# Online Tree Caching





**Marcin Bieńkowski**
University of Wrocław

Jan Marcinkowski
University of Wrocław

Maciej Pacut
University of Wrocław

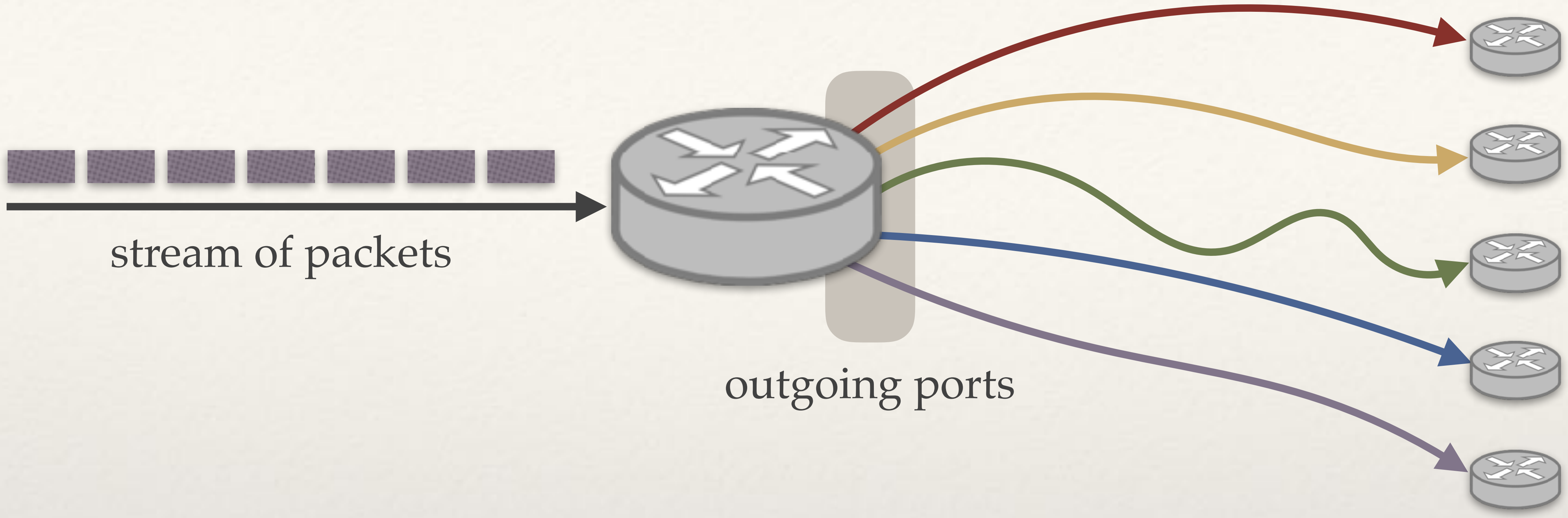
Stefan Schmid
University of Aalborg

Aleksandra Spyra
Google

**SPAA 2017**

# Motivation: routers

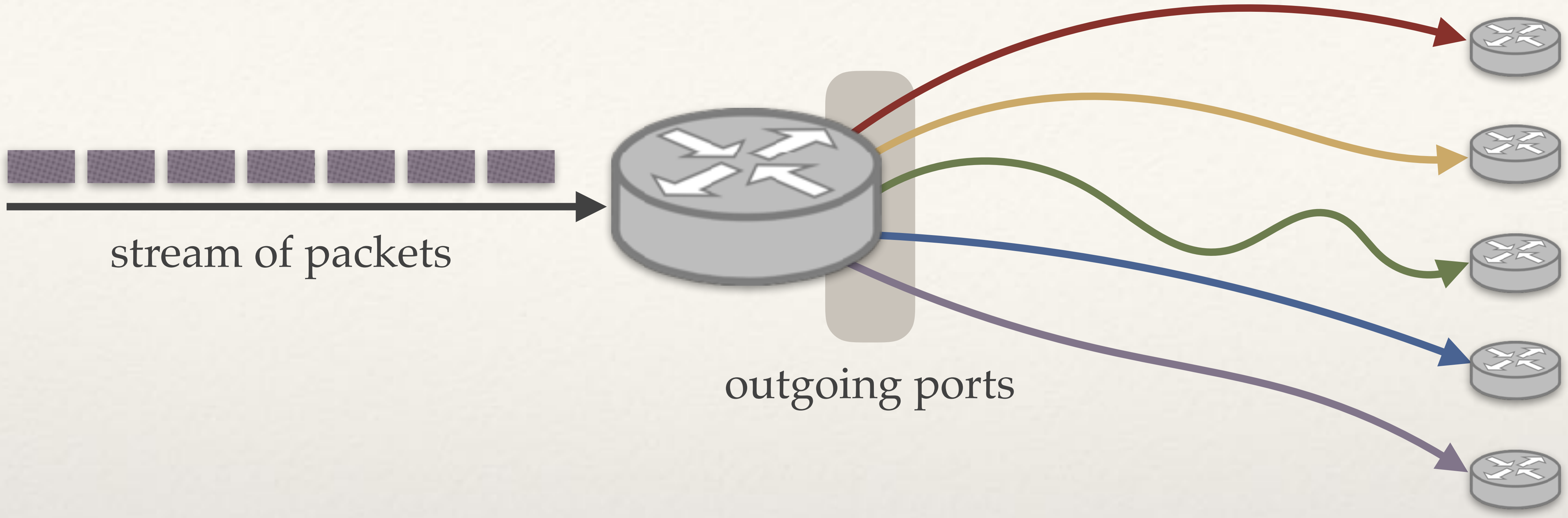
# Router




**For each incoming packet, a router:**

- takes a packet **destination address** (bit string of length $w$),
- finds a matching rule in its **forwarding table (FIB)**,
- the rule defines outgoing **port** for a packet.

# Router



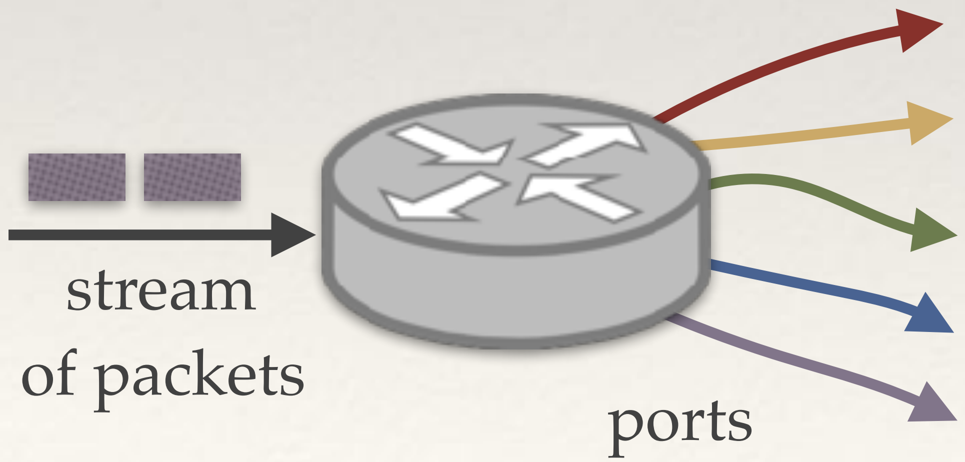

32 bits for IPv4, 128 bits for IPv6

**For each incoming packet, a router:**

- takes a packet **destination address** (bit string of length $w$),
- finds a matching rule in its **forwarding table (FIB)**,
- the rule defines outgoing **port** for a packet.

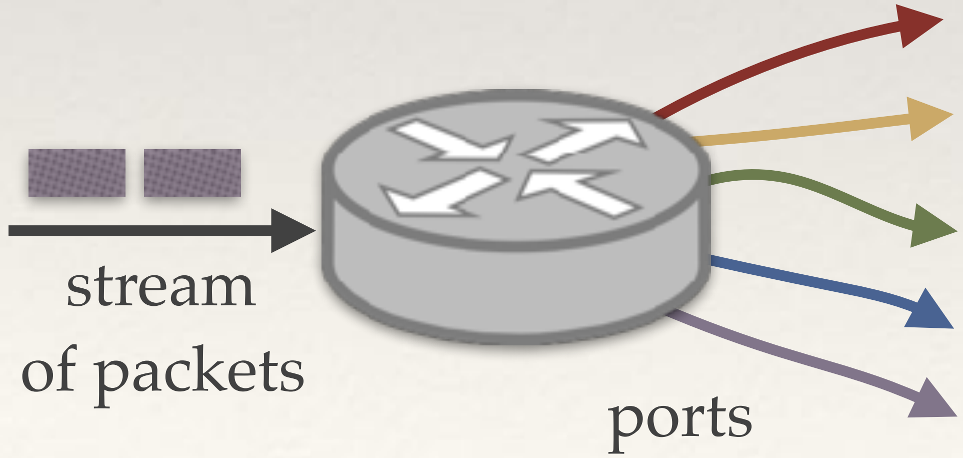
# Forwarding table (FIB)

| If packet's destination address starts with prefix | Forward it via port … |
|---|---|
| ε | port gray |
| 0 | port yellow |
| 011 | port green |
| 1 | port red |
| … | … |
| 1010 | port blue |



# Forwarding table (FIB)

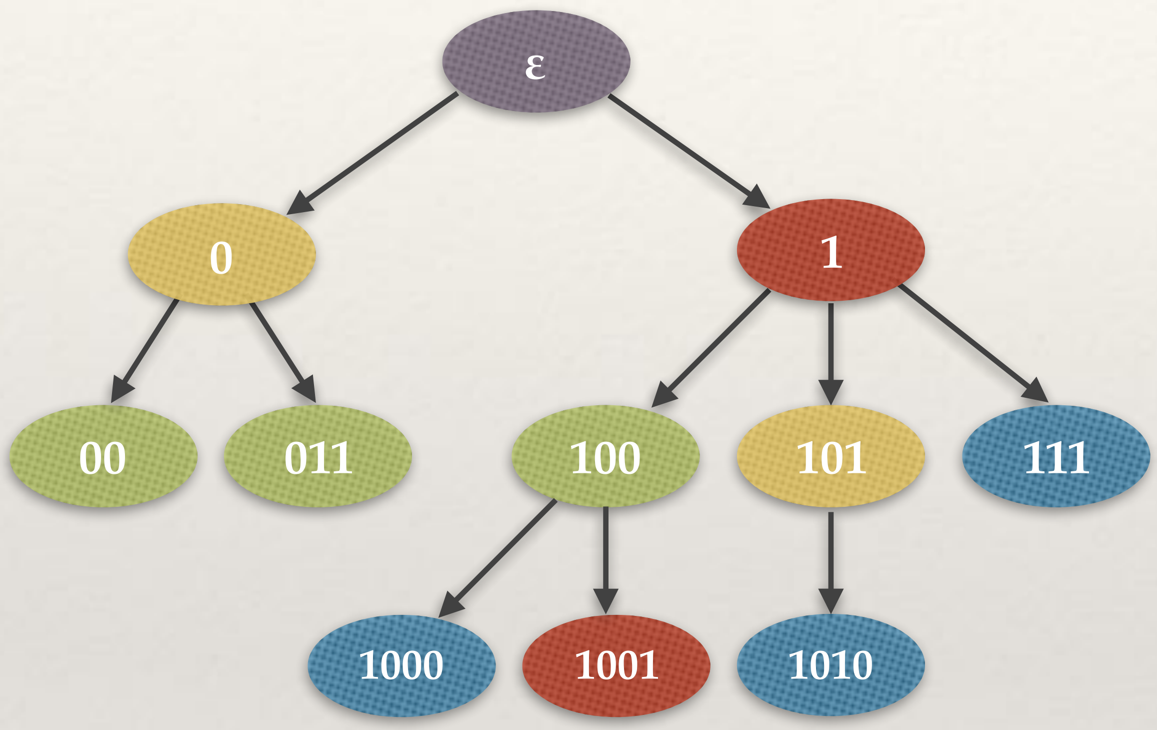
| If packet's destination address starts with prefix | Forward it via port … |
|---|---|
| ε | port gray |
| 0 | port yellow |
| 011 | port green |
| 1 | port red |
| … | … |
| 1010 | port blue |




If many rules match, choose the one that "matches best" = has longest prefix.

# Alternative representation of FIB

## Trie storing prefixes

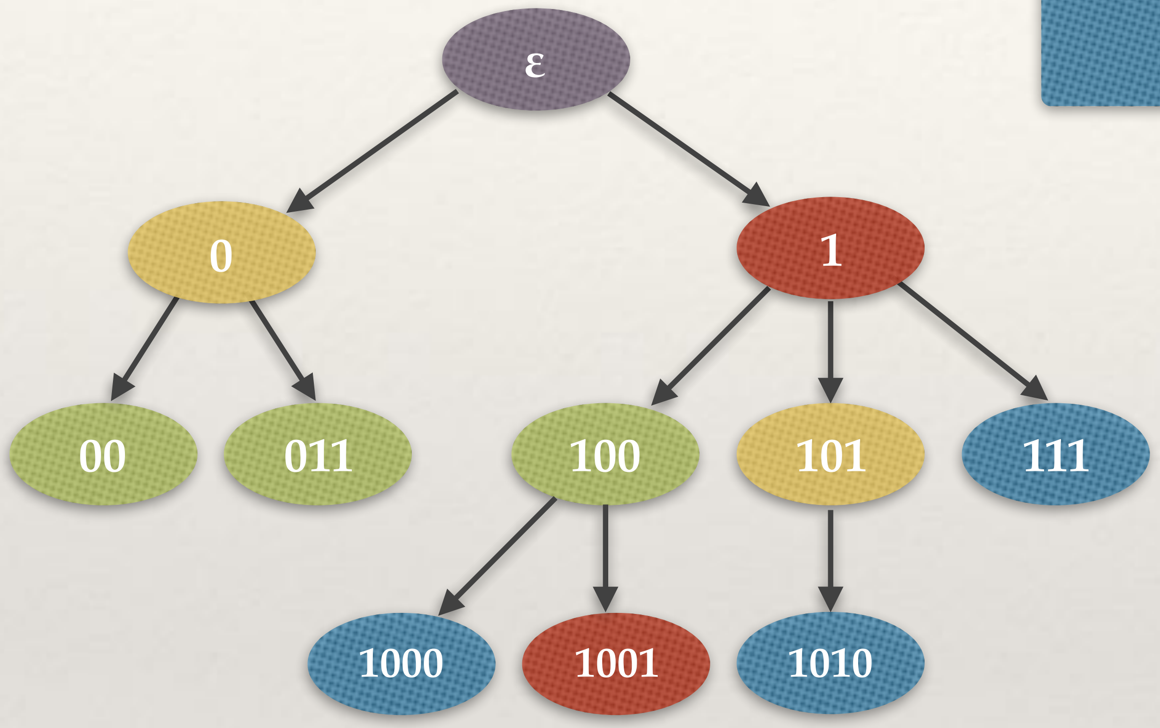



**Rule lookup = start from the root and proceed as deep as possible.**

# Alternative representation of FIB

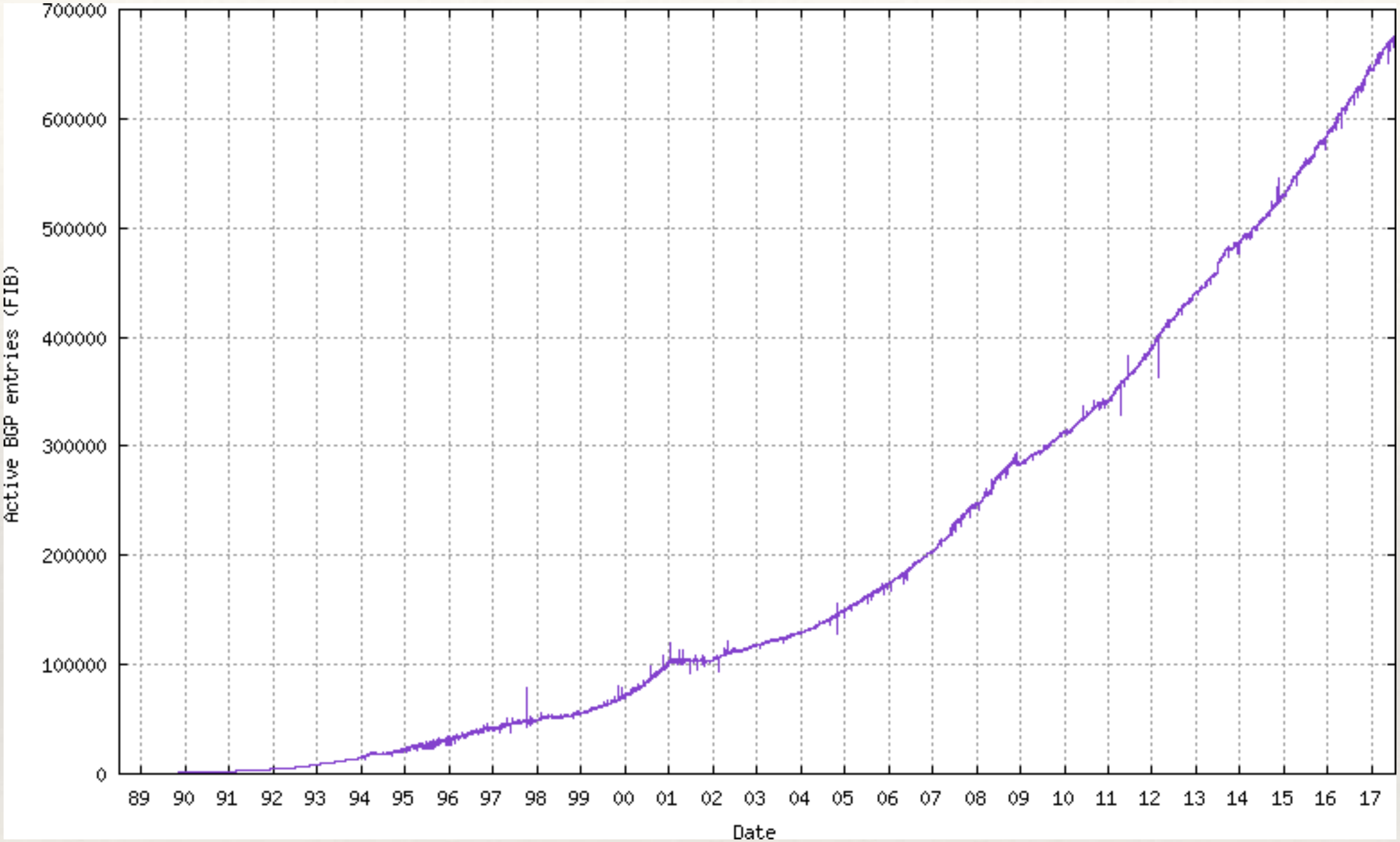
Trie storing prefixes

One of possible implementations, we do not assume it.




**Rule lookup = start from the root and proceed as deep as possible.**

# The problem: FIB size




FIB size at AS65000.
Report from `http://bgp.potaroo.net/as1221/`

- Many routers operate **at the edge of their memory capacity**.
- Upgrading memory expensive (specialized TCAM chips for fast lookups).

# Solution: FIB offloading

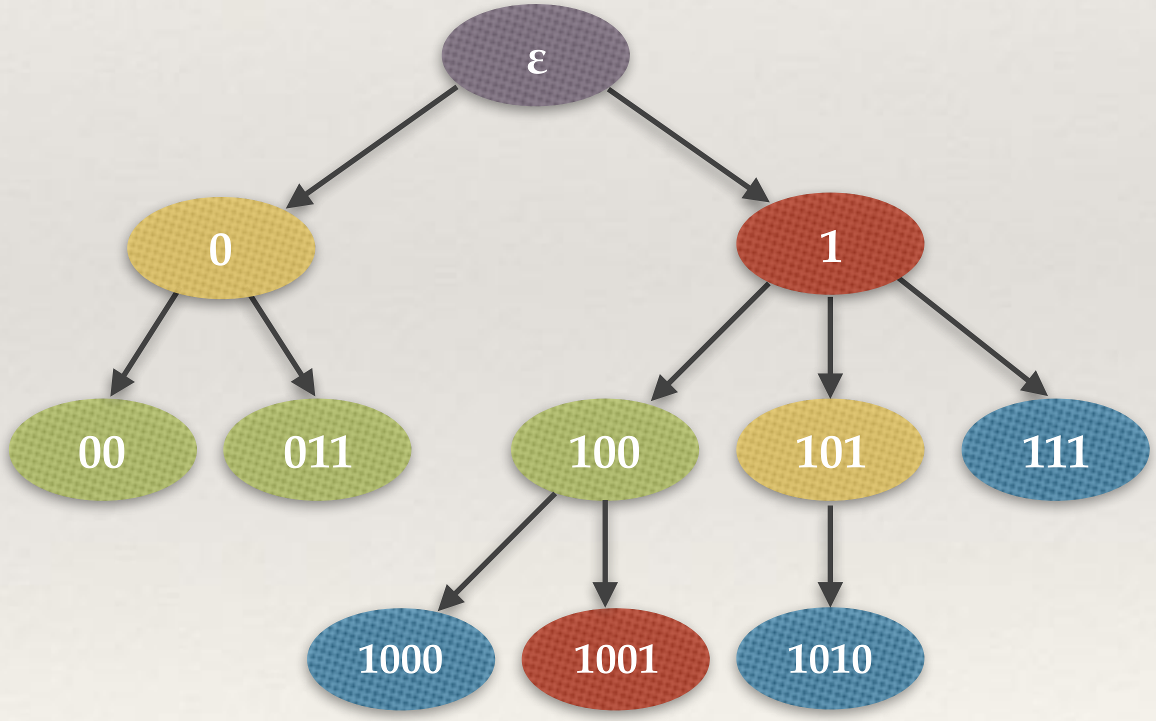
**Idea: store only subset of rules in the router**

- What rules should be kept?
- How to handle remaining rules?

