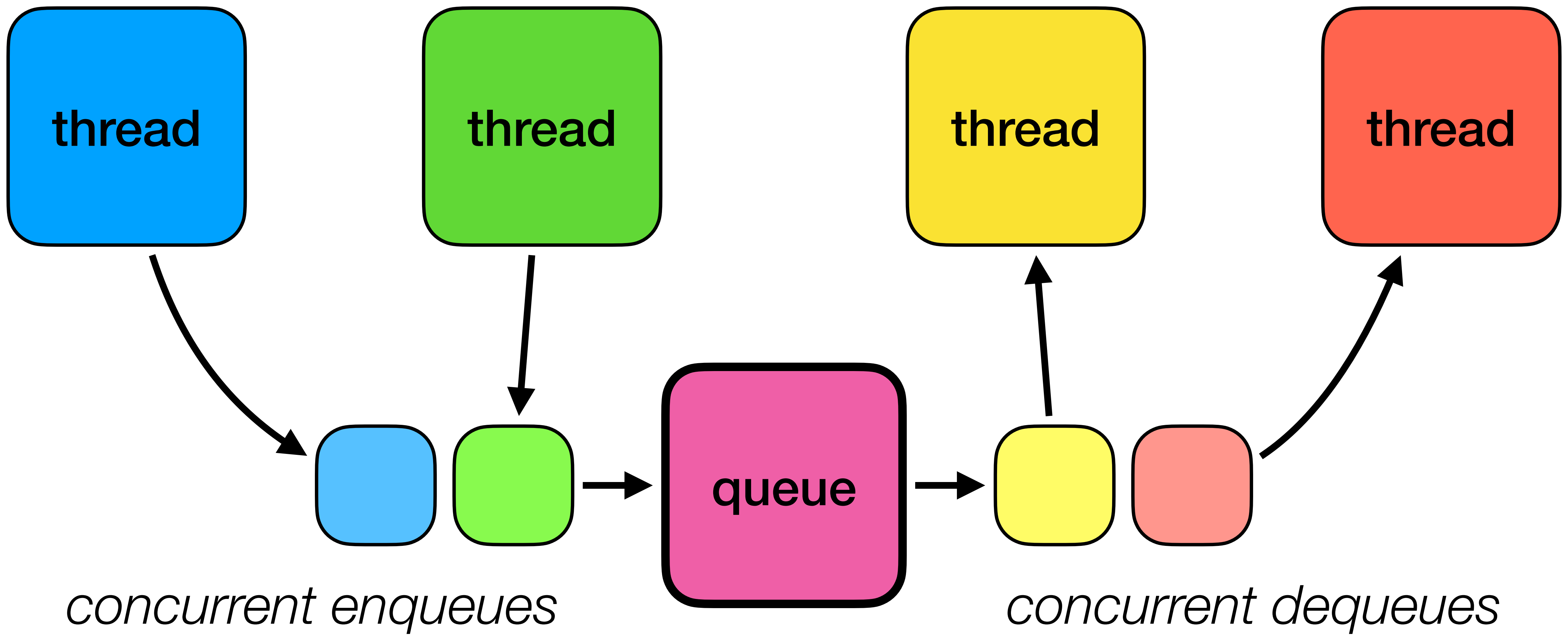
CHECKING LINEARIZABILITY: THEORETICAL LIMITS

Constantin Enea
Ecole Polytechnique

Concurrent Objects

Multi-threaded programming

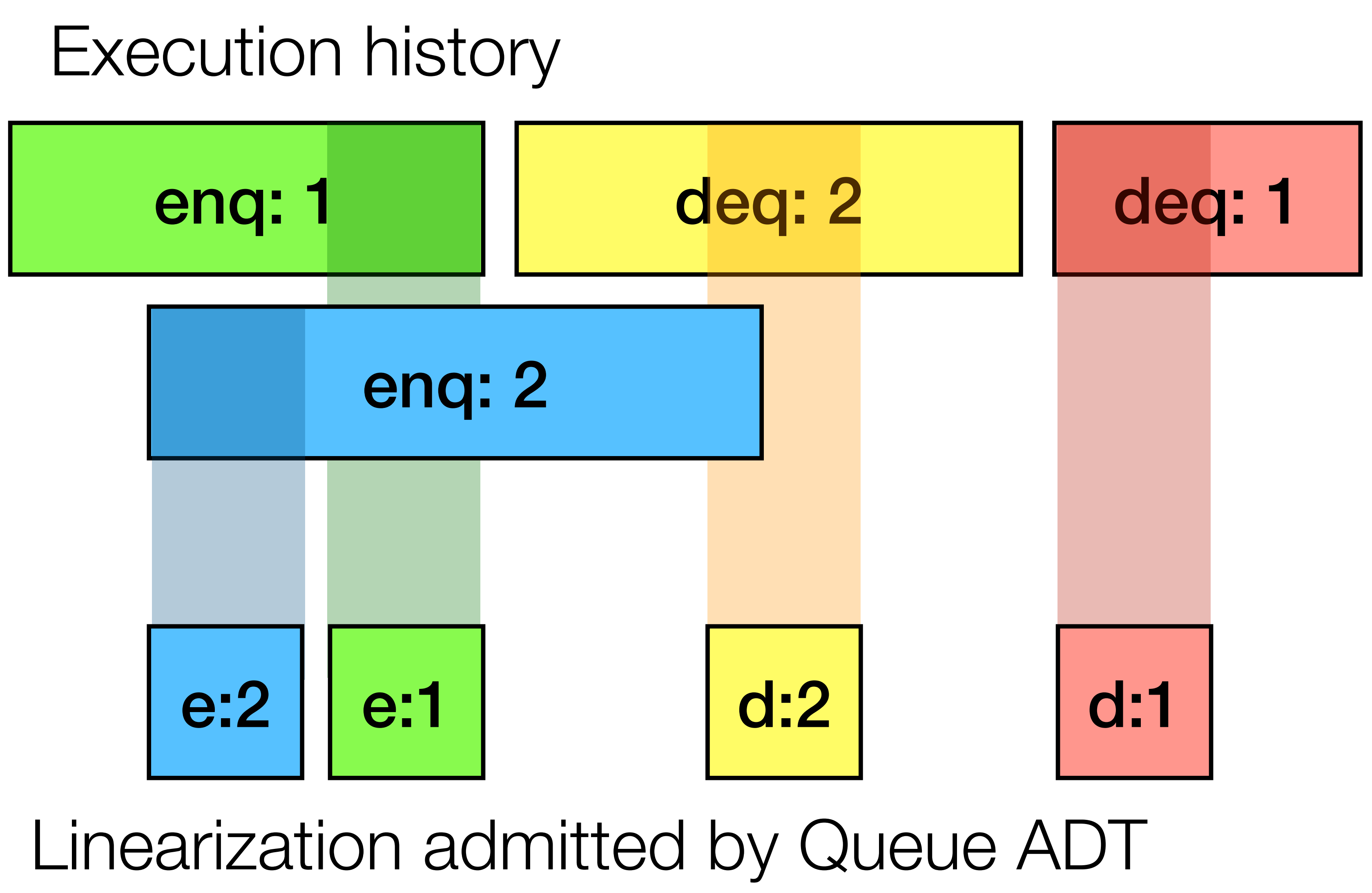


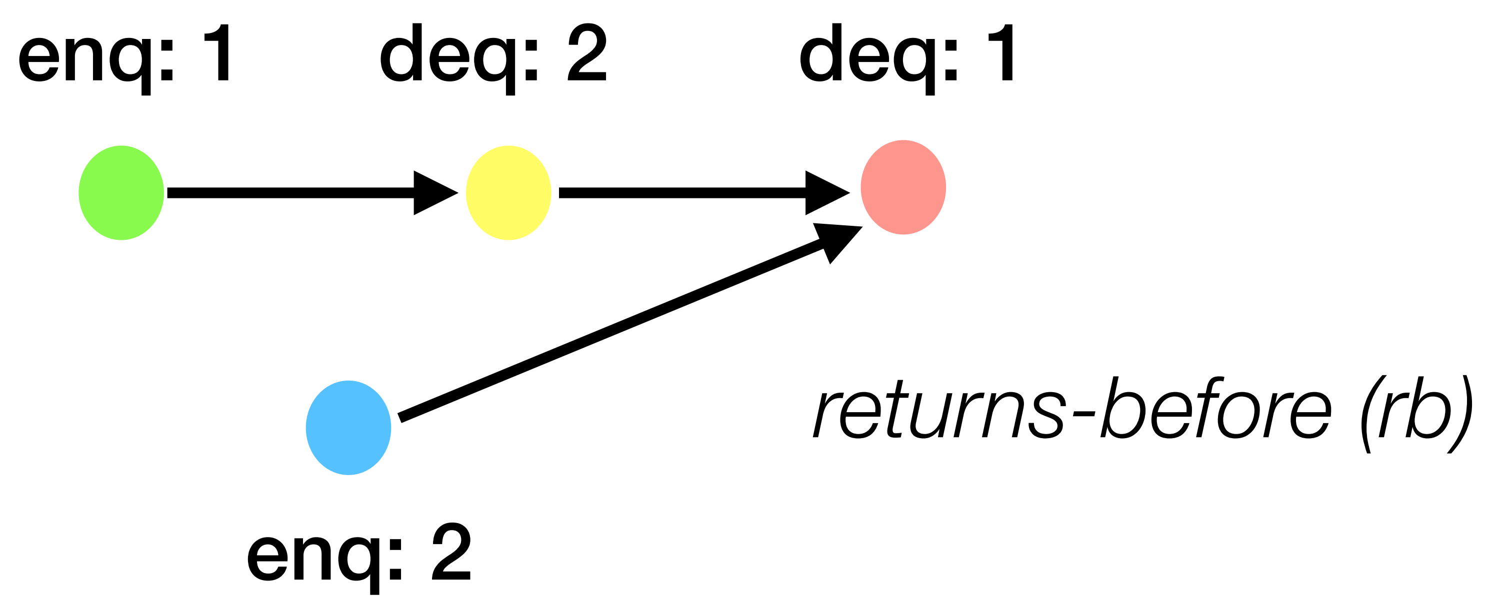
e.g. Java Development Kit SE

dozens of objects, including queues, maps, sets, lists, locks, atomic integers, …

Linearizability [Herlihy&Wing 1990]

Effects of each invocation appear to occur instantaneously

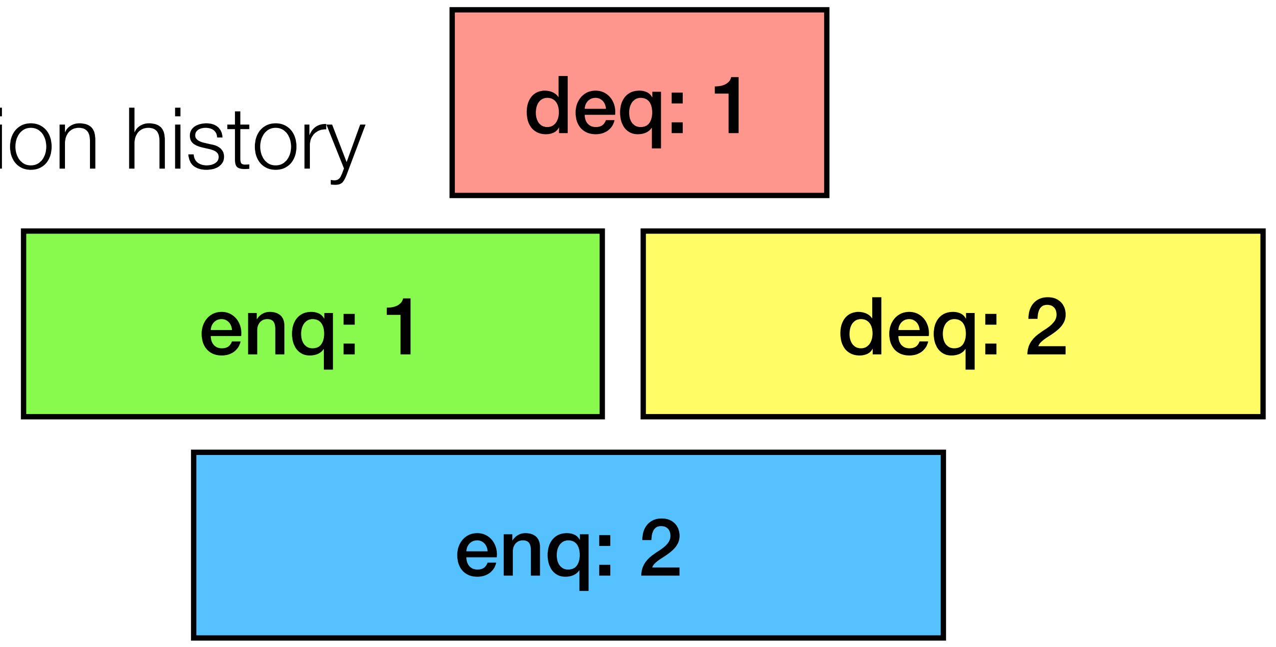




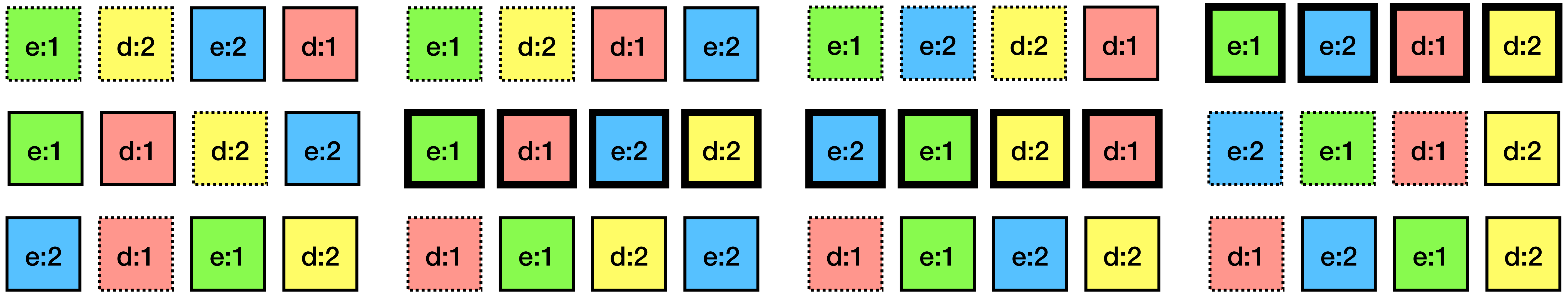
∃ lin. rb ⊆ lin ∧ lin ∈ Queue ADT

Complexity of Testing Linearizability

Execution history



Exponentially many linearizations to consider



Theorem [Gibbons.et.al.'97]

Checking linearizability for a fixed execution is NP-hard

Checking Linearizability: Complexity (finite-state implementations)

Bounded Nb. of Threads:

- EXSPACE-complete [Alur et al., 1996, Hamza 2015]

Unbounded Nb. of Threads:

- Undecidable [Bouajjani et al., 2013]
- Decidable with “fixed linearization points” [Bouajjani et al. 2013]

Alur et al. 1996: Rajeev Alur, Kenneth L. McMillan, Doron A. Peled: Model-Checking of Correctness Conditions for Concurrent Objects. LICS 1996

Bouajjani et al., 2013: Ahmed Bouajjani, Michael Emmi, Constantin Enea, Jad Hamza: Verifying Concurrent Programs against Sequential Specifications. ESOP 2013

Hamza 2015: Jad Hamza: On the Complexity of Linearizability. NETYS 2015

Checking Linearizability: Complexity (finite-state implementations)

Bounded Nb. of Threads:

- EXSPACE-complete [Alur et al., 1996, Hamza 2015]

Unbounded Nb. of Threads:

- Undecidable [Bouajjani et al., 2013]
- Decidable with “fixed linearization points” [Bouajjani et al. 2013]

Concurrent Languages

- Concurrent language = (Σ,D), where Σ is an alphabet, $D \subseteq \Sigma \times \Sigma$ (Mazurkiewicz traces - D is symmetric)
- a and b are called independent when $(a,b) \notin D$
- $\Rightarrow_D$ a relation that permutes independent symbols: for all $(a,b) \notin D$, $\sigma\, ab\, \sigma' \Rightarrow_D \sigma'\, ba\, \sigma$ (and trans. closure)
- $cl_D(L)$ = all strings σ' such that $\sigma' \Rightarrow_D \sigma$ for some $\sigma \in L$
- Ex: $\Sigma = \{a,b\}$, $L=(ab)^*$, $D=\varnothing$ and $D=\{(b,a)\}$

Specifications, Implementations

- **Specification** = a language over an alphabet containing symbols $p{:}m(a) \Rightarrow b$
  - Example: bounded-value register, bounded size queue
- **Implementation** = a language over an alphabet containing symbols $p{:}call\ m(a)$ and $p{:}ret\ m(a) \Rightarrow b$ where returns “match” previous calls
- $\Sigma_p = (\ \Sigma_{call}(p) \cup \Sigma_{ret}(p)\ )$ and $\Sigma = \cup_p \Sigma_p$

Defining Linearizability

- lin = $\bigcup_p (\Sigma_p \times \Sigma_p) \cup (\Sigma_{ret} \times \Sigma_{call})$
- Spec* = replacing $p{:}m(a){\Rightarrow}b$ with call/ret actions
- an execution σ is **linearizable** iff $\sigma \in cl_{lin}(Spec^*)$
- Impl is linearizable iff $Impl \subseteq cl_{lin}(Spec^*)$
  - this inclusion check is undecidable in general (for regular languages)

Defining Linearizability

- **Linearizability**: an execution σ is linearizable iff there is a sequence τ that contains σ and linearization points (symbols $p{:}m(a) \Rightarrow b$) s.t.:
  - every projection over "actions" of the same process is "sequential"
  - the projection over linearization point actions is included in the specification

Defining Linearizability

- $lin = \cup_p (\Sigma_p \times \Sigma_p) \cup (\Sigma_{ret} \times \Sigma_{call})$
- Spec* = replacing $p{:}m(a) \Rightarrow b$ with call/ret actions
- an execution σ is **linearizable** iff $\sigma \in cl_{lin}(Spec^*)$
- Impl is linearizable iff $Impl \subseteq cl_{lin}(Spec^*)$
  - this inclusion check is undecidable in general (for regular languages)
- $cl_{lin}(Spec^*) = (||_p L_{lin\_points}(p) \, || \, Spec) \downarrow (\Sigma_{call} \cup \Sigma_{ret})$

EXPSPACE-hardness

Problem 2 (Letter Insertion). Input: A set of *insertable* letters $A = \{a_1, \ldots, a_l\}$. An NFA N over an alphabet $\Gamma \uplus A$.

Question: For all words $w \in \Gamma^*$, does there exist a decomposition $w = w_0 \cdots w_l$, and a permutation p of $\{1, \ldots, l\}$, such that $w_0 a_{p[1]} w_1 \ldots a_{p[l]} w_l$ is accepted by N?

Reducing Letter Insertion to Linearizability:

1. there exists a word w in Γ^*, such that there is no way to insert the letters from A in order to obtain a word accepted by N
2. there exists an execution of *Lib* with k threads which is not linearizable w.r.t. S_N

k = l+2

EXPSPACE-hardness

Define k, the number of threads, to be $l+2$.
We will define a library Lib composed of

- methods $M_1, \ldots, M_l$, one for each letter of A
- methods M_γ, one for each letter of Γ
- a method M_{Tick}.

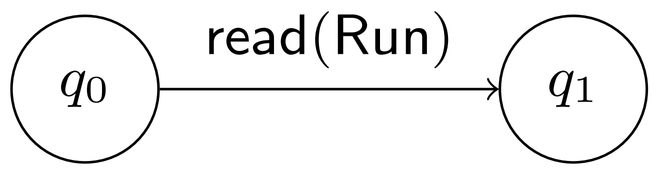


Fig. 4. Description of M_γ, $\gamma \in \Gamma$

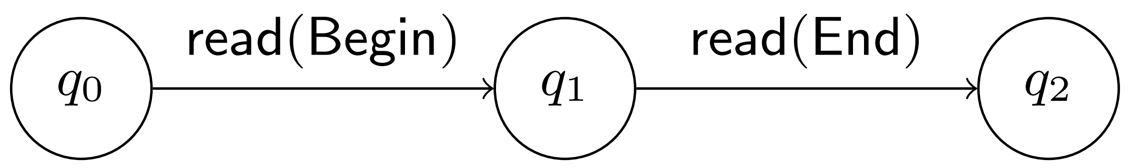


Fig. 5. Description of $M_1, \ldots, M_l$

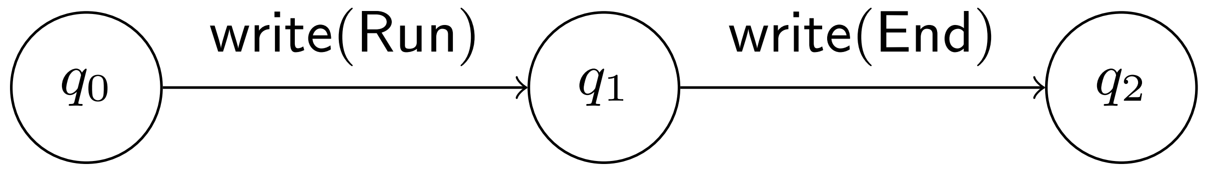


Fig. 6. Description of M_{Tick}

The specification S_N is defined as the set of words w over the alphabet $\{M_1, \ldots, M_l\} \cup \{M_{\mathsf{Tick}}\} \cup \{M_\gamma | \gamma \in \Gamma\}$ such that one the following condition holds:

- w contains 0 letter M_{Tick}, or more than 1, or
- for a letter M_i, $i \in \{1, \ldots, l\}$, w contains 0 such letter, or more than 1, or
- when projecting over the letters M_γ, $\gamma \in \Gamma$ and M_i, $i \in \{1, \ldots, l\}$, w is in N_M, where N_M is N where each letter γ is replaced by the letter M_γ, and where each letter a_i is replaced by the letter M_i.

EXPSPACE-hardness

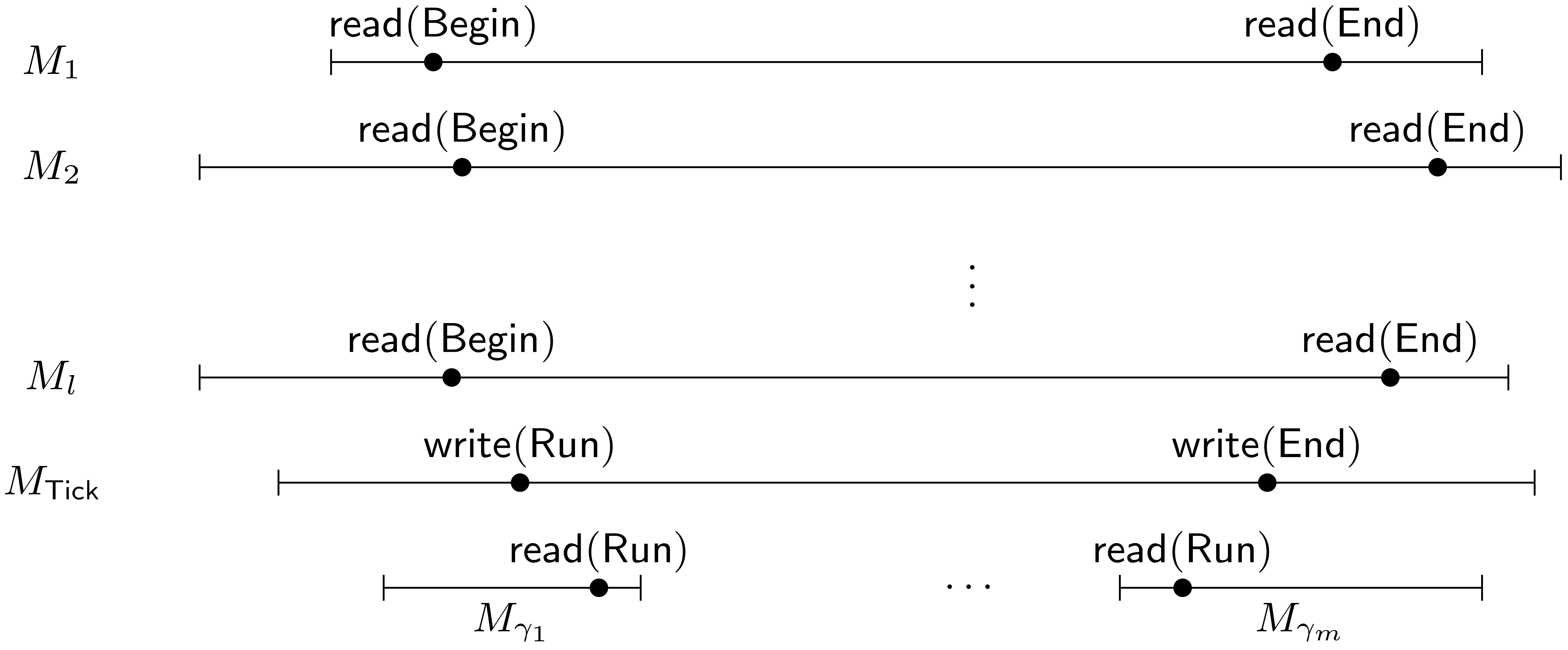


Fig. 7. Non-linearizable execution corresponding to a word $\gamma_1 \dots \gamma_m$ in which we cannot insert the letters from $A = \{a_1, \dots, a_l\}$ to make it accepted by N. The points represent steps in the automata.

Checking Linearizability: Complexity (finite-state implementations)

Bounded Nb. of Threads:

- EXSPACE-complete [Alur et al., 1996, Hamza 2015]

Unbounded Nb. of Threads:

- Undecidable [Bouajjani et al., 2013]
- Decidable with "fixed linearization points" [Bouajjani et al. 2013]

Undecidability

- Reduction from reachability in counter machines
- Given a counter machine A, we construct a library L_A and a specification S_A such that L_A is **not** linearizable w.r.t. S_A iff A reaches the target state
- L_A = transition methods T[t], increments I[c_i], decrements D[c_i] and zero-tests Z[c_i]
- L_A allows only valid sequences of transitions
- S_A allows executions which don't reach the target state, or which erroneously pass some zero-test

  – it doesn't contain $\mathtt{M}[q_f]$,
  – it ends in $\mathtt{M}[q_f]$ and it contains a prefix of the form

$$(\mathtt{M\_inc}[i]\,\mathtt{M\_dec}[i])^*(\mathtt{M\_inc}[i]^+ + \mathtt{M\_dec}[i]^+)\mathtt{M\_zero}[i]$$

  – it ends in M_f and it contains a subword of the form

$$\mathtt{M\_zero}[i](\mathtt{M\_inc}[i]\,\mathtt{M\_dec}[i])^*(\mathtt{M\_inc}[i]^+ + \mathtt{M\_dec}[i]^+)\mathtt{M\_zero}[i].$$

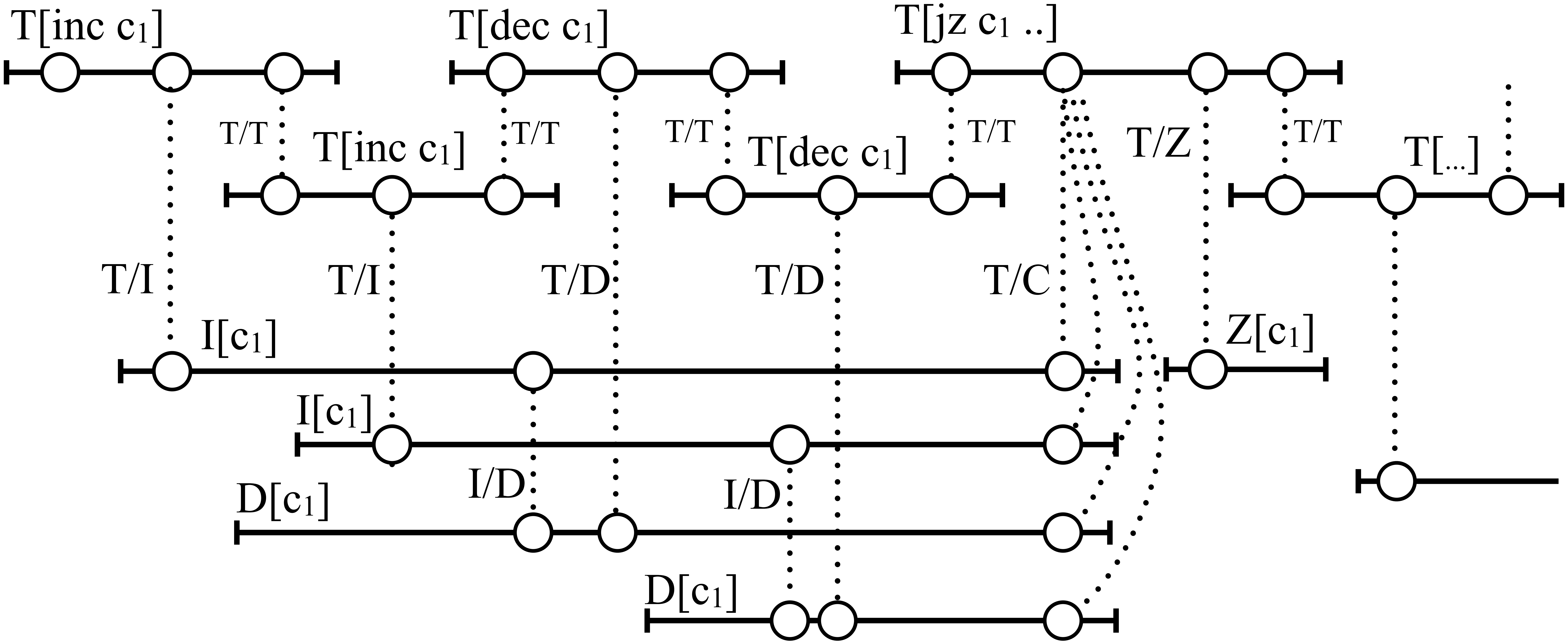
Undecidability

1. A sequence $t_1 t_2 \ldots t_i$ of $\mathcal{A}$-transitions is modeled by a pairwise-overlapping sequence of $\mathrm{T}[t_1] \cdot \mathrm{T}[t_2] \cdots \mathrm{T}[t_i]$ operations.
2. Each $\mathrm{T}[t]$-operation has a corresponding $\mathrm{I}[c_i]$, $\mathrm{D}[c_i]$, or $\mathrm{Z}[ci]$ operation, depending on whether t is, resp., an increment, decrement, or zero-test transition with counter c_i.
3. Each $\mathrm{I}[c_i]$ operation has a corresponding $\mathrm{D}[c_i]$ operation.
4. For each counter c_i, all $\mathrm{I}[c_i]$ and $\mathrm{D}[c_i]$ between $\mathrm{Z}[c_i]$ operations overlap.
5. For each counter c_i, no $\mathrm{I}[c_i]$ nor $\mathrm{D}[c_i]$ operations overlap with a $\mathrm{Z}[c_i]$ operation.
6. The number of $\mathrm{I}[c_i]$ operations between two $\mathrm{Z}[c_i]$ operations matches the number of $\mathrm{D}[c_i]$ operations.

Undecidability

- a T/T signal between T[*] operations
- for each counter c, a T/I, T/D, T/Z between T[*] operations and, resp., I[c_i], D[c_i] and Z[c_i] operations
- an I/D signal between I[c_i] and D[c_i] operations
- a T/C signal between T[t] operations and I[c_i], D[c_i] operations, for zero-testing transitions t

Undecidability



Undecidability

```
var q ∈ Q: T
var req[U]: T
var ack[U]: T
var dec[i ∈ ℕ : i < d]: T
var zero[i ∈ ℕ : i < d]: 𝔹

// for each transition ⟨q, n, q'⟩
method M[q, n, q']()
    atomic
        wait(q);
        signal(req[n]);
    atomic
        wait(ack[n]);
        signal(q');
    return ()

// for each transition ⟨q, i, q'⟩
method M[q, i, q'] ()
    atomic
        wait(q);
        zero[i] := true;
    atomic
        if !zero[i] then
            signal(q');
    return ()

// for each final state q_f
method M[q_f] ()
    wait(q_f);
    return

method M_inc[i] ()
    atomic
        if !zero[i] then
            wait(req[u_i]);
            signal(ack[u_i]);

            signal(dec[i])
    assume zero[i];
    return ()

method M_dec[i] ()
    atomic
        if !zero[i] then
            wait(dec[i]);
    atomic
        wait(req[-u_i]);
        signal(ack[-u_i]);
    assume zero[i];
    return ()

method M_zero[i] ()
    atomic
        if zero[i] then
            zero[i] := false;

    return ()
```

Checking Linearizability: Complexity (finite-state implementations)

Bounded Nb. of Threads:

- EXSPACE-complete [Alur et al., 1996, Hamza 2015]

Unbounded Nb. of Threads:

- Undecidable [Bouajjani et al., 2013]
- Decidable with “fixed linearization points” [Bouajjani et al. 2013]

Libraries

A *method* is a finite automaton $M = \langle Q, \Sigma, I, F, \hookrightarrow \rangle$ with labeled transitions $\langle m_1, v_1 \rangle \xhookrightarrow{a} \langle m_2, v_2 \rangle$ between method-local states $m_1, m_2 \in Q$ paired with finite-domain shared-state valuations $v_1, v_2 \in V$. The initial and final states $I, F \subseteq Q$ represent the method-local states passed to, and returned from, M.

A *client* of a library L is a finite automaton $C = \langle Q, \Sigma, \ell_0, \hookrightarrow \rangle$ with initial state $\ell_0 \in Q$ and transitions $\hookrightarrow \subseteq Q \times \Sigma \times Q$ labeled by the alphabet $\Sigma = \{M(m_0, m_f) : M \in L, m_0, m_f \in Q_M\}$ of library method calls

most general client $C^\star = \langle Q, \Sigma, \ell_0, \hookrightarrow \rangle$ of a library L nondeterministically calls L's methods in any order: $Q = \{\ell_0\}$ and $\hookrightarrow = Q \times \Sigma \times Q$.

Example: Coarse-Grain Sets

```
remove(k):
Node p, c
Lock l
lock l;  LP
p = H;
c = p.next
while (c.key < k)
  p = c;
  c = c.next;
if (c.key = k)
  p.next = c.next
  return true
else
  return false
unlock l
```

```
ctn(k):
Node p, c
Lock l
lock l;  LP
p = H;
c = p.next
while (c.key < k)
  p = c;
  c = c.next;
if (c.key = k)
  return true
else
  return false
unlock l
```

Example: Fine-Grain Sets

```
add(k):
Node p, c
lock(H)
p = H
c = p.next;
lock(c);
while (c.key < k)
  unlock(p);
  p = c;
  c = c.next;
  lock(c);
if (c.key = k)
  return false
else
  n = new Node(k,-)
  n.next = c
  p.next = n
  return true
  unlock(c)
  unlock(p)
```

LP

"time point at which I find a key $\geq$ input key"

"time point at which I lock the cell with key $\geq$ input key"

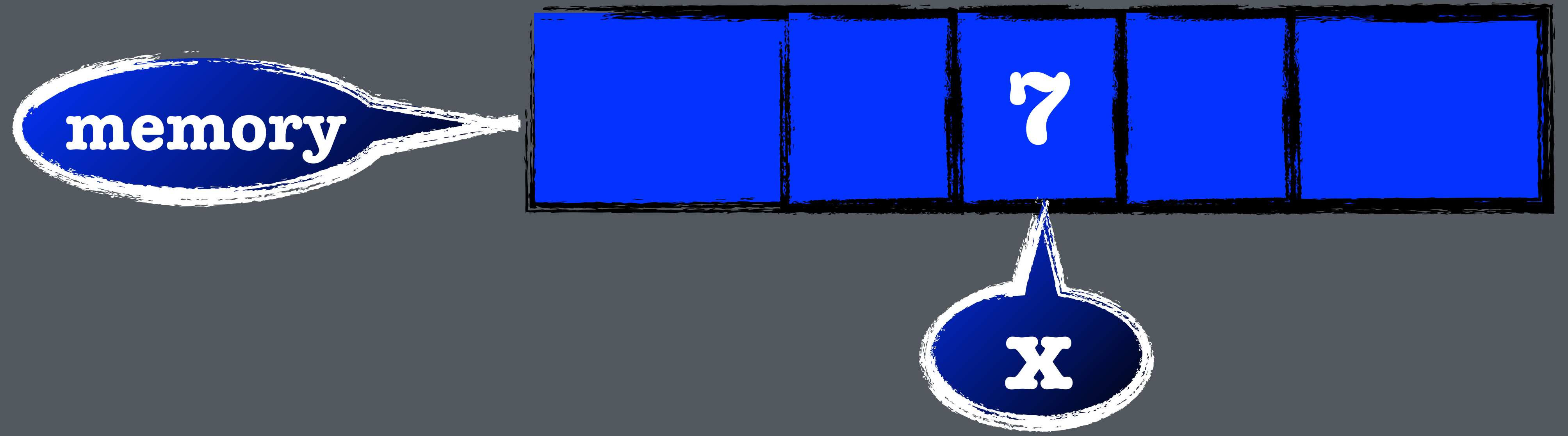
The Treiber Stack Algorithm

CAS: Compare And Swap

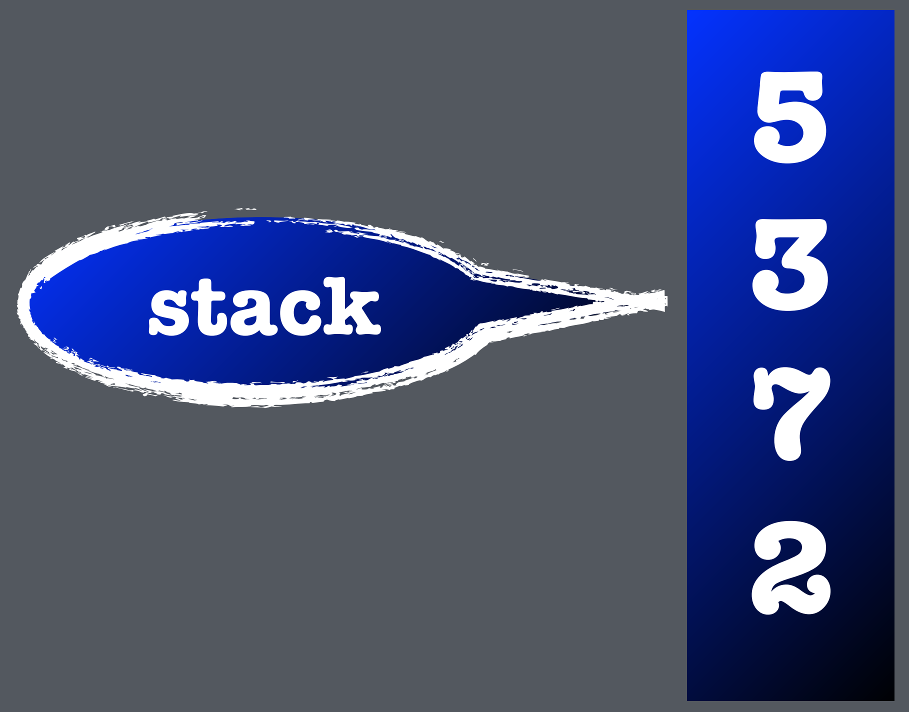
```
CAS (x,expected,new):
Integer tmp
Atomically
  tmp = x
  if (tmp == expected)
    x = new
    return true
  else
    return false
```

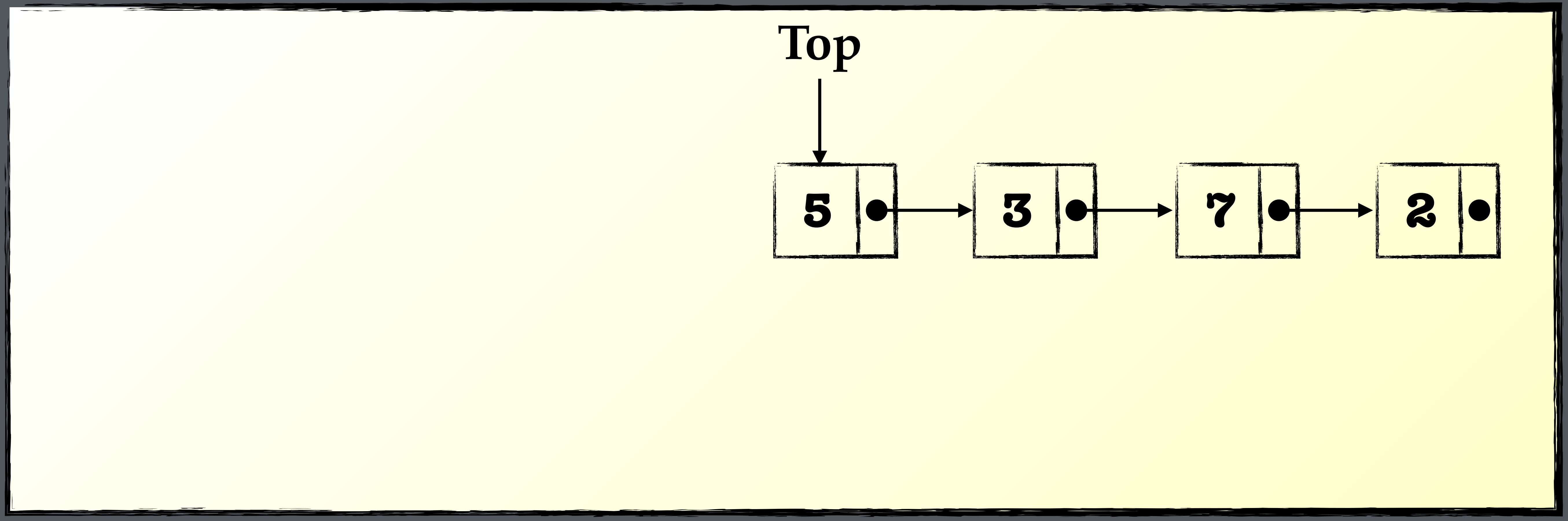
- hardware
  - Intel, AMD — CMPXCHG
  - SPARC — CAS
- software
  - java — CompareAndSet
  - C# — CompareExchange

CAS(x,5,8) **false**



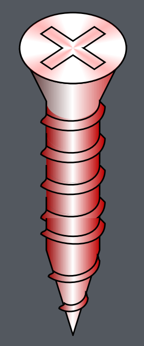
The Treiber Stack Algorithm



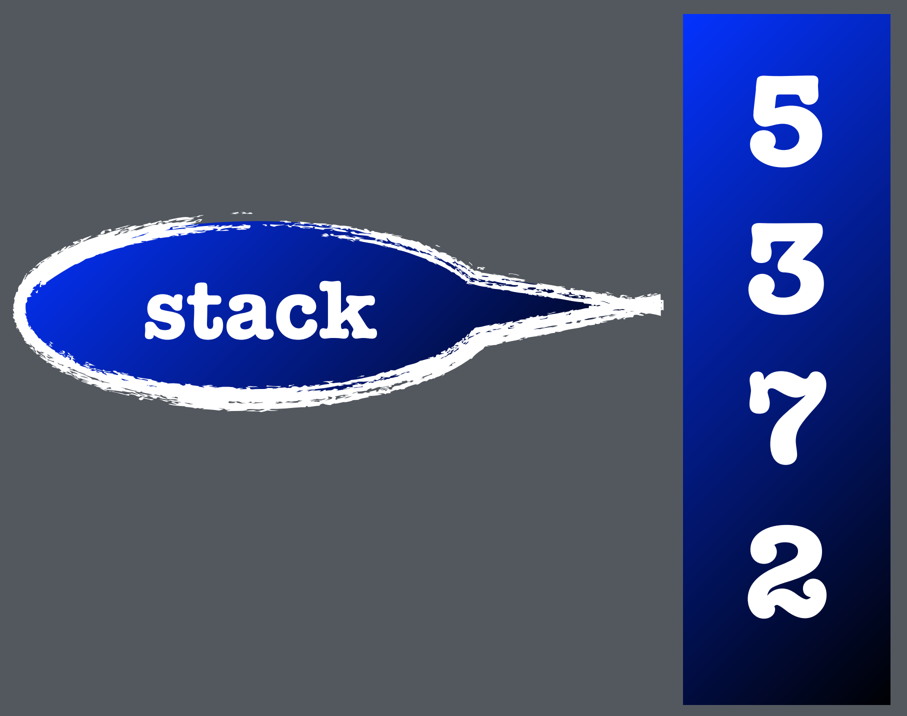


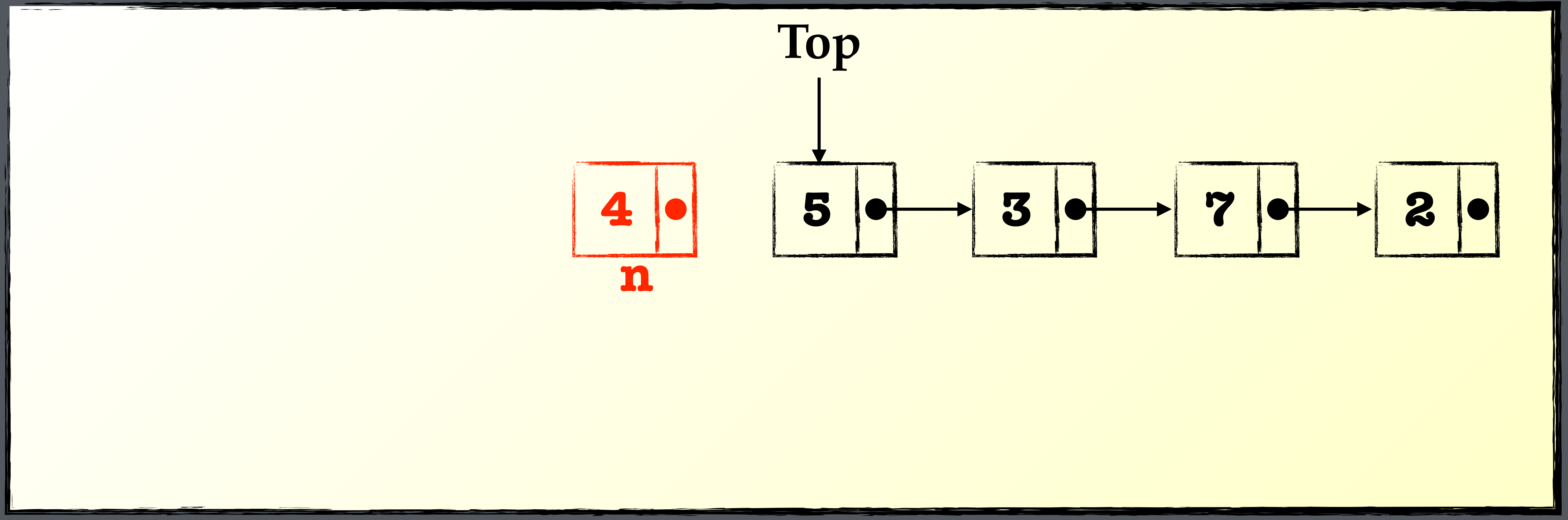
The Treiber Stack Algorithm

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```



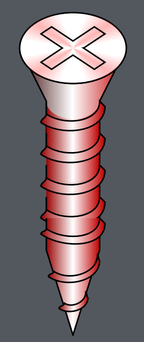
push(4)



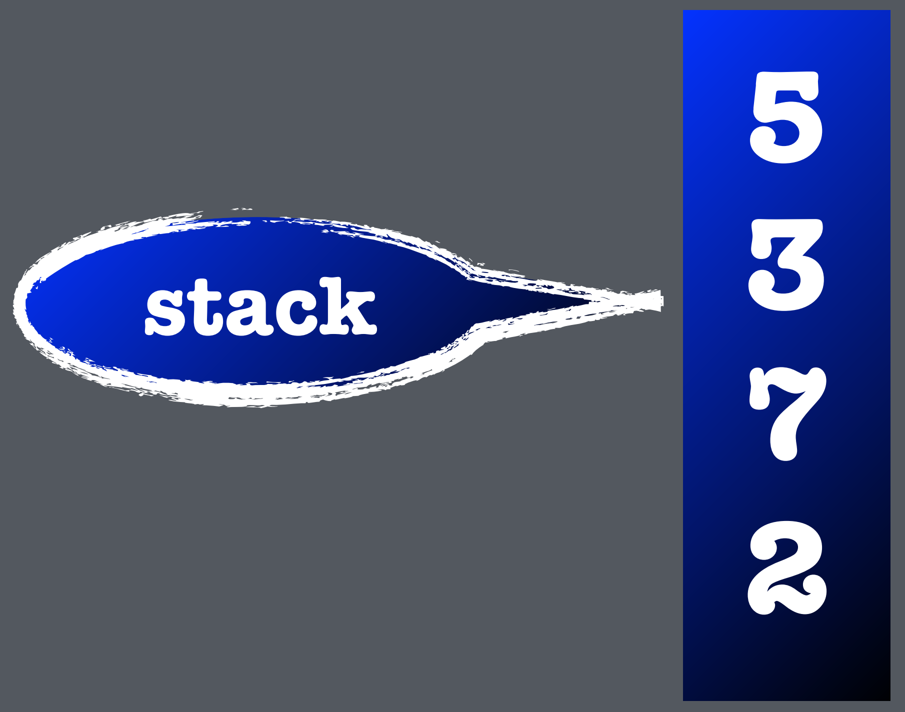


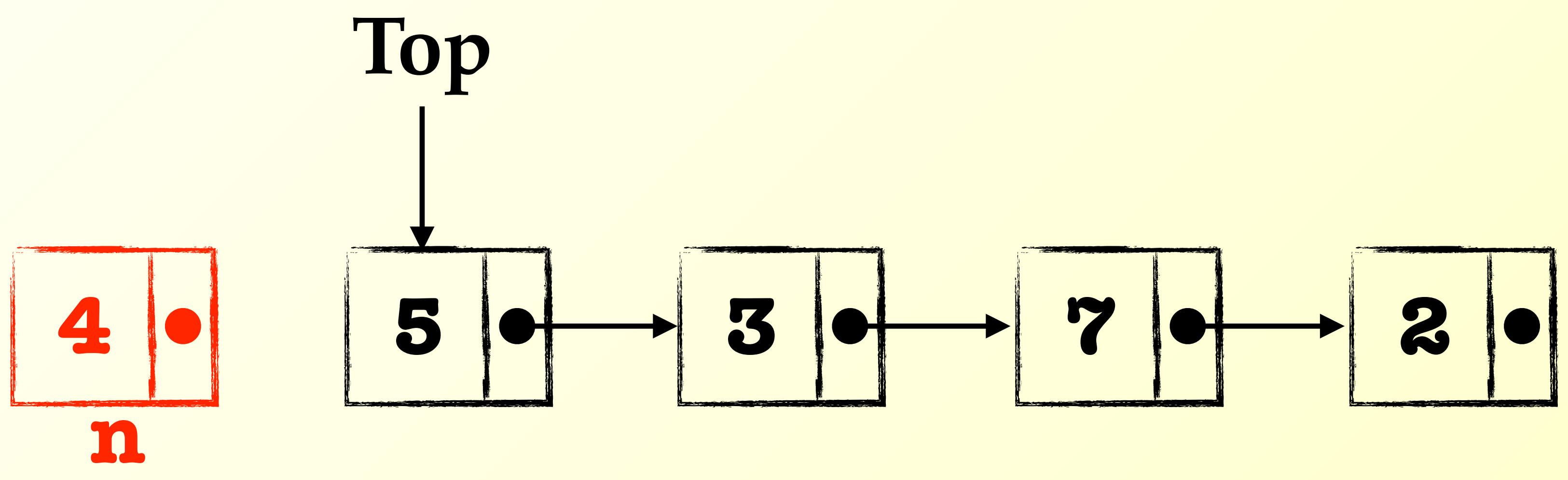
The Treiber Stack Algorithm

```
push(k):
Node t
  n = new Node(k,-)
  while (true)
    t = Top
    n.next = t
    if (CAS (Top,t,n))
      exit
```



