
....
....
....

Proof author obligations:

Multicopy with Hindsight

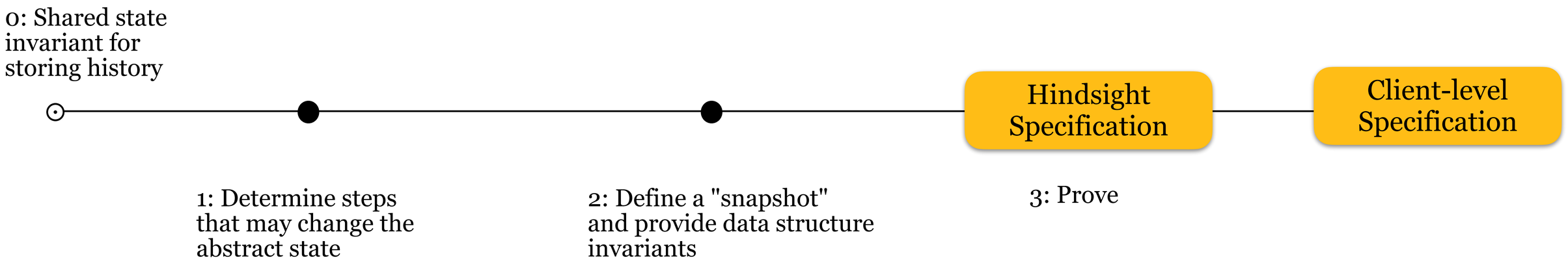
Framework provides:



Proof author obligations:

Multicopy with Hindsight

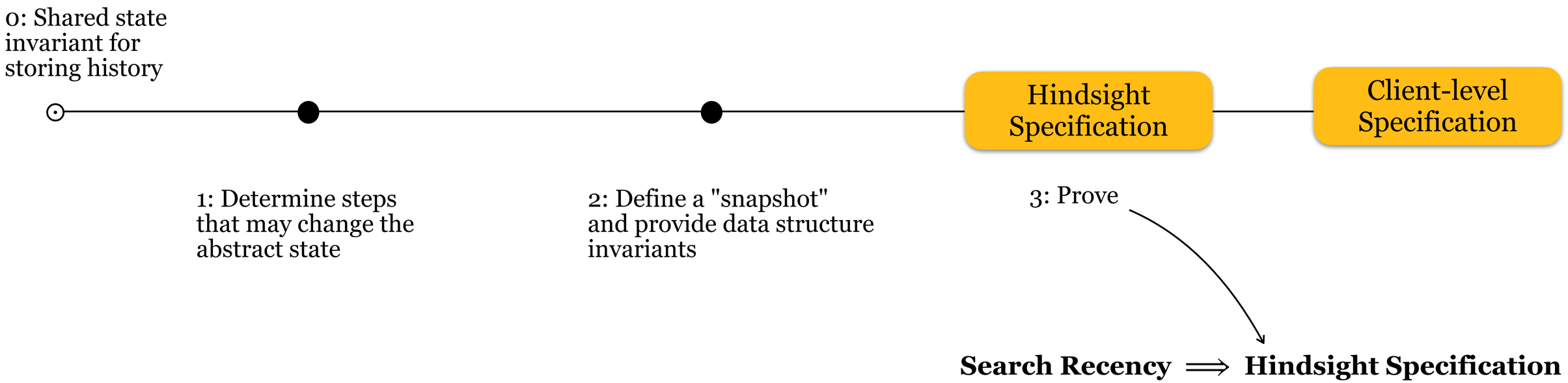
Framework provides:



Proof author obligations:

Multicopy with Hindsight

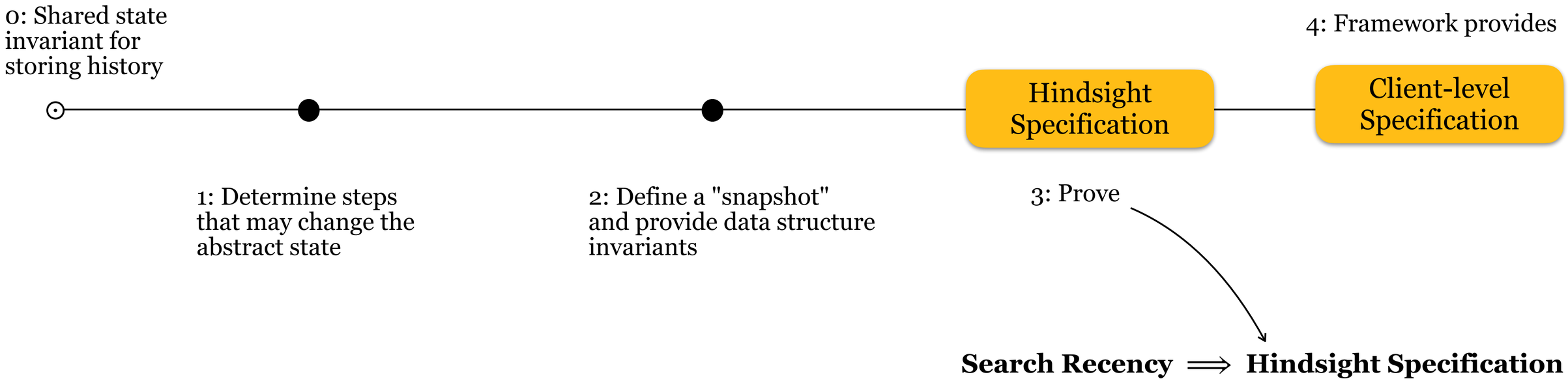
Framework provides:



Proof author obligations:

Multicopy with Hindsight

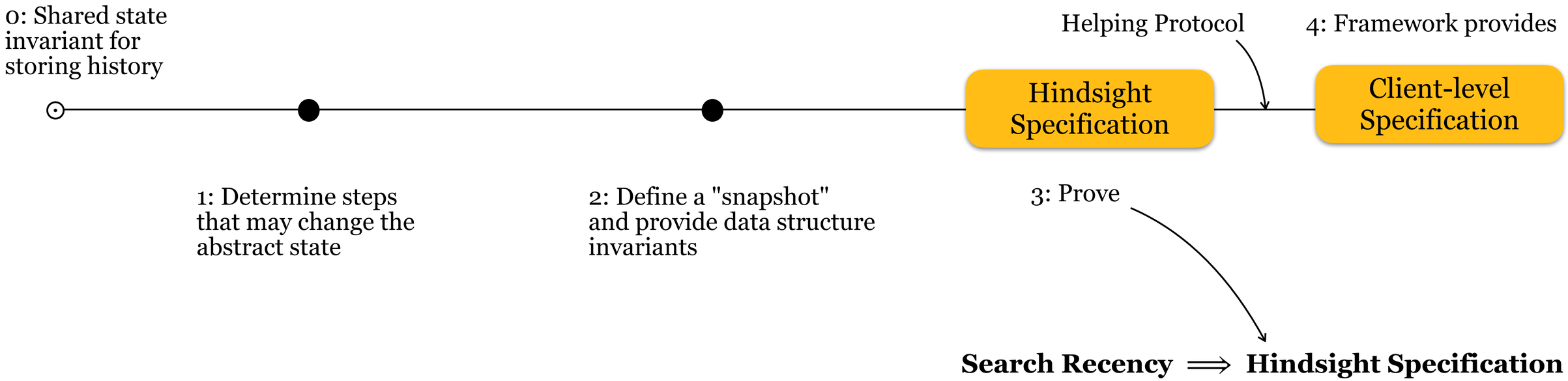
Framework provides:



Proof author obligations:

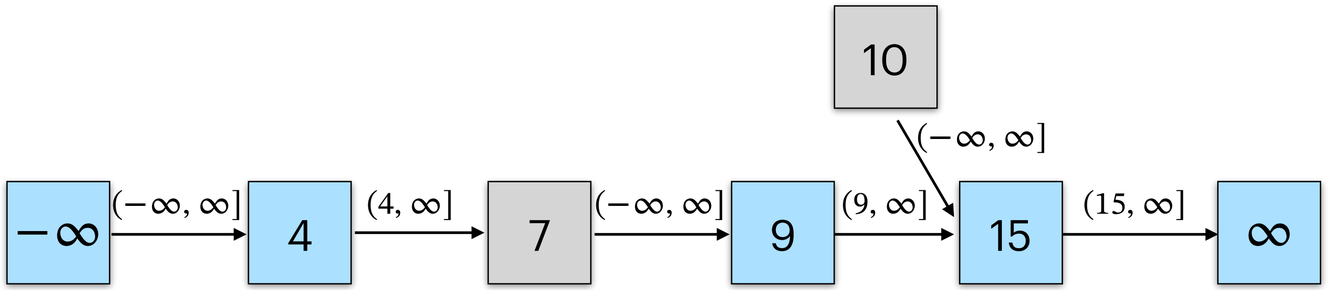
Multicopy with Hindsight

Framework provides:



Proof author obligations:

Global Graph Properties



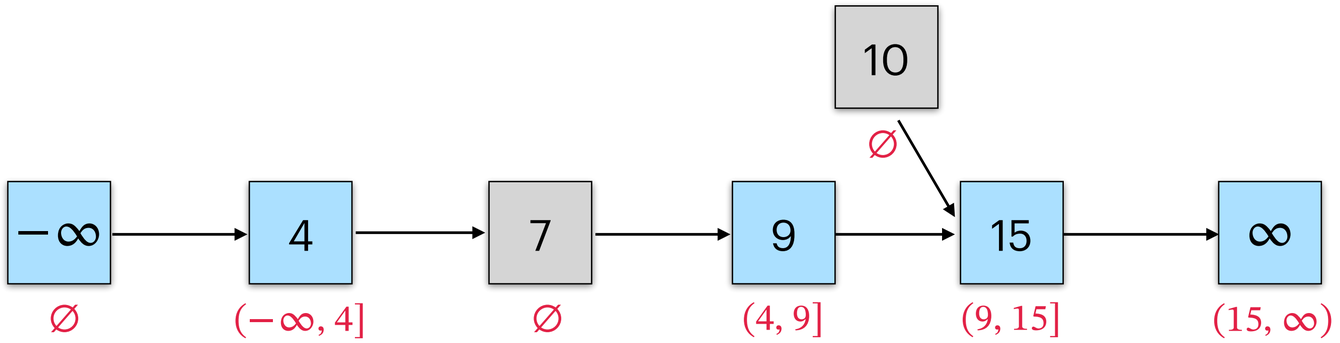
Edgeset Framework :