**Next-slide setup proposed and tested experimentally by**
**Kim, Caesar, Gerber, Rexford (PAM'09); Sarrar, Uhlig, Feldmann, Sherwood, Huang, (SIGCOMM '12); Liu, Lehman, Wang (Comp. Netw. '15); Katta, Alipourfard, Rexford, Walker (SOSR '16); …**

# FIB offloading setup

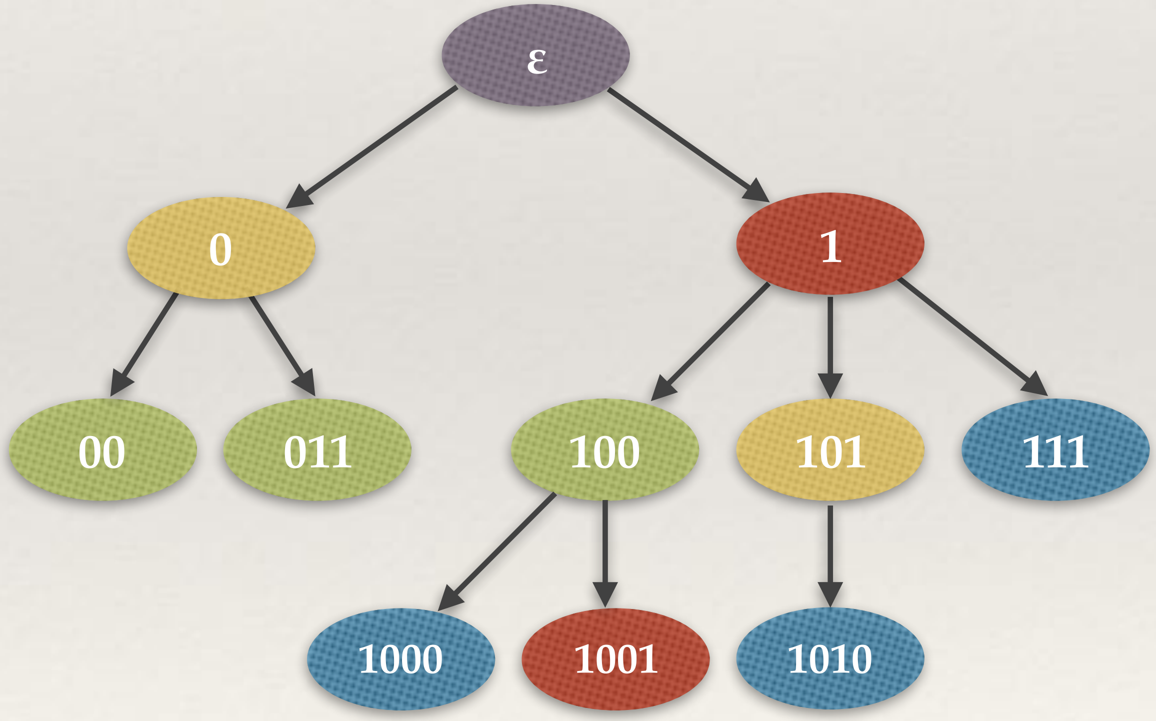
**router:** small and fast memory



# FIB offloading setup

**controller:** large and slow memory

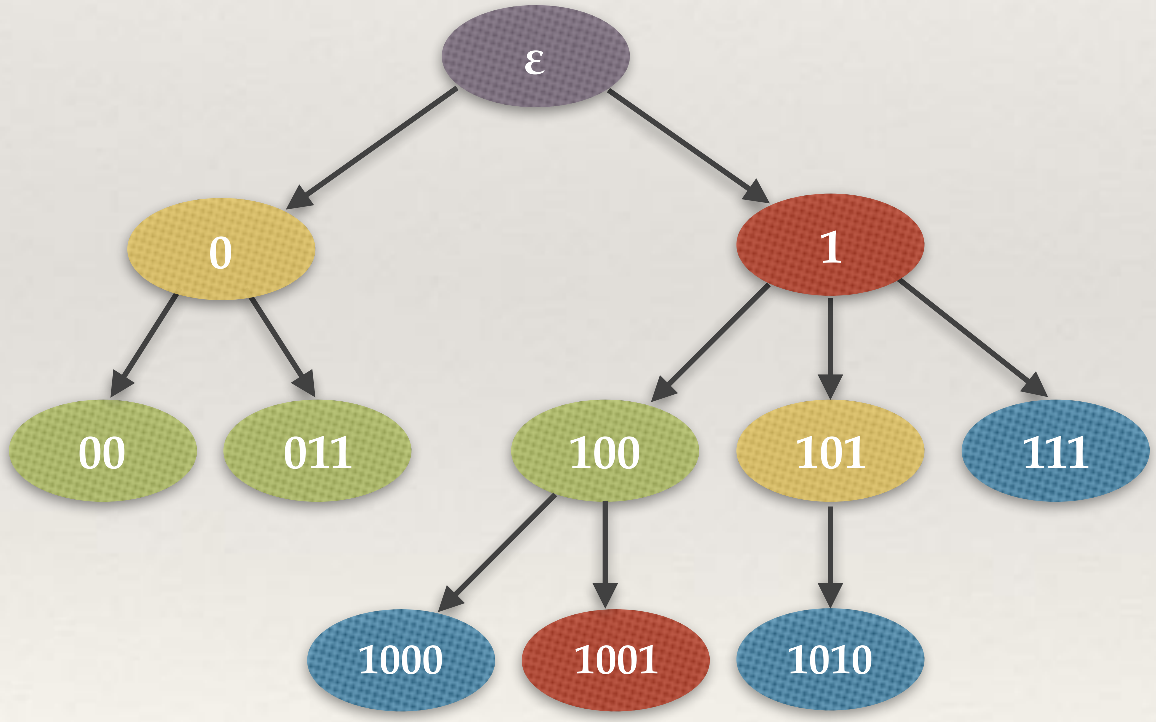
**router:** small and fast memory



# FIB offloading setup

**controller:** large and slow memory

**router:** small and fast memory



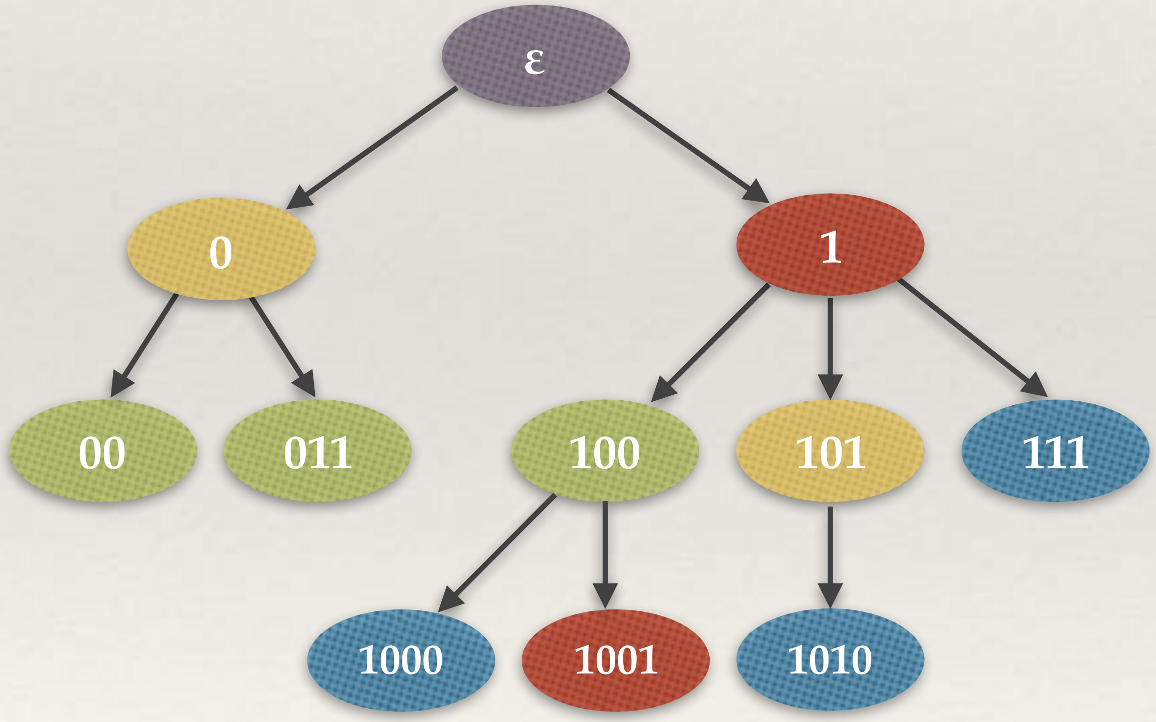

stores whole FIB

# FIB offloading setup

rule updates

**controller:** large and slow memory

**router:** small and fast memory

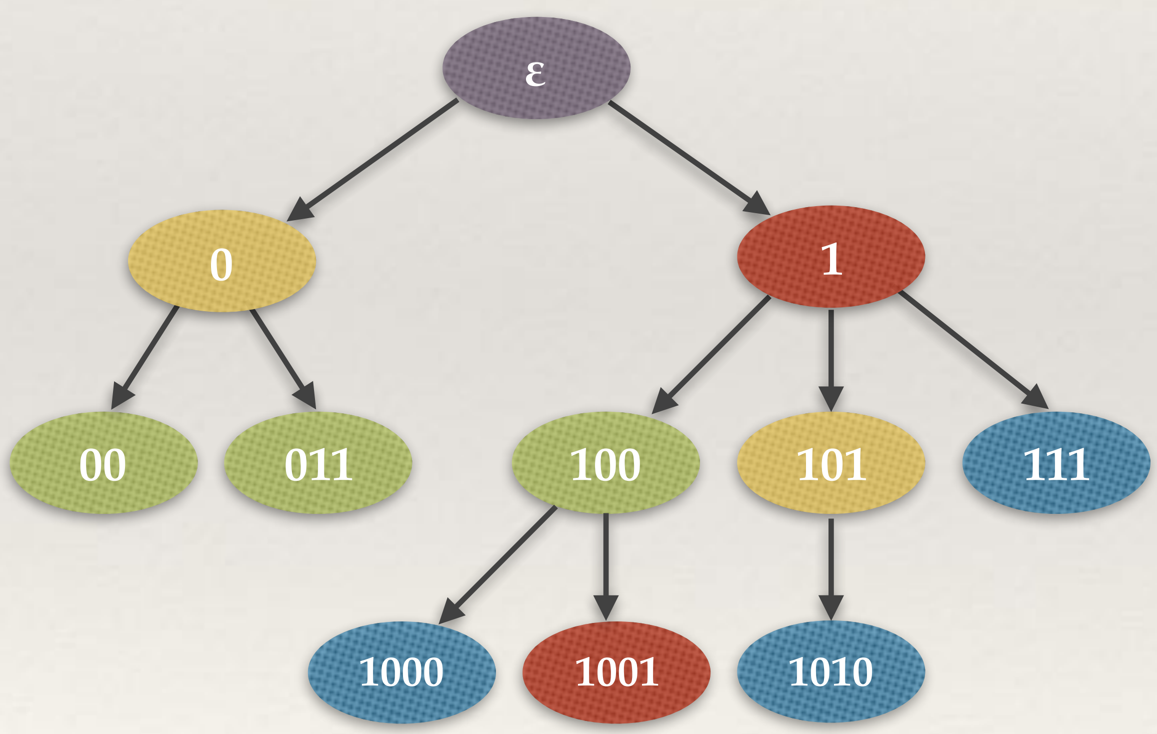



stores whole FIB

# FIB offloading setup


**controller:** large and slow memory

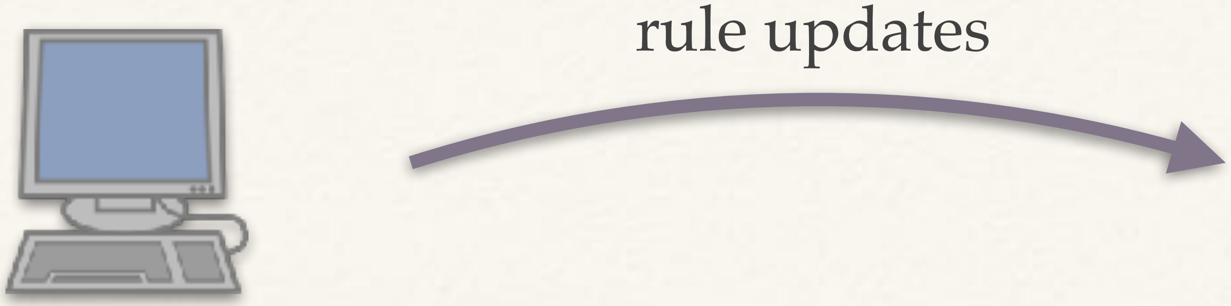
**router:** small and fast memory




stores whole FIB

stores chosen ``bottom part” of FIB

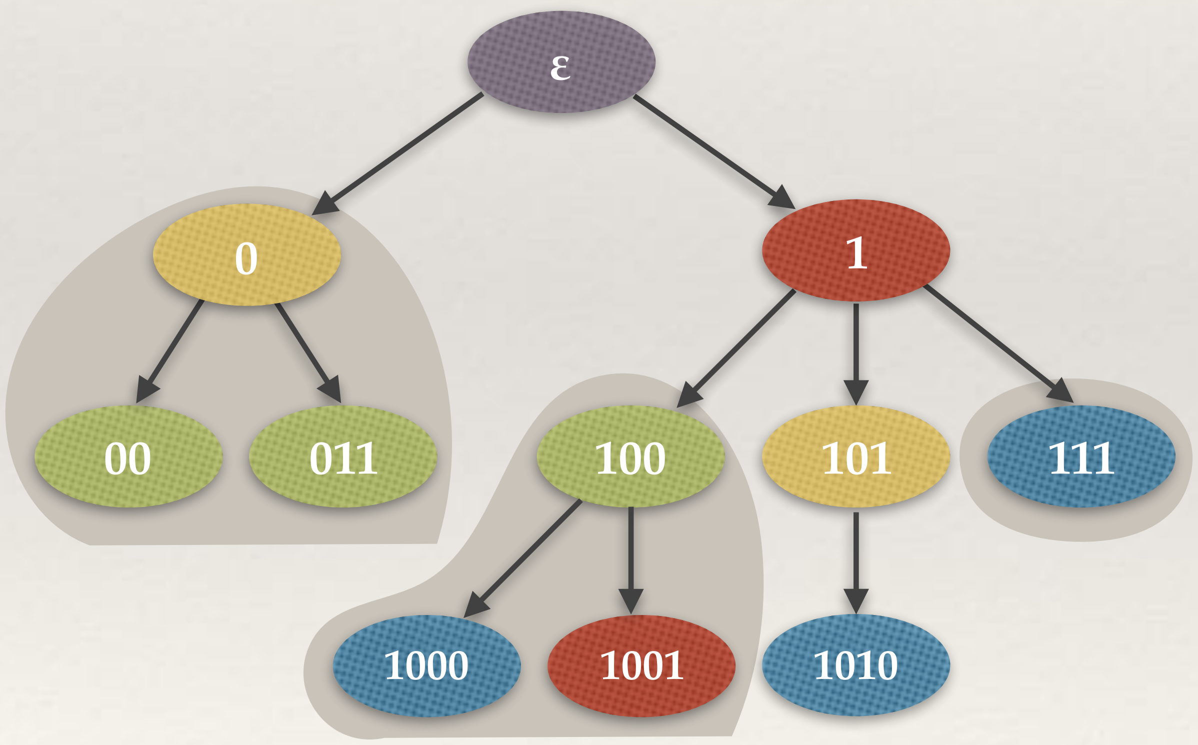
# FIB offloading setup



rule updates

**controller:** large and slow memory

**router:** small and fast memory



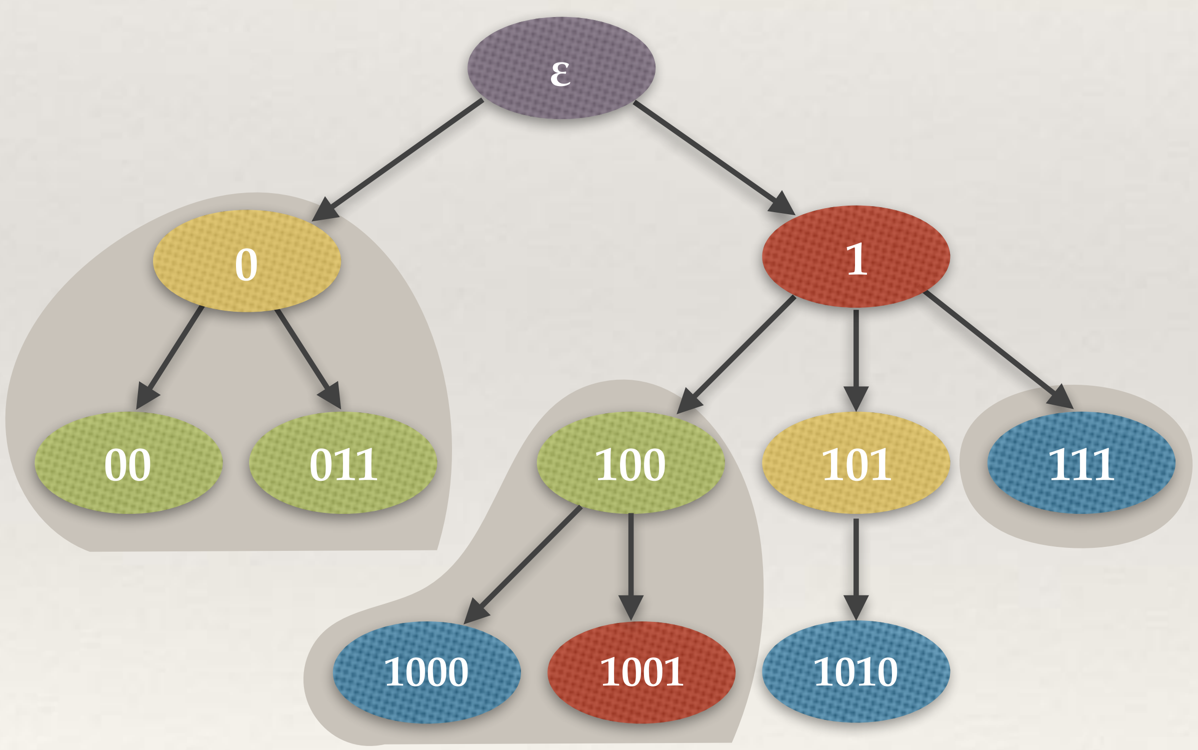

stores whole FIB

stores chosen ``bottom part'' of FIB

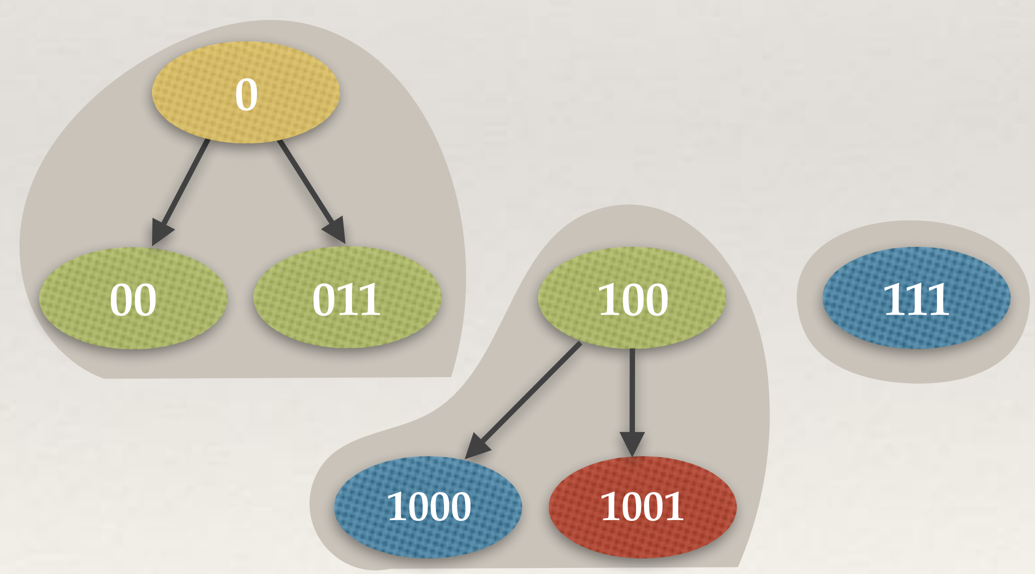
# FIB offloading setup


**controller:** large and slow memory

**router:** small and fast memory




stores whole FIB