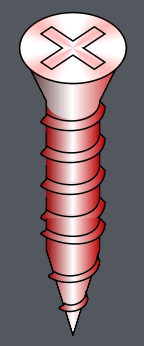
push(4)



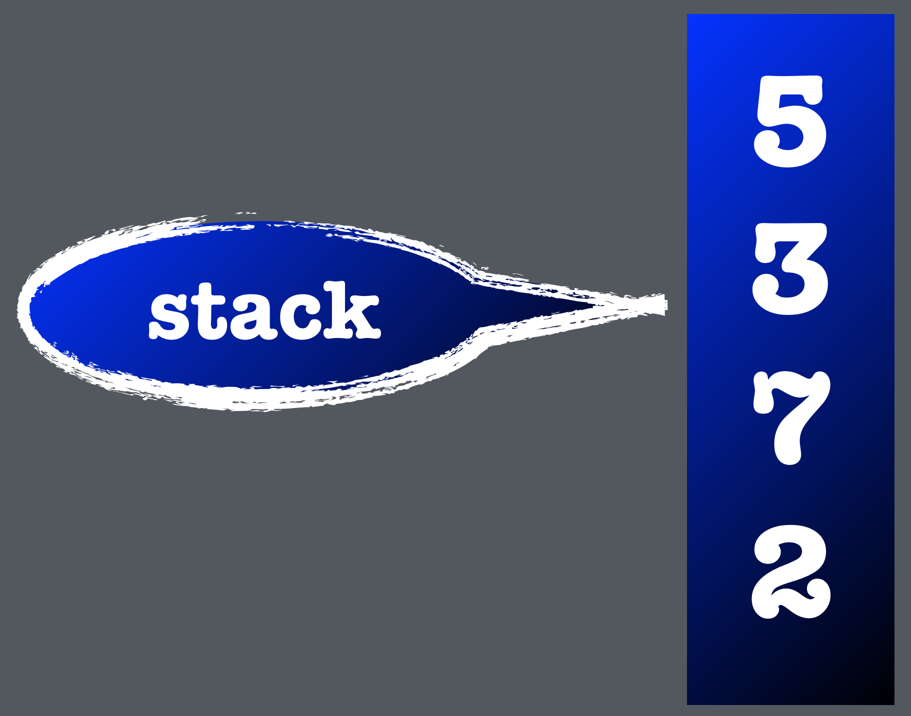


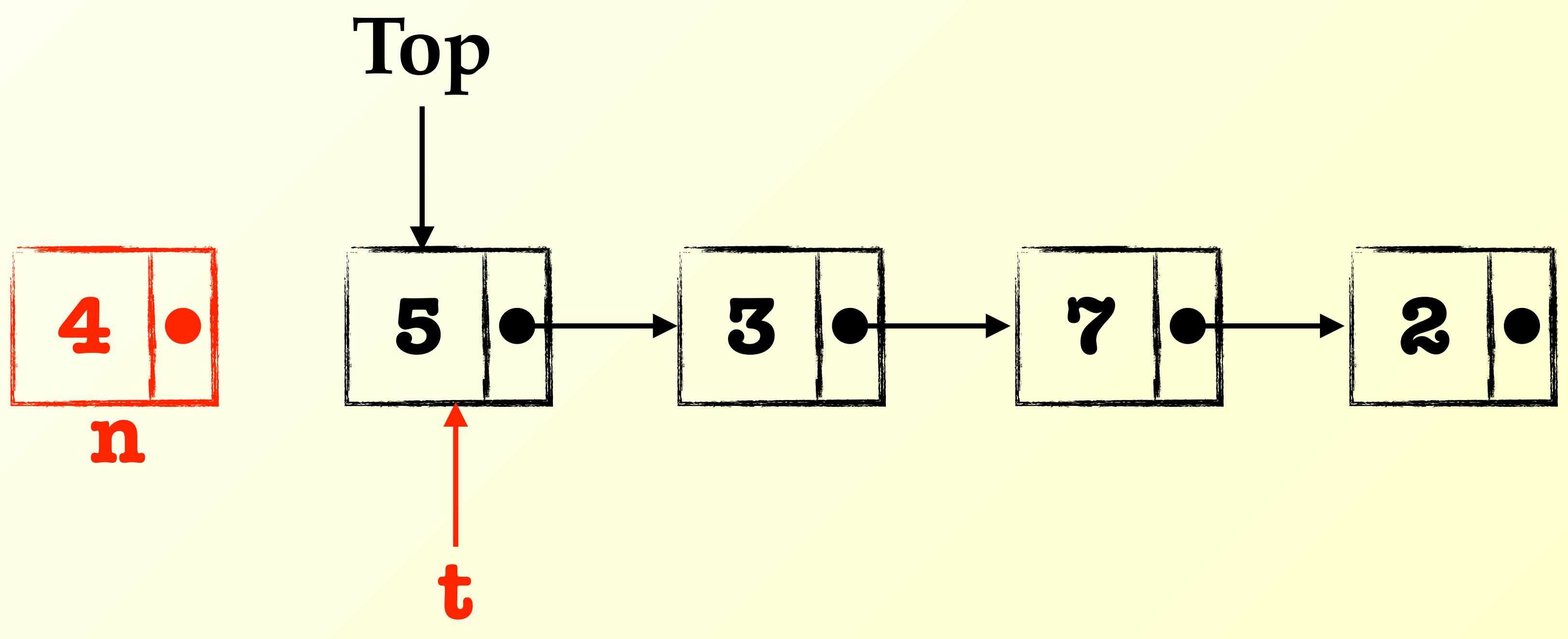
The Treiber Stack Algorithm

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```



push(4)

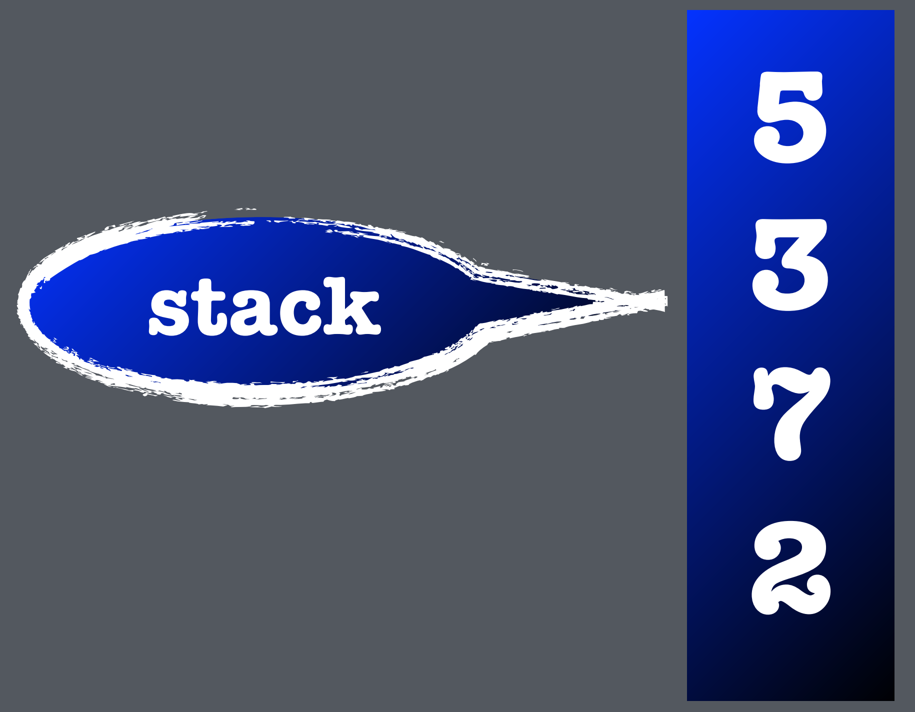


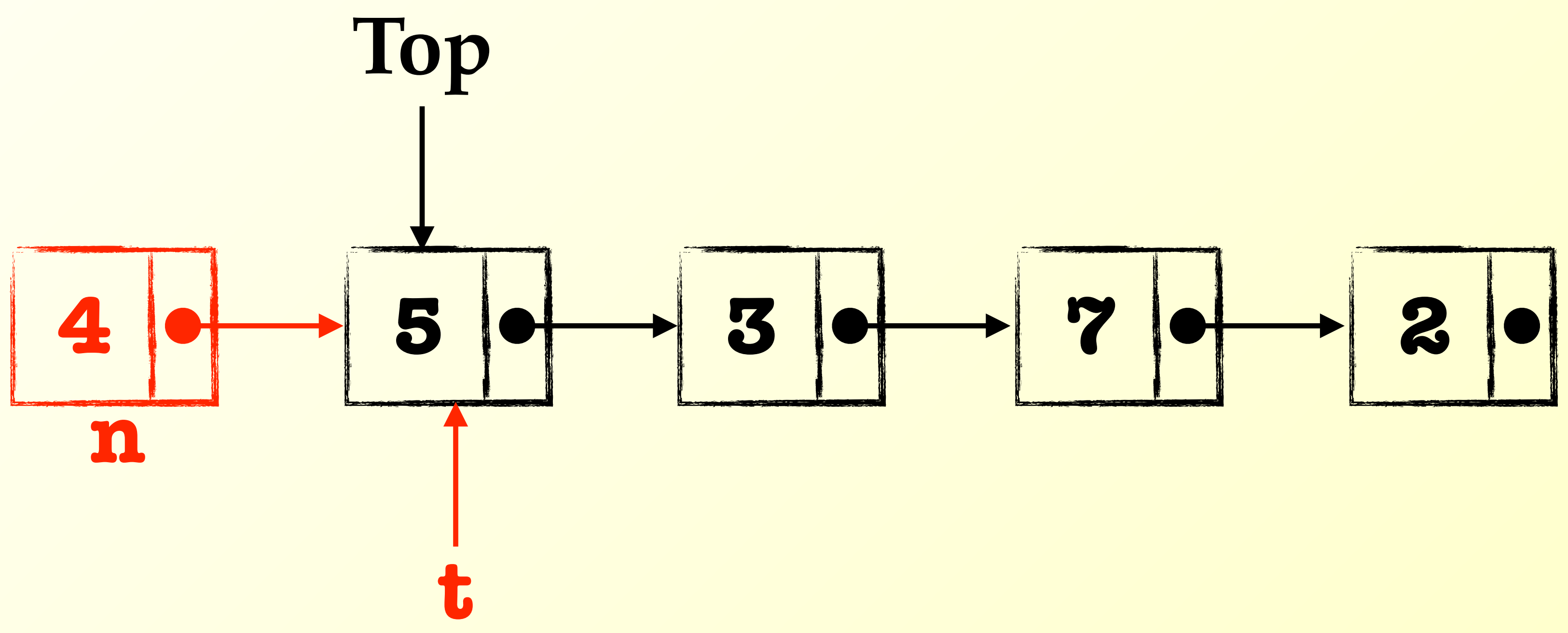


The Treiber Stack Algorithm

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

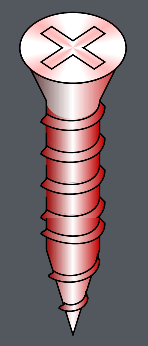
push(4)



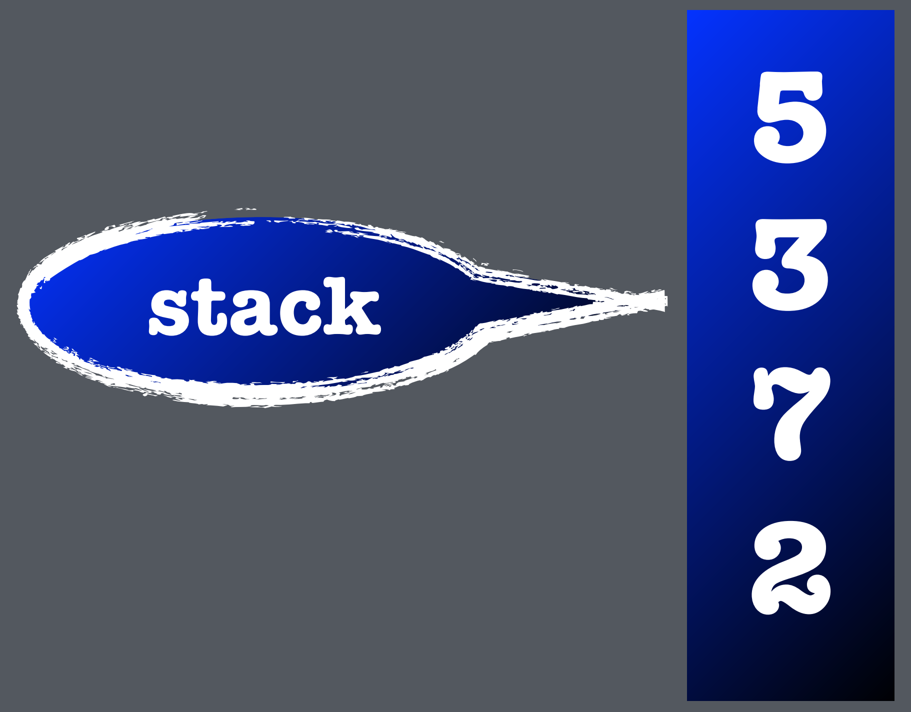


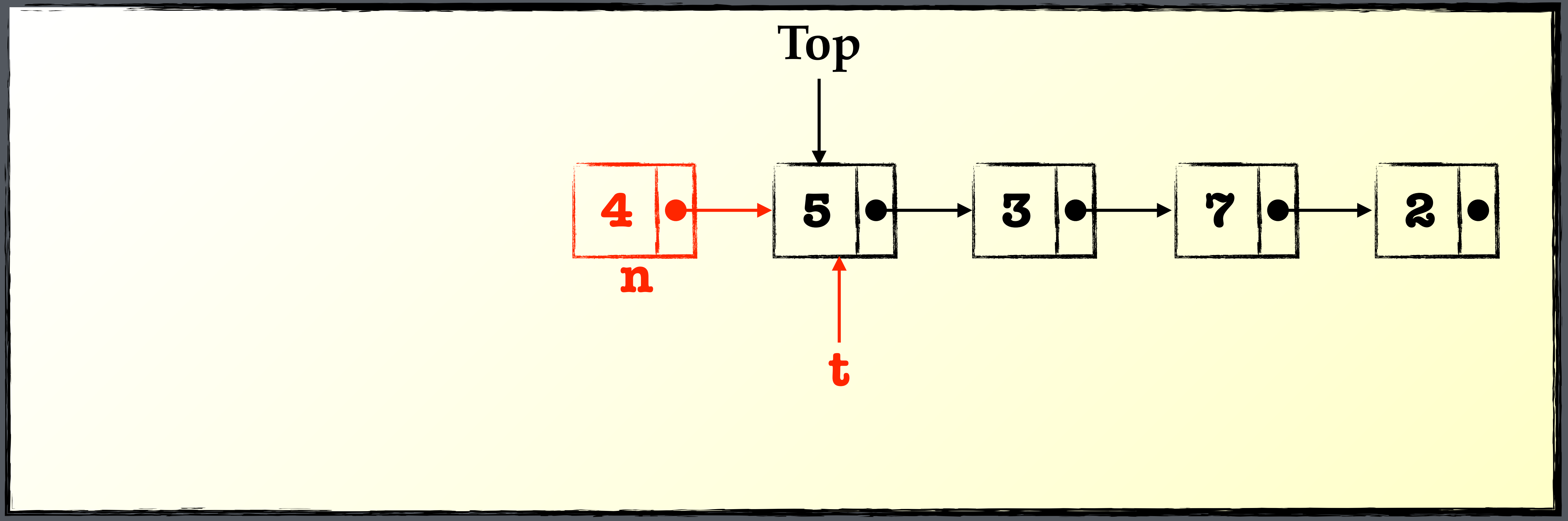
The Treiber Stack Algorithm

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```



push(4)

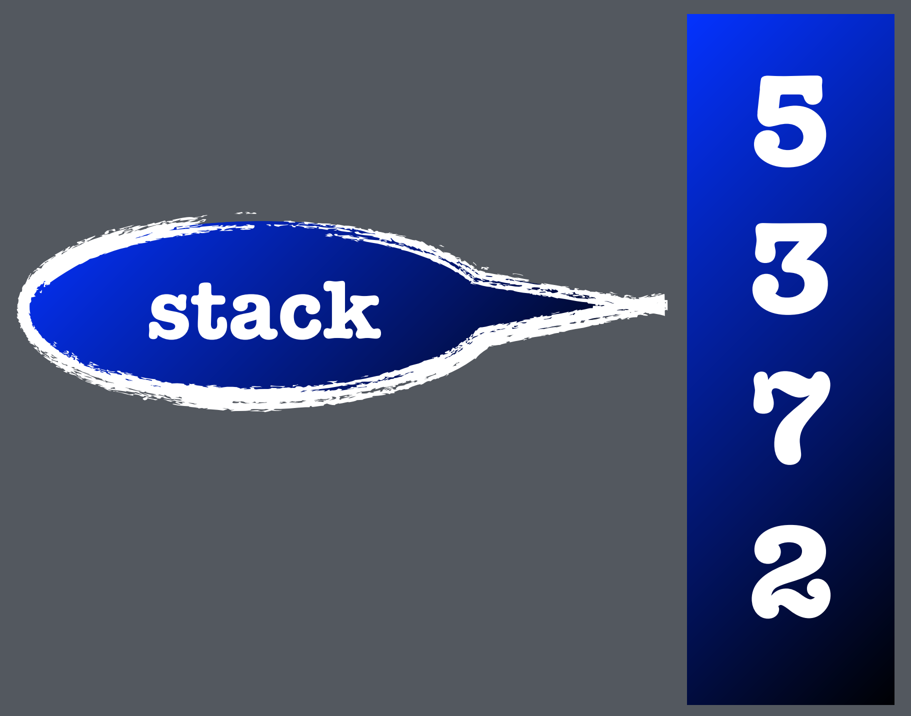


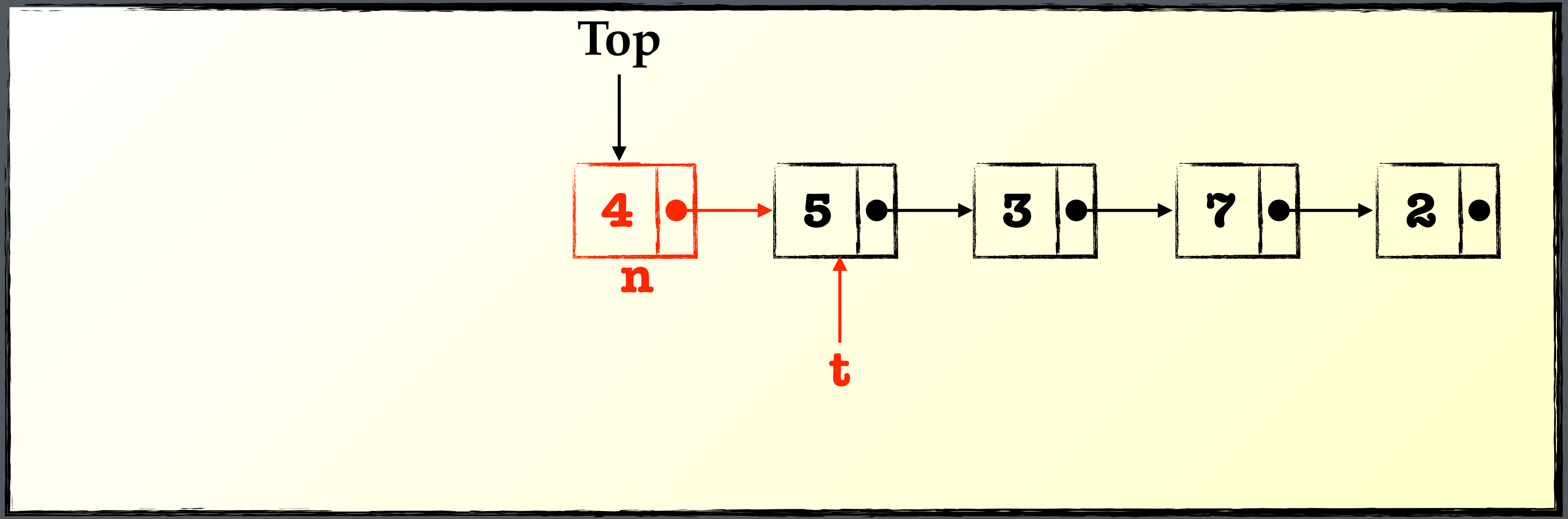


The Treiber Stack Algorithm

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

push(4)

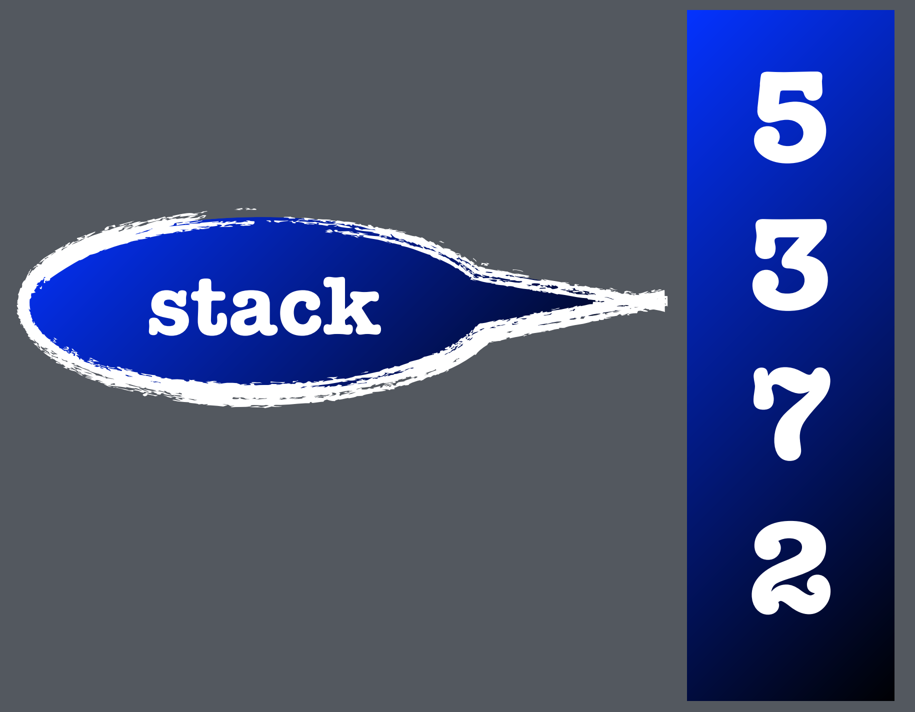


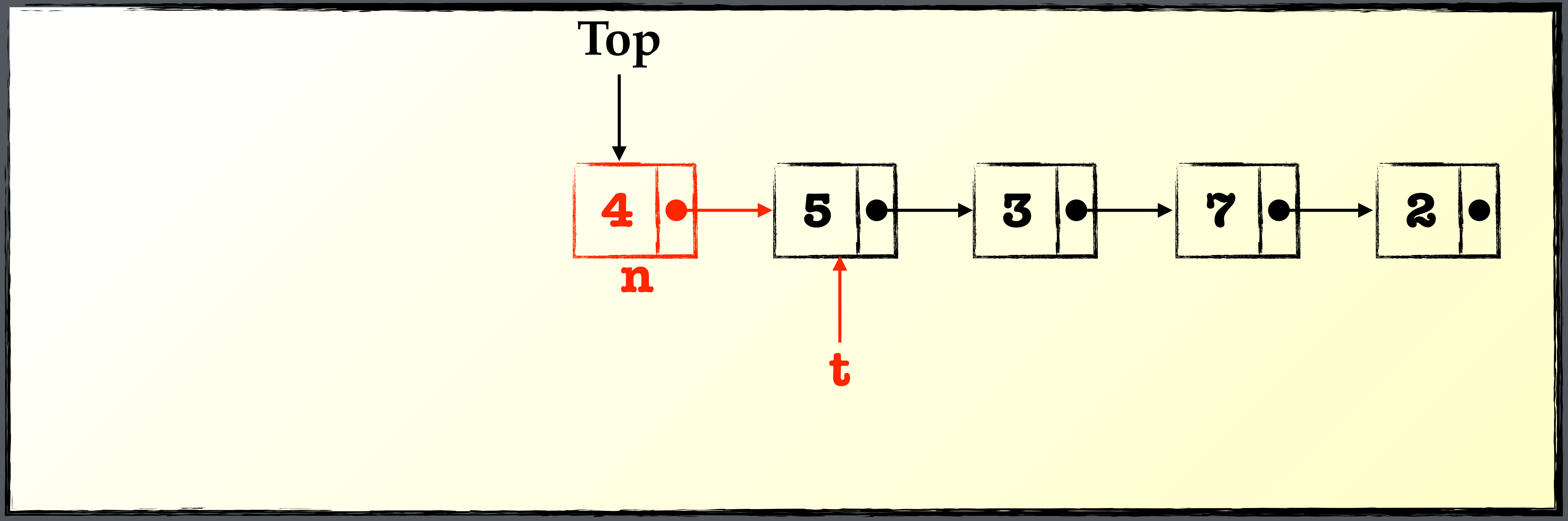


The Treiber Stack Algorithm

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

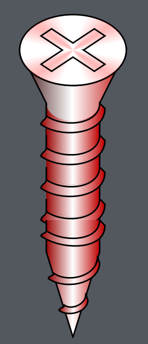
push(4)



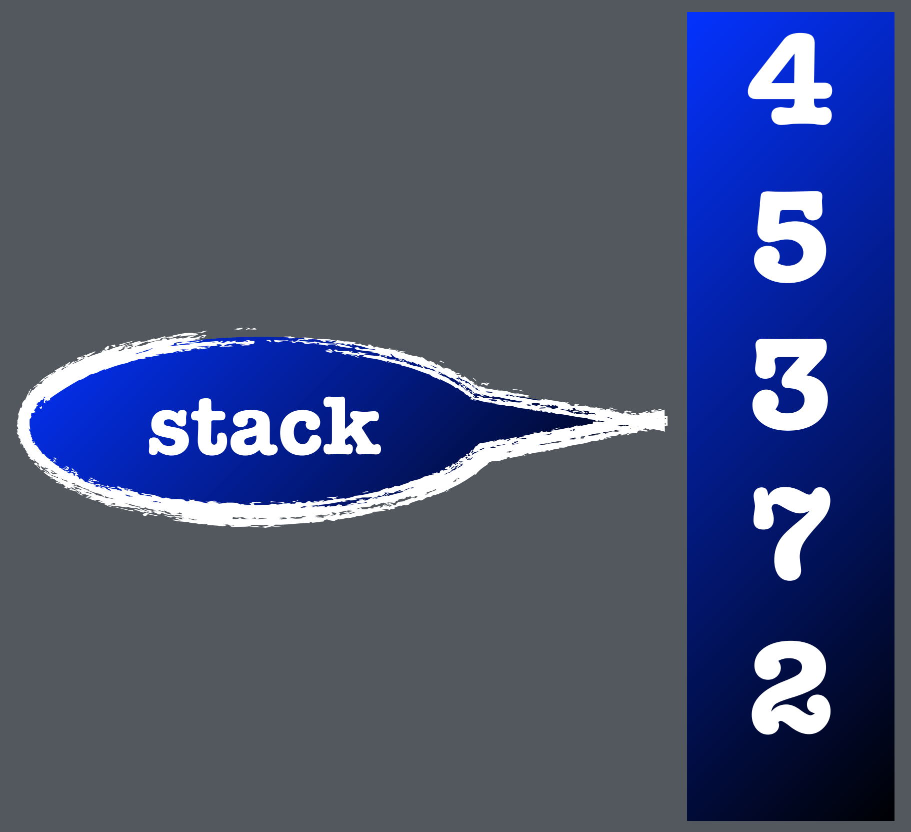


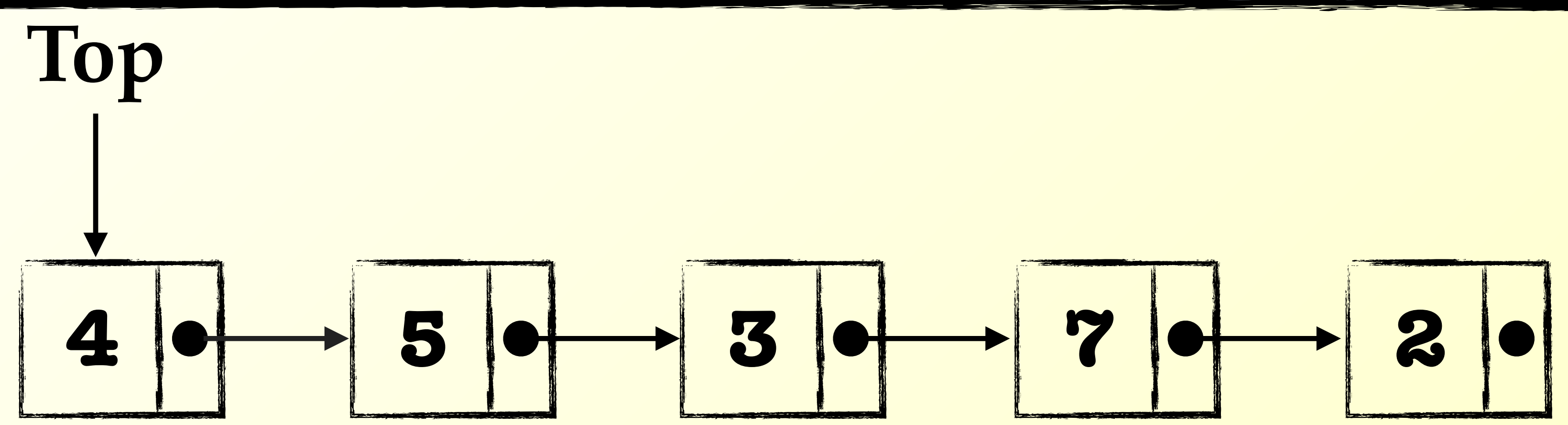
The Treiber Stack Algorithm

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```



push(4)





The Treiber Stack Algorithm

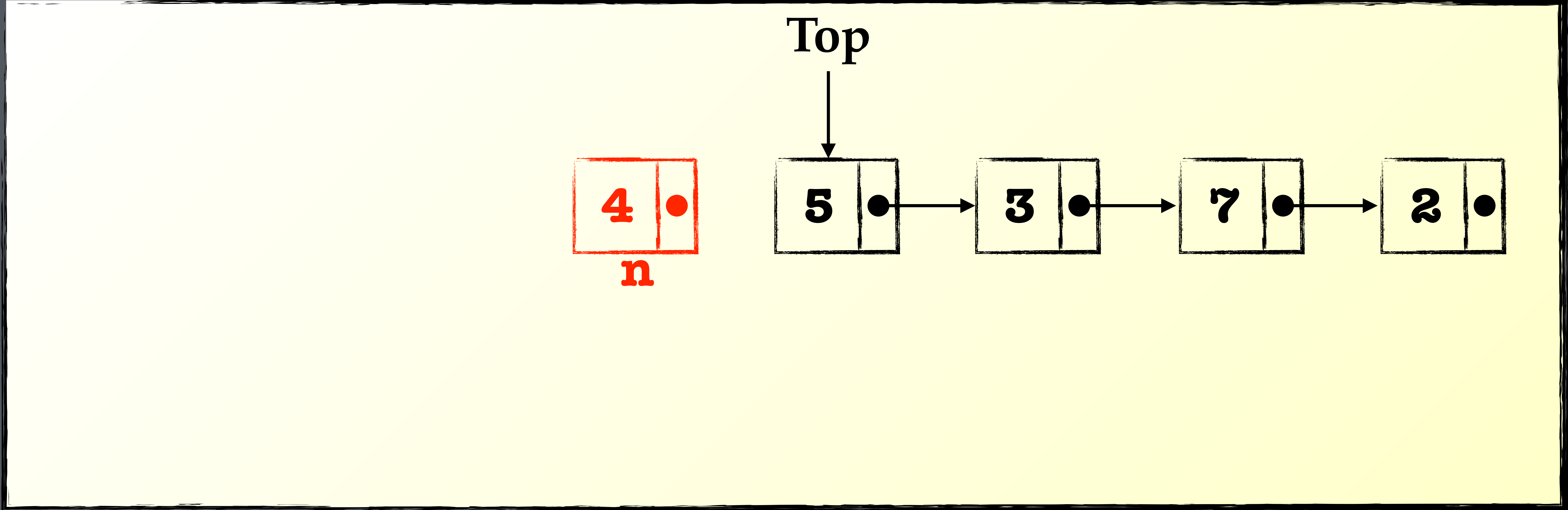
Concurrent push operations

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

push(4)

push(2)

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```



The Treiber Stack Algorithm

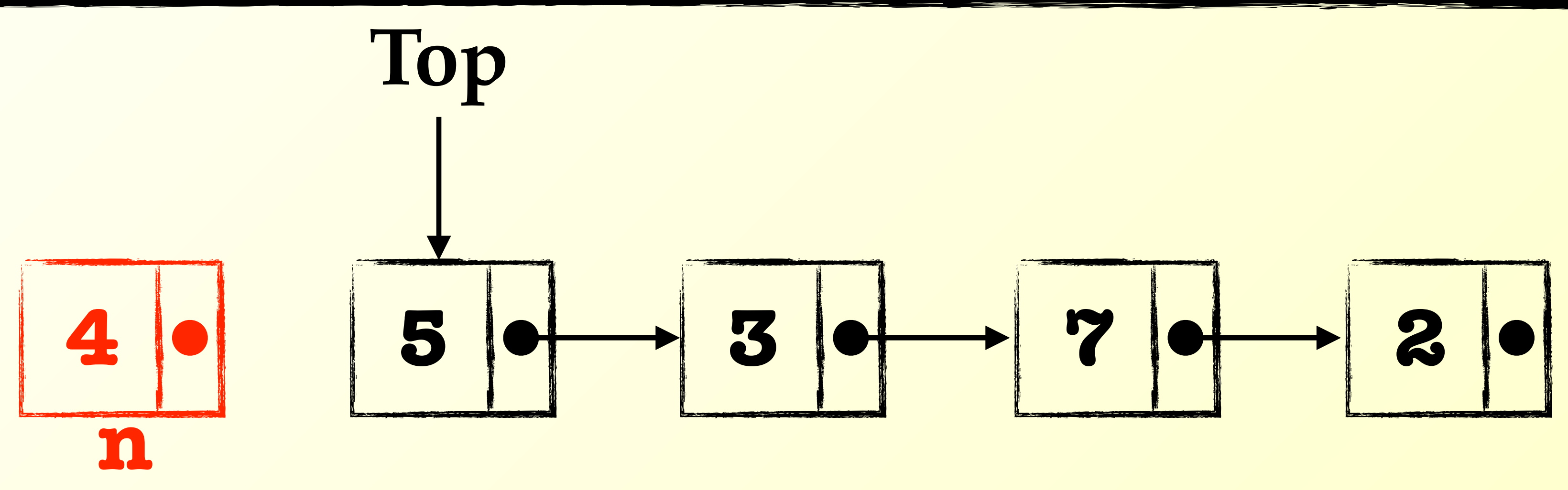
Concurrent push operations

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

push(4)

push(2)

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```



The Treiber Stack Algorithm

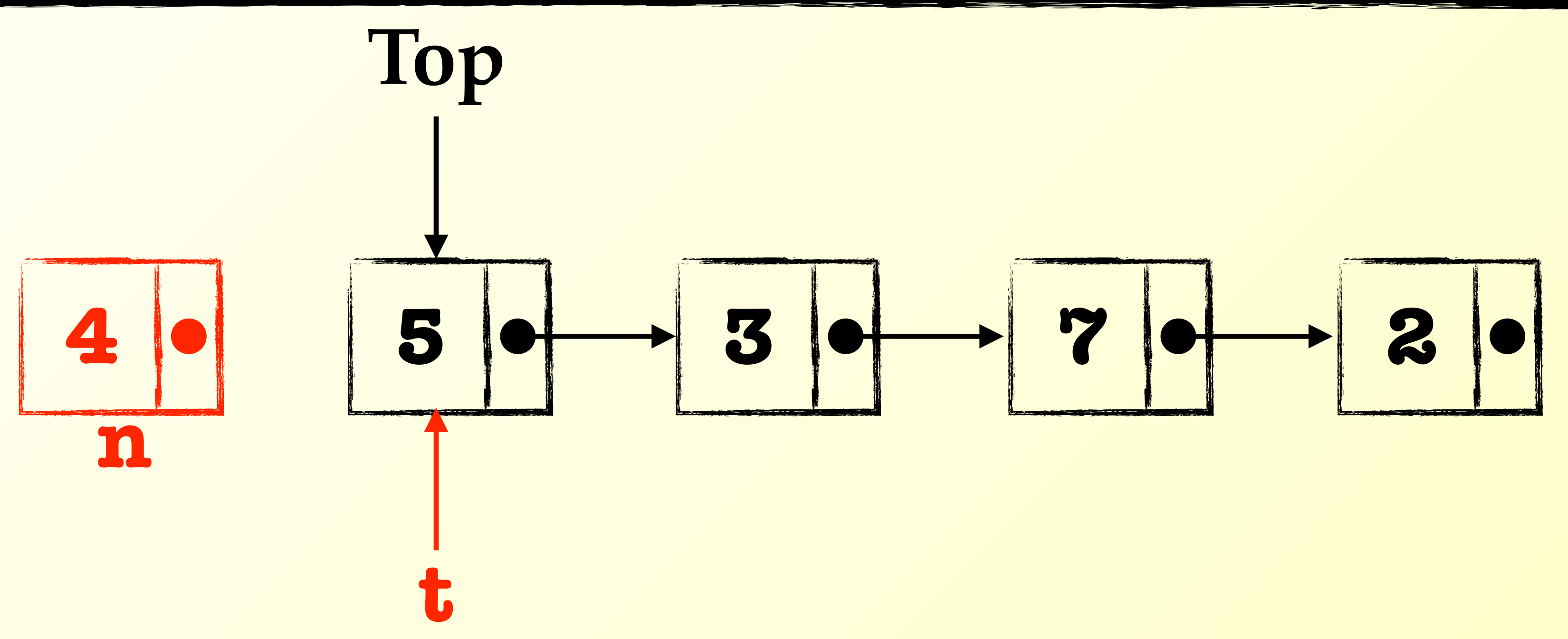
Concurrent push operations

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

push(4)

push(2)

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```



The Treiber Stack Algorithm

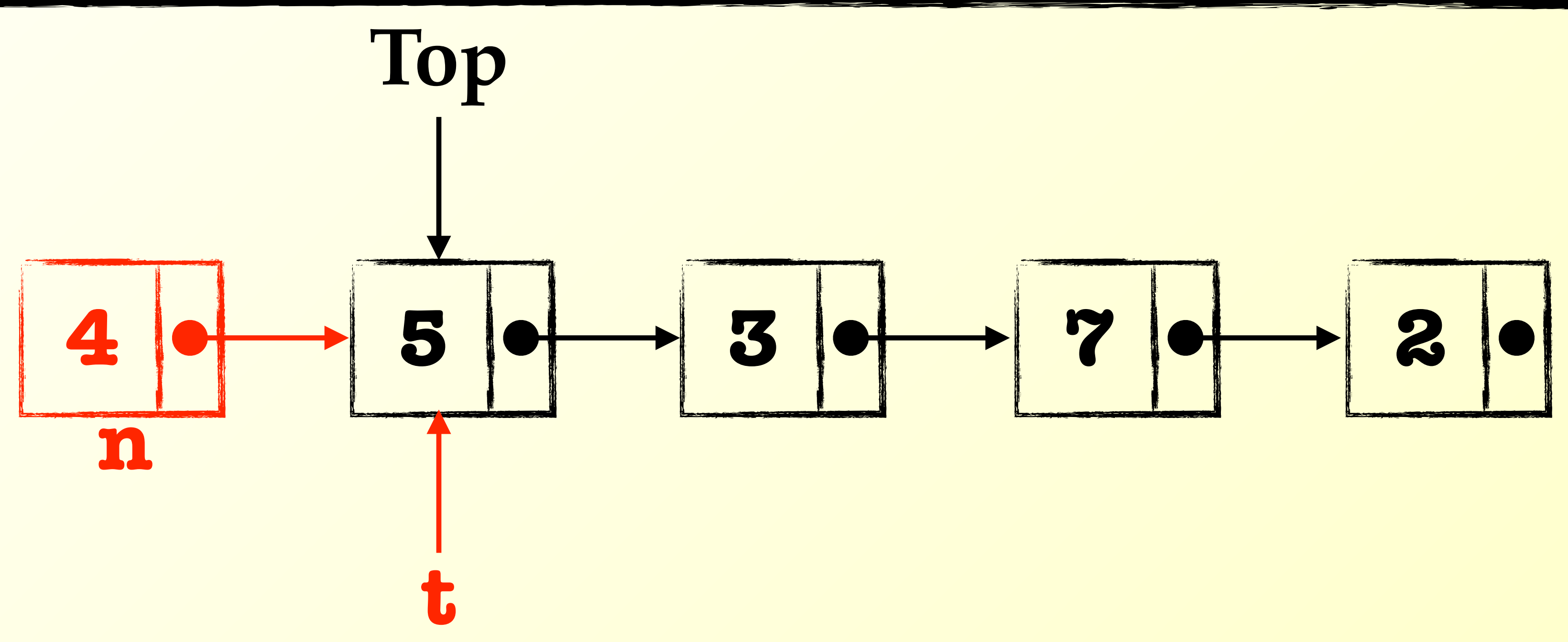
Concurrent push operations

```
push(k):
Node t
  n = new Node(k,-)
  while (true)
    t = Top
    n.next = t
    if (CAS (Top,t,n))
      exit
```

push(4)

push(2)

```
push(k):
Node t
  n = new Node(k,-)
  while (true)
    t = Top
    n.next = t
    if (CAS (Top,t,n))
      exit
```



The Treiber Stack Algorithm

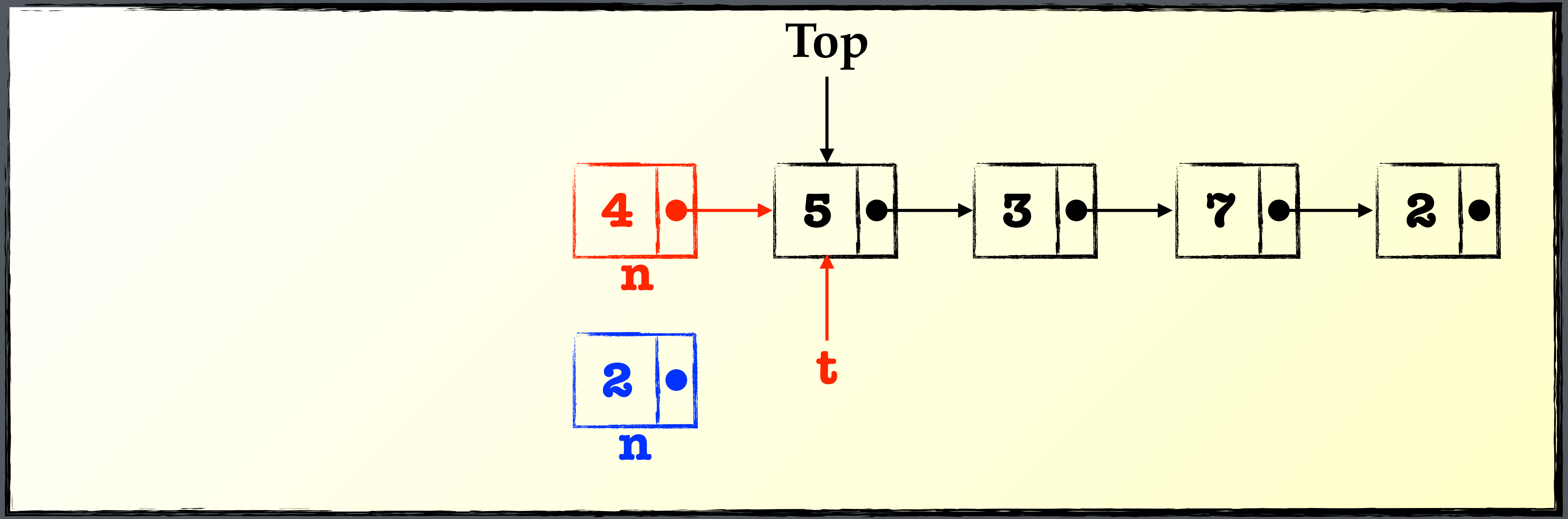
Concurrent push operations

```
push(k):
Node t
1  n = new Node(k,-)
2  while (true)
3    t = Top
4    n.next = t
5    if (CAS (Top,t,n))
6      exit
```

push(4)

push(2)

```
push(k):
Node t
1  n = new Node(k,-)
2  while (true)
3    t = Top
4    n.next = t
5    if (CAS (Top,t,n))
6      exit
```



The Treiber Stack Algorithm

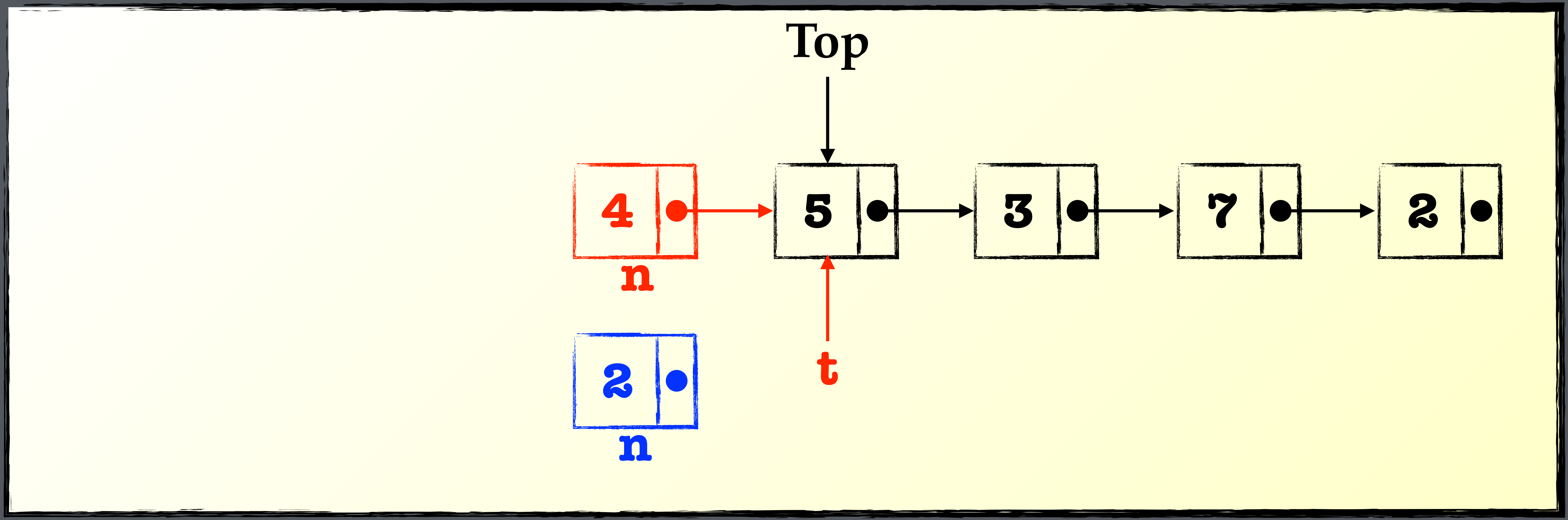
Concurrent push operations

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

push(4)

push(2)

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```



The Treiber Stack Algorithm

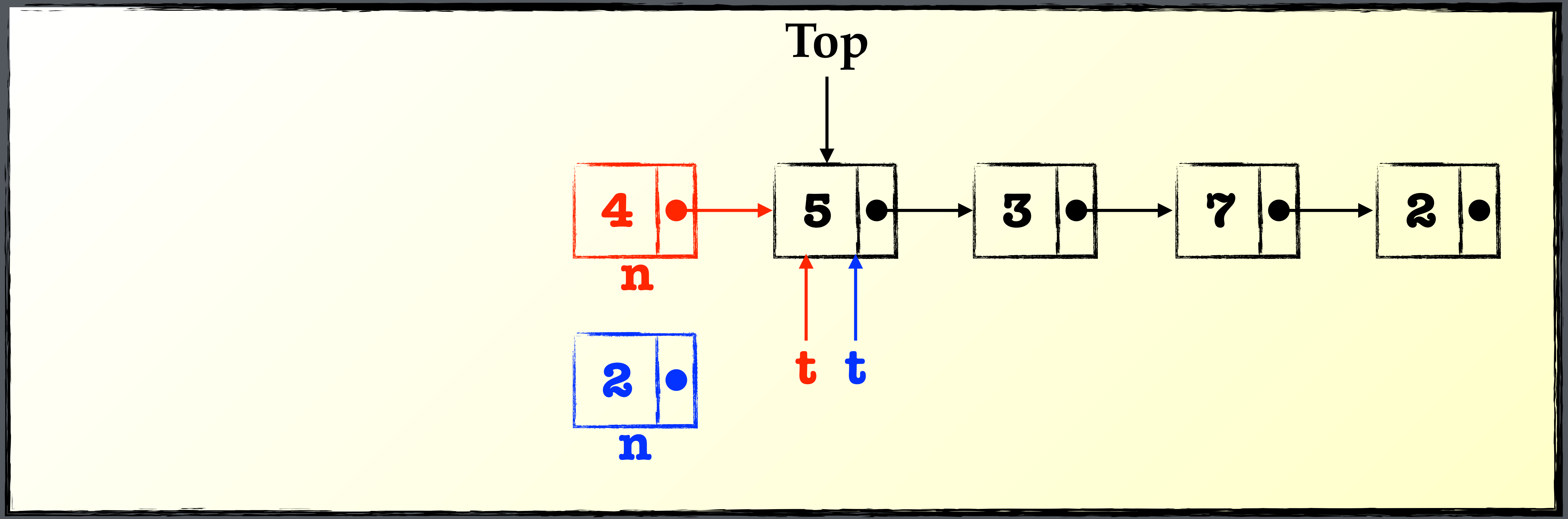
Concurrent push operations

```
push(k):
Node t
 n = new Node(k,-)
 while (true)
   t = Top
   n.next = t
   if (CAS (Top,t,n))
     exit
```

push(4)

push(2)

```
push(k):
Node t
 n = new Node(k,-)
 while (true)
   t = Top
   n.next = t
   if (CAS (Top,t,n))
     exit
```