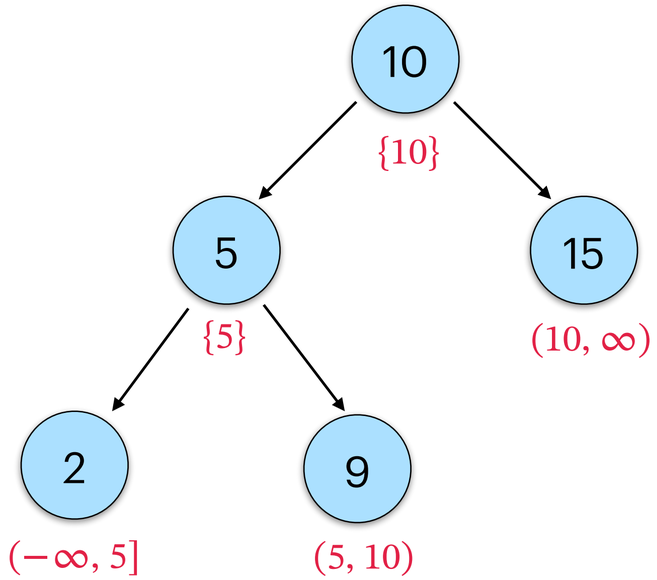
- Dennis E. Shasha and Nathan Goodman. *Concurrent Search Structure Algorithms*. [Database Syst. 1988]

Flow Framework :

- Siddharth Krishna, Dennis E. Shasha and Thomas Wies. *Go with the flow: compositional abstractions for concurrent data structures*. [POPL 2018]
- Siddharth Krishna, Alexander J. Summers and Thomas Wies. *Local reasoning for global graph properties*. [ESOP 2020]

Keysets





keyset(n) := "set of keys n is *responsible* for"

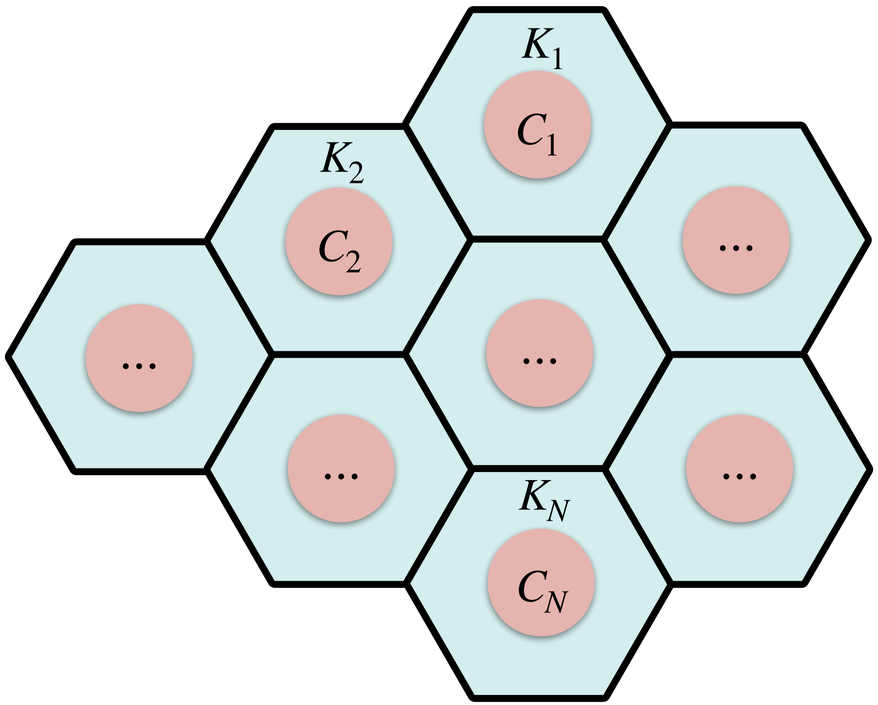
Two properties of keysets:

- Keysets partition the keyspace.
- For each node, its contents is a subset of its keyset.

Keyset Principle:

$$k \in \text{keyset}(n) \implies (k \in C(n) \iff k \in C)$$

Keyset RA



Iris resource algebra (RA) for keysets:

$$(K, C) = (K_1, C_1) \cdot (K_2, C_2) \ \cdots \ (K_N, C_N)$$

- $K = K_1 \uplus K_2 \uplus \cdots \uplus K_N$
- $C = C_1 \uplus C_2 \uplus \cdots \uplus C_N$
- $C_1 \subseteq K_1, C_2 \subseteq K_2, \ldots, C_N \subseteq K_N$

Critical piece in PLDI20 and ECOOP24!

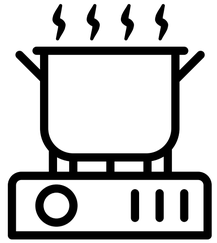
Outline

Step 1:
Find a class of structures with common correctness reasoning

- ECOOP24 : (Lock-free, single-copy) linked lists and skiplists
- OOPSLA21 : (Lock-based, multicopy) Log-Structured Merge (LSM) Trees

Step 2:
Develop enabling technology

- Template Algorithms
- Hindsight Framework
- Keyset Reasoning

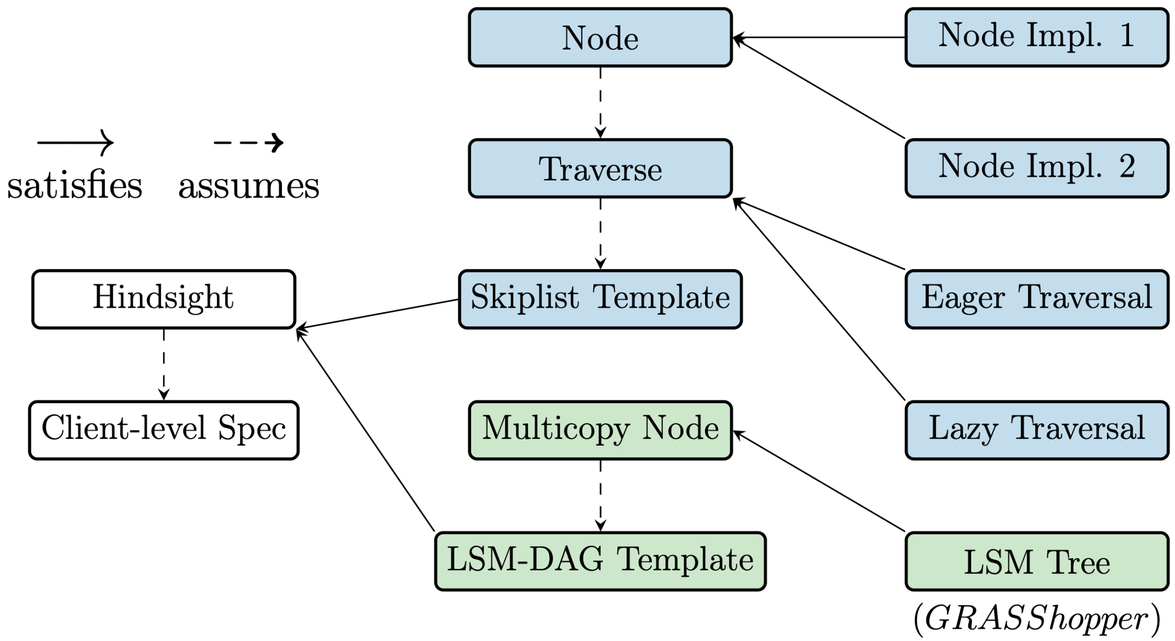


Step 3:
Formalize the proof

- Evaluation

Evaluation

- History stored as shared state invariant using a combination of authoritative and agreement RA.
- Heavy use of Coq's module system.
- Code available publicly on Github.



(*GRASShopper*)

Skiplist Template (Iris/Coq)

Module	Code	Proof	Total	Time
Flow Library	0	5330	5330	33
Hindsight	0	950	950	11
Client-level Spec	9	329	338	18
Skiplist	12	1693	1705	26
Skiplist Init(∗)	6	319	325	15
Skiplist Search(∗)	7	62	69	6
Skiplist Insert(∗)	37	3457	3494	111
Skiplist Delete(∗)	28	2401	2429	72
Node Impl. 1	118	908	1026	35
Node Impl. 2	106	836	942	35
Eager Traversal	38	1165	1203	96
Lazy Traversal	47	2063	2110	145
Total	**408**	**19513**	**19921**	**603**
Herlihy-Shavit	234	9933	10167	361

(time in seconds)

Evaluation - Multicopy

- Original proofs for the multicopy template from OOPSLA21.
- Hindsight proofs use the hindsight framework.
- Original proofs use a bespoke helping protocol, while hindsight proofs avoid this.
- ≈53% proof reduction.

Multicopy Template (Iris/Coq)

Module	Original	Hindsight
Defs	866	–
Client-level Spec	434	–
LSM	741	540
Search	411	399
Upsert	327	371
Total	**2779**	**1310**

Acknowledgements

- **Advisors:** Prof. Thomas Wies and Prof. Dennis Shasha.
- **Committee members:** Prof. Joseph Tassarotti, Prof. Aurojit Panda & Prof. Constantin Enea.
- **Other collaborators:** Dr. Kedar Namjoshi and Dr. Siddharth Krishna

Thanks also to the rest of the NYU family!

Thank you!

- Data Structure invariants
- Proof of the method

Hindsight Specification

Data Structure Agnostic

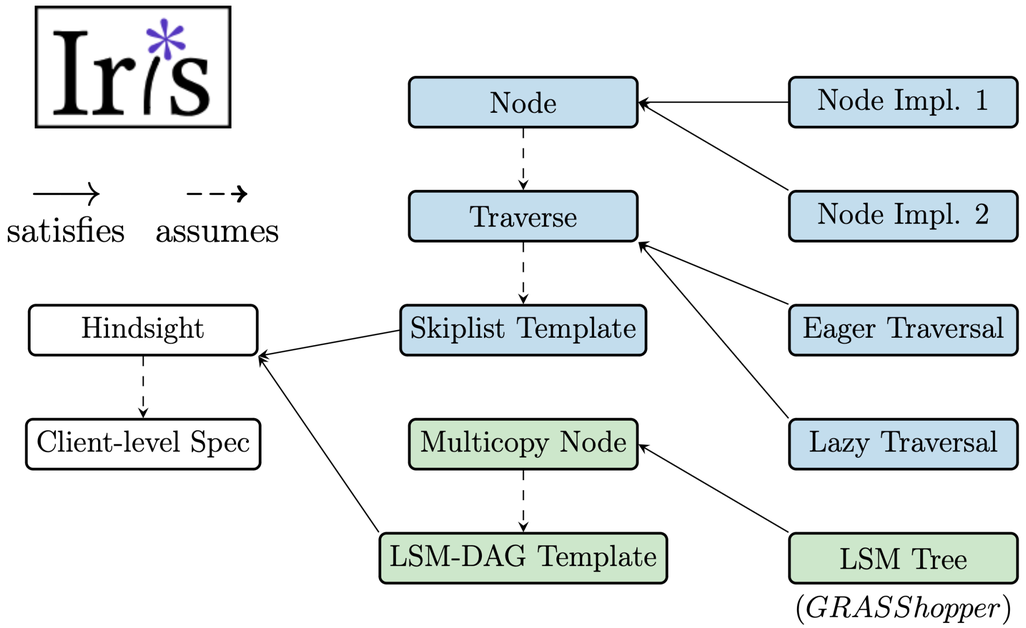
- Prophecy variables
- Helping Protocol

Client-level Specification

Template Algorithms

```
1  let search k =
2    let ps = allocArr ⊥ hd in
3    let cs = allocArr ⊥ tl in
4    let _, _, res = traverse ps cs k in
5    res
6
7  let delete k =
8    let ps = allocArr ⊥ hd in
9    let cs = allocArr ⊥ tl in
10   let p, c, res = traverse ps cs k in
11   if not res then
12     false
13   else
14     maintainanceOp_del c;
15     match markNode 0 c with
16     | Success -> traverse ps cs k; true
17     | Failure -> false
```

```
18 let insert k =
19   let ps = allocArr ⊥ hd in
20   let cs = allocArr ⊥ tl in
21   let p, c, res = traverse ps cs k in
22   if res then
23     false
24   else
25     let h = randomNum ⊥ in
26     let e = createNode k h cs in
27     match changeNext 0 p c e with
28     | Success ->
29       maintainanceOp_ins k ps cs e; true
30     | Failure -> insert k
```