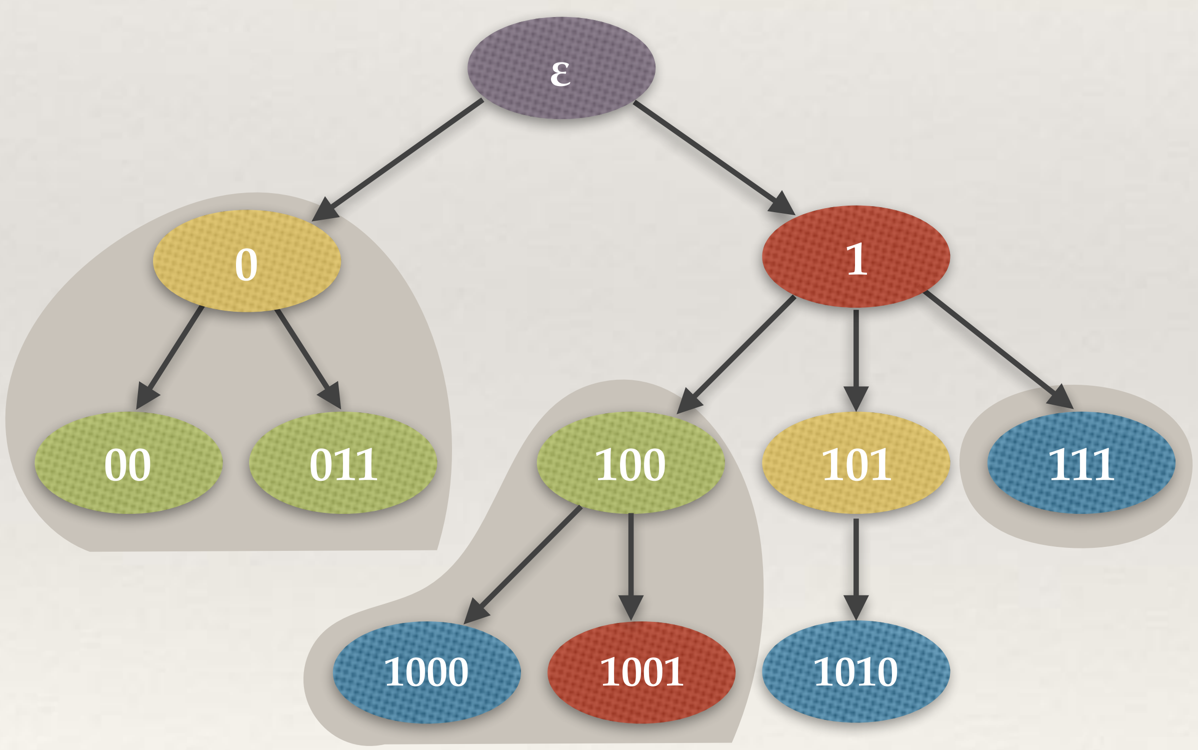



stores chosen ``bottom part'' of FIB

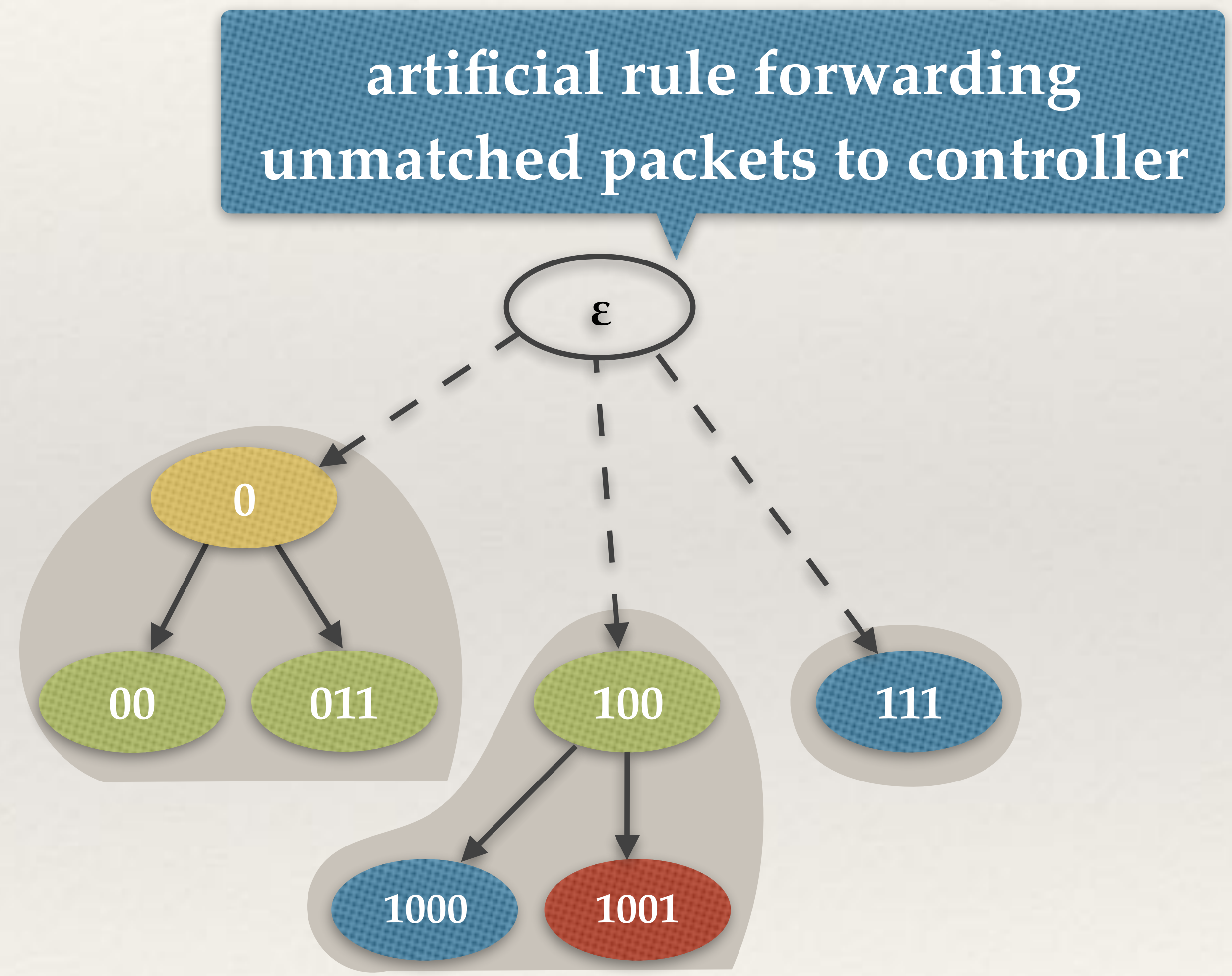
# FIB offloading setup


**controller:** large and slow memory

**router:** small and fast memory




stores whole FIB

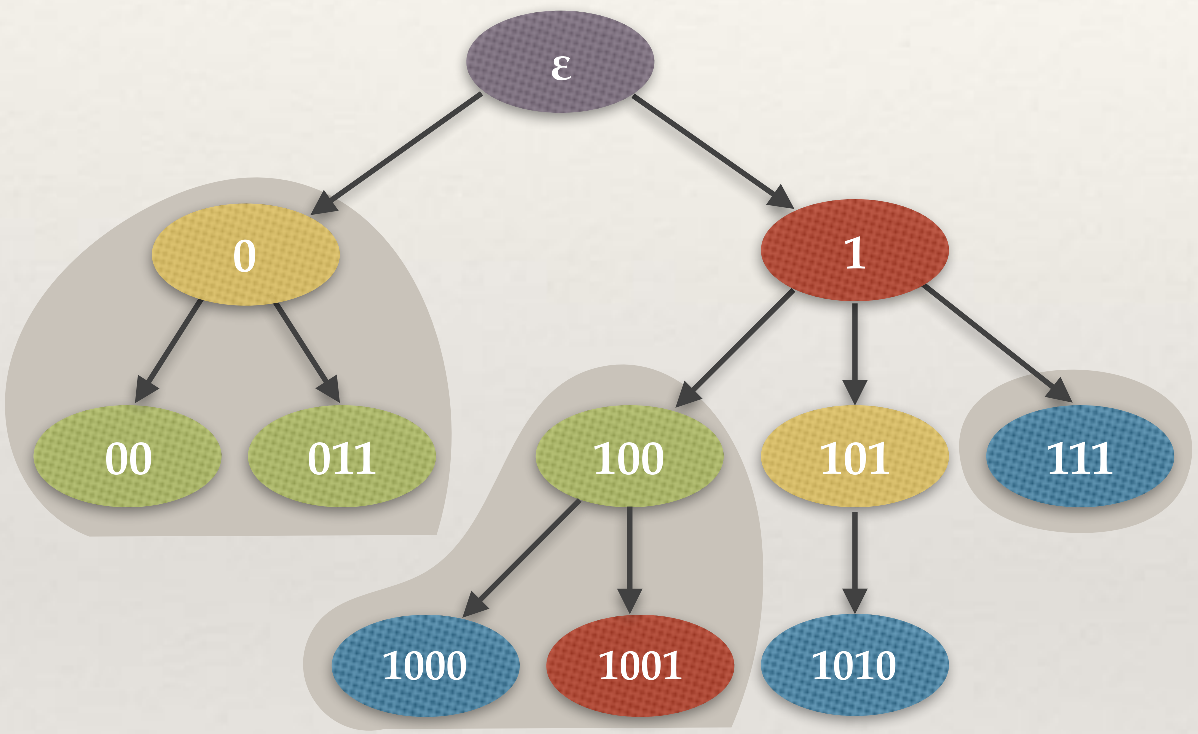


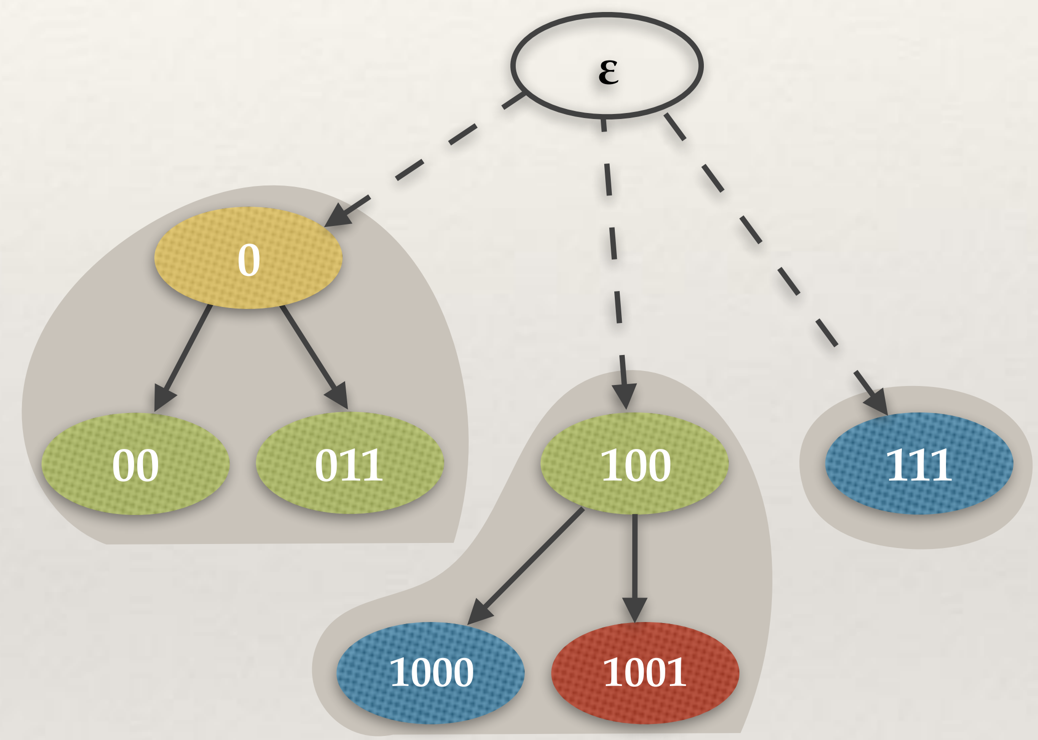

stores chosen ``bottom part'' of FIB

# Arriving packet (case 1)

**controller:** stores whole FIB

**router:** stores chosen ``bottom part'' of FIB



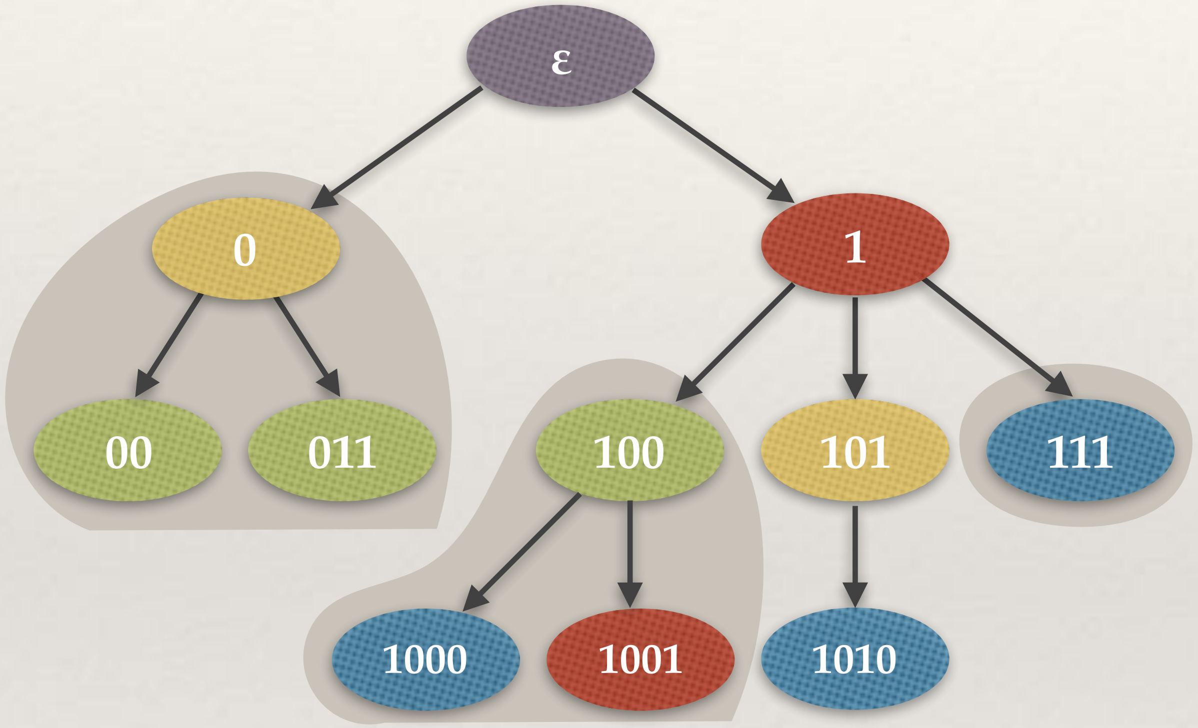

**If packet was matched by BOTTOM rules (e.g., destination = 111111…)**

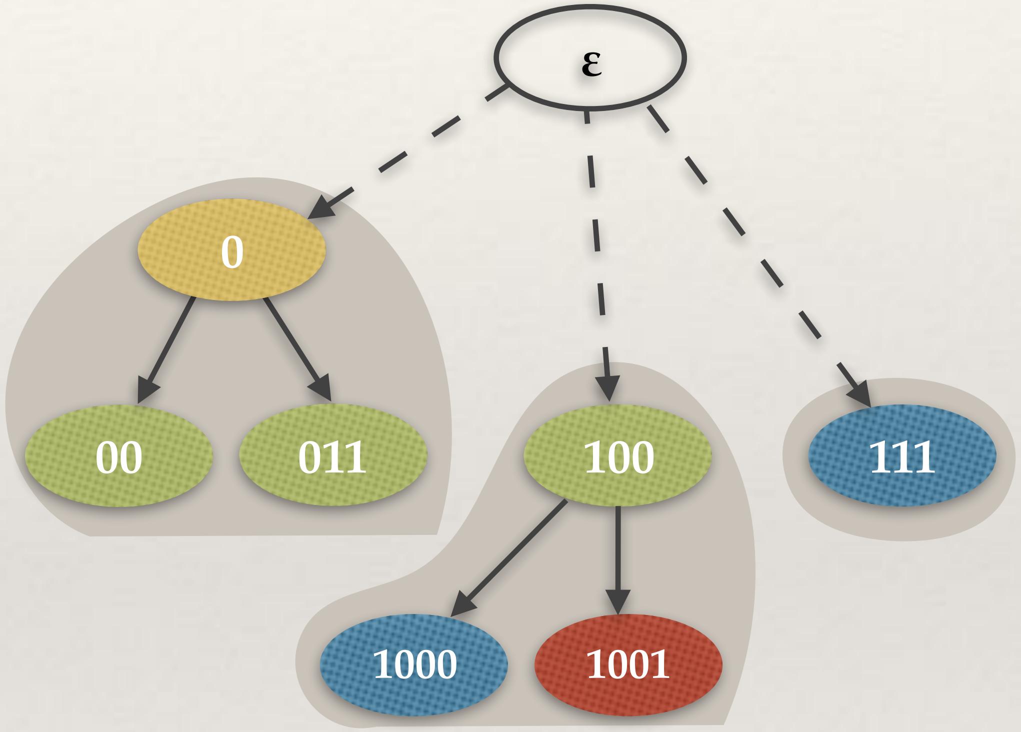
… it is still matched by bottom rules and processed at router.

# Arriving packet (case 2)

**controller:** stores whole FIB

**router:** stores chosen ``bottom part'' of FIB




**If packet was matched by TOP rules (e.g., destination = 1100000…)**

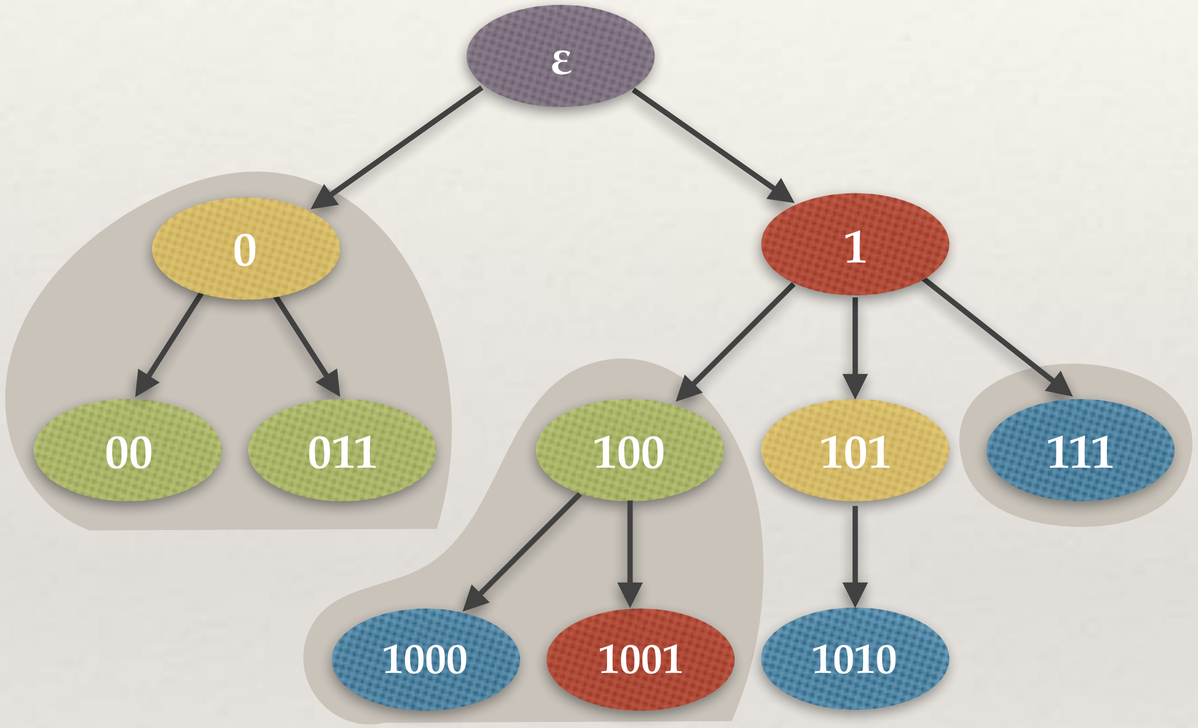
… it is matched by default route, and forwarded to the controller.

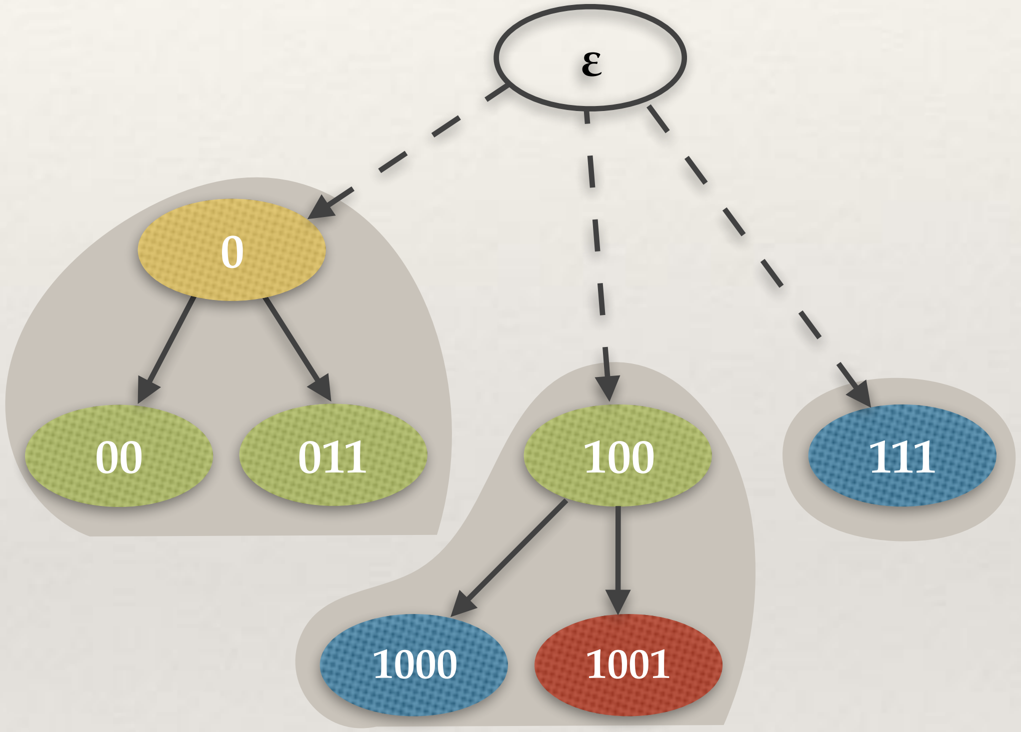
# Arriving packet (case 2)

Controller finds a port and returns TAGGED packet to router

**controller:** stores whole FIB

**router:** stores chosen ``bottom part'' of FIB




**If packet was matched by TOP rules (e.g., destination = 1100000…)**

… it is matched by default route, and forwarded to the controller.