The Treiber Stack Algorithm

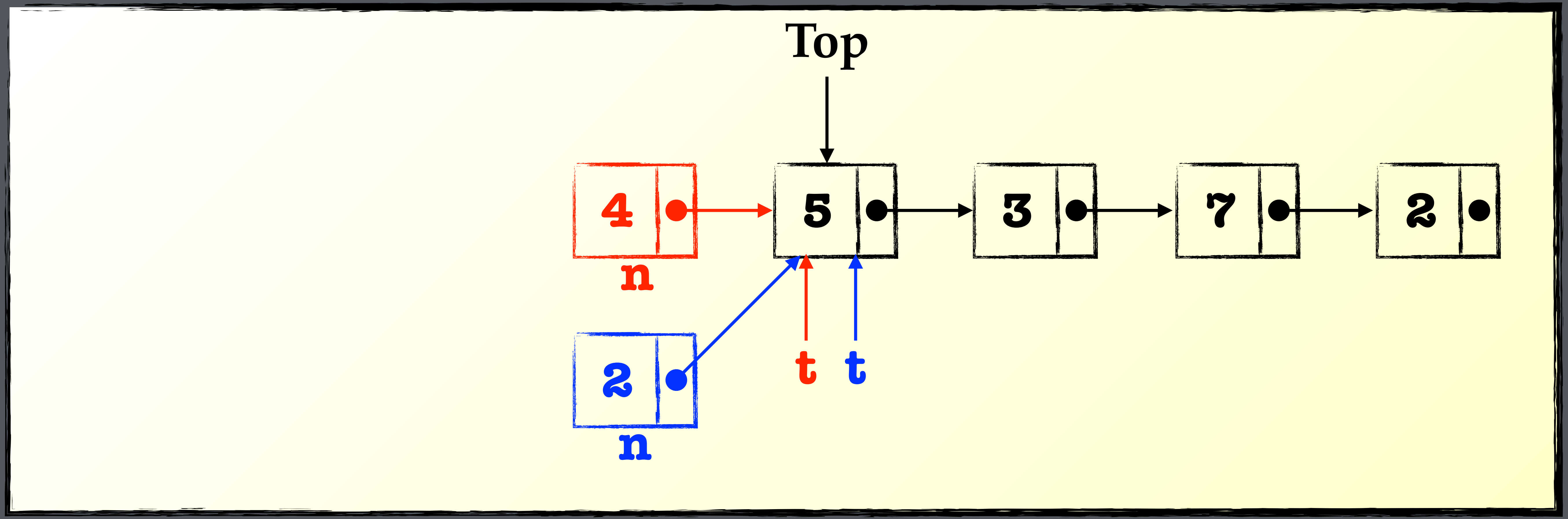
Concurrent push operations

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

push(4)

push(2)

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```



The Treiber Stack Algorithm

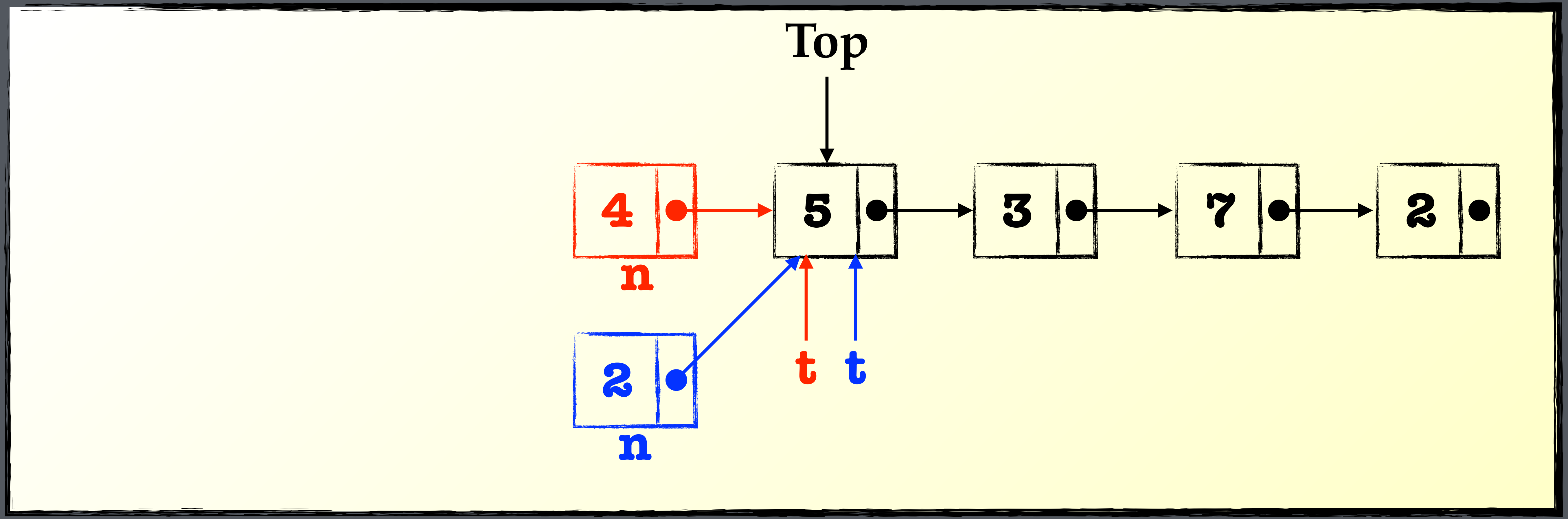
Concurrent push operations

```
push(k):
Node t
  n = new Node(k,-)
  while (true)
    t = Top
    n.next = t
    if (CAS (Top,t,n))
      exit
```

push(4)

push(2)

```
push(k):
Node t
  n = new Node(k,-)
  while (true)
    t = Top
    n.next = t
    if (CAS (Top,t,n))
      exit
```



The Treiber Stack Algorithm

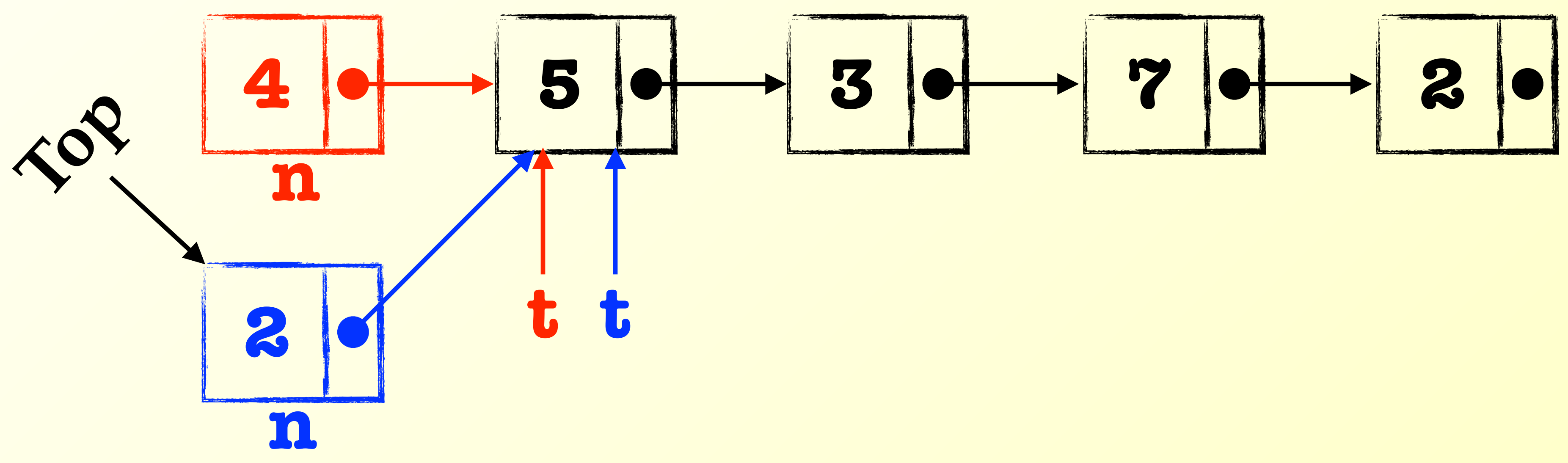
Concurrent push operations

```
push(k):
Node t
  n = new Node(k,-)
  while (true)
    t = Top
    n.next = t
    if (CAS (Top,t,n))
      exit
```

push(4)

push(2)

```
push(k):
Node t
  n = new Node(k,-)
  while (true)
    t = Top
    n.next = t
    if (CAS (Top,t,n))
      exit
```



The Treiber Stack Algorithm

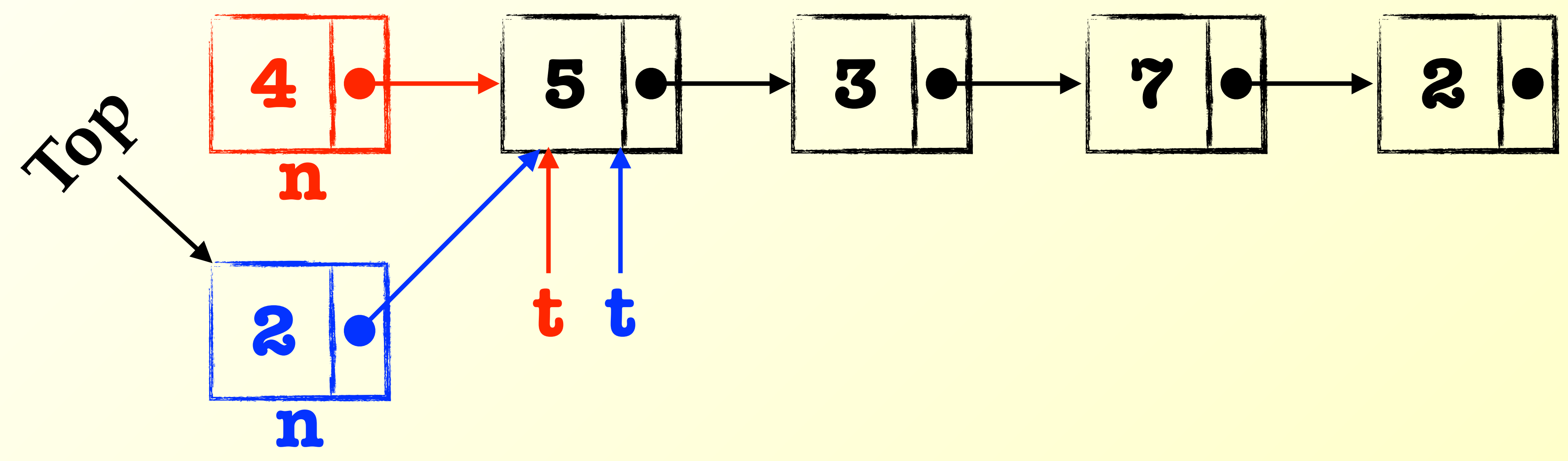
Concurrent push operations

```
push(k):
Node t
  n = new Node(k,-)
  while (true)
    t = Top
    n.next = t
    if (CAS (Top,t,n))
      exit
```

push(4)

push(2)

```
push(k):
Node t
  n = new Node(k,-)
  while (true)
    t = Top
    n.next = t
    if (CAS (Top,t,n))
      exit
```



The Treiber Stack Algorithm

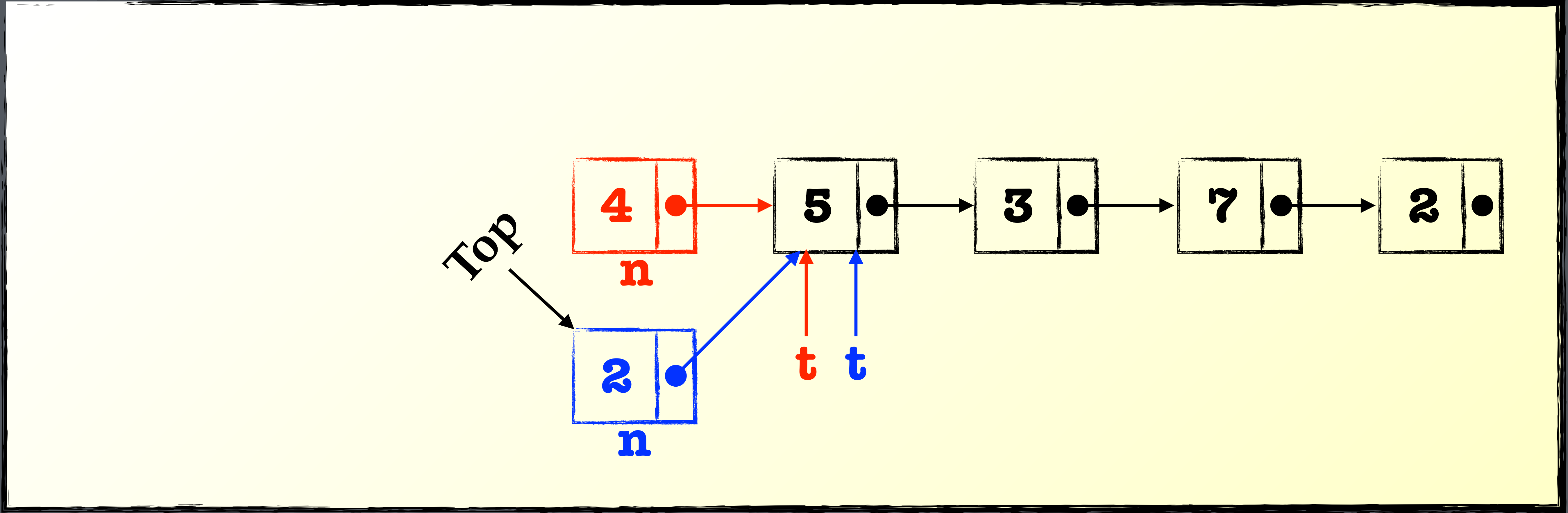
Concurrent push operations

```
push(k):
Node t
  n = new Node(k,-)
  while (true)
    t = Top
    n.next = t
    if (CAS (Top,t,n))
      exit
```

push(4)

push(2)

```
push(k):
Node t
  n = new Node(k,-)
  while (true)
    t = Top
    n.next = t
    if (CAS (Top,t,n))
      exit
```



The Treiber Stack Algorithm

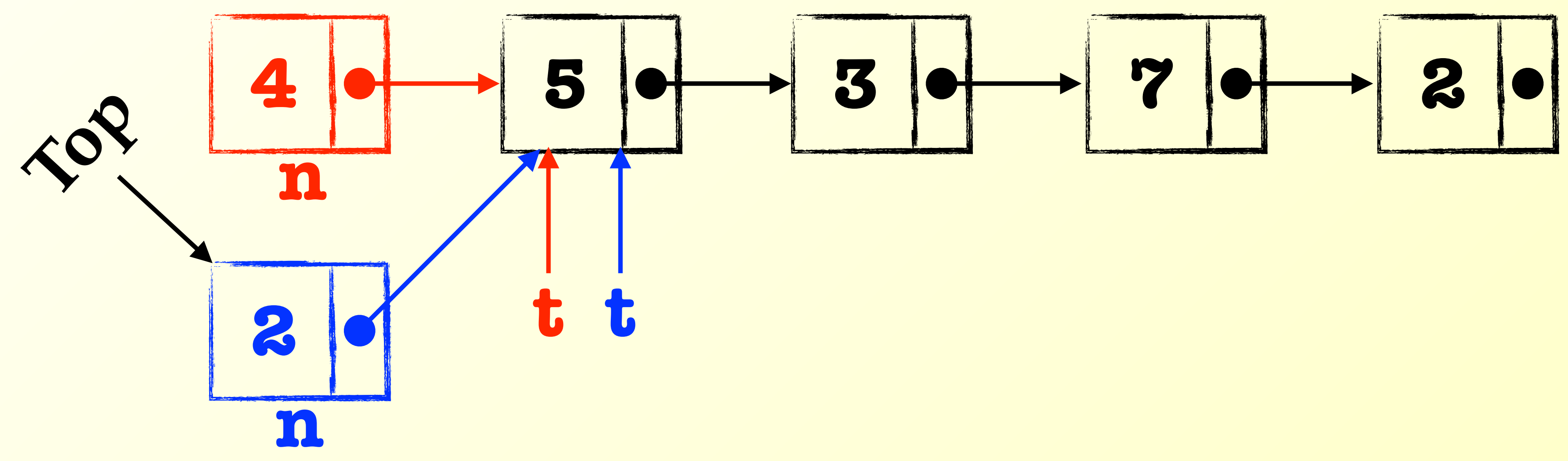
Concurrent push operations

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

push(4)

push(2)

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

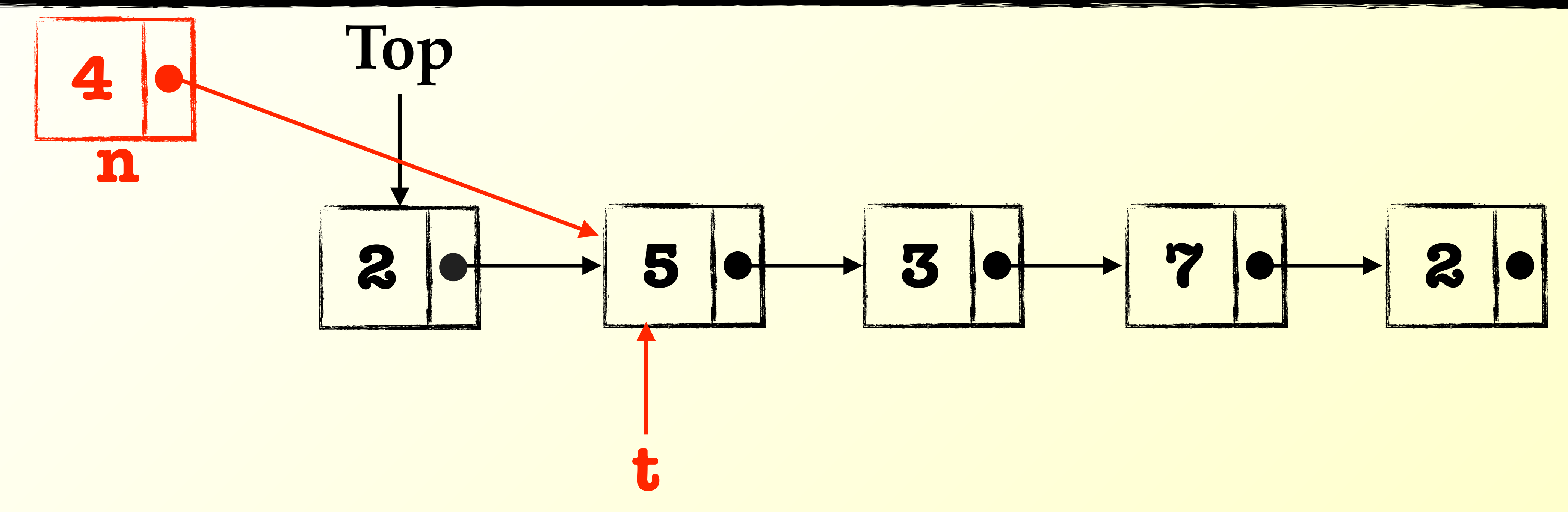


The Treiber Stack Algorithm

Concurrent push operations

```
push(k):
Node t
1  n = new Node(k,-)
2  while (true)
3    t = Top
4    n.next = t
5    if (CAS (Top,t,n))
6      exit
```

push(4)

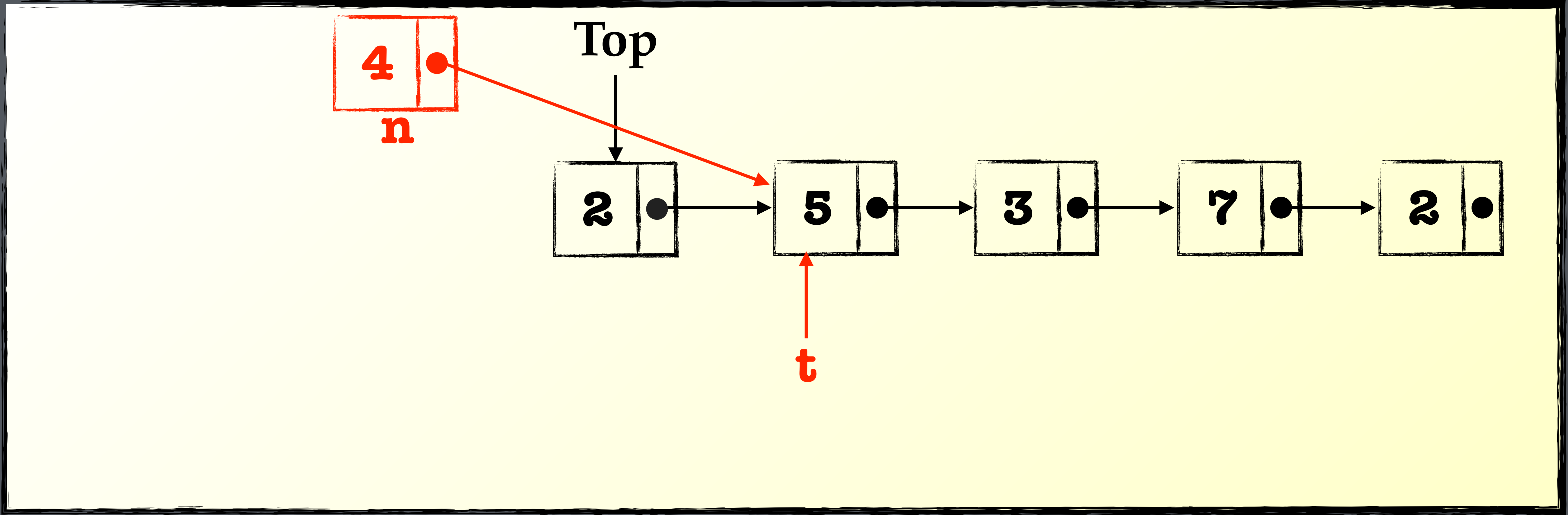


The Treiber Stack Algorithm

Concurrent push operations

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

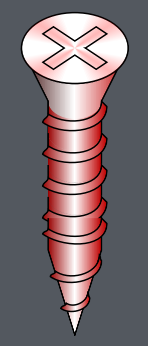
push(4)



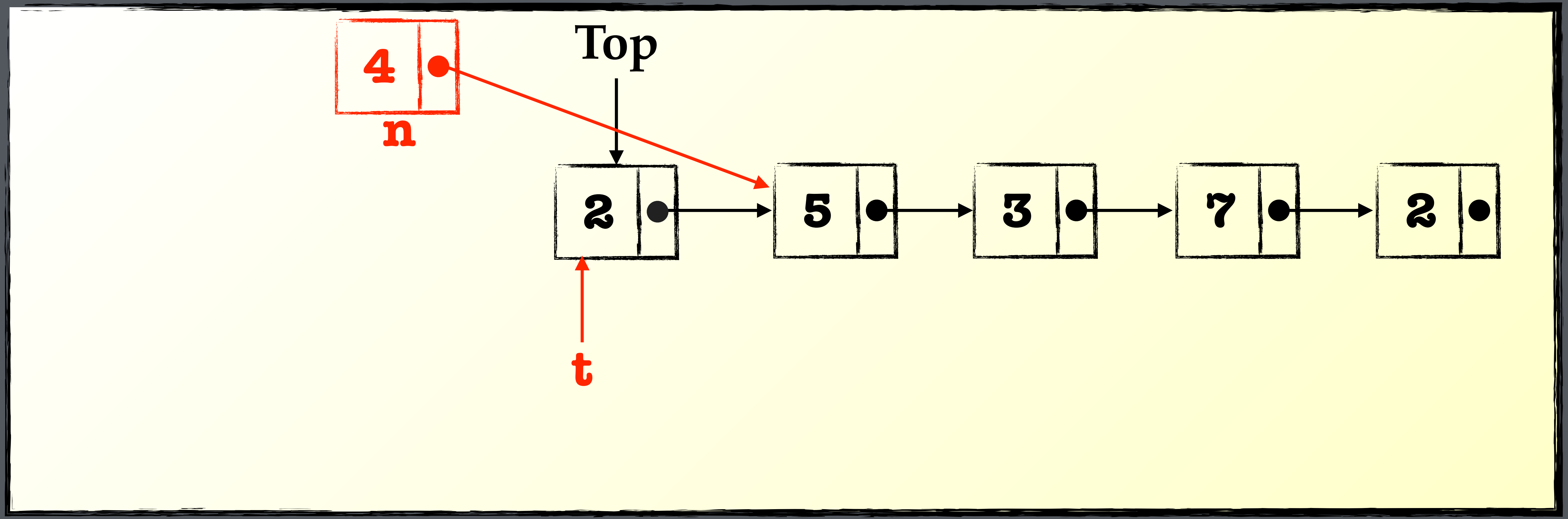
The Treiber Stack Algorithm

Concurrent push operations

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```



push(4)

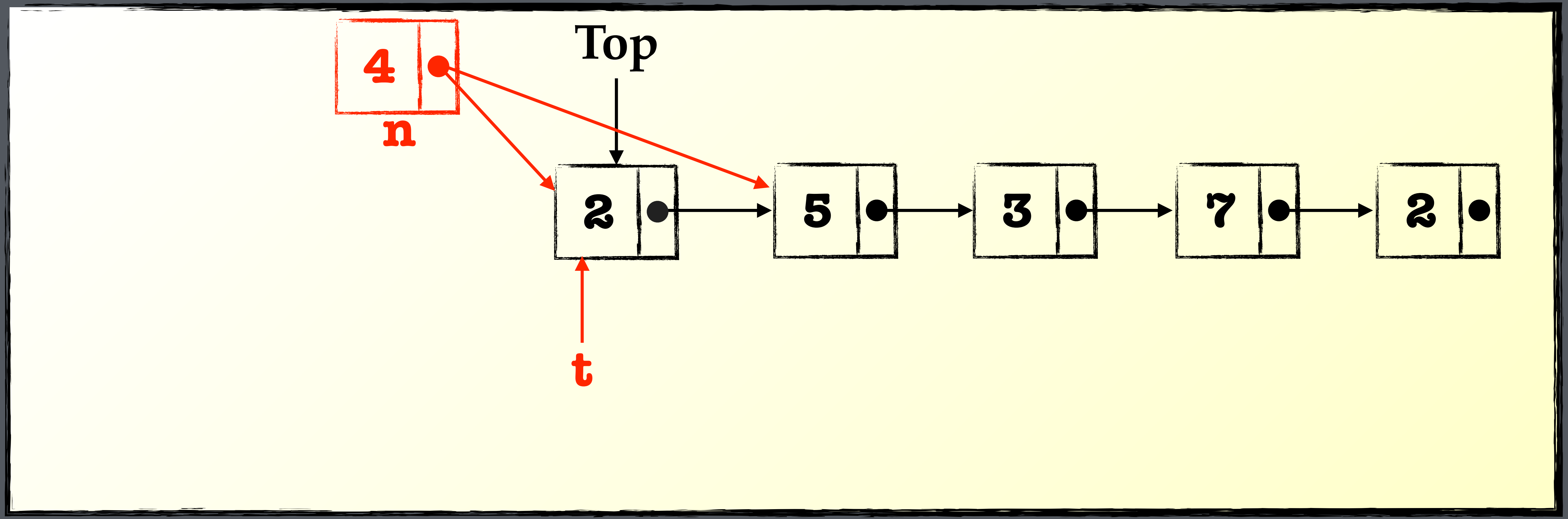


The Treiber Stack Algorithm

Concurrent push operations

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

push(4)



The Treiber Stack Algorithm

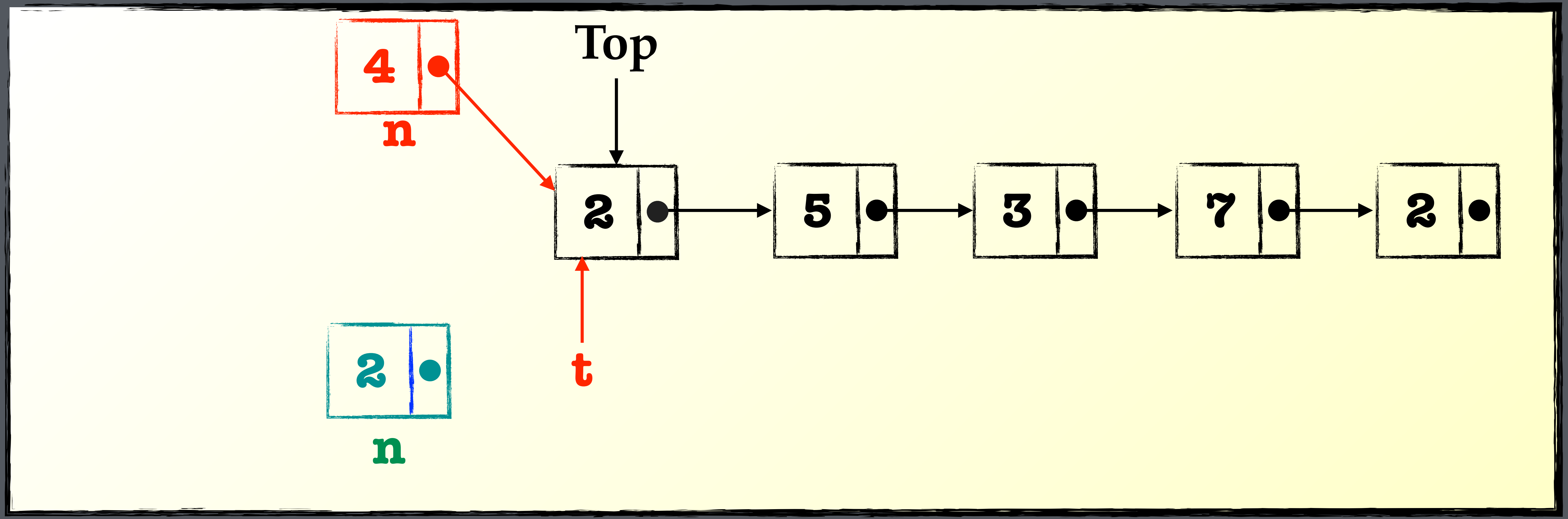
Concurrent push operations

```
push(k):
Node t
1  n = new Node(k,-)
2  while (true)
3    t = Top
4    n.next = t
5    if (CAS (Top,t,n))
6      exit
```

push(4)

push(2)

```
push(k):
Node t
1  n = new Node(k,-)
2  while (true)
3    t = Top
4    n.next = t
5    if (CAS (Top,t,n))
6      exit
```



The Treiber Stack Algorithm

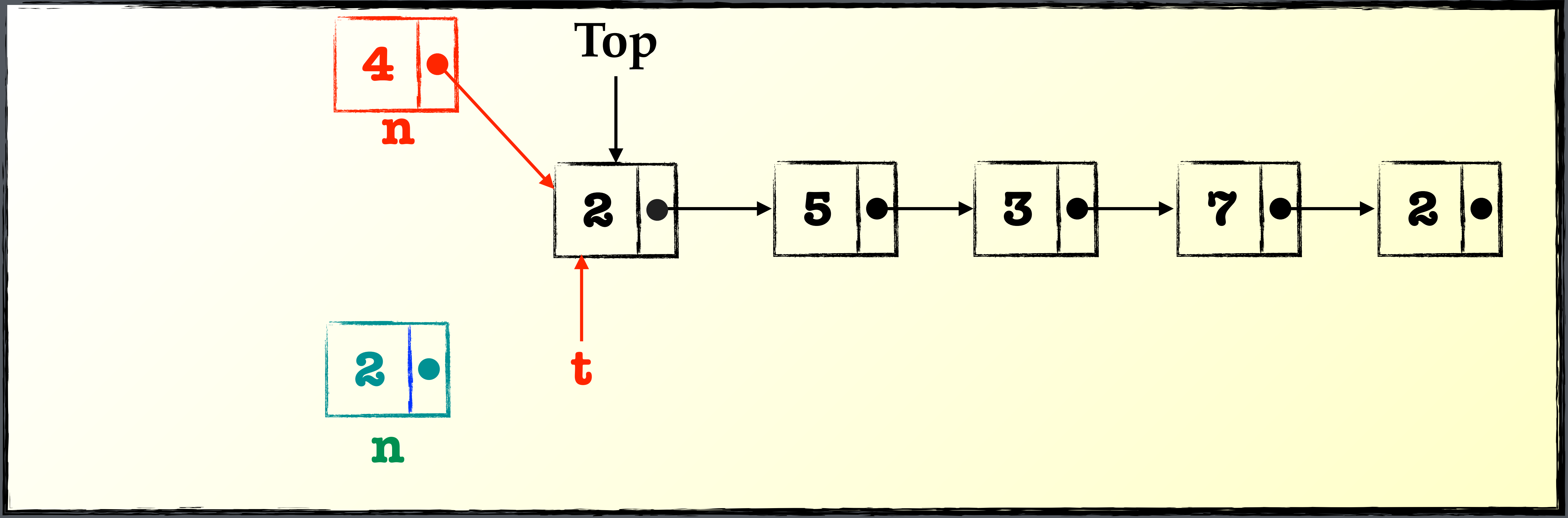
Concurrent push operations

```
push(k):
Node t
  n = new Node(k,-)
  while (true)
    t = Top
    n.next = t
    if (CAS (Top,t,n))
      exit
```

push(4)

push(2)

```
push(k):
Node t
  n = new Node(k,-)
  while (true)
    t = Top
    n.next = t
    if (CAS (Top,t,n))
      exit
```



The Treiber Stack Algorithm

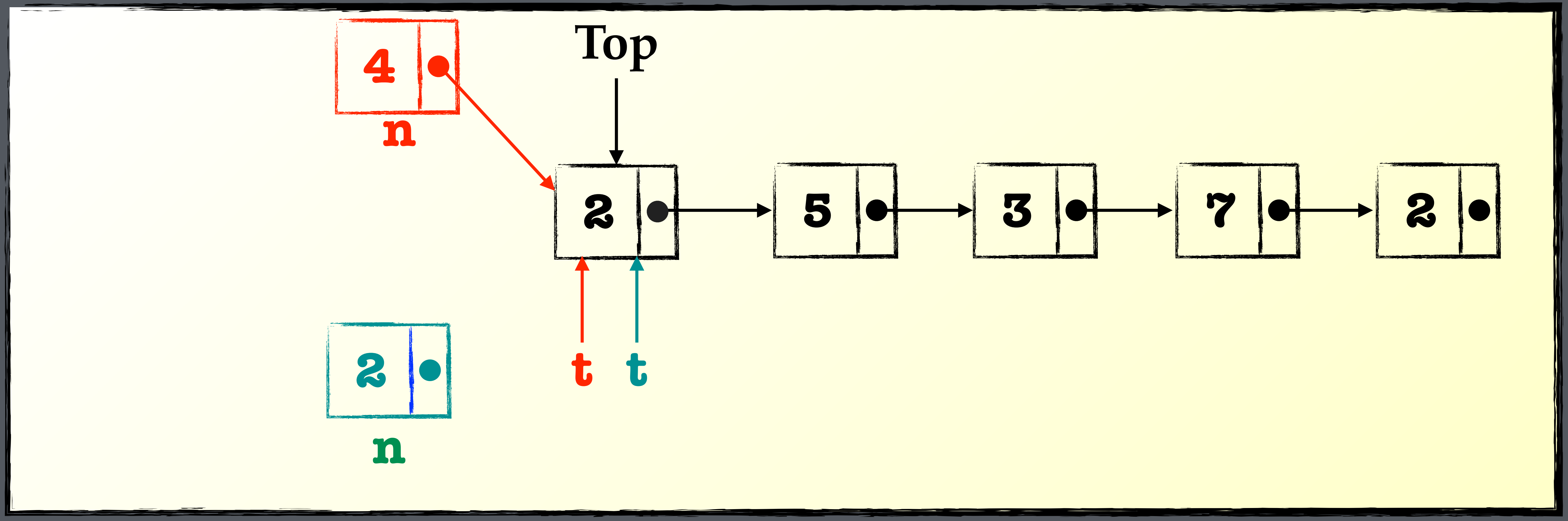
Concurrent push operations

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

push(4)

push(2)

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```



The Treiber Stack Algorithm

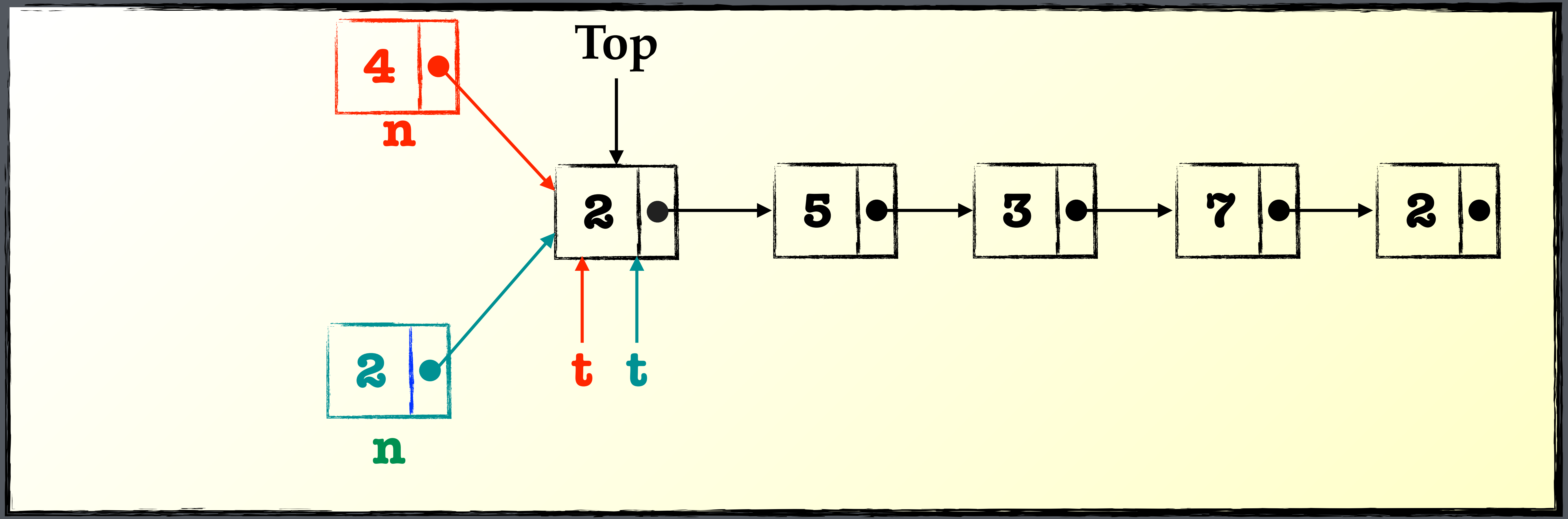
Concurrent push operations

```
push(k):
Node t
  n = new Node(k,-)
  while (true)
    t = Top
    n.next = t
    if (CAS (Top,t,n))
      exit
```

push(4)

push(2)

```
push(k):
Node t
  n = new Node(k,-)
  while (true)
    t = Top
    n.next = t
    if (CAS (Top,t,n))
      exit
```



The Treiber Stack Algorithm

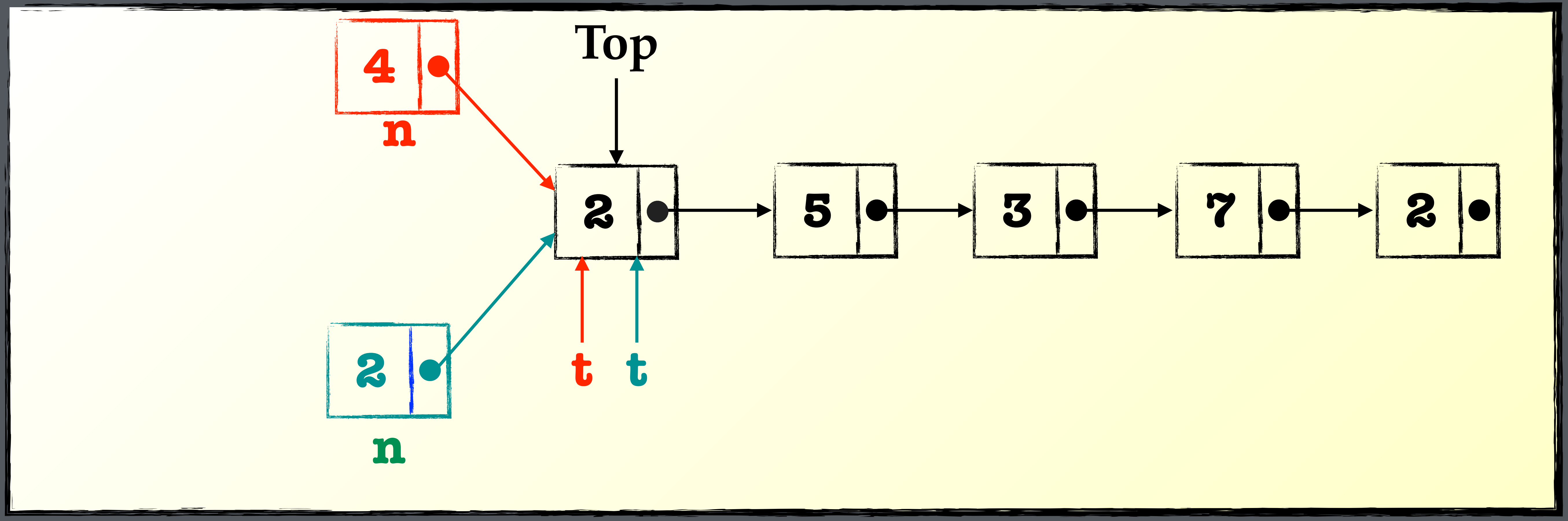
Concurrent push operations

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

push(4)

push(2)

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```



The Treiber Stack Algorithm

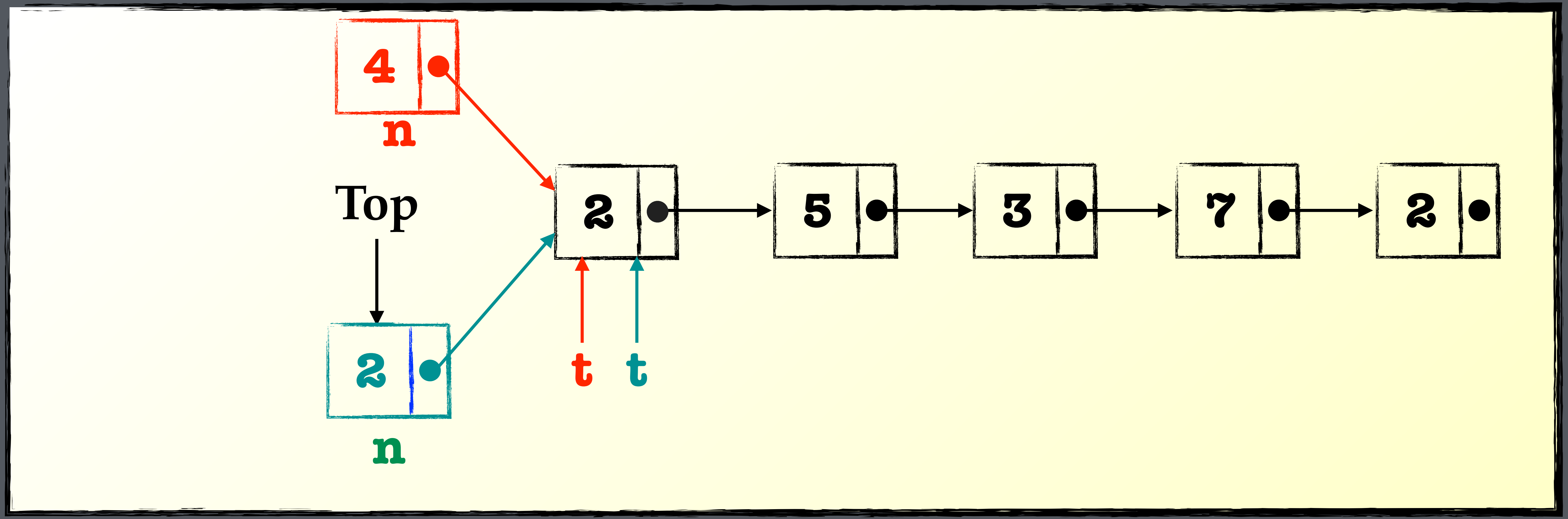
Concurrent push operations

```
push(k):
Node t
  n = new Node(k,-)
  while (true)
    t = Top
    n.next = t
    if (CAS (Top,t,n))
      exit
```

push(4)

push(2)

```
push(k):
Node t
  n = new Node(k,-)
  while (true)
    t = Top
    n.next = t
    if (CAS (Top,t,n))
      exit
```



The Treiber Stack Algorithm

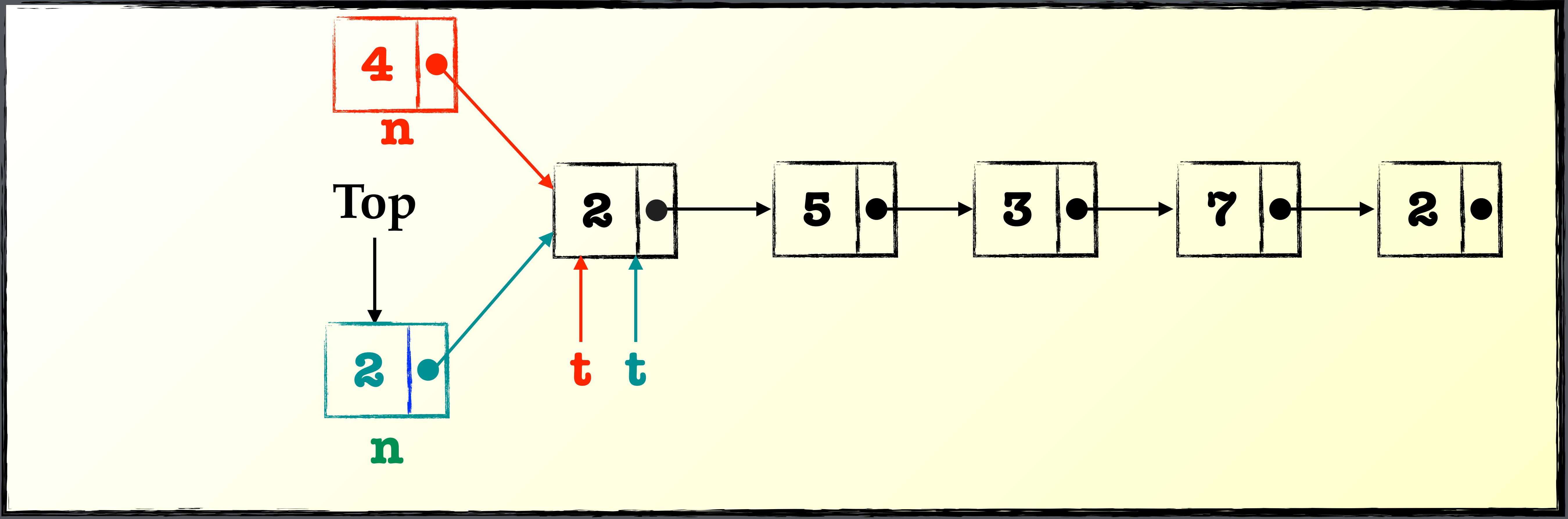
Concurrent push operations

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

push(4)

push(2)

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

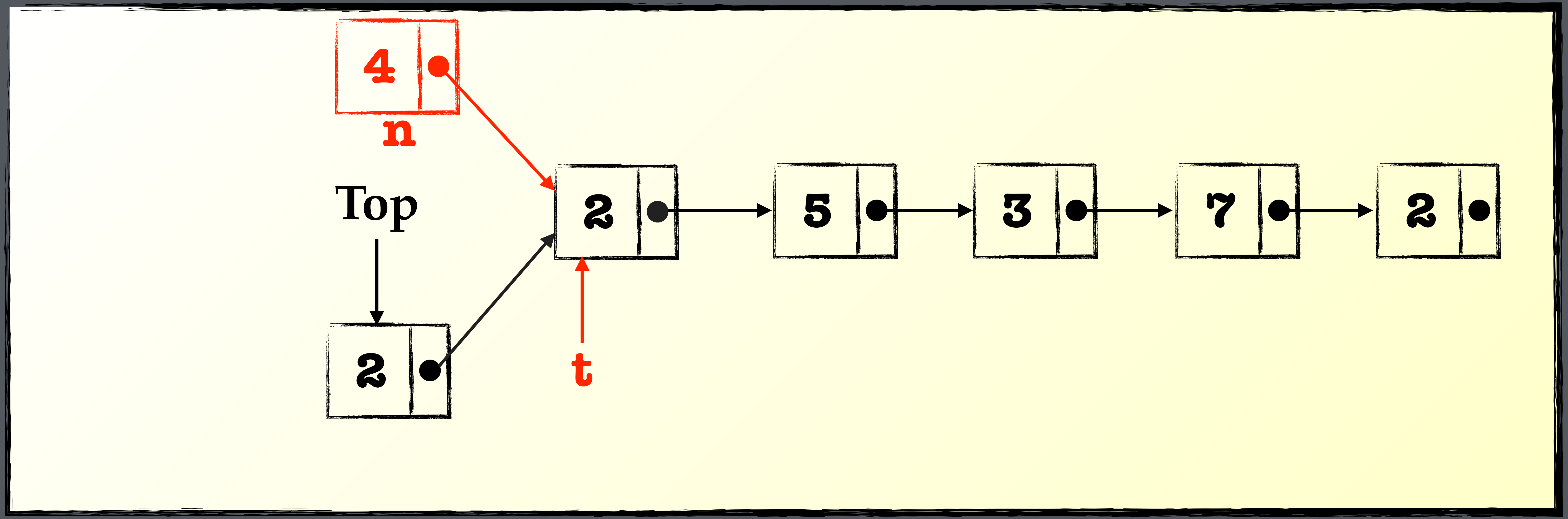


The Treiber Stack Algorithm

Concurrent push operations

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

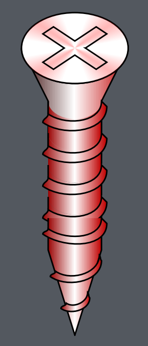
push(4)