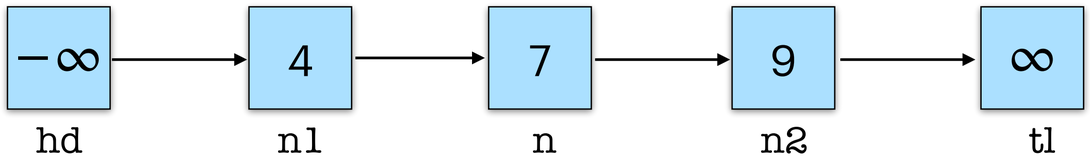
Backup Slides

Template Algorithms

```
1  let search k =
2    let ps = allocArr L hd in
3    let cs = allocArr L tl in
4    let _, _, res = traverse ps cs k in
5    res
6
7  let delete k =
8    let ps = allocArr L hd in
9    let cs = allocArr L tl in
10   let p, c, res = traverse ps cs k in
11   if not res then
12     false
13   else
14     maintainanceOp_del c;
15     match markNode 0 c with
16     | Success -> traverse ps cs k; true
17     | Failure -> false
```

```
18 let insert k =
19   let ps = allocArr L hd in
20   let cs = allocArr L tl in
21   let p, c, res = traverse ps cs k in
22   if res then
23     false
24   else
25     let h = randomNum L in
26     let e = createNode k h cs in
27     match changeNext 0 p c e with
28     | Success ->
29       maintainanceOp_ins k ps cs e; true
30     | Failure -> insert k
```

delete(7)

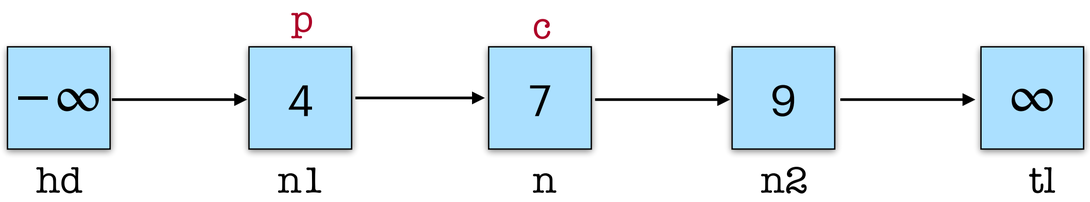


Template Algorithms

```
1  let search k =
2    let ps = allocArr L hd in
3    let cs = allocArr L tl in
4    let _, _, res = traverse ps cs k in
5    res
6
7  let delete k =
8    let ps = allocArr L hd in
9    let cs = allocArr L tl in
10   let p, c, res = traverse ps cs k in
11   if not res then
12     false
13   else
14     maintainanceOp_del c;
15     match markNode 0 c with
16     | Success -> traverse ps cs k; true
17     | Failure -> false
```

```
18 let insert k =
19   let ps = allocArr L hd in
20   let cs = allocArr L tl in
21   let p, c, res = traverse ps cs k in
22   if res then
23     false
24   else
25     let h = randomNum L in
26     let e = createNode k h cs in
27     match changeNext 0 p c e with
28     | Success ->
29       maintainanceOp_ins k ps cs e; true
30     | Failure -> insert k
```

delete(7)

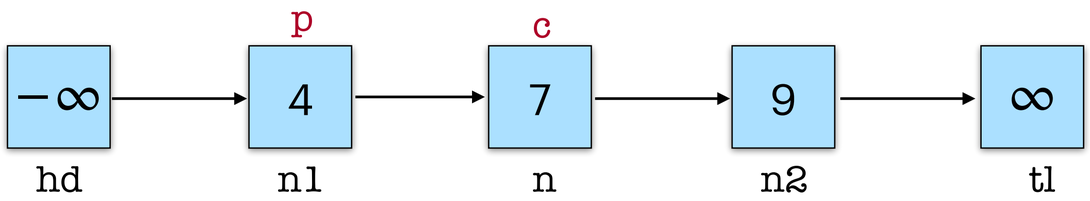


Template Algorithms

```
1  let search k =
2    let ps = allocArr L hd in
3    let cs = allocArr L tl in
4    let _, _, res = traverse ps cs k in
5    res
6
7  let delete k =
8    let ps = allocArr L hd in
9    let cs = allocArr L tl in
10   let p, c, res = traverse ps cs k in
11   if not res then
12     false
13   else
14     maintainanceOp_del c;
15     match markNode 0 c with
16     | Success -> traverse ps cs k; true
17     | Failure -> false
```

```
18 let insert k =
19   let ps = allocArr L hd in
20   let cs = allocArr L tl in
21   let p, c, res = traverse ps cs k in
22   if res then
23     false
24   else
25     let h = randomNum L in
26     let e = createNode k h cs in
27     match changeNext 0 p c e with
28     | Success ->
29       maintainanceOp_ins k ps cs e; true
30     | Failure -> insert k
```

delete(7)

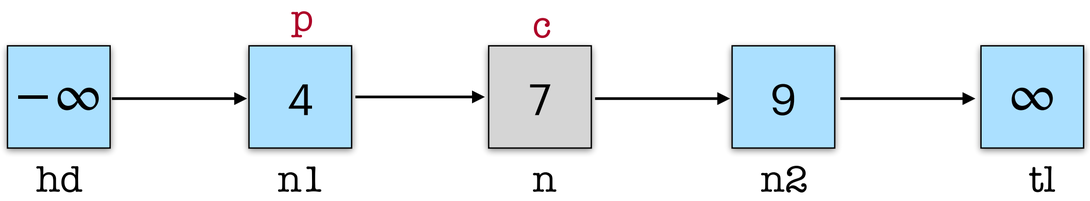


Template Algorithms

```
1  let search k =
2    let ps = allocArr L hd in
3    let cs = allocArr L tl in
4    let _, _, res = traverse ps cs k in
5    res
6
7  let delete k =
8    let ps = allocArr L hd in
9    let cs = allocArr L tl in
10   let p, c, res = traverse ps cs k in
11   if not res then
12     false
13   else
14     maintainanceOp_del c;
15     match markNode 0 c with
16     | Success -> traverse ps cs k; true
17     | Failure -> false
```

```
18 let insert k =
19   let ps = allocArr L hd in
20   let cs = allocArr L tl in
21   let p, c, res = traverse ps cs k in
22   if res then
23     false
24   else
25     let h = randomNum L in
26     let e = createNode k h cs in
27     match changeNext 0 p c e with
28     | Success ->
29       maintainanceOp_ins k ps cs e; true
30     | Failure -> insert k
```

delete(7)



Template Algorithms

```
let maintainanceOp_del_rec i h pm c =
  if i < h-1 then
    let idx = pm[i] in
    markNode idx c;
    maintainanceOp_del_rec (i+1) h pm c
  else
    ()

let maintainanceOp_del c =
  let h = getHeight c in
  let pm = permute h in
  maintainanceOp_del 0 h pm c
```

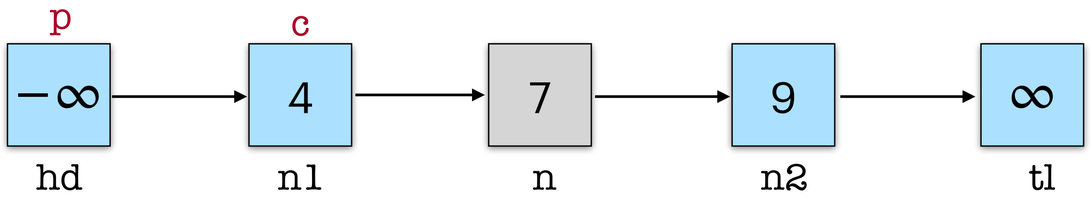
```
let maintainanceOp_ins_rec i h pm ps cs e =
  if i < h-1 then
    let idx = pm[i] in
    let p = ps[idx] in
    let c = cs[idx] in
    match changeNext idx p c e with
    | Success ->
      maintainanceOp_ins_rec (i+1) h pm ps cs e
    | Failure ->
      traverse ps cs k;
      maintainanceOp_ins_rec i h pm ps cs e
  else
    ()

let maintainanceOp_ins k ps cs e =
  let h = getHeight e in
  let pm = permute h in
  maintainanceOp_ins 0 h pm ps cs e
```

Template Algorithms

```
1  let eager_i i k p c =
2    match findNext i c with
3    | cn, true ->
4      match changeNext i p c cn with
5      | Success -> eager_i i k p cn
6      | Failure -> traverse ps cs k
7    | cn, false ->
8      let kc = getKey c in
9      if kc < k then
10       eager_i i k c cn
11     else
12       let res = (kc = k ? true : false) in
13       (p, c, res)
```

```
14 let eager_rec i ps cs k =
15   let p = ps[i+1] in
16   let c, _ = findNext i p in
17   let p', c', res = eager_i i k p c in
18   ps[i] <- p';
19   cs[i] <- c';
20   if i = 0 then
21     (p', c', res)
22   else
23   eager_rec (i-1) ps cs k
24
25 let traverse ps cs k =
26   eager_rec (L - 2) ps cs k
```

search(9)

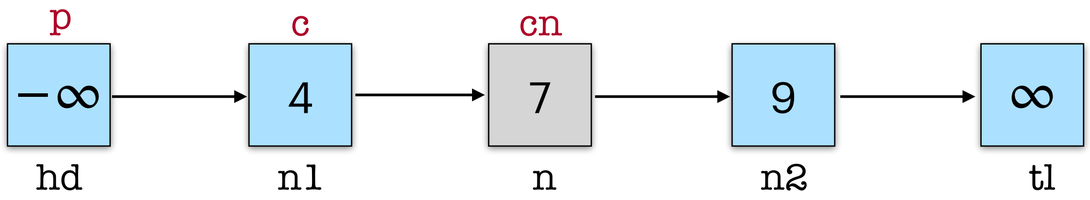


Template Algorithms

```
1  let eager_i i k p c =
2    match findNext i c with
3    | cn, true ->
4      match changeNext i p c cn with
5      | Success -> eager_i i k p cn
6      | Failure -> traverse ps cs k
7    | cn, false ->
8      let kc = getKey c in
9      if kc < k then
10       eager_i i k c cn
11     else
12       let res = (kc = k ? true : false) in
13       (p, c, res)
```

```
14 let eager_rec i ps cs k =
15   let p = ps[i+1] in
16   let c, _ = findNext i p in
17   let p', c', res = eager_i i k p c in
18   ps[i] <- p';
19   cs[i] <- c';
20   if i = 0 then
21     (p', c', res)
22   else
23   eager_rec (i-1) ps cs k
24
25 let traverse ps cs k =
26   eager_rec (L - 2) ps cs k
```

search(9)

