# Verifying Lock-free Search Structure Templates

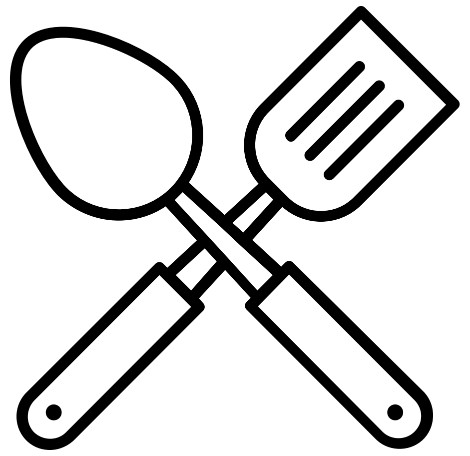
Nisarg Patel
(with Dennis Shasha and Thomas Wies)

# Recipe for modular verification

Step 1:

Find a class of structures with common correctness reasoning

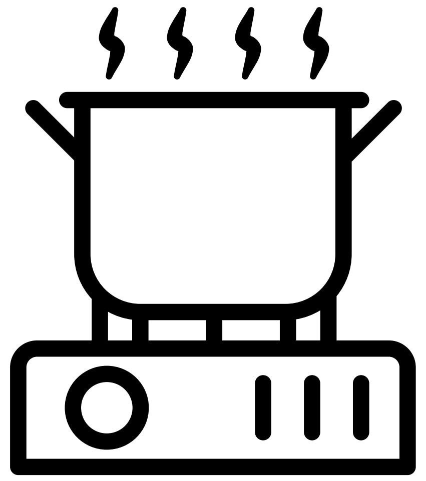
- PLDI20 : (Lock-based, single copy) B-trees, Hash-tables, linked lists
- OOPSLA21 : (Lock-based, multicopy) Log-Structured Merge (LSM) Trees



Step 2:

Develop enabling technology

- Template Algorithms
- Edgeset Framework
- Flow Framework



Step 3:

Formalize the proof

- Resource Algebras
- Supports proof modularity

Iris

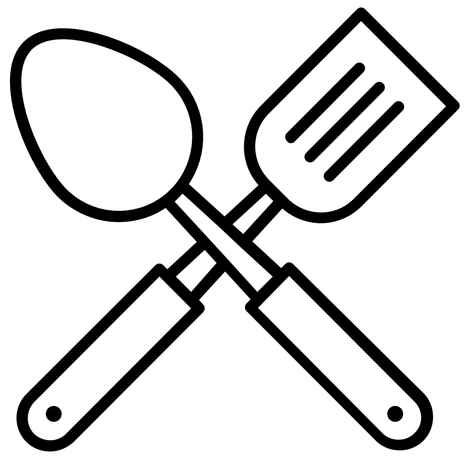

- Siddharth Krishna et al. *Verifying concurrent search structure templates*. [PLDI 2020]
- Nisarg Patel et al. *Verifying concurrent multicopy search structures*. [OOPSLA 2021]

# Outline

Step 1:

Find a class of structures with common correctness reasoning

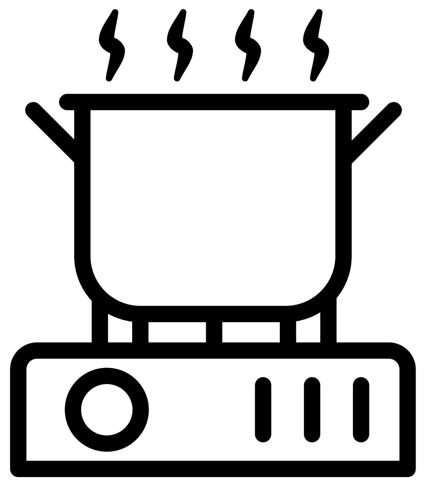
- ECOOP24 : (Lock-free) linked lists and skiplists



Step 2:

Develop enabling technology

- Template Algorithms
- Hindsight Framework



Step 3:

Formalize the proof

- Evaluation

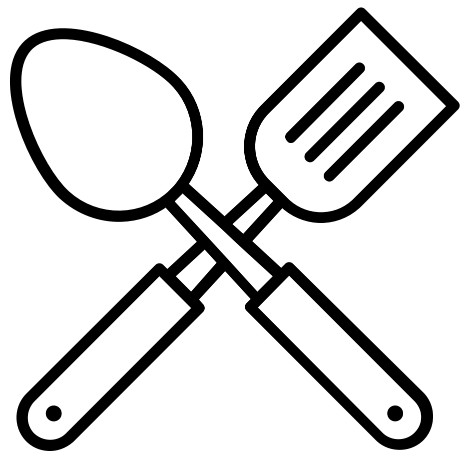
- Nisarg Patel, Dennis E. Shasha and Thomas Wies. *Verifying lock-free search structure templates*. [ECOOP 2024]

# Outline

Step 1:

Find a class of structures with common correctness reasoning

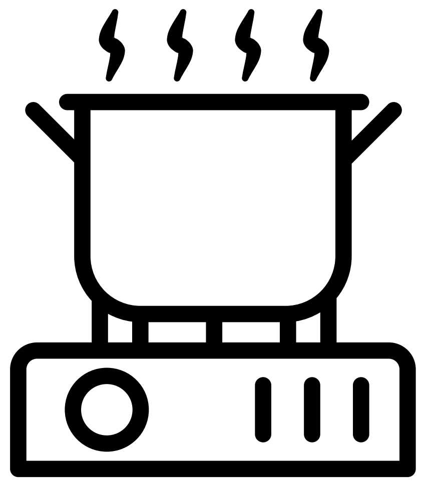
- ECOOP24 : (Lock-free) linked lists and skiplists



Step 2:

Develop enabling technology

- Template Algorithms
- Hindsight Framework

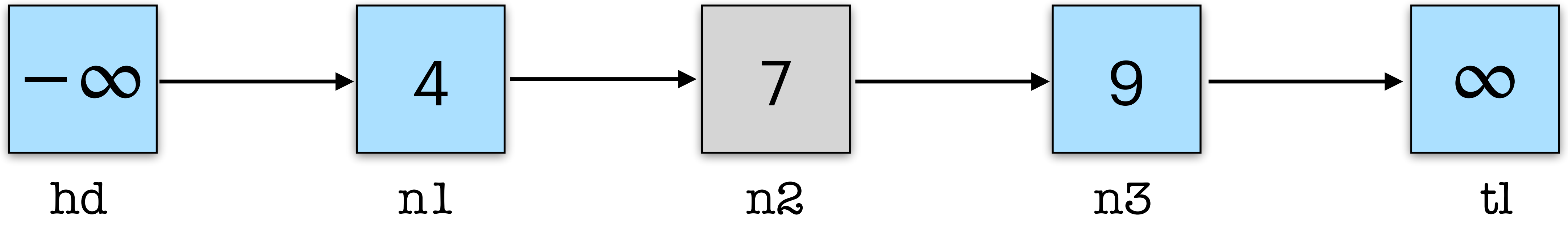


Step 3:

Formalize the proof

- Evaluation

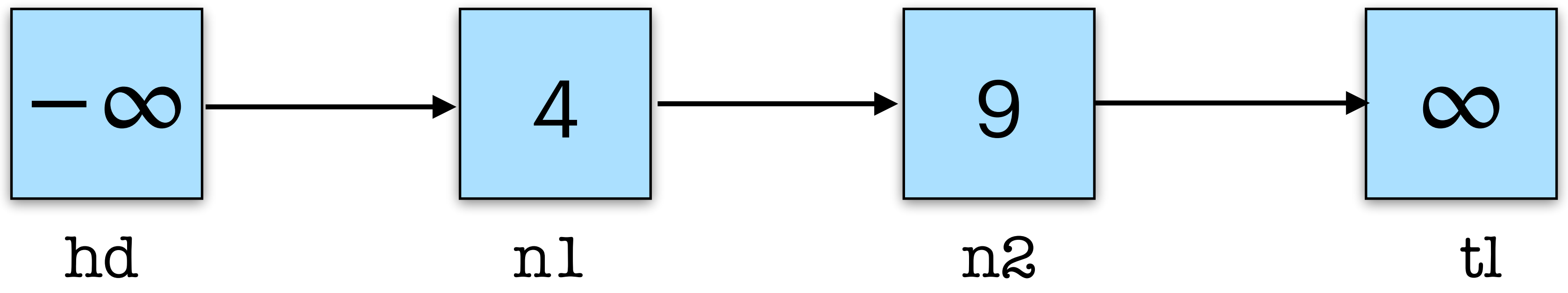
# Michael's Set



# 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)
```

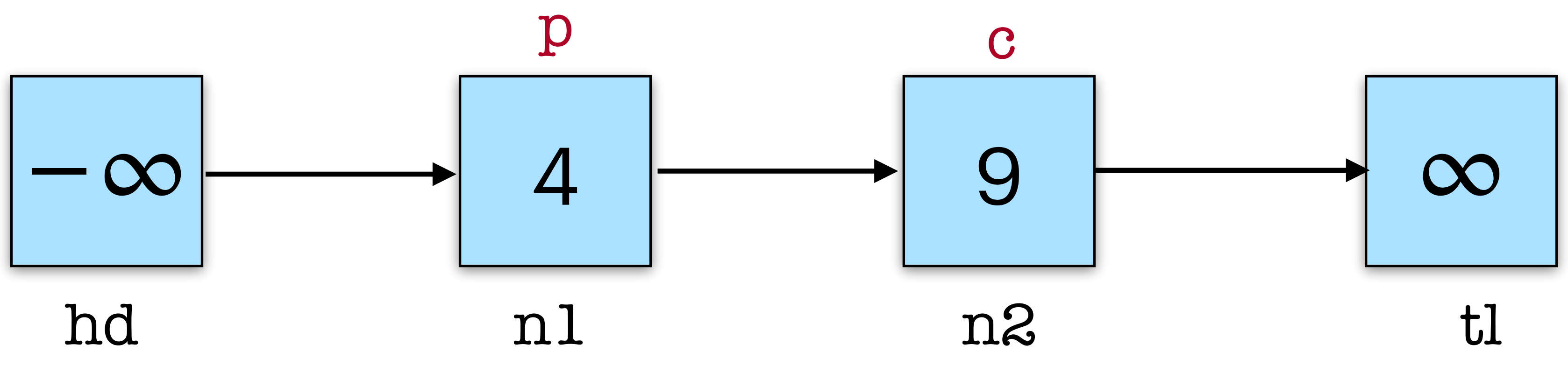
insert(7)



# 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)
```

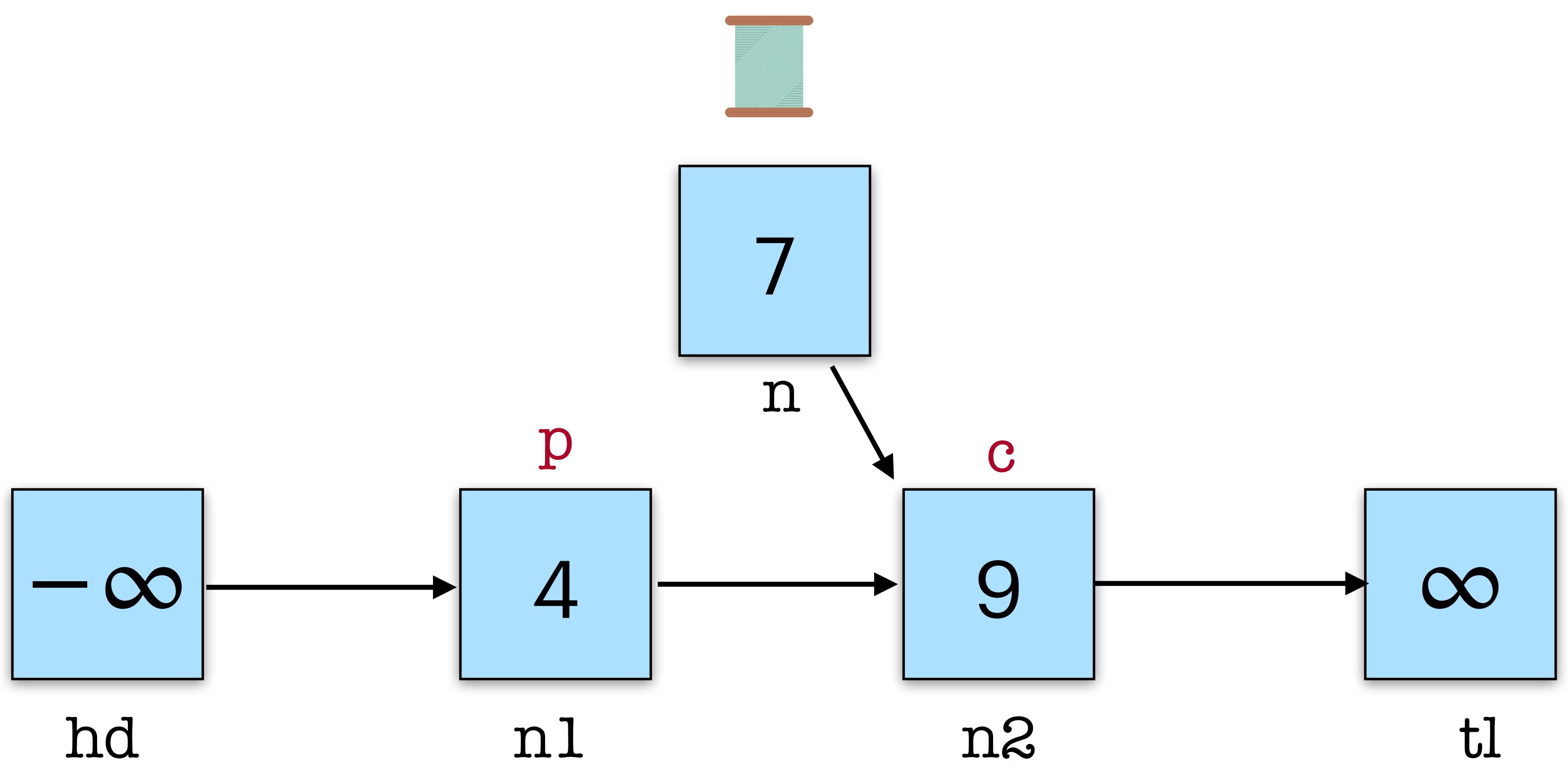
insert(7)



# 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)
```

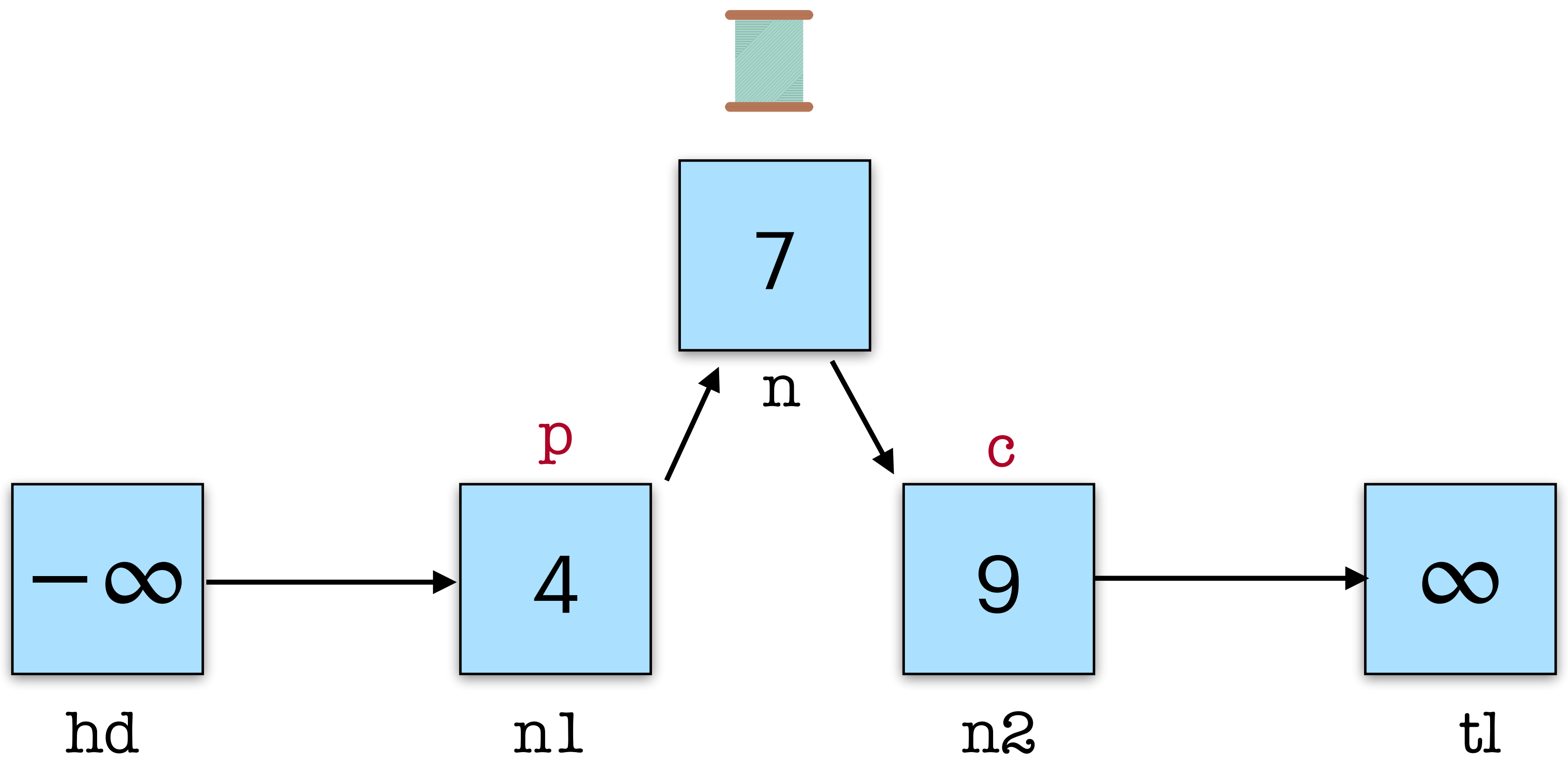
insert(7)



# 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)

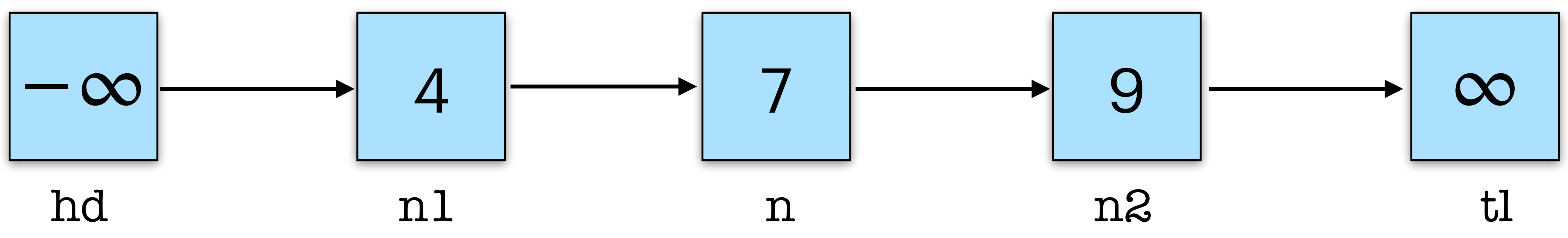


# Michael's Set

```
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
```

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) =

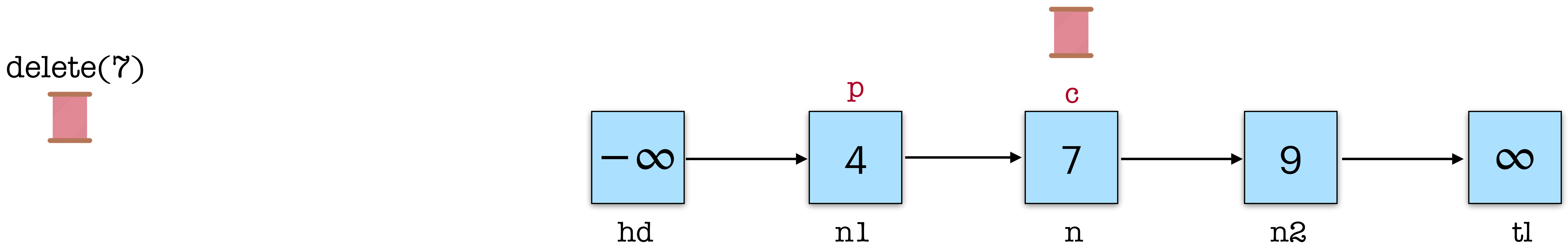
p, c, res = find(k);

**if** (not res) **then** false

**else**

**if** MARK(c)

**then** true **else** false

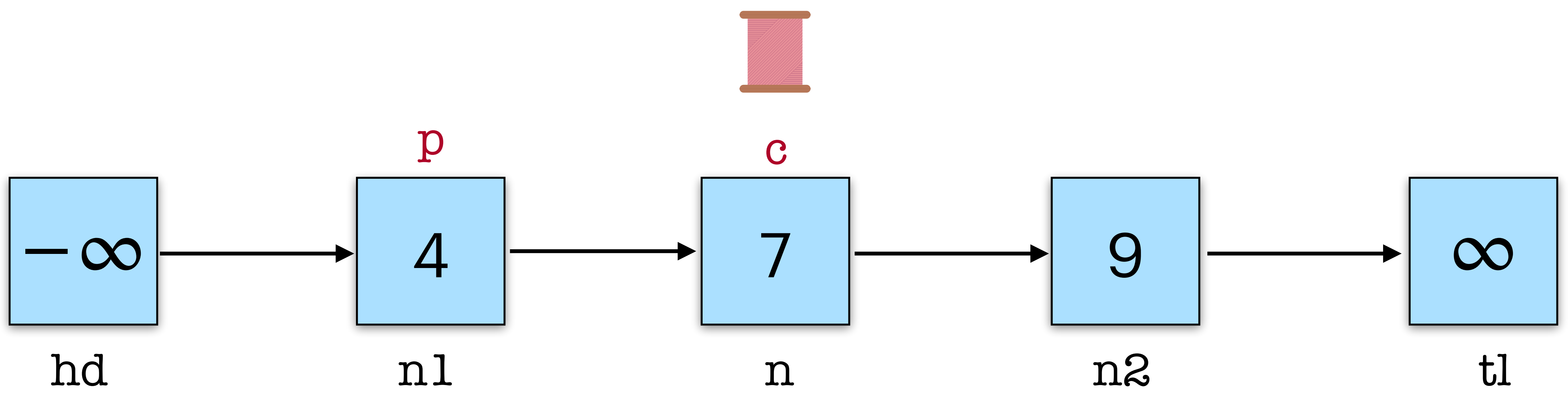


# Michael's Set

```
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
```

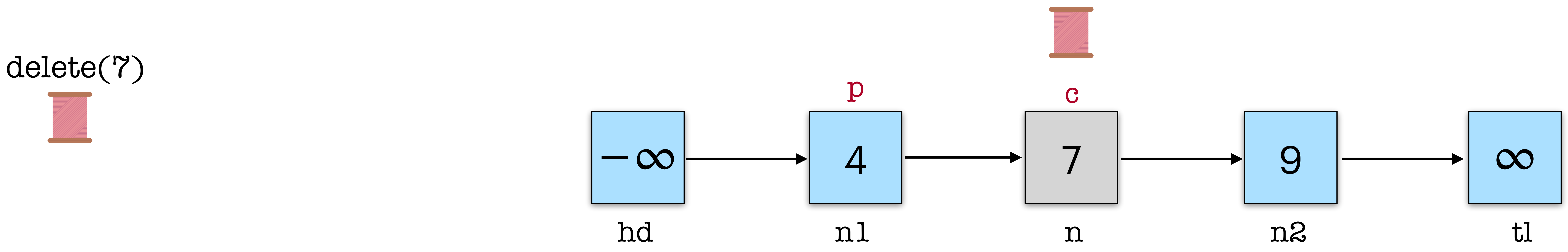
delete(7)



# 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
```



# Michael's Set

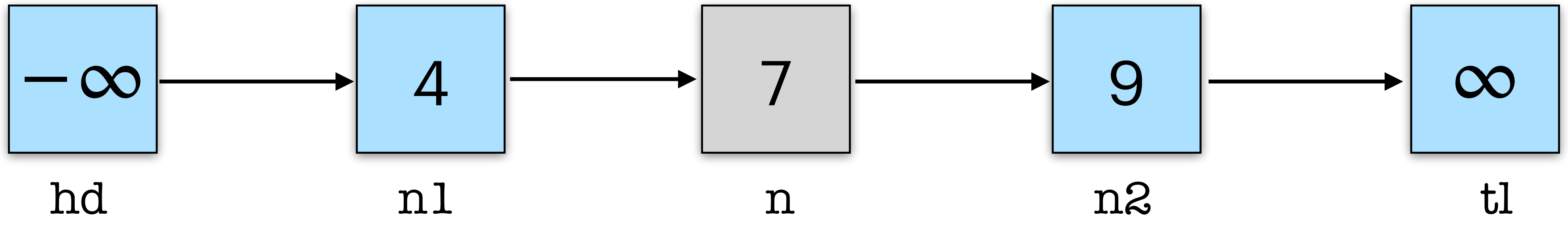
```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

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

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

```
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(9)