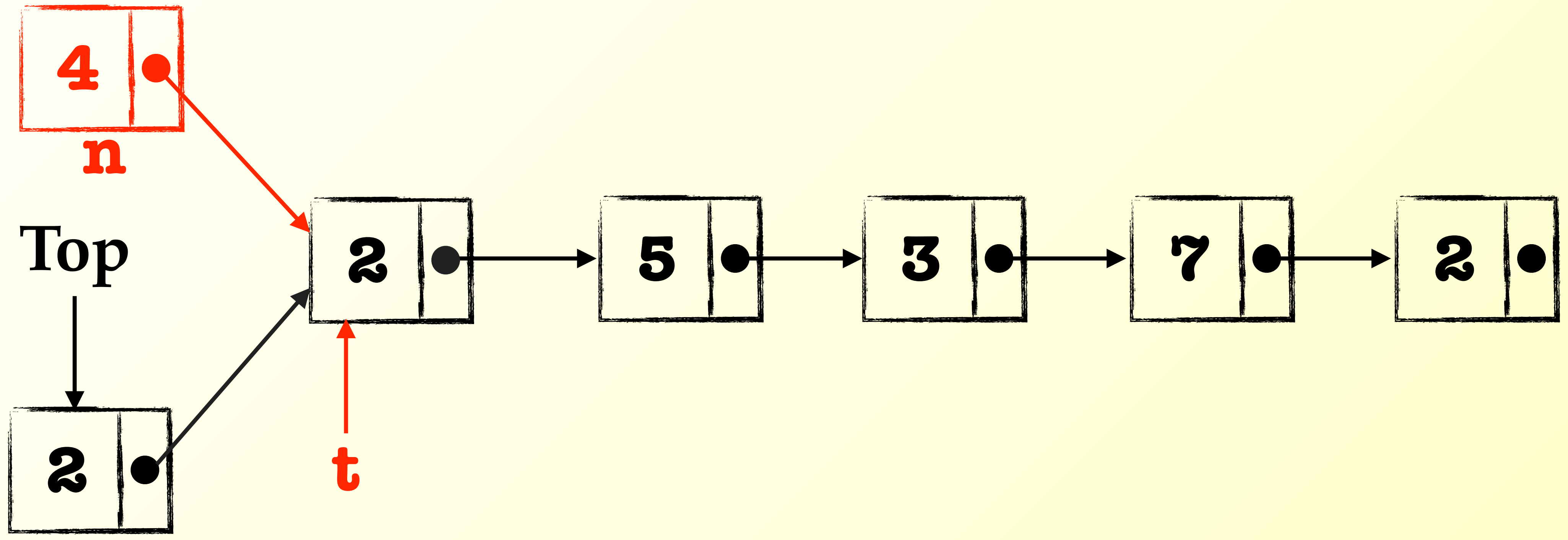
The Treiber Stack Algorithm

Concurrent push operations

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

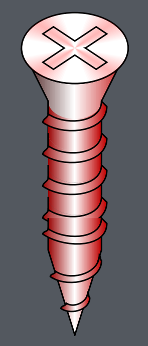


push(4)



The Treiber Stack Algorithm

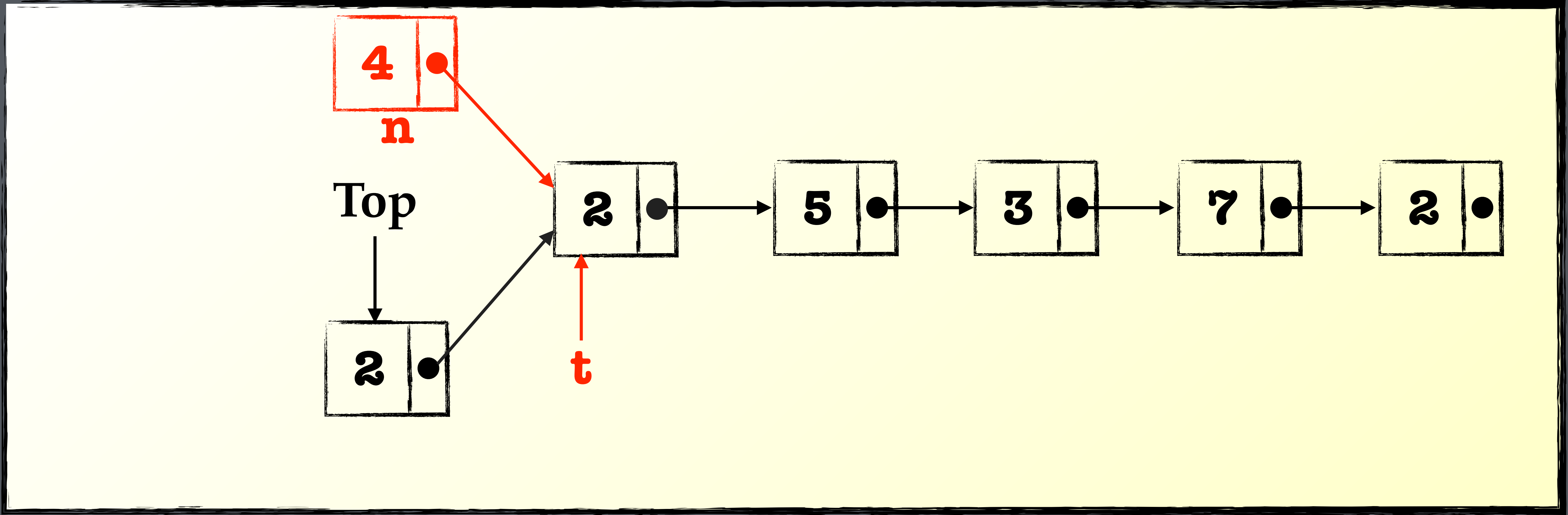
```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```



push(4)

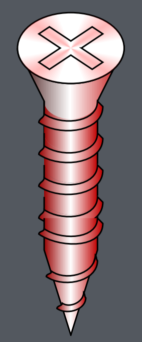
not wait-free (starvation)

lock-free



The Treiber Stack Algorithm

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

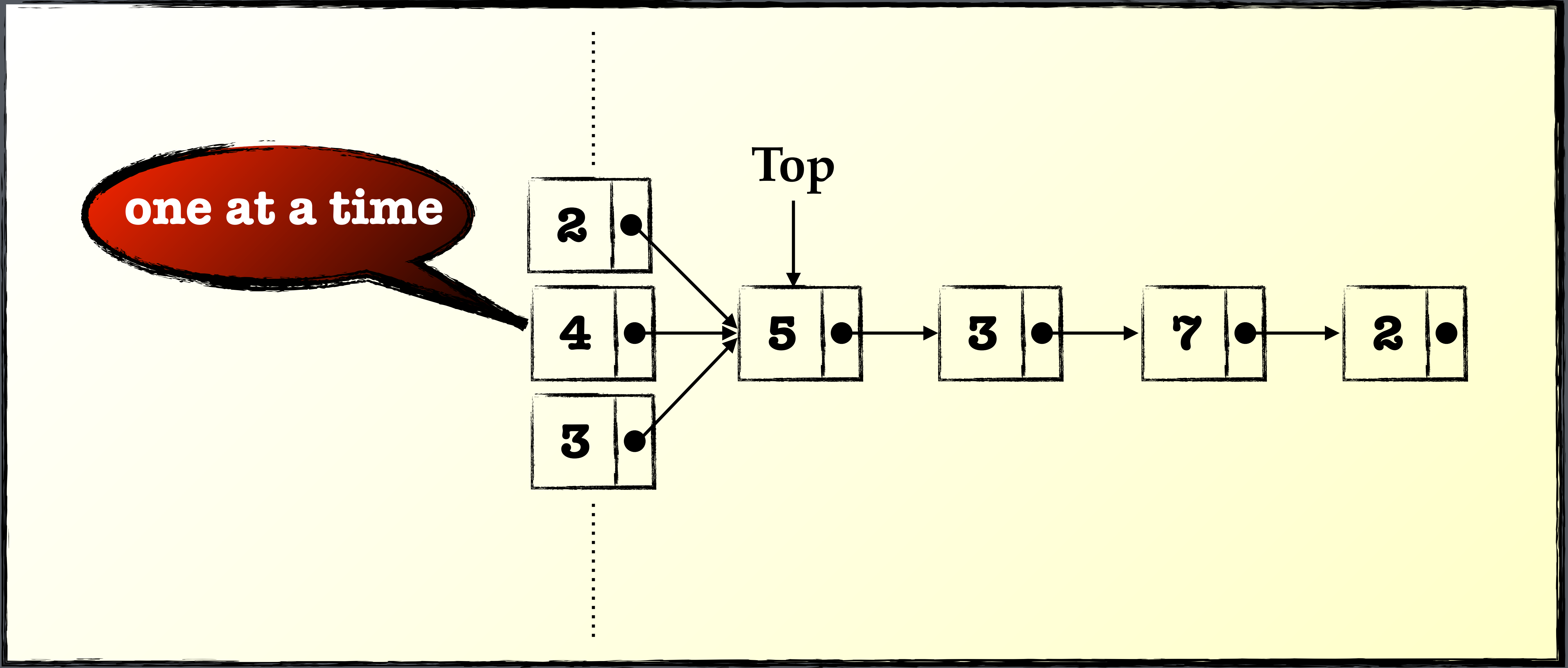


push(4)

not wait-free (starvation)

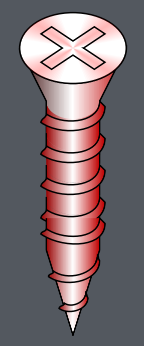
lock-free

essentially sequential

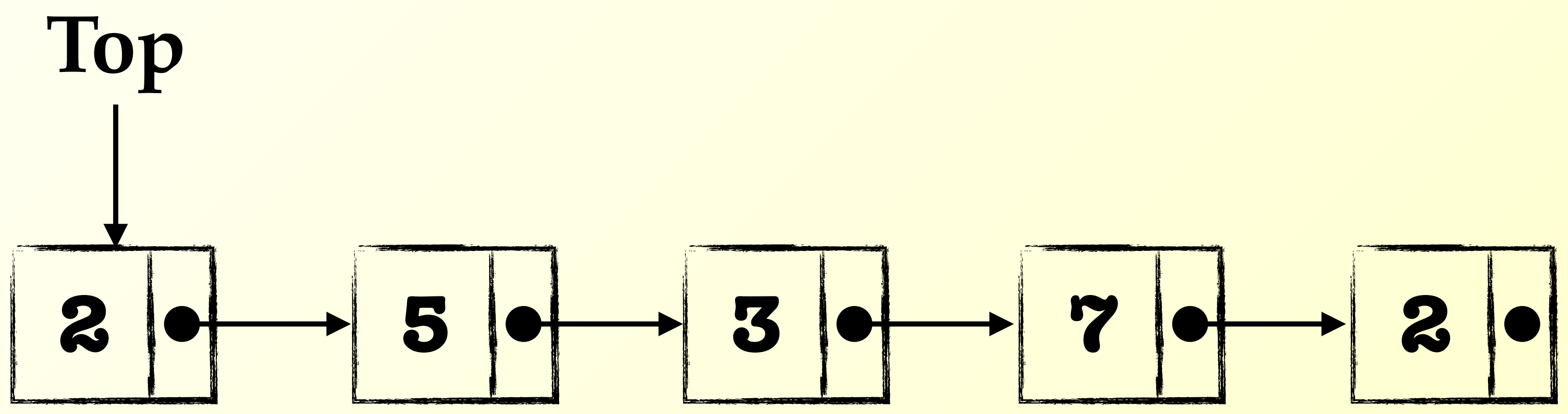


The Treiber Stack Algorithm

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```



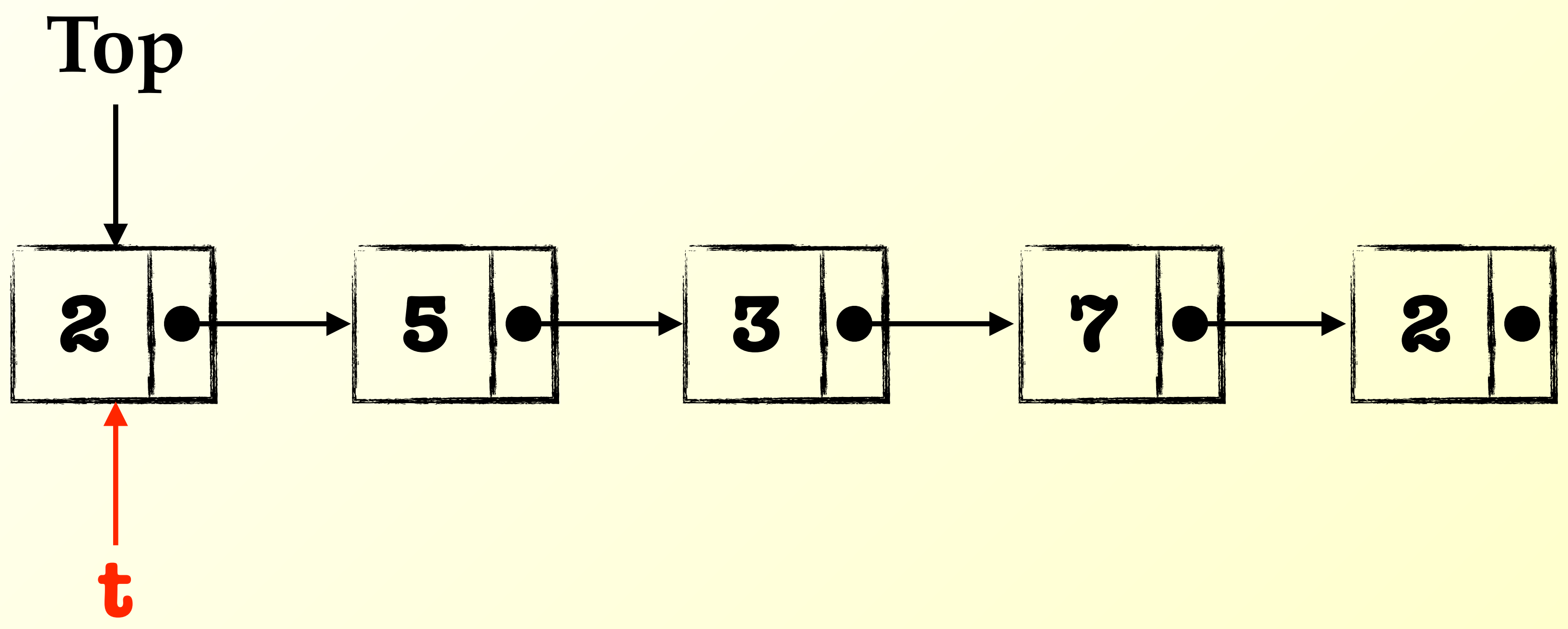
pop



The Treiber Stack Algorithm

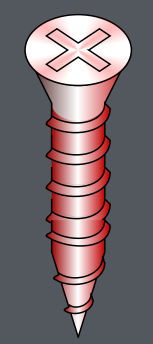
```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

pop

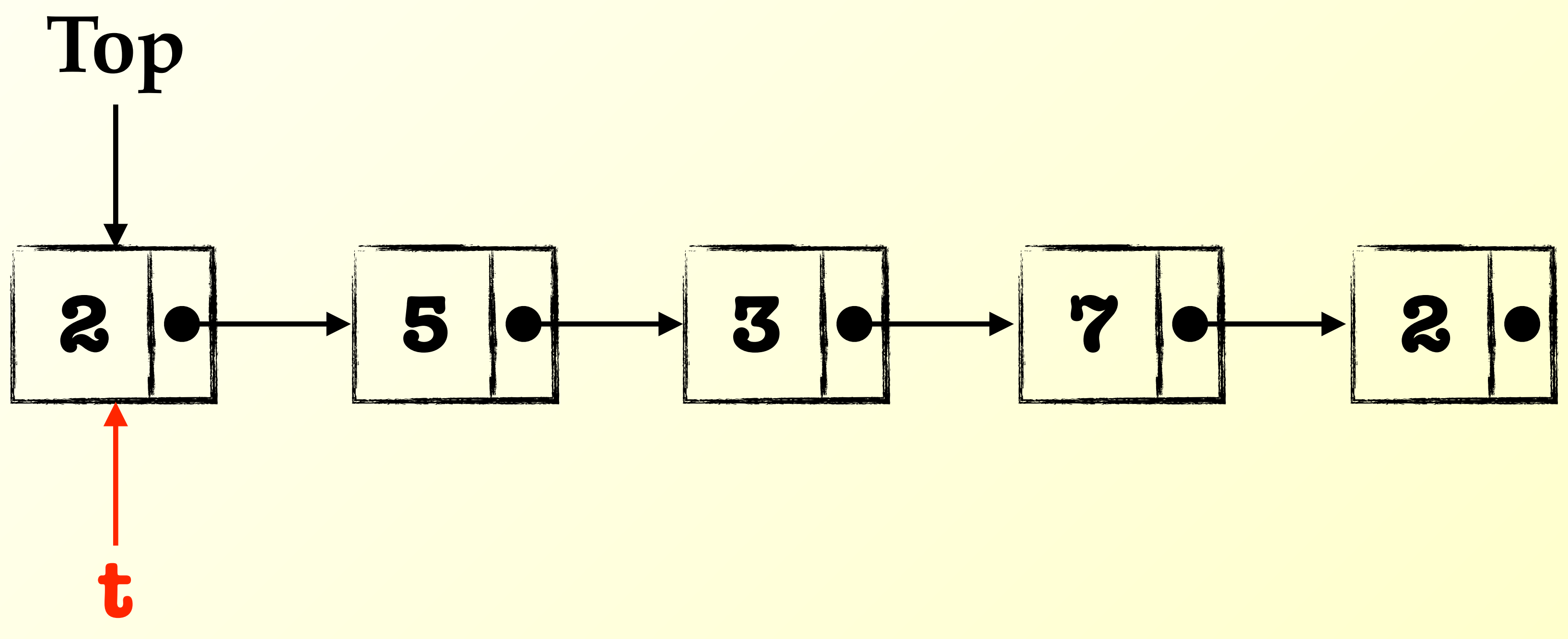


The Treiber Stack Algorithm

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```



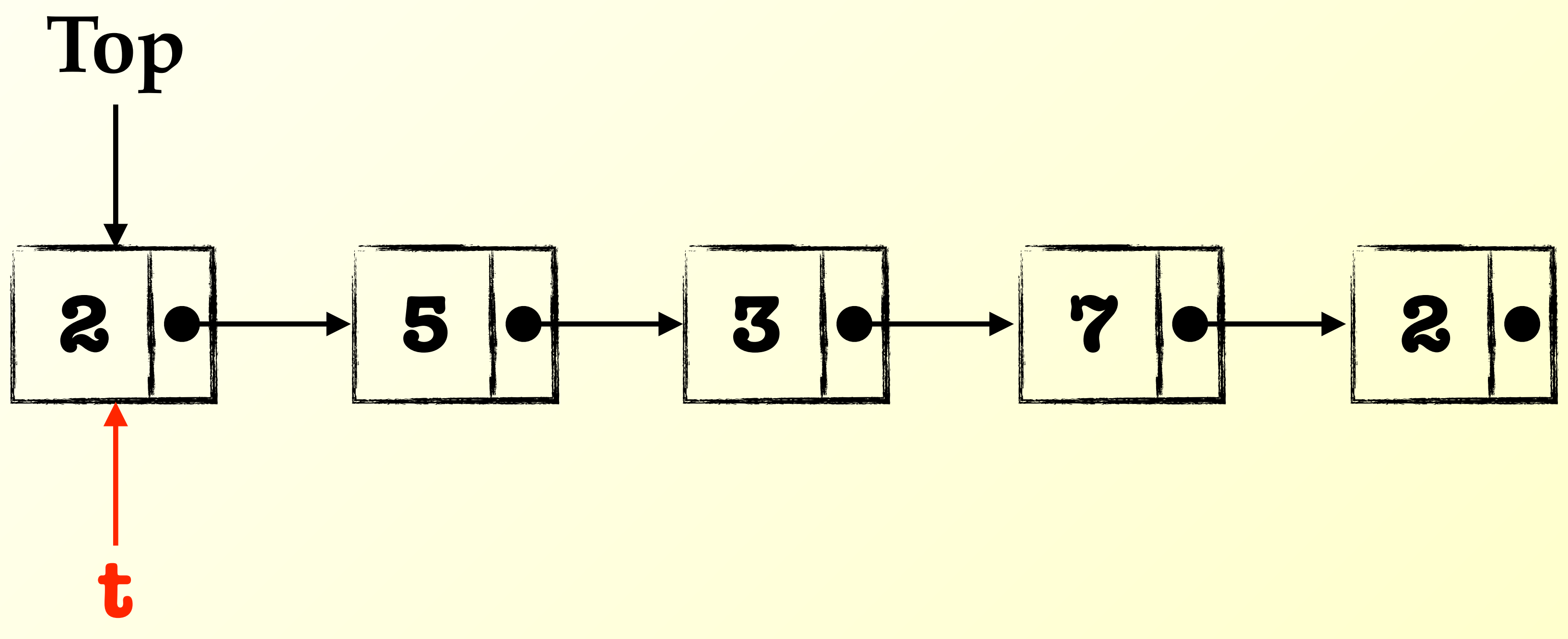
pop



The Treiber Stack Algorithm

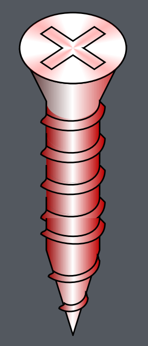
```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

pop

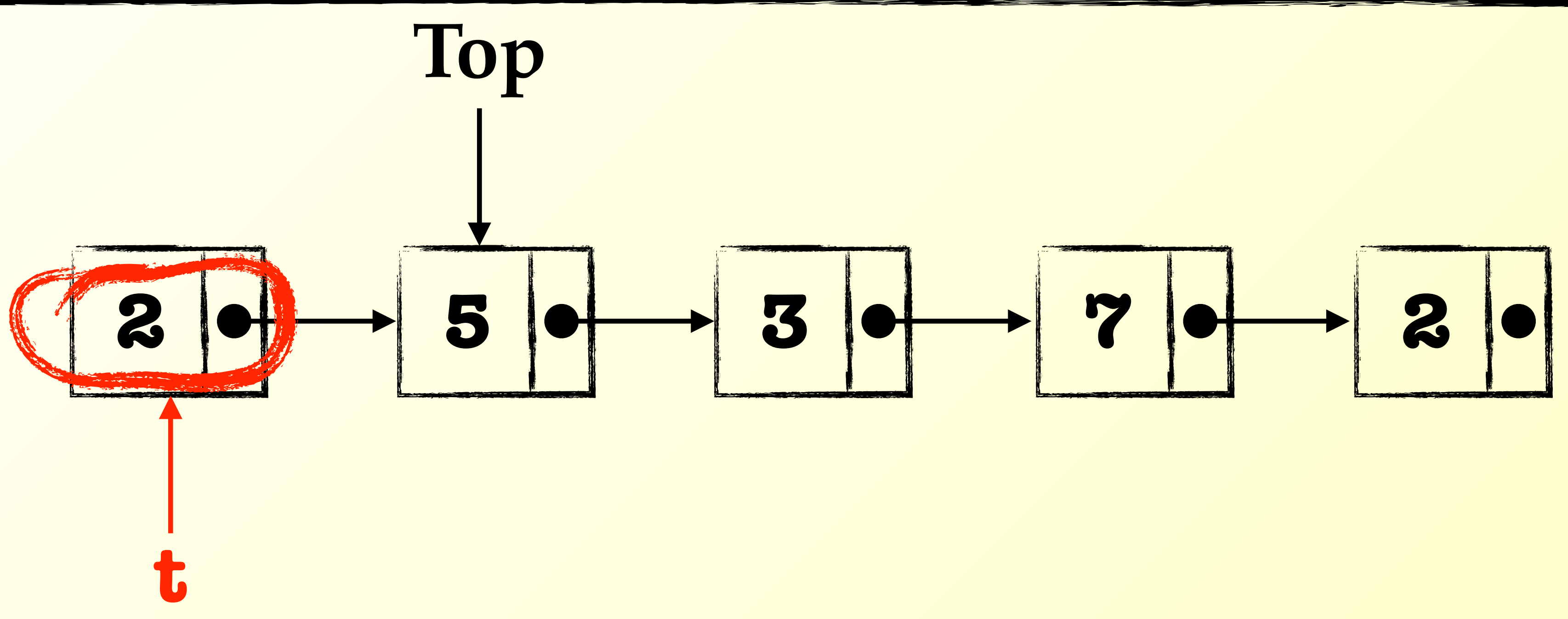


The Treiber Stack Algorithm

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

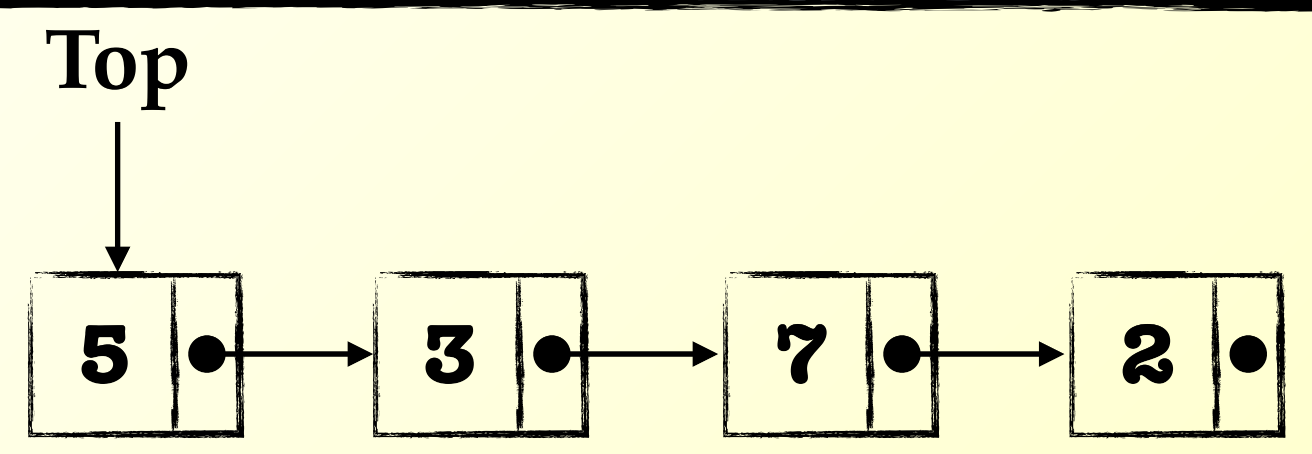


pop



The Treiber Stack Algorithm

Concurrent pop operations

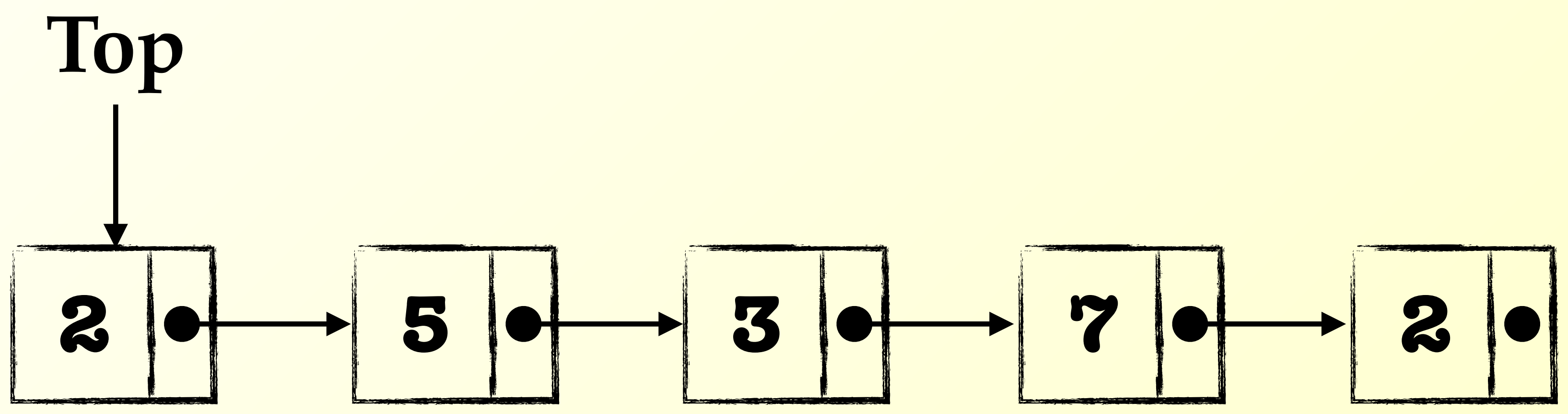


The Treiber Stack Algorithm

Concurrent pop operations

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

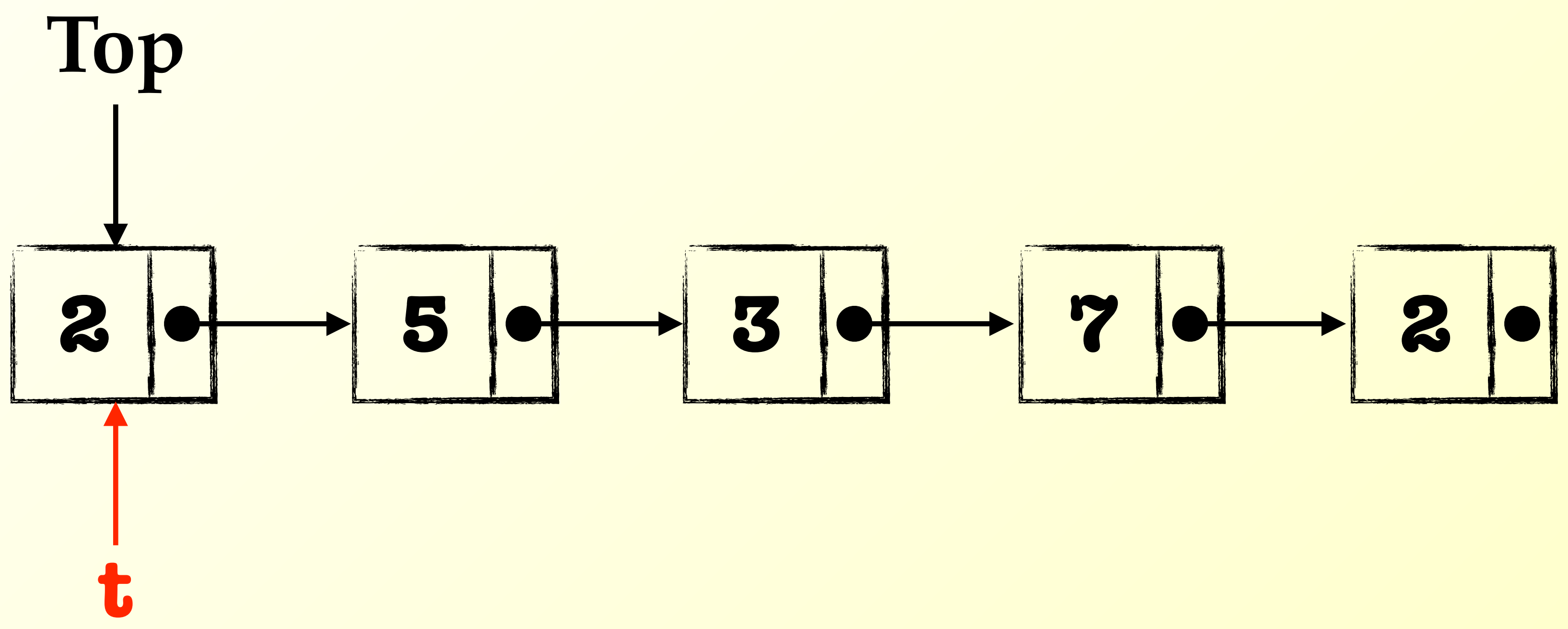
pop



The Treiber Stack Algorithm

Concurrent pop operations

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```



The Treiber Stack Algorithm

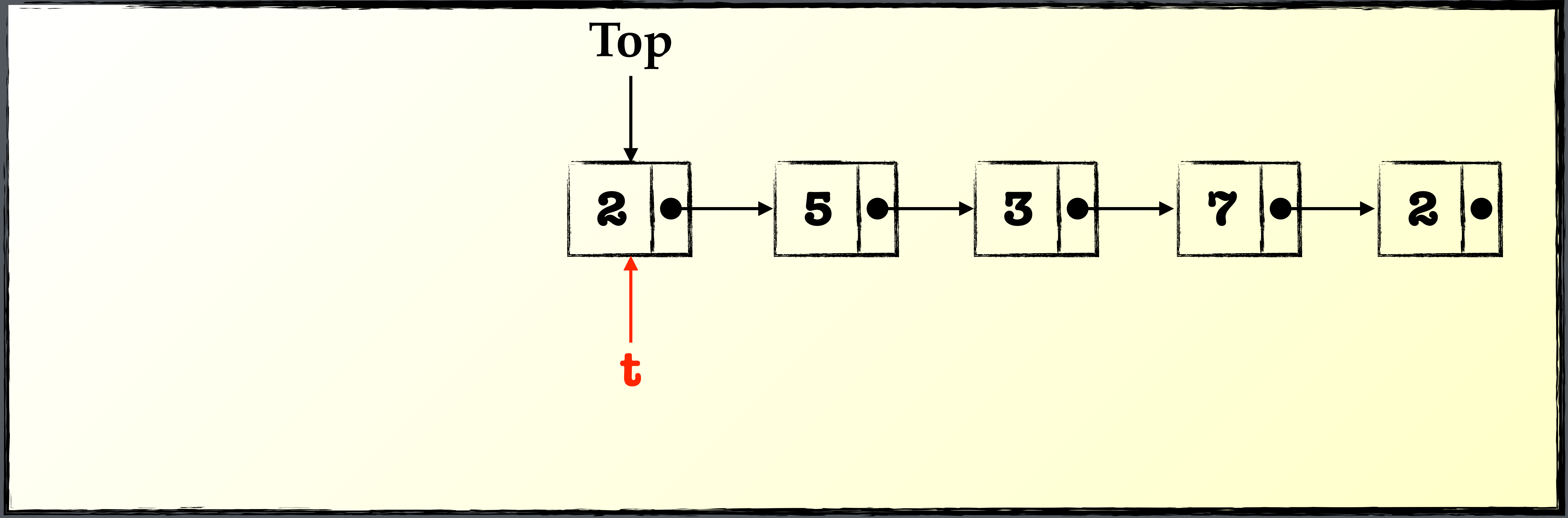
Concurrent pop operations

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

pop

pop

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```



The Treiber Stack Algorithm

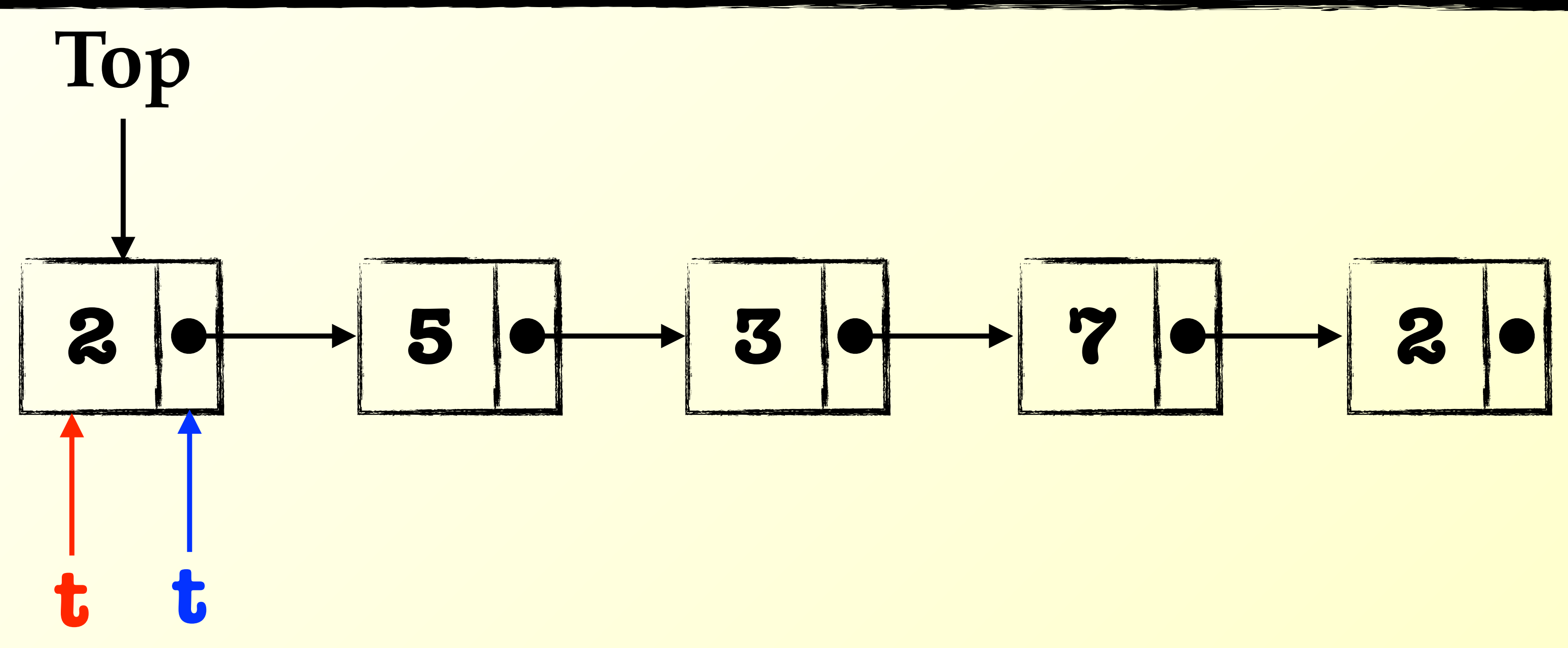
Concurrent pop operations

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

pop

pop

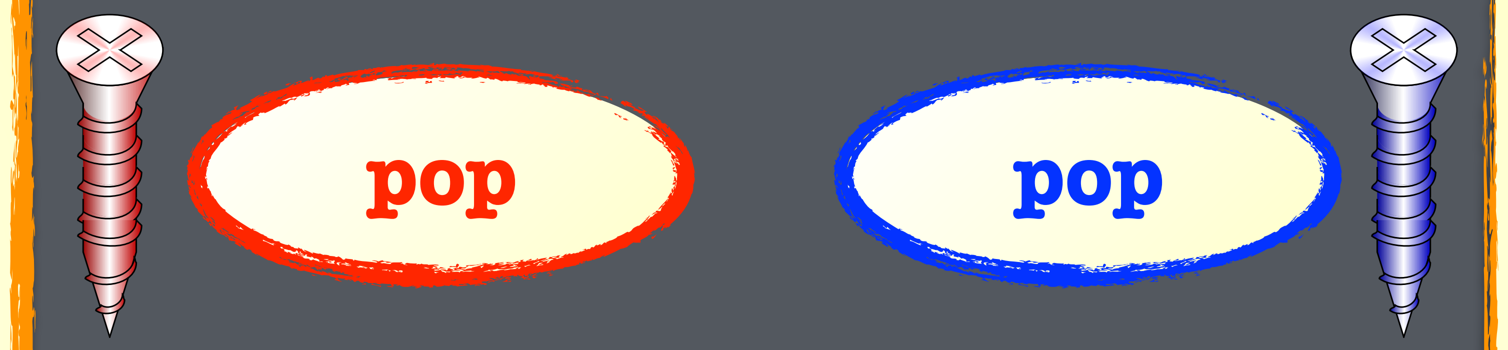
```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```



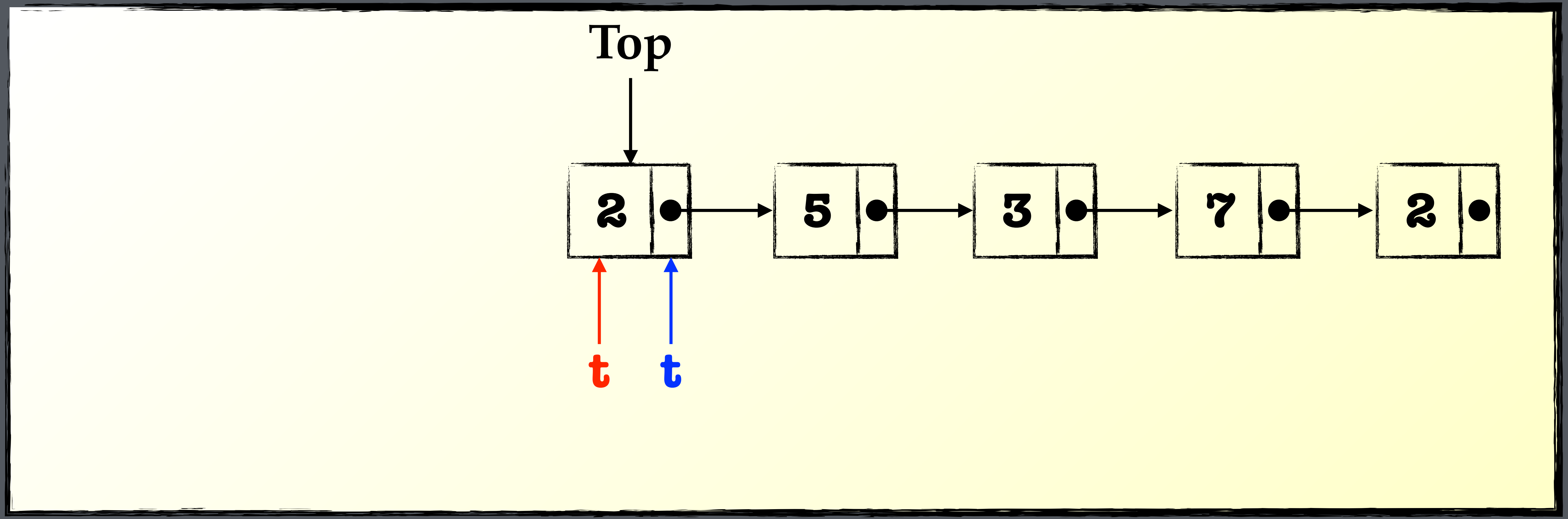
The Treiber Stack Algorithm

Concurrent pop operations

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```




```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```



The Treiber Stack Algorithm

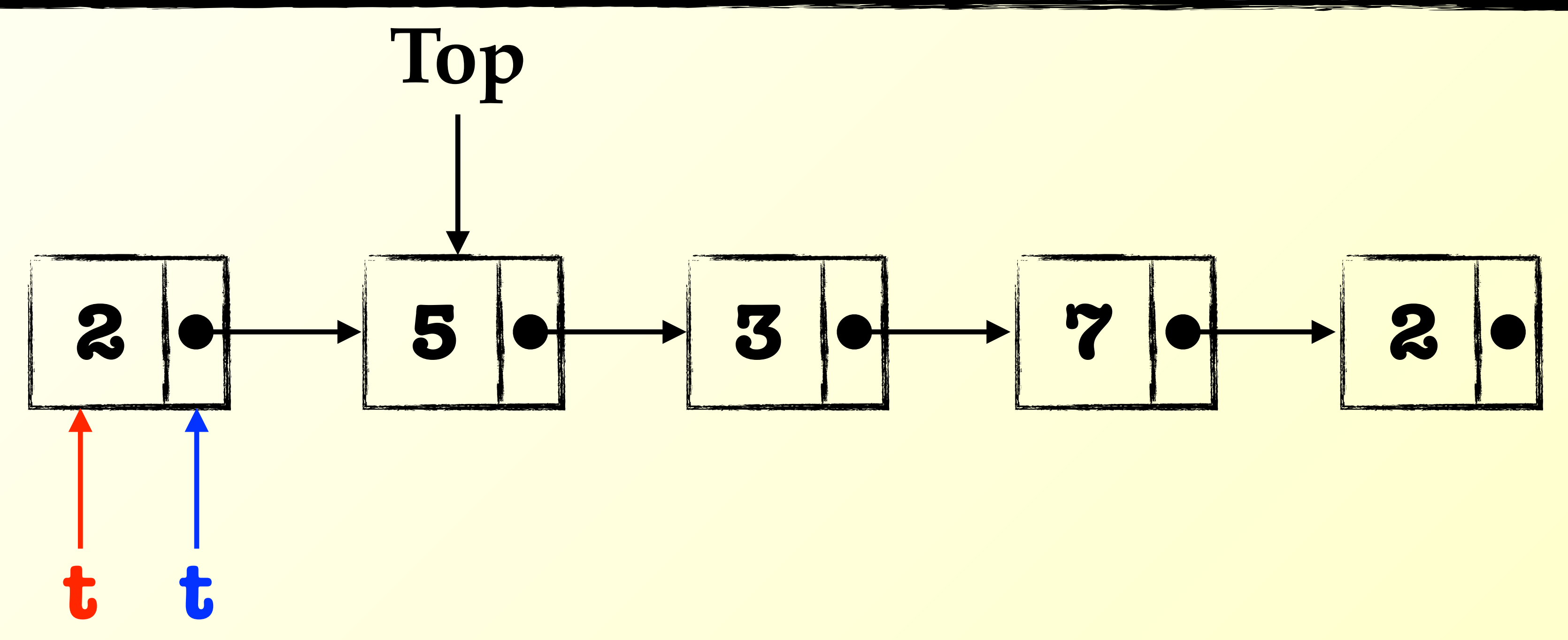
Concurrent pop operations

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

pop

pop

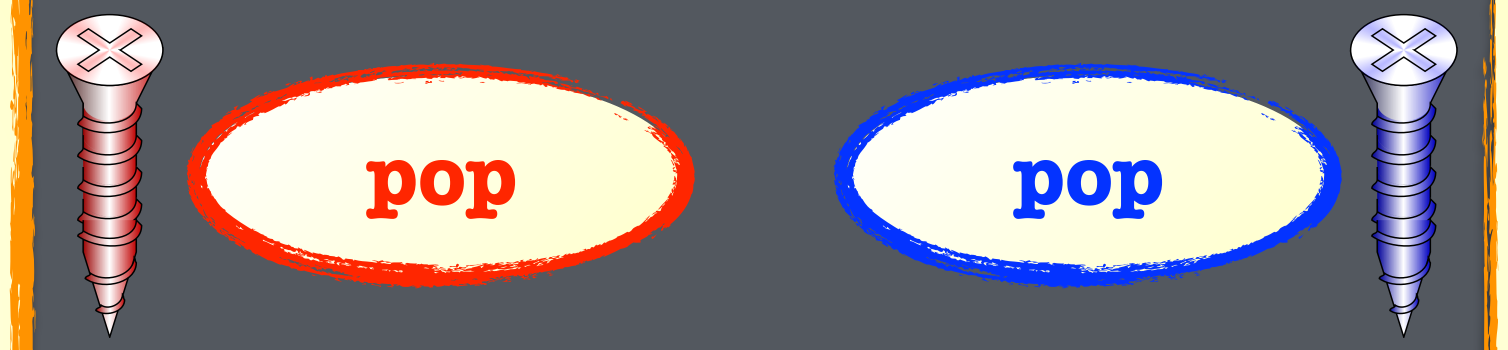
```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```