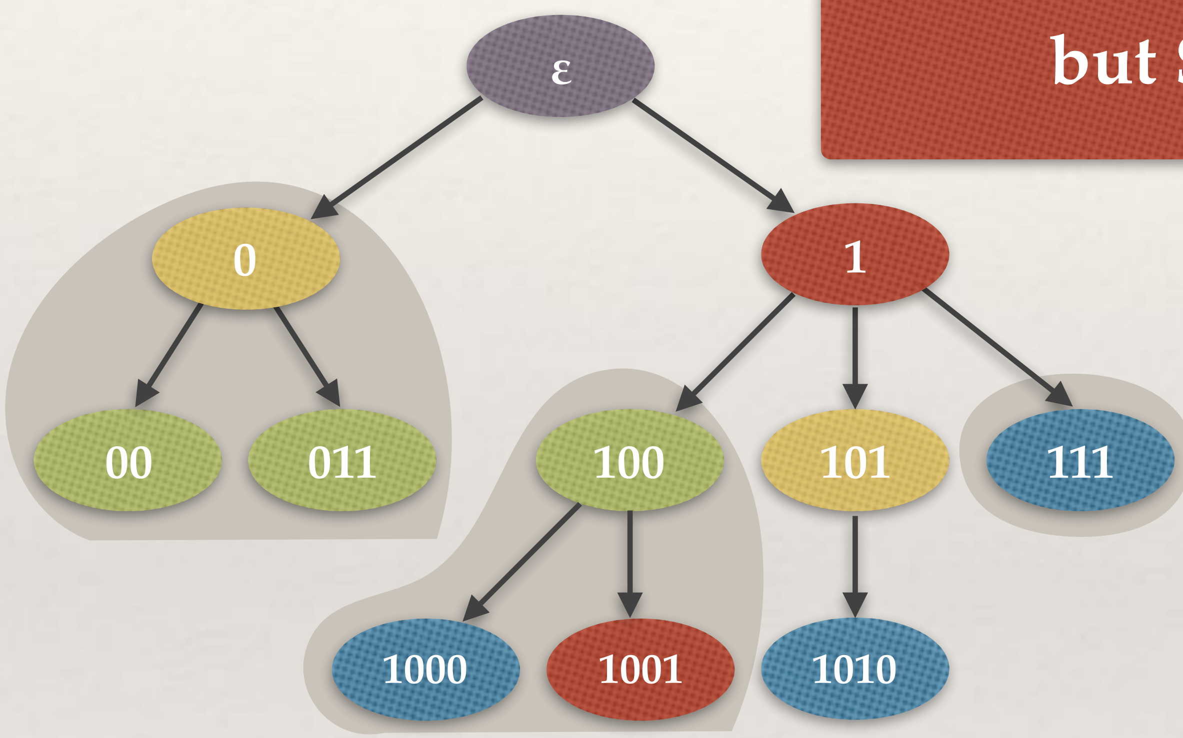
# Arriving packet (case 2)

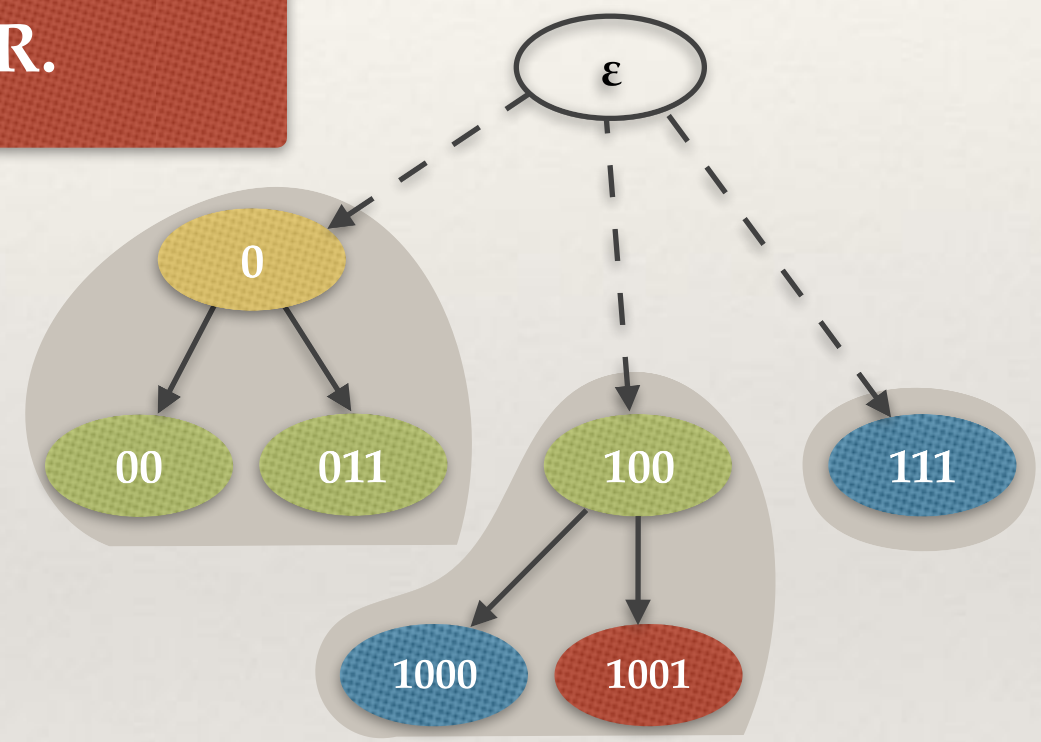
**Controller finds a port and returns TAGGED packet to router**

**controller:** stores whole FIB

s chosen ``bottom part'' of FIB

**Same forwarding behavior but SLOWER.**




**If packet was matched by TOP rules (e.g., destination = 1100000…)**

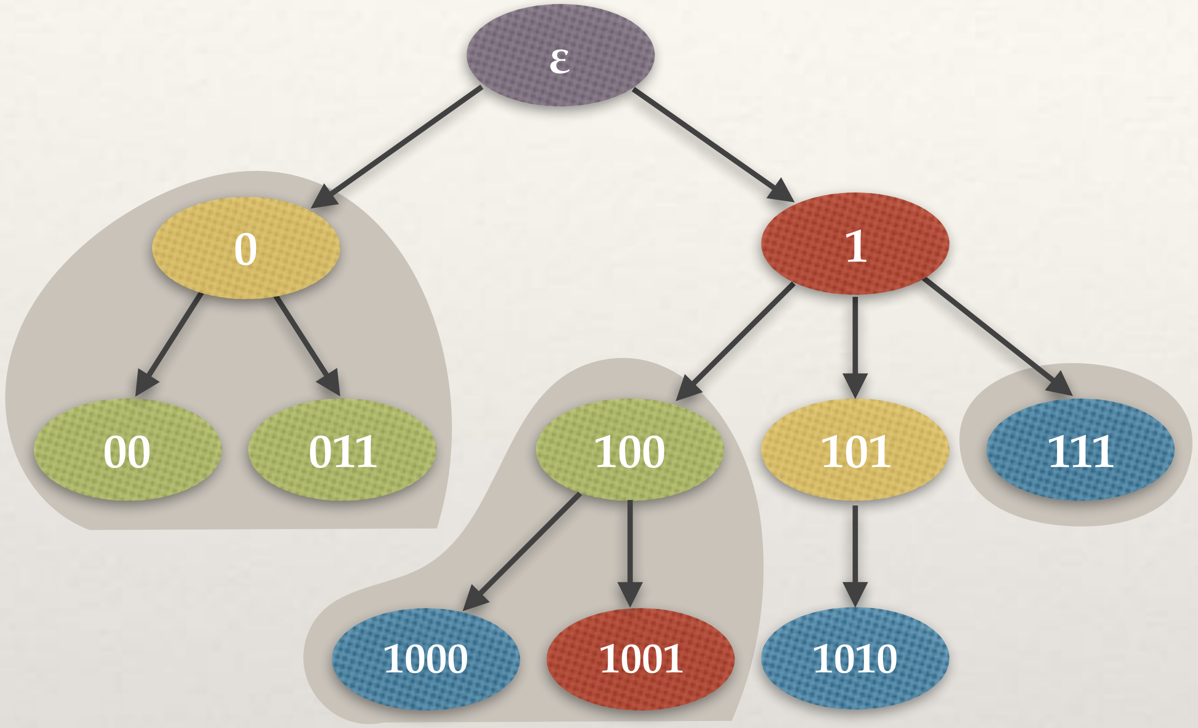
… it is matched by default route, and forwarded to the controller.

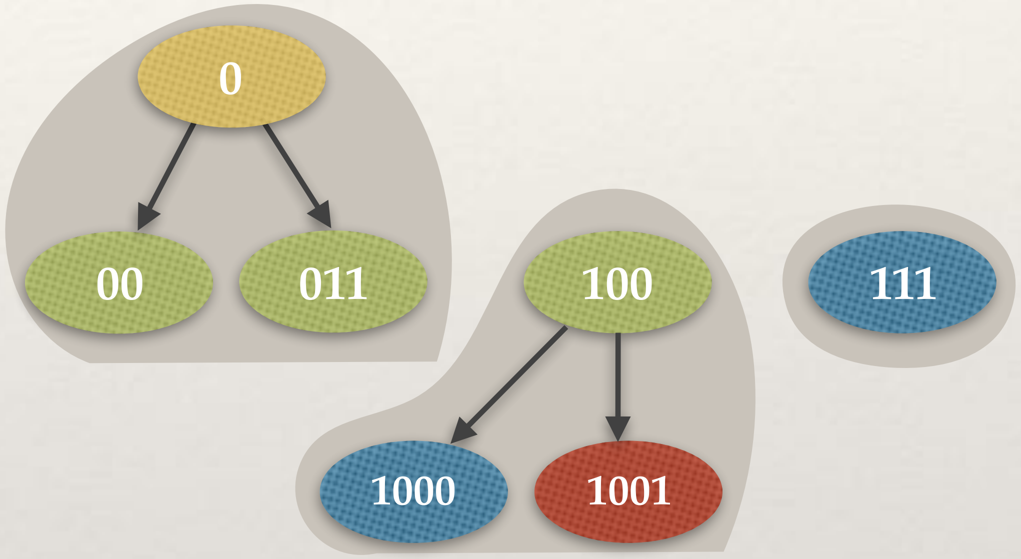
# Abstraction: caching

# Tree caching

**controller:**
tree $T$ of all items

**router (cache):**
subforest of $T$, at most $k$ items



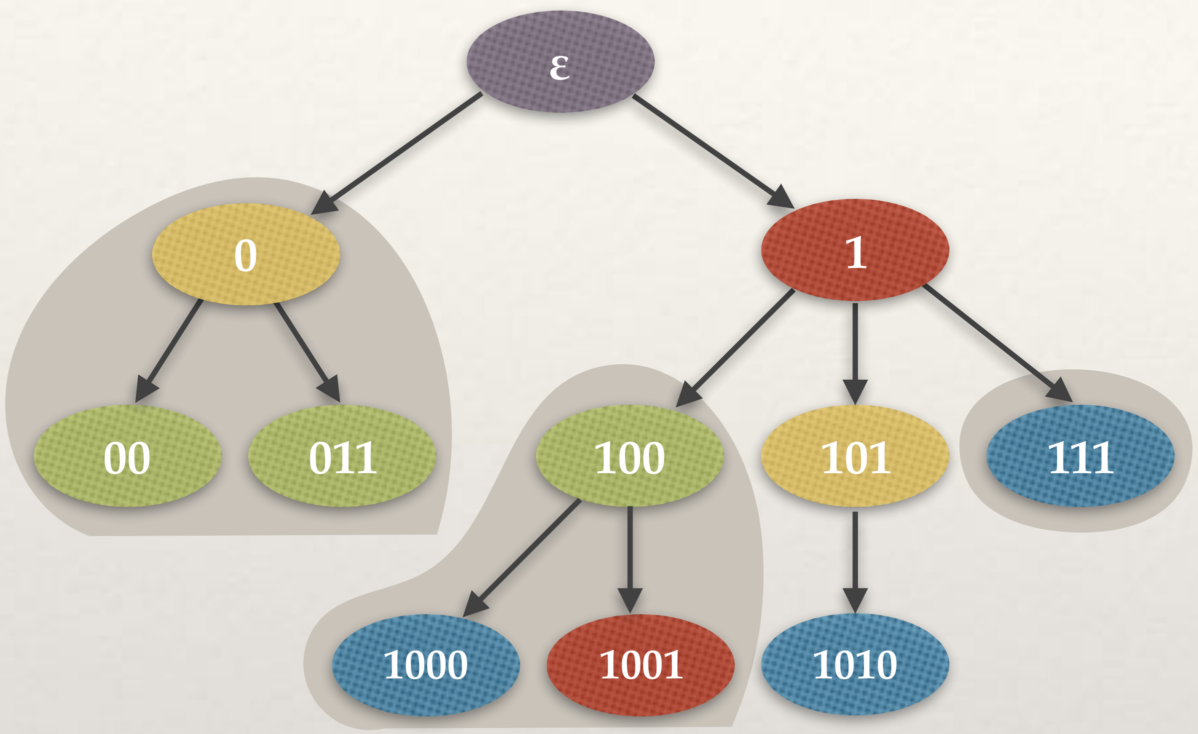



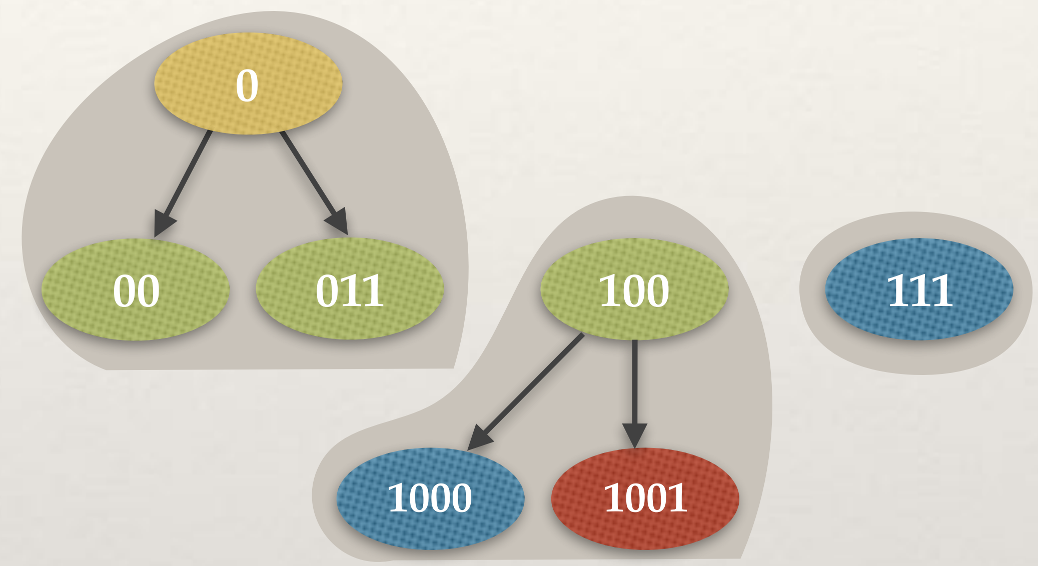

- **Input** = sequence of requests to items.
- Cache hit → cost 0, cache miss → cost 1.
- Changing cache (single item fetch or eviction) → cost $\alpha \geq 1$.

# Tree caching

**controller:**
tree $T$ of all items

**router (cache):**
subforest of $T$, at most $k$ items



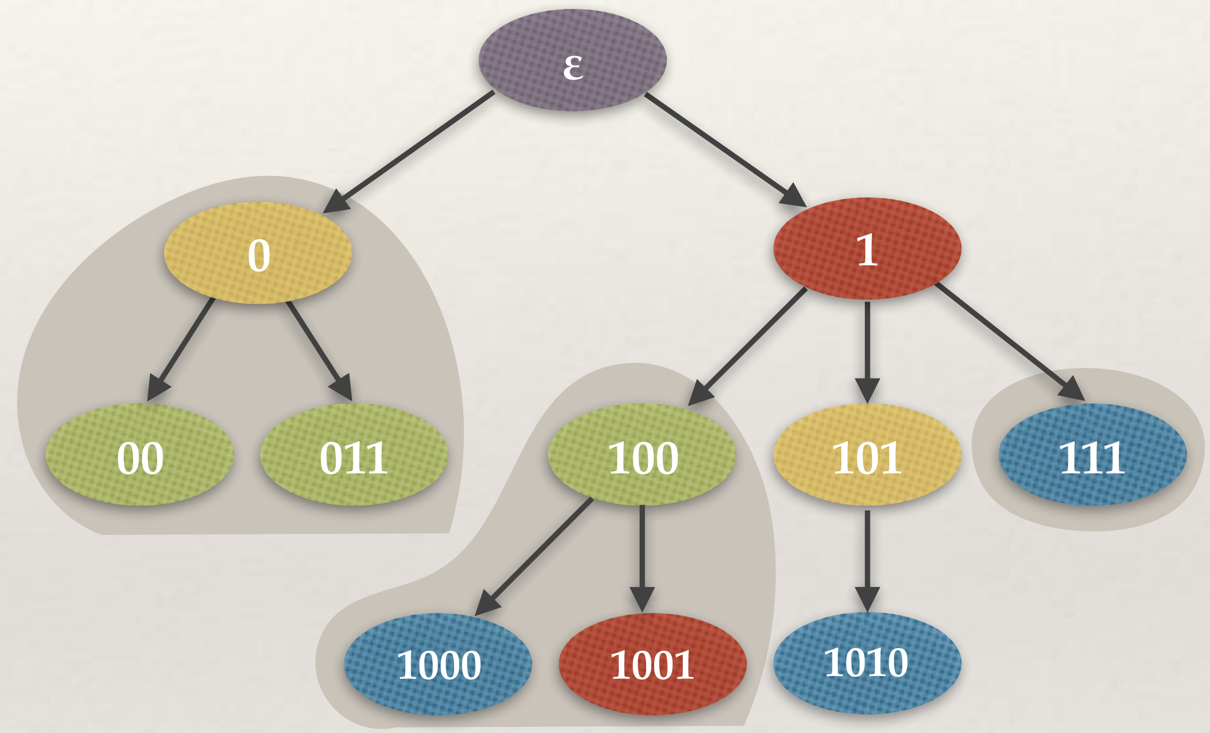




- **Input** = sequence of requests to items.
- Cache hit → cost 0, cache miss → cost 1.
- Changing cache (single item fetch or eviction) → cost $\alpha \geq 1$.

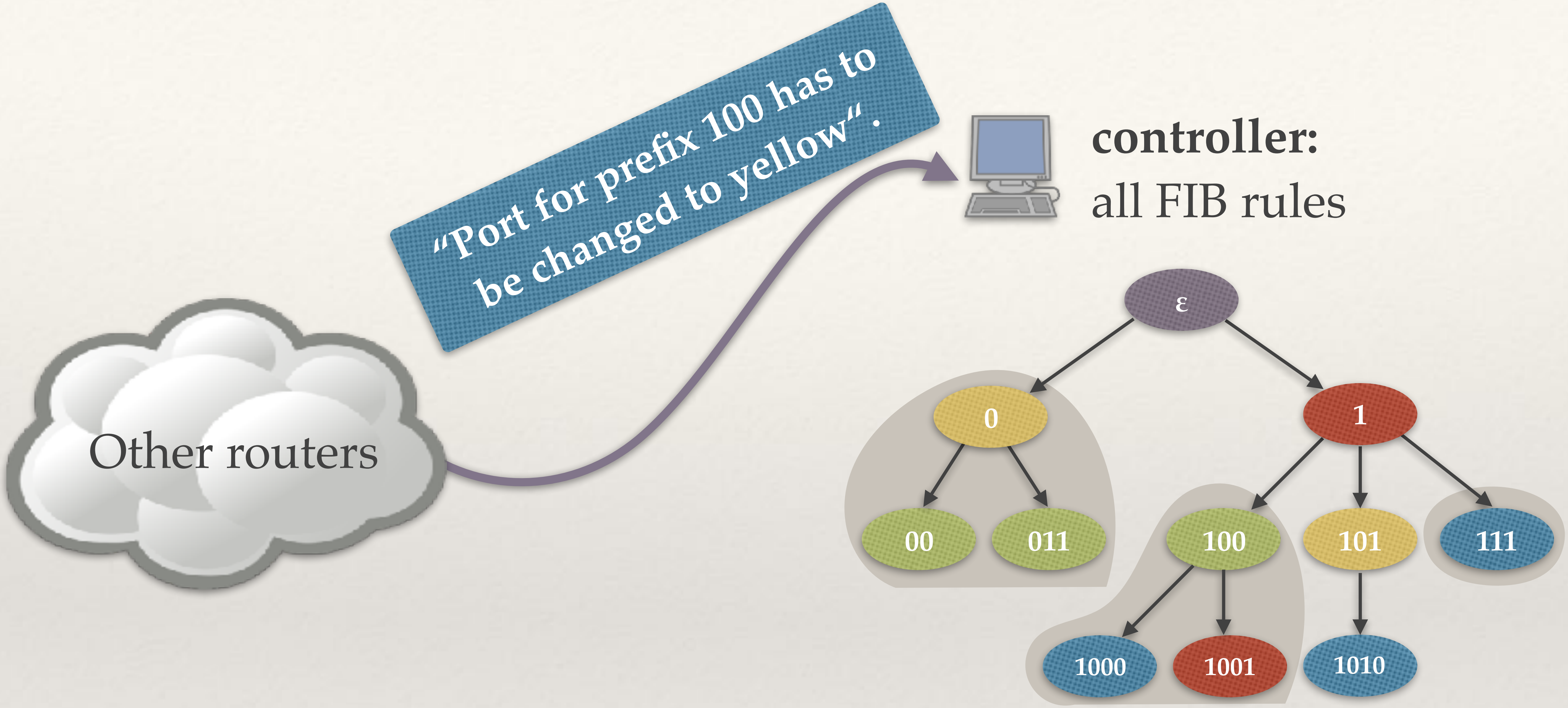
**"Caching with bypassing and tree dependencies between items"**