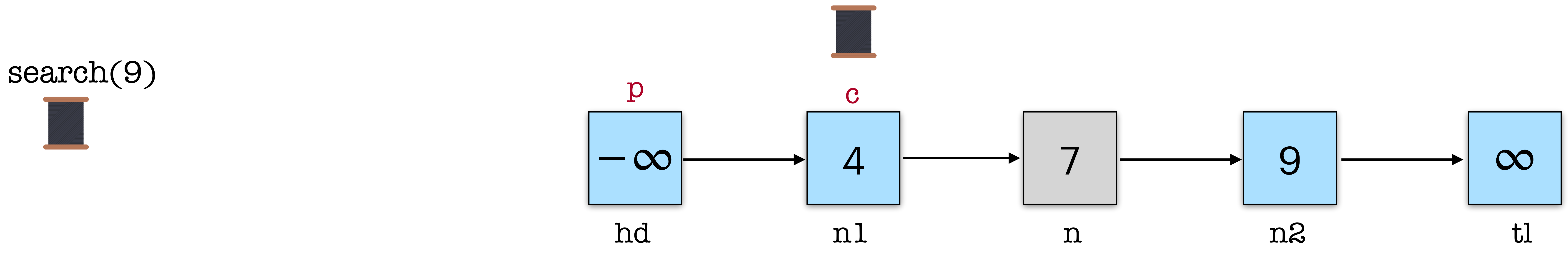
# Michael's Set

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

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

```
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
```



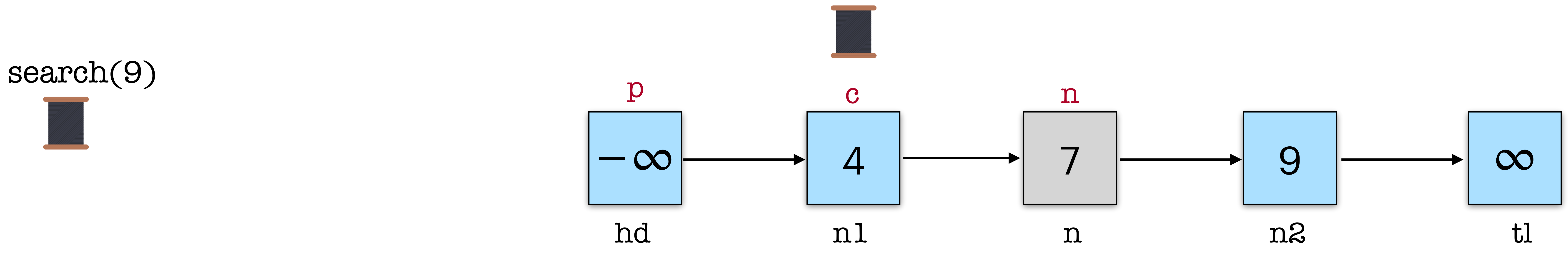
# Michael's Set

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

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

```
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

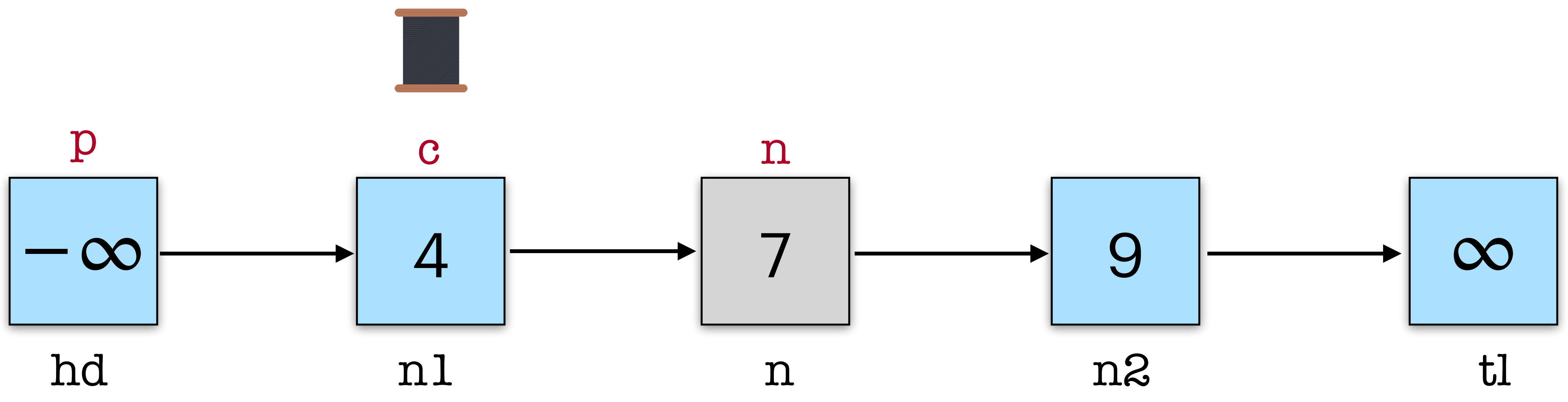
```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

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

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

```
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(9)



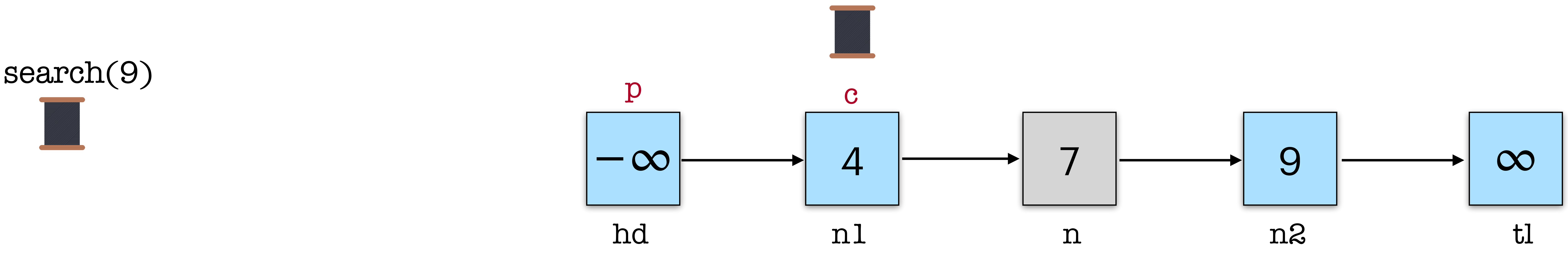
# Michael's Set

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

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

```
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

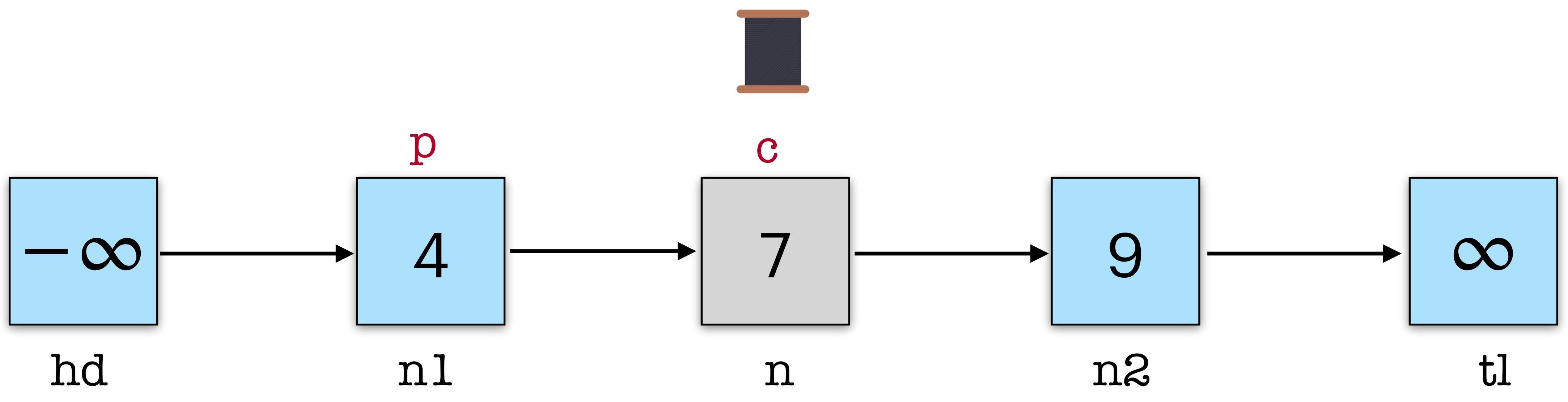
```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

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

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

```
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(9)



# Michael's Set

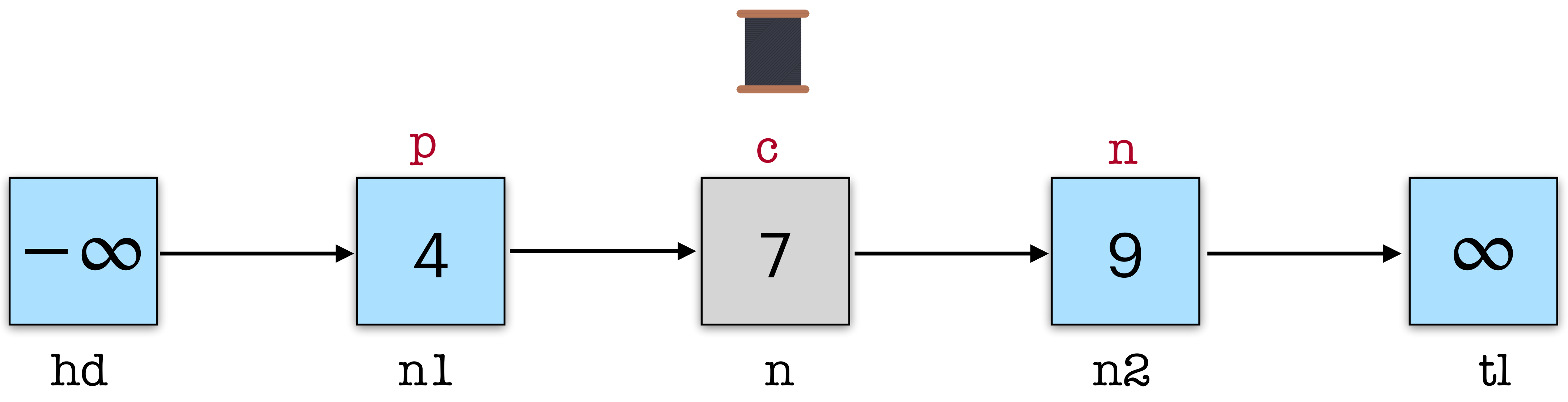
```
traverse(k, p, c) =
  (n, b) = c.next;
  if b == 0 then
    if CAS(p.next, (c,0), (n,0))
    then traverse(k, p, n) else find(k)
  else
    if c.key < k then traverse(k, c, n)
    else
      res = c.key == k;
      (p, c, res)
```

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

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

```
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(9)



# Michael's Set

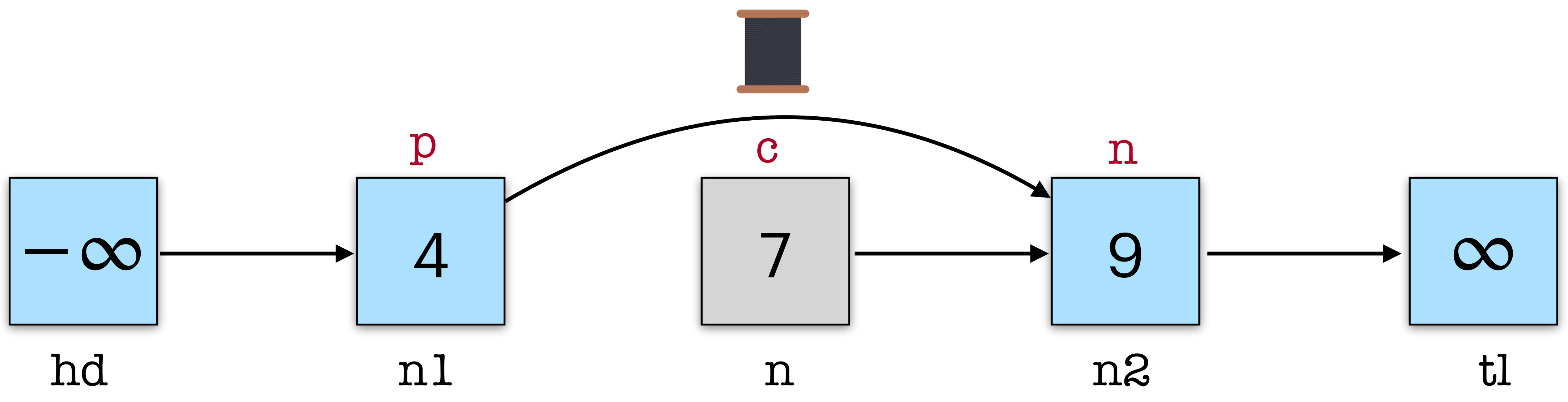
```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
      if CAS(p.next, (c,0), (n,0))
      then traverse(k, p, n) else find(k)
    else
      if c.key < k then traverse(k, c, n)
      else
        res = c.key == k;
        (p, c, res)
```

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

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

```
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(9)



# Michael's Set

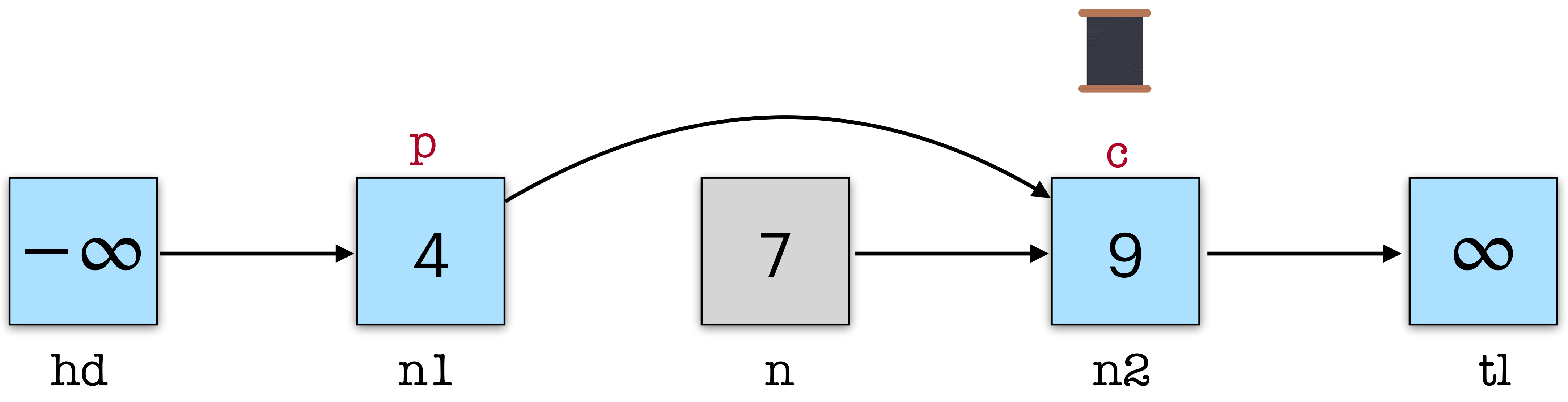
```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

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

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

```
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)
```

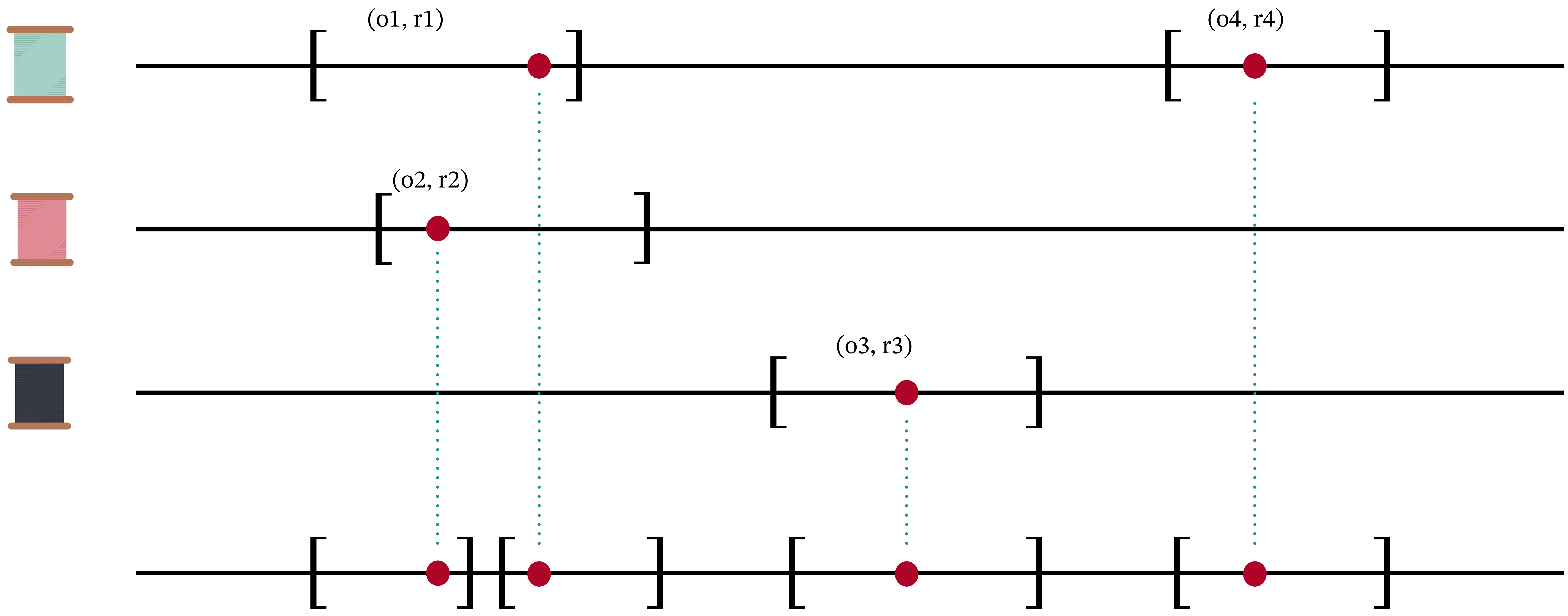
```
delete(k) =
    p, c, res = find(k);
    if (not res) then false
    else
        if MARK(c)
        then true else false
```

search(9)



# Linearizability

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



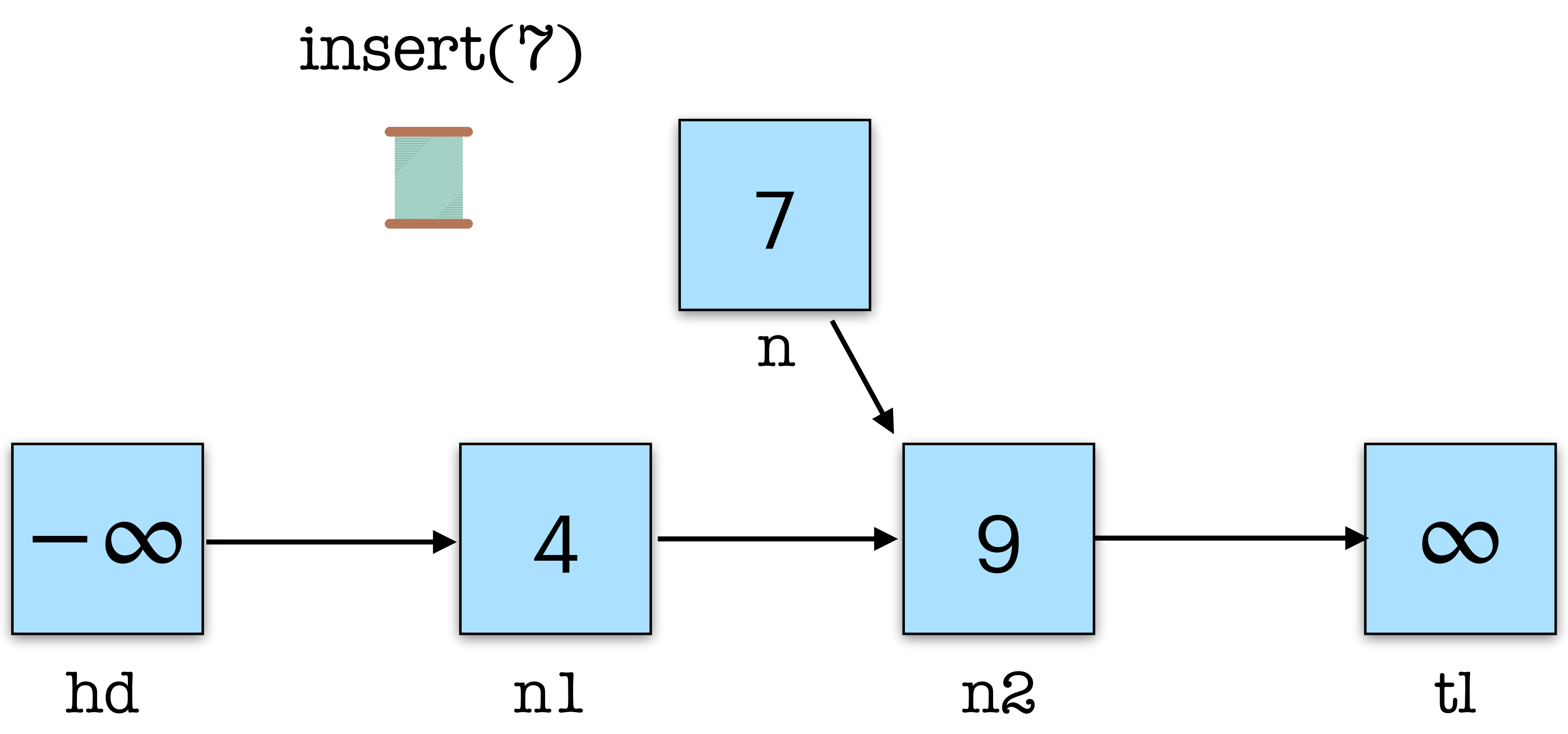
# Linearization Points

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

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

```
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
```



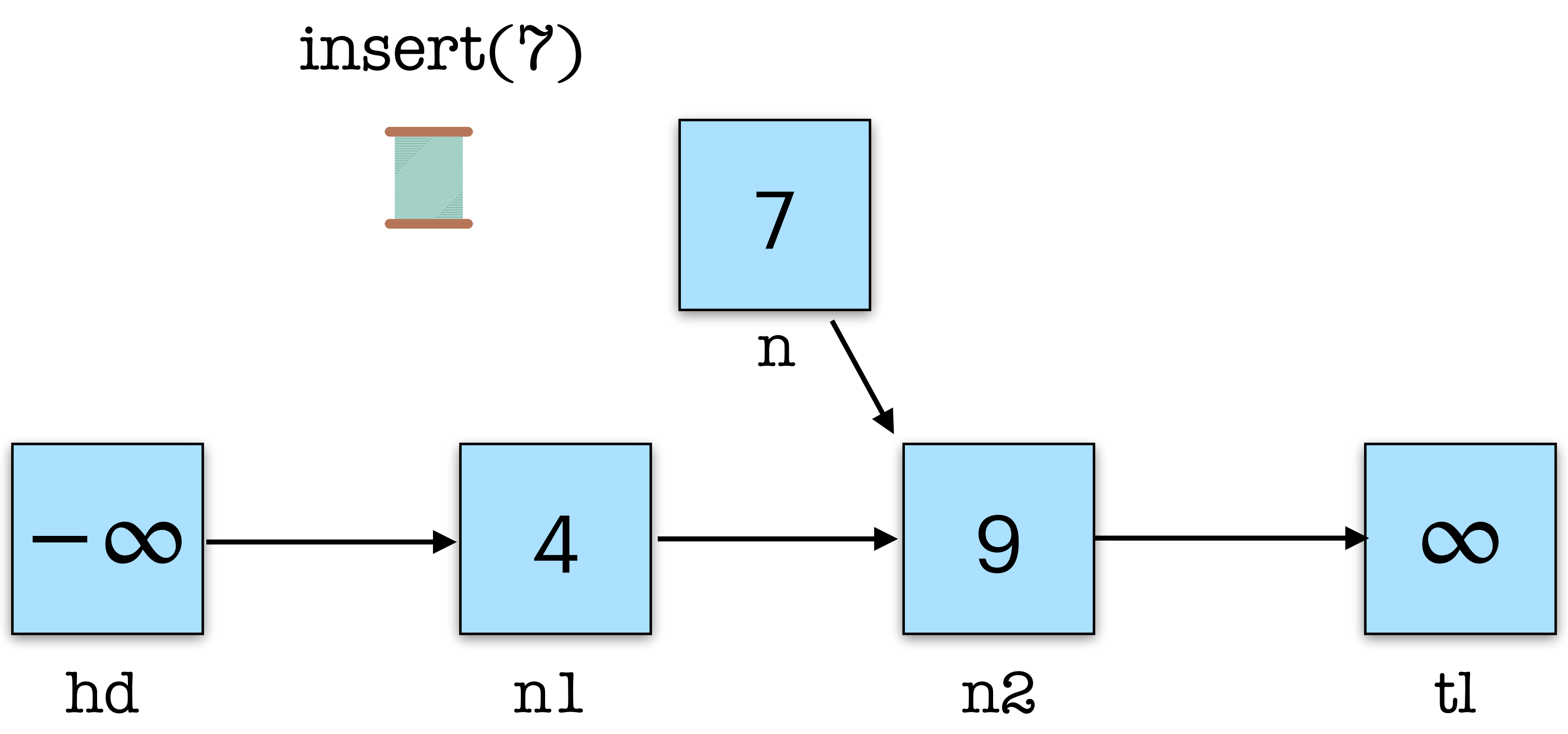
# Linearization Points

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

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