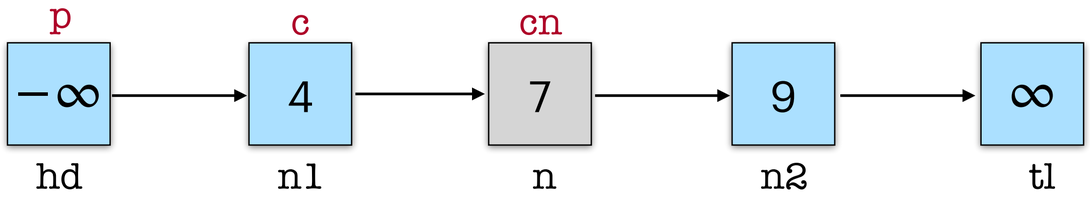


Template Algorithms

```
1  let eager_i i k p c =
2    match findNext i c with
3    | cn, true ->
4      match changeNext i p c cn with
5      | Success -> eager_i i k p cn
6      | Failure -> traverse ps cs k
7    | cn, false ->
8      let kc = getKey c in
9      if kc < k then
10       eager_i i k c cn
11     else
12       let res = (kc = k ? true : false) in
13       (p, c, res)
```

```
14 let eager_rec i ps cs k =
15   let p = ps[i+1] in
16   let c, _ = findNext i p in
17   let p', c', res = eager_i i k p c in
18   ps[i] <- p';
19   cs[i] <- c';
20   if i = 0 then
21     (p', c', res)
22   else
23   eager_rec (i-1) ps cs k
24
25 let traverse ps cs k =
26   eager_rec (L - 2) ps cs k
```

search(9)

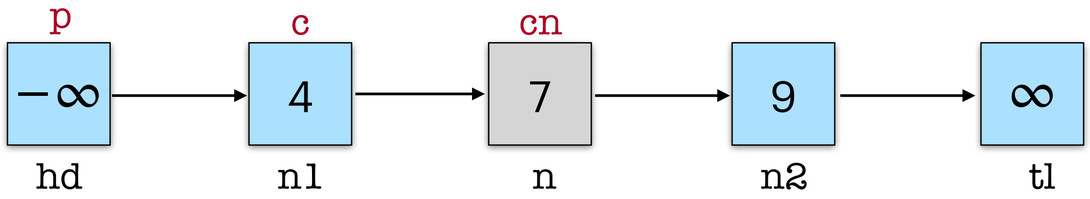


Template Algorithms

```
1  let eager_i i k p c =
2    match findNext i c with
3    | cn, true ->
4      match changeNext i p c cn with
5      | Success -> eager_i i k p cn
6      | Failure -> traverse ps cs k
7    | cn, false ->
8      let kc = getKey c in
9      if kc < k then
10       eager_i i k c cn
11     else
12       let res = (kc = k ? true : false) in
13       (p, c, res)
```

```
14 let eager_rec i ps cs k =
15   let p = ps[i+1] in
16   let c, _ = findNext i p in
17   let p', c', res = eager_i i k p c in
18   ps[i] <- p';
19   cs[i] <- c';
20   if i = 0 then
21     (p', c', res)
22   else
23   eager_rec (i-1) ps cs k
24
25 let traverse ps cs k =
26   eager_rec (L - 2) ps cs k
```

search(9)

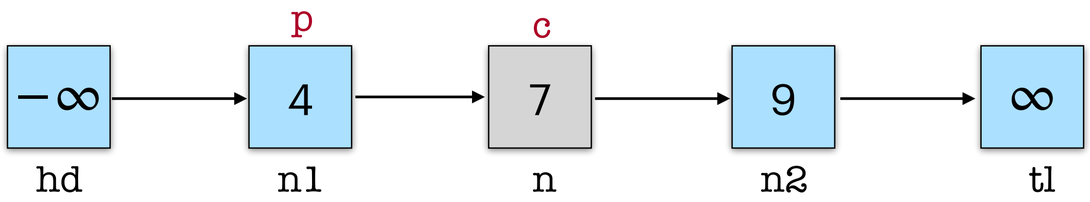


Template Algorithms

```
1  let eager_i i k p c =
2    match findNext i c with
3    | cn, true ->
4      match changeNext i p c cn with
5      | Success -> eager_i i k p cn
6      | Failure -> traverse ps cs k
7    | cn, false ->
8      let kc = getKey c in
9      if kc < k then
10       eager_i i k c cn
11     else
12       let res = (kc = k ? true : false) in
13       (p, c, res)
```

```
14 let eager_rec i ps cs k =
15   let p = ps[i+1] in
16   let c, _ = findNext i p in
17   let p', c', res = eager_i i k p c in
18   ps[i] <- p';
19   cs[i] <- c';
20   if i = 0 then
21     (p', c', res)
22   else
23   eager_rec (i-1) ps cs k
24
25 let traverse ps cs k =
26   eager_rec (L - 2) ps cs k
```

search(9)

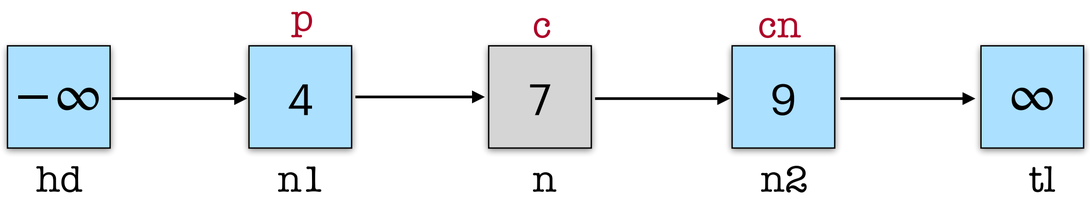


Template Algorithms

```
1  let eager_i i k p c =
2    match findNext i c with
3    | cn, true ->
4      match changeNext i p c cn with
5      | Success -> eager_i i k p cn
6      | Failure -> traverse ps cs k
7    | cn, false ->
8      let kc = getKey c in
9      if kc < k then
10       eager_i i k c cn
11     else
12       let res = (kc = k ? true : false) in
13       (p, c, res)
```

```
14 let eager_rec i ps cs k =
15   let p = ps[i+1] in
16   let c, _ = findNext i p in
17   let p', c', res = eager_i i k p c in
18   ps[i] <- p';
19   cs[i] <- c';
20   if i = 0 then
21     (p', c', res)
22   else
23   eager_rec (i-1) ps cs k
24
25 let traverse ps cs k =
26   eager_rec (L - 2) ps cs k
```

search(9)

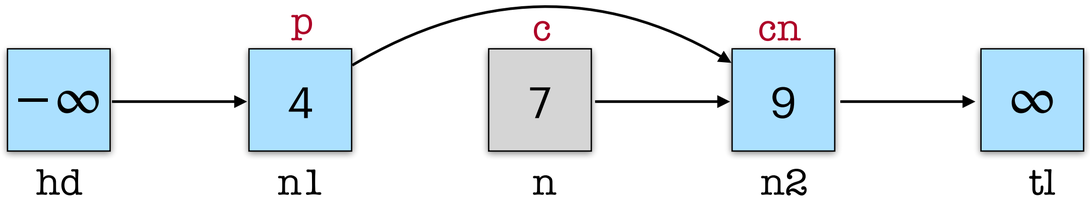


Template Algorithms

```
1  let eager_i i k p c =
2    match findNext i c with
3    | cn, true ->
4      match changeNext i p c cn with
5      | Success -> eager_i i k p cn
6      | Failure -> traverse ps cs k
7    | cn, false ->
8      let kc = getKey c in
9      if kc < k then
10       eager_i i k c cn
11     else
12       let res = (kc = k ? true : false) in
13       (p, c, res)
```

```
14 let eager_rec i ps cs k =
15   let p = ps[i+1] in
16   let c, _ = findNext i p in
17   let p', c', res = eager_i i k p c in
18   ps[i] <- p';
19   cs[i] <- c';
20   if i = 0 then
21     (p', c', res)
22   else
23   eager_rec (i-1) ps cs k
24
25 let traverse ps cs k =
26   eager_rec (L - 2) ps cs k
```

search(9)

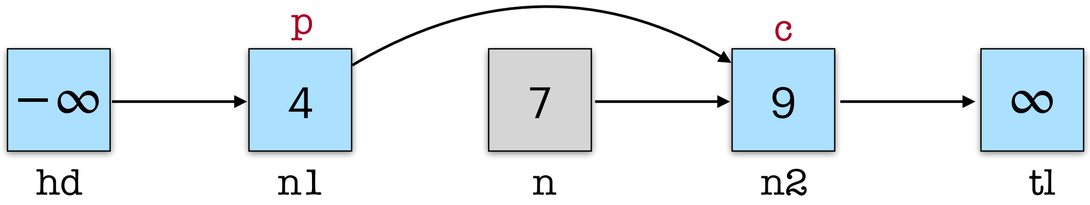


Template Algorithms

```
1  let eager_i i k p c =
2    match findNext i c with
3    | cn, true ->
4      match changeNext i p c cn with
5      | Success -> eager_i i k p cn
6      | Failure -> traverse ps cs k
7    | cn, false ->
8      let kc = getKey c in
9      if kc < k then
10       eager_i i k c cn
11     else
12       let res = (kc = k ? true : false) in
13       (p, c, res)
```

```
14 let eager_rec i ps cs k =
15   let p = ps[i+1] in
16   let c, _ = findNext i p in
17   let p', c', res = eager_i i k p c in
18   ps[i] <- p';
19   cs[i] <- c';
20   if i = 0 then
21     (p', c', res)
22   else
23   eager_rec (i-1) ps cs k
24
25 let traverse ps cs k =
26   eager_rec (L - 2) ps cs k
```

search(9)