# External updates to rule

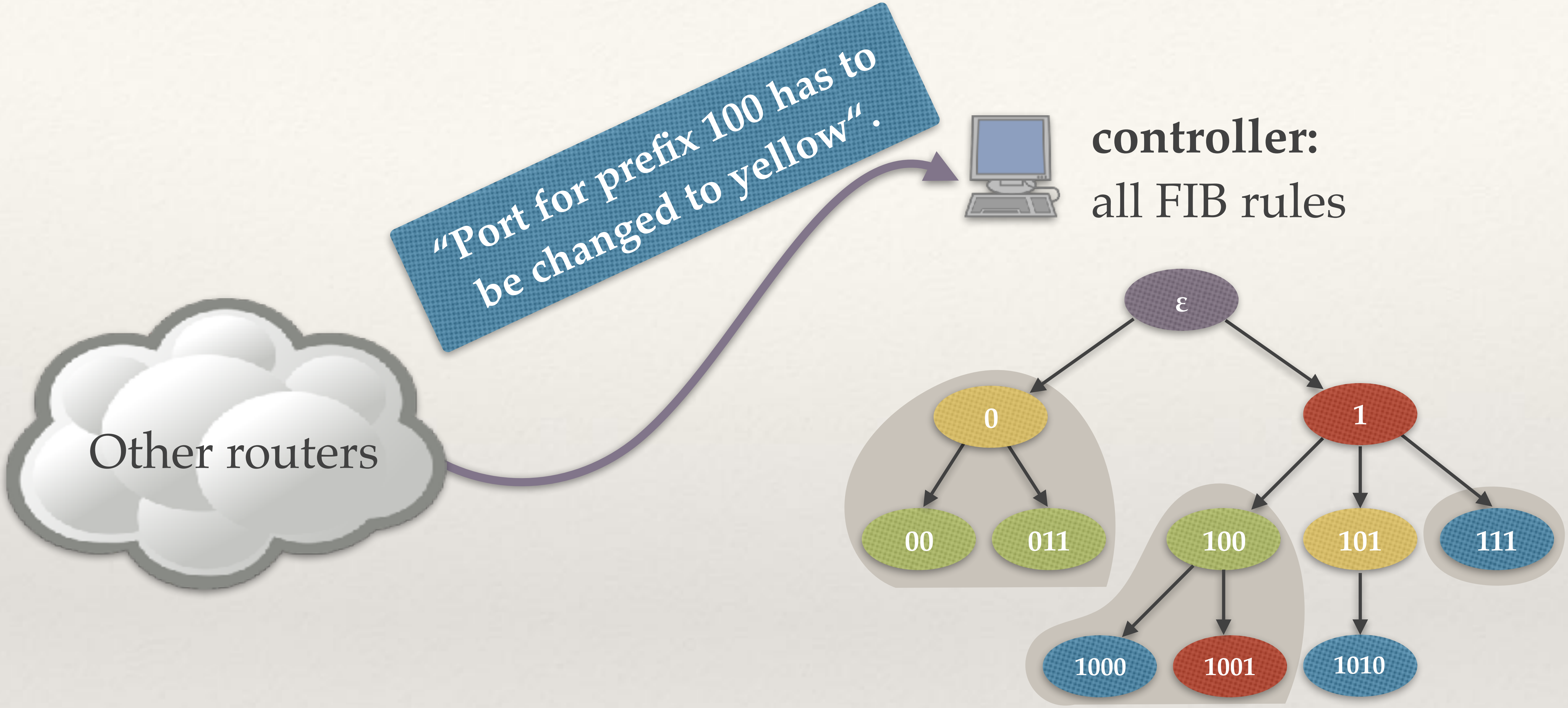

**controller:**
all FIB rules



# External updates to rule



# External updates to rule

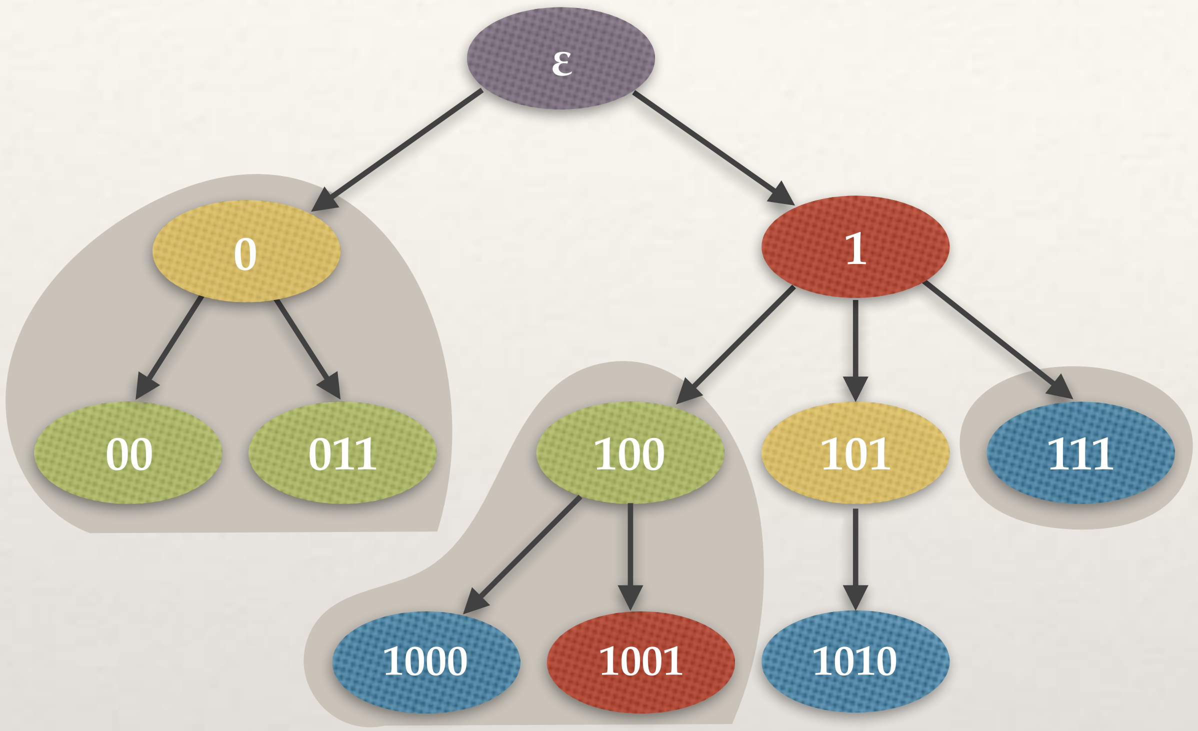


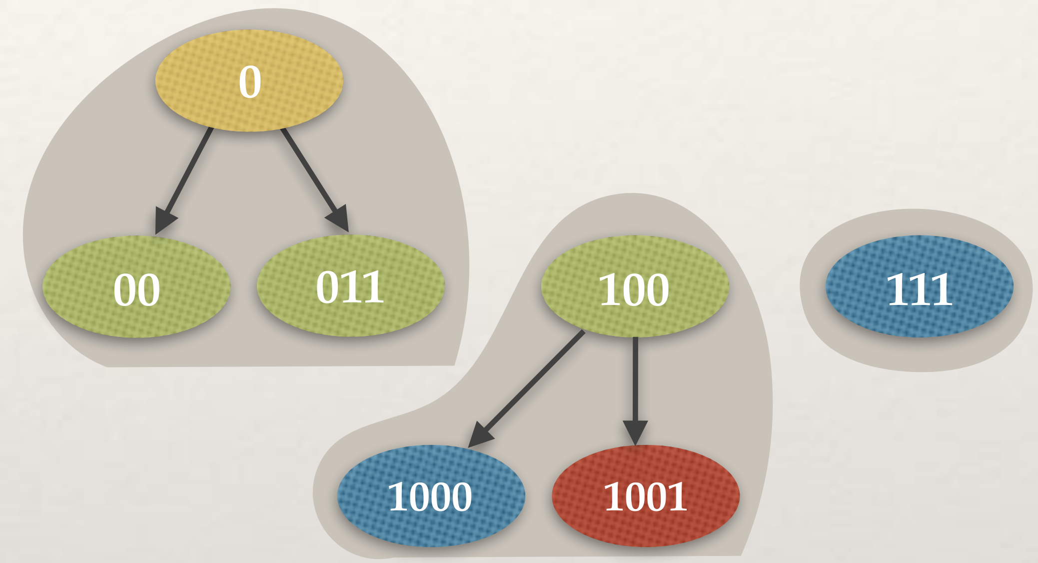

- If the updated rule is also stored at router (*is cached*), it needs to be updated → cost $\alpha \geq 1$.

# Tree caching (final version)

**controller:**
tree $T$ of all items

**router (cache):**
sub-forest of $T$, at most $k$ items



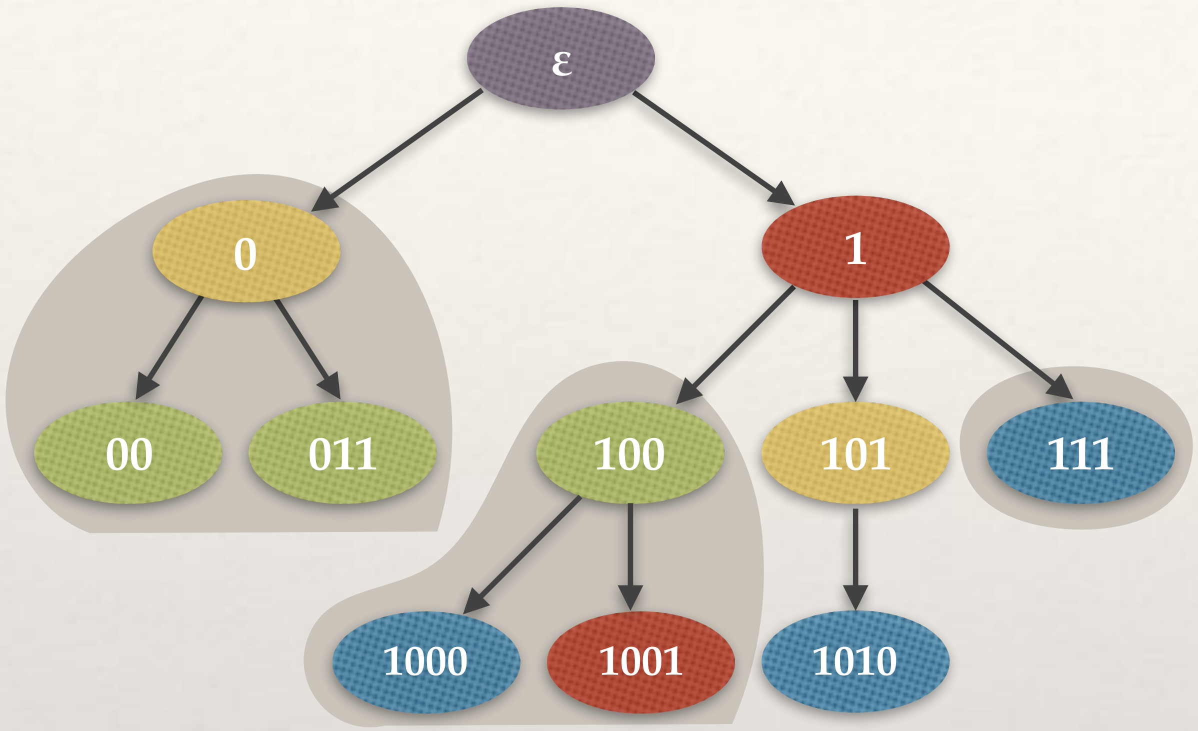



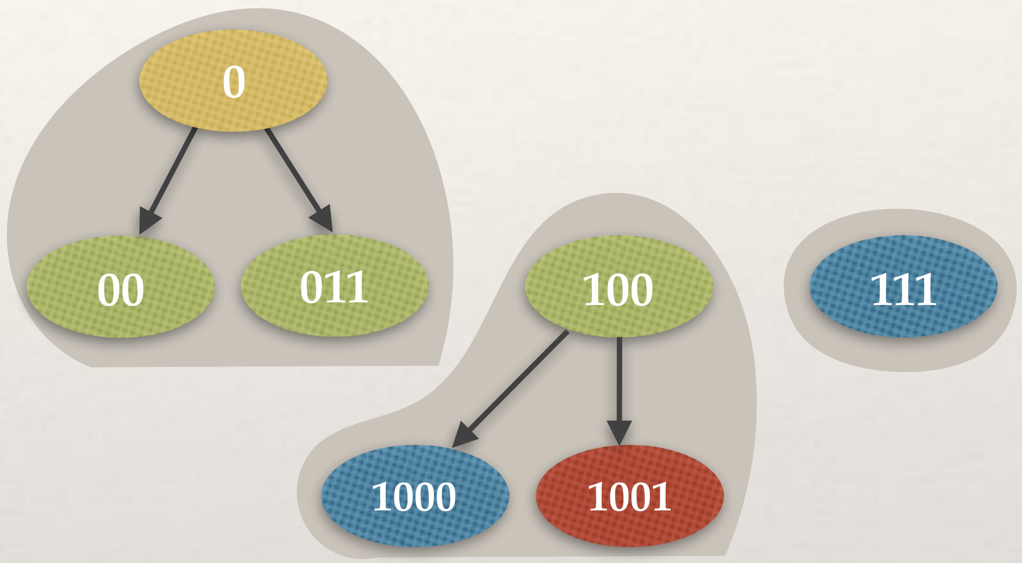

- **Input** = sequence of requests to items
  - *Positive request*: cost 1 iff item **is not cached**.
  - *Negative request:* cost 1 iff item **is cached**.
- Changing cache (single item fetch or eviction) → cost $\alpha \geq 1$.

# Tree caching (final version)

**controller:**
tree $T$ of all items

**router (cache):**
sub-forest of $T$, at most $k$ items




- **Input** = sequence of requests to items
  - ***Positive request***: cost 1 iff item **is not cached**.
  - ***Negative request:*** cost 1 iff item **is cached**.
- Changing cache (single item fetch or eviction) → cost $\alpha \geq 1$.

**Actual costs can be simulated by these.**

# Algorithm

# Our results

**Performance measure**

- Online problem.
- Goal: minimize the competitive ratio ($\max_I$ ALG($I$) / OPT($I$)).

# Our results

**Performance measure**

- Online problem. has cache of size $k_{\mathrm{ALG}}$
- Goal: minimize the competitive ratio ($\max_I$ ALG($I$) / OPT($I$)).

# Our results

**Performance measure**

has cache of size $k_{\mathrm{OPT}} \leq k_{\mathrm{ALG}}$

has cache of size $k_{\mathrm{ALG}}$

- Online problem.
- Goal: minimize the competitive ratio ($\max_I$ ALG($I$) / OPT($I$)).

# Our results

**Performance measure**

has cache of size $k_{OPT} \leq k_{ALG}$

has cache of size $k_{ALG}$

- Online problem.
- Goal: minimize the competitive ratio ($\max_I$ ALG($I$) / OPT($I$)).

**Tree caching**

- $O(\, k_{ALG} \,/\, (k_{ALG} - k_{OPT} + 1) * \mathit{height}(T)\,)$-competitive algorithm.
- Lower bound of $\Omega(\, k_{ALG} \,/\, (k_{ALG} - k_{OPT} + 1)\,)$.

# Our results

**Performance measure**

has cache of size $k_{\text{OPT}} \leq k_{\text{ALG}}$

has cache of size $k_{\text{ALG}}$

- Online problem.
- Goal: minimize the competitive ratio ($\max_I$ ALG($I$) / OPT($I$)).

**Tree caching**

- $O(\, k_{\text{ALG}} \,/\, (k_{\text{ALG}} - k_{\text{OPT}} + 1) * \mathit{height}(\mathrm{T})\,)$-competitive algorithm.
- Lower bound of $\Omega(\, k_{\text{ALG}} \,/\, (k_{\text{ALG}} - k_{\text{OPT}} + 1)\,)$.

By a reduction to caching problem

# This talk

**Algorithm for the infinite cache case**

- Captures core difficulty of the problem

# This talk

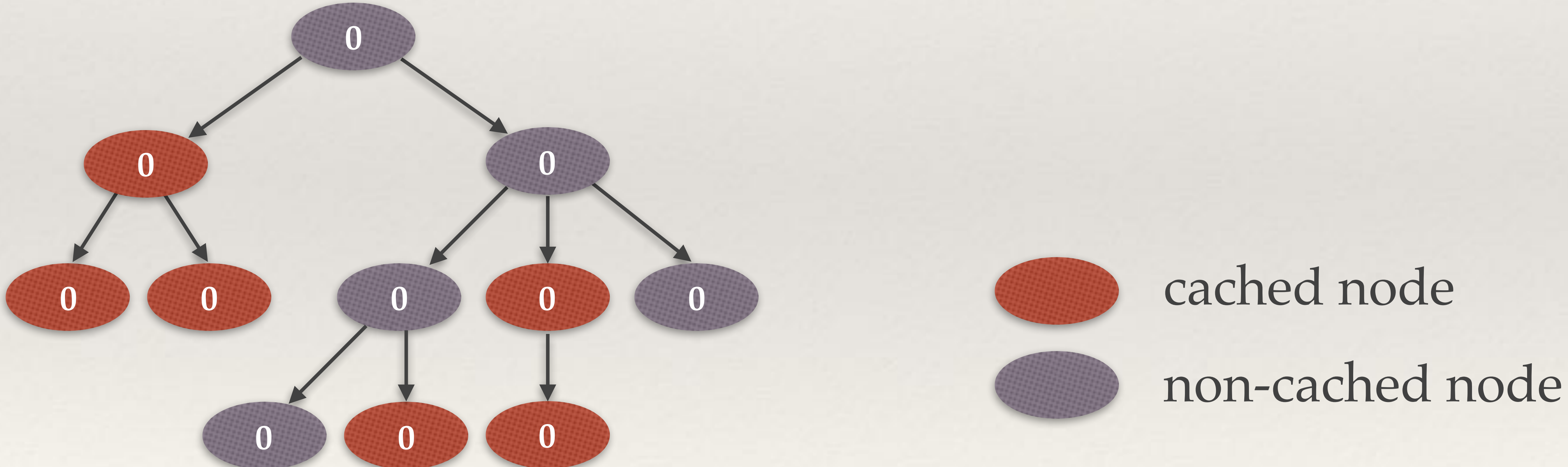
**Algorithm for the infinite cache case**

- Captures core difficulty of the problem
- Still non-trivial because of negative requests!
- *O(depth(T))*-competitive algorithm for this case.

# Our counter-based algorithm

- **Without loss of generality: all requests cost 1**
  (positive at non-cached nodes, negative at cached ones)

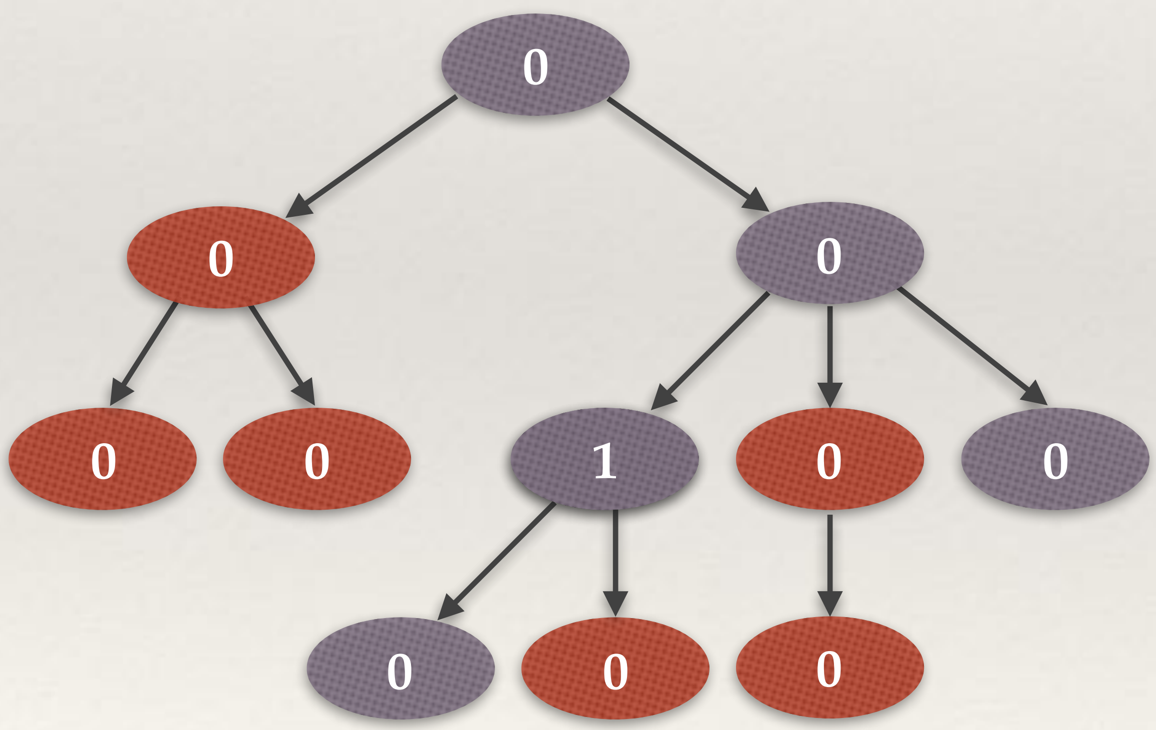
- **Counter** = number of requests at node from the last fetch / eviction of this node.

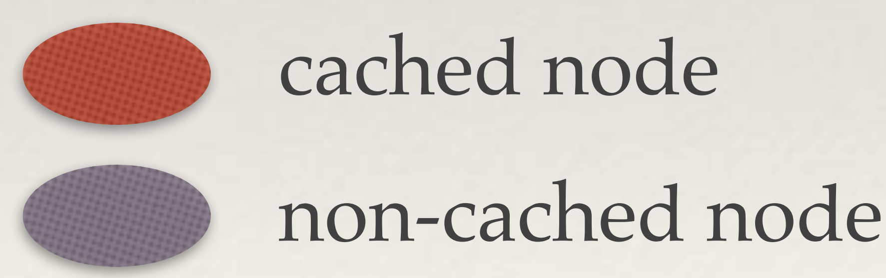



Assume: $\alpha = 2$.

# Our counter-based algorithm

- **Without loss of generality: all requests cost 1**
  (positive at non-cached nodes, negative at cached ones)
- **Counter** = number of requests at node from the last fetch / eviction of this node.

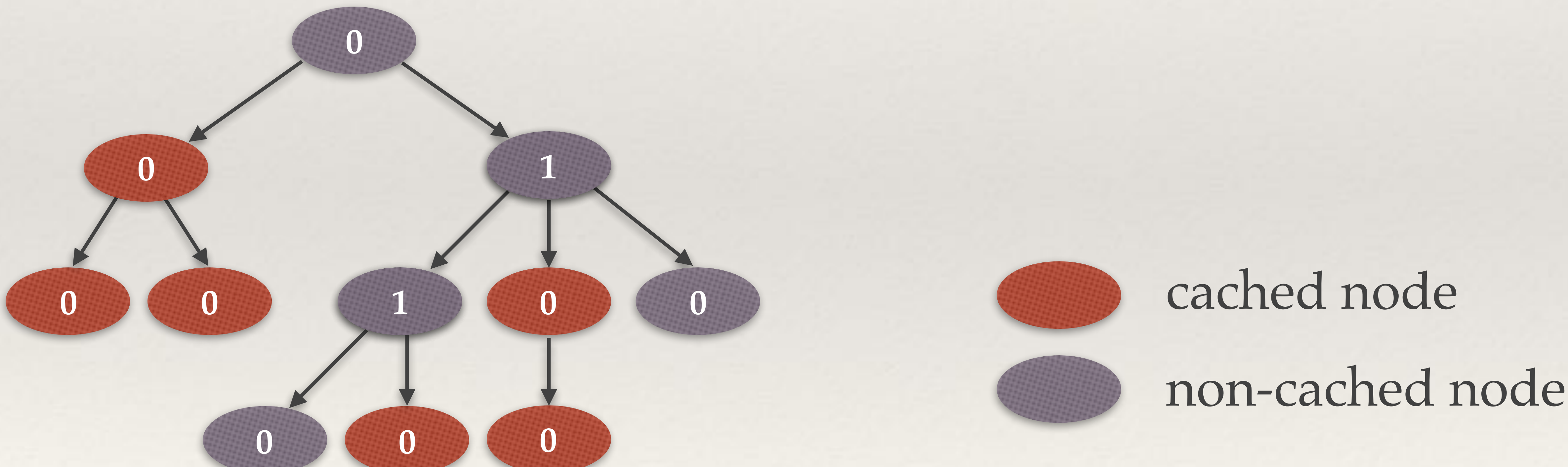



Assume: $\alpha = 2$.