```
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
```




Modifying Linearization Points

# Linearization Points

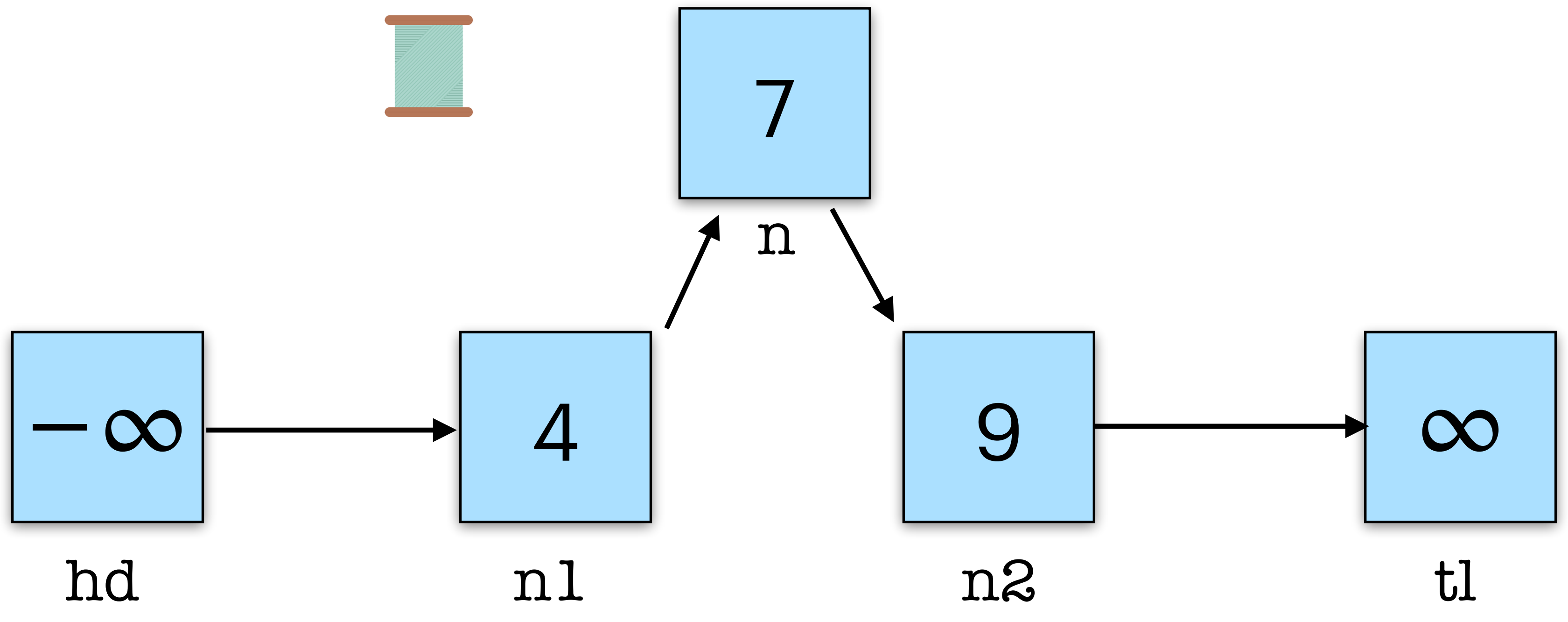
```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

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

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

```
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
```

insert(7)




Modifying Linearization Points

# Linearization Points

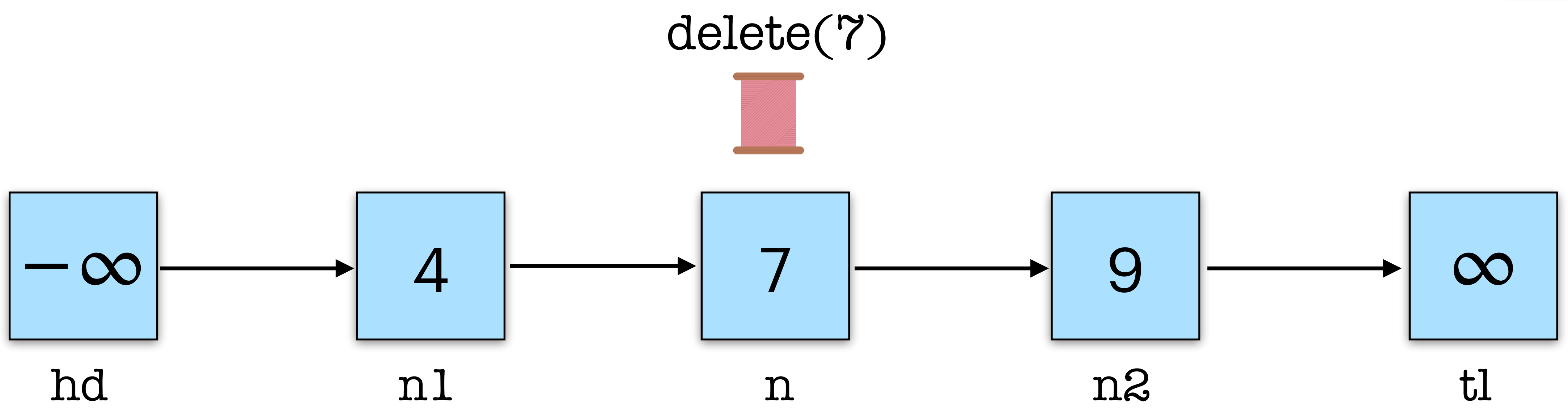
```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

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

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

```
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
```

Modifying Linearization Points



# Linearization Points

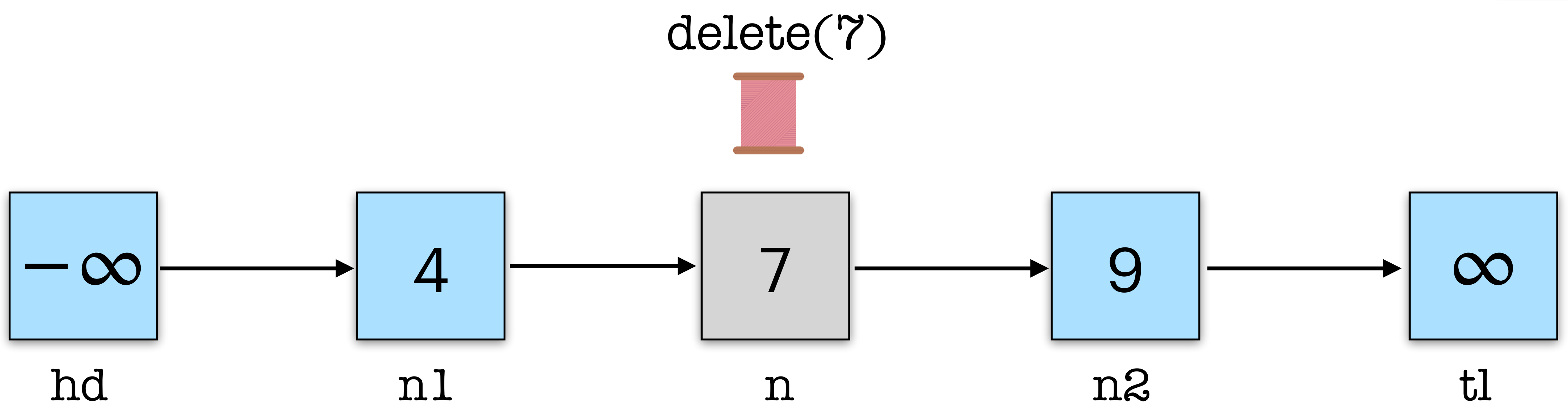
```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

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

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

```
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
```

Modifying Linearization Points



# Linearization Points

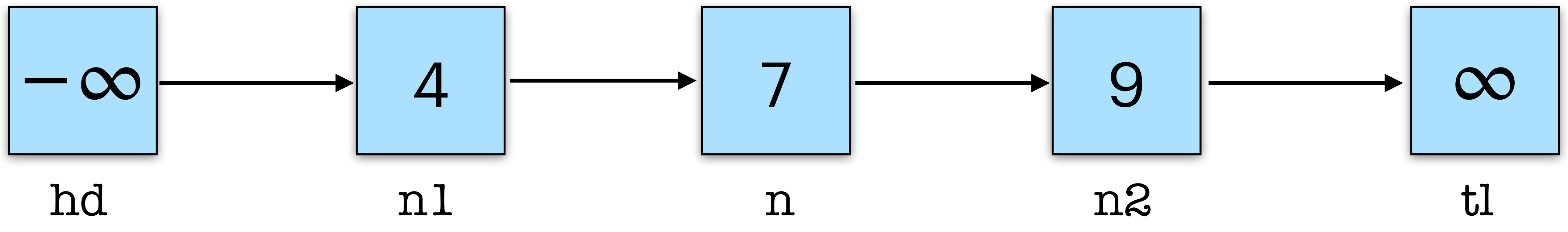
```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

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

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

```
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
```

Unmodifying Linearization Points?



# Linearization Points

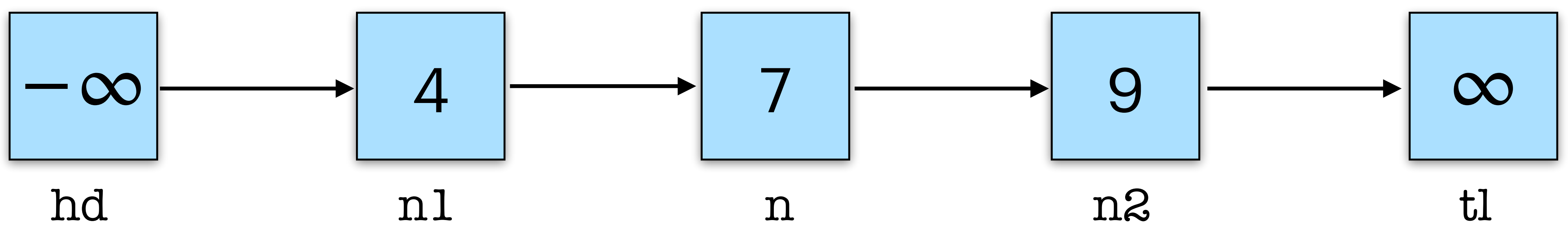
```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

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

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

```
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
```

Unmodifying Linearization Points?
**Future-dependent, external!**



# Linearization Points

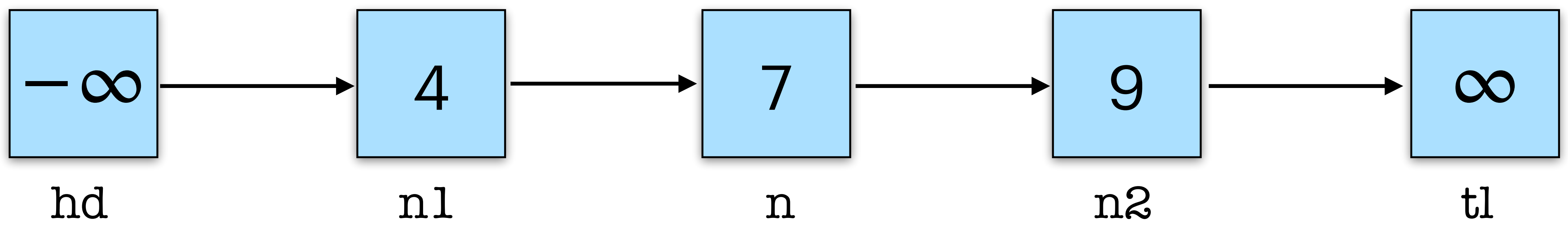
```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

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

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

```
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(7)



# Linearization Points

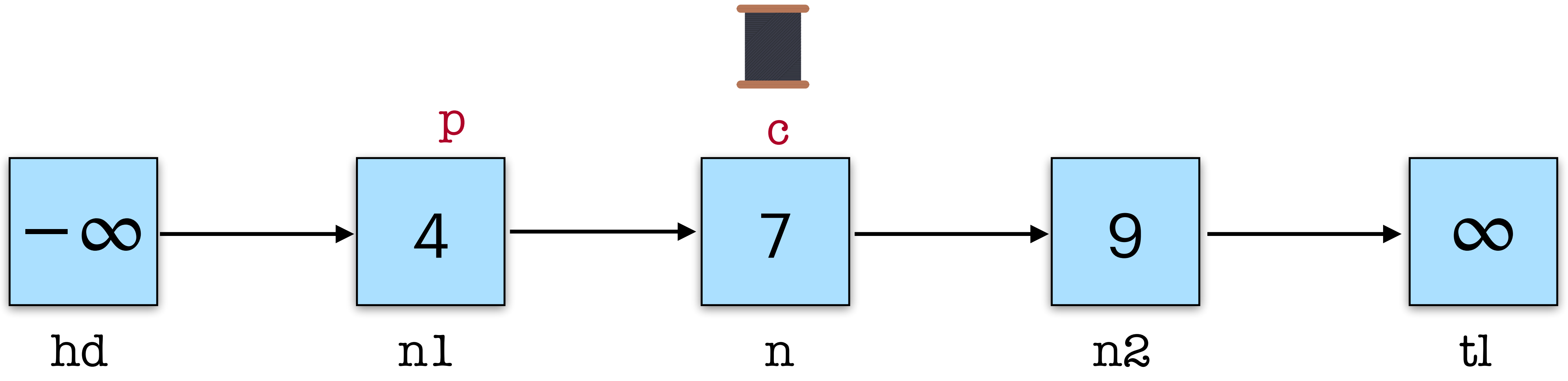
```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

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

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

```
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(7)



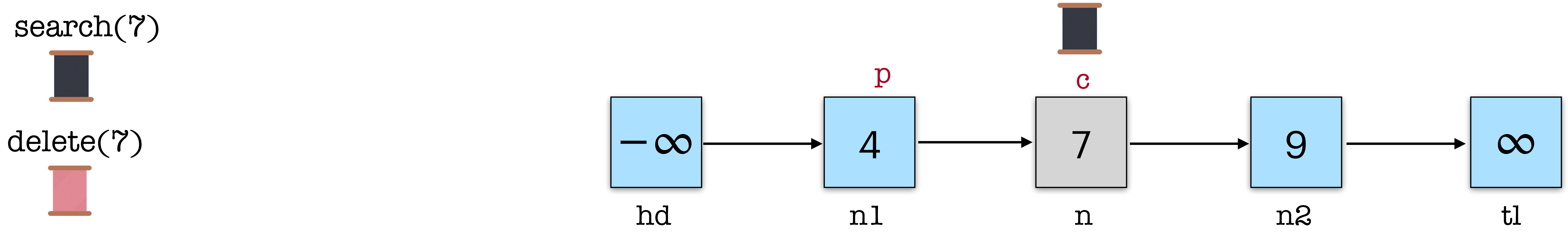
# Linearization Points

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

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

```
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
```



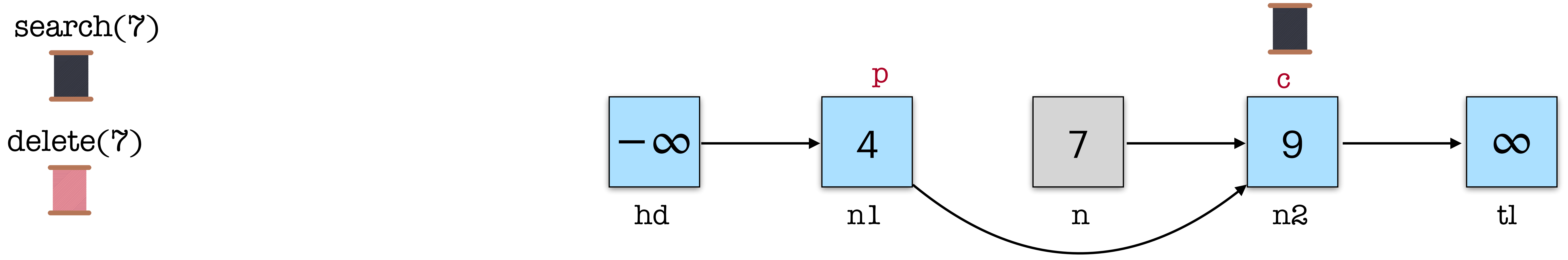
# Linearization Points

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

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

```
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
```



# Linearization Points

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

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

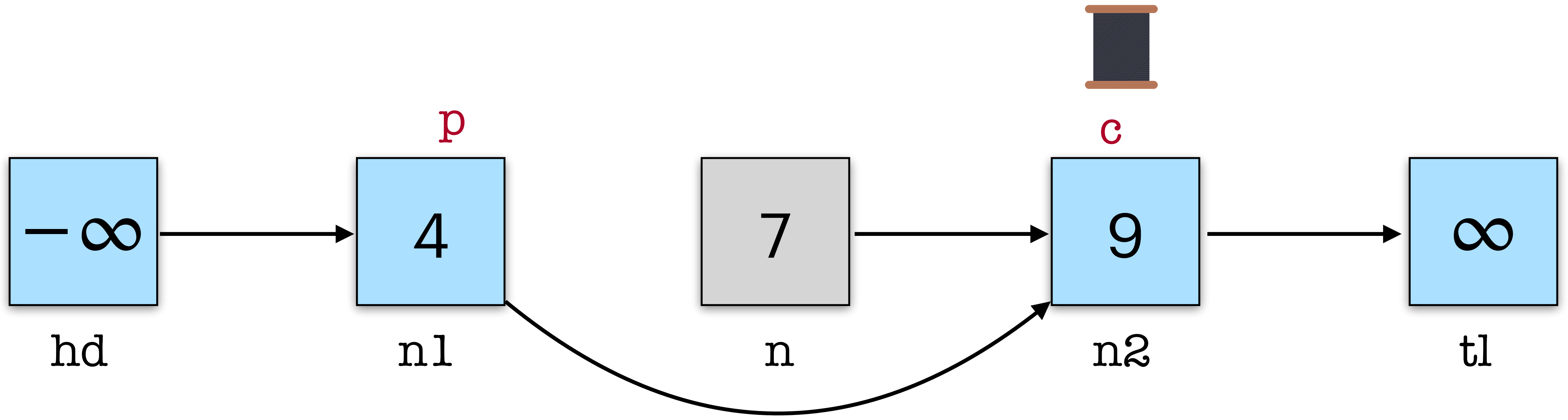
```
insert(k) =
    p, c, res = find(k);
    if res then false
    else
        n = new_node(k, c);
        if CAS(p.next, (c, 0), (n, 0))
        then true else insert(k)
```

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

search(7)

delete(7)

insert(7)



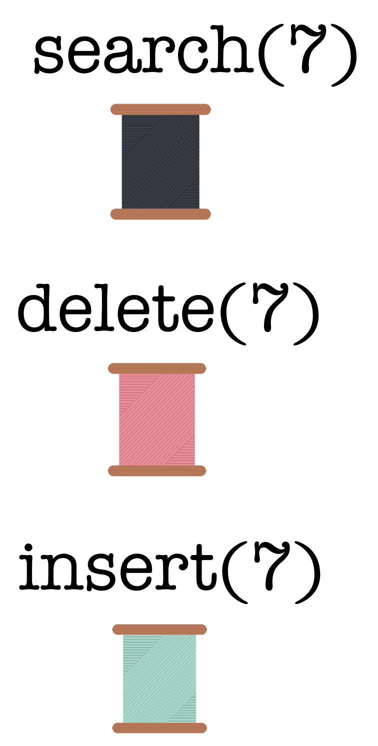
# Linearization Points

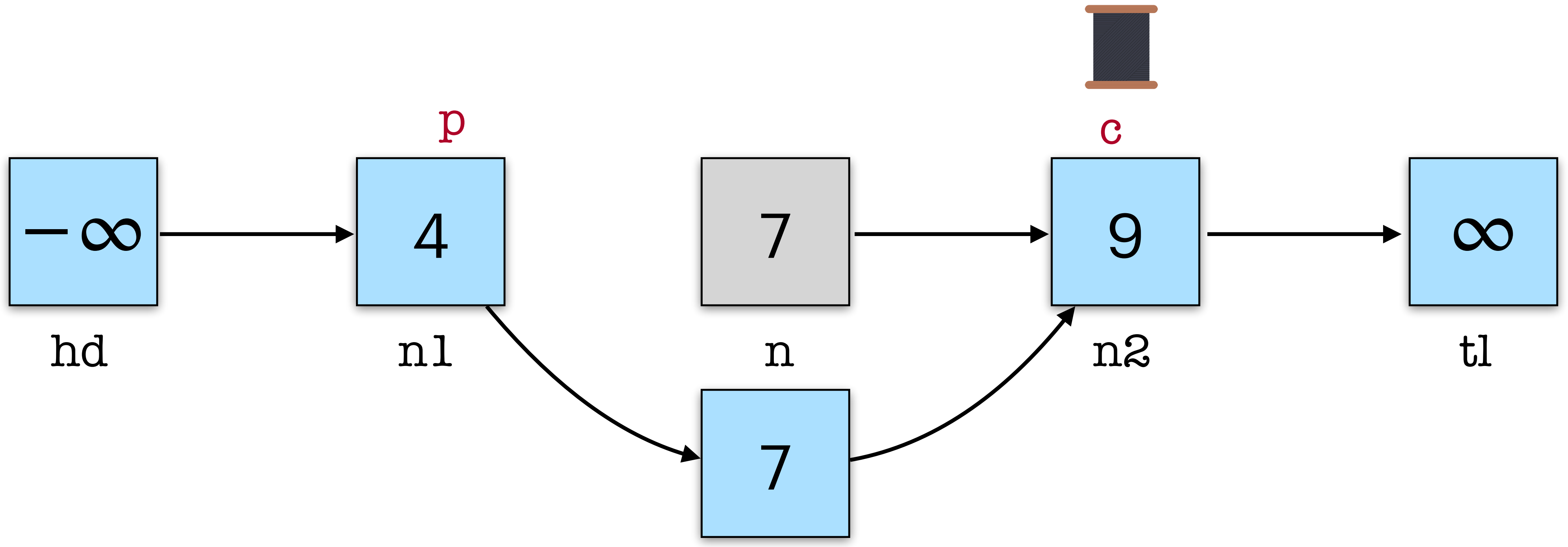
```
traverse(k, p, c) =
    (n, b) = c.next;
    if b == 0 then
        if CAS(p.next, (c,0), (n,0))
        then traverse(k, p, n) else find(k)
    else
        if c.key < k then traverse(k, c, n)
        else
            res = c.key == k;
            (p, c, res)
```

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

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

```
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
```



# 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 := p.key < 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 := p.key < 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*, p.key < 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*, p.key < k && next(p) = c && mark(p) = false

Hindsight Reasoning

# 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*, p.key < 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]

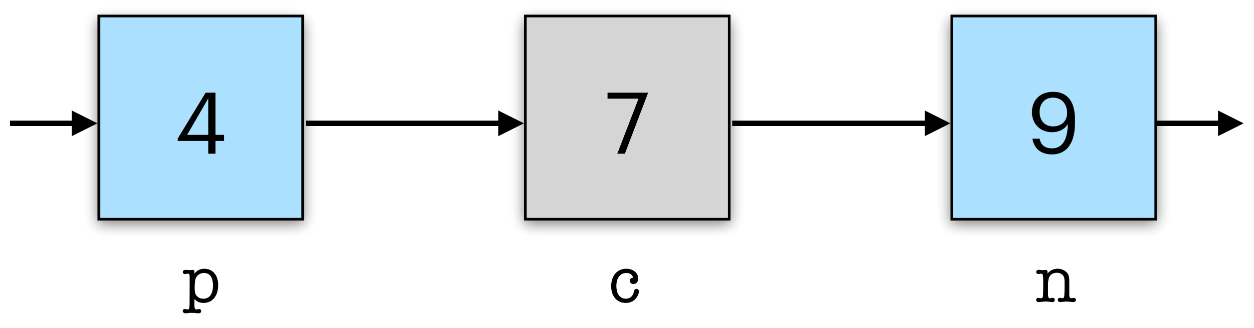
# Intuitive Proof

```
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*, p.key < k && next(p) = c && mark(p) = false

search(7)



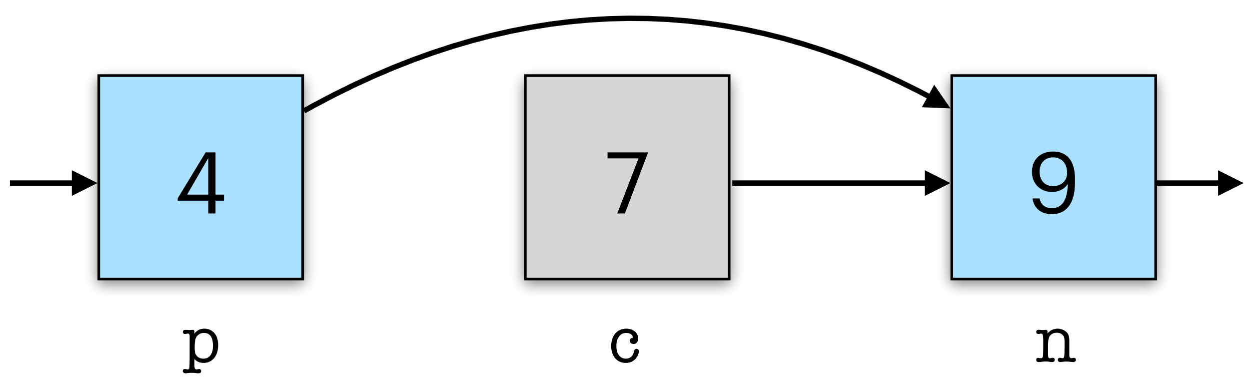
# Intuitive Proof