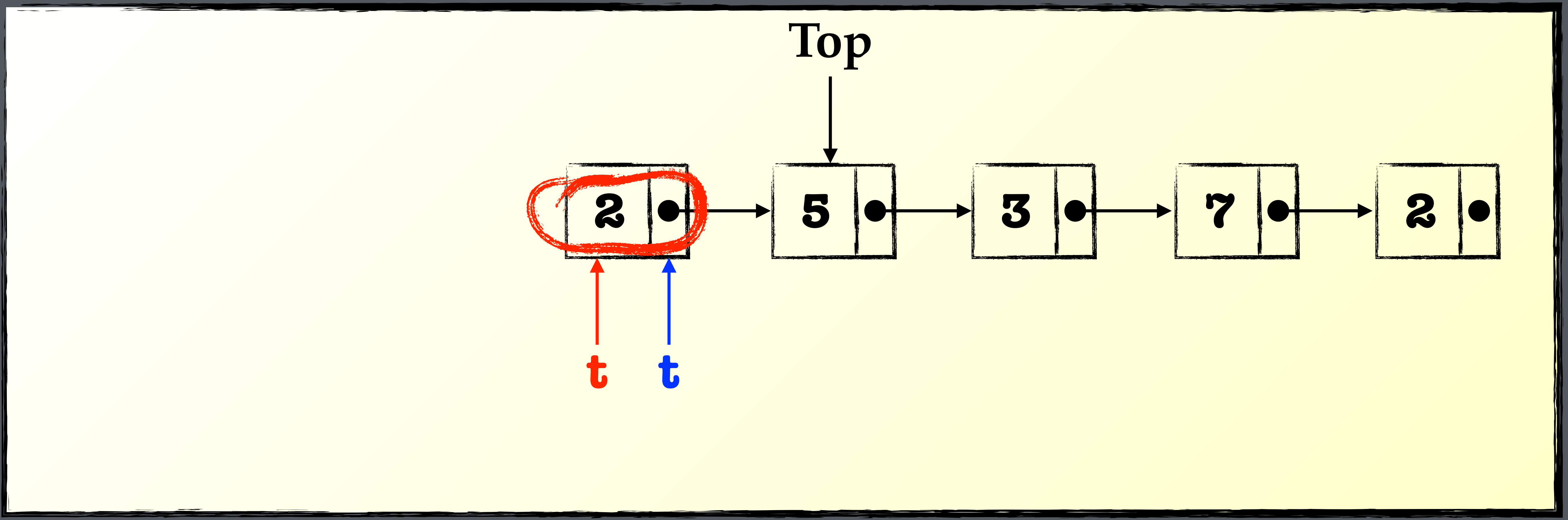
The Treiber Stack Algorithm

Concurrent pop operations

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```




```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```



The Treiber Stack Algorithm

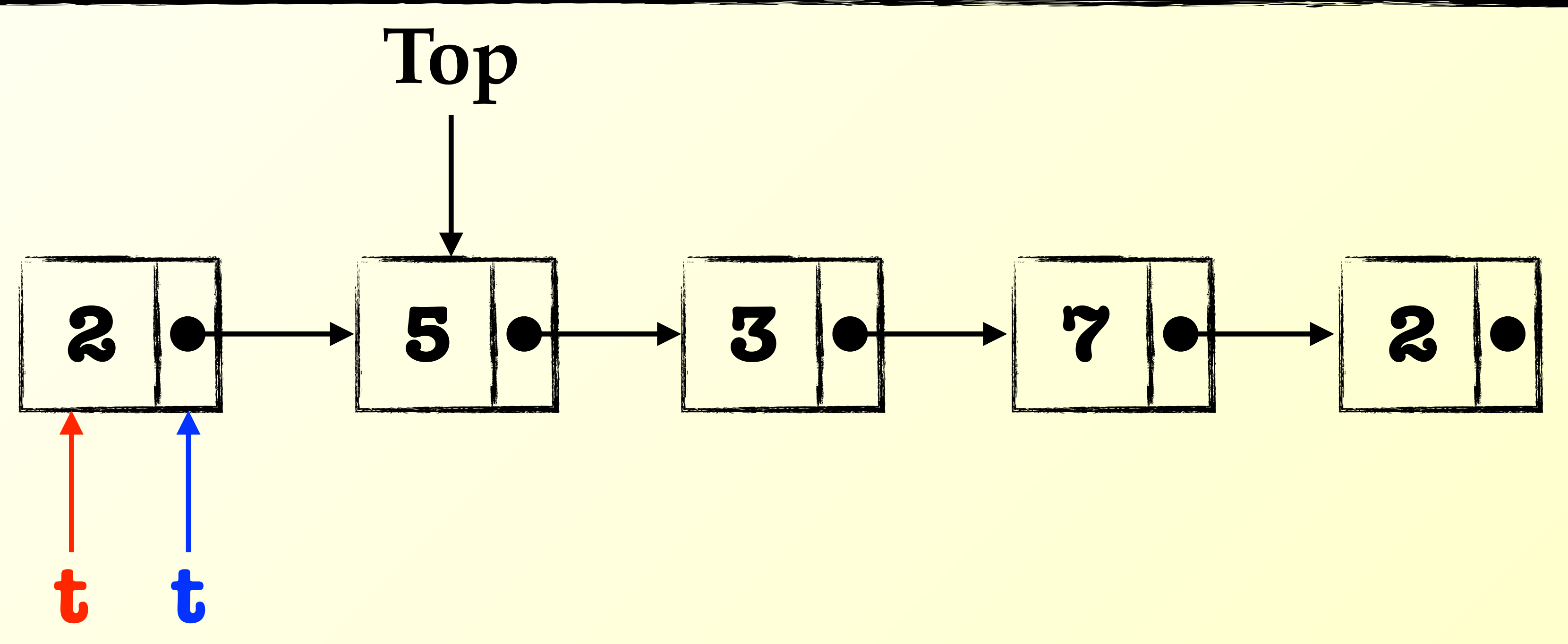
Concurrent pop operations

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

pop

pop

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

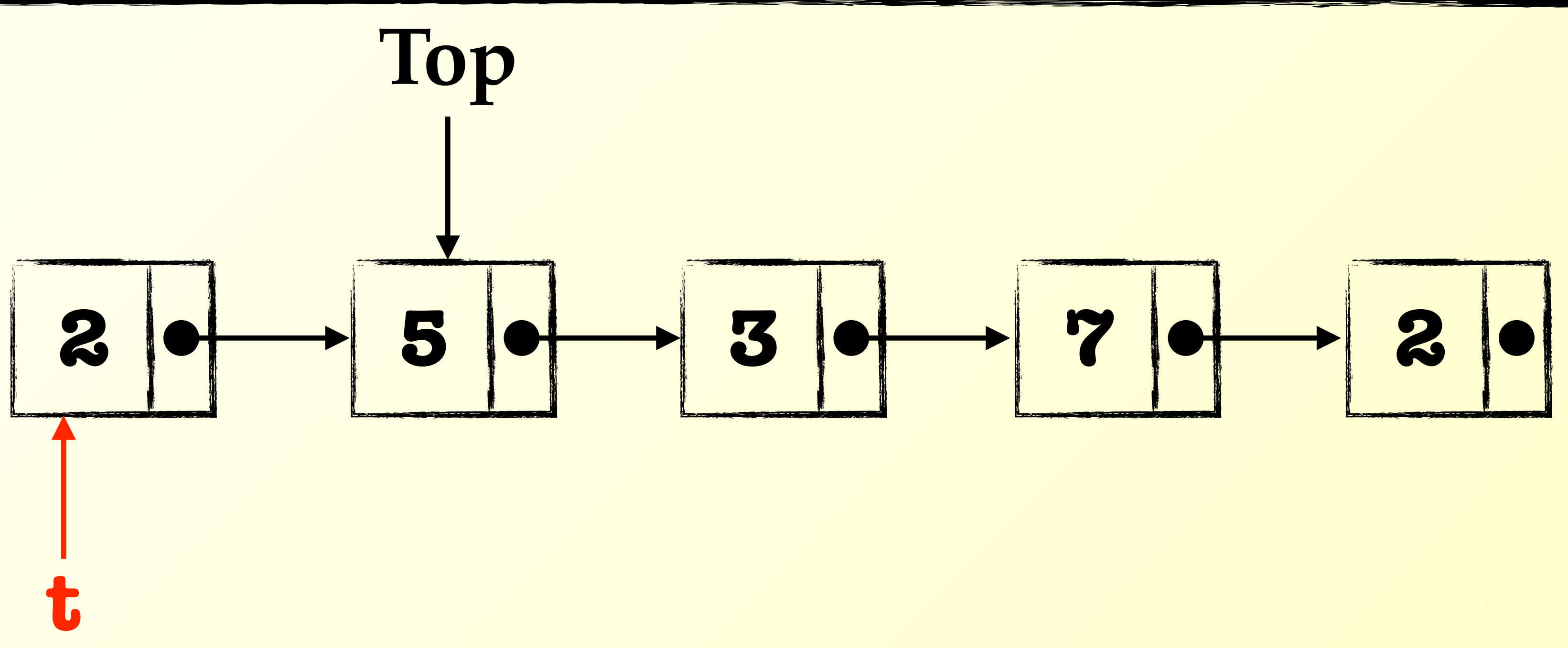


The Treiber Stack Algorithm

Concurrent pop operations

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

pop

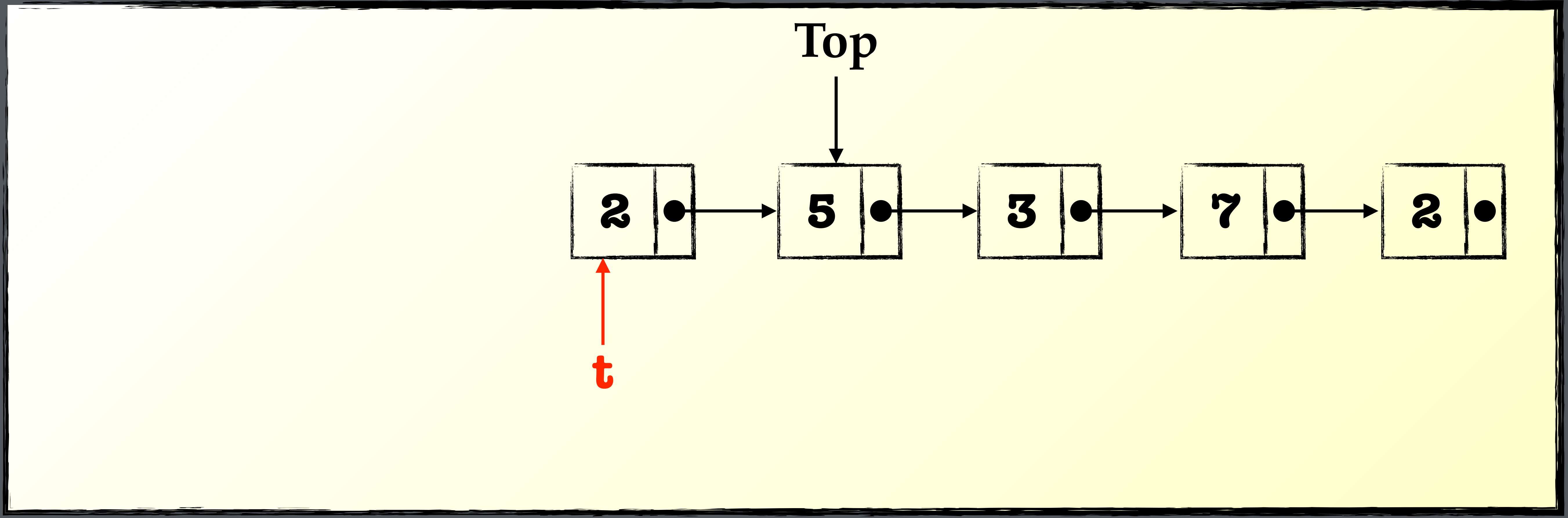


The Treiber Stack Algorithm

Concurrent pop operations

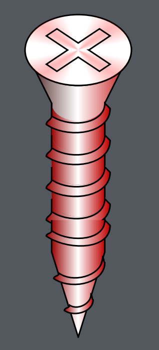
```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

pop

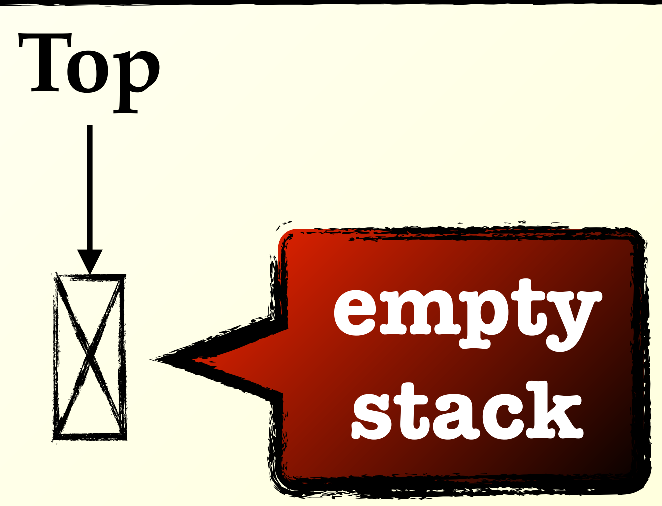


The Treiber Stack Algorithm

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

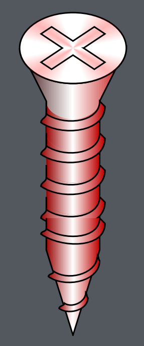


pop

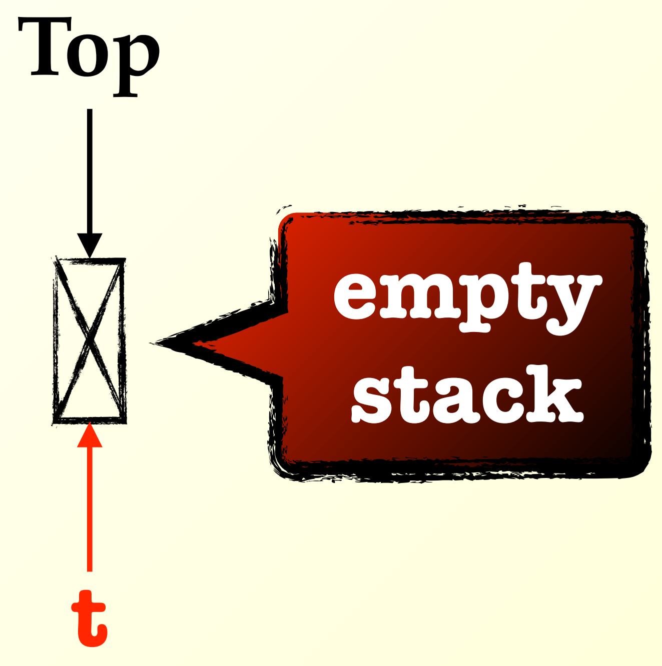


The Treiber Stack Algorithm

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

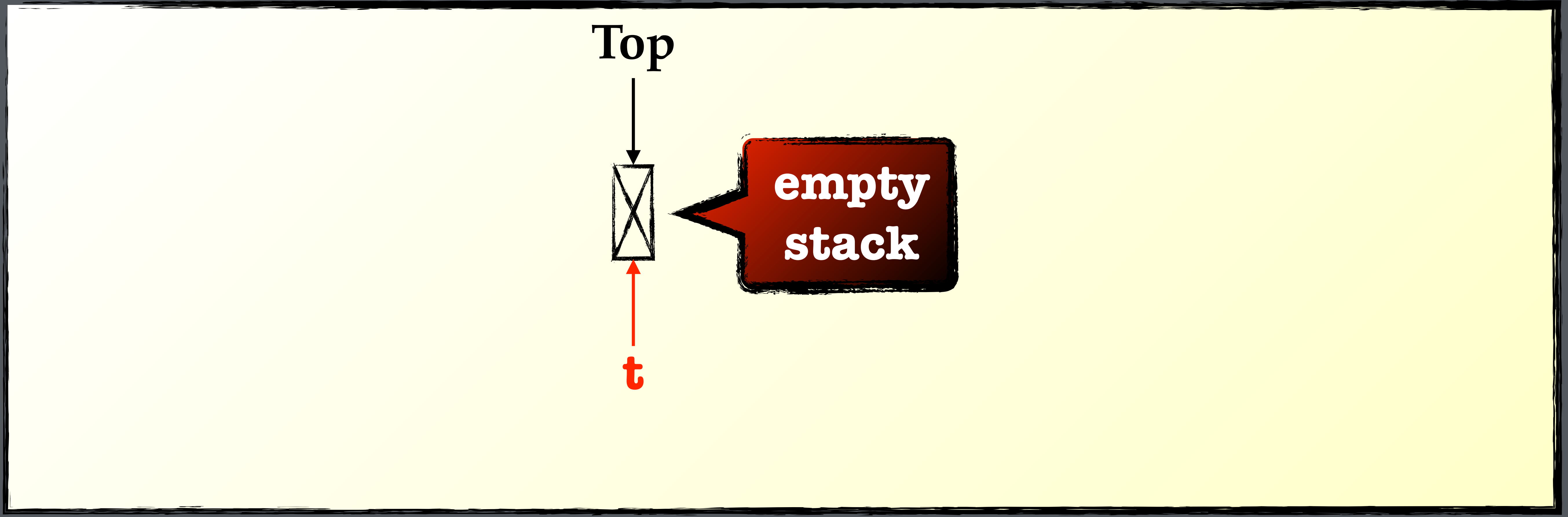


pop



The Treiber Stack Algorithm

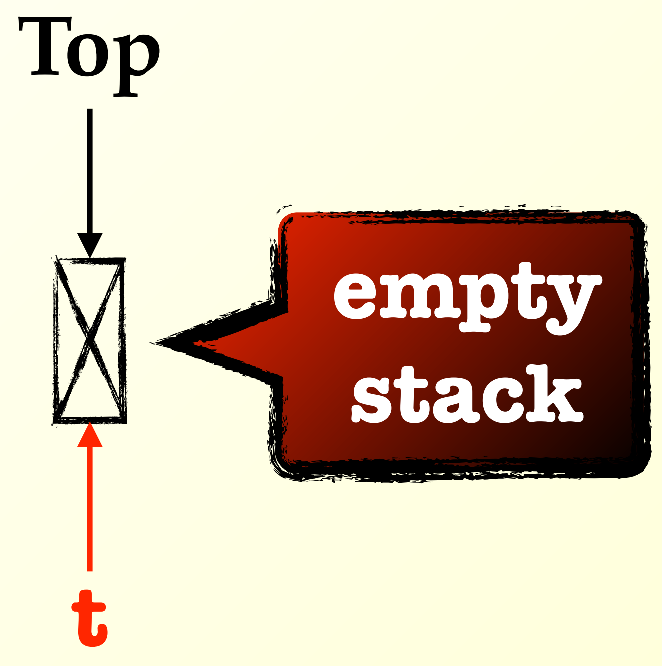
```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```



The Treiber Stack Algorithm

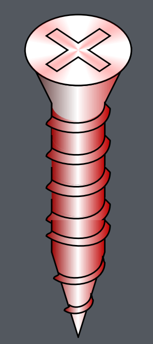
```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

pop

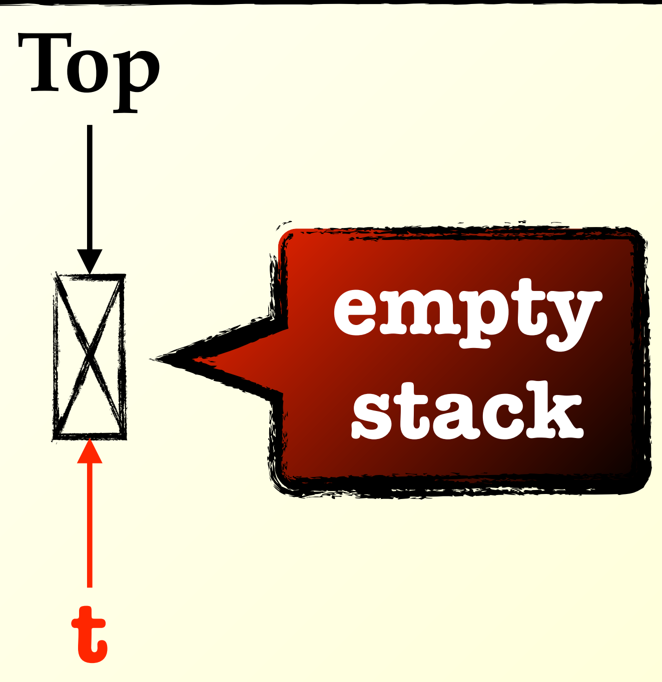


The Treiber Stack Algorithm

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```



pop



The Treiber Stack Algorithm

Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

when successful

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

when empty

when successful

The Treiber Stack Algorithm

Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

when successful

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

when empty

?

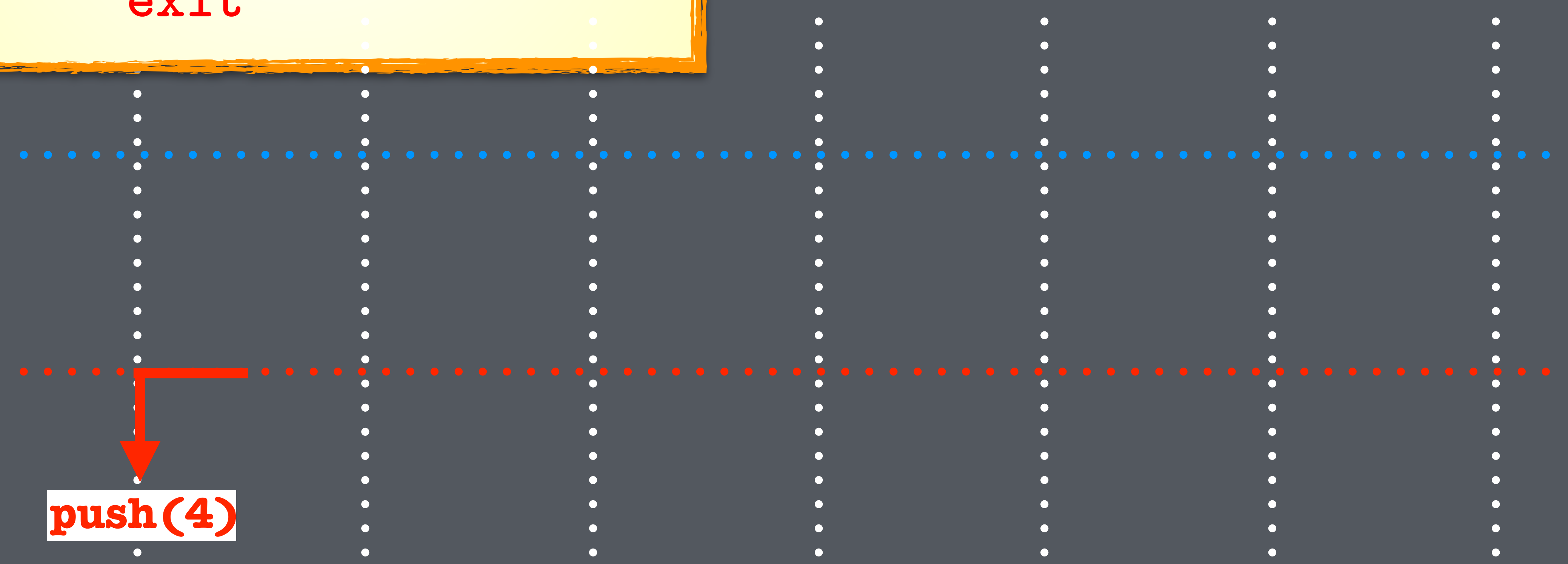
when successful

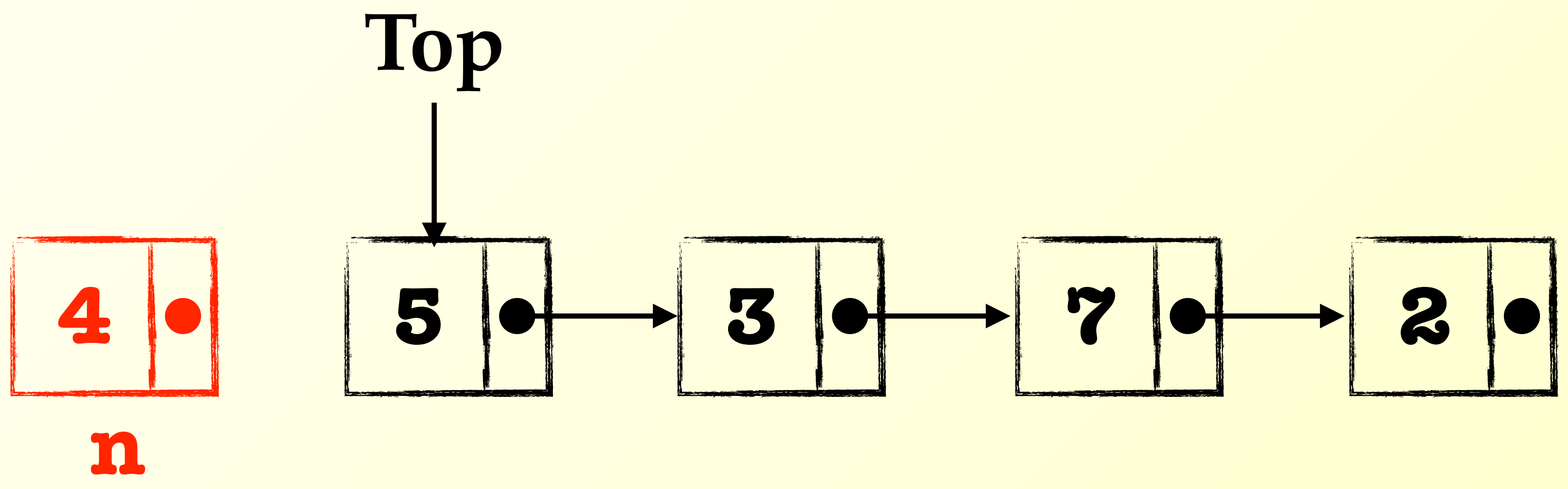
The Treiber Stack Algorithm

Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

push(4)



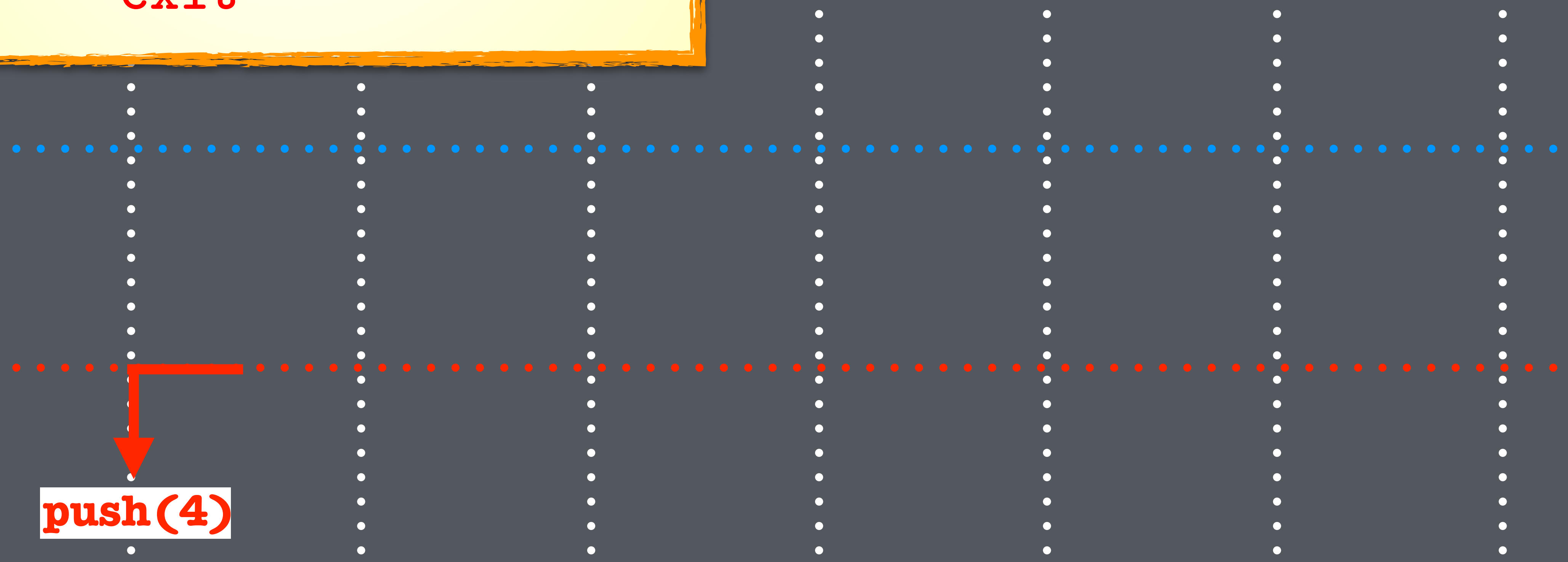


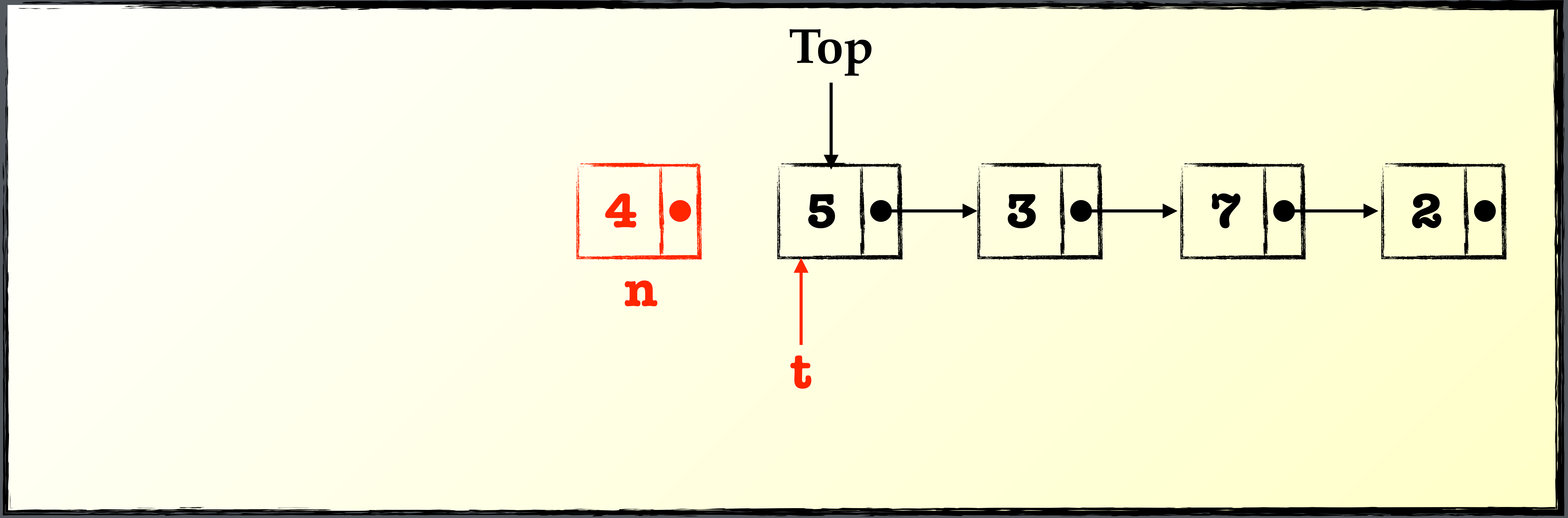
The Treiber Stack Algorithm

Linearization Policy

```
push(k):
Node t
   n = new Node(k,-)
   while (true)
     t = Top
     n.next = t
     if (CAS (Top,t,n))
       exit
```

push(4)

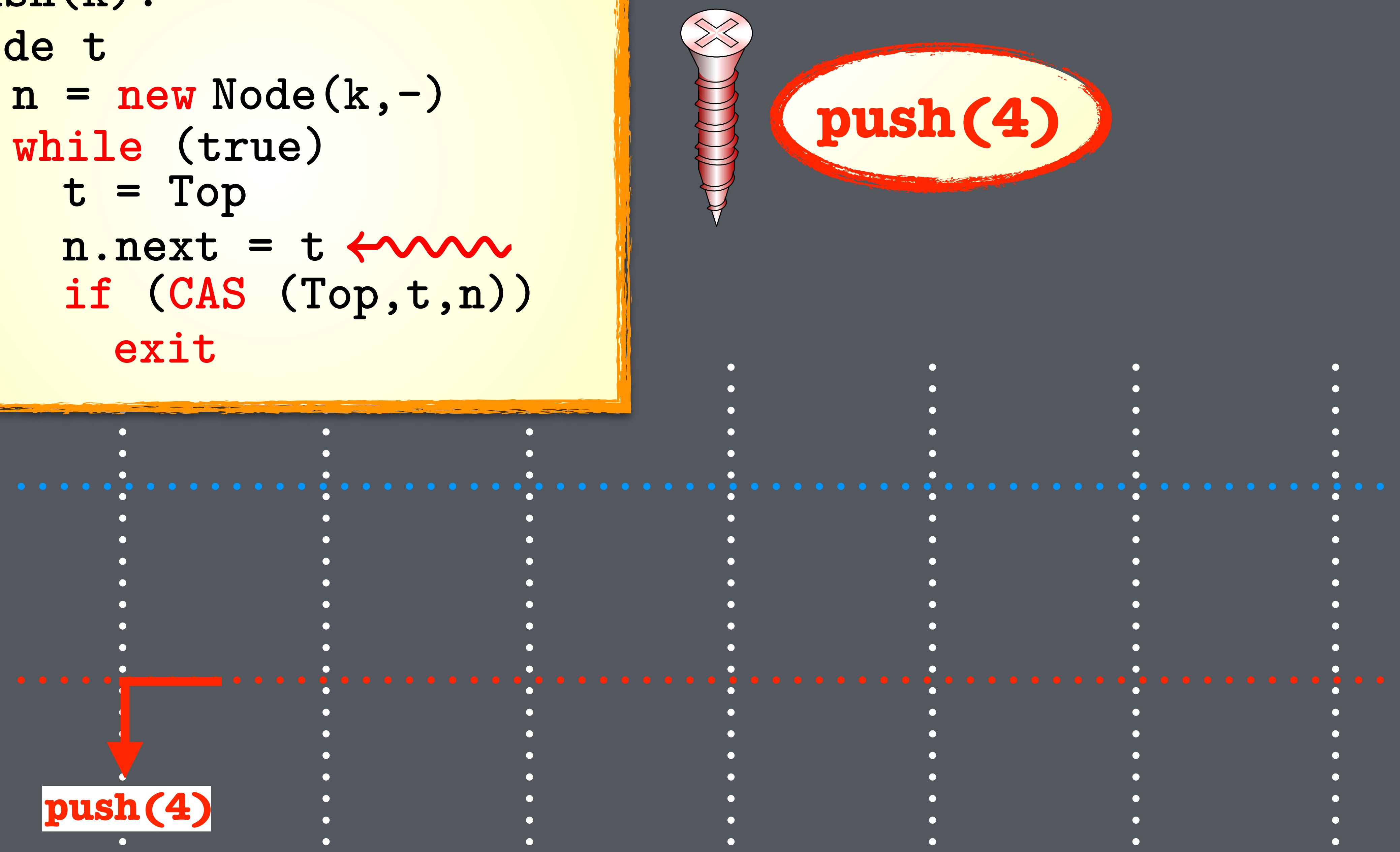


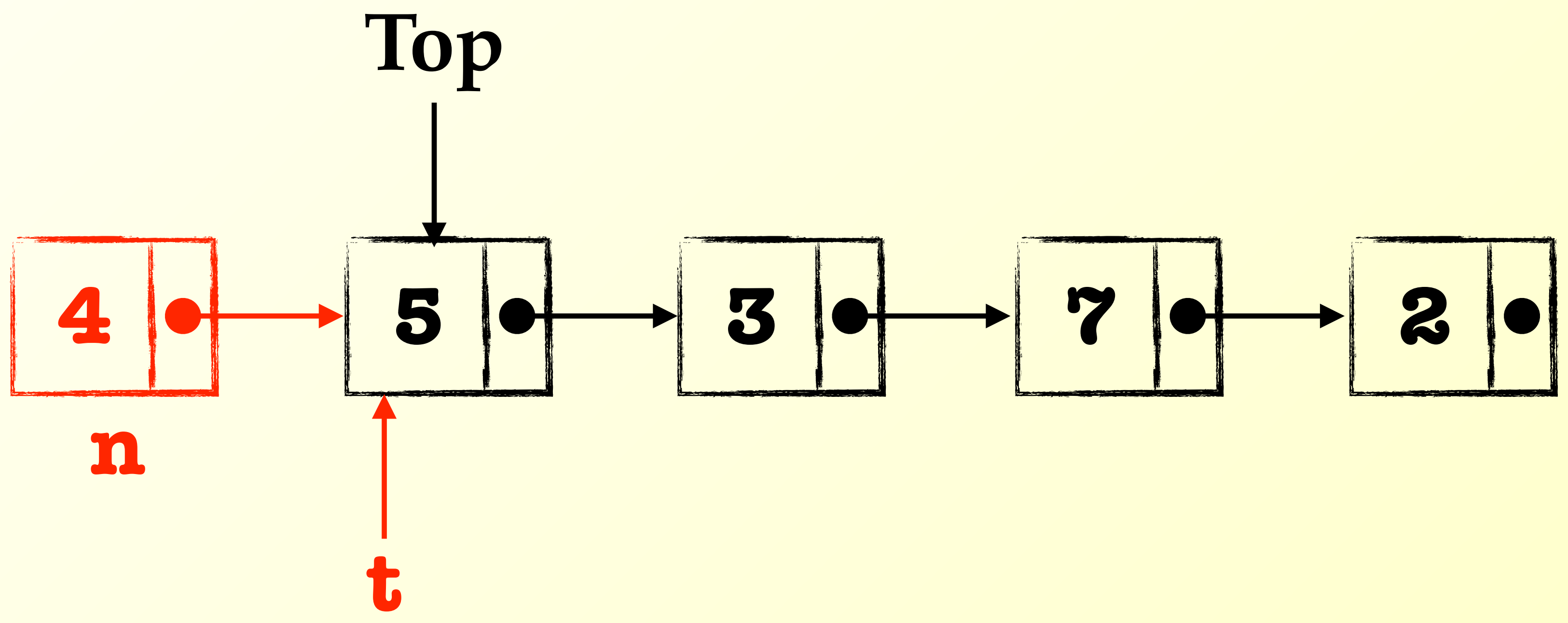


The Treiber Stack Algorithm

Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

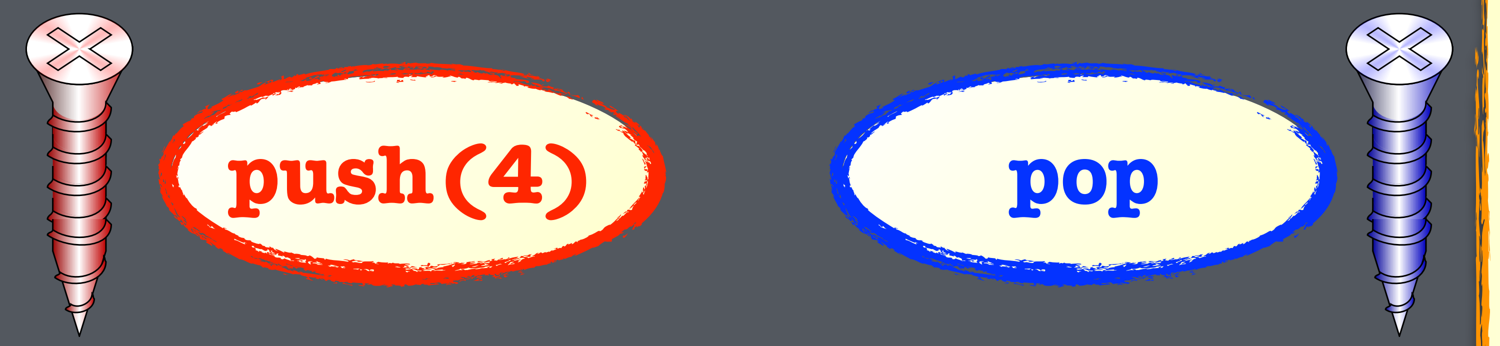




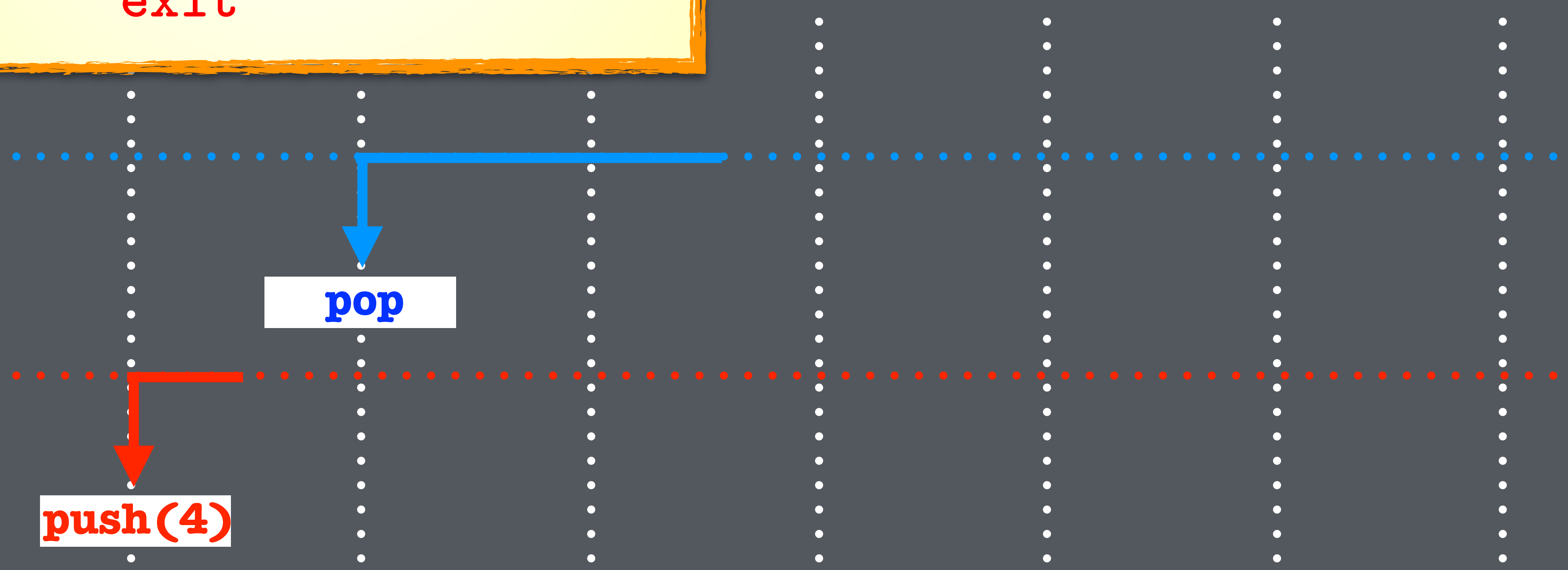
The Treiber Stack Algorithm

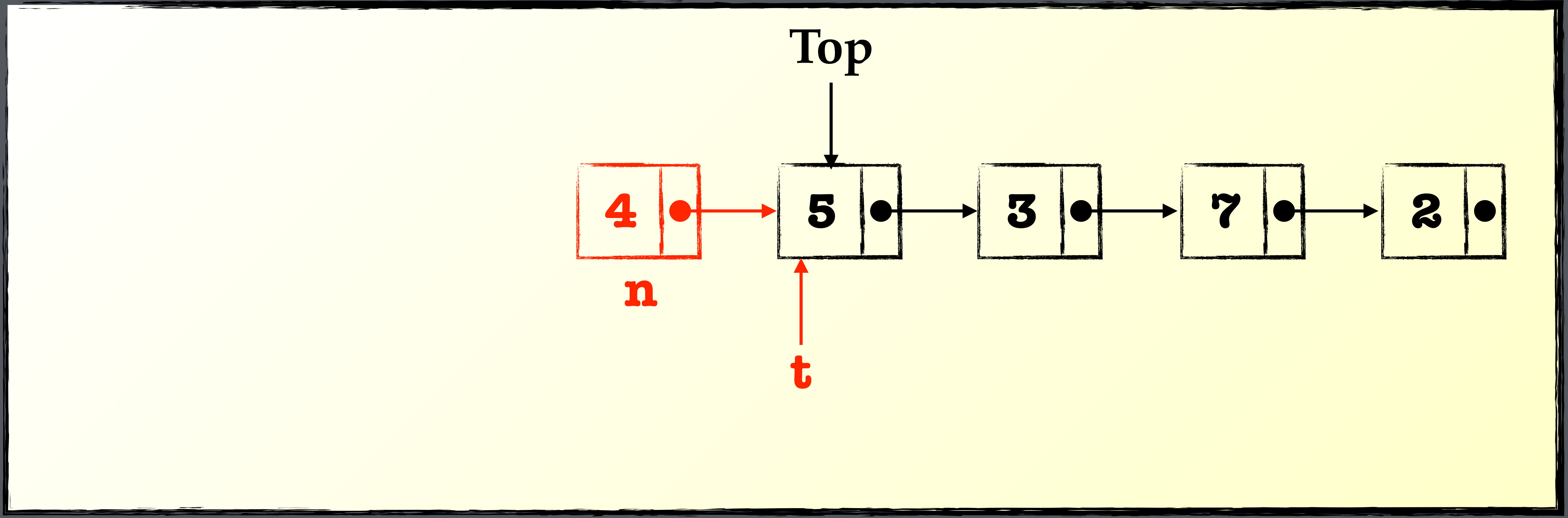
Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```




```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```





The Treiber Stack Algorithm

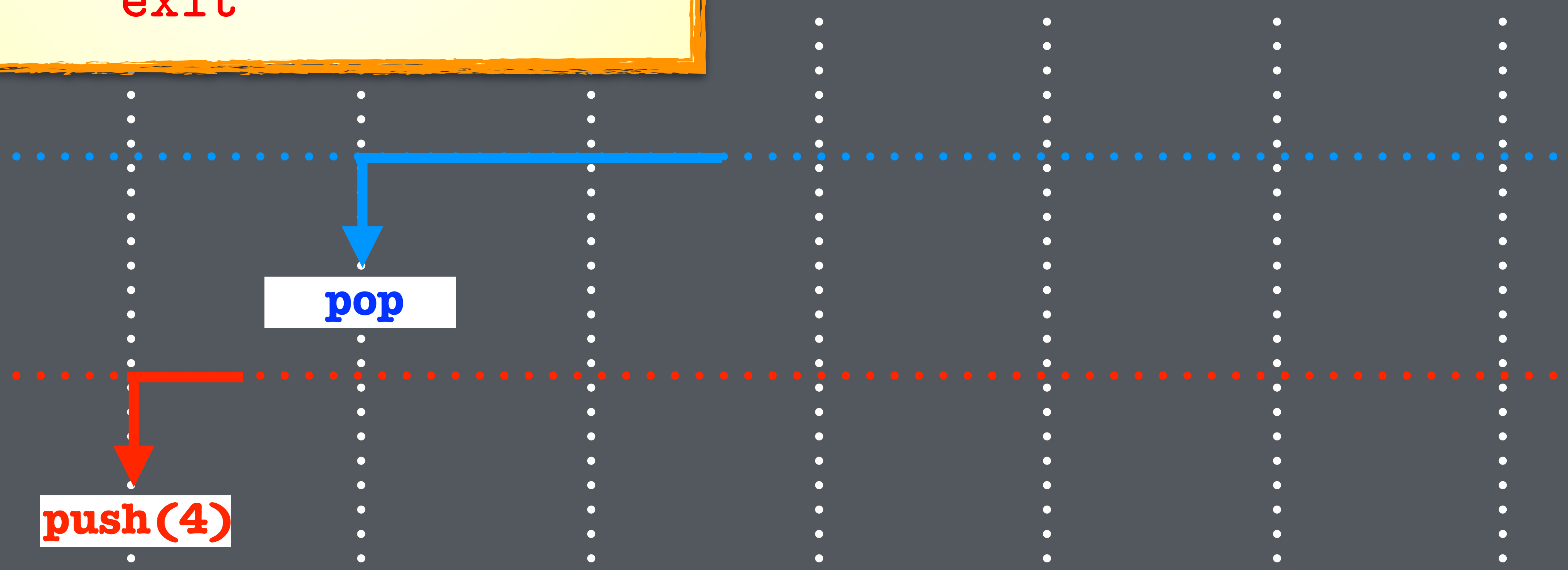
Linearization Policy

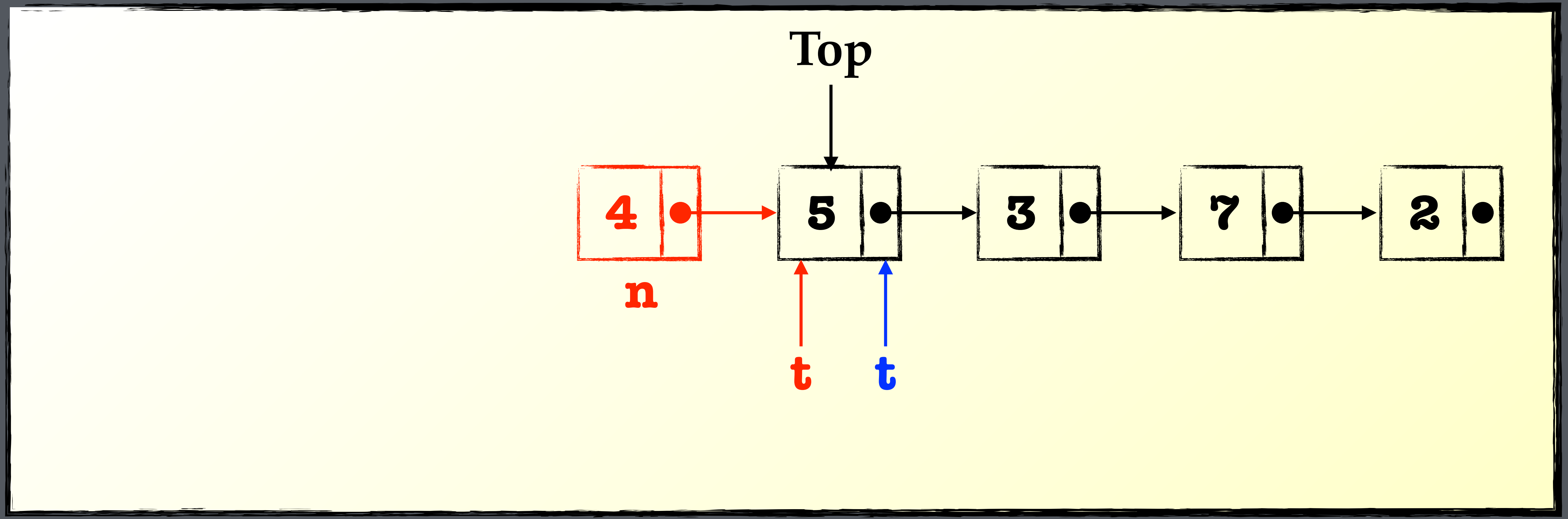
```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

push(4)