# Our counter-based algorithm

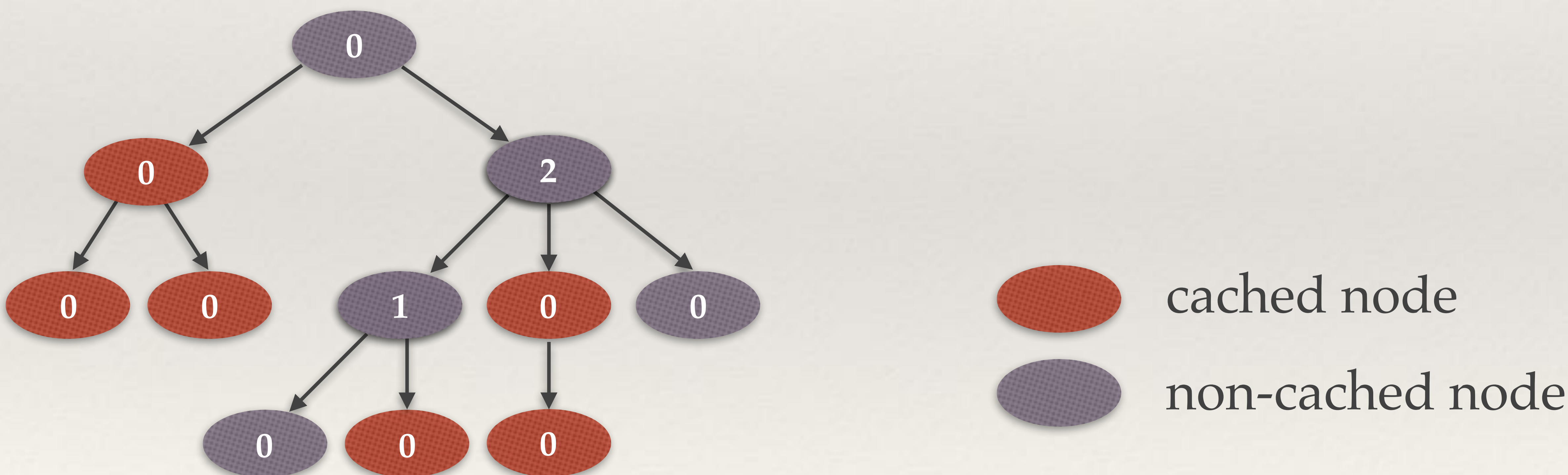
- **Without loss of generality: all requests cost 1** (positive at non-cached nodes, negative at cached ones)
- **Counter** = number of requests at node from the last fetch / eviction of this node.




Assume: $\alpha = 2$.

# Our counter-based algorithm

- **Without loss of generality: all requests cost 1** (positive at non-cached nodes, negative at cached ones)
- **Counter** = number of requests at node from the last fetch / eviction of this node.

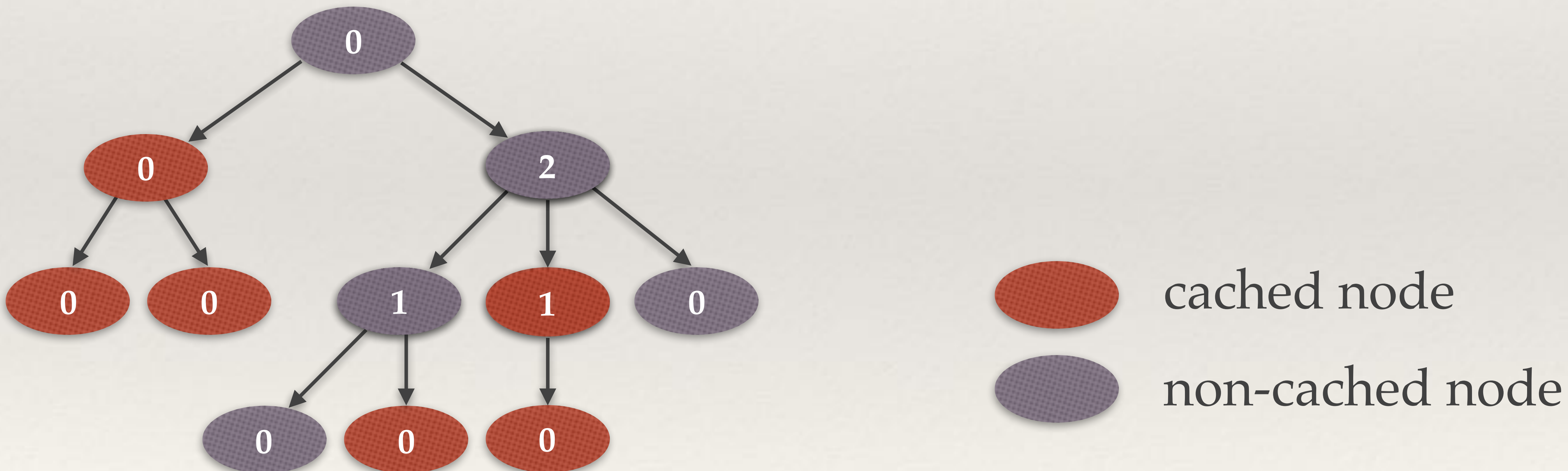



Assume: $\alpha = 2$.

# Our counter-based algorithm

- **Without loss of generality: all requests cost 1**
  (positive at non-cached nodes, negative at cached ones)

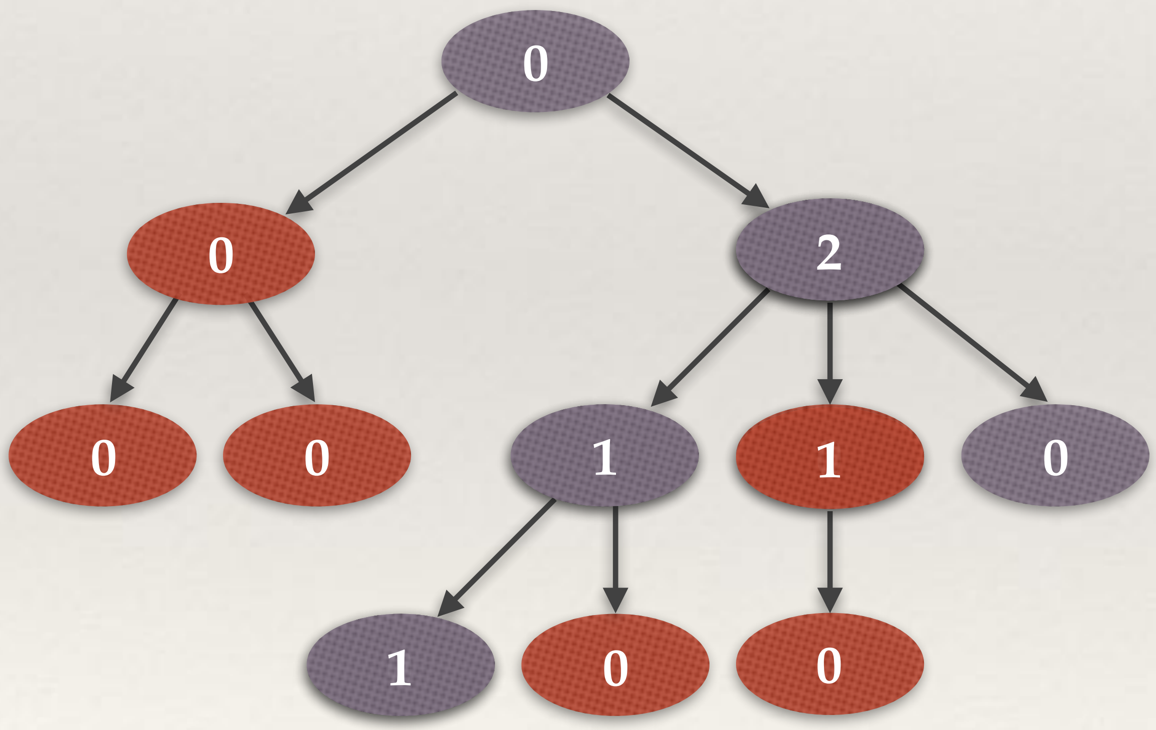
- **Counter** = number of requests at node from the last fetch / eviction of this node.

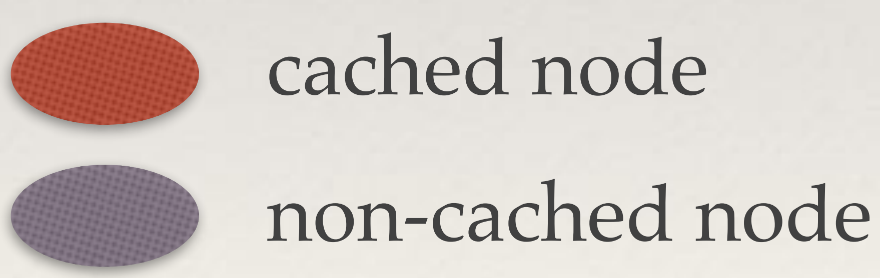



Assume: $\alpha = 2$.

# Our counter-based algorithm

- **Without loss of generality: all requests cost 1** (positive at non-cached nodes, negative at cached ones)
- **Counter** = number of requests at node from the last fetch / eviction of this node.

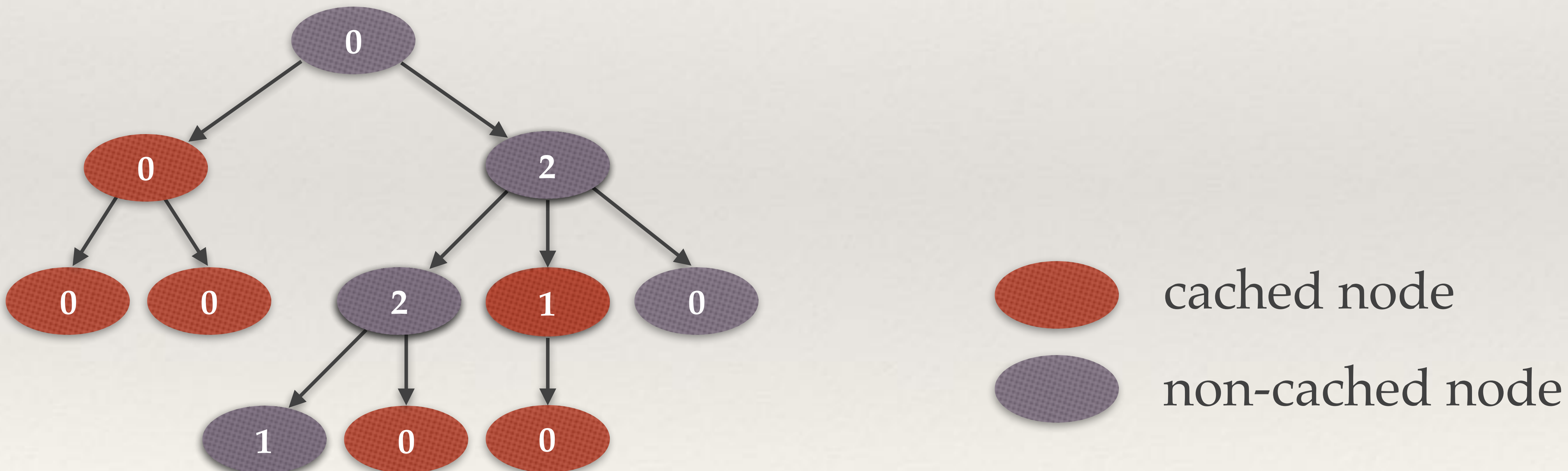



Assume: $\alpha = 2$.

# Our counter-based algorithm

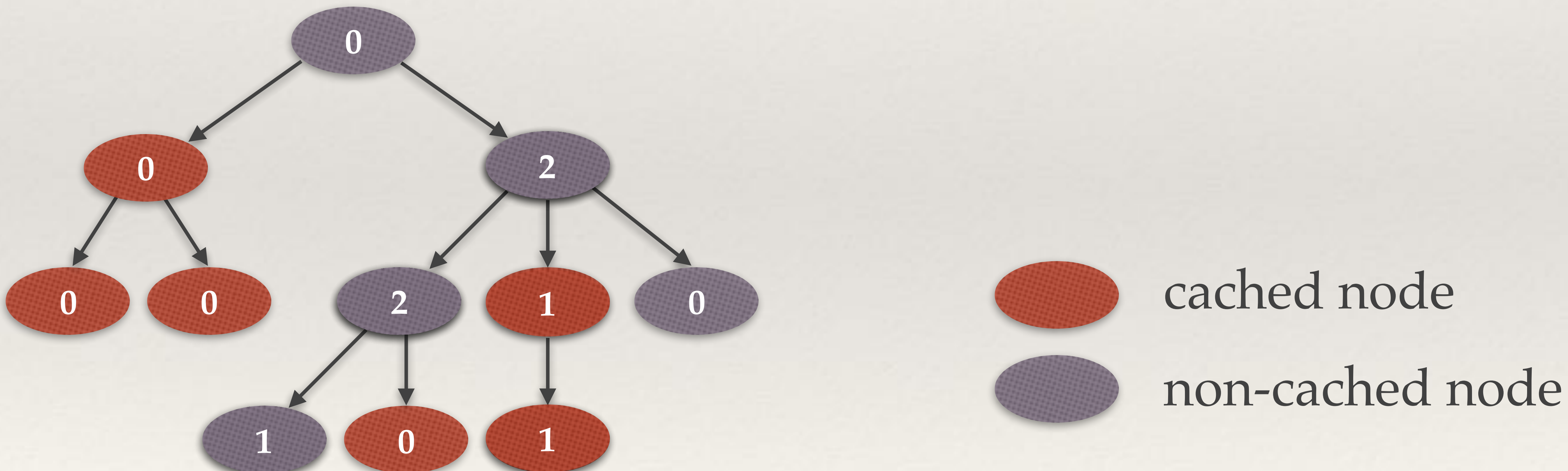
- **Without loss of generality: all requests cost 1** (positive at non-cached nodes, negative at cached ones)
- **Counter** = number of requests at node from the last fetch / eviction of this node.




Assume: $\alpha = 2$.

# Our counter-based algorithm

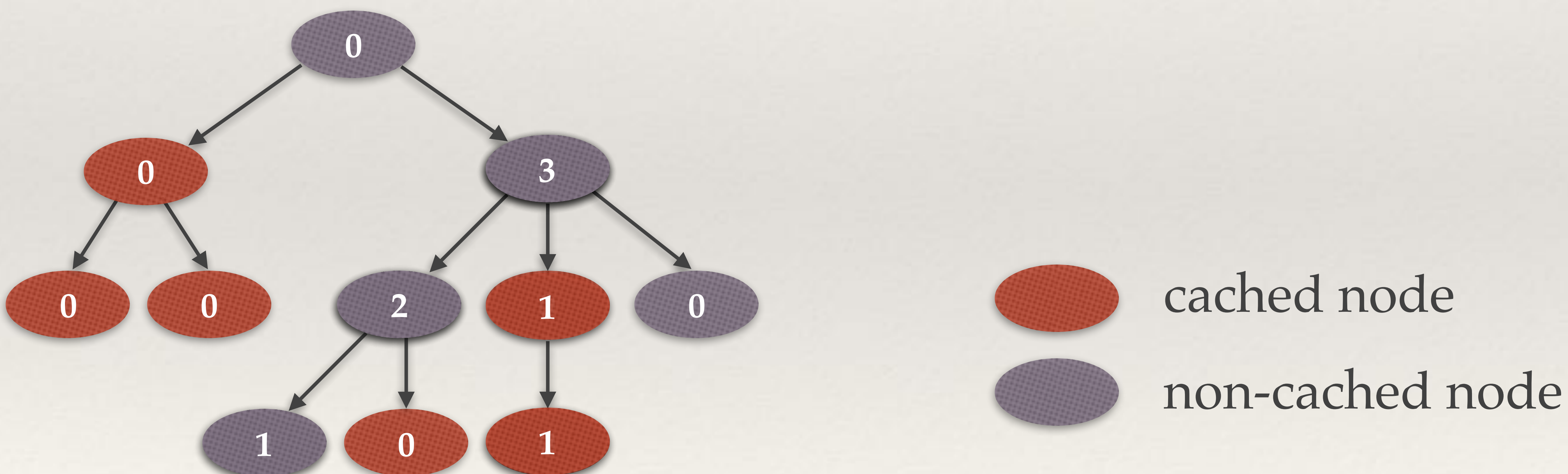
- **Without loss of generality: all requests cost 1** (positive at non-cached nodes, negative at cached ones)
- **Counter** = number of requests at node from the last fetch / eviction of this node.




Assume: $\alpha = 2$.

# Our counter-based algorithm

- **Without loss of generality: all requests cost 1** (positive at non-cached nodes, negative at cached ones)
- **Counter** = number of requests at node from the last fetch / eviction of this node.

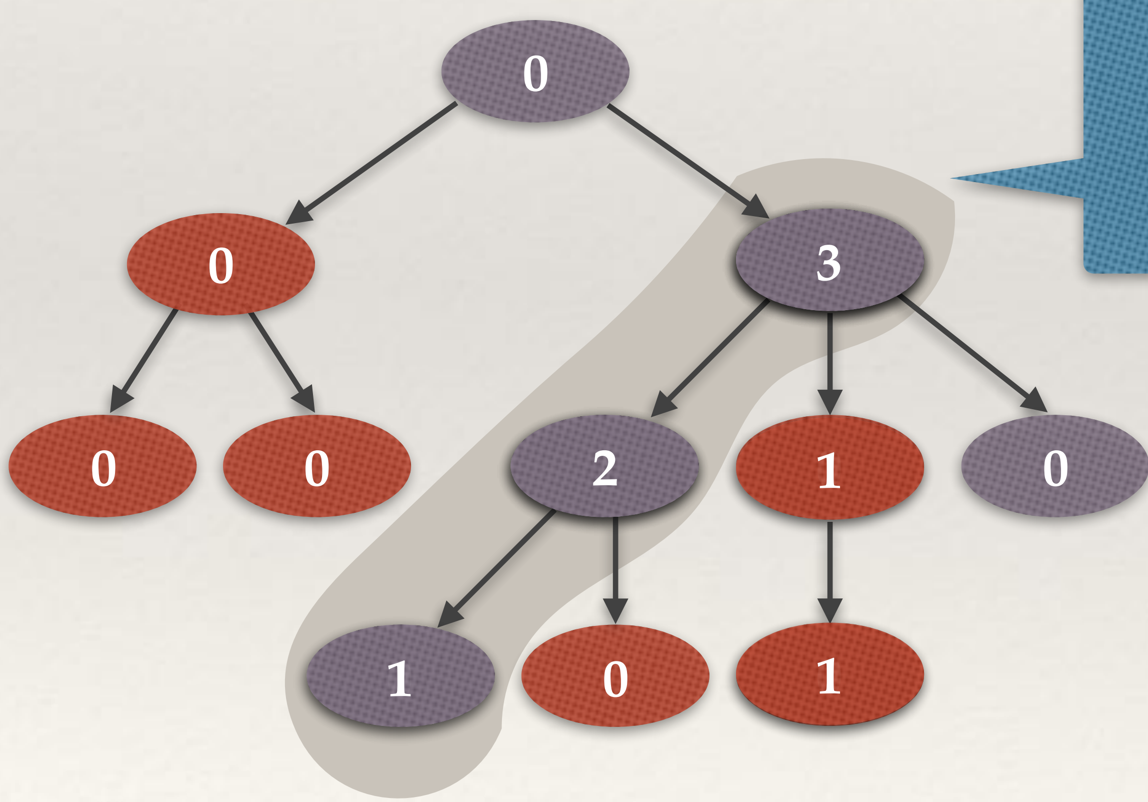



Assume: $\alpha = 2$.

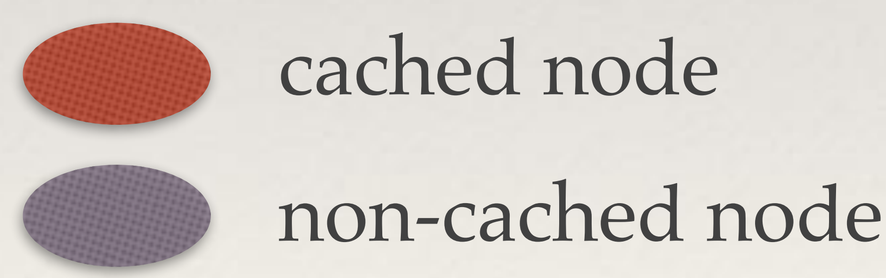
# Our counter-based algorithm

- **Without loss of generality: all requests cost 1**
  (positive at non-cached nodes, negative at cached ones)

- **Counter** = number of requests at node from the last fetch / eviction of this node.



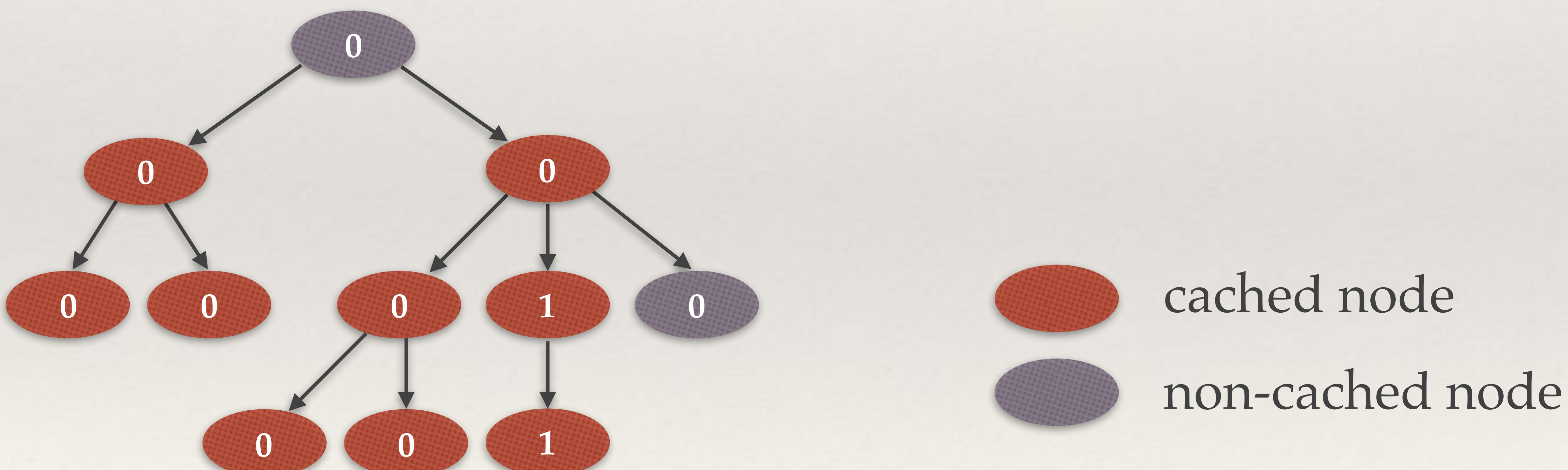

**Sum of counters at X nodes = X $\cdot$ $\alpha$.**
**AND**
**If fetched, the cache remains valid.**




Assume: $\alpha = 2$.