```
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*, p.key < k && next(p) = c && mark(p) = false

search(7)



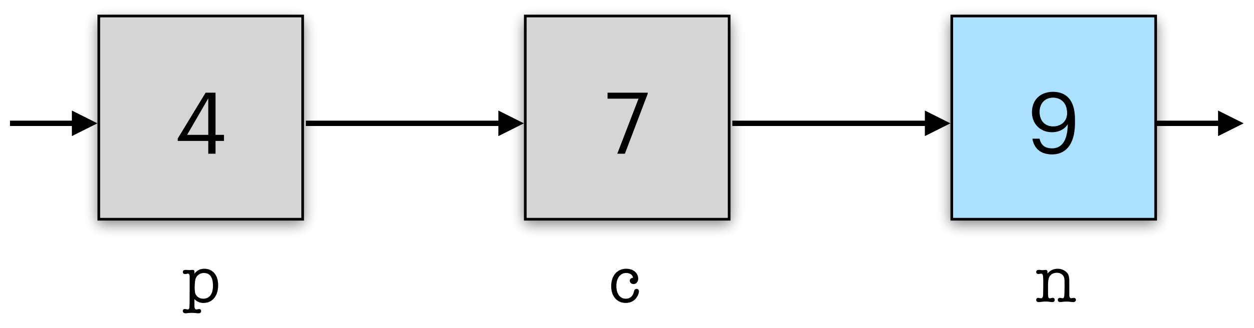
# Intuitive Proof

```
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*, p.key < k && next(p) = c && mark(p) = false

search(7)

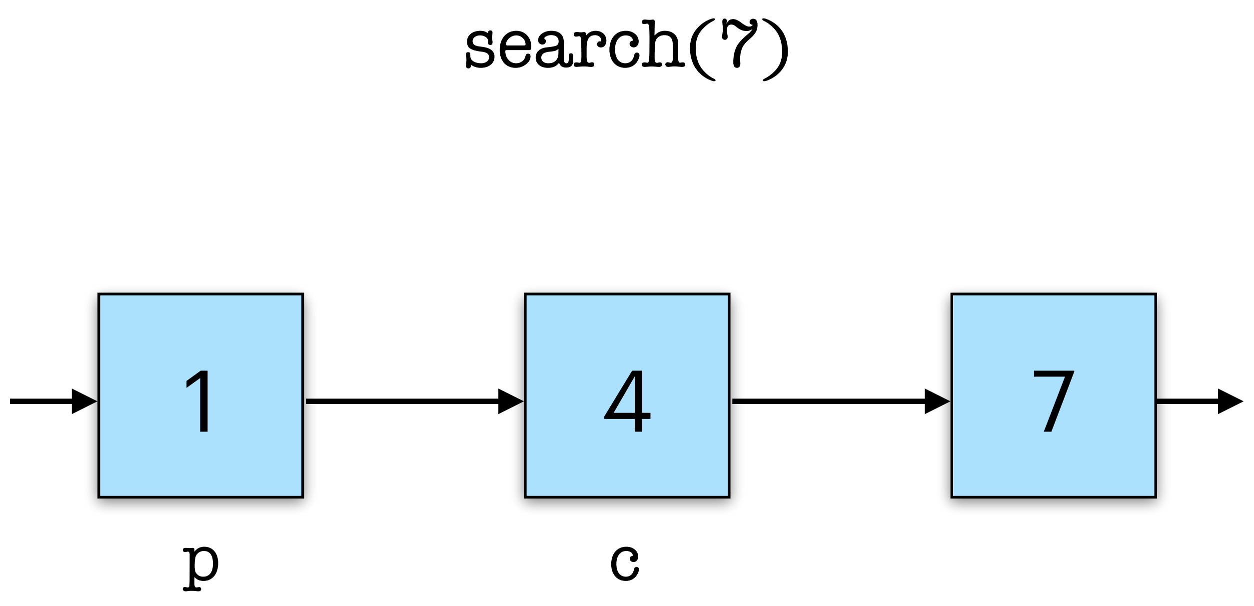


# Intuitive Proof

```
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*, p.key < k && next(p) = c && mark(p) = false



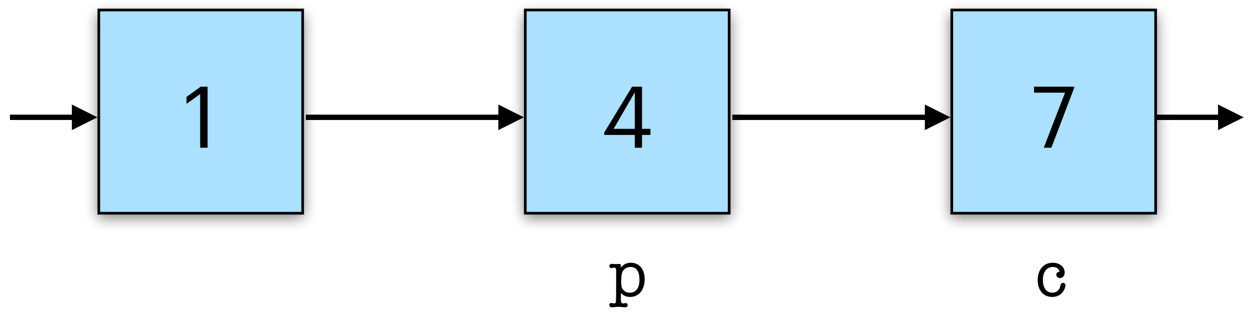
# Intuitive Proof

```
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*, p.key < k && next(p) = c && mark(p) = false

search(7)



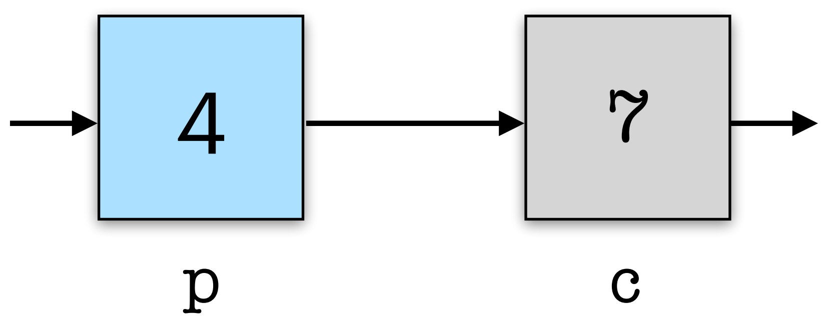
# Intuitive Proof

```
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*, p.key < k && next(p) = c && mark(p) = false

search(7)



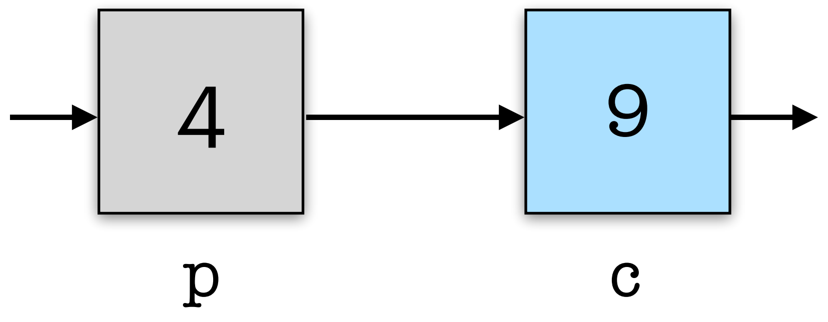
# Intuitive Proof

```
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*, p.key < k && next(p) = c && mark(p) = false

search(7)

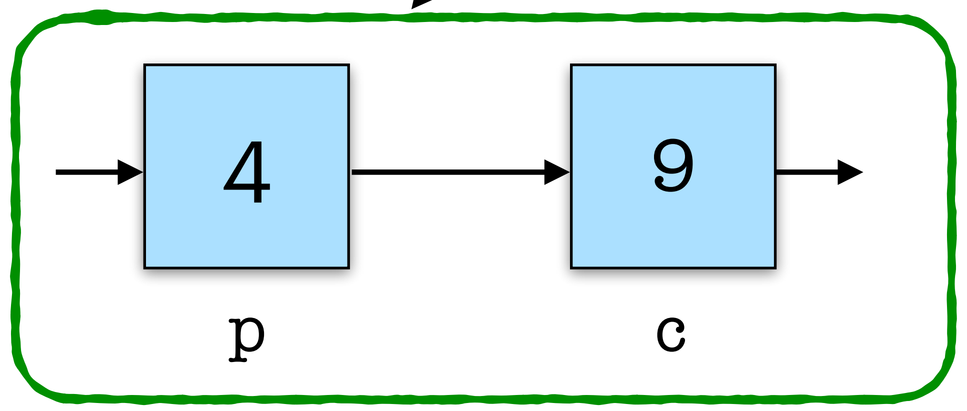


# Intuitive Proof

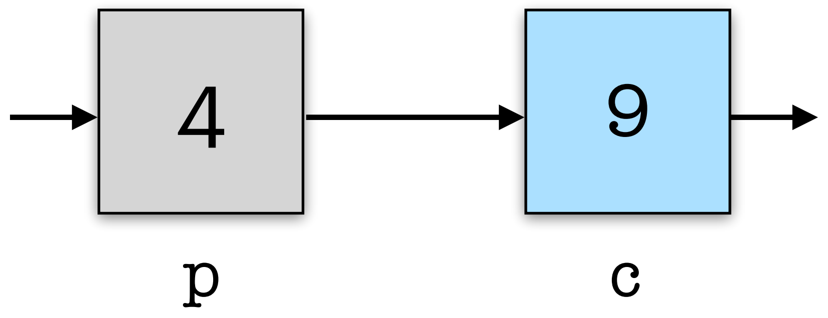
```
find(k) =
    n = hd.next;
    p, c, res = traverse(k, hd, n)

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*, p.key < k && next(p) = c && mark(p) = false




search(7)

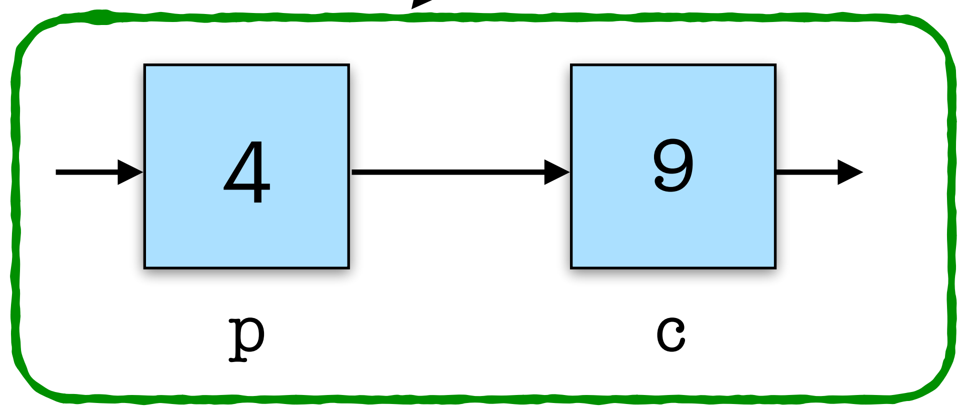


# Intuitive Proof

```
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.

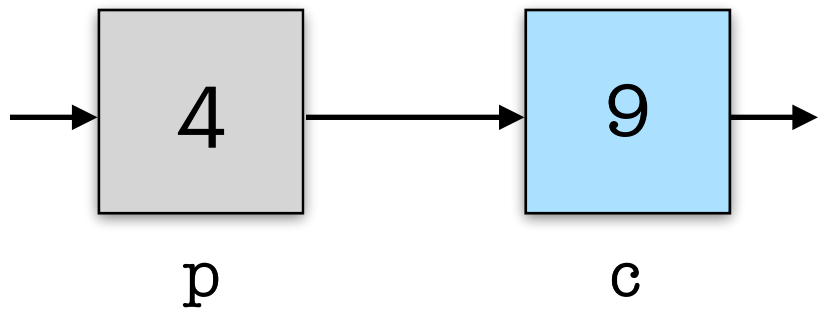
traversal invariant := *at some point*, p.key < k && next(p) = c && mark(p) = false




search(7)

1. A node once marked remains marked.
2. A node's key never changes.
3. hd-list is sorted.

....



# Michael's Set

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

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

```
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 CAS(c.next, (c, 0), (c,1))
        then true else false
```

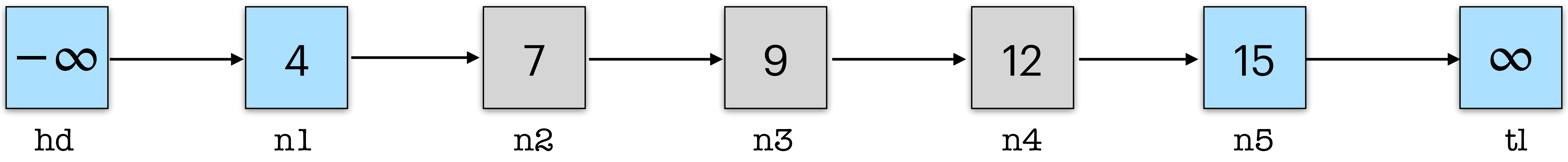
# Harris List

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

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

```
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
```



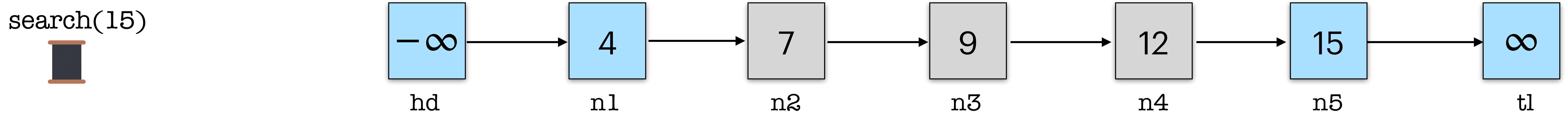
# Harris List

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

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

```
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
```



# Harris List

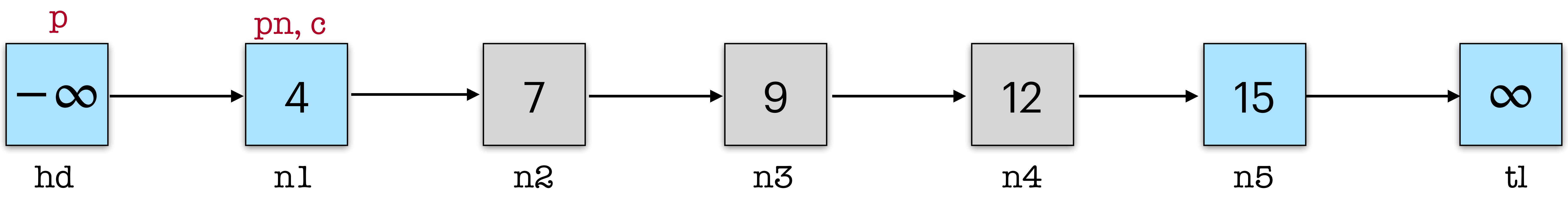
```
traverse(k, p, pn, c) =
    (n, b) = c.next;
    if b == 0 then
        traverse(k, p, pn, n)
    else
        if CAS(p.next, (pn, 0), (c, 0)) then
            if c.key < k then traverse(c, n, n)
            else
                res = c.key == k;
                (p, c, res)
        else find(k)
```

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

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

```
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(15)



# Harris List

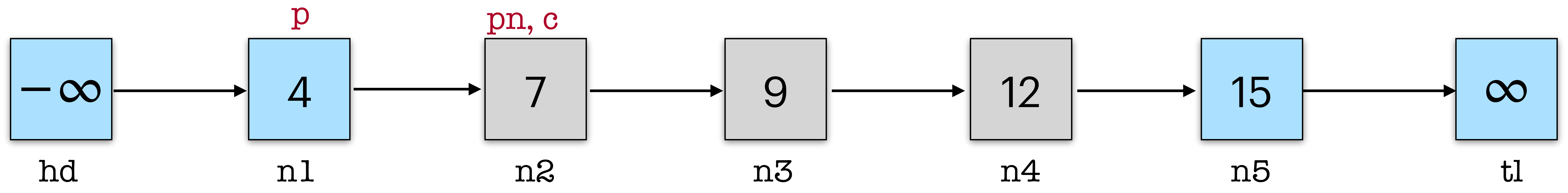
```
traverse(k, p, pn, c) =
  (n, b) = c.next;
  if b == 0 then
    traverse(k, p, pn, n)
  else
    if CAS(p.next, (pn, 0), (c, 0)) then
      if c.key < k then traverse(c, n, n)
      else
        res = c.key == k;
        (p, c, res)
    else find(k)
```

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

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

```
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(15)



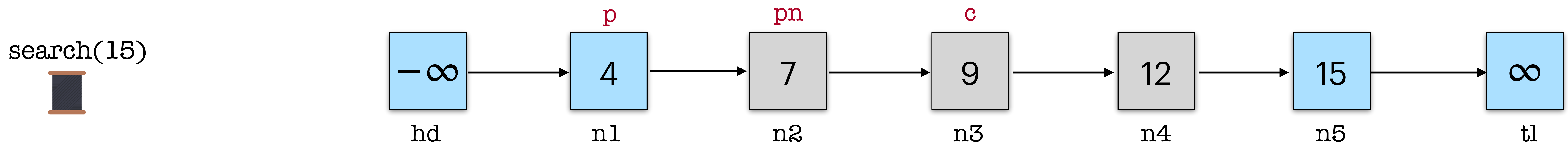
# Harris List

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

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

```
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
```



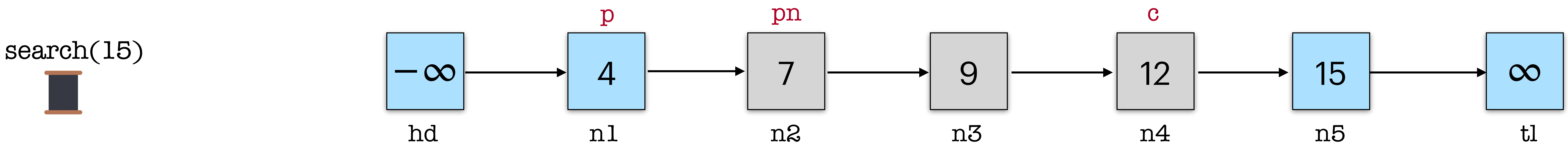
# Harris List

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

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

```
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
```



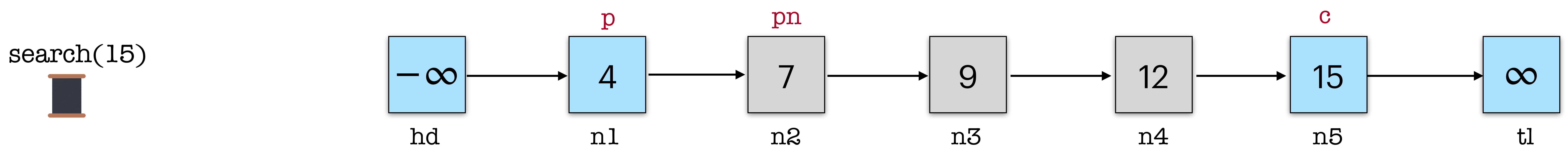
# Harris List

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

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

```
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
```



# Harris List

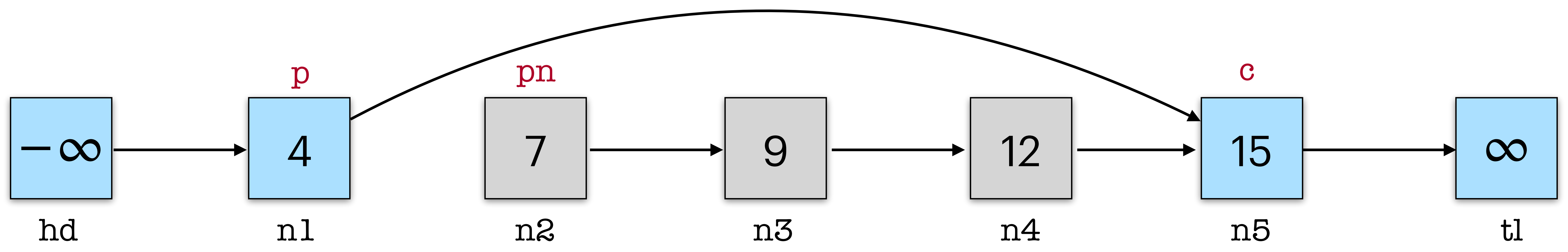
```
traverse(k, p, pn, c) =
  (n, b) = c.next;
  if b == 0 then
    traverse(k, p, pn, n)
  else
    if CAS(p.next, (pn, 0), (c, 0)) then
      if c.key < k then traverse(c, n, n)
      else
        res = c.key == k;
        (p, c, res)
    else find(k)
```

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

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