pop

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

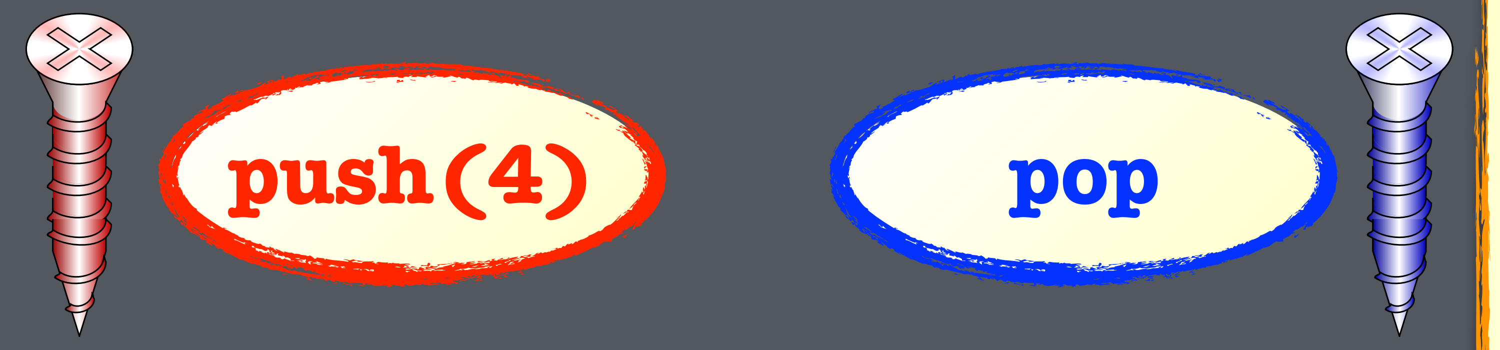




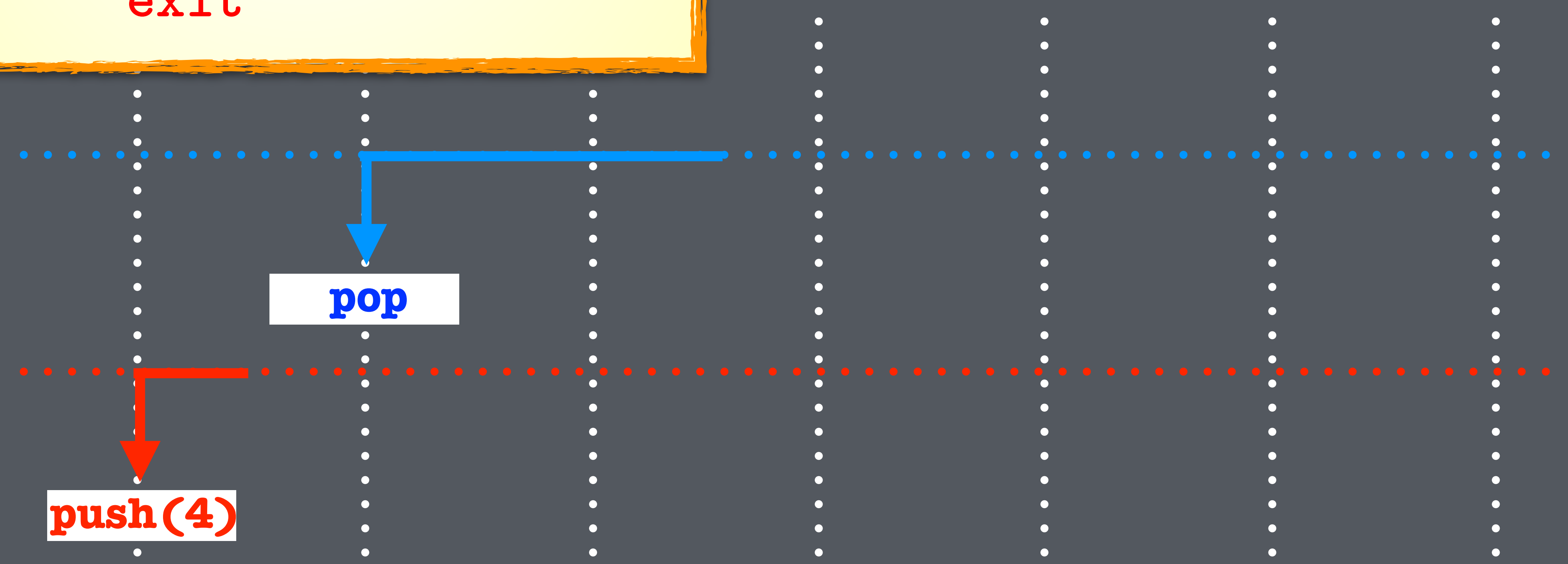
The Treiber Stack Algorithm

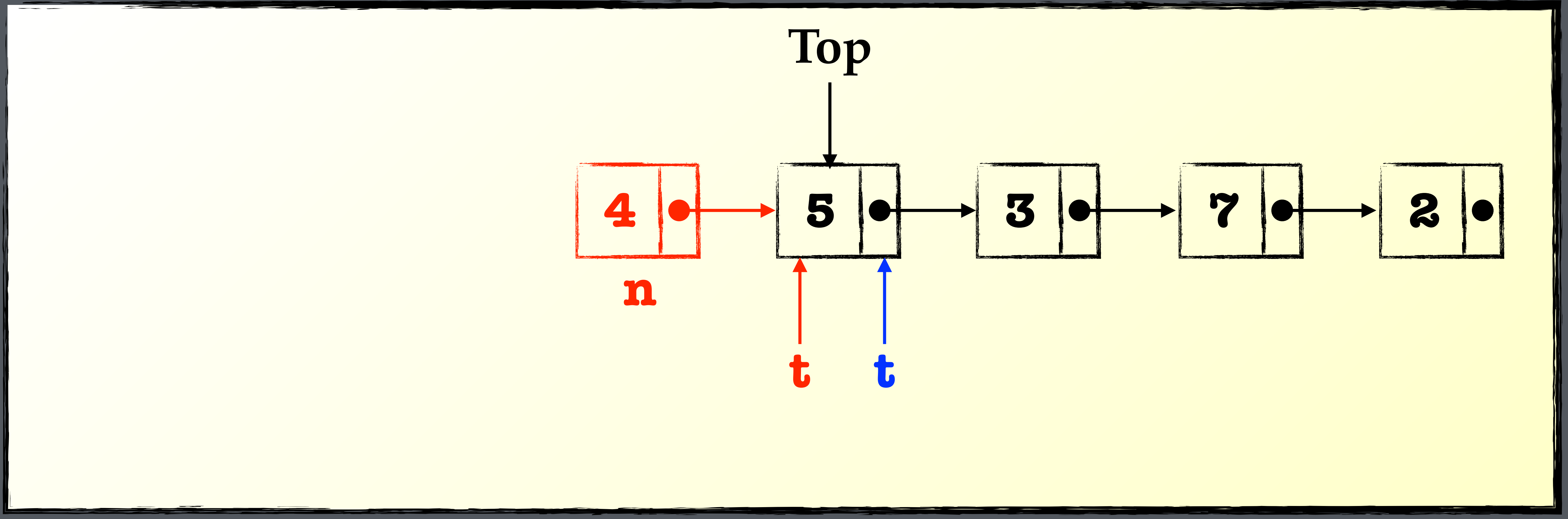
Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```




```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

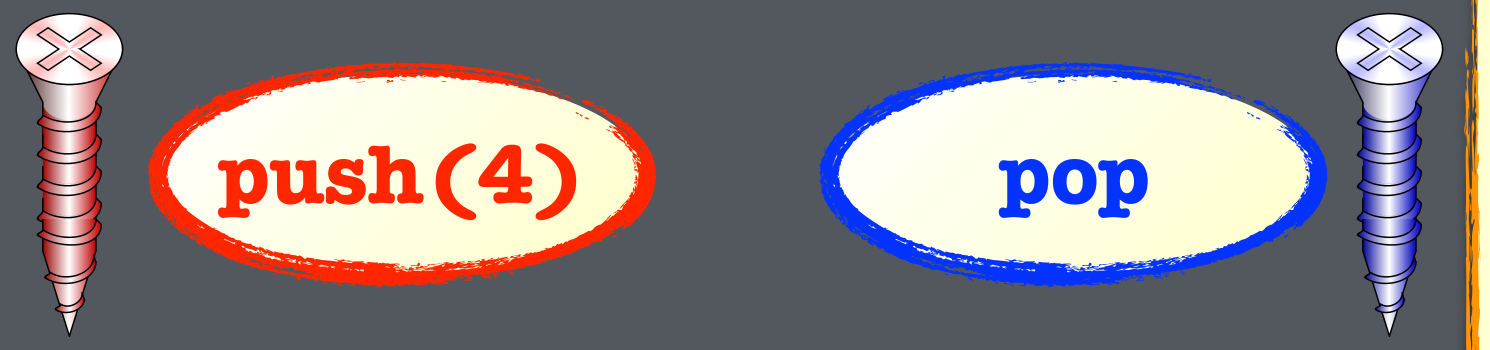




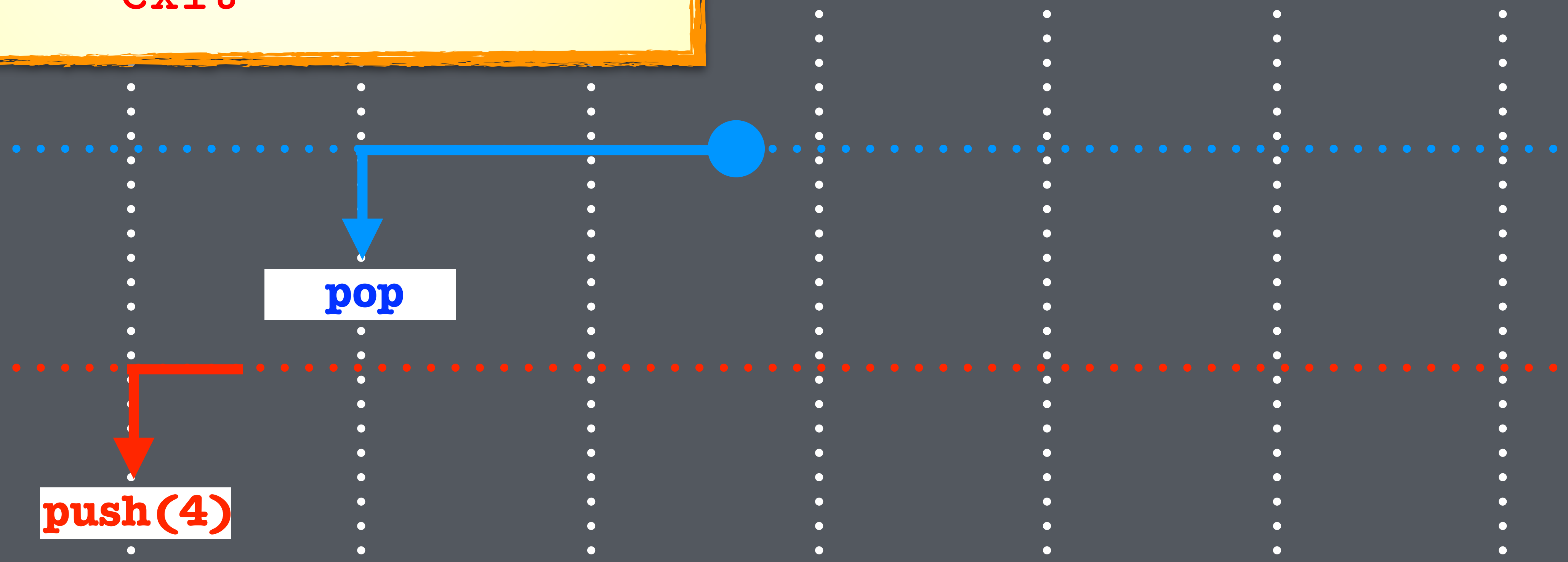
The Treiber Stack Algorithm

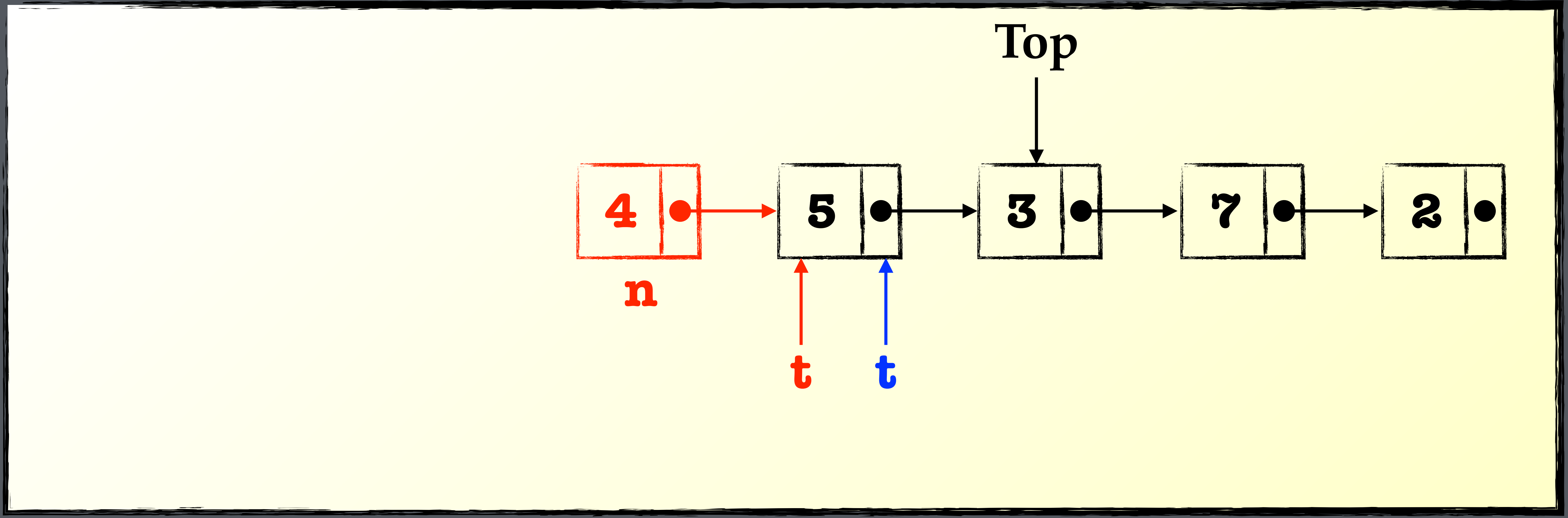
Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```




```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

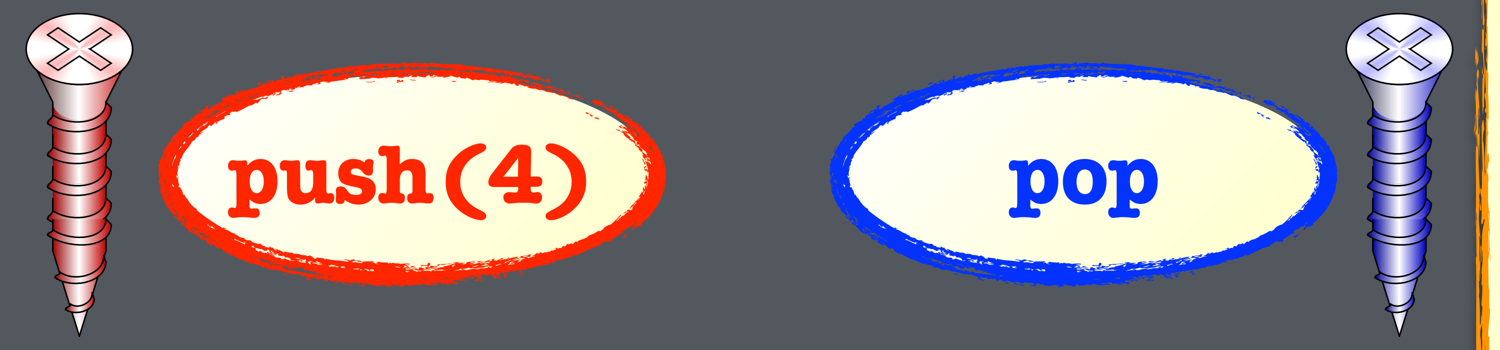




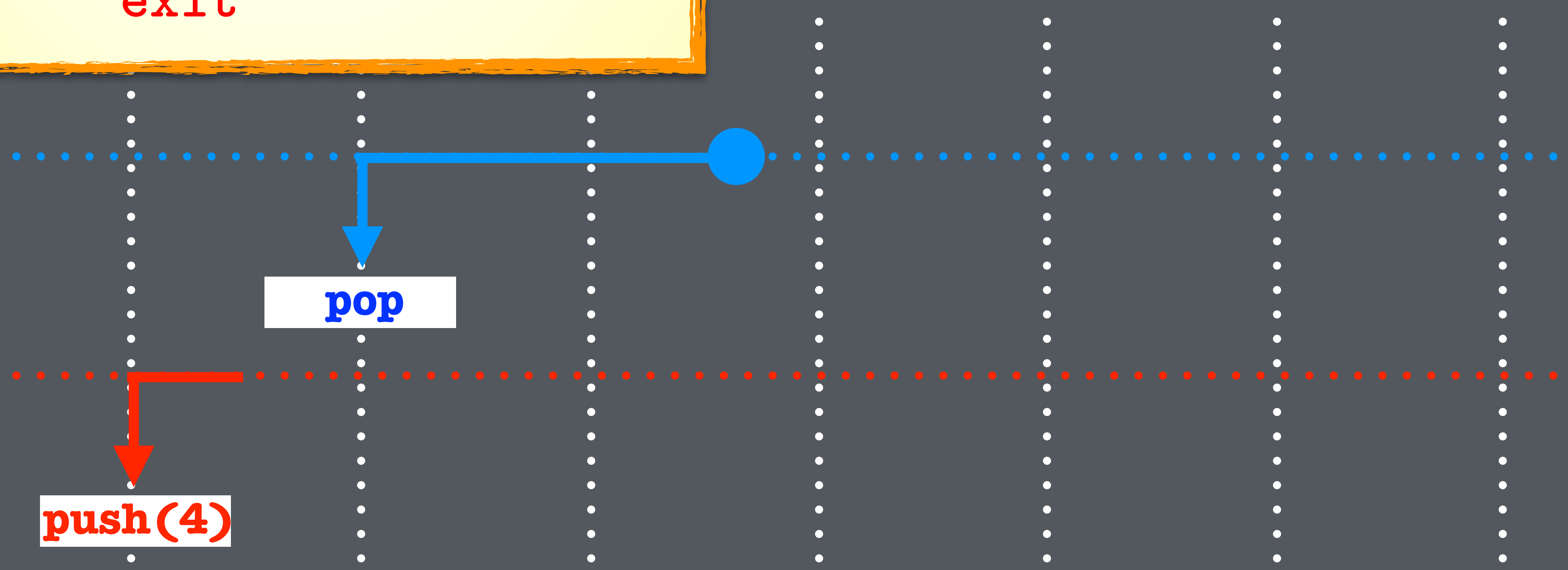
The Treiber Stack Algorithm

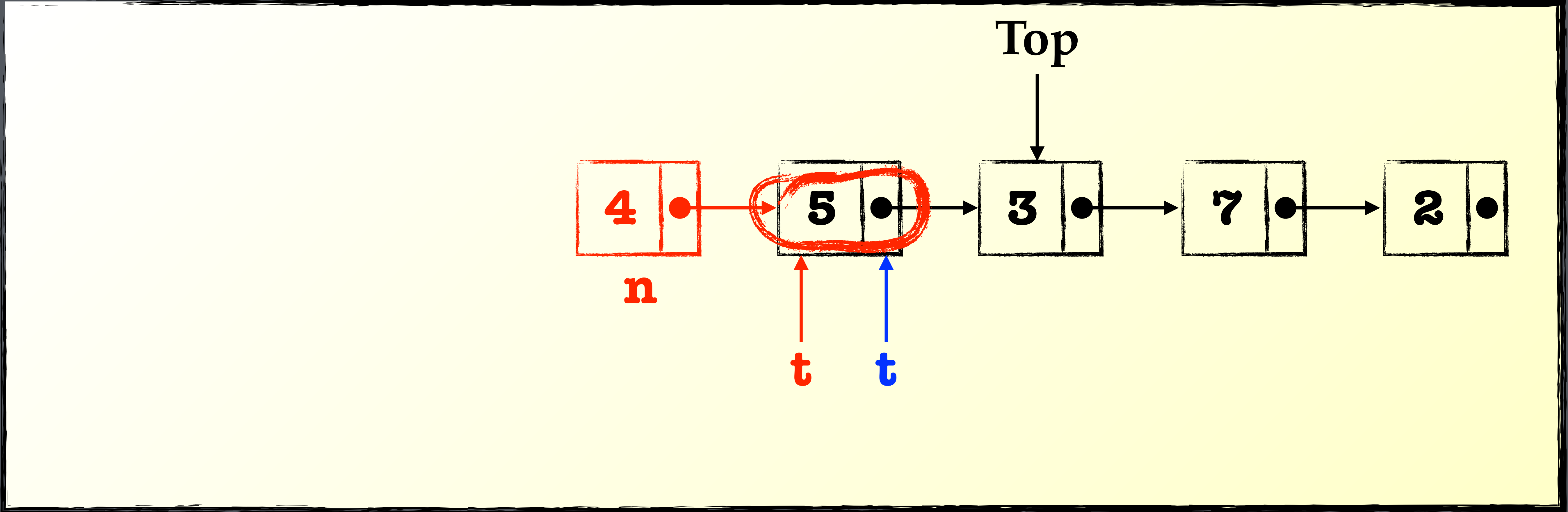
Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```




```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

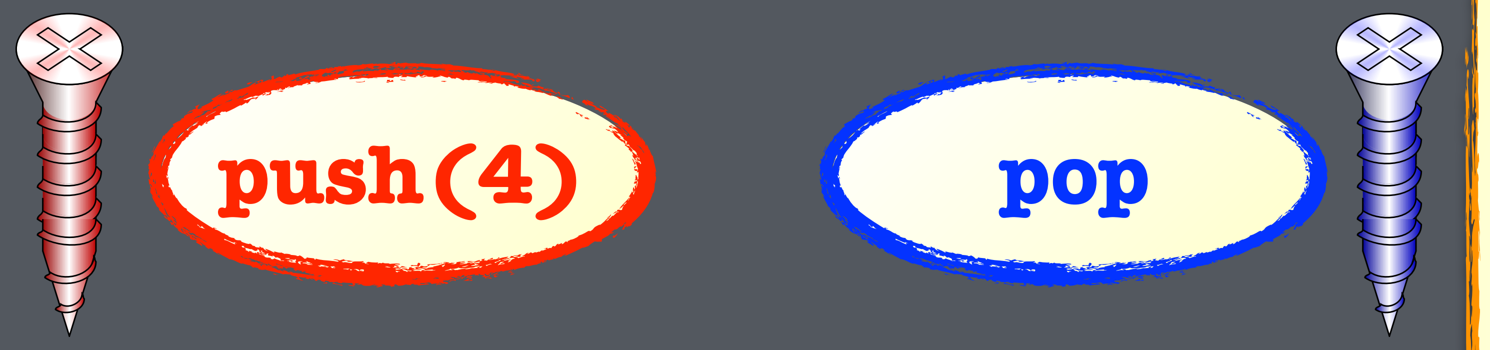




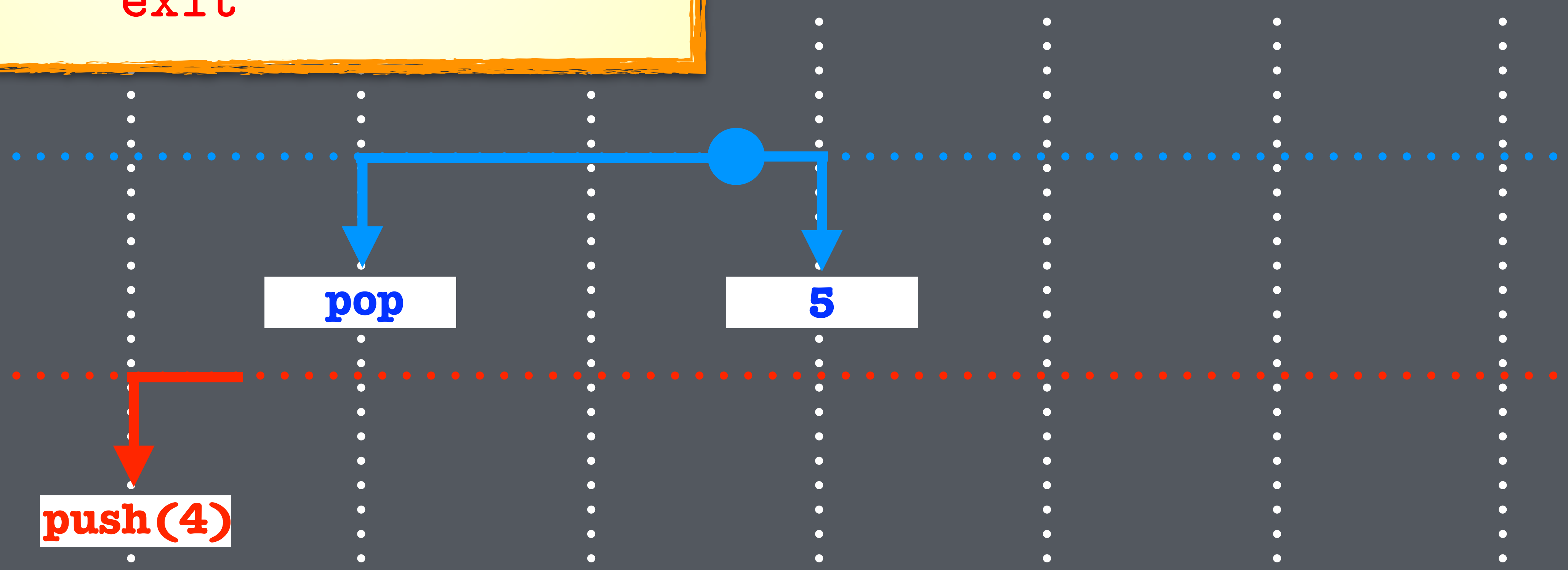
The Treiber Stack Algorithm

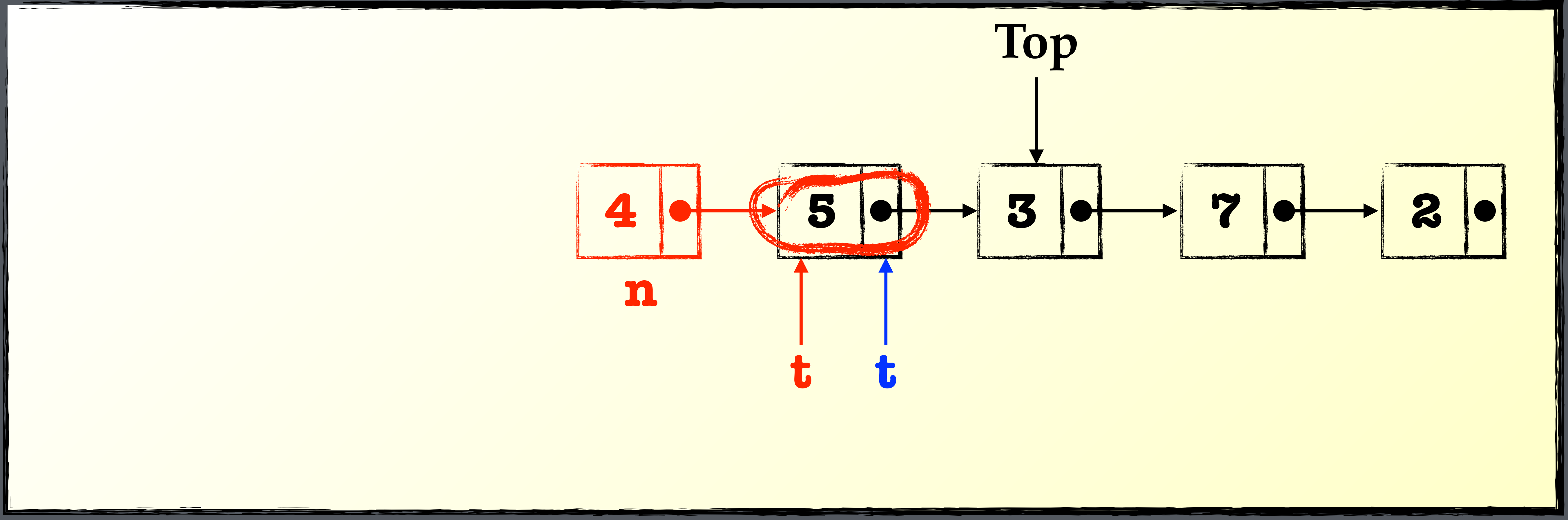
Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```




```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```



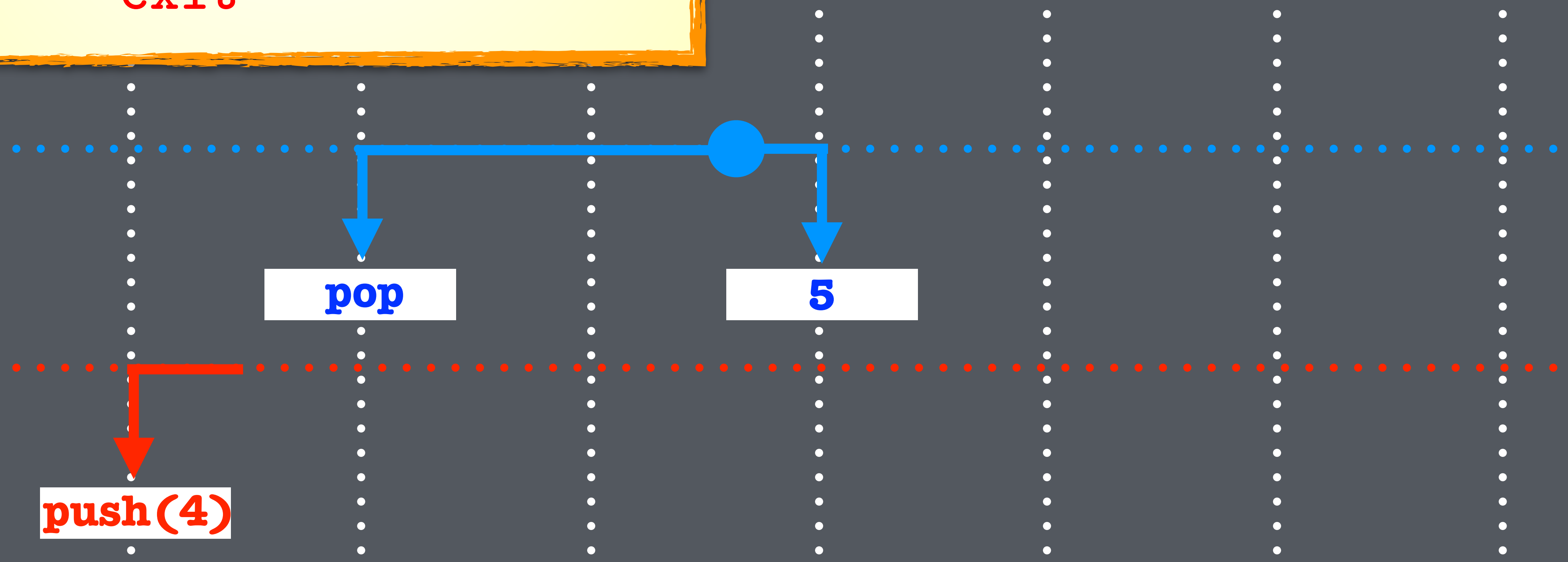


The Treiber Stack Algorithm

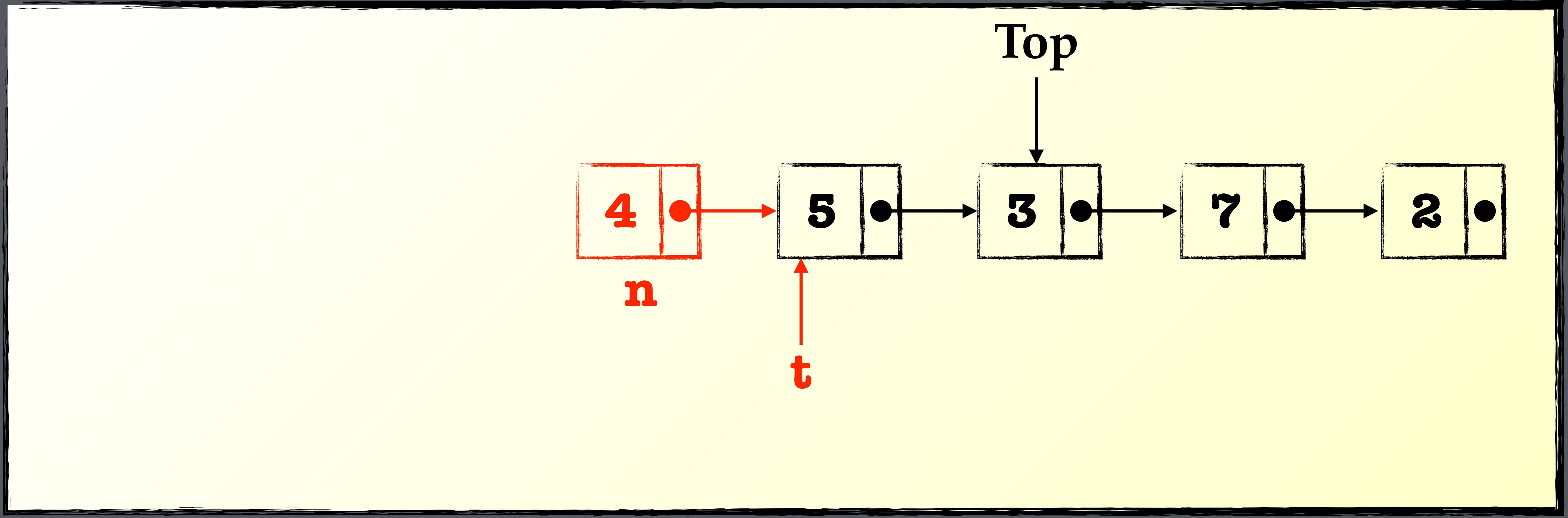
Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

push(4)



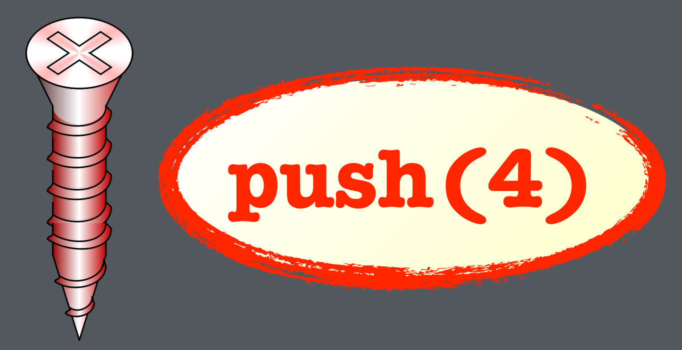

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```



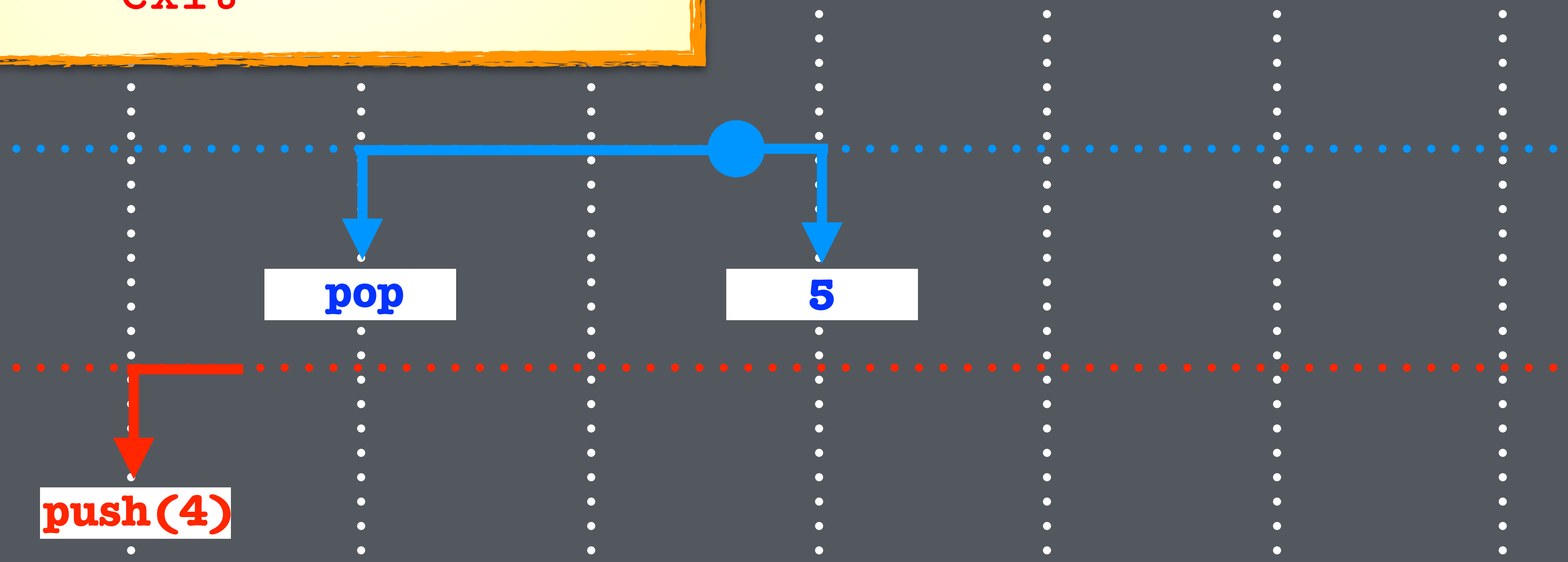
The Treiber Stack Algorithm

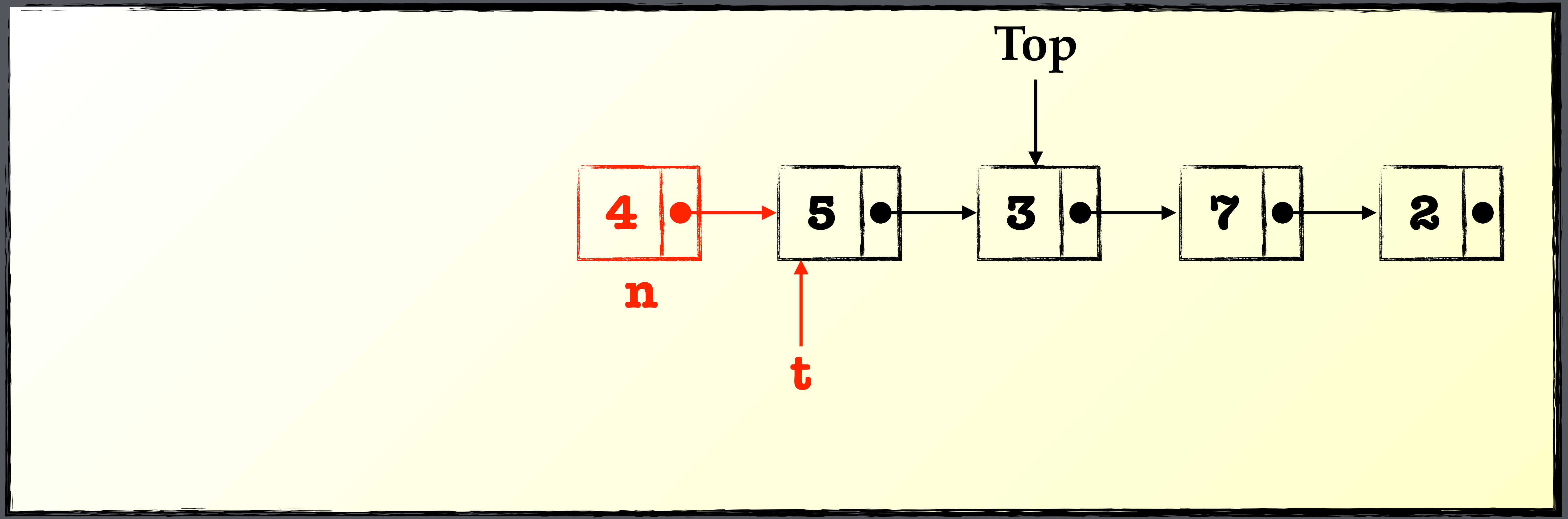
Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```




```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

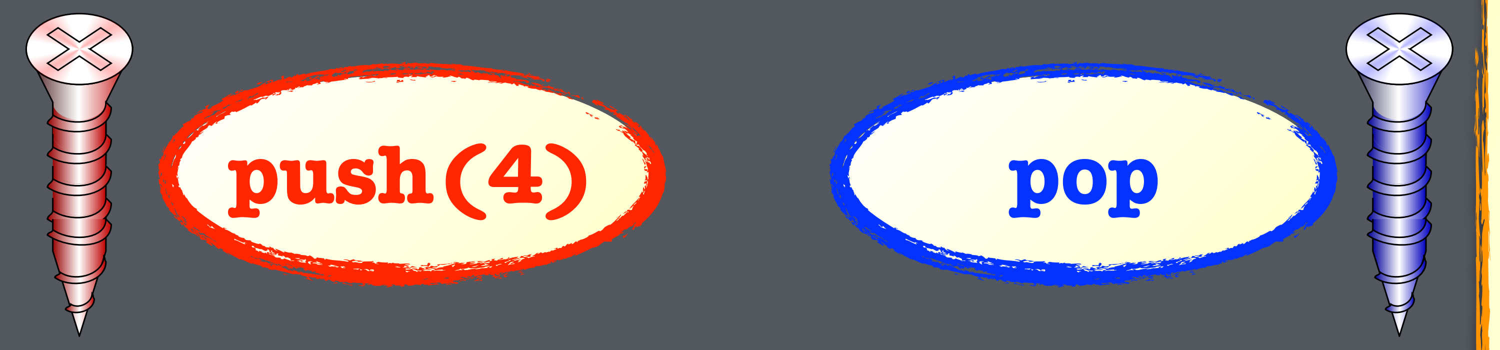




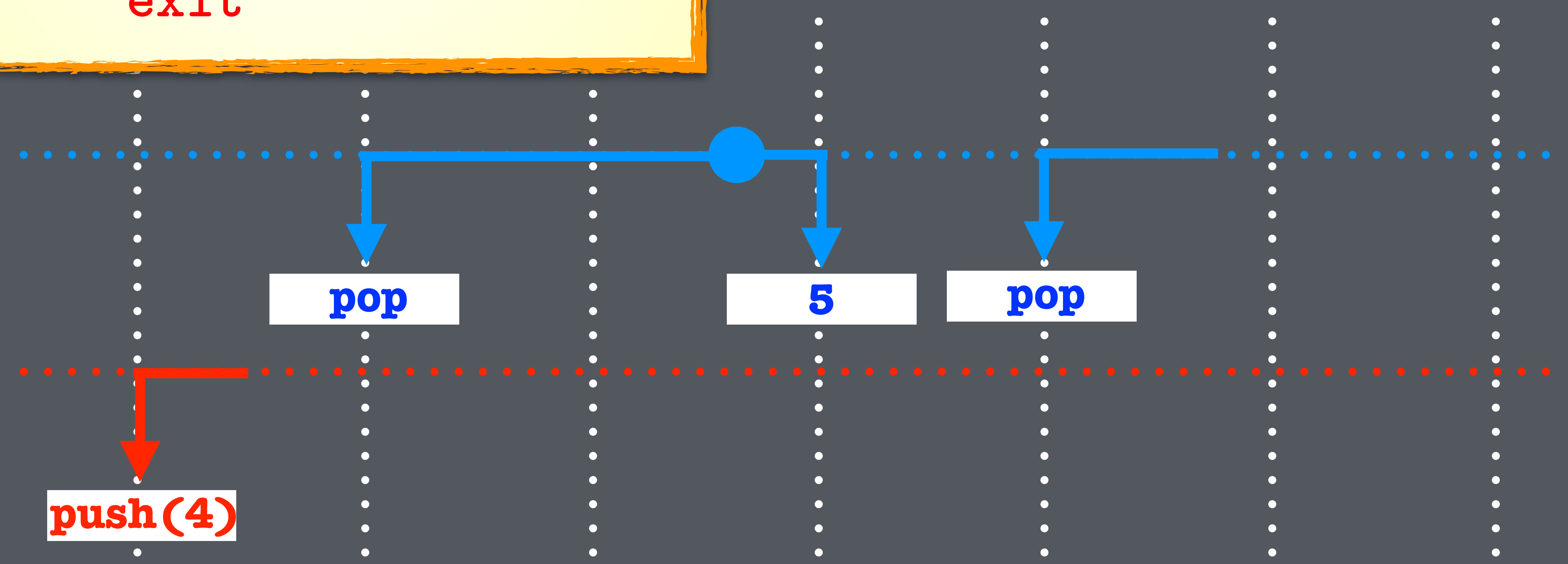
The Treiber Stack Algorithm

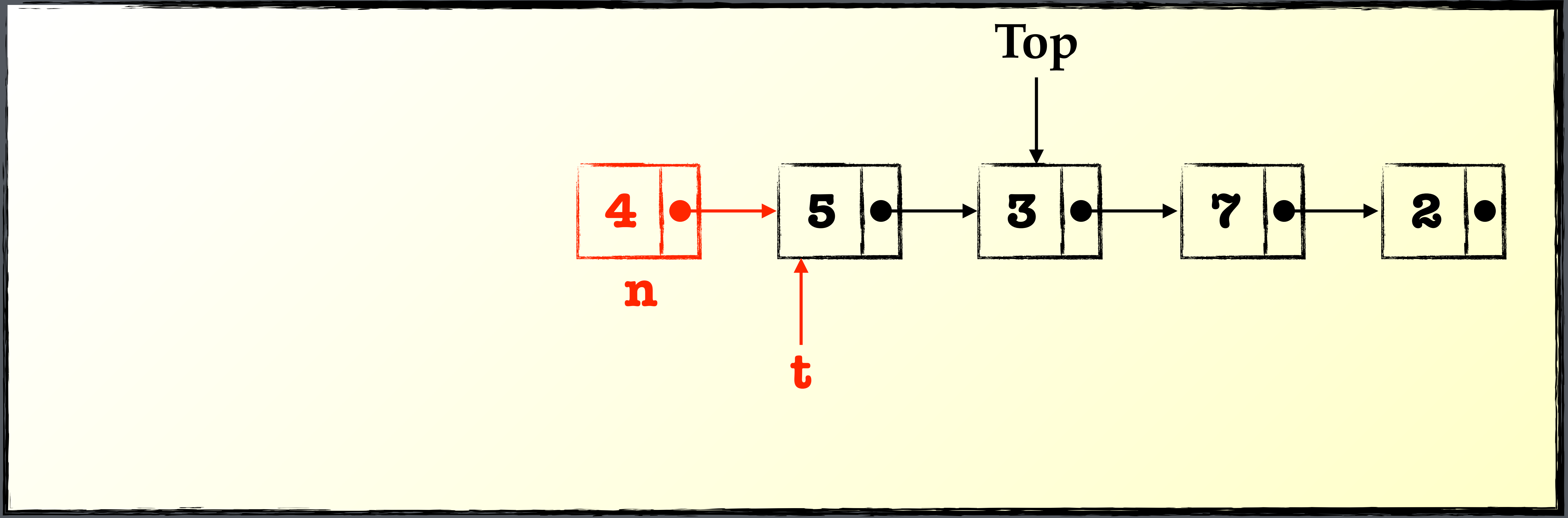
Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```




```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

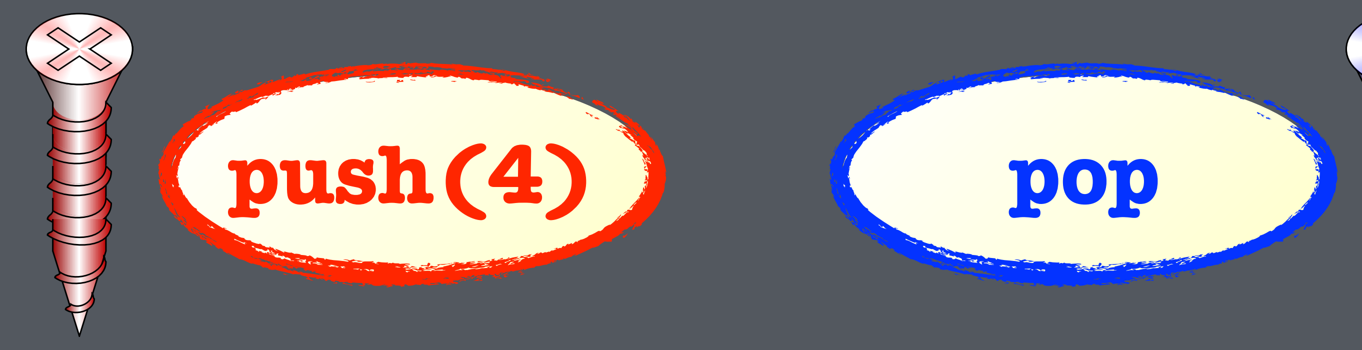




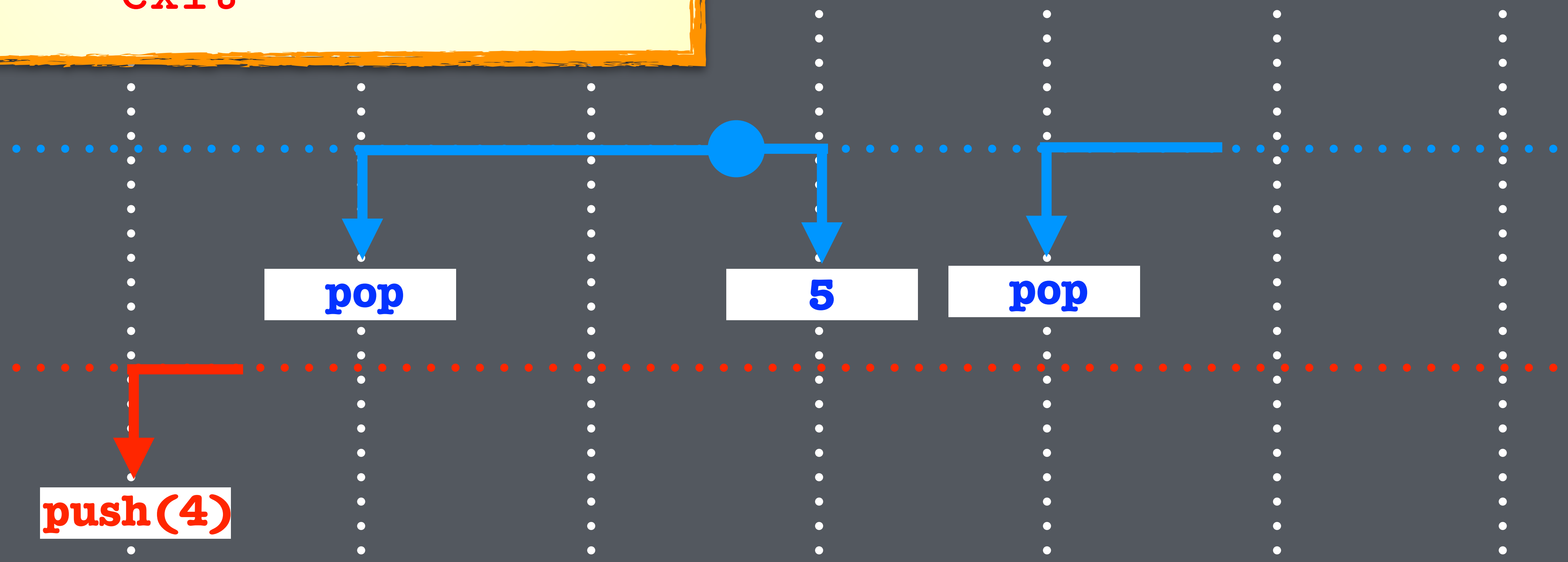
The Treiber Stack Algorithm

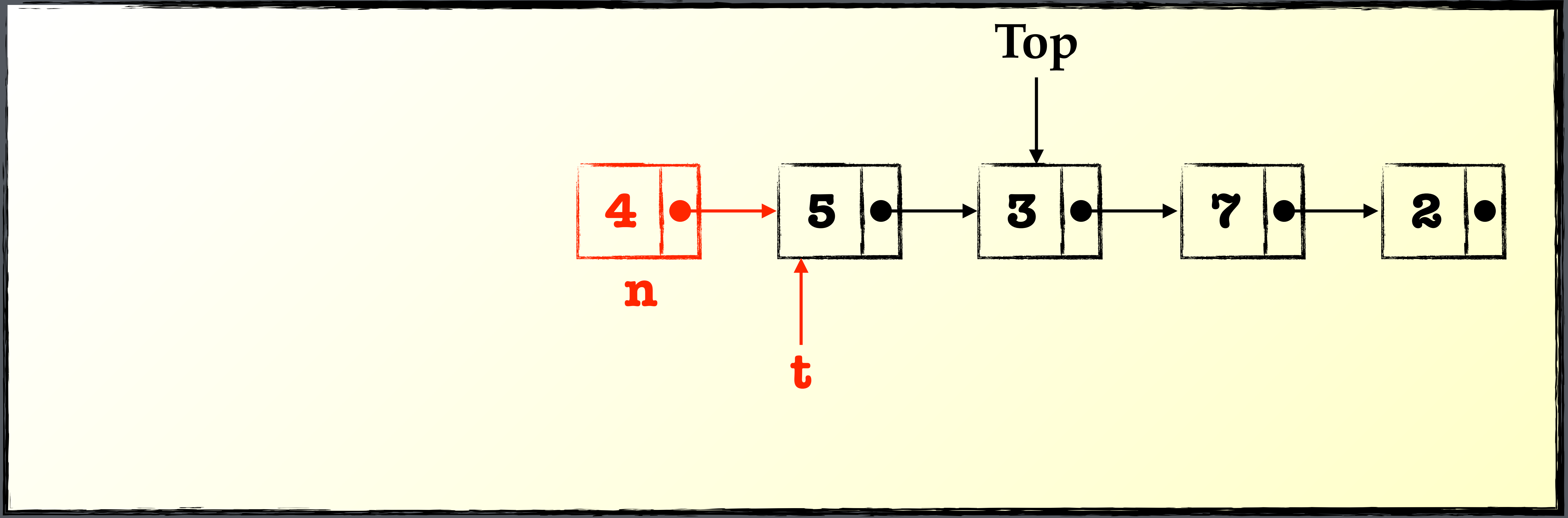
Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```




```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```





The Treiber Stack Algorithm

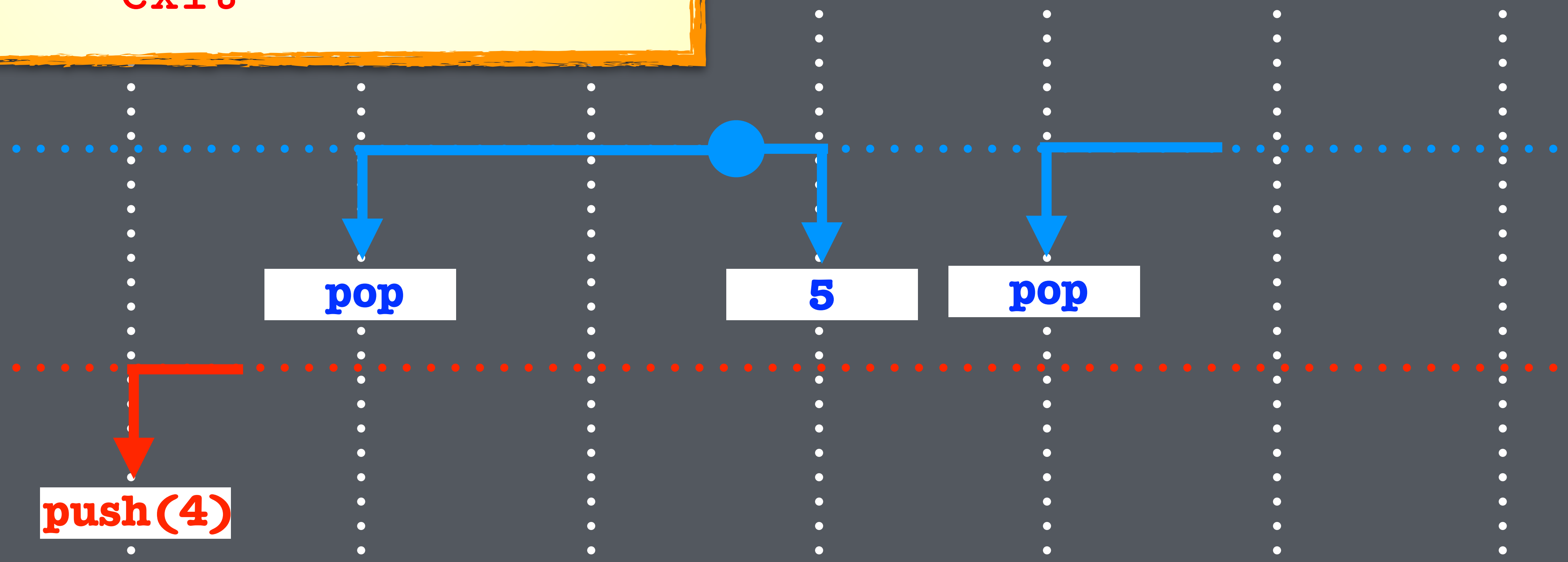
Linearization Policy

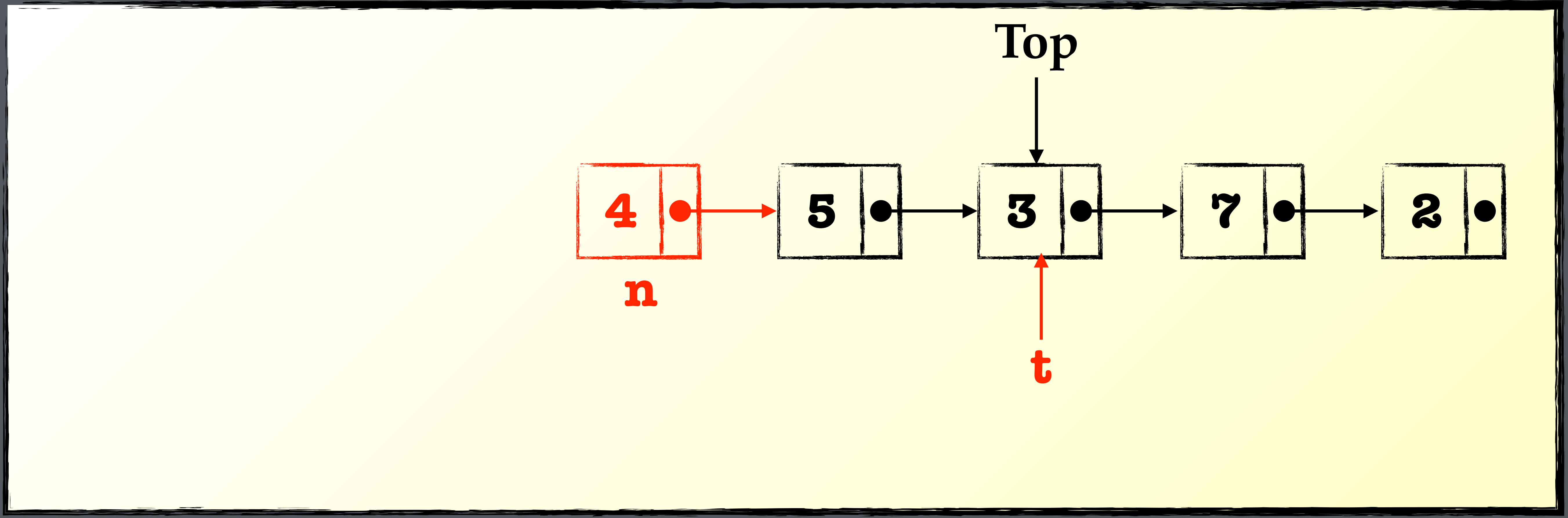
```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