# Our counter-based algorithm

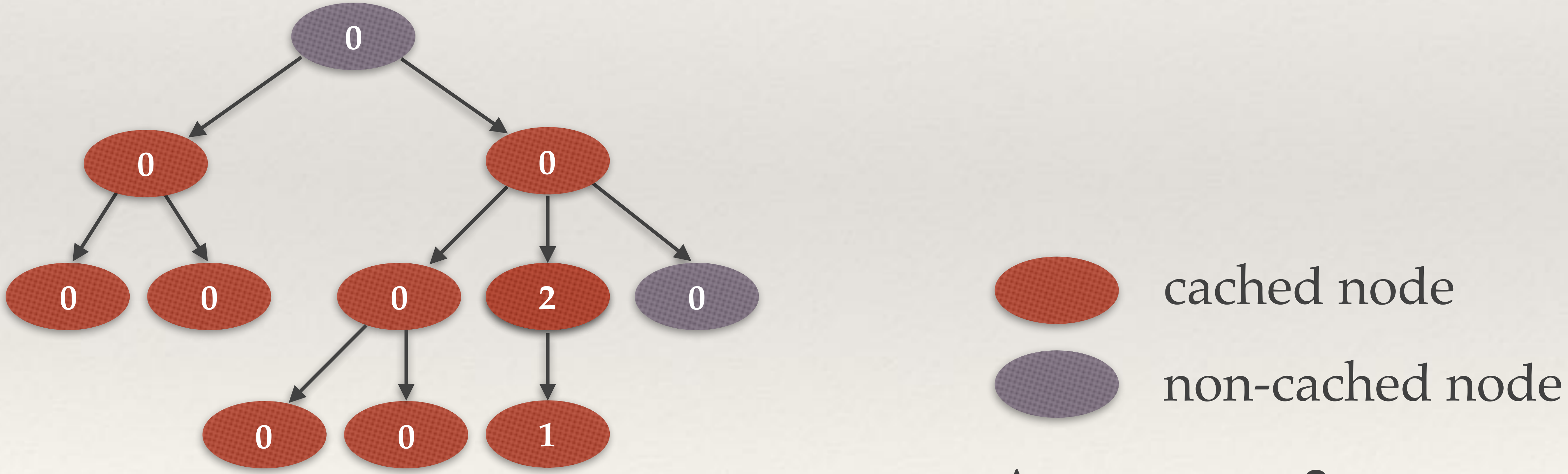
- **Without loss of generality: all requests cost 1** (positive at non-cached nodes, negative at cached ones)
- **Counter** = number of requests at node from the last fetch / last eviction of this node.




Assume: $\alpha = 2$.

# Our counter-based algorithm

- **Without loss of generality: all requests cost 1**
  (positive at non-cached nodes, negative at cached ones)
- **Counter** = number of requests at node from the last fetch / last eviction of this node.

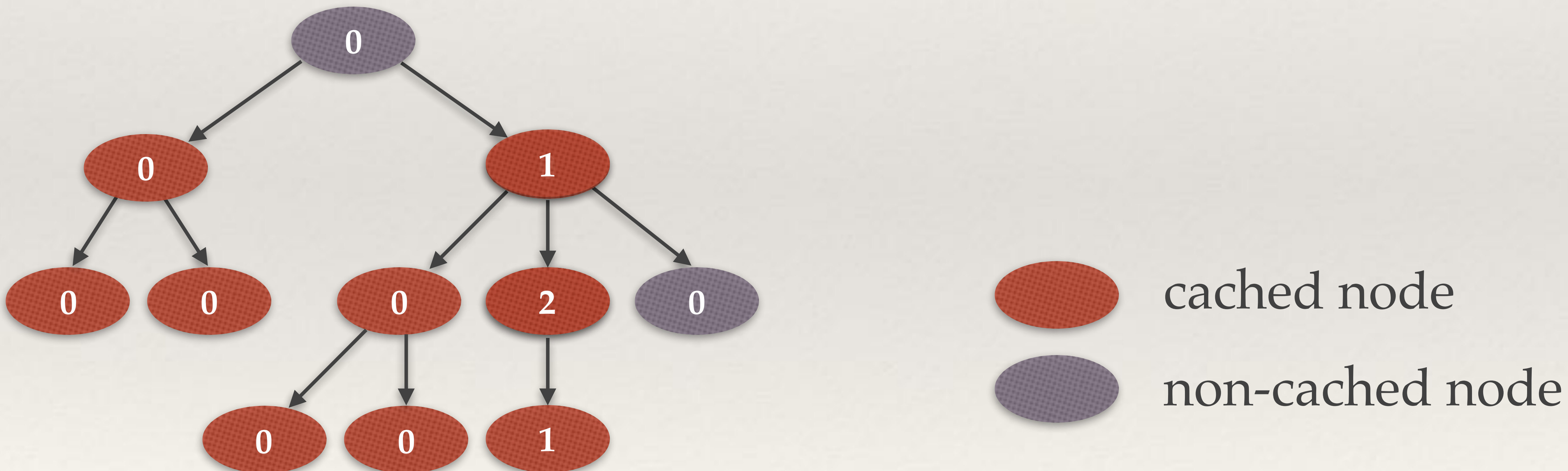



Assume: $\alpha = 2$.

# Our counter-based algorithm

- **Without loss of generality: all requests cost 1**
(positive at non-cached nodes, negative at cached ones)

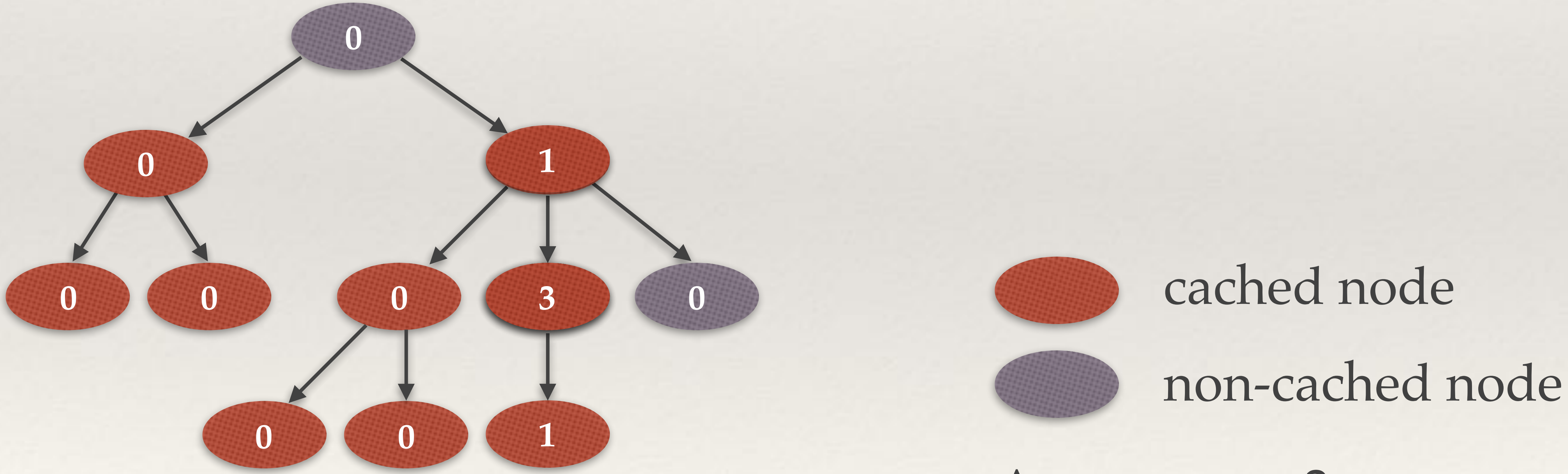
- **Counter** = number of requests at node from the last fetch / last eviction of this node.




Assume: $\alpha = 2$.

# Our counter-based algorithm

- **Without loss of generality: all requests cost 1**
(positive at non-cached nodes, negative at cached ones)
- **Counter** = number of requests at node from the last fetch / last eviction of this node.

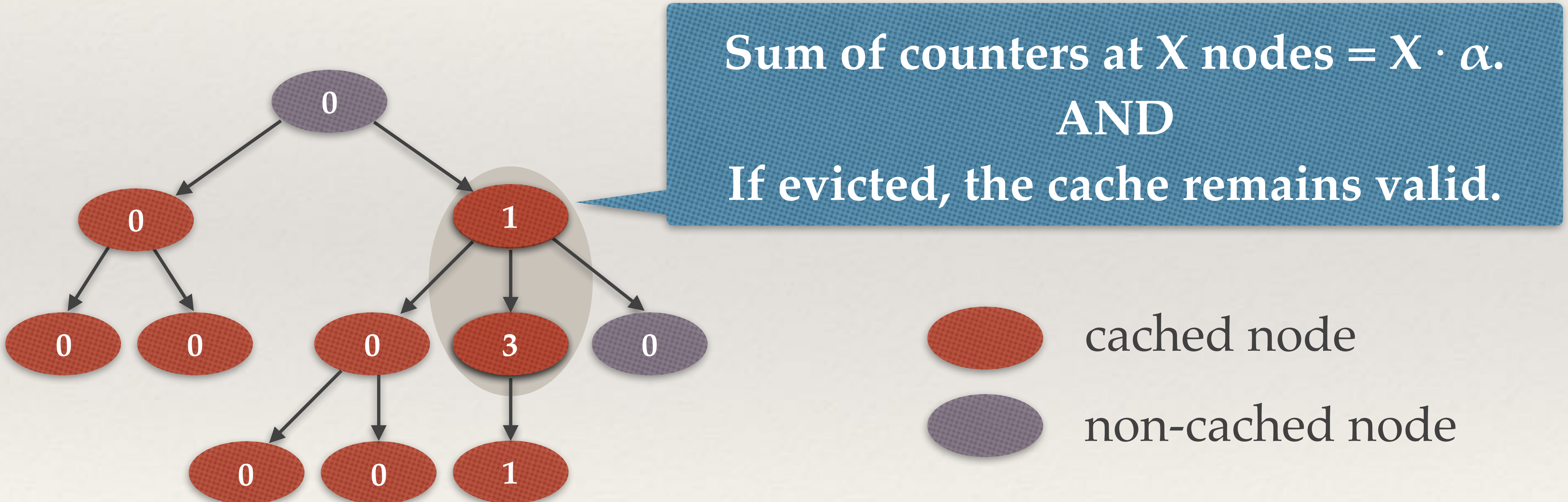



Assume: $\alpha = 2$.

# Our counter-based algorithm

- **Without loss of generality: all requests cost 1**
(positive at non-cached nodes, negative at cached ones)

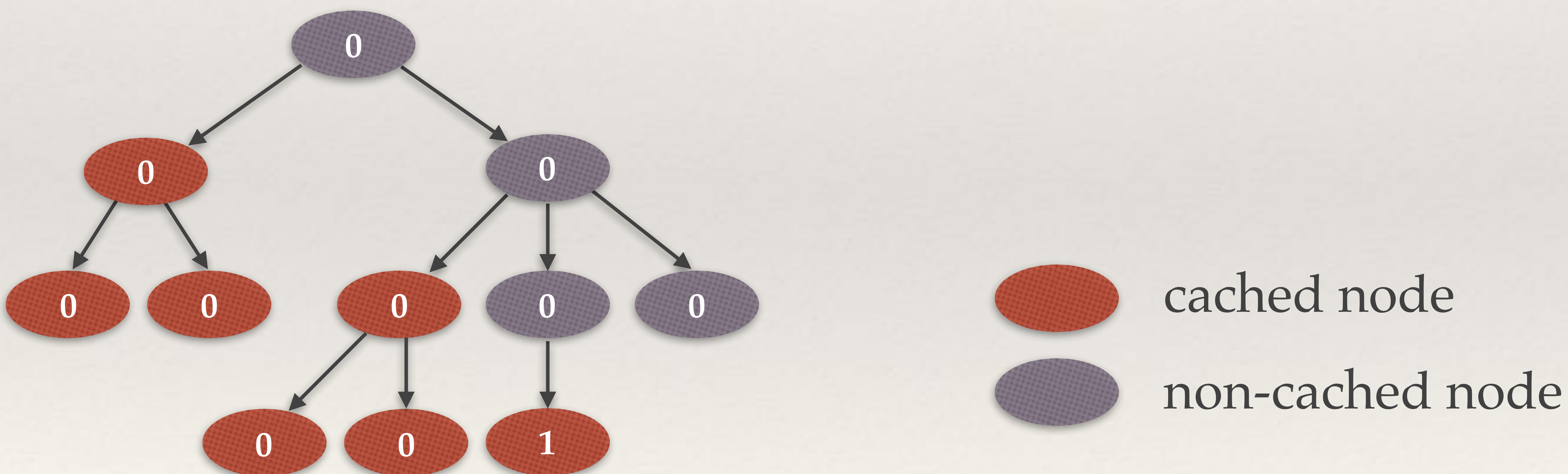
- **Counter** = number of requests at node from the last fetch /
last eviction of this node.




Assume: $\alpha = 2$.

# Our counter-based algorithm

- **Without loss of generality: all requests cost 1** (positive at non-cached nodes, negative at cached ones)
- **Counter** = number of requests at node from the last fetch / last eviction of this node.



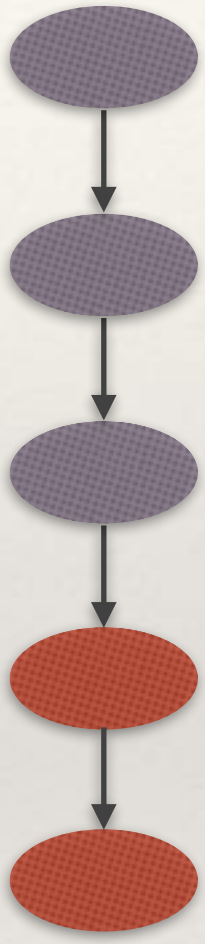

Assume: $\alpha = 2$.

# Glimpse of analysis

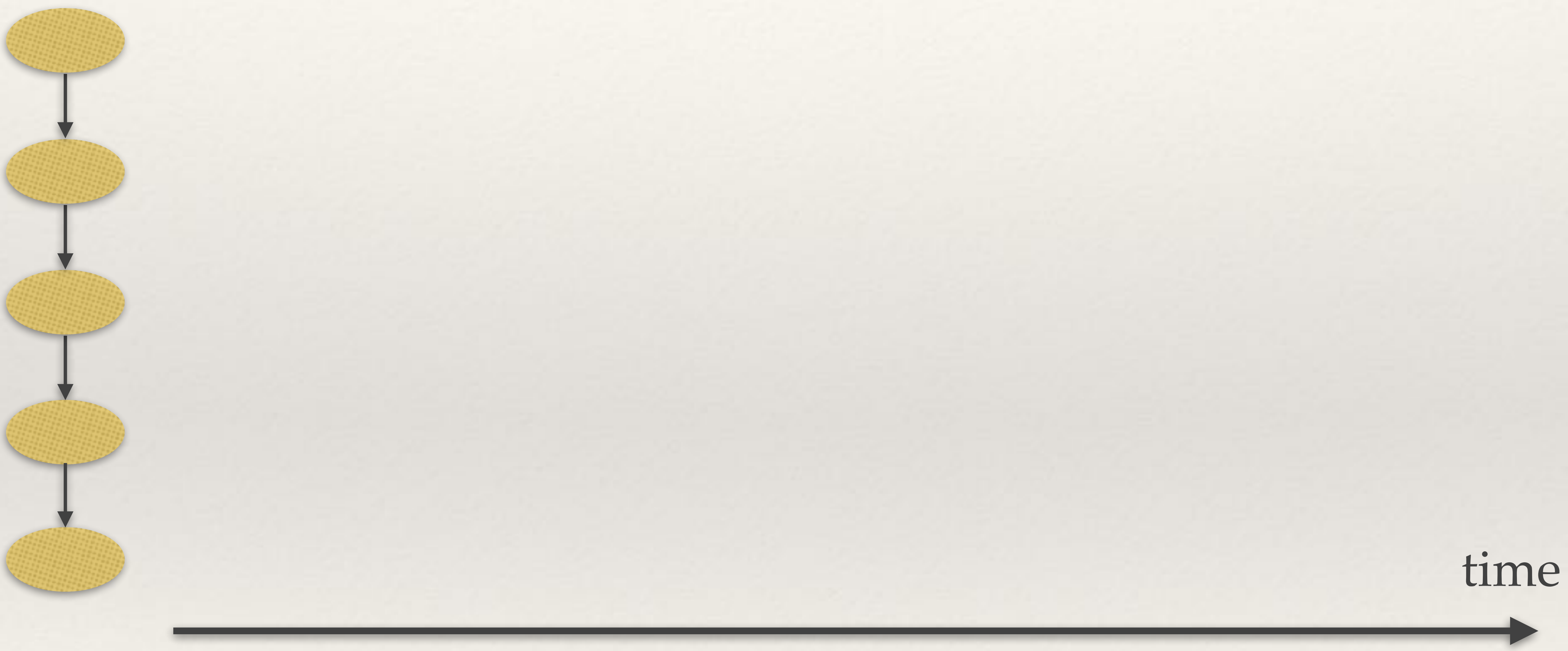
# The plan

**Analysis when tree *T* is a line**

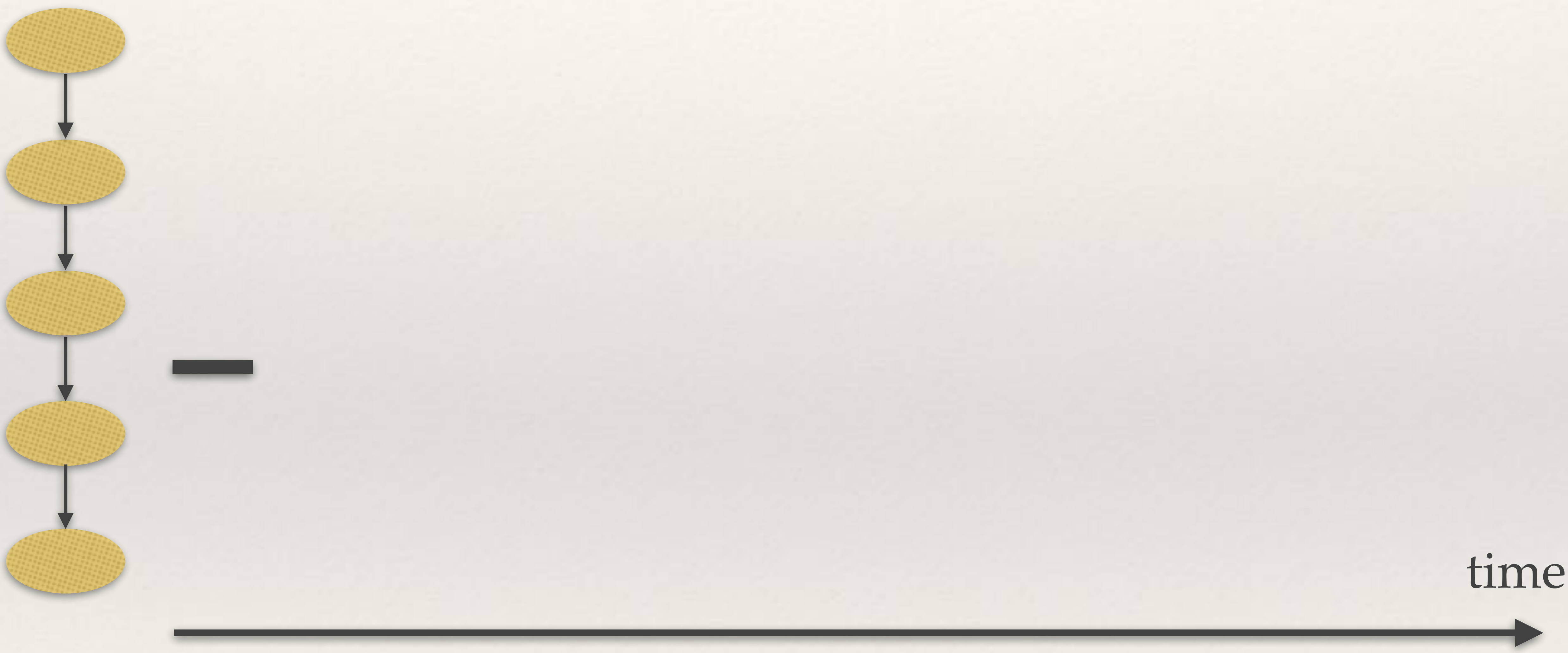
- Nicer geometry.
- Omits gory details of the general case.