```
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)
```

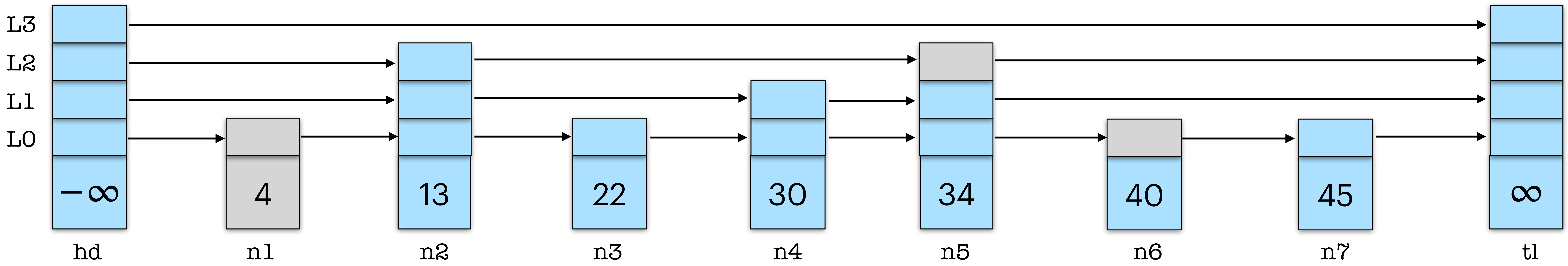
```
delete(k) =
  p, c, res = find(k);
  if (not res) then false
  else
    if MARK(c)
    then true else false
```

search(15)



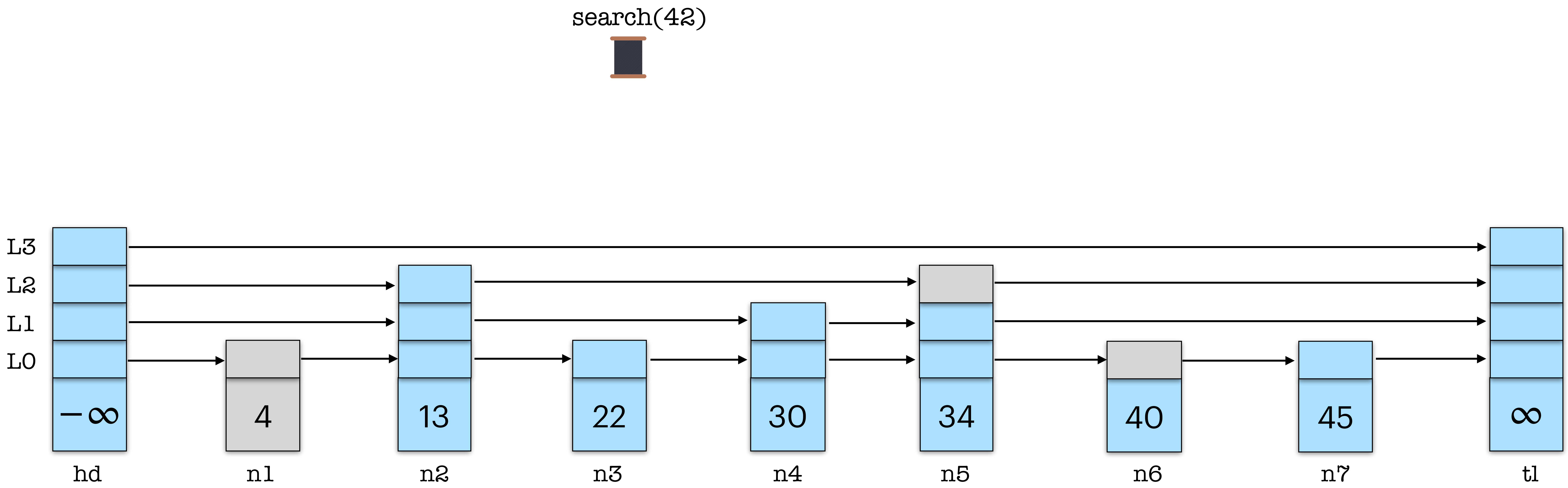
# Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist



# Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

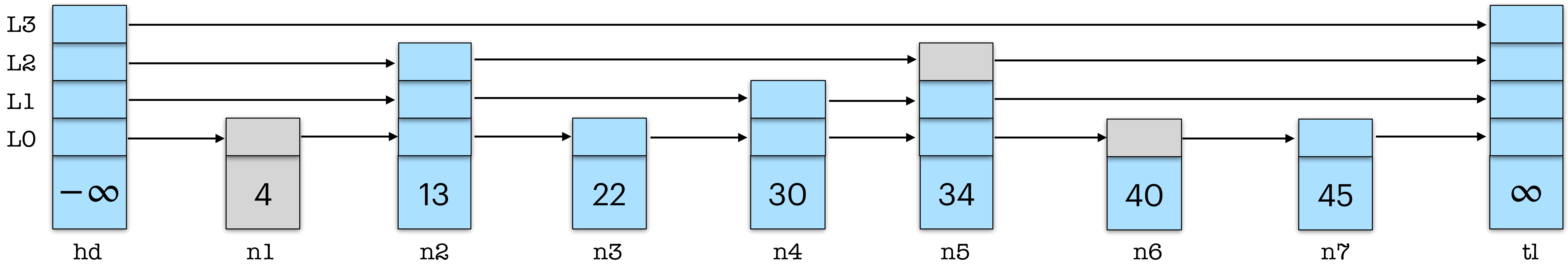


# Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

search(42)

| | pred | succ |
|---|---|---|
| L3 | | |
| L2 | | |
| L1 | | |
| L0 | | |

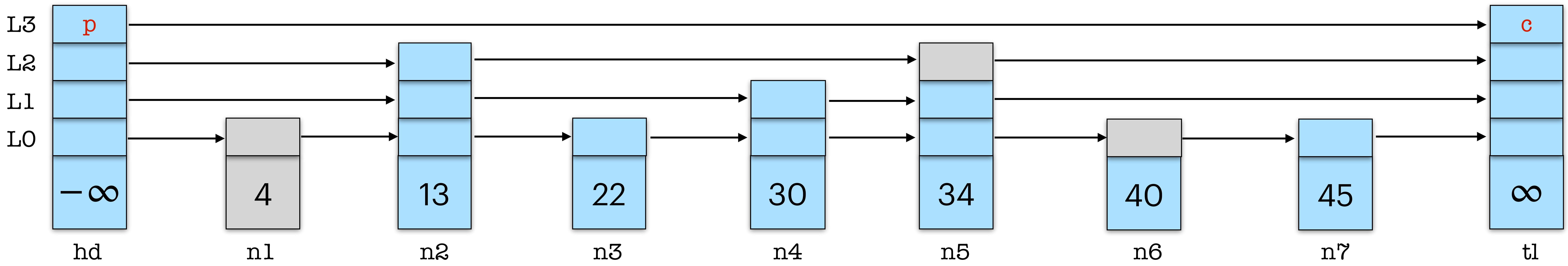


# Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

search(42)

| | pred | succ |
|---|---|---|
| L3 | hd | tl |
| L2 | | |
| L1 | | |
| L0 | | |

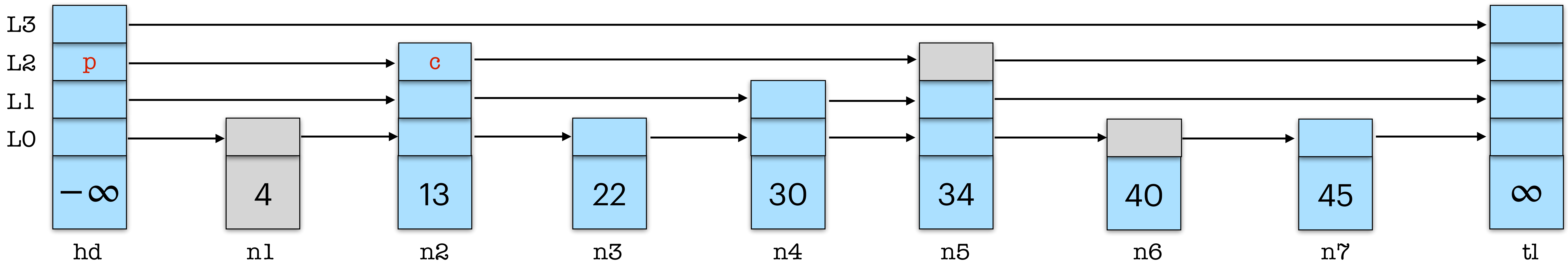


# Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

search(42)

| | pred | succ |
|---|---|---|
| L3 | hd | tl |
| L2 | | |
| L1 | | |
| L0 | | |

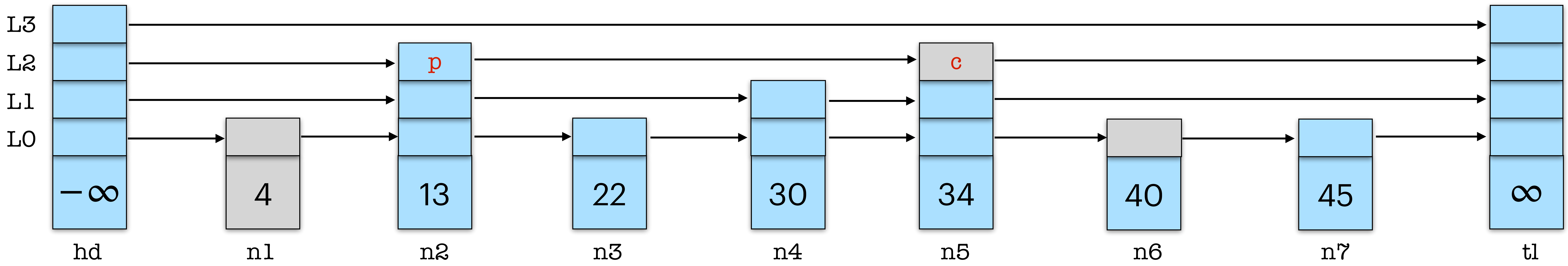


# Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

search(42)

| | pred | succ |
|---|---|---|
| L3 | hd | tl |
| L2 | | |
| L1 | | |
| L0 | | |

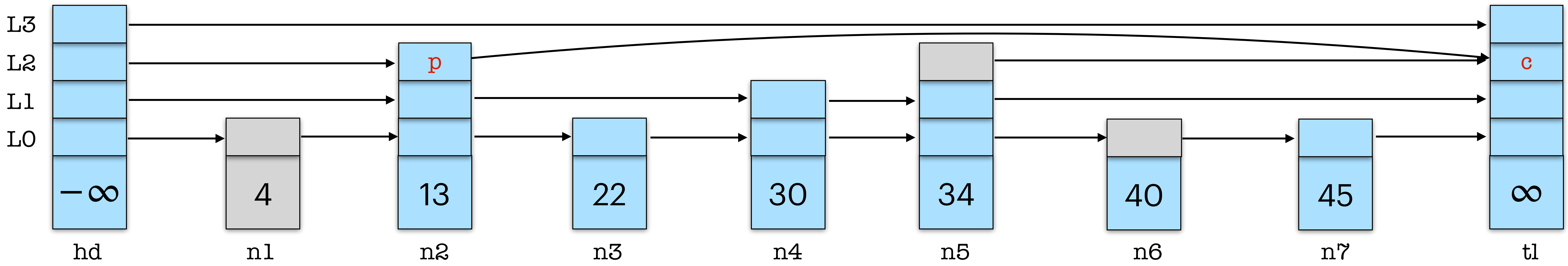


# Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

search(42)

| | pred | succ |
|---|---|---|
| L3 | hd | tl |
| L2 | n2 | tl |
| L1 | | |
| L0 | | |

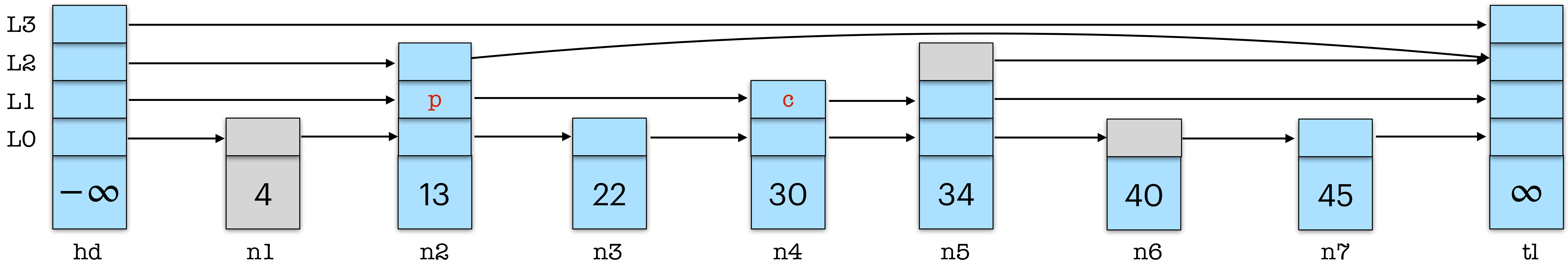


# Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

search(42)

| | pred | succ |
|---|---|---|
| L3 | hd | tl |
| L2 | n2 | tl |
| L1 | | |
| L0 | | |

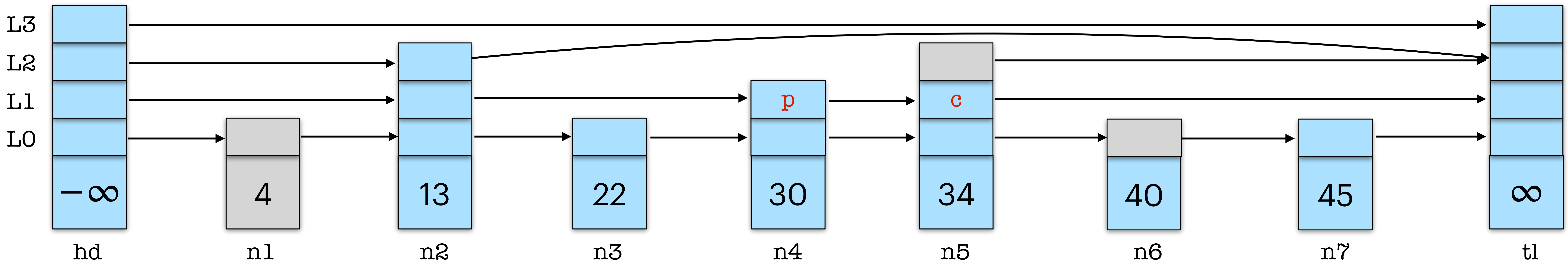


# Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

search(42)

| | pred | succ |
|---|---|---|
| L3 | hd | tl |
| L2 | n2 | tl |
| L1 | | |
| L0 | | |

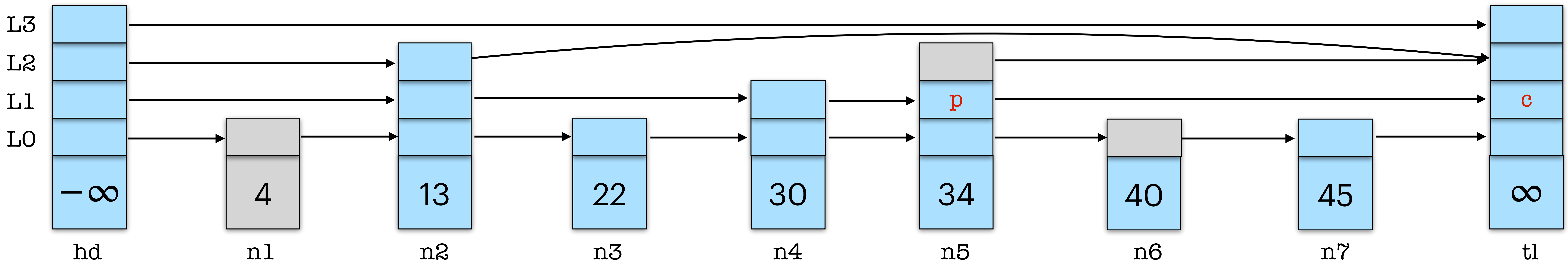


# Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

search(42)

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

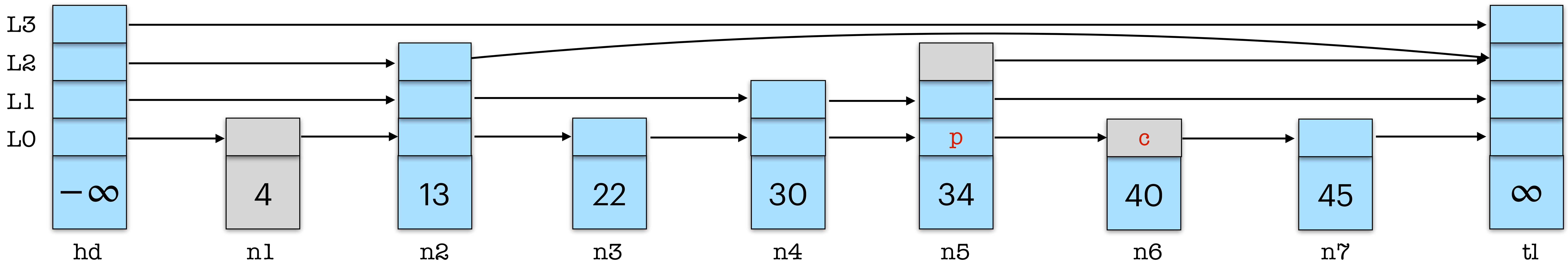


# Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

search(42)

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

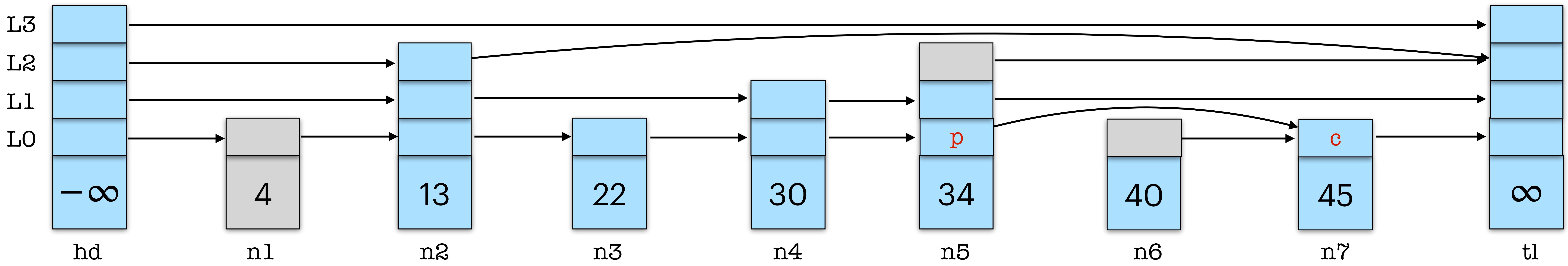


# Skiplists

Michael's Set + Levels ≈ Herlihy-Shavit Skiplist

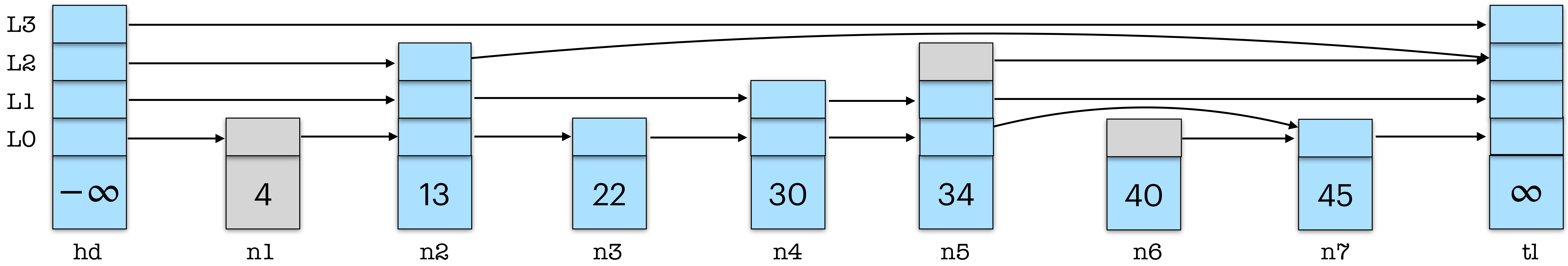
search(42)

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



# Skiplists

Harris List + Levels ≈ ??

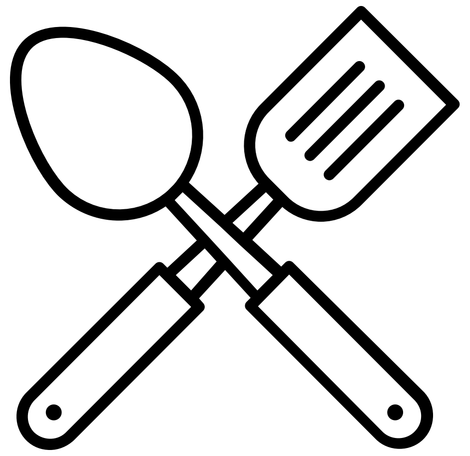


# Outline

Step 1:

Find a class of structures with common correctness reasoning

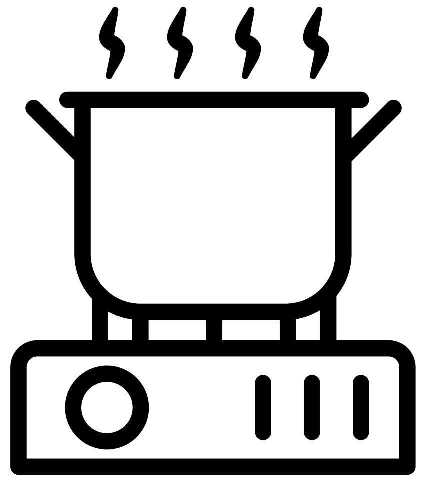
- ECOOP24 : (Lock-free) linked lists and skiplists



Step 2:

Develop enabling technology

- Template Algorithms
- Hindsight Framework



Step 3:

Formalize the proof

- Evaluation

# Template Algorithms

```
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
```

# Template Algorithms

```
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
```

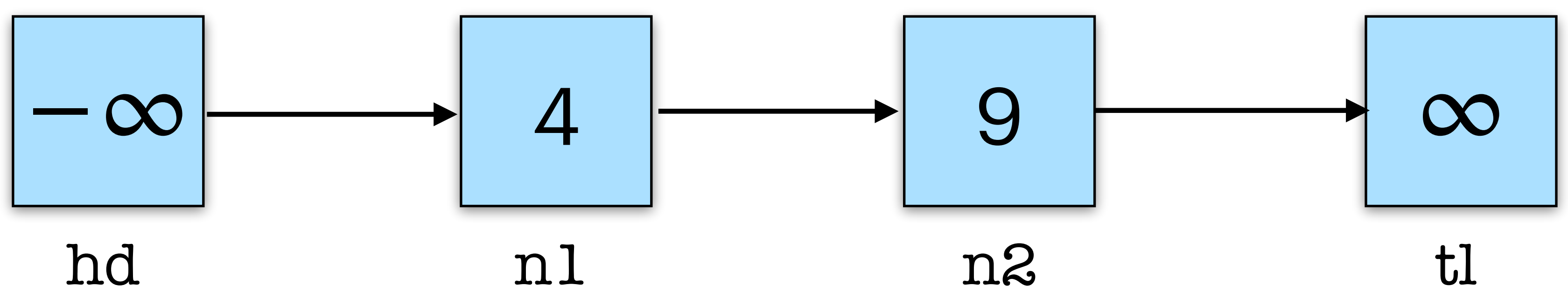
```
let insert k =
  let ps = allocArr L hd in
  let cs = allocArr L tl in
  let p, c, res = traverse ps cs k in
  if res then
    false
  else
    let h = randomNum L in
    let e = createNode k h cs in
    match changeNext 0 p c e with
    | Success ->
      maintainanceOp_ins k ps cs e; true
    |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

# Template Algorithms

```
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
```

insert(7)

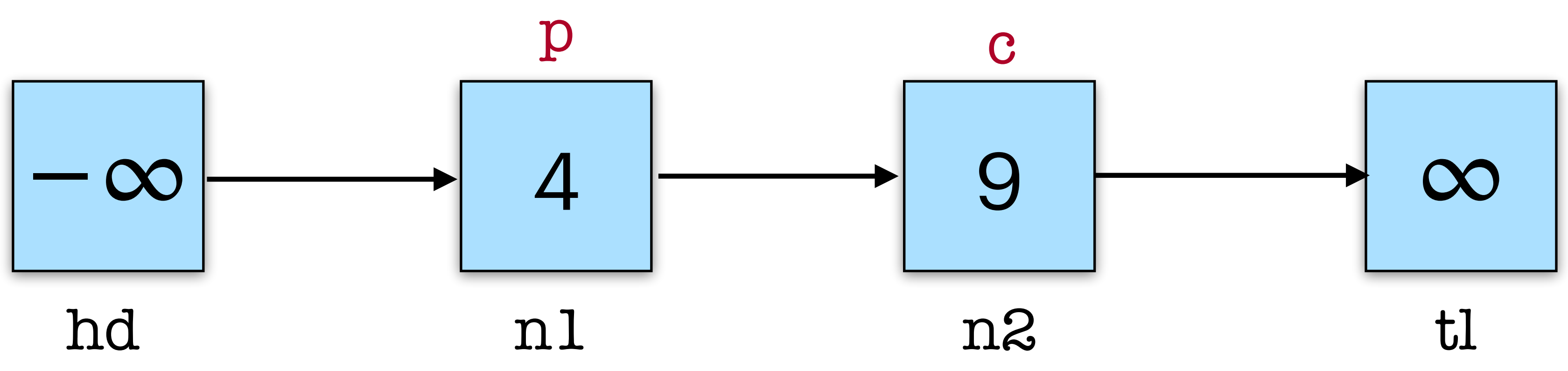


# Template Algorithms

```
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
```

insert(7)

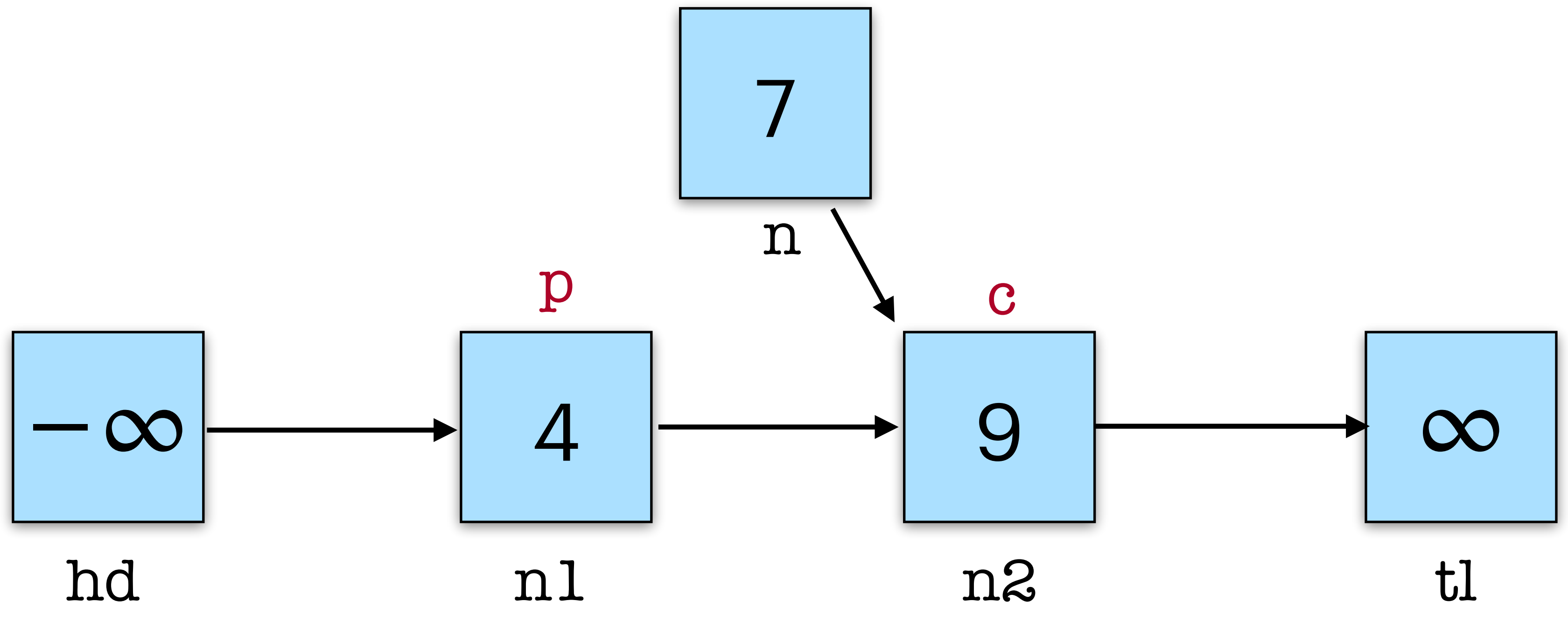


# Template Algorithms

```
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
```

insert(7)