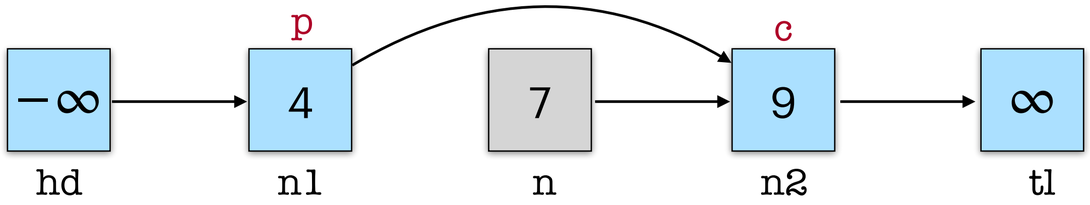


Template Algorithms

```
1  let eager_i i k p c =
2    match findNext i c with
3    | cn, true ->
4      match changeNext i p c cn with
5      | Success -> eager_i i k p cn
6      | Failure -> traverse ps cs k
7    | cn, false ->
8      let kc = getKey c in
9      if kc < k then
10       eager_i i k c cn
11     else
12       let res = (kc = k ? true : false) in
13       (p, c, res)
```

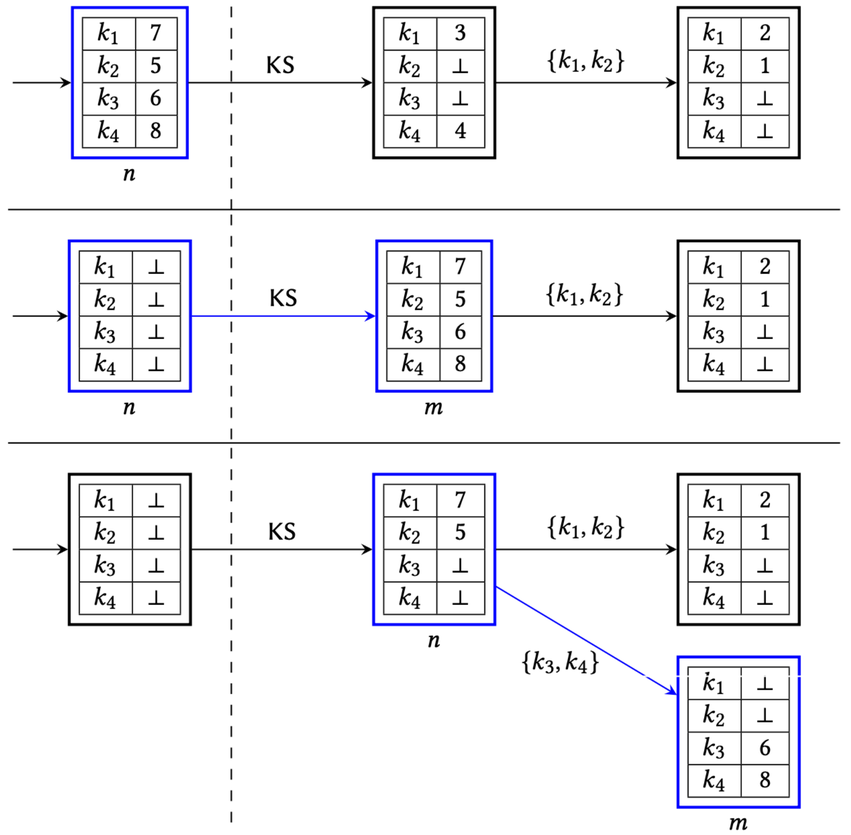
```
14 let eager_rec i ps cs k =
15   let p = ps[i+1] in
16   let c, _ = findNext i p in
17   let p', c', res = eager_i i k p c in
18   ps[i] <- p';
19   cs[i] <- c';
20   if i = 0 then
21     (p', c', res)
22   else
23   eager_rec (i-1) ps cs k
24
25 let traverse ps cs k =
26   eager_rec (L - 2) ps cs k
```

search(9)

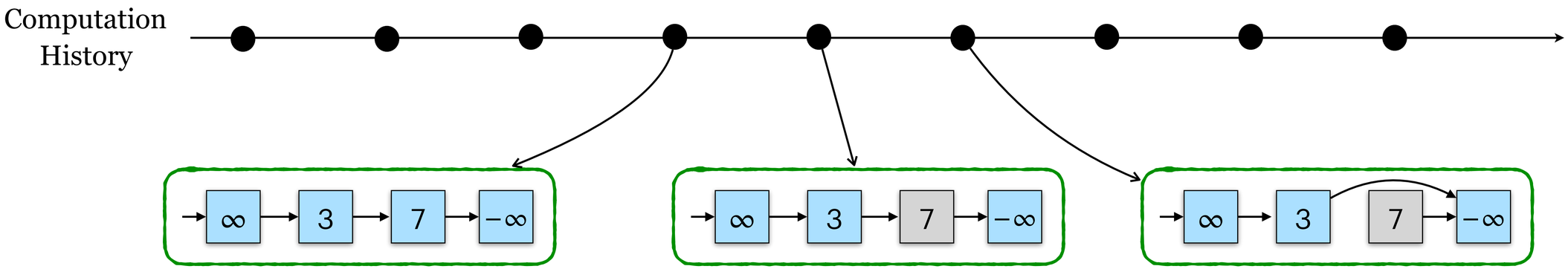


Templates Algorithms (Multicopy)

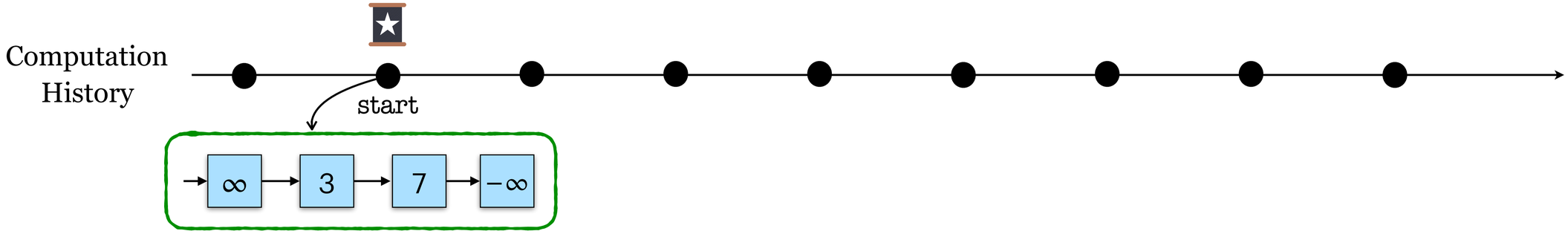
```
1  let rec compact r n =
2    lockNode n;
3    if atCapacity r n then begin
4      match chooseNext r n with
5      | Some m ->
6        lockNode m;
7        mergeContents r n m;
8        unlockNode n;
9        unlockNode m;
10       compact r m
11     | None ->
12       let m = allocNode () in
13       insertNode r n m;
14       mergeContents r n m;
15       unlockNode n;
16       unlockNode m;
17       compact r m
18   end
19   else
20     unlock n
```



Helping Protocol

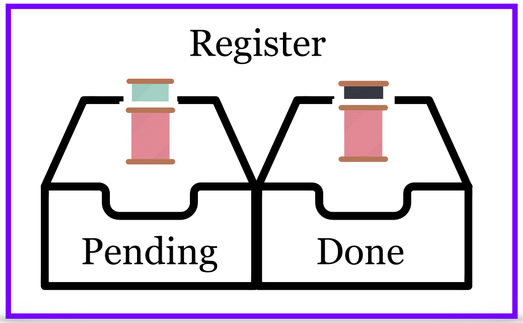


Helping Protocol

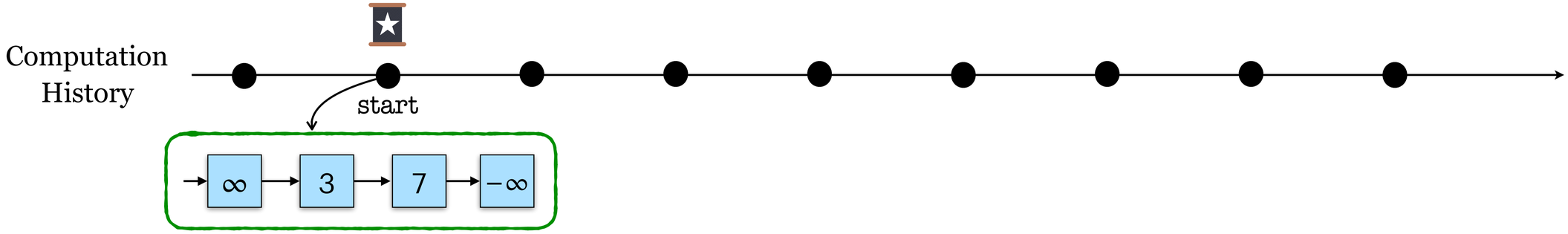


search(9)

Pr(No-upd, true)

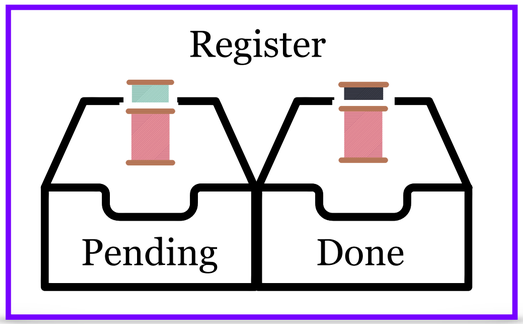


Helping Protocol



search(9)

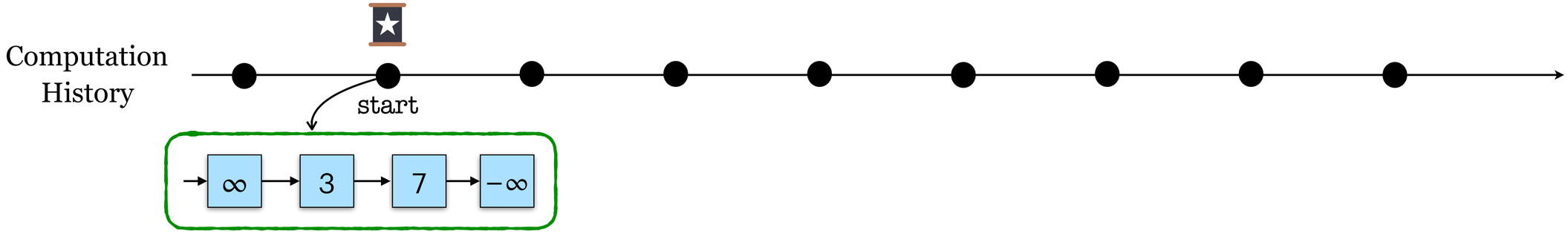
Pr(No-upd, true)



Pending(t_{id}) := Forall t between start(t_{id}) and present, seq. spec of t_{id} does not hold.

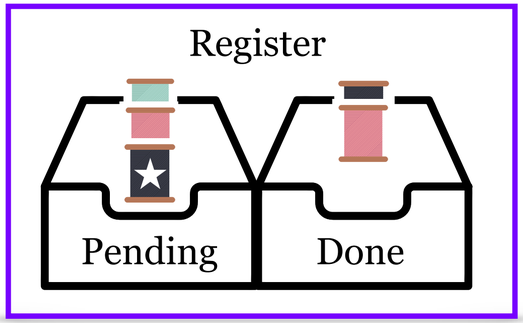
Done(t_{id}) := Exists t between start(t_{id}) and present, seq. spec of t_{id} does hold.