push(4)

pop

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

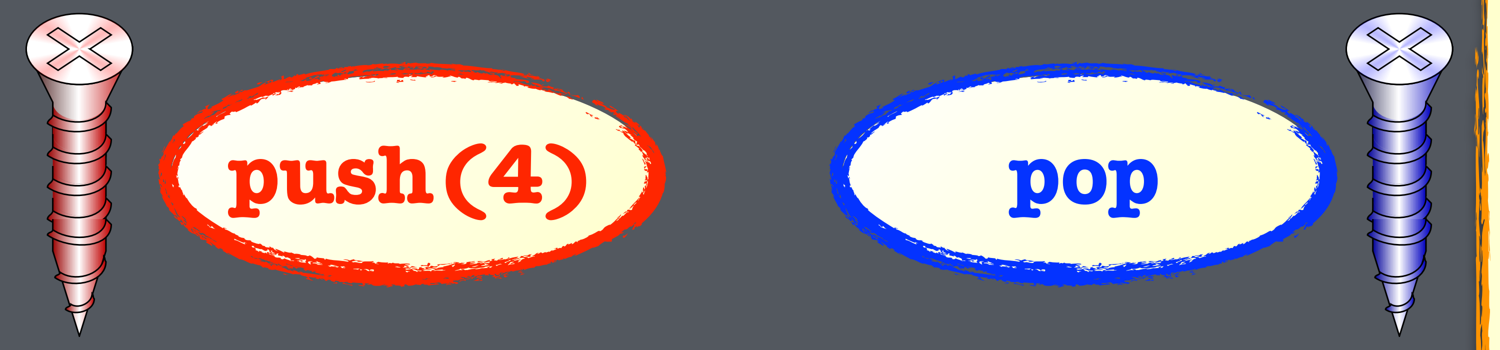




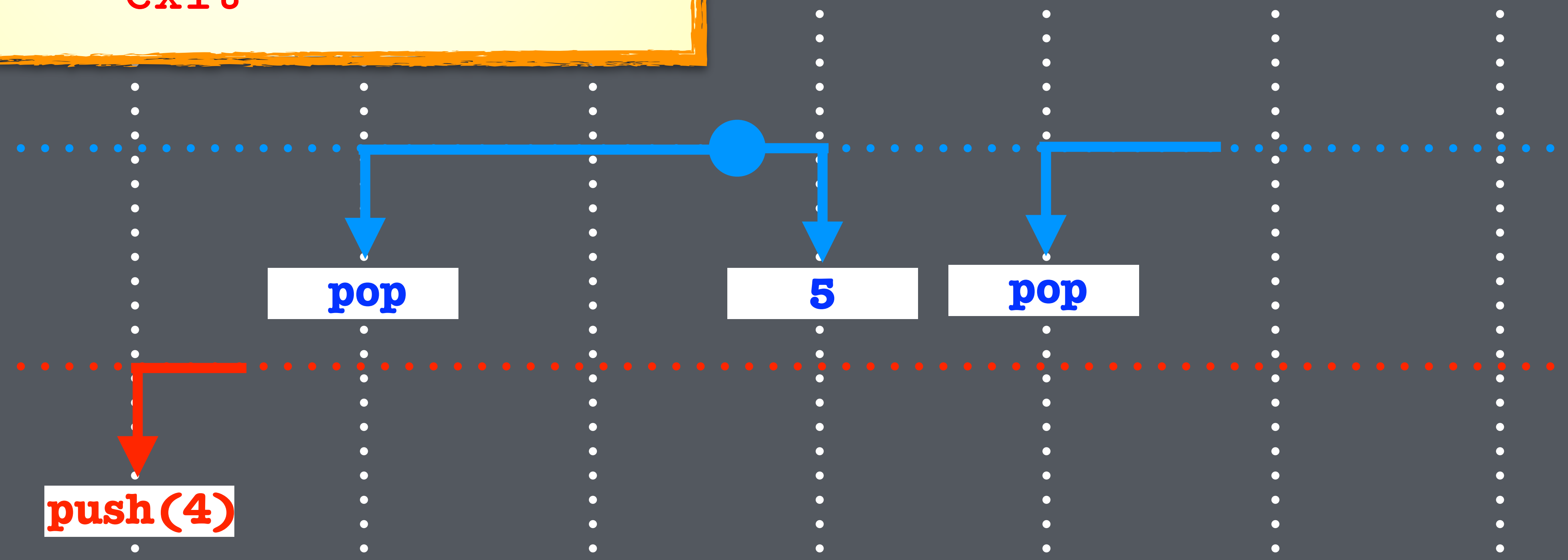
The Treiber Stack Algorithm

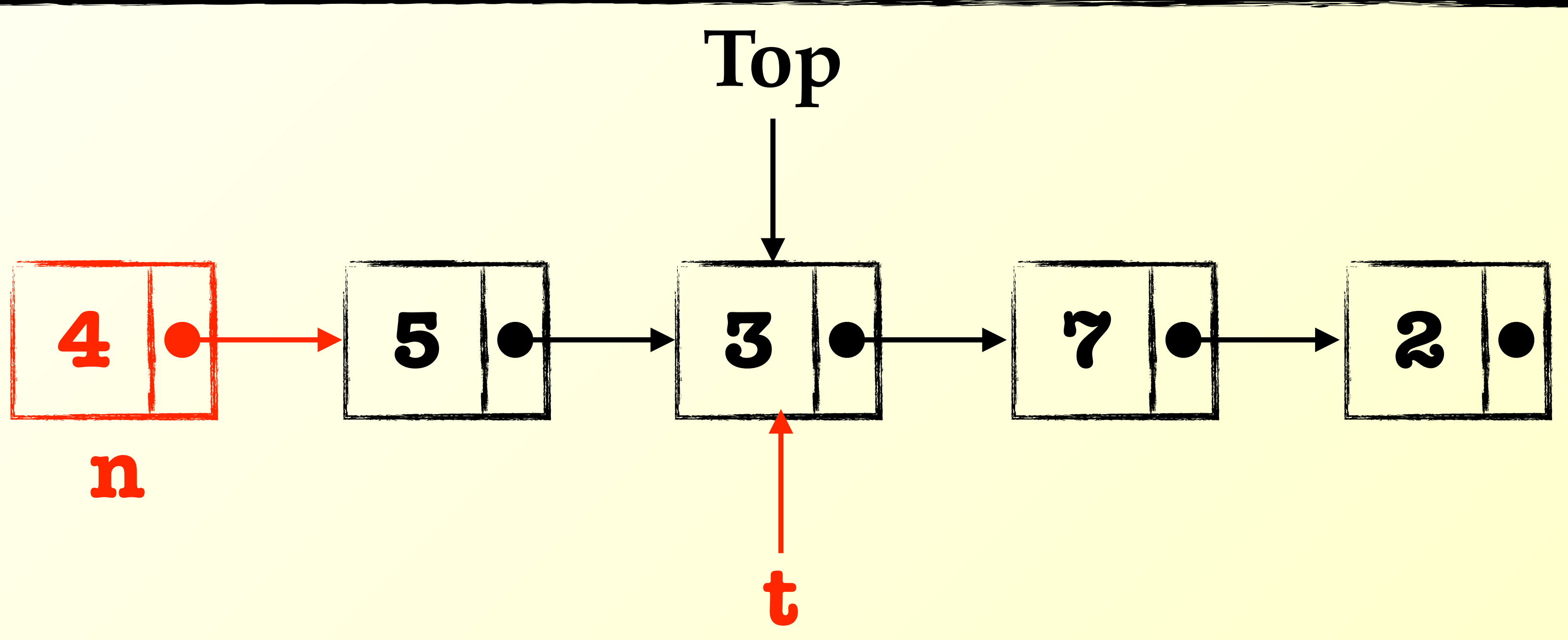
Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```




```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```





The Treiber Stack Algorithm

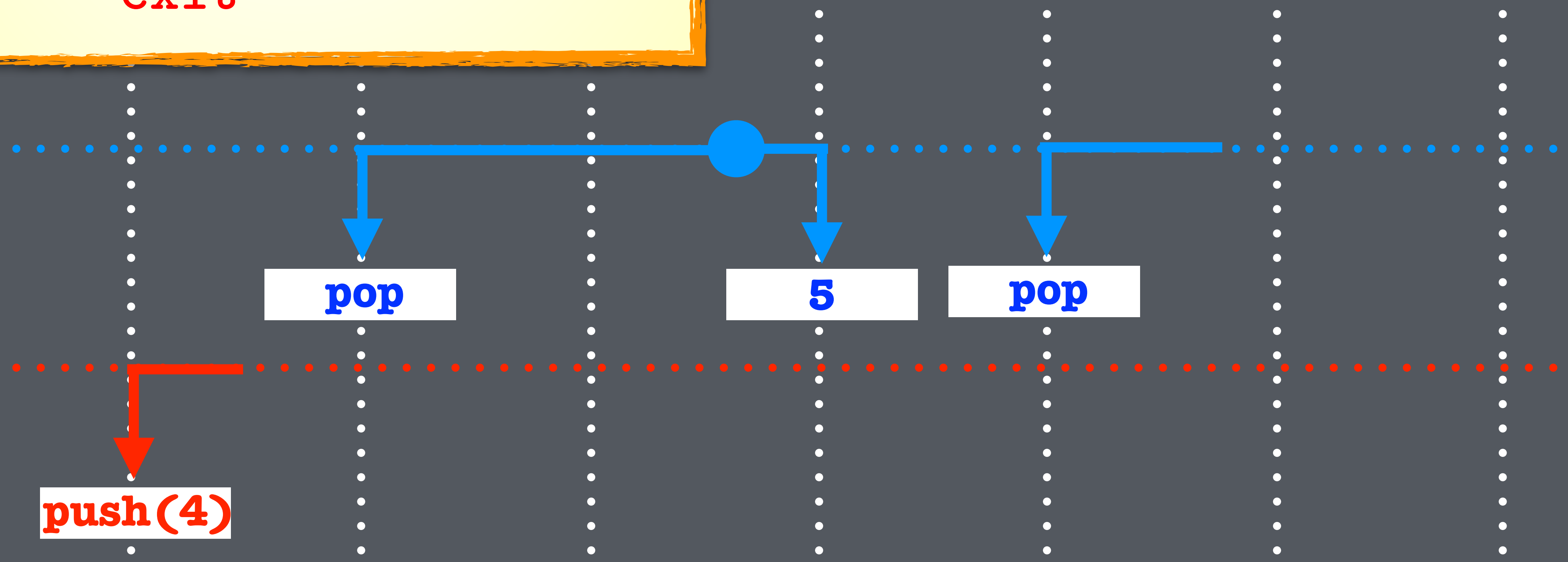
Linearization Policy

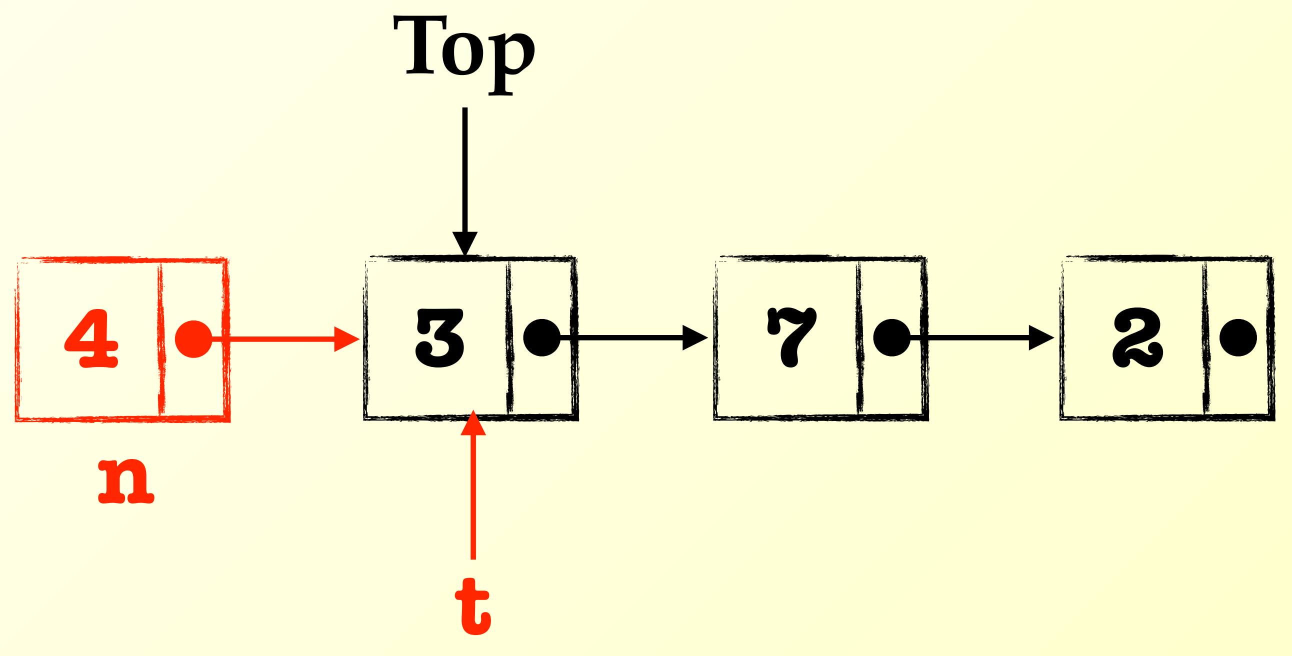
```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

push(4)

pop

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

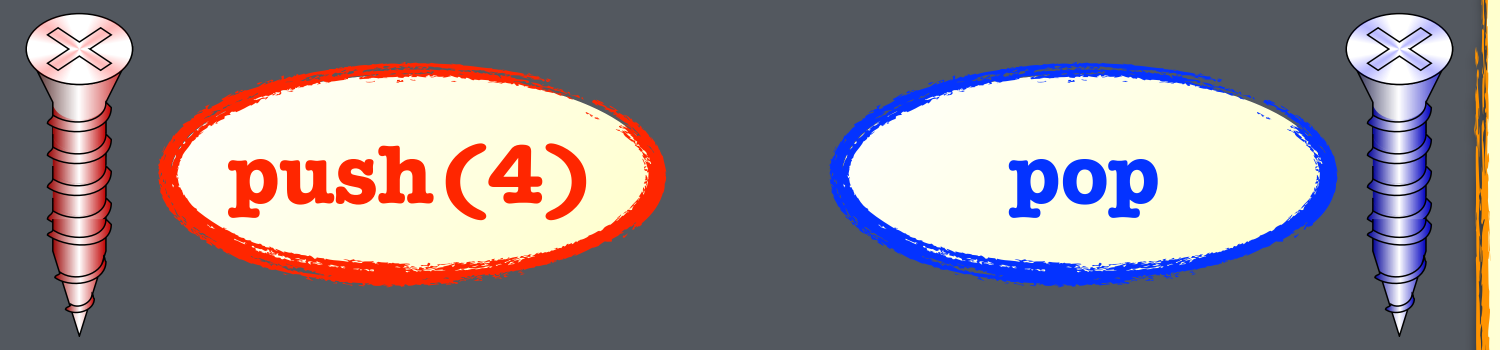




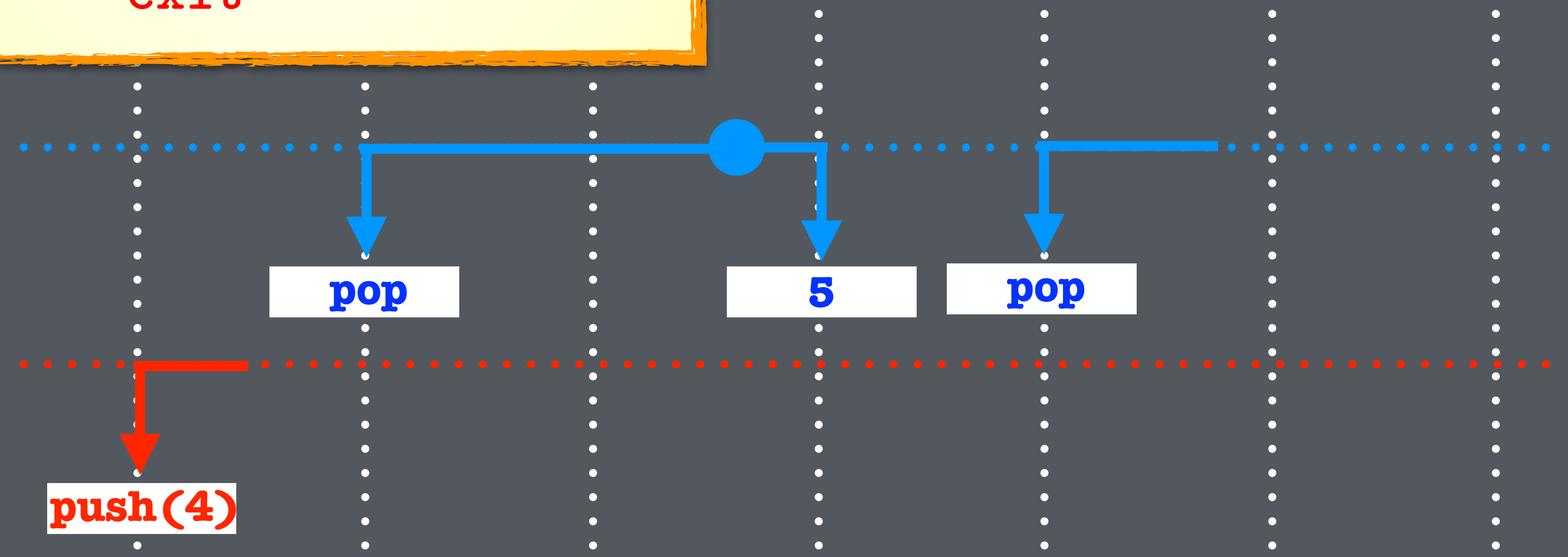
The Treiber Stack Algorithm

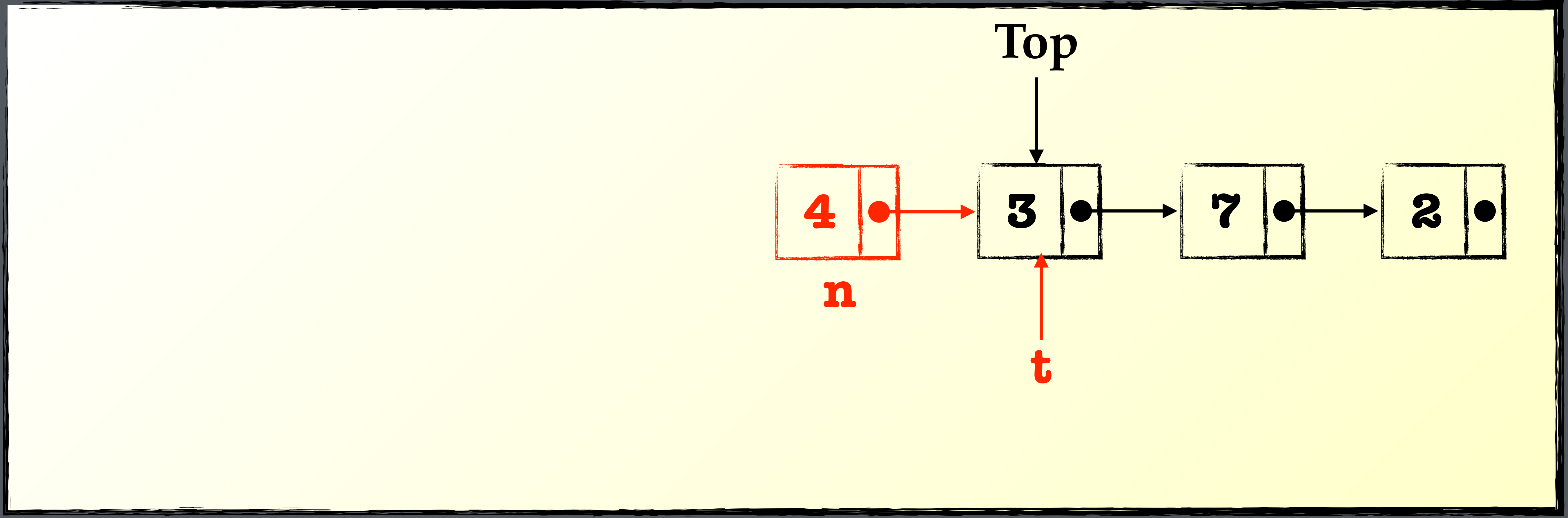
Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```




```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

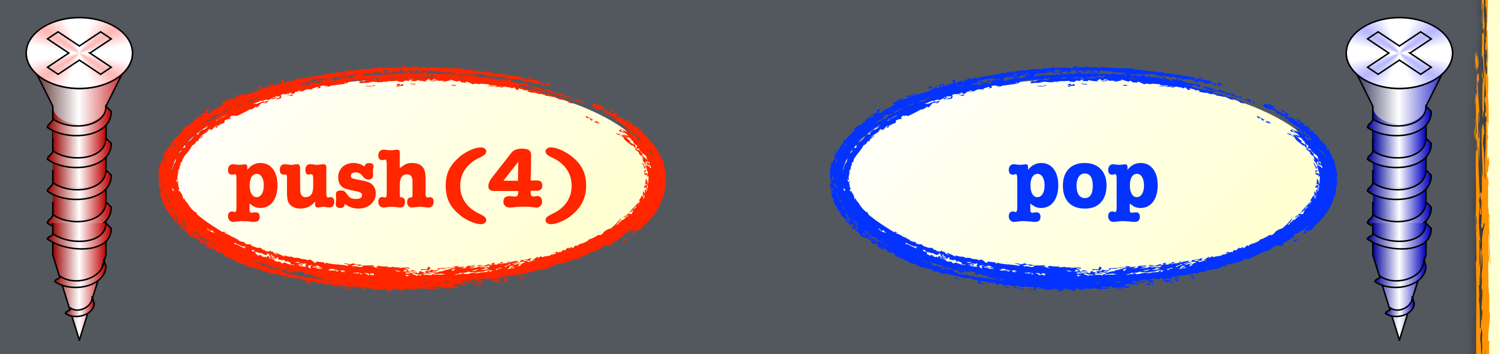




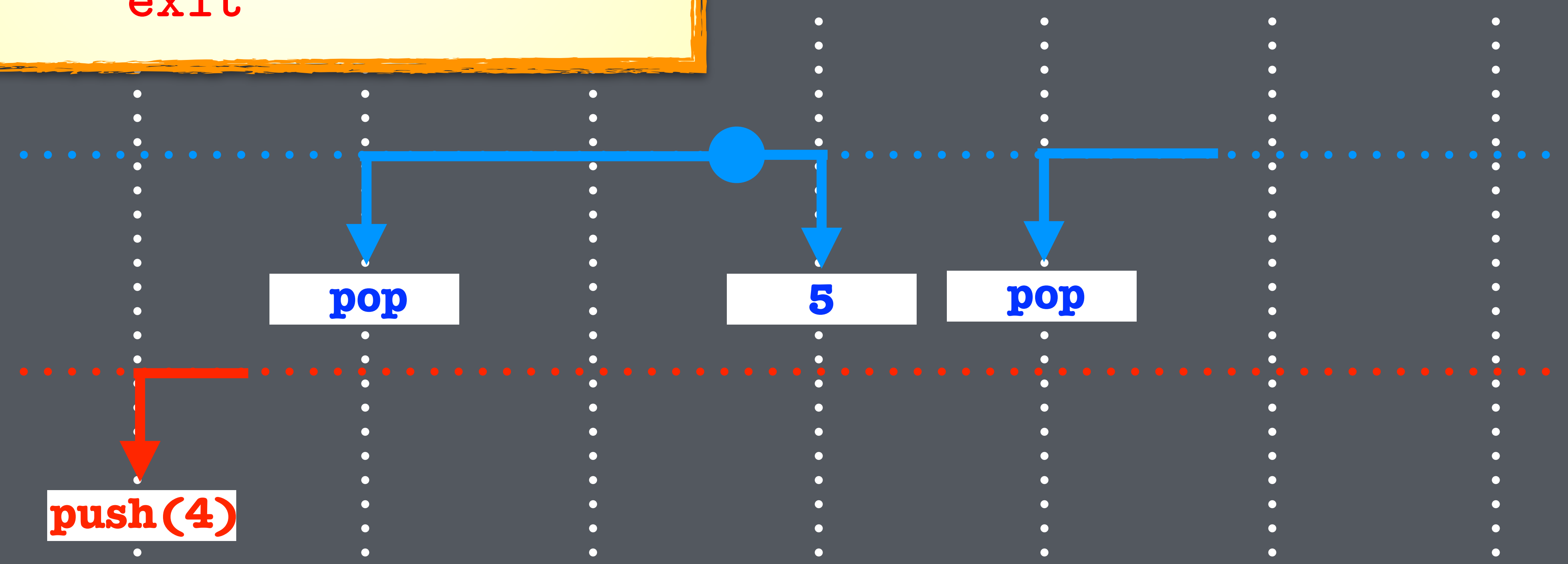
The Treiber Stack Algorithm

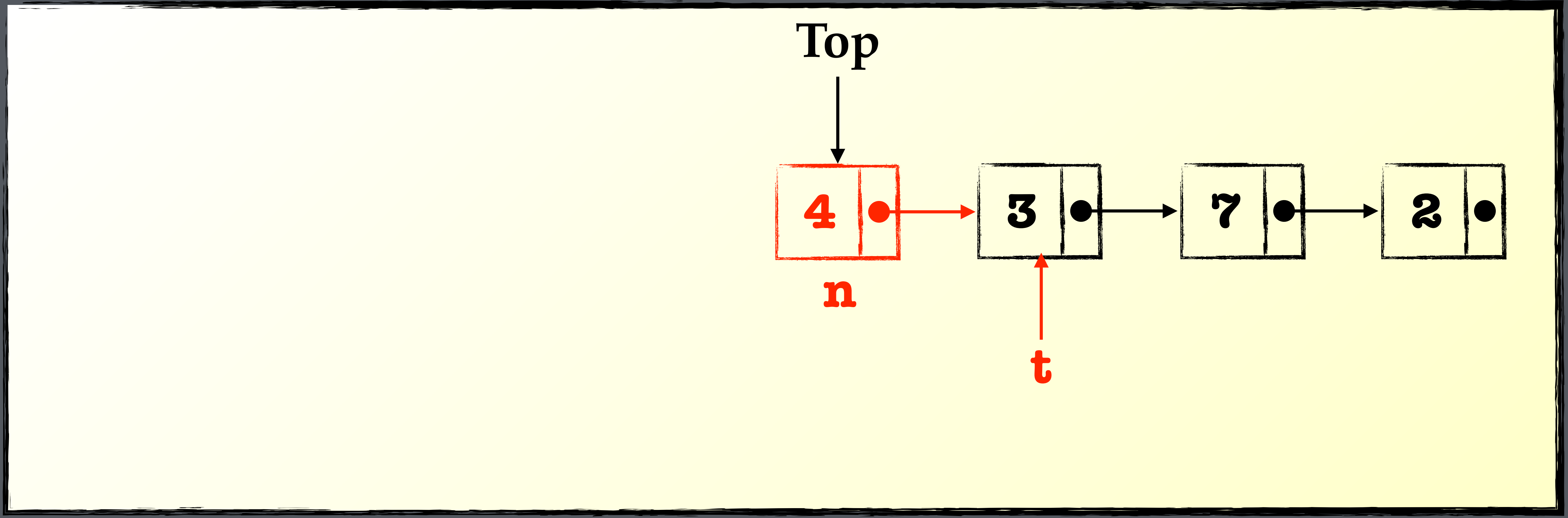
Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```




```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

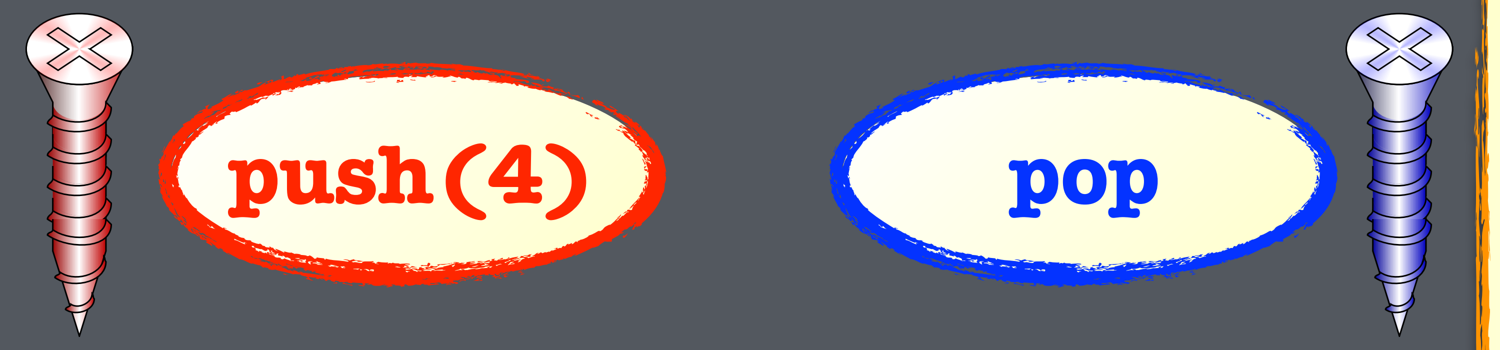




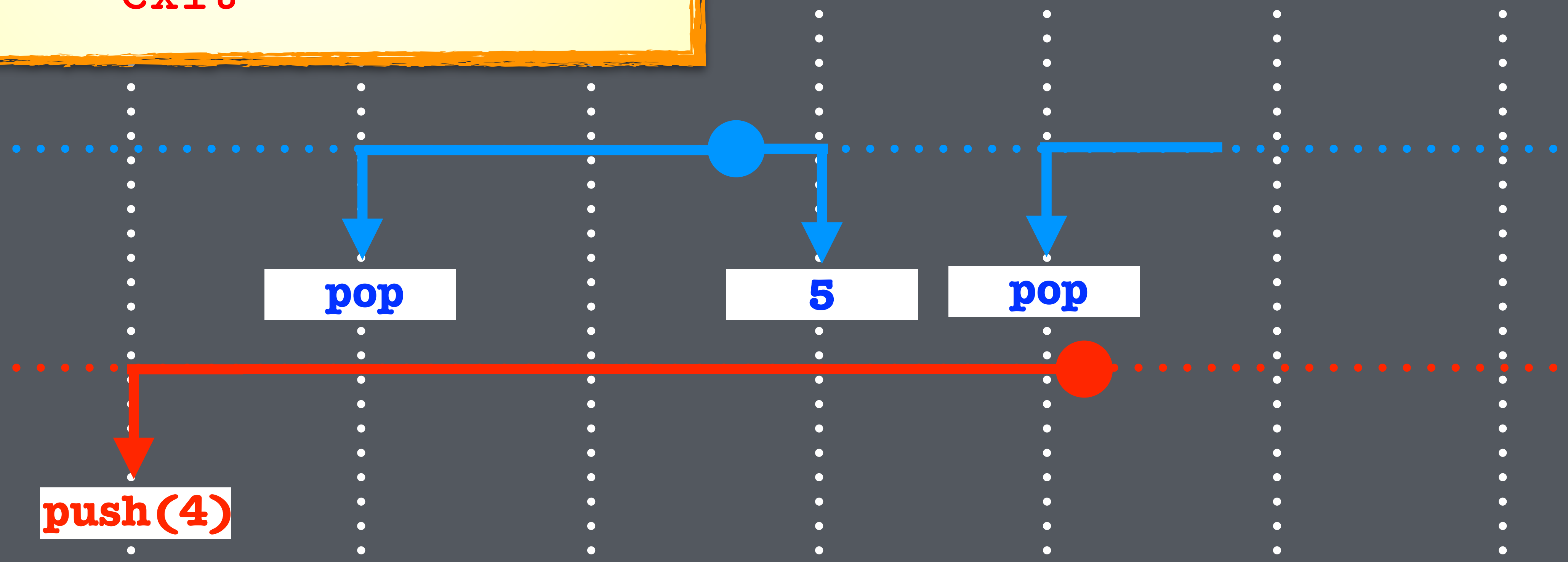
The Treiber Stack Algorithm

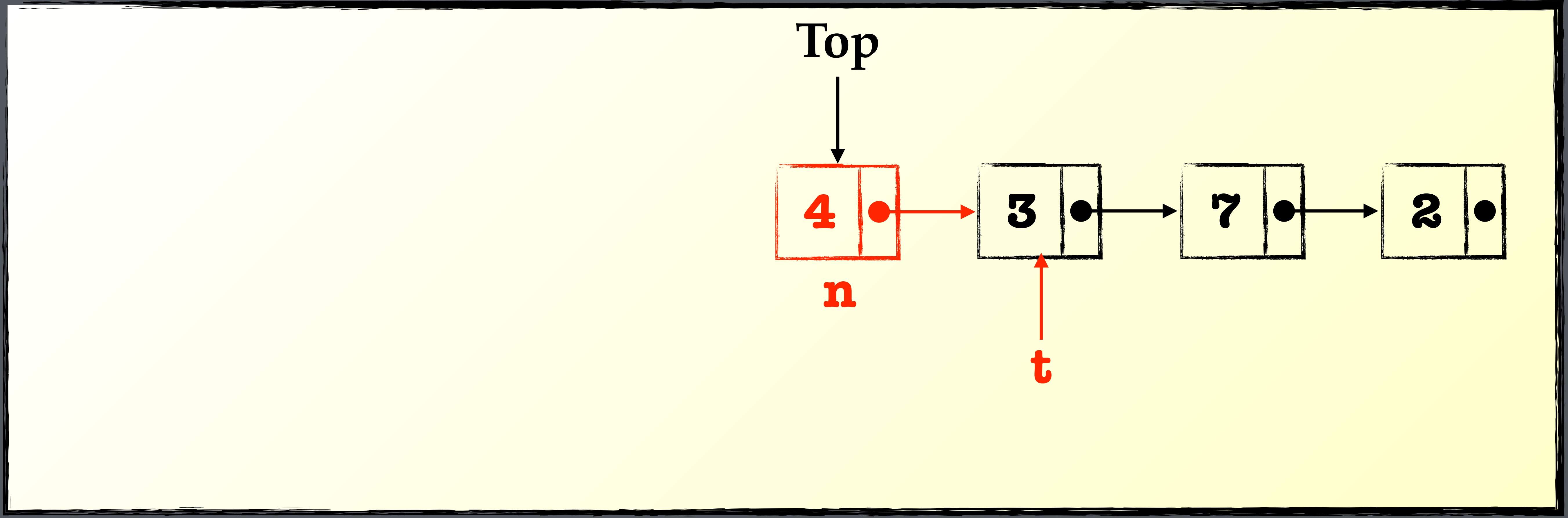
Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```




```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

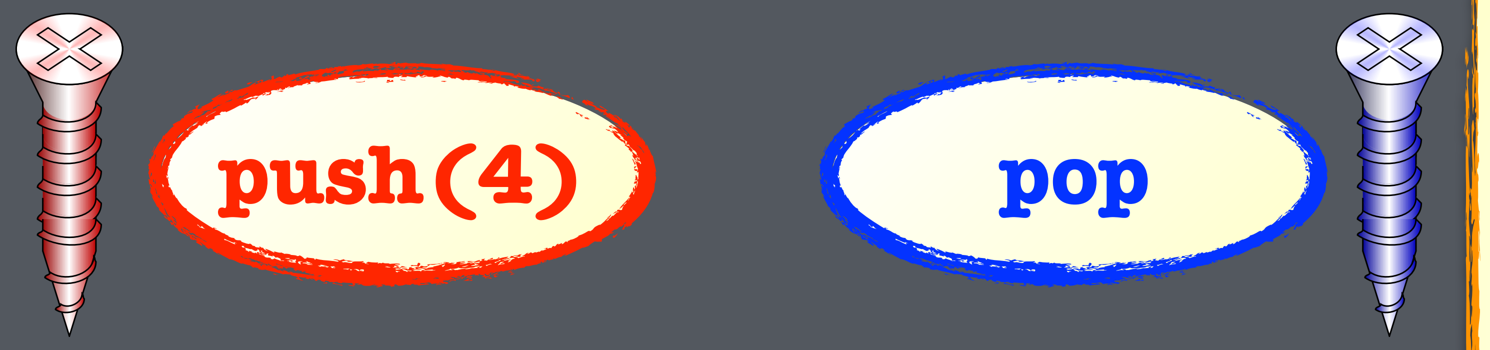




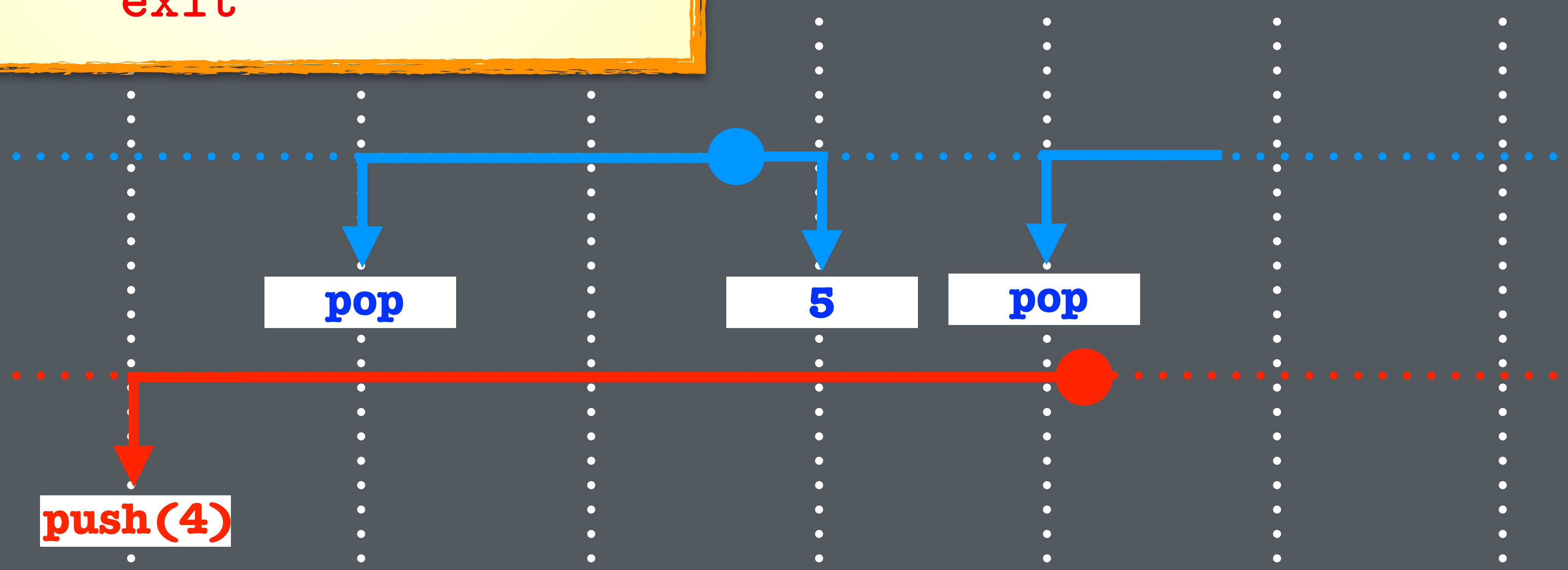
The Treiber Stack Algorithm

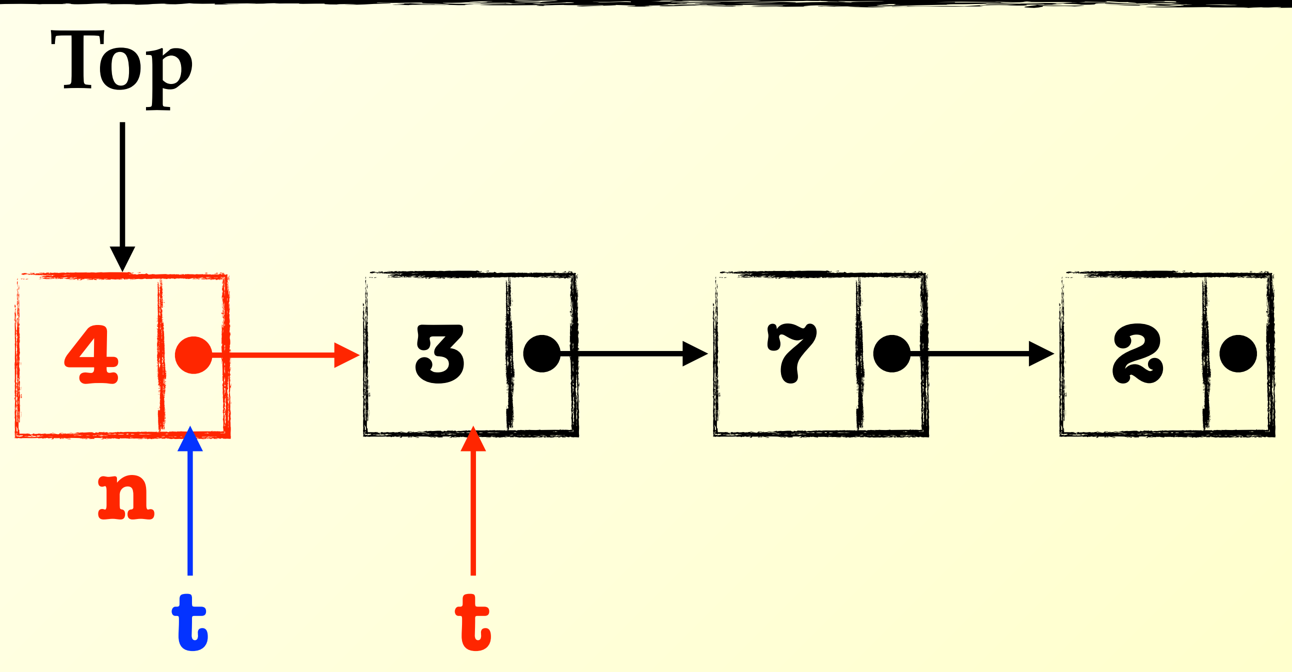
Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```