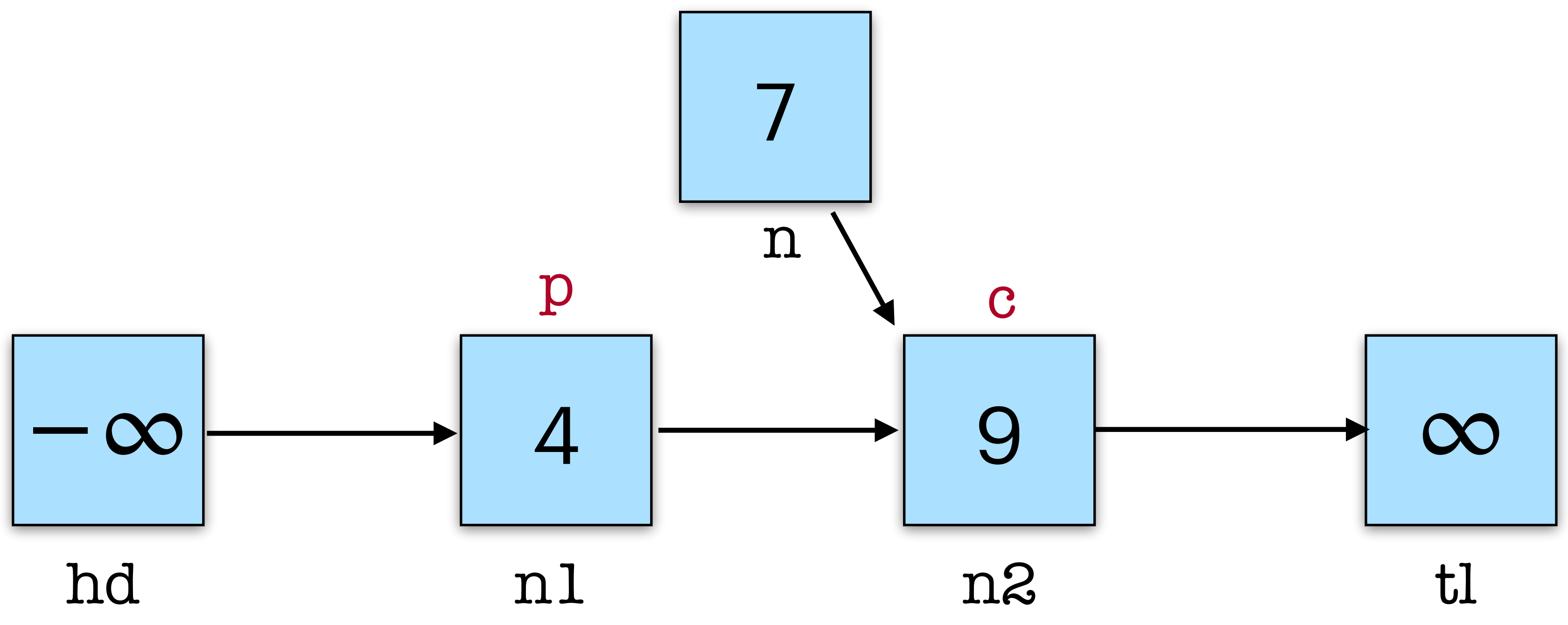


# Template Algorithms

```
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
```

insert(7)

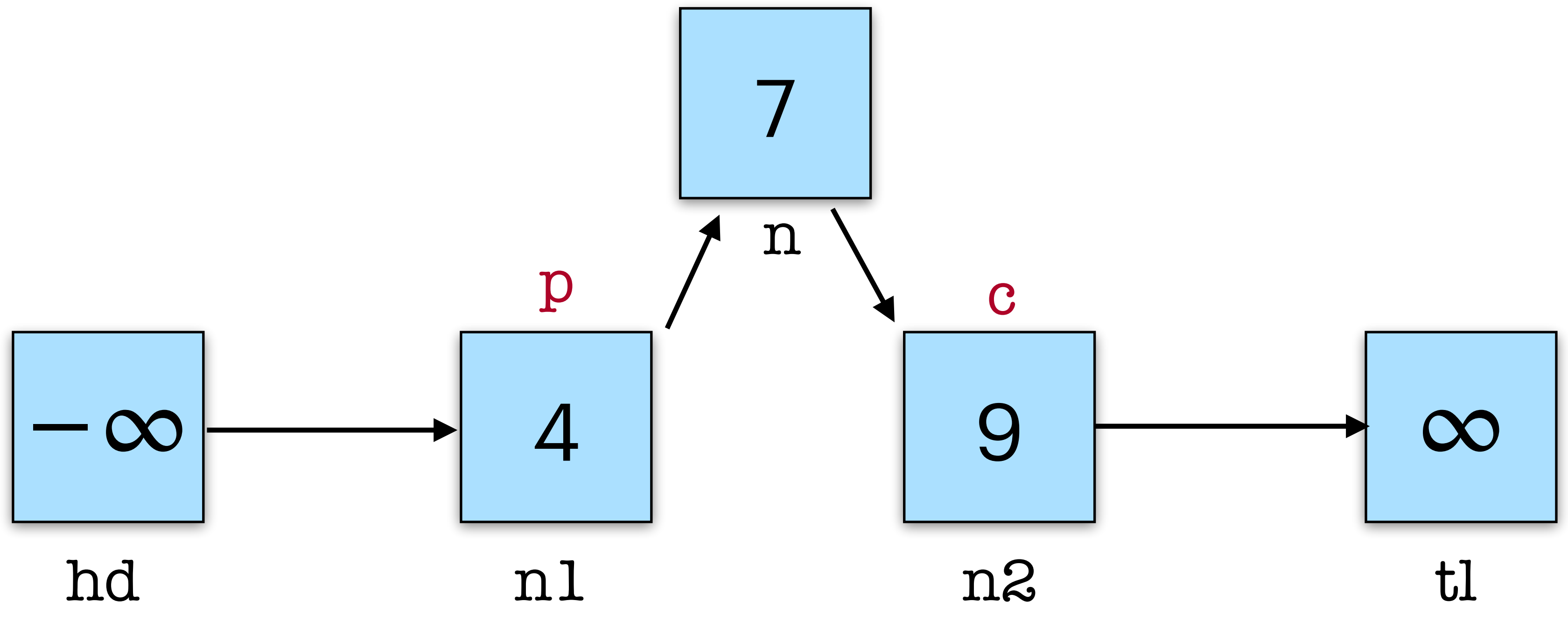


# Template Algorithms

```
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
```

insert(7)



# Template Algorithms

```
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
```

# Template Algorithms

```
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
```

# Template Algorithms

```
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
```

# Template Algorithms

```
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
```

$$\{\ \mathrm{Node}(n, k, m, n')\ \}\ \mathrm{markNode}\ 0\ n\ \ \{\ \mathrm{Node}(n, k, m[0 \mapsto \mathrm{true}], n')\ \}$$

# Atomic Triples

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

# Atomic Triples

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

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

# Atomic Triples

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

Client-level Specification

intuitive proof

- Data Structure invariants
- Proof of the method

# Atomic Triples

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

Client-level Specification

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

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

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

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

Client-level Specification

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

Hindsight Specification

Client-level Specification

- Data Structure invariants
- Proof of the method

- Prophecy variables
- Helping Protocol

**Hindsight Specification** :

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

# Hindsight Framework

- Data Structure invariants
- Proof of the method

**Hindsight Specification**

Data Structure Agnostic

- Prophecy variables
- Helping Protocol

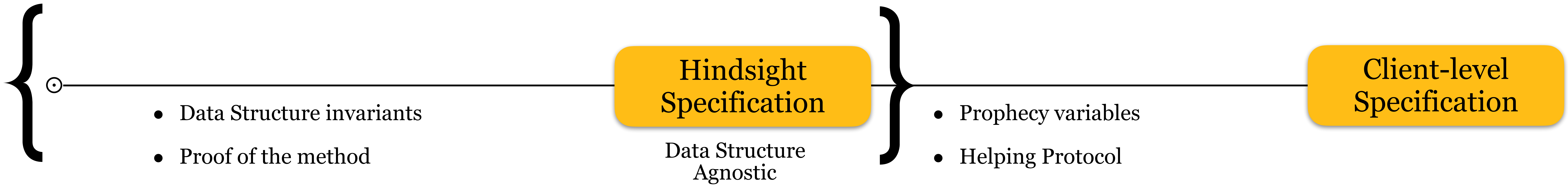
**Client-level Specification**

**Hindsight Specification** :

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

# Hindsight Framework

Proof Author Obligations

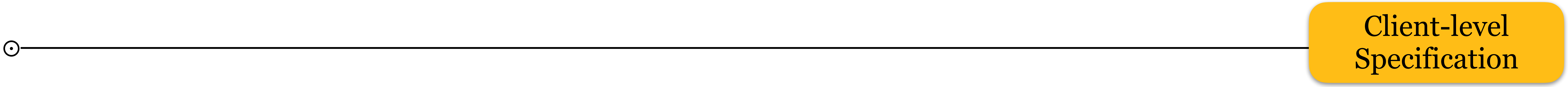



**Hindsight Specification** :

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

# Proof Author POV

**Framework provides:**

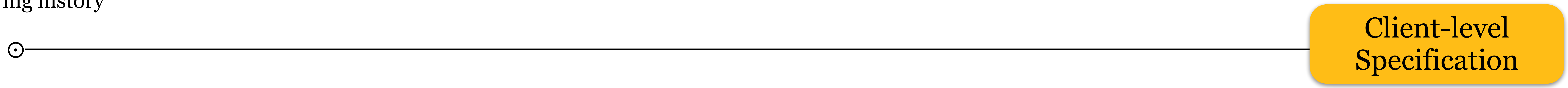



**Proof author obligations:**

# Proof Author POV

**Framework provides:**

0: Shared state
invariant for
storing history




**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:**

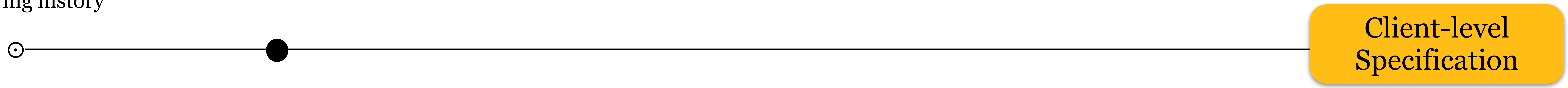
```
let delete k =
  let ps = allocArr L hd in
  let cs = allocArr L tl in
  let p, c, res = traverse ps cs k in
  if not res then
    false
  else
    maintainanceOp_del c;
    match markNode 0 c with
    | Success -> traverse ps cs k; true
    | Failure -> false
```

```
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:**

0: Shared state
invariant for
storing history

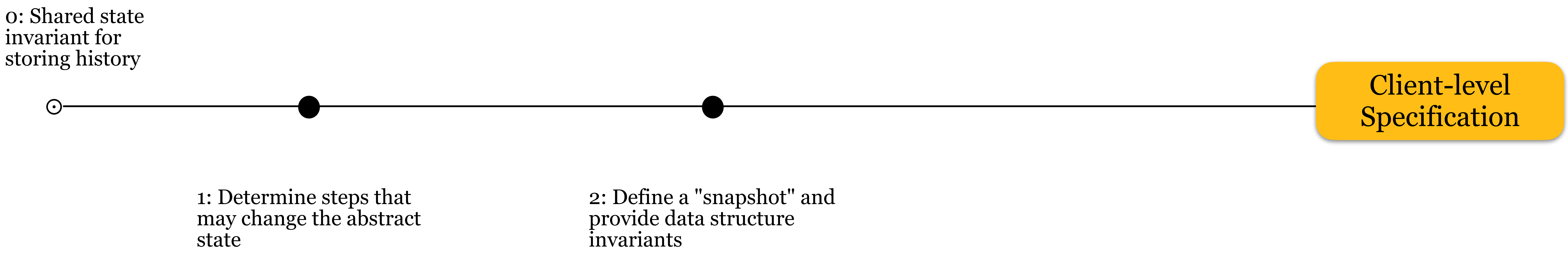



1: Determine steps that
may change the abstract
state

**Proof author obligations:**

# Proof Author POV

**Framework provides:**




**Proof author obligations:**

# Proof Author POV

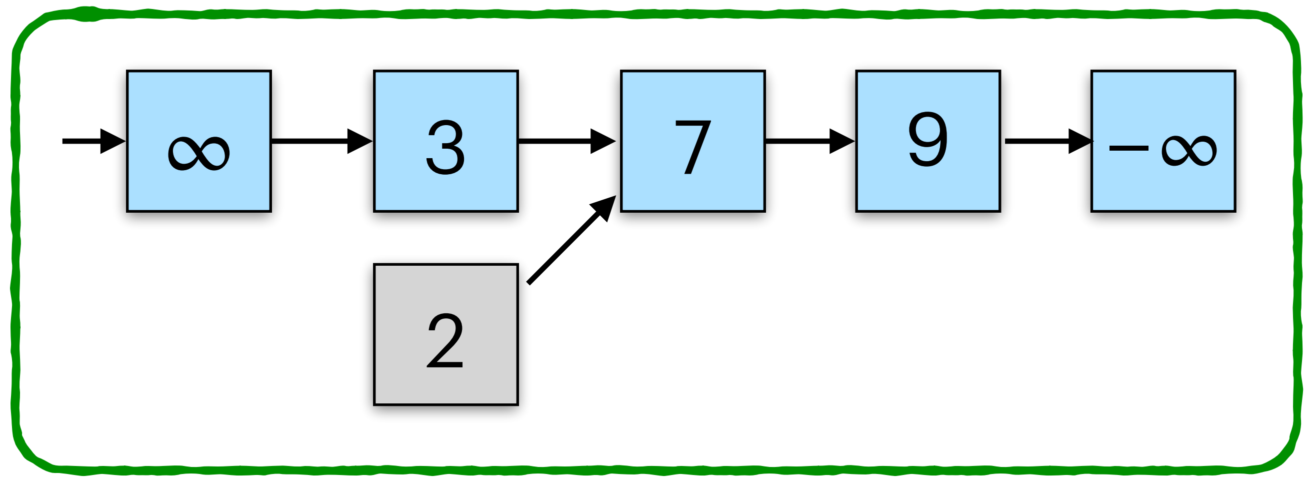
**Framework provides:**

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



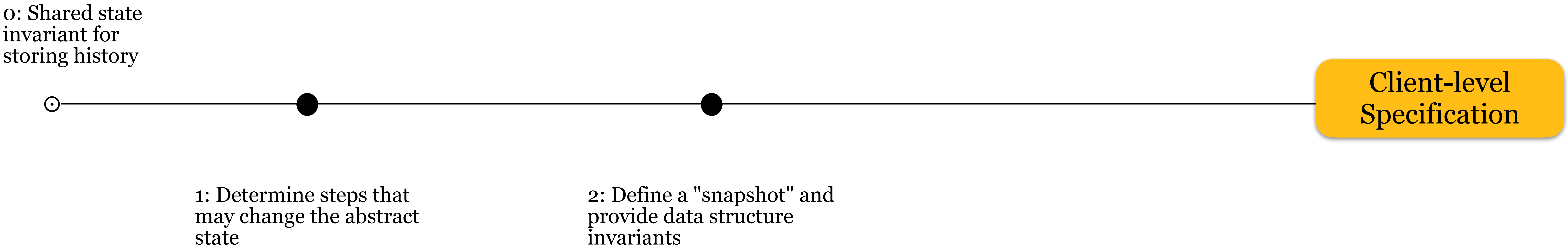

1. A node once marked remains marked.
2. A node's key never changes.
3. hd-list is sorted.

....

**Proof author obligations:**

# Proof Author POV

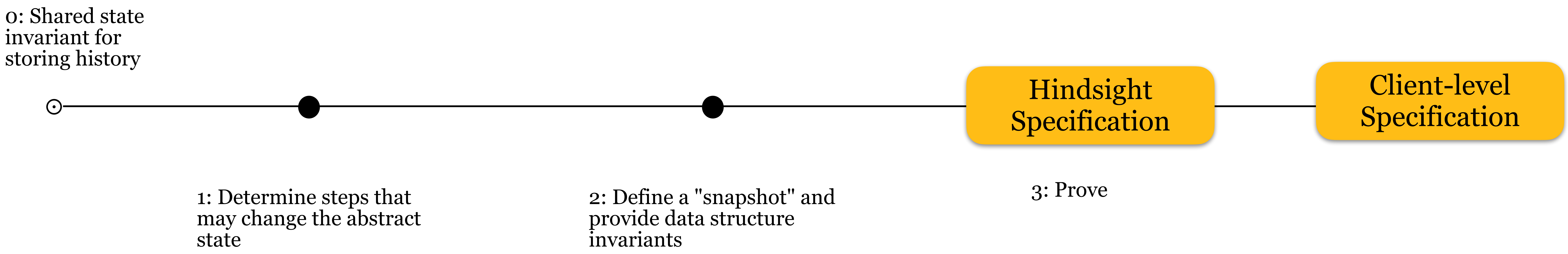
**Framework provides:**




**Proof author obligations:**

# Proof Author POV

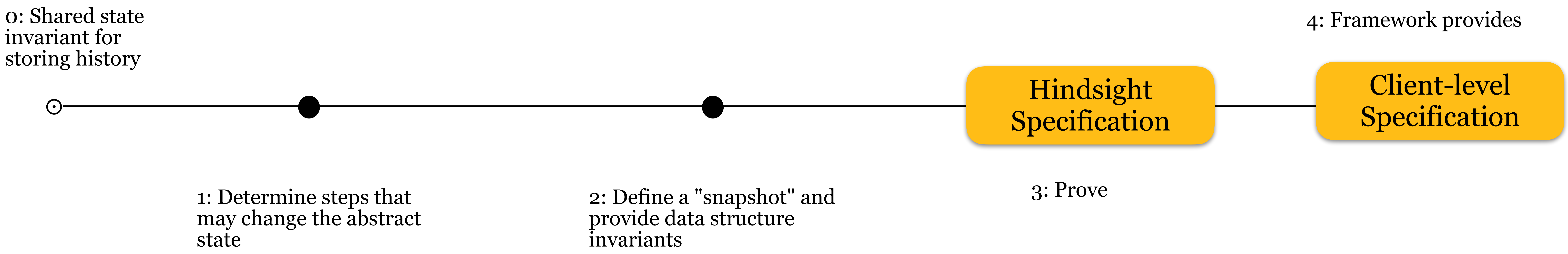
**Framework provides:**




**Proof author obligations:**

# Proof Author POV

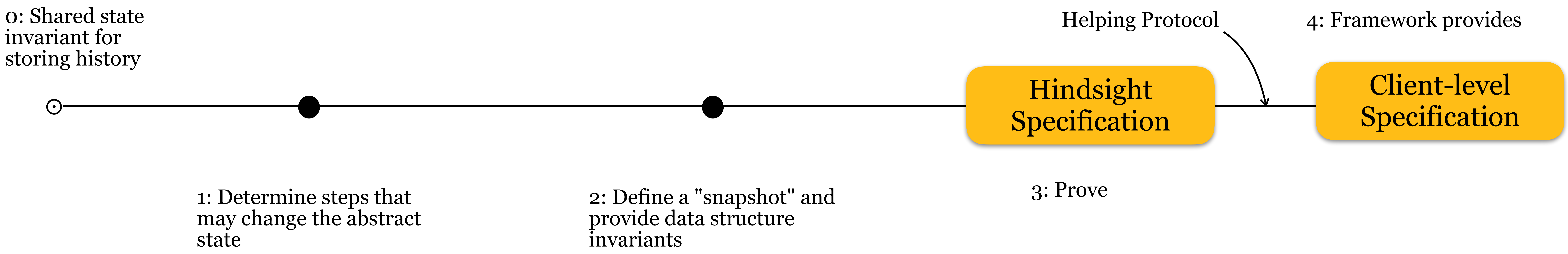
**Framework provides:**




**Proof author obligations:**

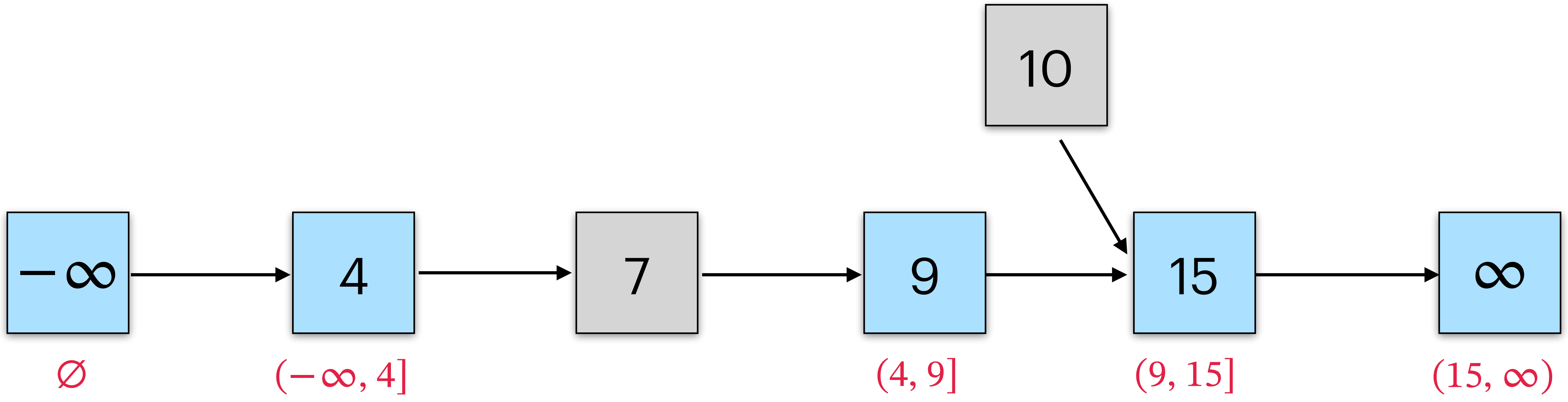
# Proof Author POV

**Framework provides:**




**Proof author obligations:**

# Global Graph Properties




Edgeset Framework :

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

Flow Framework :

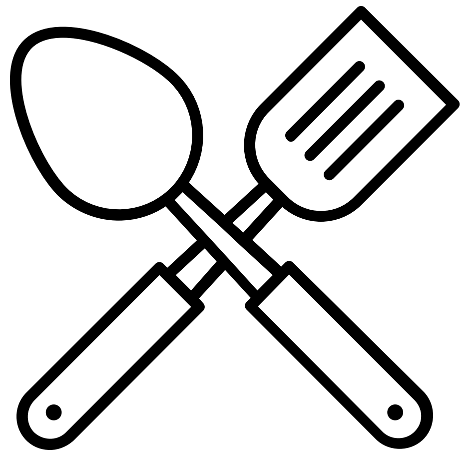
- Siddharth Krishna, Dennis E. Shasha and Thomas Wies. *Go with the flow: compositional abstractions for concurrent data structures*. [POPL 2018]
- Siddharth Krishna, Alexander J. Summers and Thomas Wies. *Local reasoning for global graph properties*. [ESOP 2020]
- Siddharth Krishna et al., *Verifying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

# Outline

Step 1:

Find a class of structures with common correctness reasoning

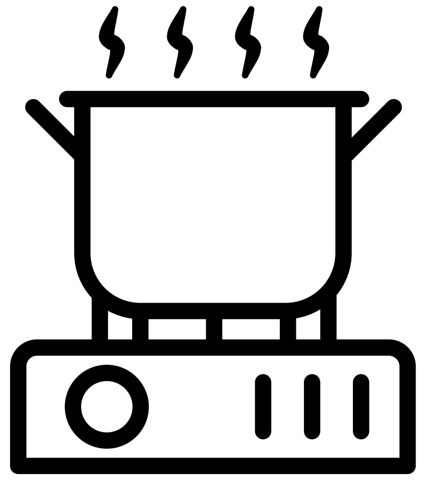
- ECOOP24 : (Lock-free) linked lists and skiplists



Step 2:

Develop enabling technology

- Template Algorithms
- Hindsight Framework



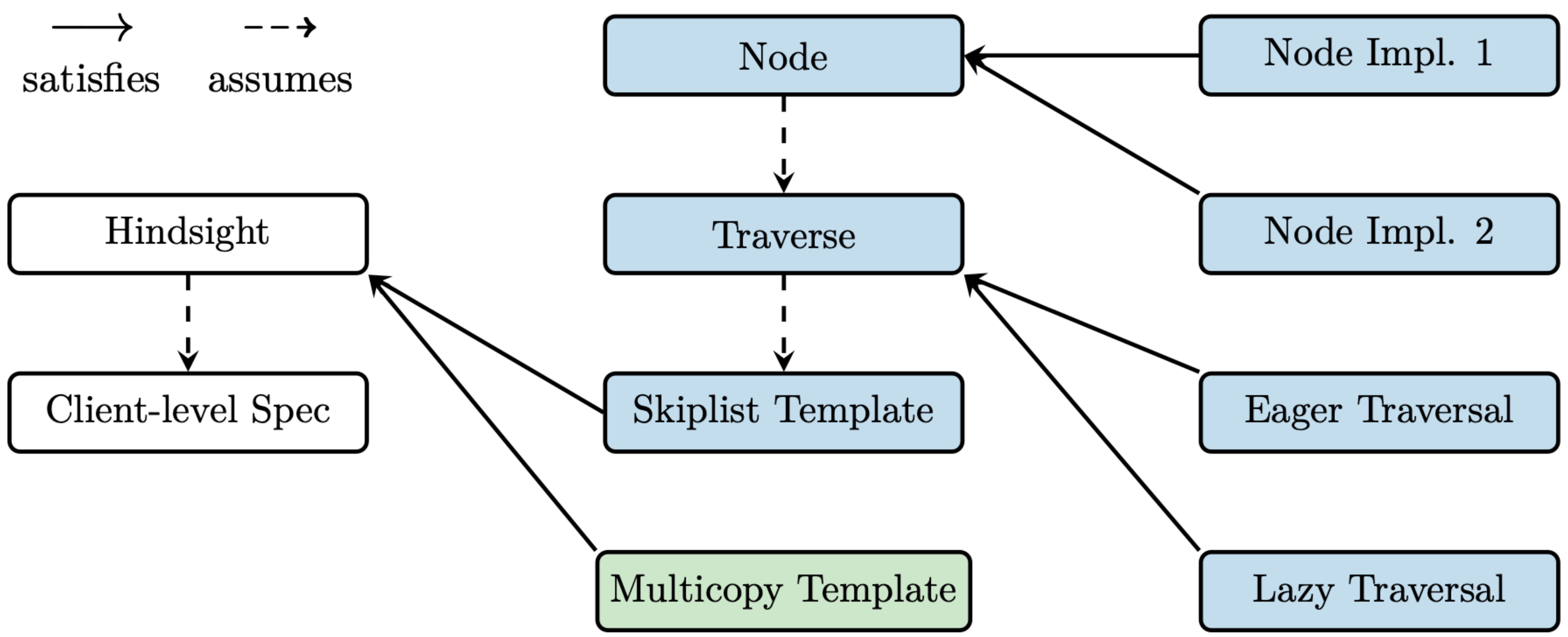
Step 3:

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, artifact available on Zenodo.




**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 |

# 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** |

# Thank you!

- Data Structure invariants
- Proof of the method

**Hindsight Specification**

Data Structure Agnostic

- Prophecy variables
- Helping Protocol

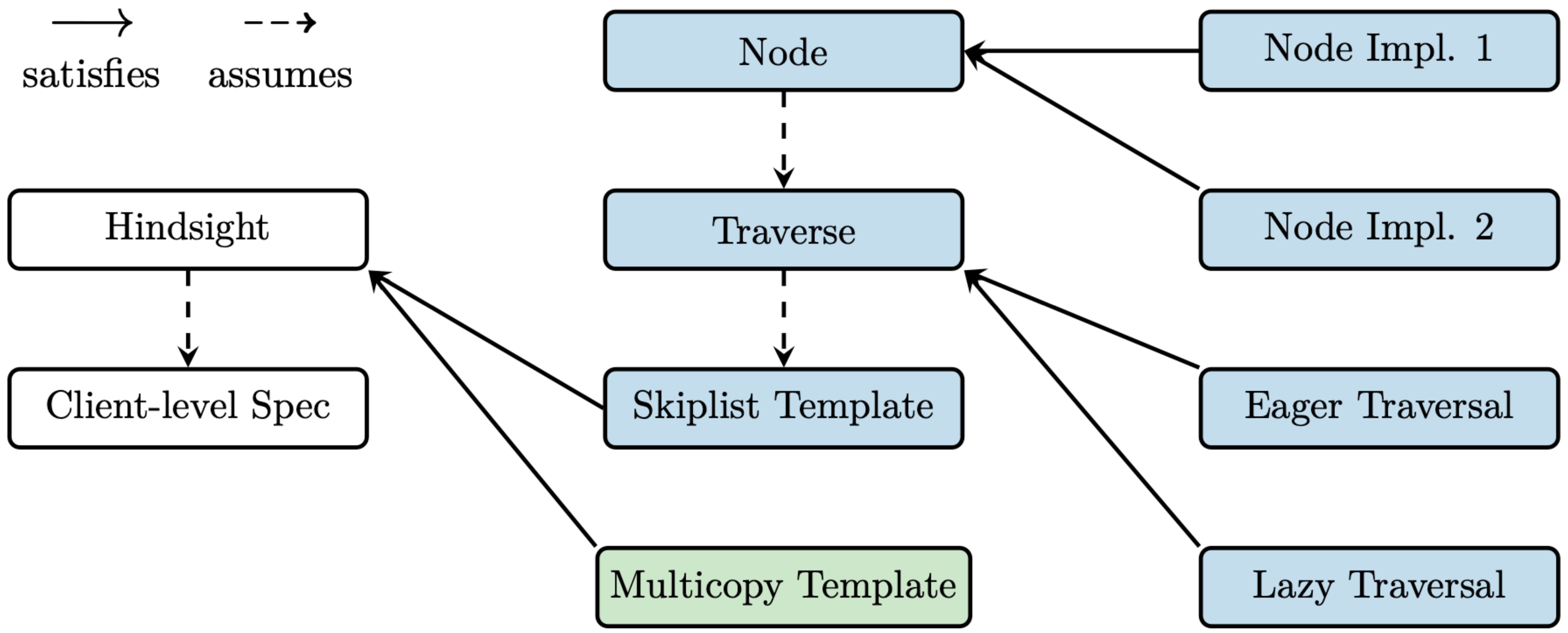
**Client-level Specification**

## Template Algorithms