Helping Protocol

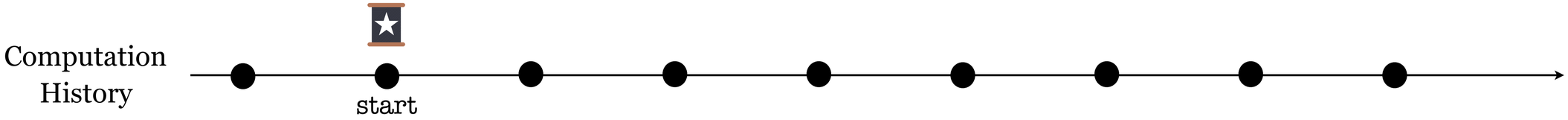


search(9)

Pr(No-upd, true)

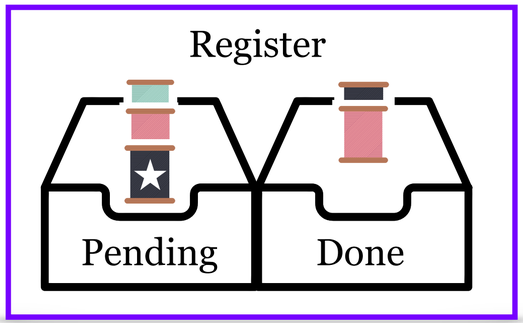


Helping Protocol

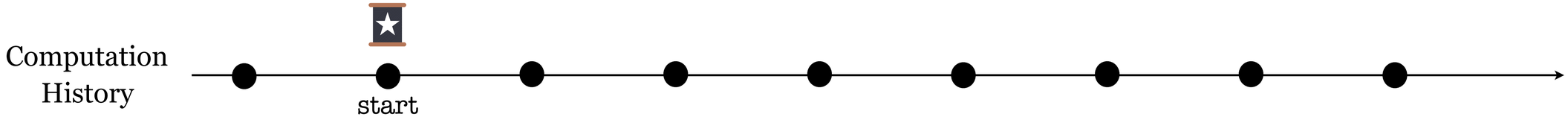


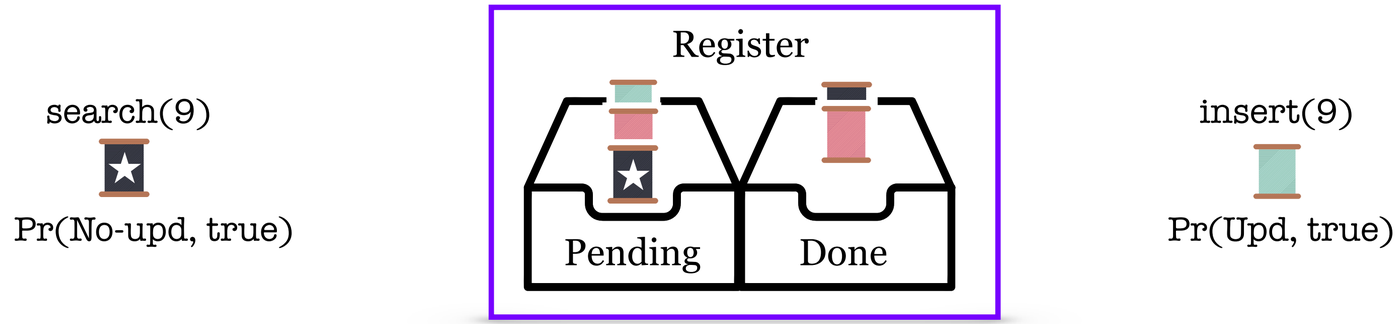
search(9)

Pr(No-upd, true)

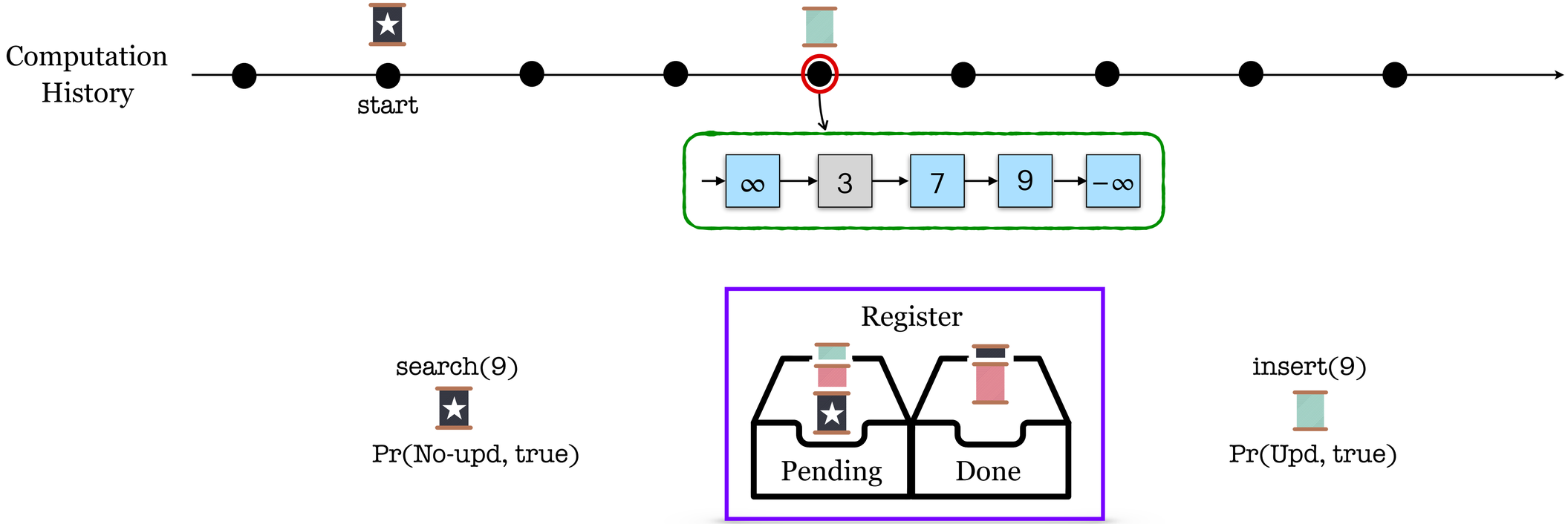


Helping Protocol

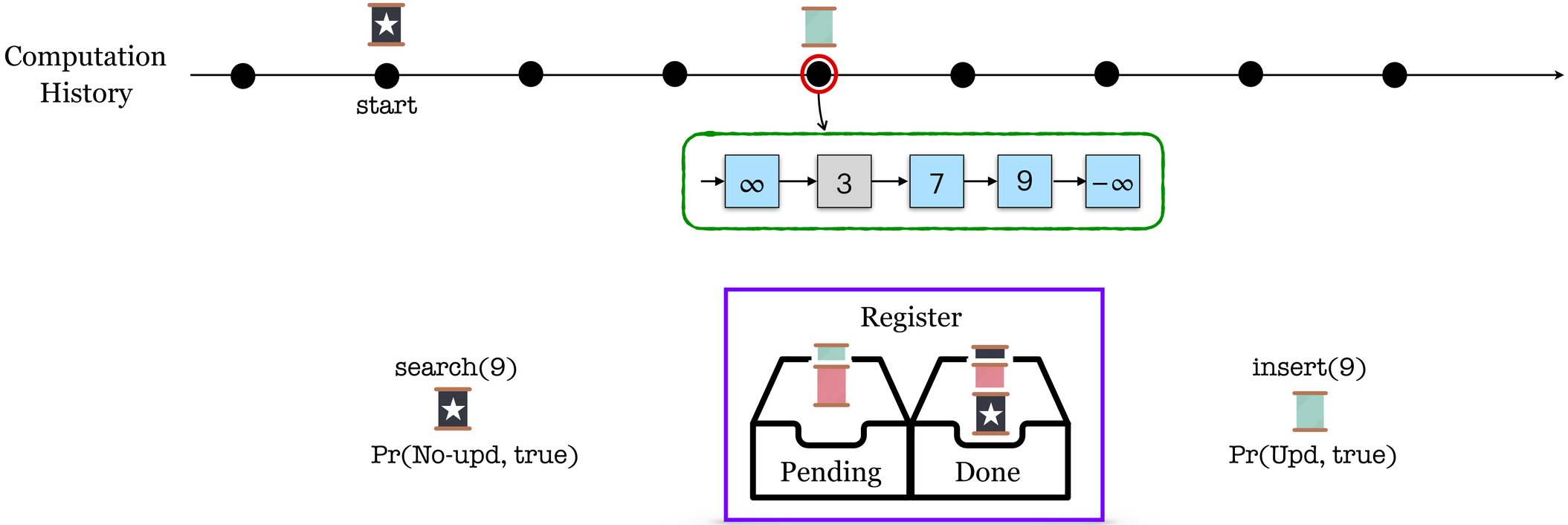




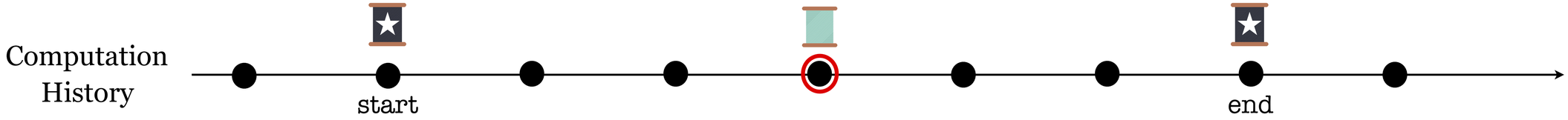
Helping Protocol



Helping Protocol



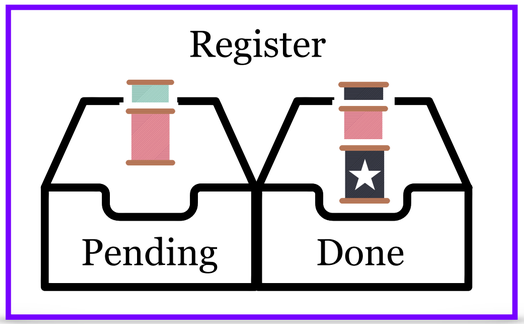
Helping Protocol



How does it know it is **Done**?
Hindsight Specification!

search(9)

Pr(No-upd, true)



insert(9)

Pr(Upd, true)