```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```





The Treiber Stack Algorithm

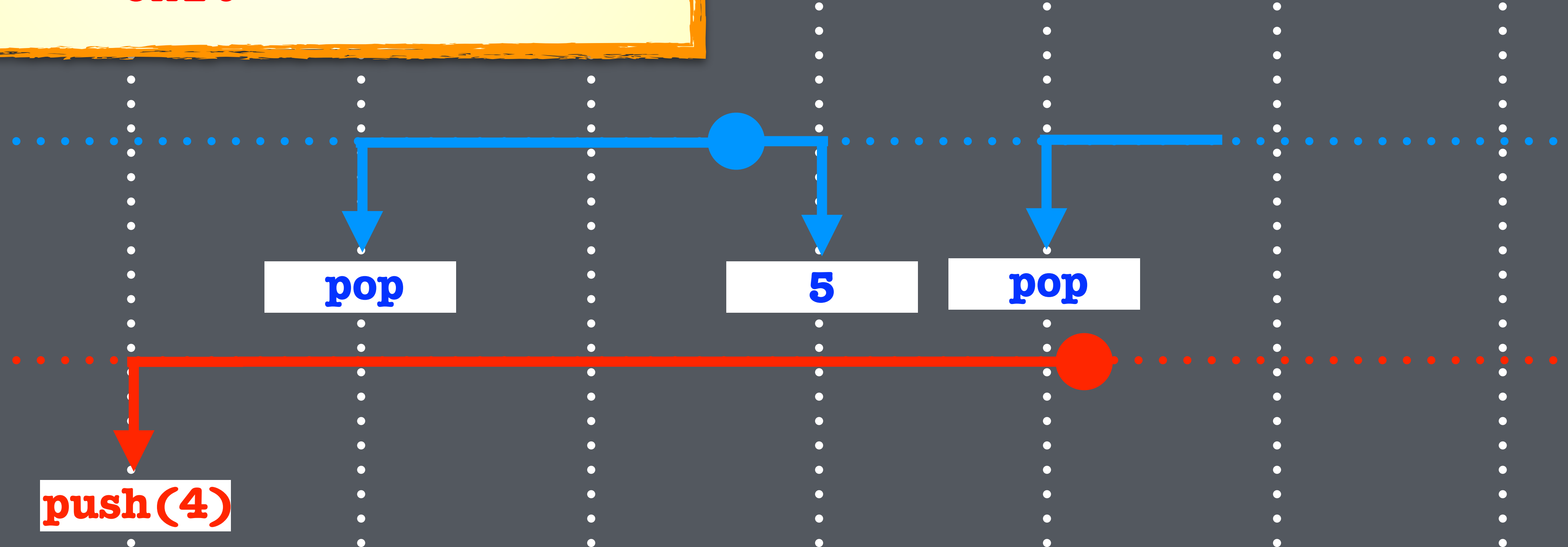
Linearization Policy

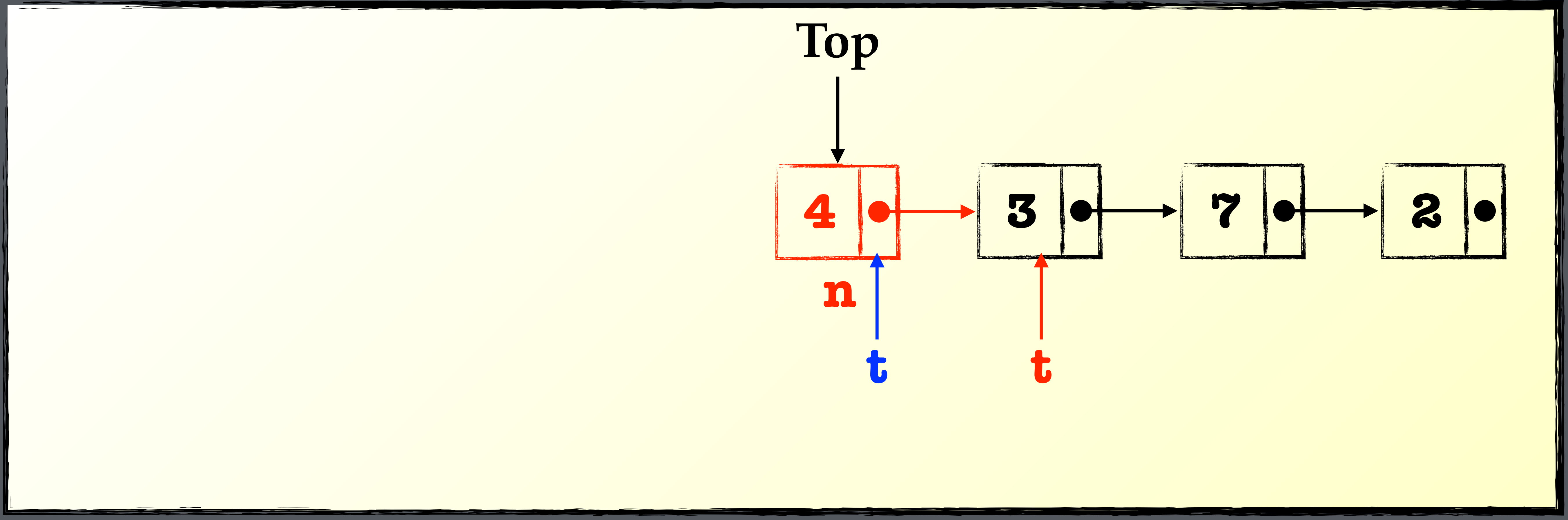
```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

push(4)

pop

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```





The Treiber Stack Algorithm

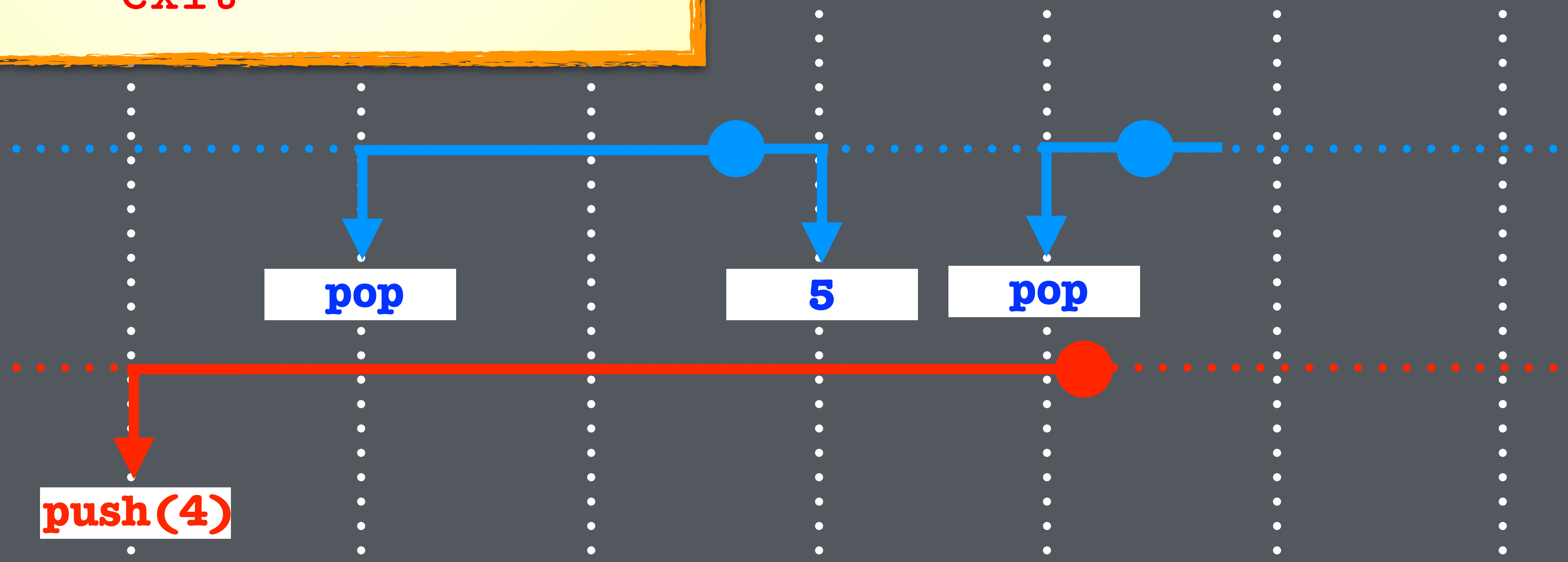
Linearization Policy

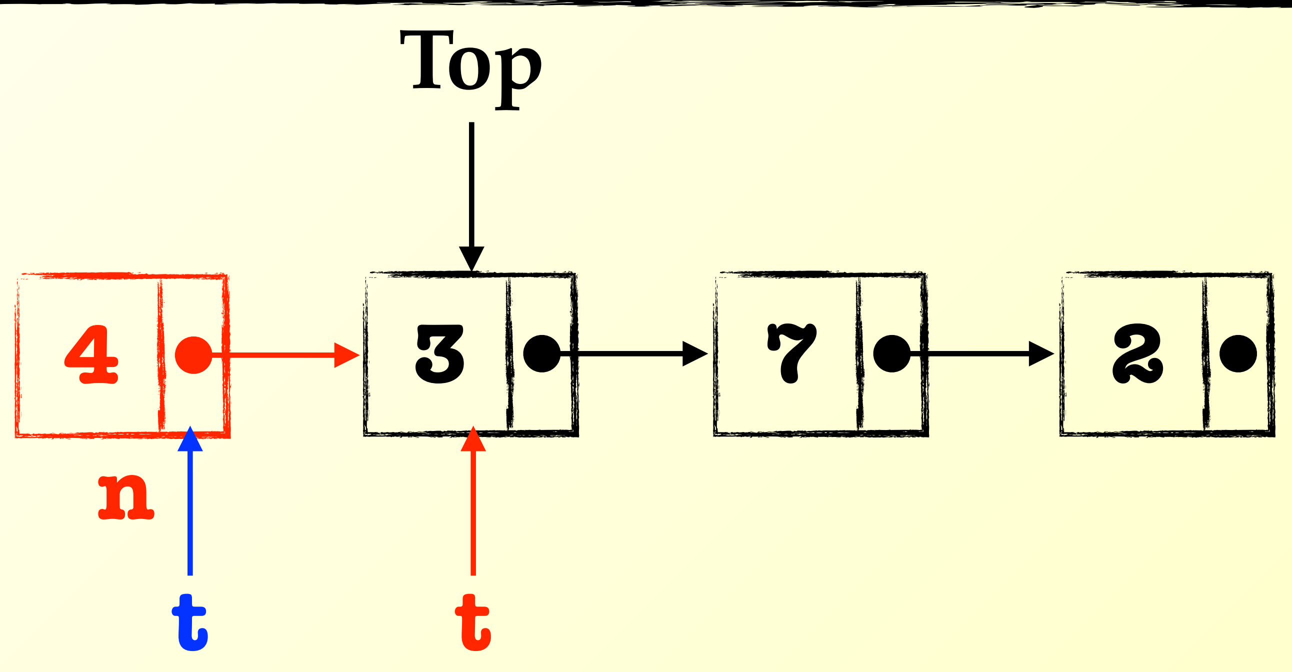
```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

push(4)

pop

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```





The Treiber Stack Algorithm

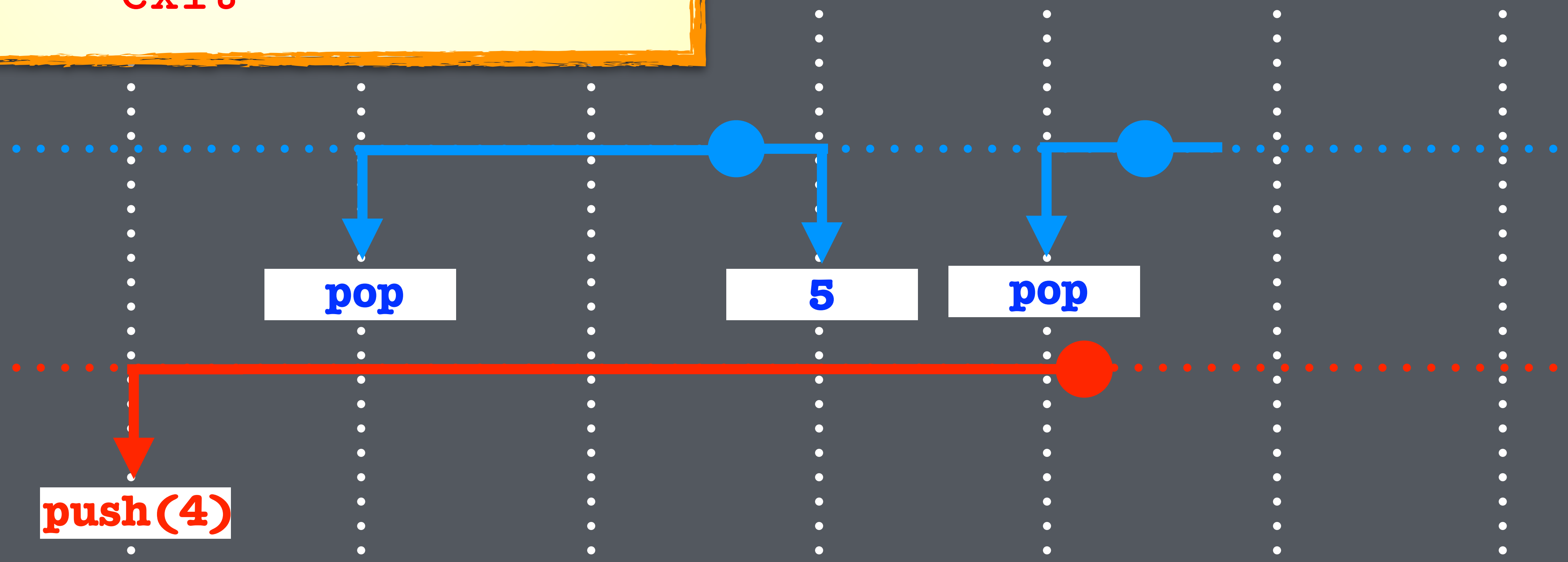
Linearization Policy

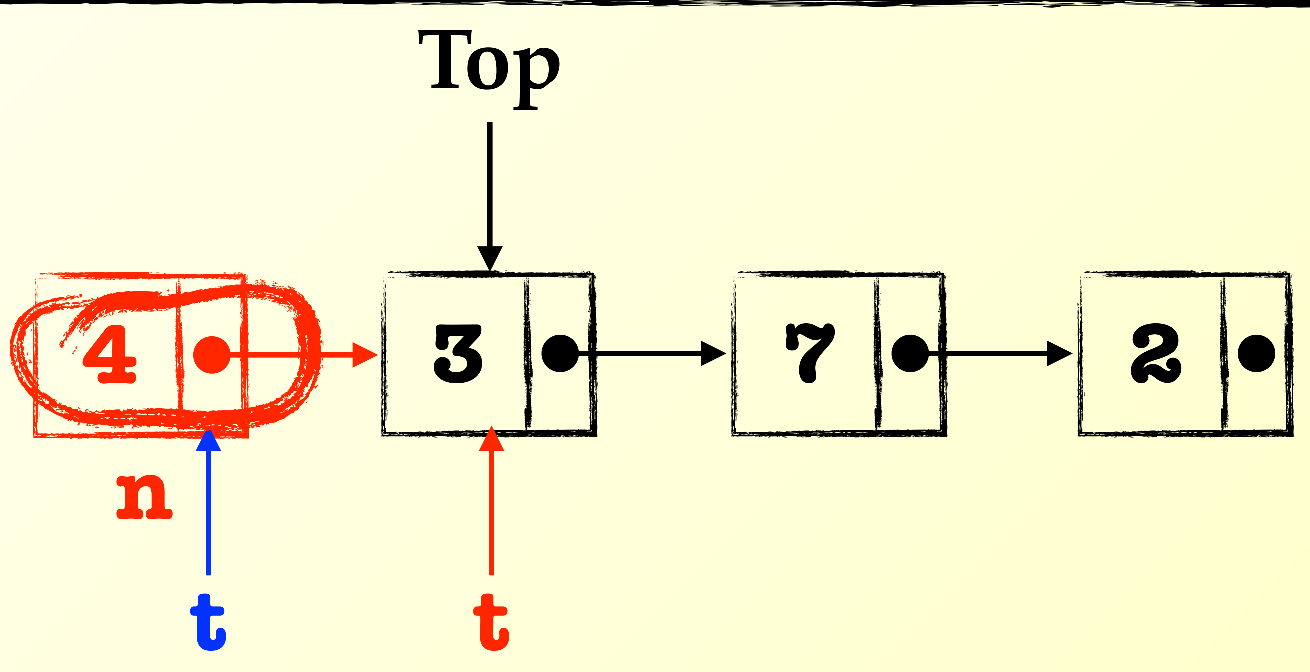
```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```

push(4)

pop

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

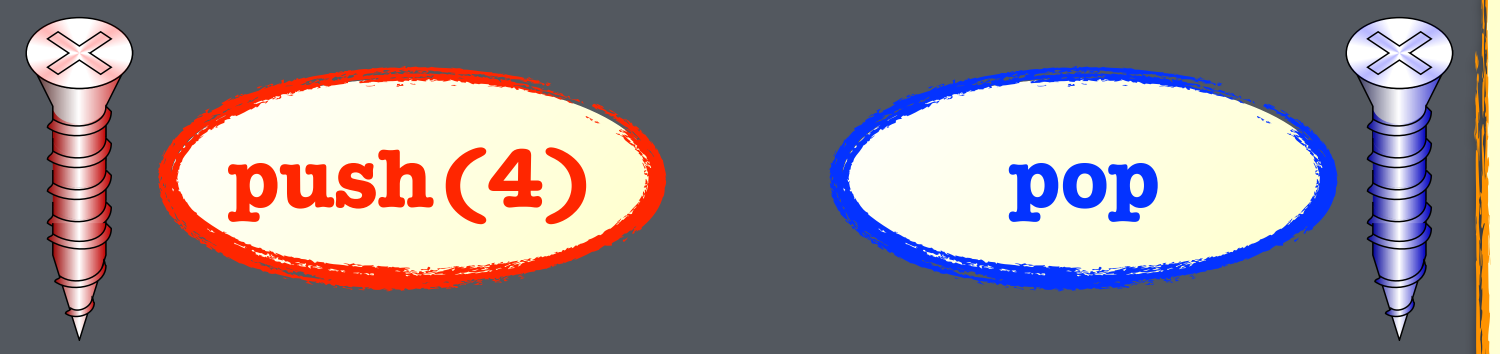




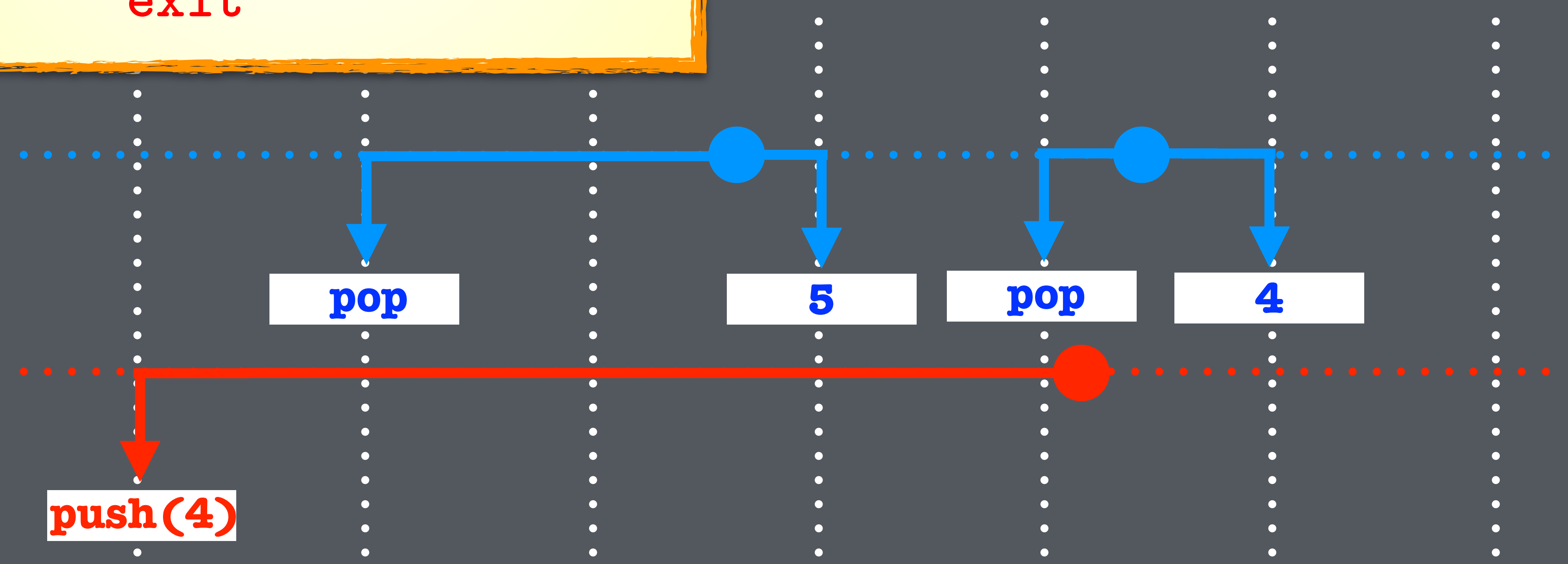
The Treiber Stack Algorithm

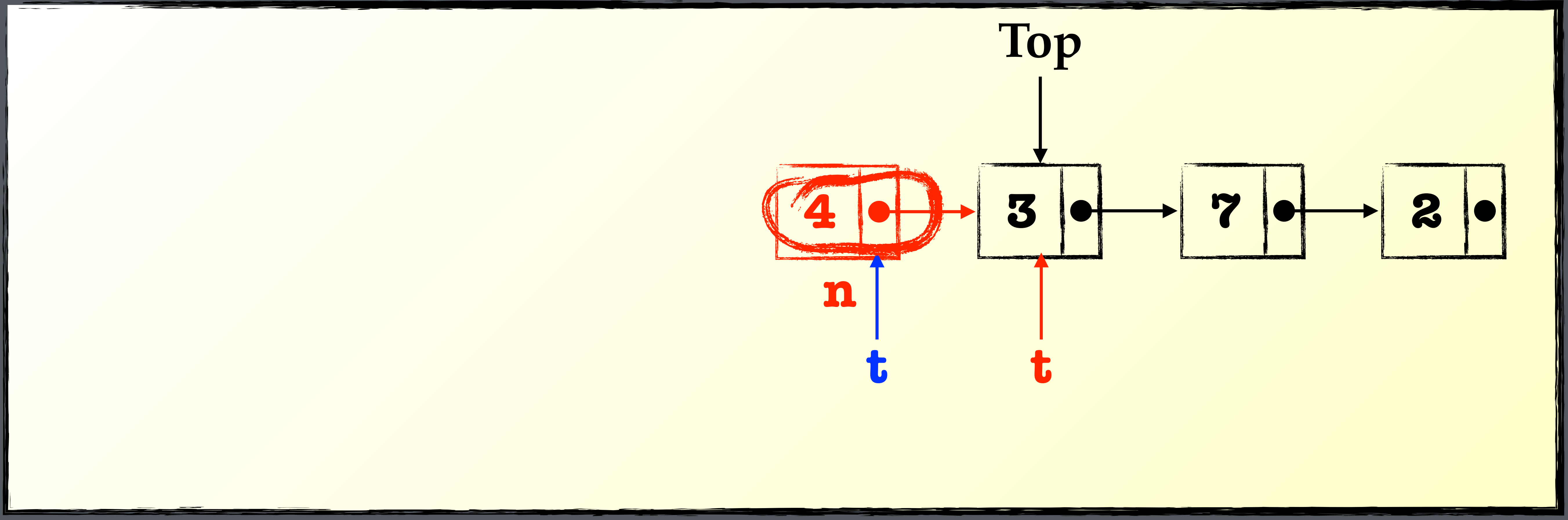
Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```




```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```

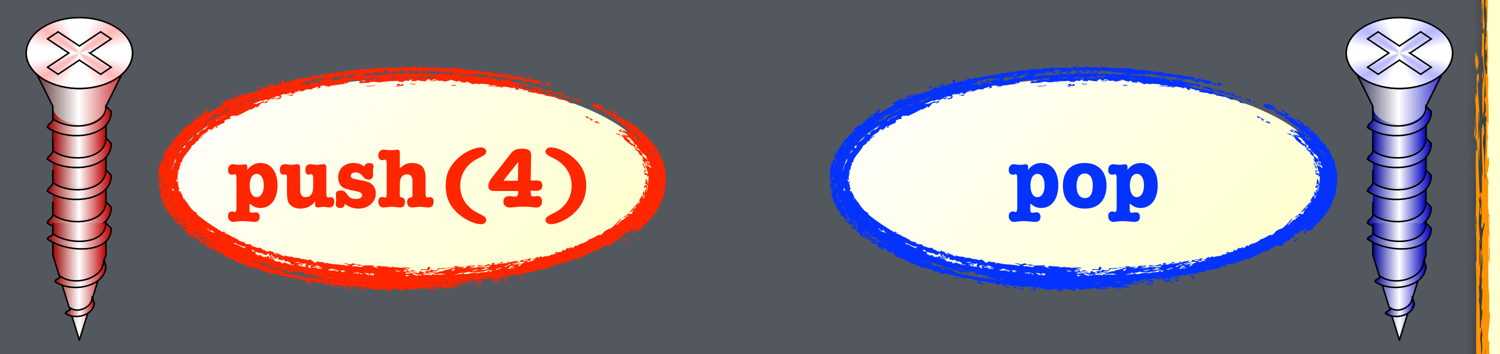




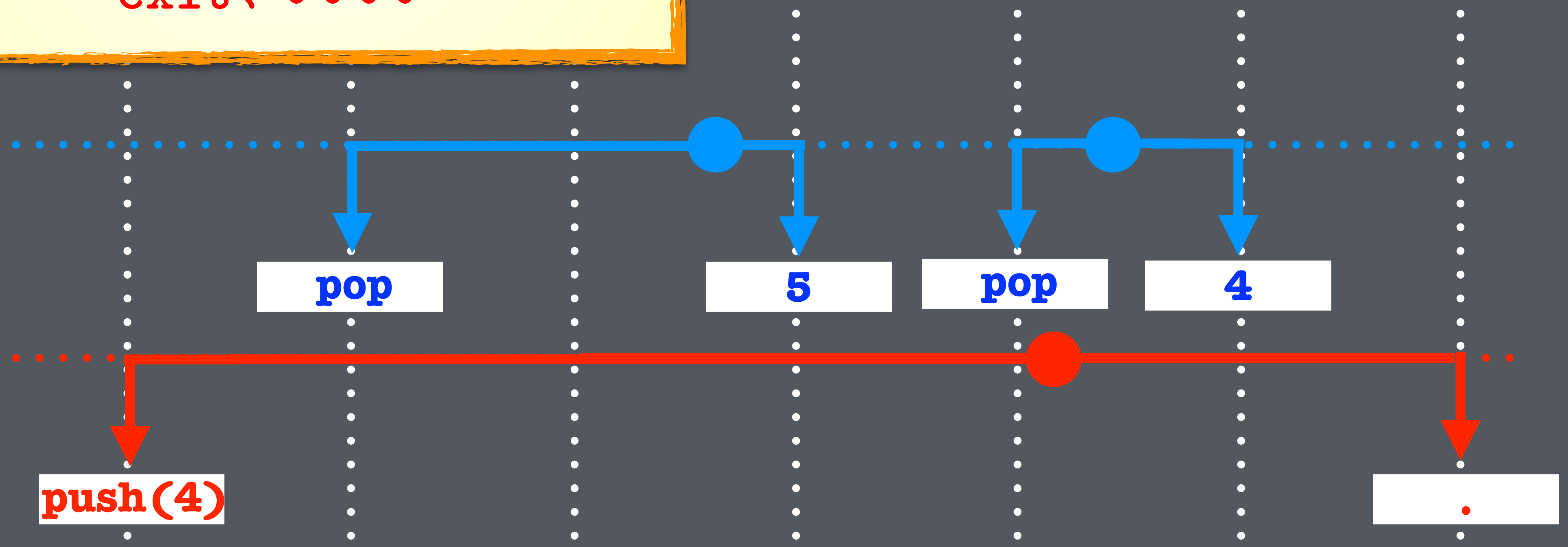
The Treiber Stack Algorithm

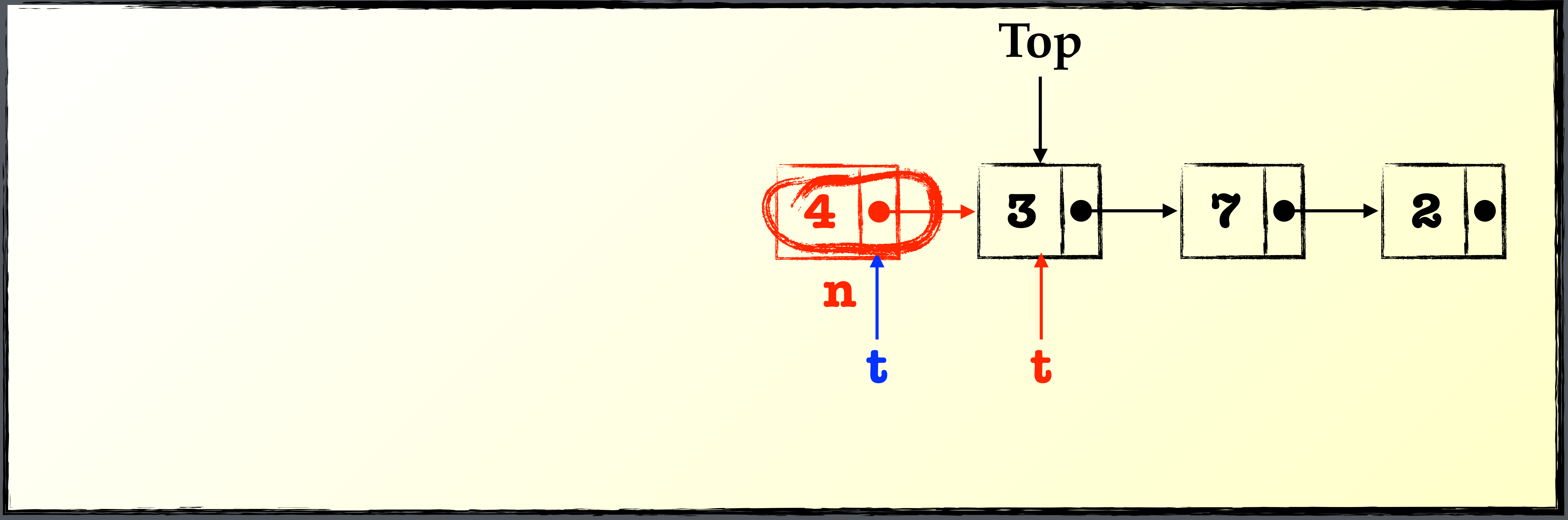
Linearization Policy

```
push(k):
Node t
n = new Node(k,-)
while (true)
  t = Top
  n.next = t
  if (CAS (Top,t,n))
    exit
```



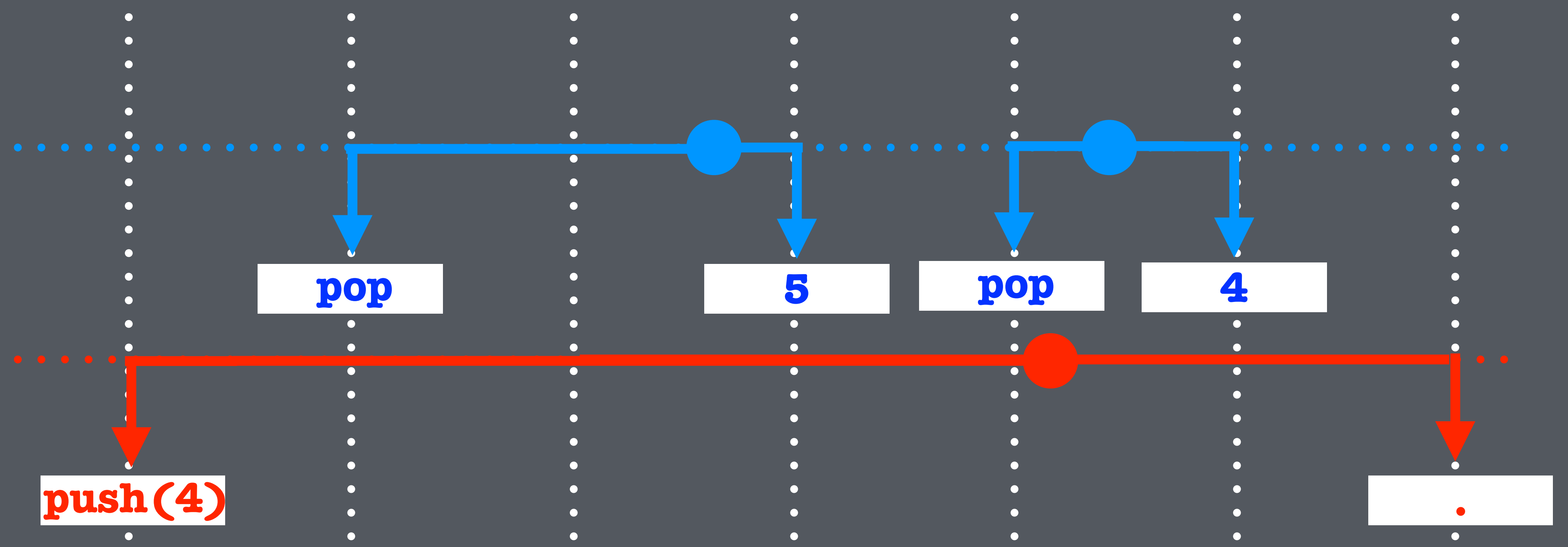

```
pop:
Node t
while (true)
  t = Top
  if (t = NULL)
    return *
    exit
  if (CAS (Top,t,t.next))
    return t.val
    exit
```





The Treiber Stack Algorithm

Linearization Policy



Libraries

A *configuration* $c = \langle v, u \rangle$ of $L[C]$ is a shared memory valuation $v \in V$, along with a map u mapping each thread $t \in \mathbb{N}$ to a tuple $u(t) = \langle \ell, m_0, m \rangle$, composed of a client-local state $\ell \in Q_C$, along with initial and current method states $m_0, m \in Q_L \cup \{\bot\}$; $m_0 = m = \bot$ when thread t is not executing a library

$$\textsc{Internal}\ \frac{u_1(t) = \langle \ell, m_0, m_1 \rangle \quad \langle m_1, v_1 \rangle \xhookrightarrow{a} \langle m_2, v_2 \rangle \quad u_2 = u_1\,(t \mapsto \langle \ell, m_0, m_2 \rangle)}{\langle v_1, u_1 \rangle \xrightarrow[L[C]]{\langle a, t \rangle} \langle v_2, u_2 \rangle}$$

$$\textsc{Call}\ \frac{u_1(t) = \langle \ell_1, \bot, \bot \rangle \quad m_0 \in I_M \quad \ell_1 \xhookrightarrow{M(m_0, m_f)}_{\mathrm{C}} \ell_2 \quad u_2 = u_1\,(t \mapsto \langle \ell_1, m_0, m_0 \rangle)}{\langle v, u_1 \rangle \xrightarrow[L[C]]{\mathsf{call}(M, m_0, t)} \langle v, u_2 \rangle}$$

$$\textsc{Return}\ \frac{u_1(t) = \langle \ell_1, m_0, m_f \rangle \quad m_f \in F_M \quad \ell_1 \xhookrightarrow{M(m_0, m_f)}_{\mathrm{C}} \ell_2 \quad u_2 = u_1\,(t \mapsto \langle \ell_2, \bot, \bot \rangle)}{\langle v, u_1 \rangle \xrightarrow[L[C]]{\mathsf{ret}(M, m_f, t)} \langle v, u_2 \rangle}$$

Fig. 1. The transition relation $\to_{L[C]}$ for the library-client composition $L[C]$.

VASS model

We associate to each concurrent system $L[C]$ a *canonical VASS*,[2] denoted $\mathcal{A}_{L[C]}$, whose states are the set of shared-memory valuations, and whose vector components count the number of threads in each thread-local state; a transition of $\mathcal{A}_{L[C]}$ from $\langle v_1, \boldsymbol{n}_1 \rangle$ to $\langle v_2, \boldsymbol{n}_2 \rangle$ updates the shared-memory valuation from v_1 to v_2 and the local state of some thread t from $u_1(t)$ to $u_2(t)$ by decrementing the $u_1(t)$-component of $\boldsymbol{n}_1$, and incrementing the $u_2(t)$-component, to derive $\boldsymbol{n}_2$.

Specifications

A *specification* S of a library L is a language over the *specification alphabet*

$$\Sigma_S \stackrel{\text{def}}{=} \{M[m_0, m_f] : M \in L, m_0, m_f \in Q_M\}.$$

Definition 2 (Linearizability [20]). *A trace* τ *is* S-linearizable *when there exists a completion*[4] π *of a strict, serial permutation of* τ *such that* $(\pi \mid S) \in S$*.*

Specifications

The *pending closure* of a specification S, denoted $\overline{S}$ is the set of S-images of serial sequences which have completions whose S-images are in S:

$$\overline{S} \stackrel{\text{def}}{=} \{(\sigma \mid S) \in \overline{\Sigma}^*_S : \exists \sigma' \in \Sigma^*_S.\ (\sigma' \mid S) \in S \text{ and } \sigma' \text{ is a completion of } \sigma\}.$$

Specifications

The *pending closure* of a specification S, denoted $\overline{S}$ is the set of S-images of serial sequences which have completions whose S-images are in S:

$$\overline{S} \stackrel{\text{def}}{=} \{(\sigma \mid S) \in \overline{\Sigma}_S^* : \exists \sigma' \in \Sigma_S^*.\ (\sigma' \mid S) \in S \text{ and } \sigma' \text{ is a completion of } \sigma\}.$$

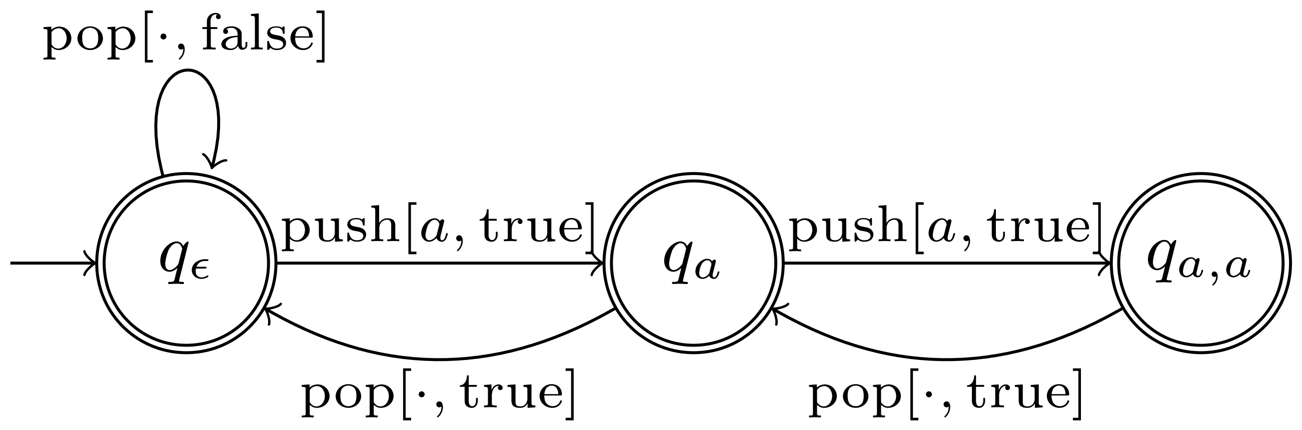


Fig. 2. The sequential specification of two-element stacks containing the (abstract) value a, given as the language of a finite automaton, whose operation alphabet indicates both the argument and return values.

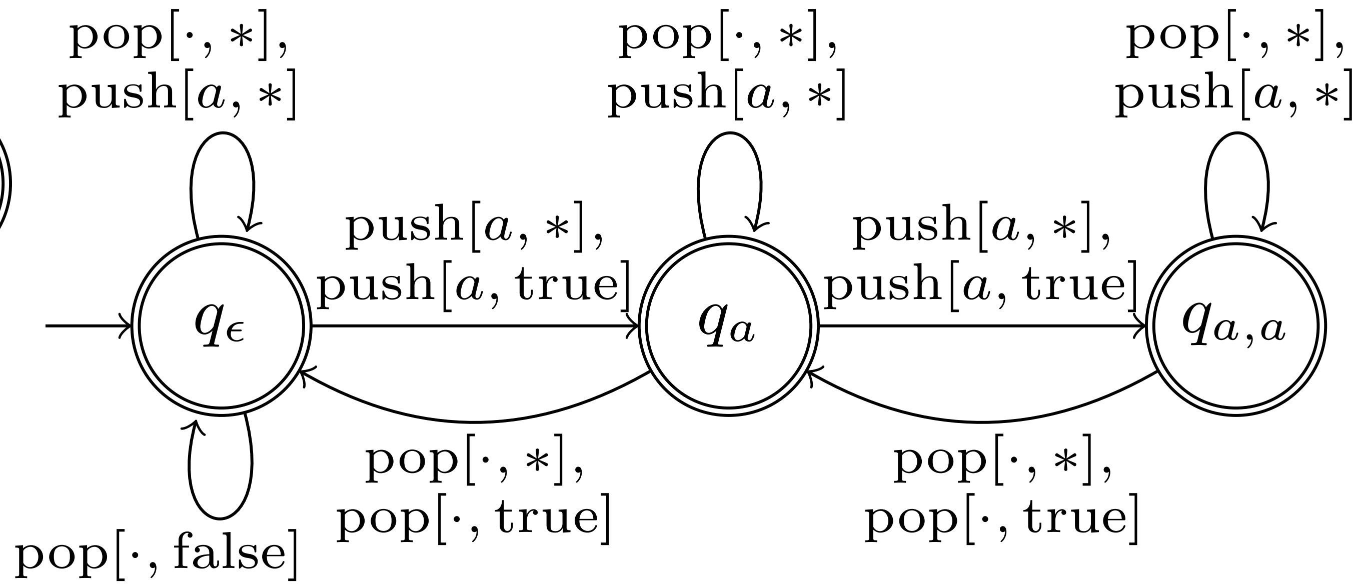


Fig. 3. The pending closure of the stack specification from Figure 2.

Specifications

The *pending closure* of a specification S, denoted $\overline{S}$ is the set of S-images of serial sequences which have completions whose S-images are in S:

$$\overline{S} \stackrel{\text{def}}{=} \{(\sigma \mid S) \in \overline{\Sigma}_S^* : \exists \sigma' \in \Sigma_S^*.\ (\sigma' \mid S) \in S \text{ and } \sigma' \text{ is a completion of } \sigma\}.$$

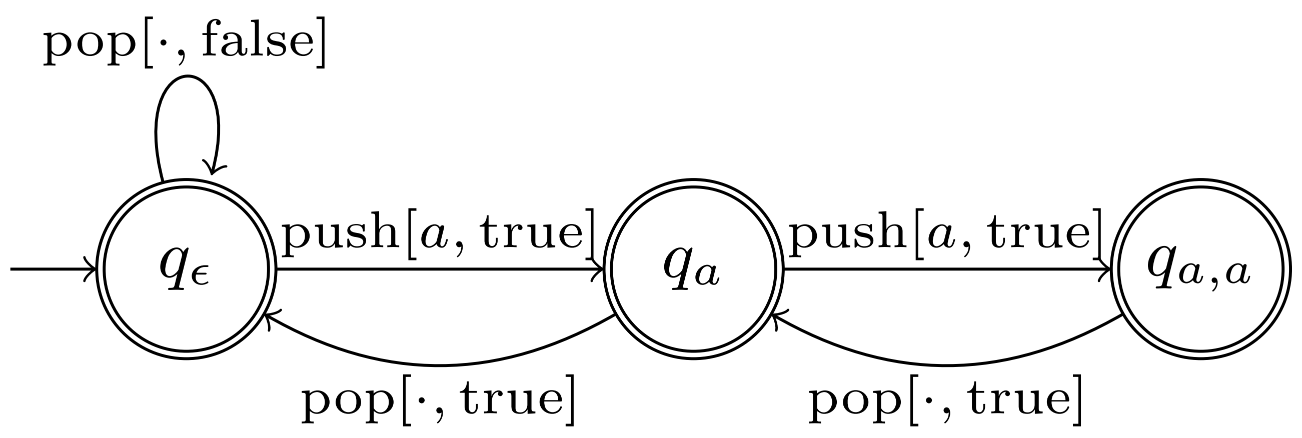


Fig. 2. The sequential specification of two-element stacks containing the (abstract) value a, given as the language of a finite automaton, whose operation alphabet indicates both the argument and return values.

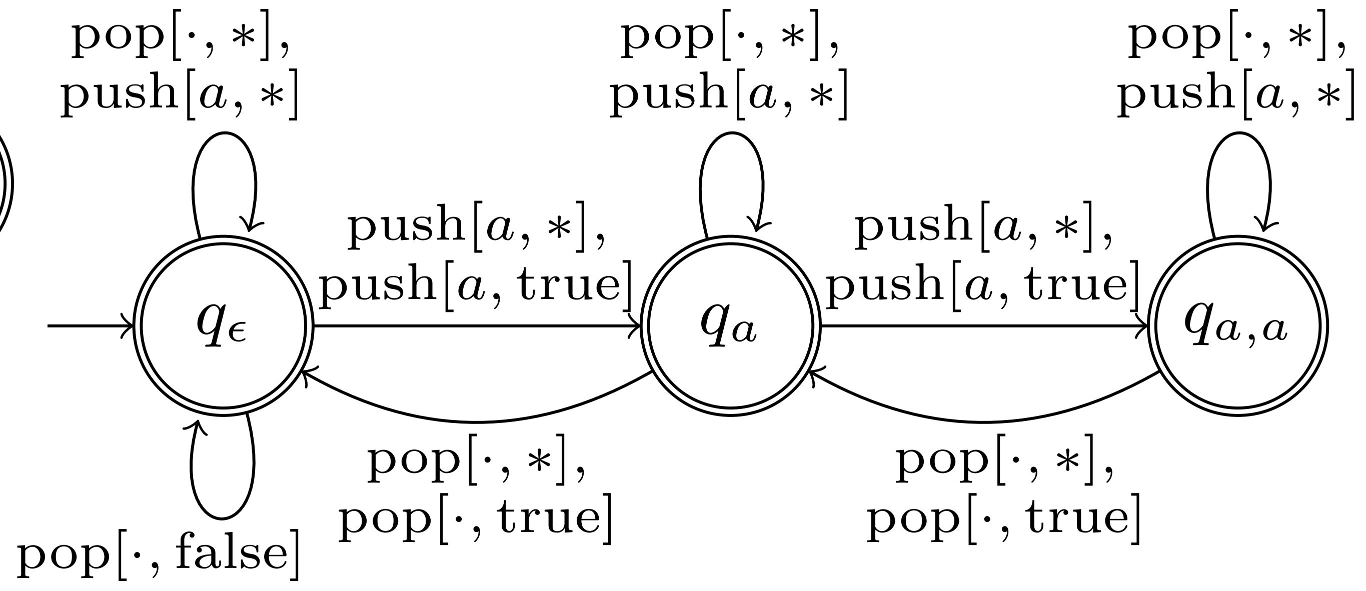


Fig. 3. The pending closure of the stack specification from Figure 2.

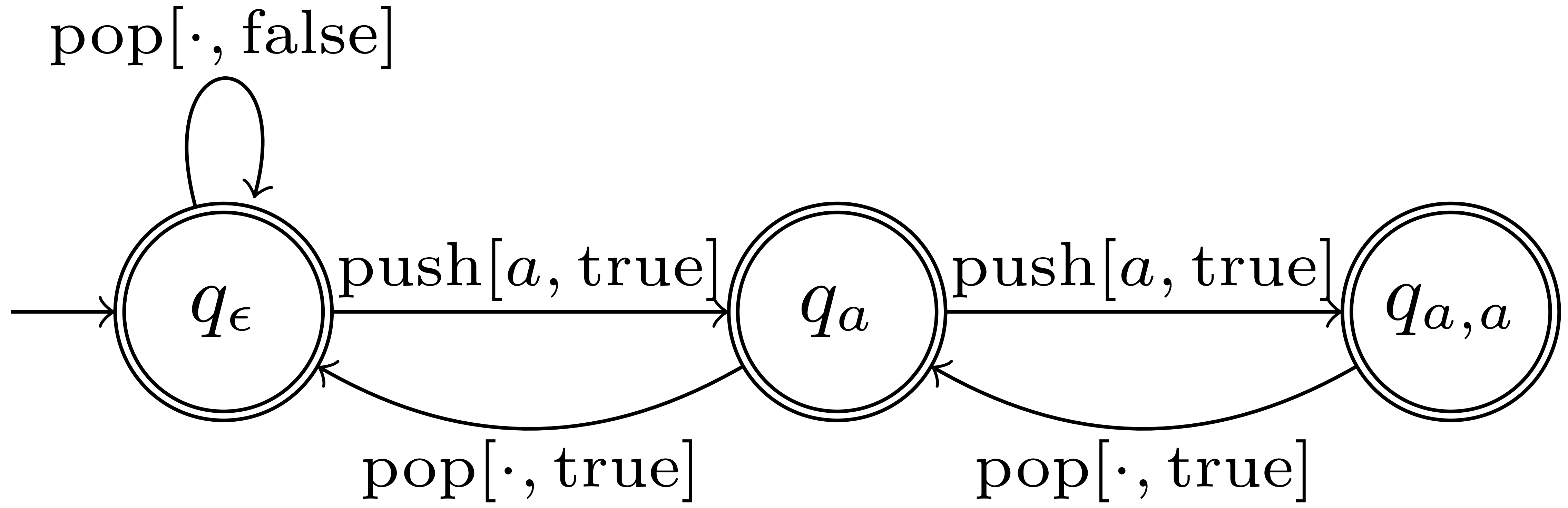
Lemma 1. *The pending closure $\overline{S}$ of a regular specification S is regular.*

Lemma 2. *A trace τ is S-linearizable if and only if there exists a strict, serial permutation π of τ such that $(\pi \mid S) \in \overline{S}$.*

Read-only operations

Given a method M of a library L and $m_0, m_f \in Q_M$, an $M[m_0, m_f]$-operation θ is *read-only* for a specification S if and only if for all $w_1, w_2, w_3 \in \Sigma_S^*$,

1. If $w_1 \cdot M[m_0, m_f] \cdot w_2 \in S$ then $w_1 \cdot M[m_0, m_f]^k \cdot w_2 \in S$ for all $k \geq 0$, and
2. If $w_1 \cdot M[m_0, m_f] \cdot w_2 \in S$ and $w_1 \cdot w_3 \in S$ then $w_1 \cdot M[m_0, m_f] \cdot w_3 \in S$.



Linearization points

The *control graph* $G_M = \langle Q_M, E \rangle$ is the quotient of a method M's transition system by shared-state valuations V: $\langle m_1, a, m_2 \rangle \in E$ iff $\langle m_1, v_1 \rangle \hookrightarrow^{\mathrm{a}}_{\mathrm{M}} \langle m_2, v_2 \rangle$ for some $v_1, v_2 \in V$. A function $\mathsf{LP} : L \to \wp(\Sigma_L)$ is called a *linearization-point mapping* when for each $M \in L$:

1. each symbol $a \in \mathsf{LP}(M)$ labels at most one transition of M,
2. any directed path in G_M contains at most one symbol of $\mathsf{LP}(M)$, and
3. all directed paths in G_M containing $a \in \mathsf{LP}(M)$ reach the same $m_a \in F_M$.

An action $\langle a, i \rangle$ of an M-operation is called a *linearization point* when $a \in \mathsf{LP}(M)$, and operations containing linearization points are said to be *effectuated*; $\mathsf{LP}(\theta)$ denotes the unique linearization point of an effectuated operation θ. A *read-points mapping* $\mathsf{RP} : \Theta \to \mathbb{N}$ for an action sequence σ with operations Θ maps each read-only operation θ to the index $\mathsf{RP}(\theta)$ of an internal θ-action in σ.

Exercices (1)

- Does the Herlihy & Wing queue admit **fixed** linearization points ?

```
void enq(int x) {
  i = back++; items[i] = x;
}
int deq() {
  while (1) {
    range = back - 1;
    for (int i = 0; i <= range; i++) {
      x = swap(items[i],null);
      if (x != null) return x;
} } }
```

Static linearizability

An action sequence σ is called *effectuated* when every completed operation of σ is either effectuated or read-only, and an effectuated completion σ' of σ is *effect preserving* when each effectuated operation of σ also appears in σ'. Given a linearization-point mapping LP, and a read-points mapping RP of an action sequence σ, we say a permutation π of σ is *point preserving* when every two operations of π are ordered by their linearization/read points in σ.

Definition 4. *A trace τ is* $\langle S, \mathsf{LP}\rangle$-linearizable *when τ is effectuated, and there exists a read-points mapping* RP *of τ, along with an effect-preserving completion π of a strict, point-preserving, and serial permutation of τ such that $(\pi \mid S) \in S$.*

Definition 5 (Static Linearizability). *The system $L[C]$ is S-*static linearizable *when $L[C]$ is $\langle S, \mathsf{LP}\rangle$-linearizable for some mapping* LP.

Checking Static Linerizability

- A_S = a deterministic automaton recognizing the Specification
- we define a monitor to be composed with L[C] that simulates the Specification
  - methods have a new local variable RO which is initially ∅ (records return values of read-only operations)
    - if mf ∈ RO in an invocation of M, then M[m0,mf] is read-only and a state of A_S in which $M[m_0,m_f]$ is enabled has been observed
  - L[C] executes a linearization point => the state of the Specification is advanced to the $M[m_0,m_f]$ successor (m_0 is the initial state of the current operation and m_f is the unique final state reachable from this lin. point)
  - L[C] executes an internal action from an $M[m_0,*]$ operation => RO is enriched with every m_f such that $M[m_0,m_f]$ is read-only and enabled in the current specification state
  - L[C] executes the return of an $M[m_0,m_f]$ read-only operation => if $m_f \notin$ RO then the monitor goes to an error state

EXPSPACE-hardness

- Reduce control state reachability in VASS (which is EXPSPACE-complete) to static linearizability
  - Use the library from the undecidability proof without the zero-test method (the specification excludes only executions not reaching the target state)

Checking Linearizability: Complexity (finite-state implementations)

Bounded Nb. of Threads:

- EXSPACE-complete [Alur et al., 1996, Hamza 2015]

Unbounded Nb. of Threads:

- Undecidable [Bouajjani et al., 2013]
- Decidable with “fixed linearization points” [Bouajjani et al. 2013]

Alur et al. 1996: Rajeev Alur, Kenneth L. McMillan, Doron A. Peled: Model-Checking of Correctness Conditions for Concurrent Objects. LICS 1996

Bouajjani et al., 2013: Ahmed Bouajjani, Michael Emmi, Constantin Enea, Jad Hamza: Verifying Concurrent Programs against Sequential Specifications. ESOP 2013

Hamza 2015: Jad Hamza: On the Complexity of Linearizability. NETYS 2015