```
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
```

Iris



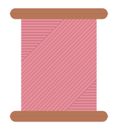
# Backup Slides

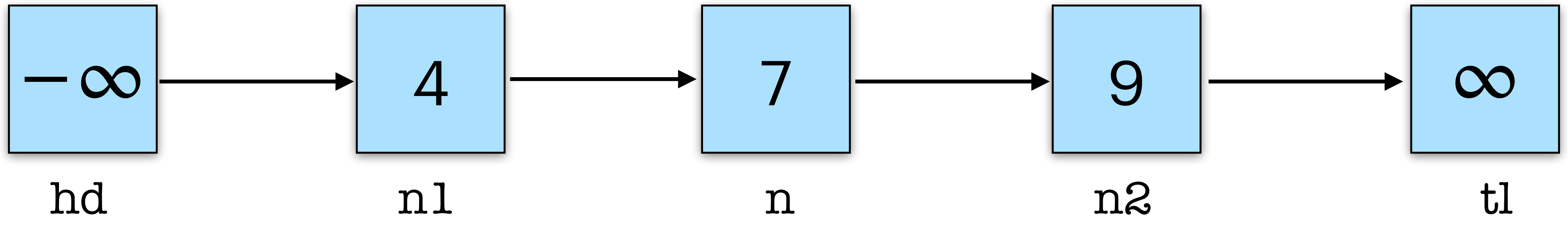
# Template Algorithms

```
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
```

delete(7)



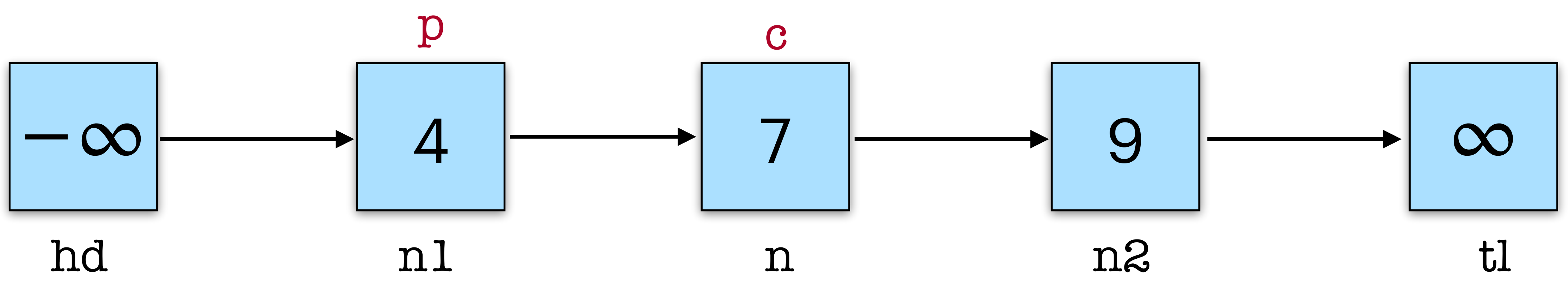


# Template Algorithms

```
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
```

delete(7)

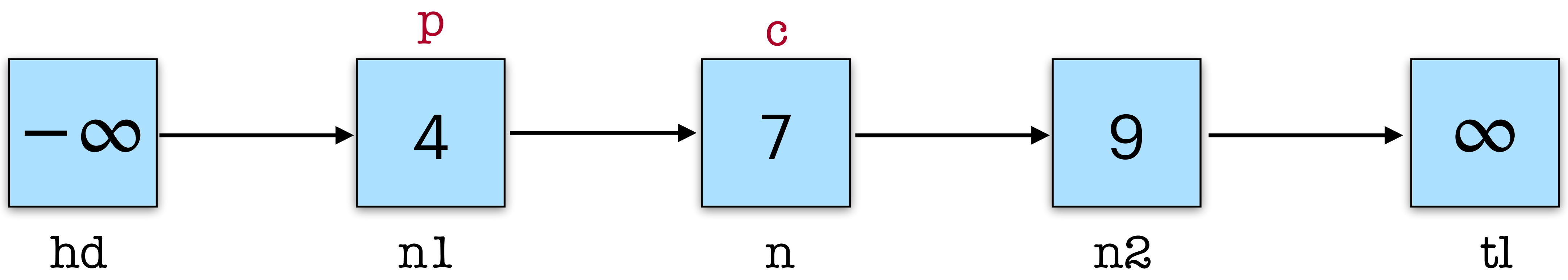


# Template Algorithms

```
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
```

delete(7)

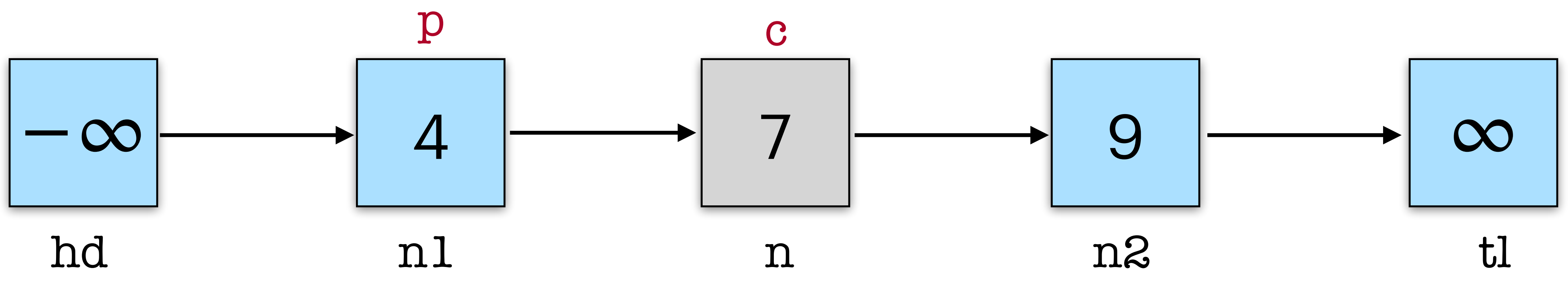


# Template Algorithms

```
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
```

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
```

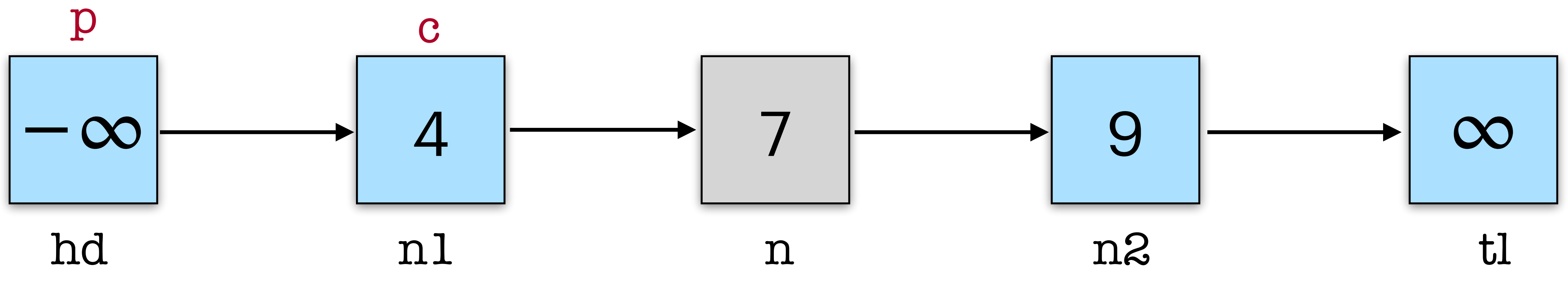
```
let maintainanceOp_ins_rec i h pm ps cs e =
  if i < h-1 then
    let idx = pm[i] in
    let p = ps[idx] in
    let c = cs[idx] in
    match changeNext idx p c e with
    | Success ->
      maintainanceOp_ins_rec (i+1) h pm ps cs e
    | Failure ->
      traverse ps cs k;
      maintainanceOp_ins_rec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

```
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
```

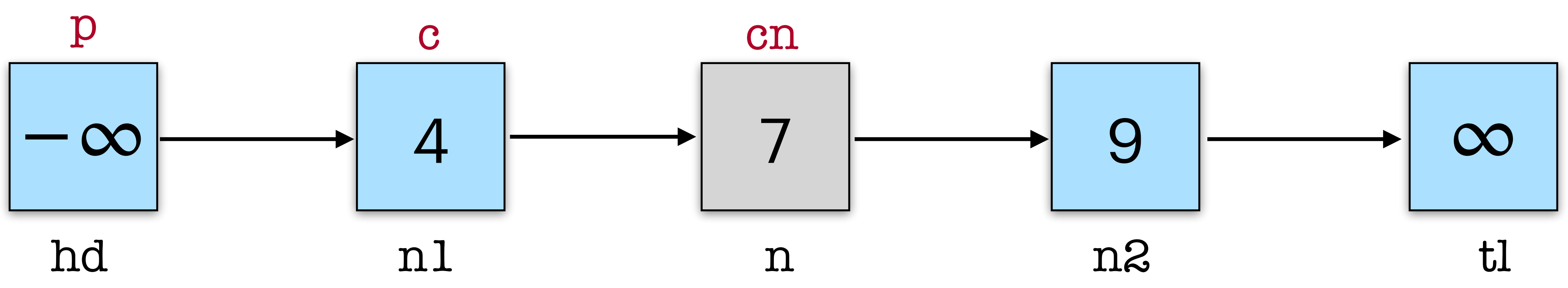
search(9)



# Template Algorithms

```
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
```

search(9)

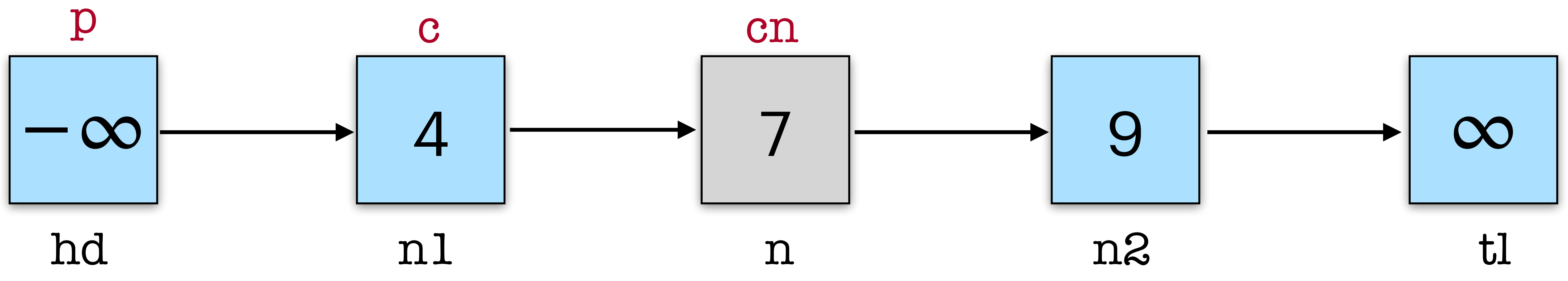


# Template Algorithms

```
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
```

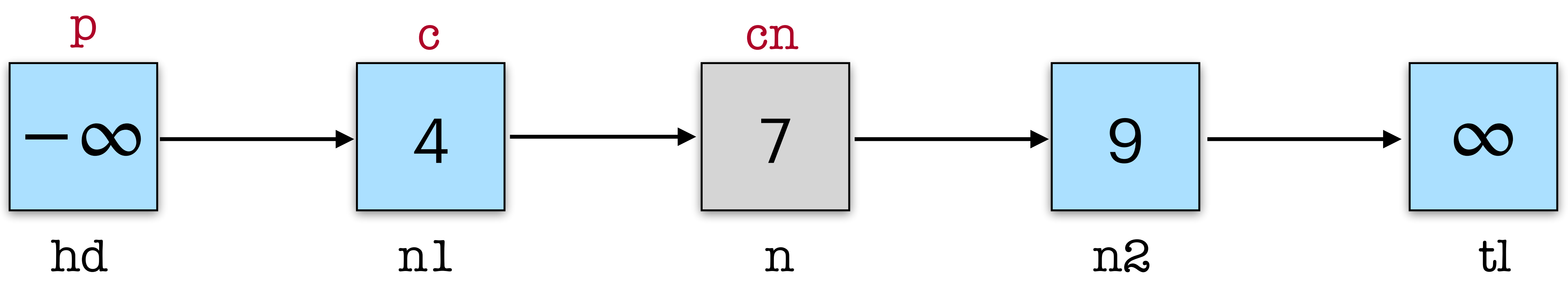
search(9)



# Template Algorithms

```
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
```

search(9)

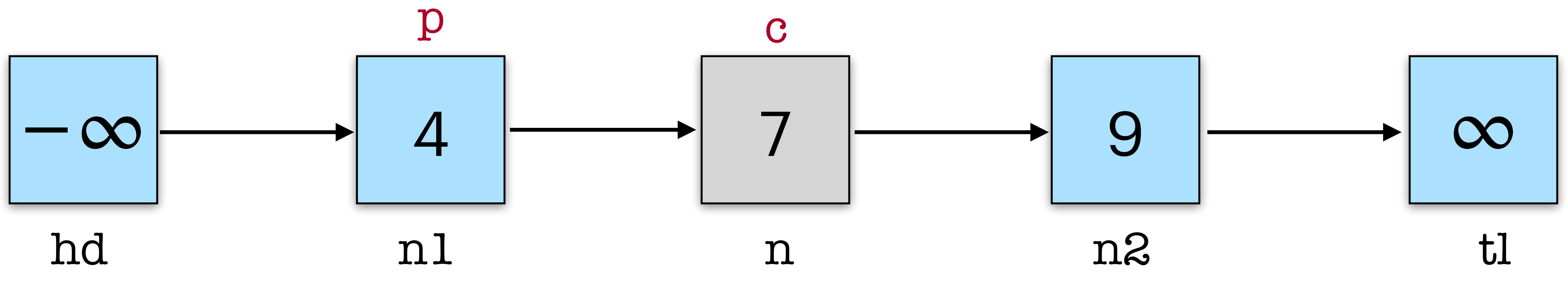


# Template Algorithms

```
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
```

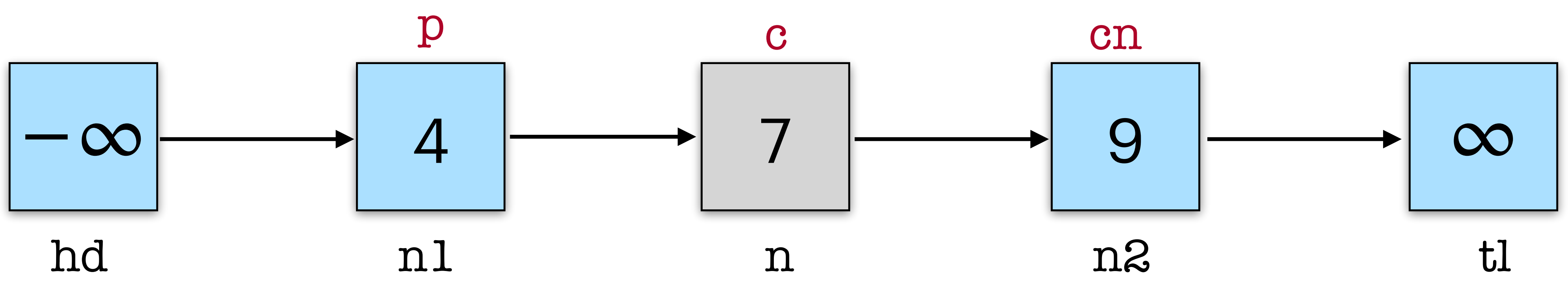
search(9)



# Template Algorithms

```
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
```

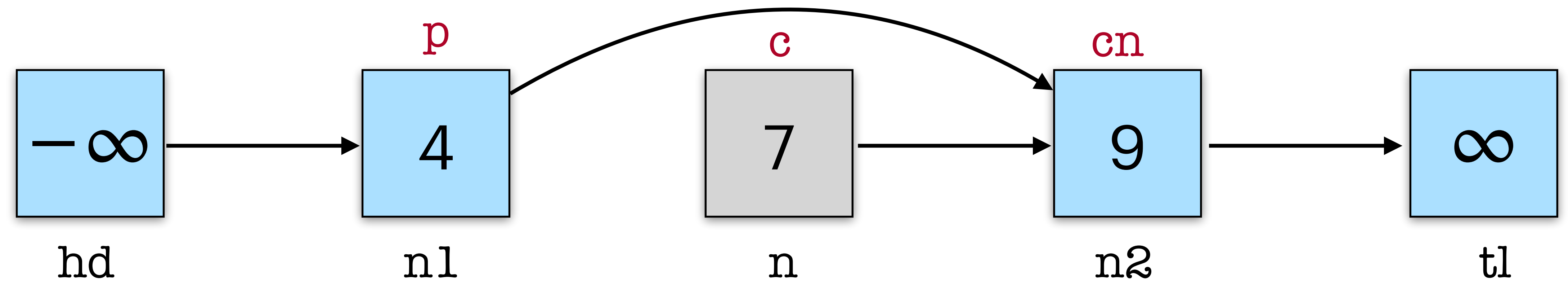
search(9)



# Template Algorithms

```
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
```

search(9)

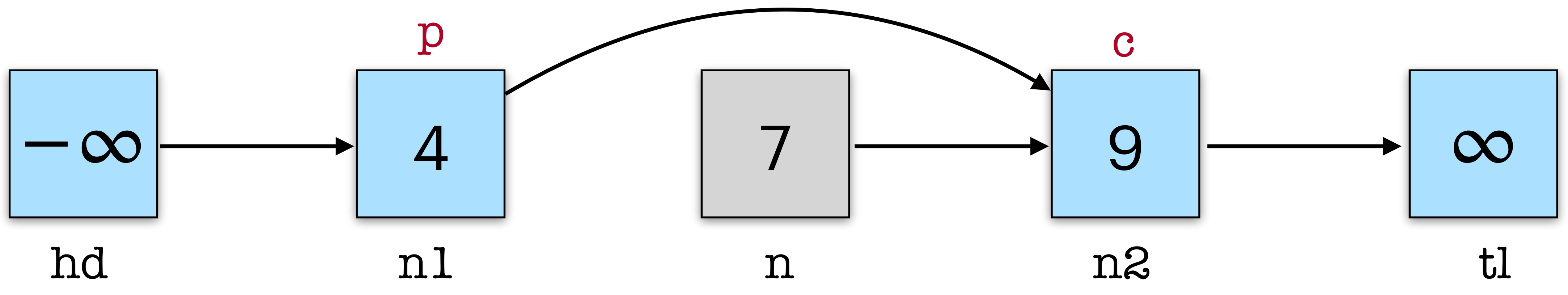


# Template Algorithms

```
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
```

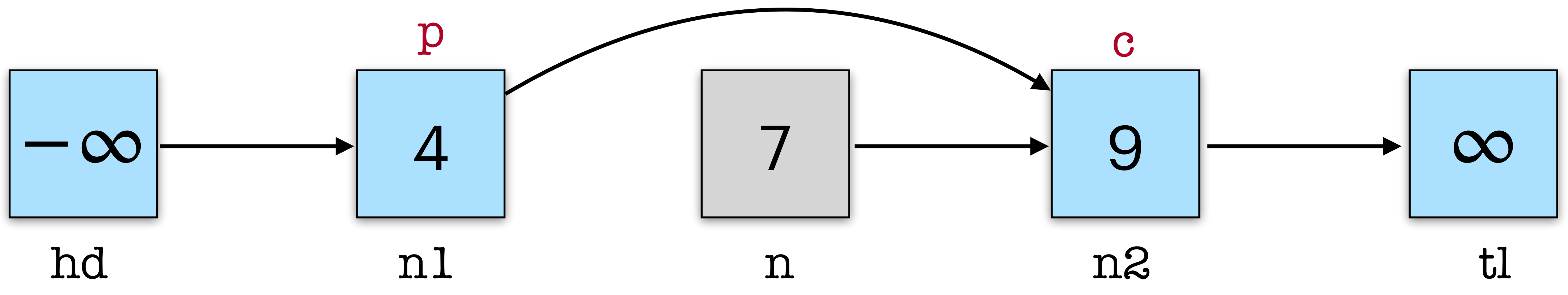
search(9)



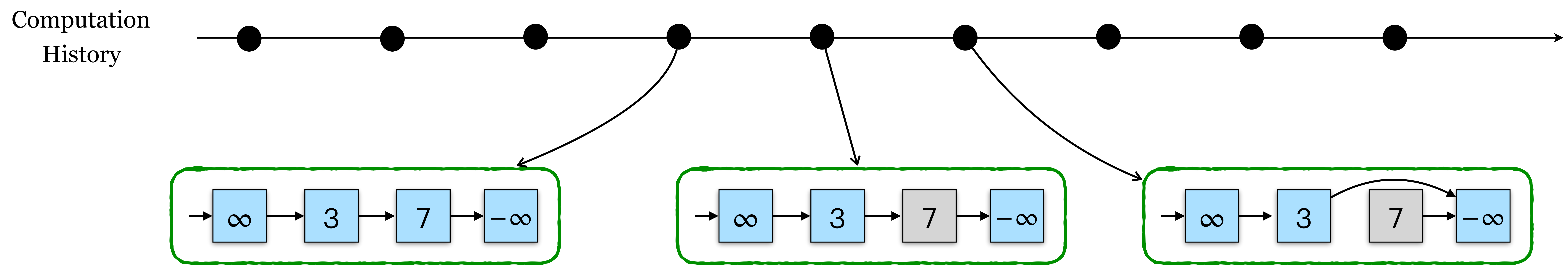
# Template Algorithms