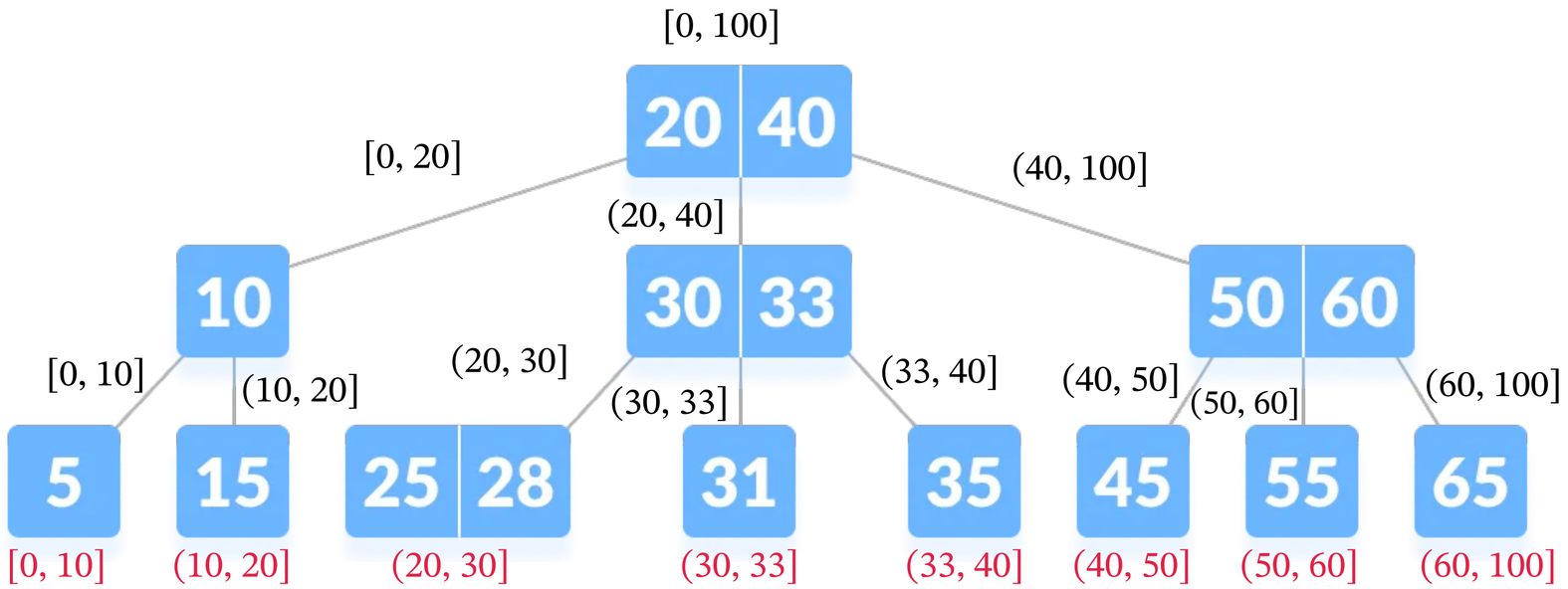
Hindsight Framework

Hindsight Specification :

- Precondition : Modifying LP ⟶ Postcondition : Receipt of linearization
- Precondition : Unmodifying LP ⟶ Postcondition : at some point during the execution, $\Psi(\mathtt{op}, \mathtt{k}, \mathtt{C}, \mathtt{C}', \mathtt{res})$ was true

Framework provides:	**Proof author obligations:**
• Prophecy instantiation	• Determine steps that potentially change the abstract state
• Helping protocol	• Define a "snapshot" of the data structure and provide invariants
• Mechanism for storing history of computation	• Prove the hindsight specification for each operation

Keysets



$$k \in \text{keyset}(n) \rightarrow (k \in C(n) \leftrightarrow k \in C)$$

Expressed using the Flow Framework [POPL18, ESOP20, PLDI20]

Template Algorithms

```
1  let search k =
2    let ps = allocArr L hd in
3    let cs = allocArr L tl in
4    let _, _, res = traverse ps cs k in
5    res
6
7  let delete k =
8    let ps = allocArr L hd in
9    let cs = allocArr L tl in
10   let p, c, res = traverse ps cs k in
11   if not res then
12     false
13   else
14     maintainanceOp_del c;
15     match markNode 0 c with
16     | Success -> traverse ps cs k; true
17     | Failure -> false
```

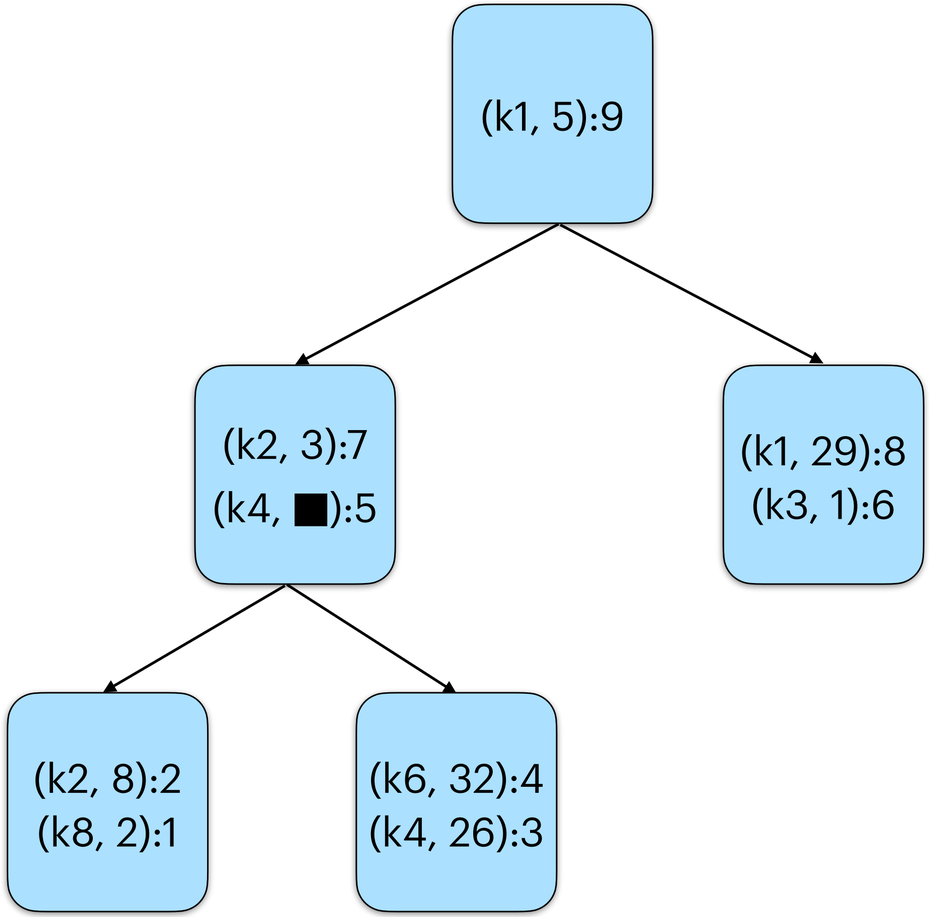
```
18 let insert k =
19   let ps = allocArr L hd in
20   let cs = allocArr L tl in
21   let p, c, res = traverse ps cs k in
22   if res then
23     false
24   else
25     let h = randomNum L in
26     let e = createNode k h cs in
27     match changeNext 0 p c e with
28     | Success ->
29       maintainanceOp_ins k ps cs e; true
30     | Failure -> insert k
```

Approach :

1. Verify the templates assuming the specification traverse, maintenance and helper functions.
2. Instantiate traverse, etc. and show they satisfy the required specifications.

   { Node(n, k, m, n') } markNode 0 n { Node(n, k, m[0 ↦ true], n') }

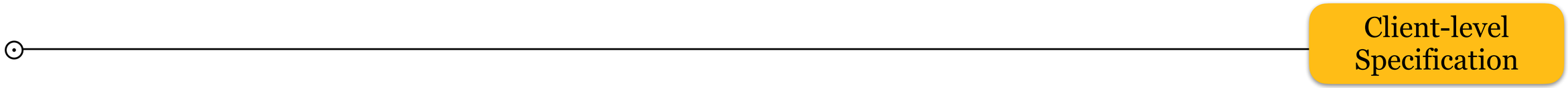
Multicopy Search Structures



Hindsight Specification

Proof Author POV

Framework provides:



Proof author obligations:

Proof Author POV

Framework provides:

Proof author obligations:

Proof Author POV

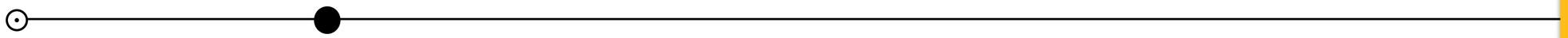
Framework provides:

Proof author obligations:

Proof Author POV

Framework provides:

0: Shared state invariant for storing history



Client-level Specification

1: Determine steps that may change the abstract state

Proof author obligations:

```
let delete k =
  let ps = allocArr L hd in
  let cs = allocArr L tl in
  let p, c, res = traverse ps cs k in
  if not res then
    false
  else
    maintainanceOp del c;
    match markNode 0 c with
    | Success -> traverse ps cs k; true
    | Failure -> false
```

```
let insert k =
  let ps = allocArr L hd in
  let cs = allocArr L tl in
  let p, c, res = traverse ps cs k in
  if res then
    false
  else
    let h = randomNum L in
    let e = createNode k h cs in
    match changeNext 0 p c e with
    | Success ->
      maintainanceOp_ins k ps cs e; true
    | Failure -> insert k
```

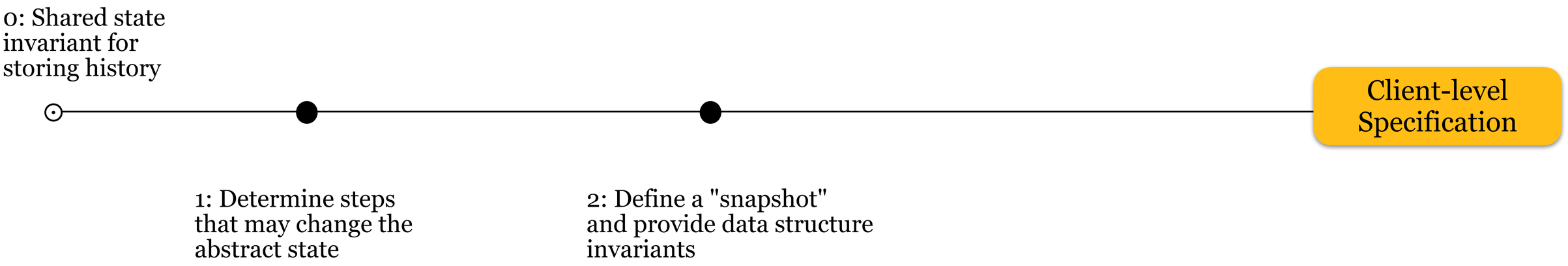
Proof Author POV

Framework provides:

Proof author obligations:

Proof Author POV

Framework provides:



Proof author obligations:

Proof Author POV

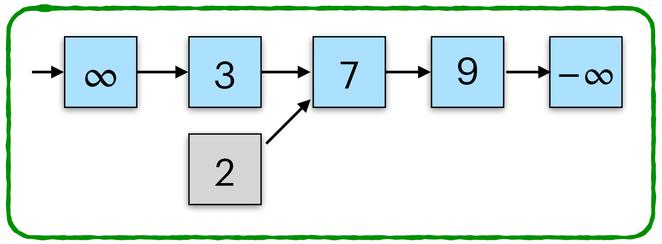
Framework provides:

0: Shared state invariant for storing history

Client-level Specification

1: Determine steps that may change the abstract state

2: Define a "snapshot" and provide data structure invariants

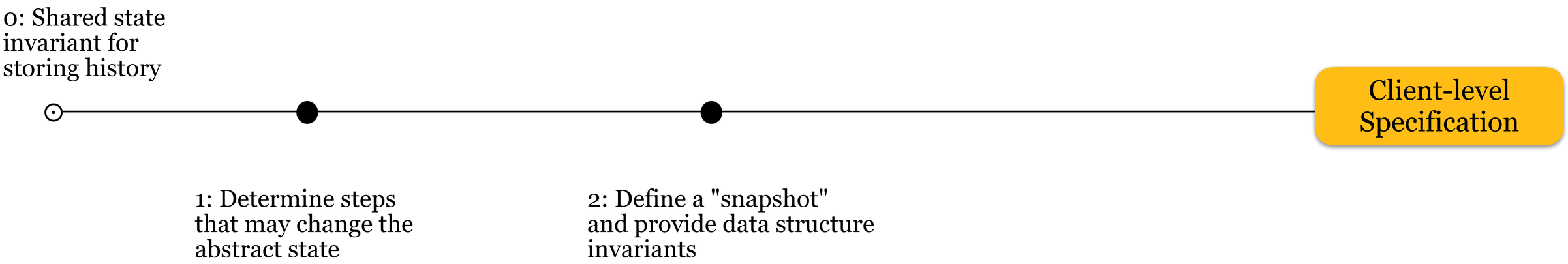


1. A node once marked remains marked.
2. The key of a node key never changes.
3. hd-list is sorted.
....

Proof author obligations:

Proof Author POV

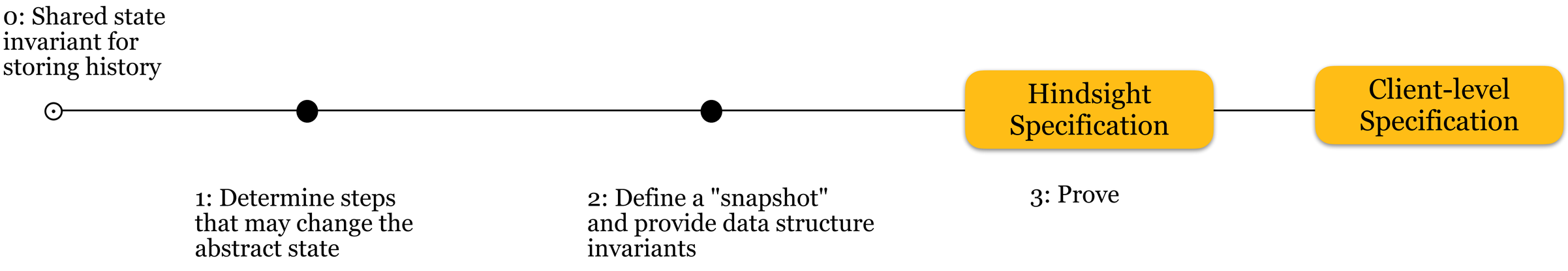
Framework provides:



Proof author obligations:

Proof Author POV

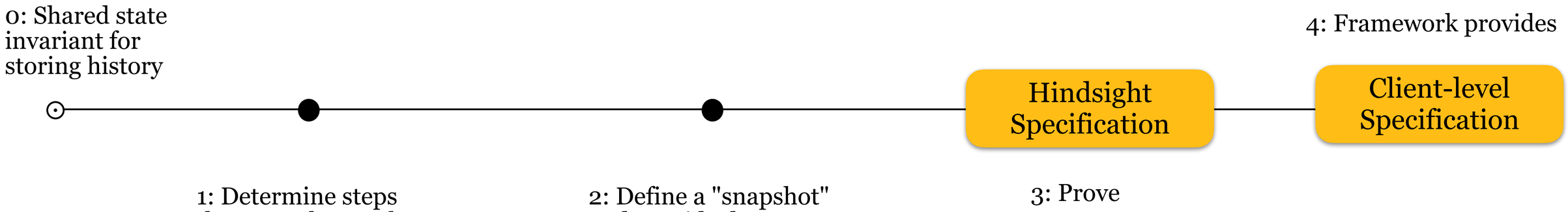
Framework provides:



Proof author obligations:

Proof Author POV

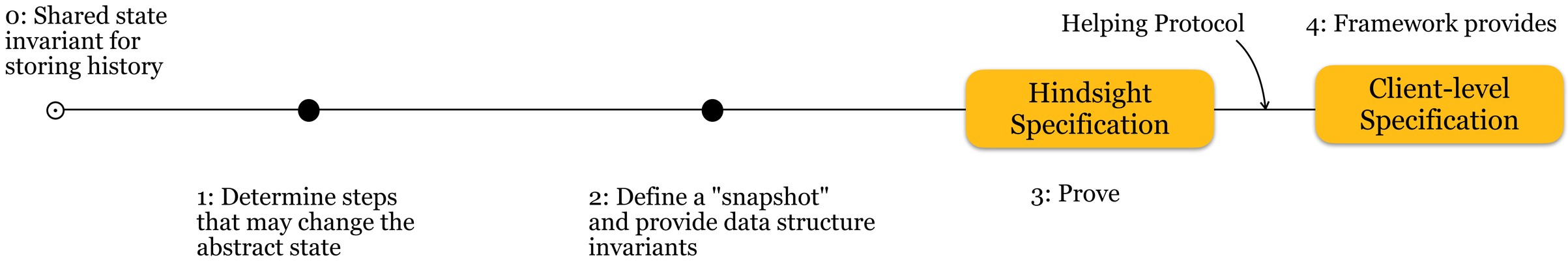
Framework provides:



Proof author obligations:

Proof Author POV

Framework provides:



Proof author obligations:

Proof Author POV

Framework provides:

Proof author obligations:

Proof Author POV

Framework provides:

Proof author obligations:

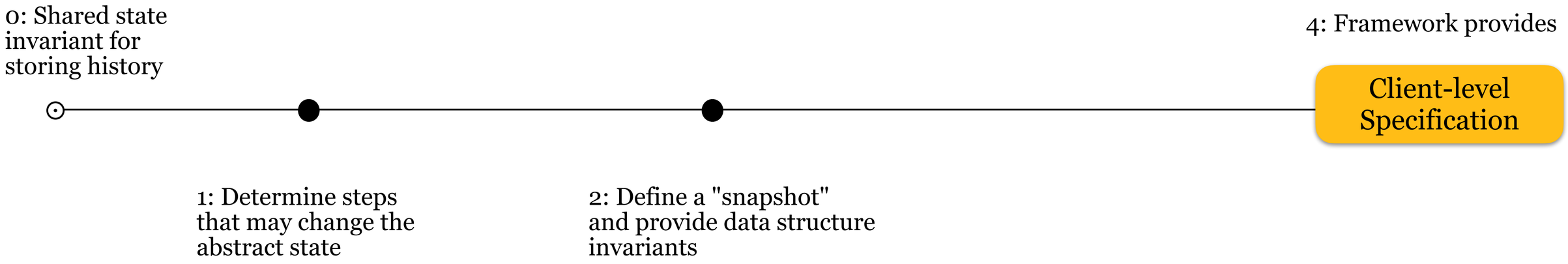
Proof Author POV

Framework provides:

Proof author obligations:

Proof Author POV

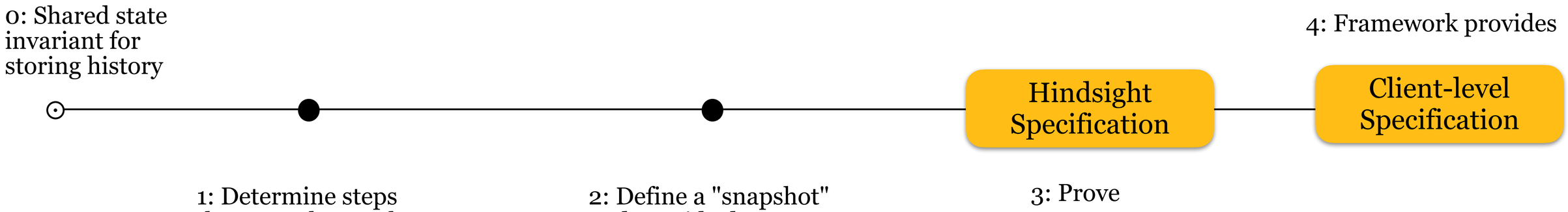
Framework provides:



Proof author obligations:

Proof Author POV

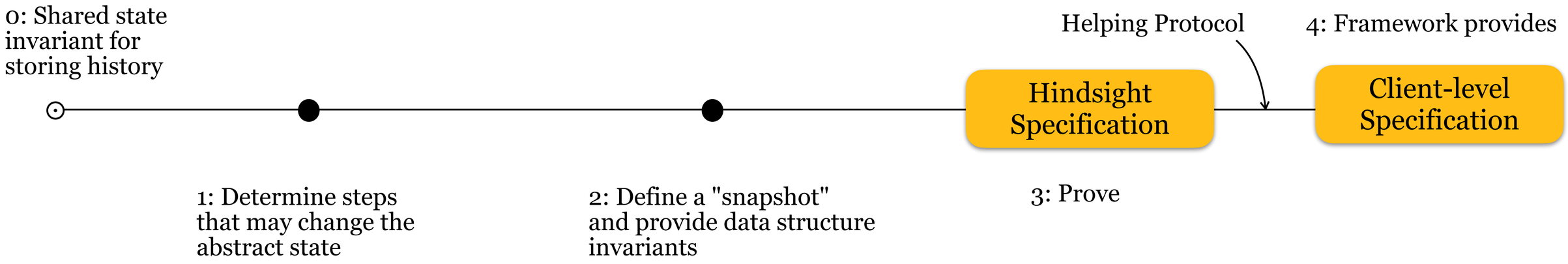
Framework provides:



Proof author obligations:

Proof Author POV

Framework provides:



Proof author obligations:

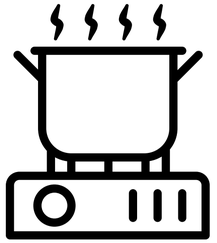
Recipe for modular verification

Step 1:
Find a class of structures with common correctness reasoning

- PLDI20 : (Lock-based, single copy) B-trees, Hash-tables, linked lists

Step 2:
Develop enabling technology

- Template Algorithms
- Edgeset Framework
- Flow Framework



Step 3:
Formalize the proof

- Resource Algebras
- Supports proof modularity

- Siddharth Krishna et al. *Verifying concurrent search structure templates*. [PLDI 2020]