```
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
```

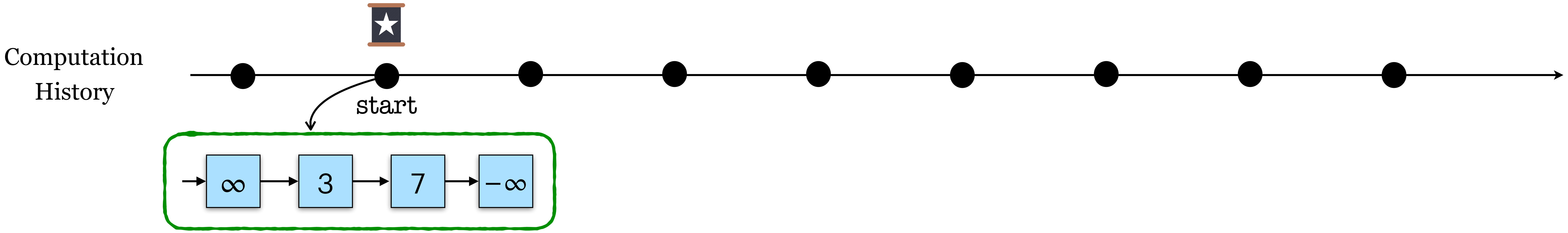
search(9)

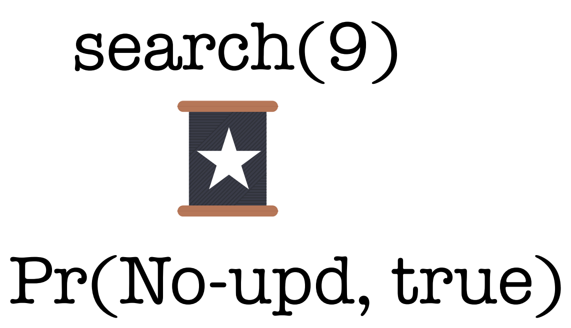


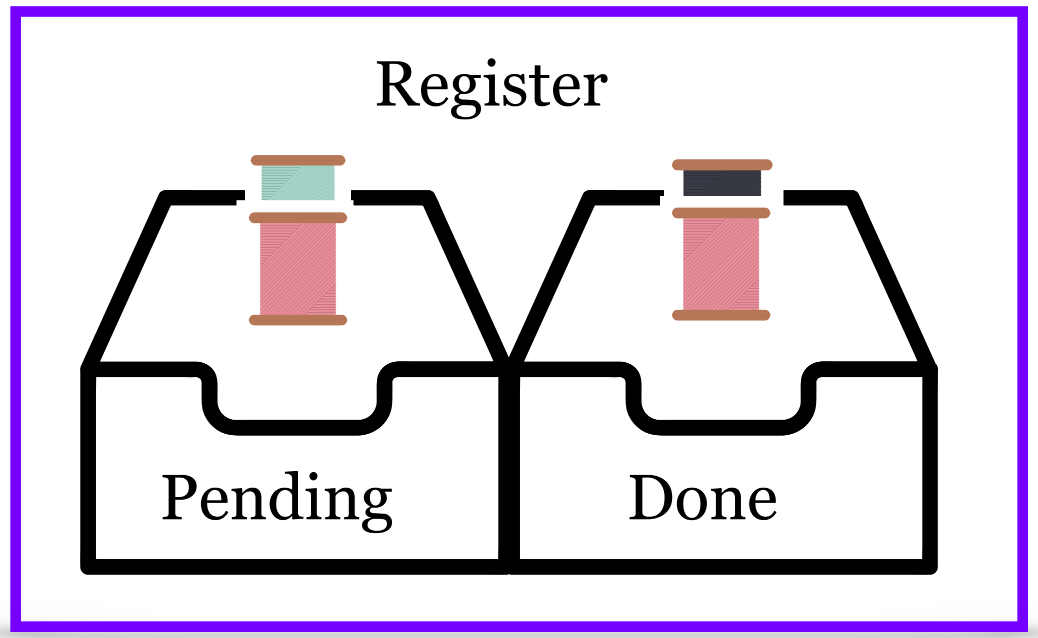
# Helping Protocol



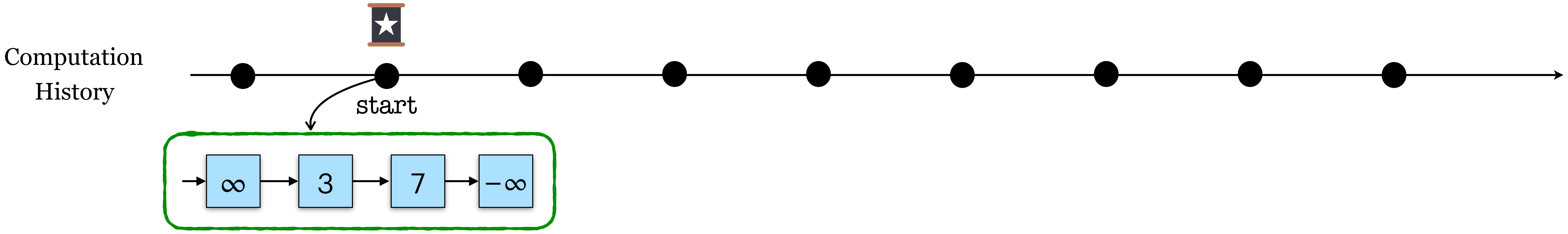
# Helping Protocol

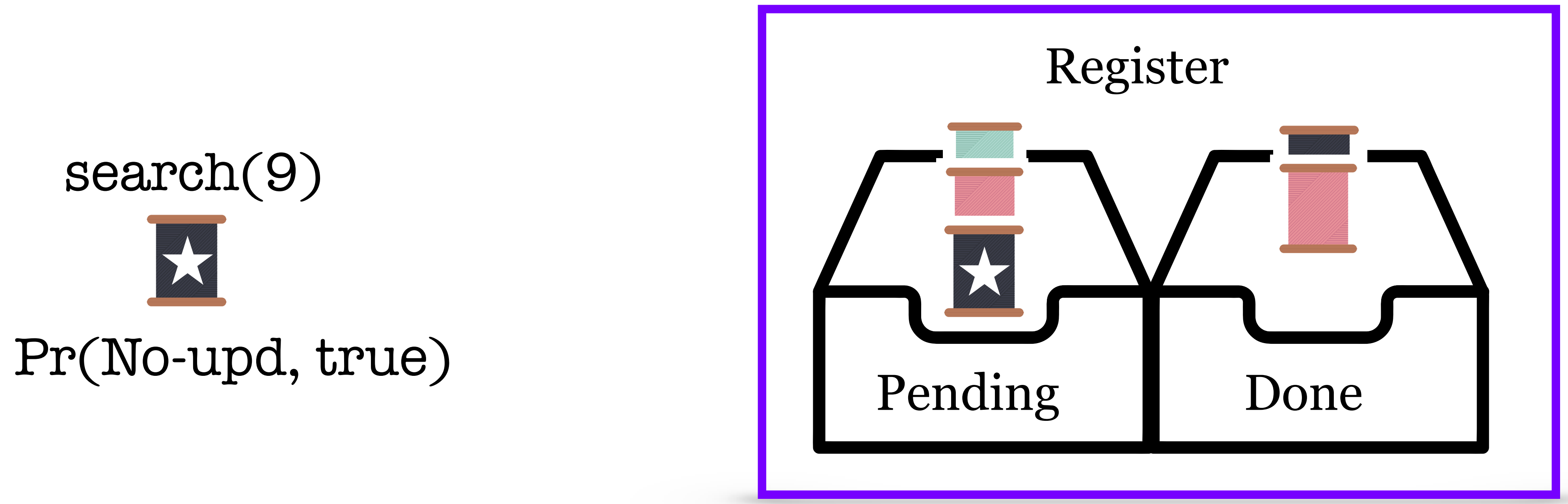


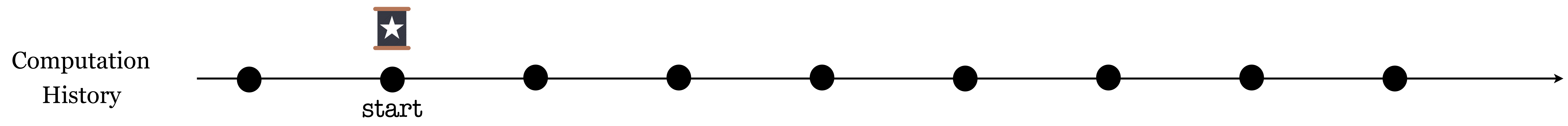



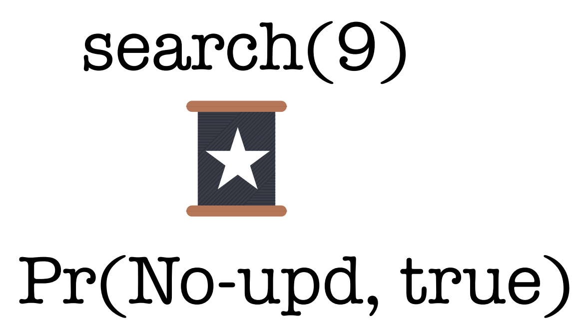
# Helping Protocol

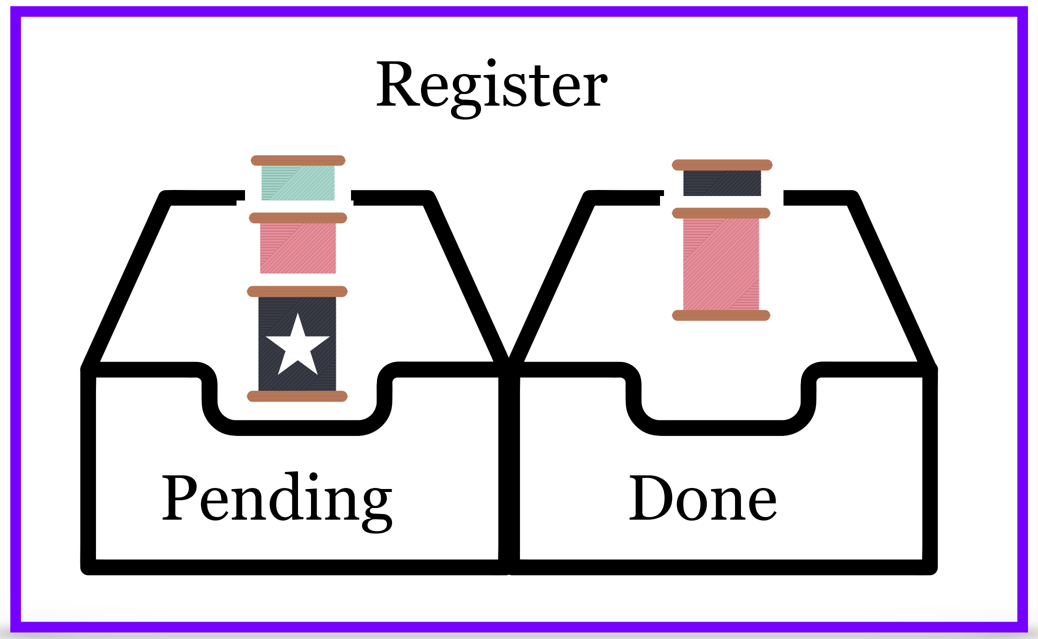


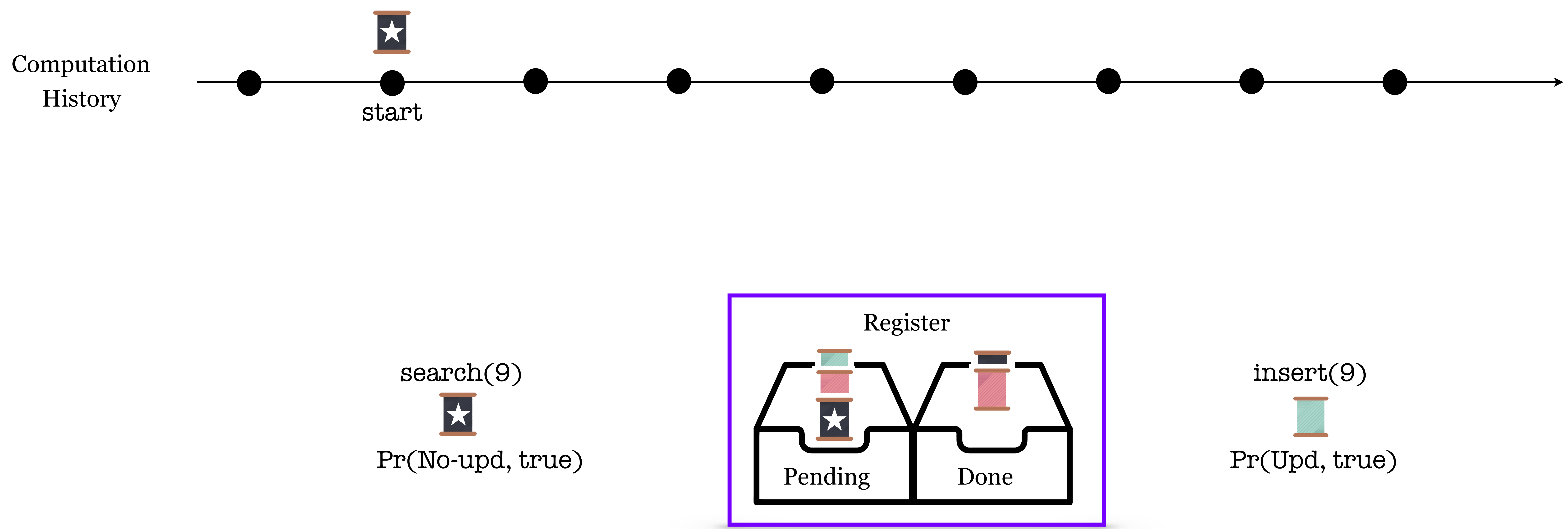
# Helping Protocol



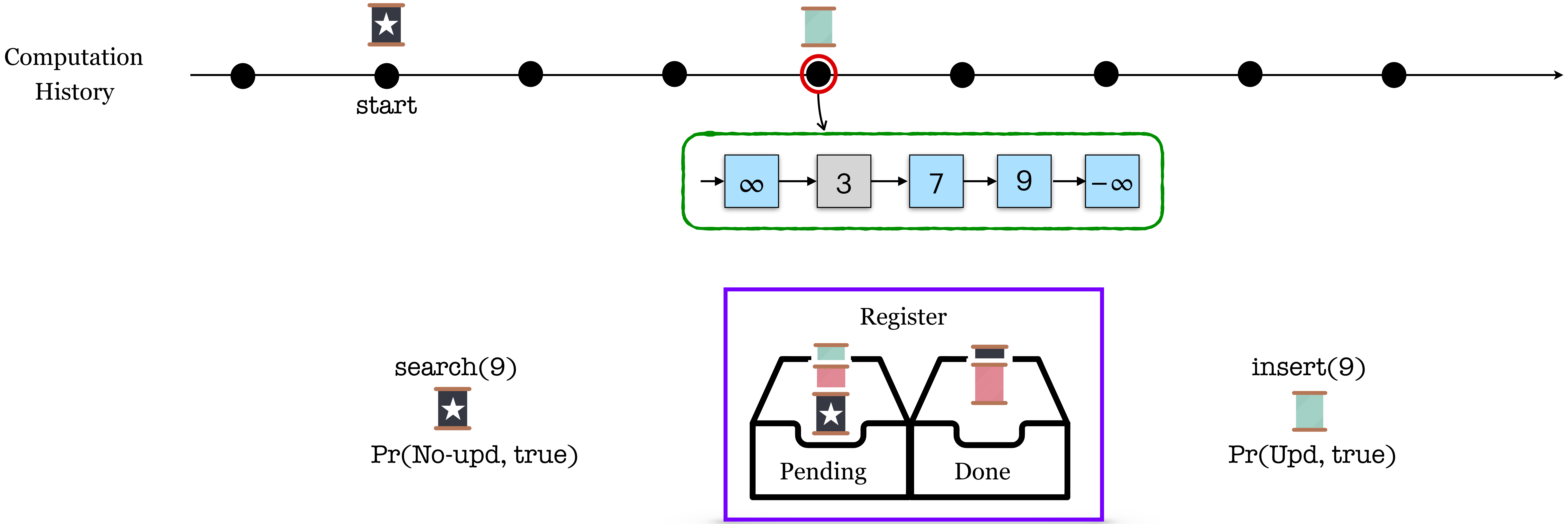



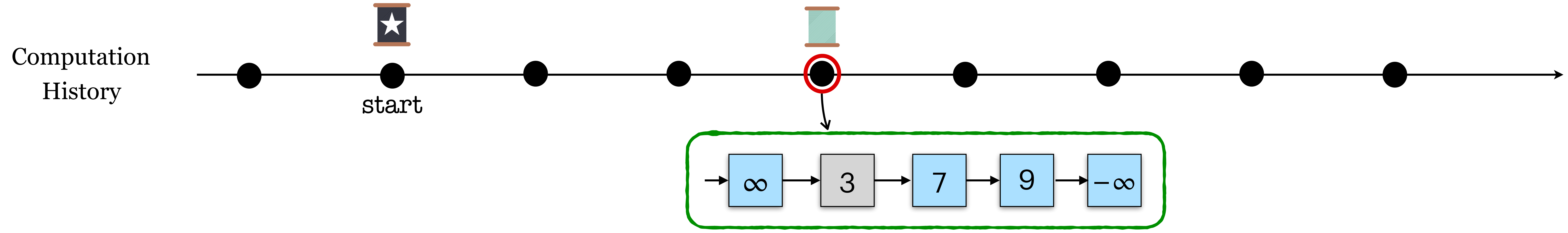
# Helping Protocol

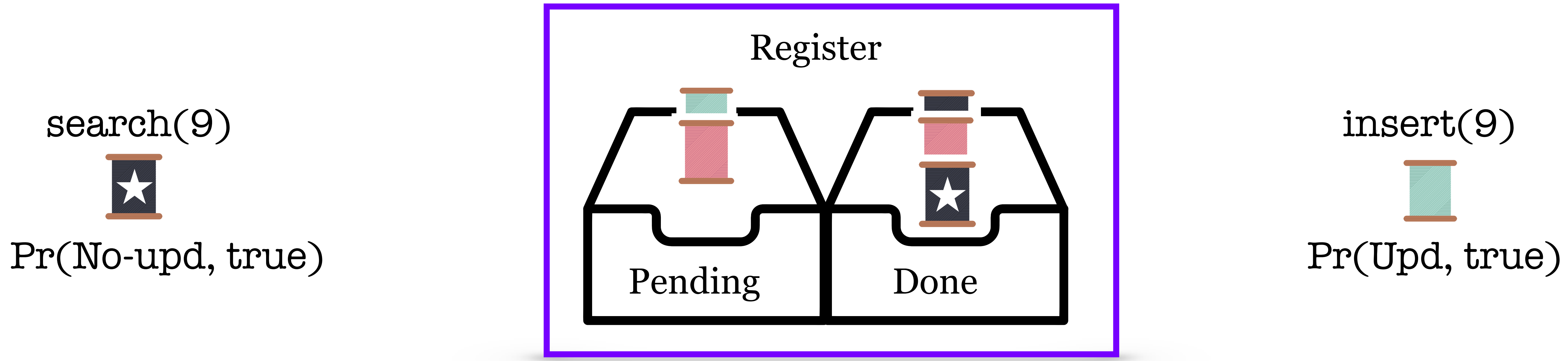


# Helping Protocol

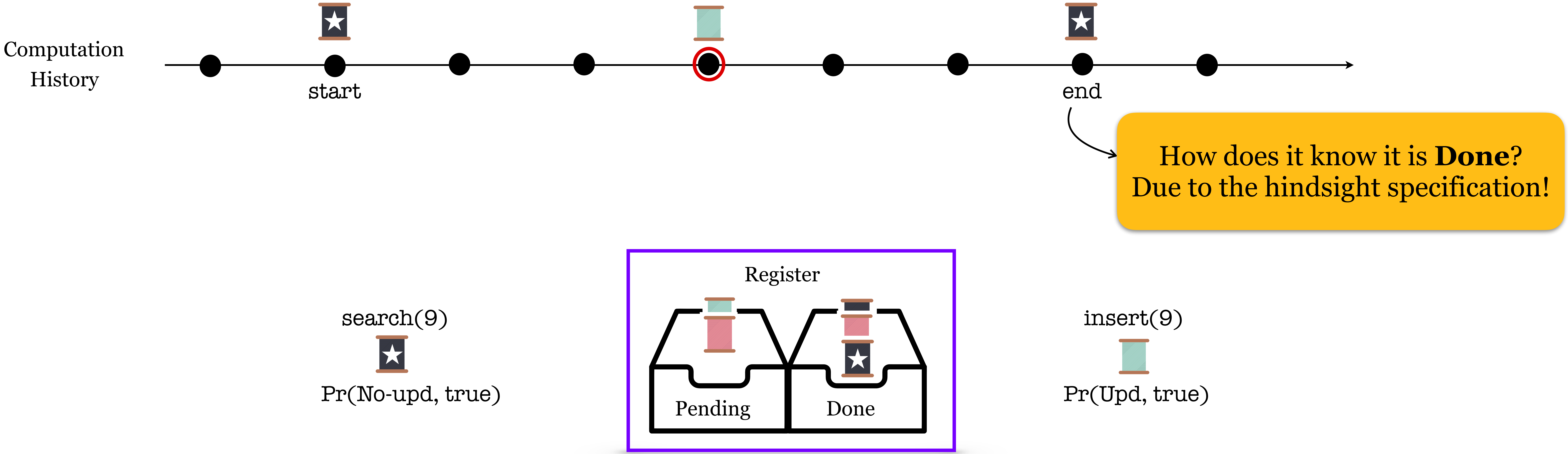


# Helping Protocol



# Helping Protocol



# Hindsight Framework

- Data Structure invariants
- Proof of the method

**Hindsight Specification**

- Prophecy variables
- Helping Protocol

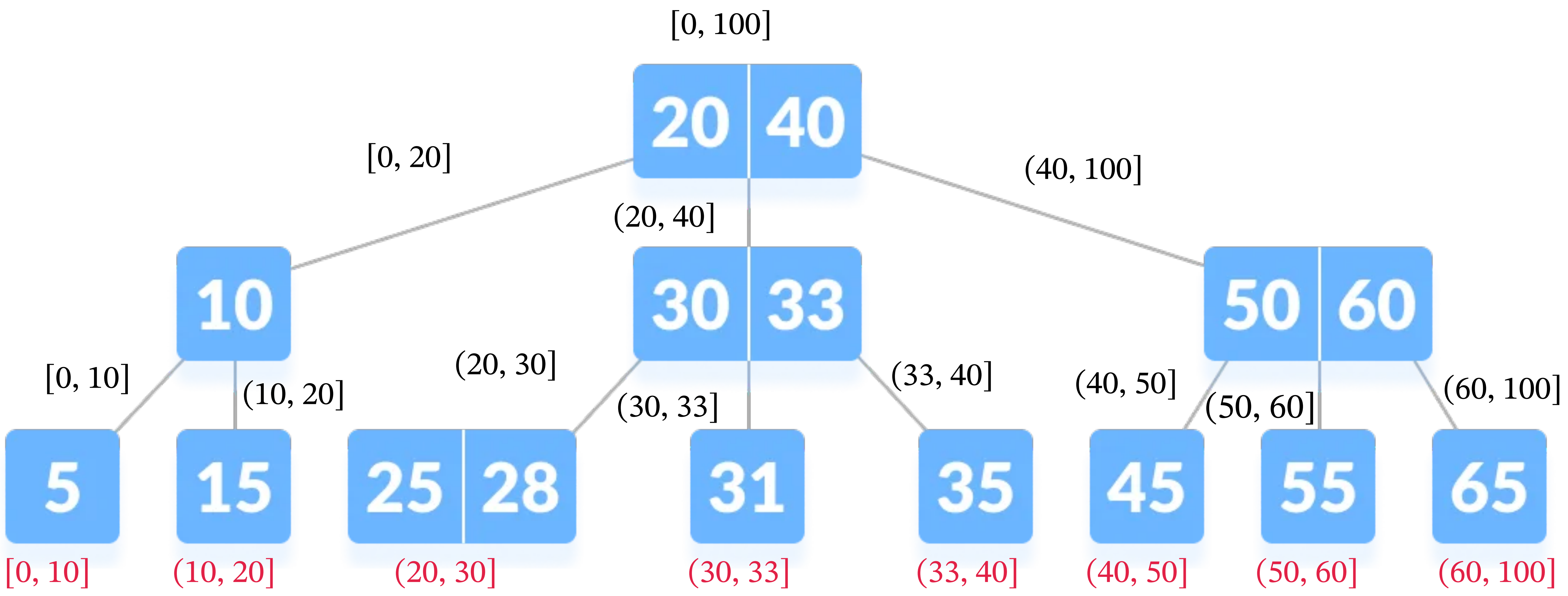
**Client-level Specification**

**Hindsight Specification** :

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




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

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

# Template Algorithms

```
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
```

Approach :

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

$\{ \text{Node}(n, k, m, n') \}$ markNode 0 n $\{ \text{Node}(n, k, m[0 \mapsto \text{true}], n') \}$