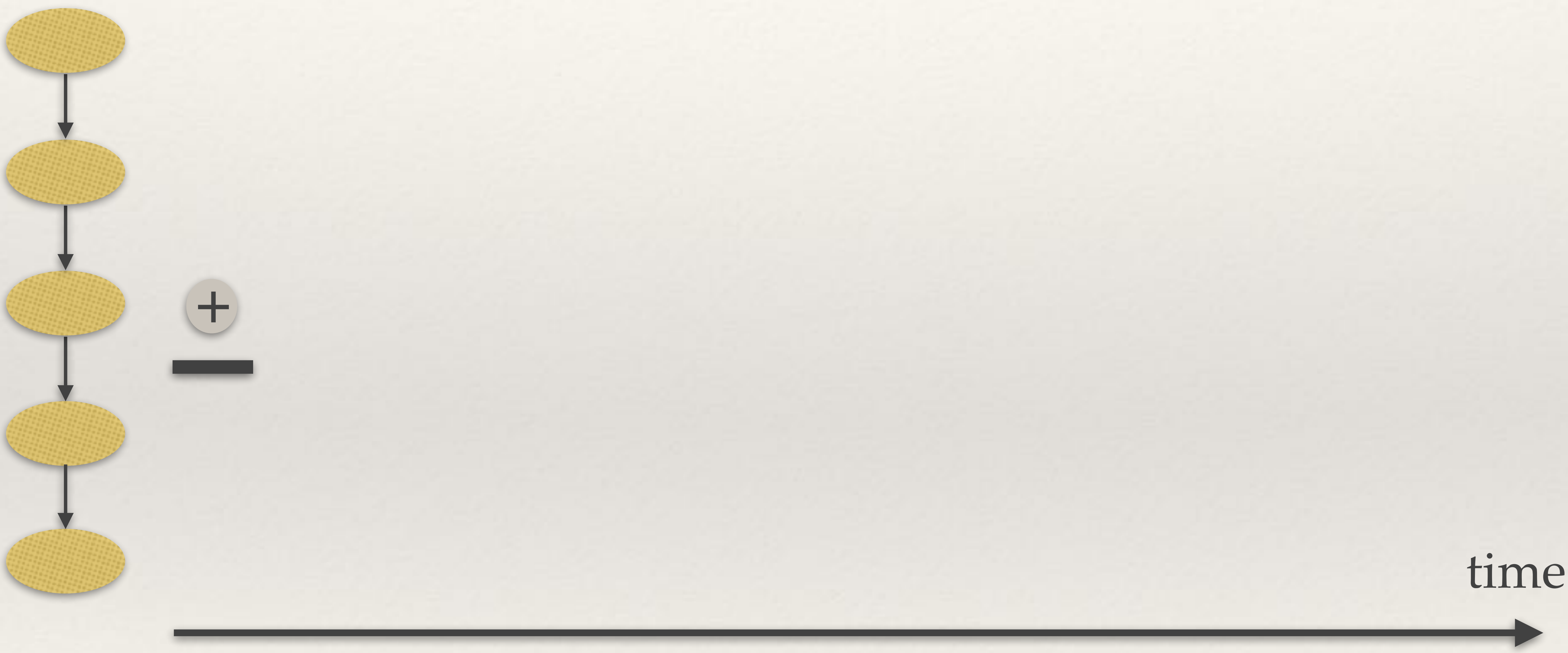
# Bounding cost of ALG



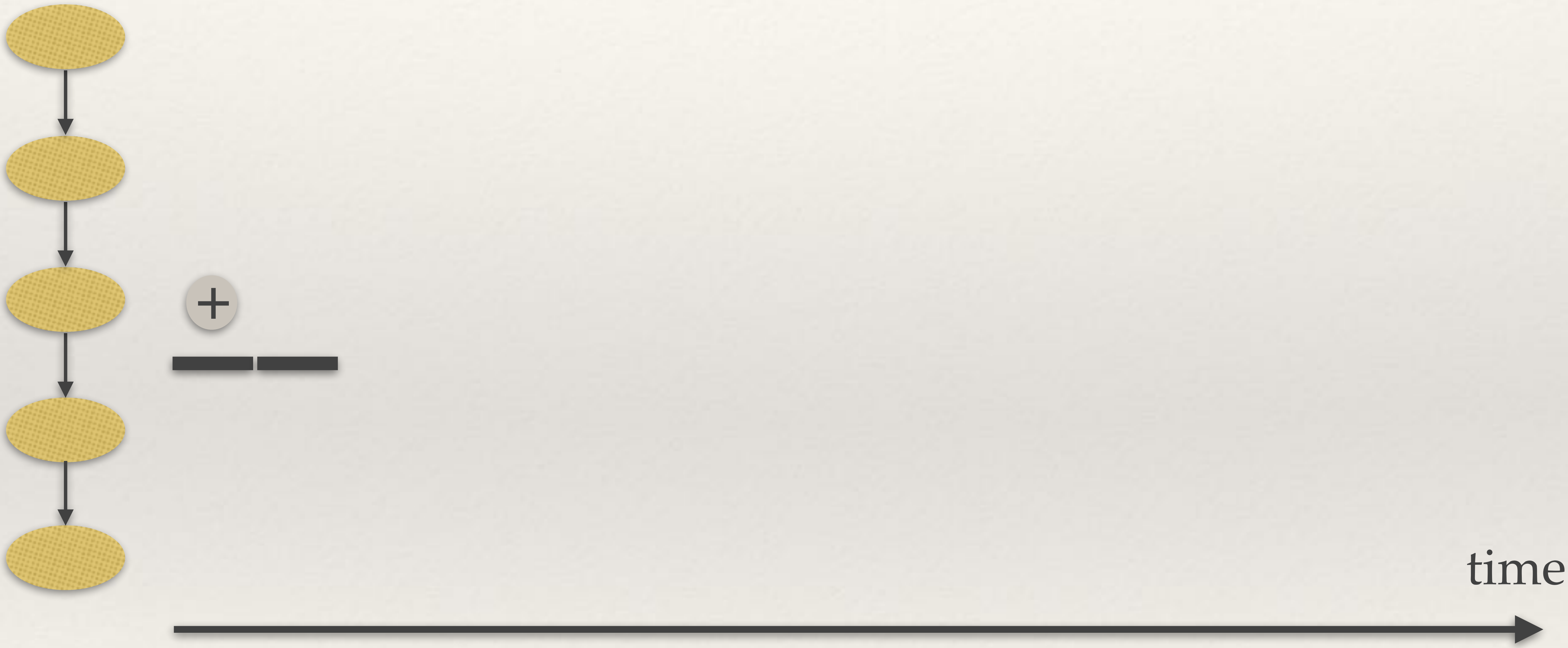
# Bounding cost of ALG



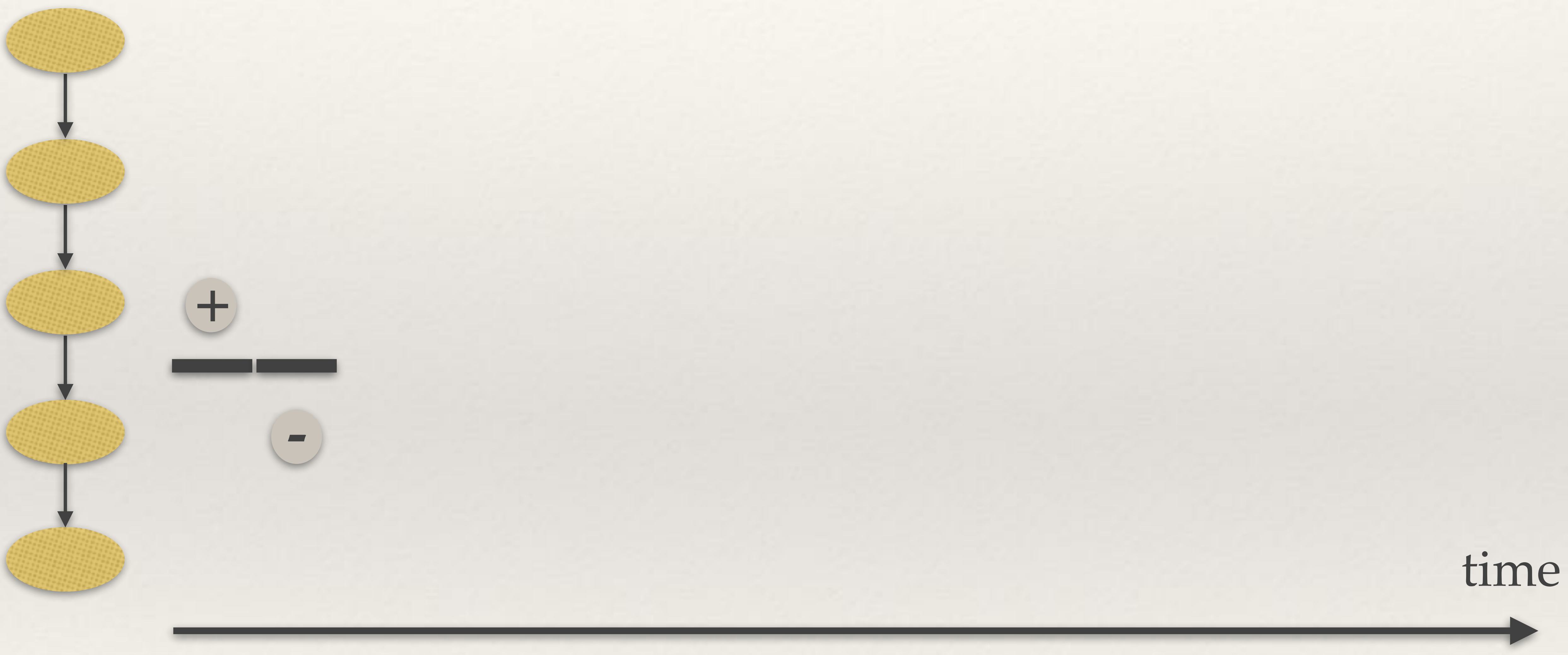
# Bounding cost of ALG



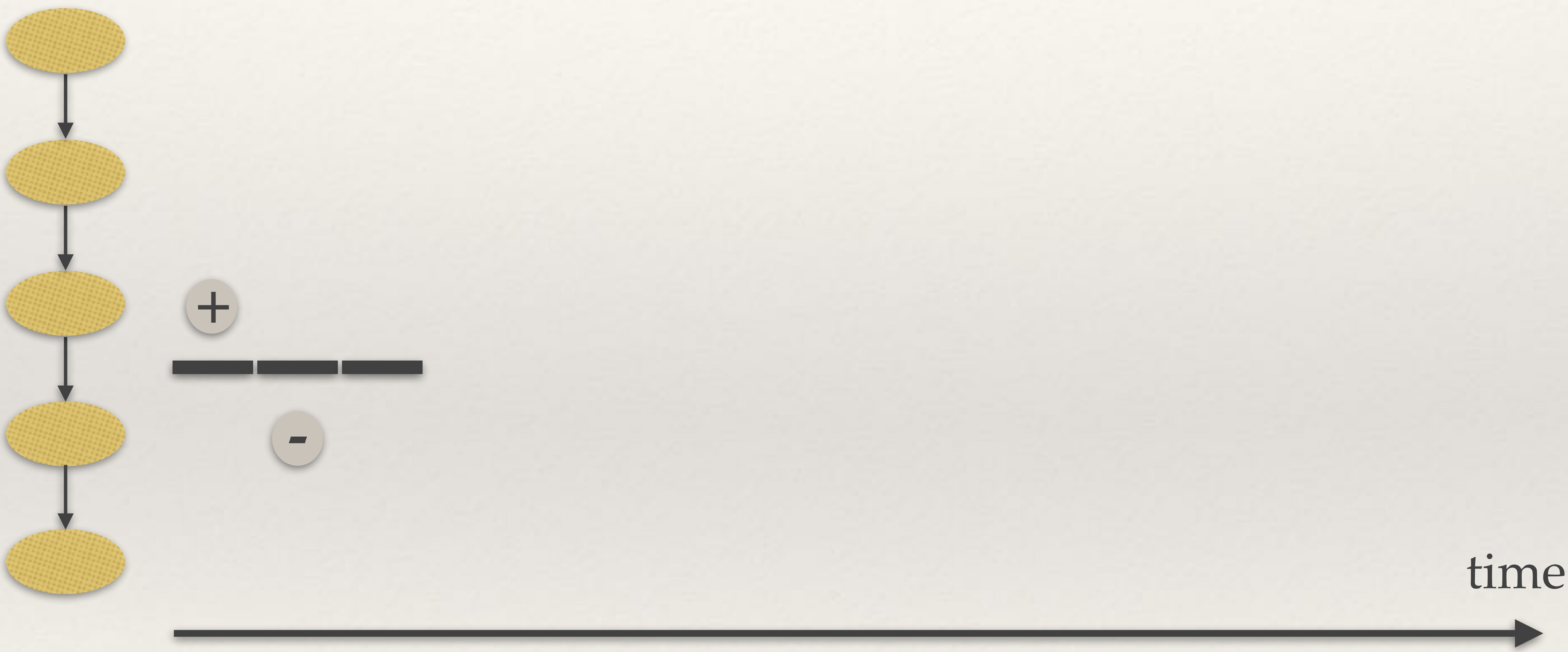
# Bounding cost of ALG



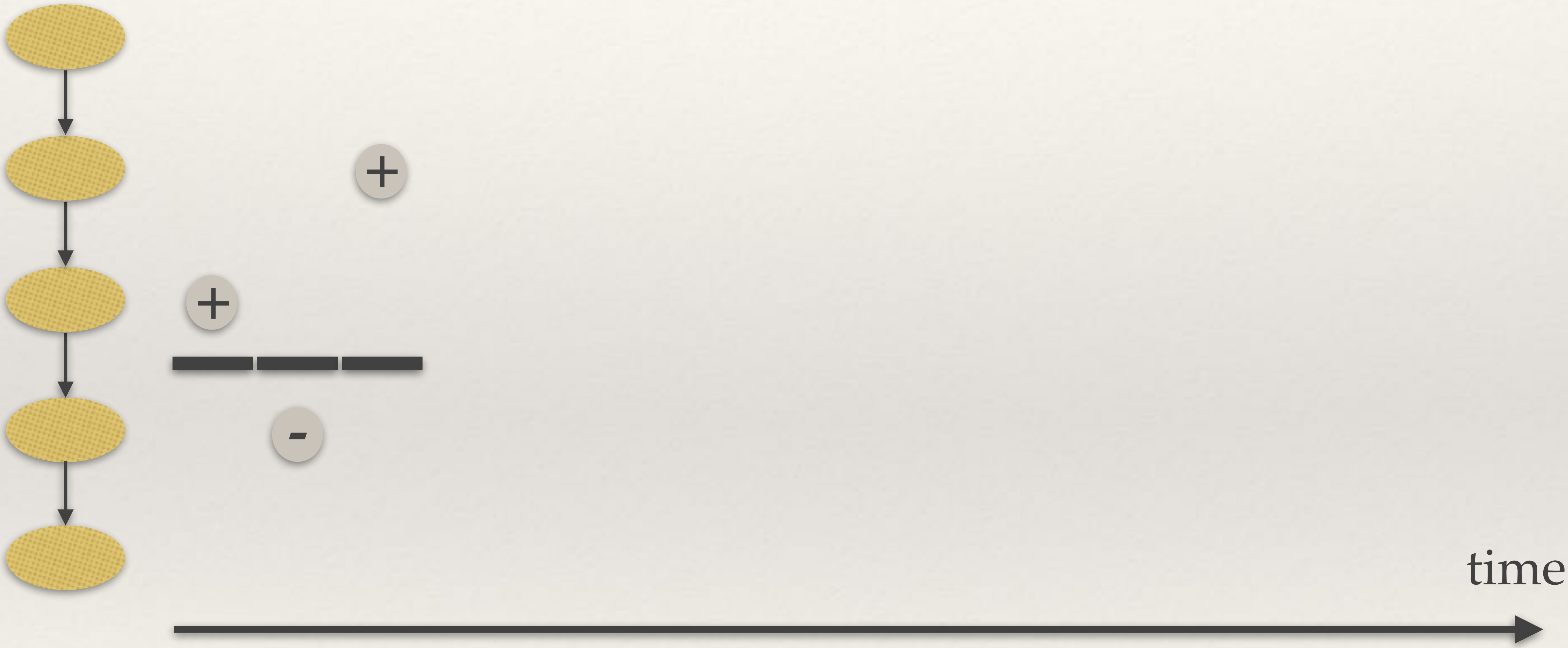
# Bounding cost of ALG



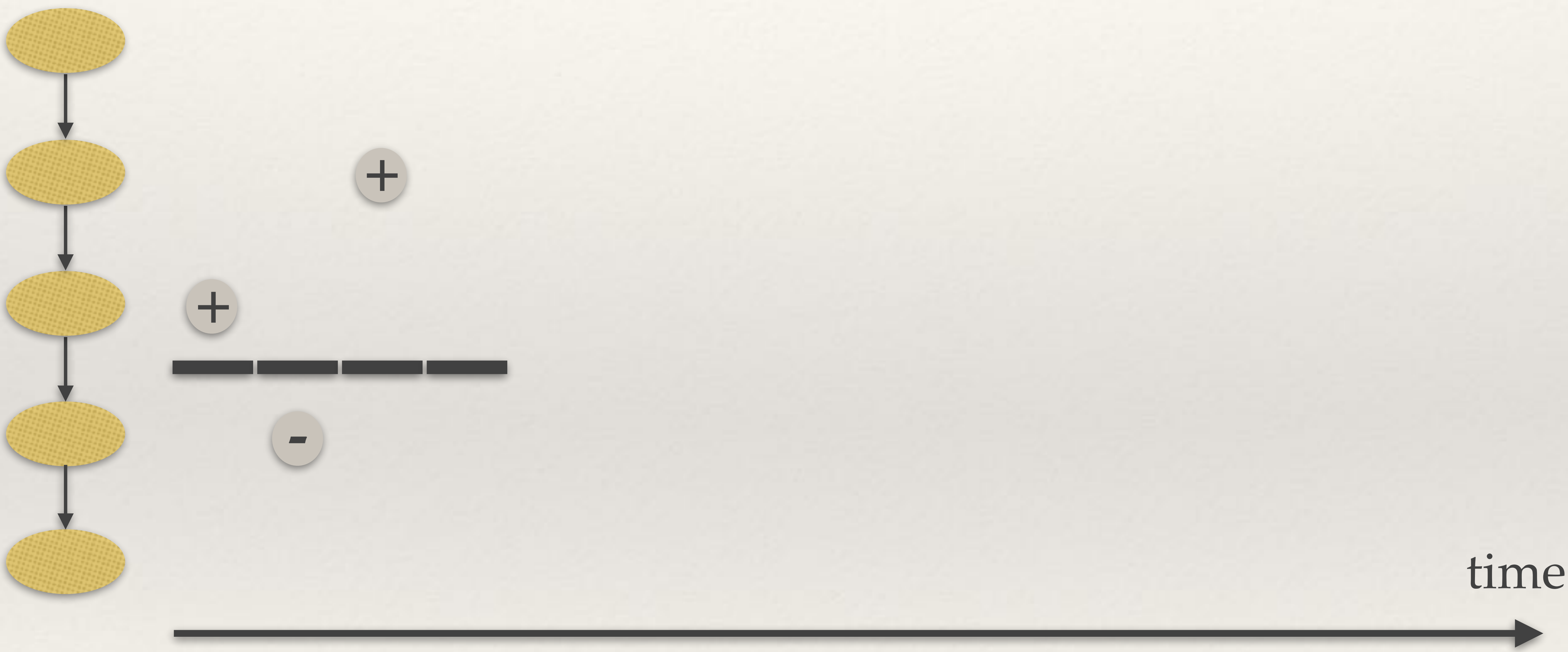
# Bounding cost of ALG



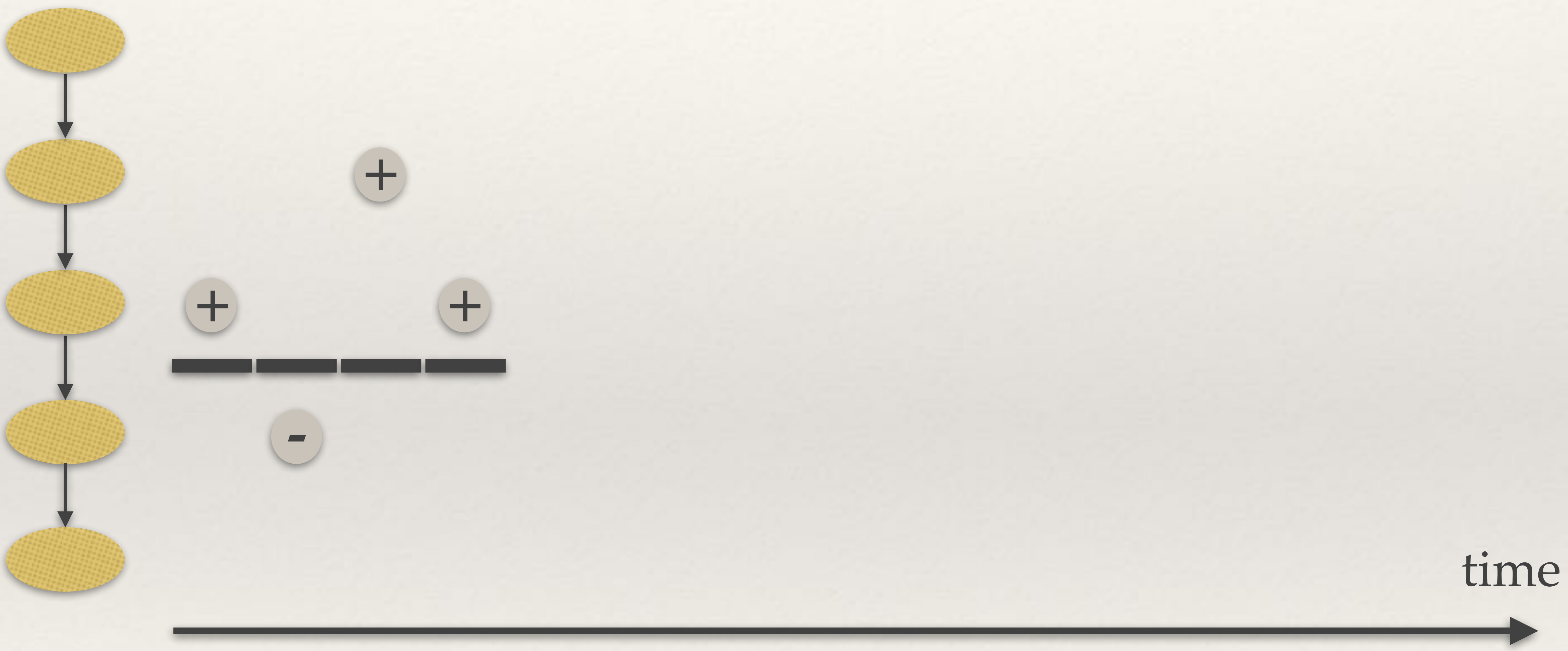
# Bounding cost of ALG



# Bounding cost of ALG



# Bounding cost of ALG



# Bounding cost of ALG

# Bounding cost of ALG

# Bounding cost of ALG

# Bounding cost of ALG

# Bounding cost of ALG

# Bounding cost of ALG

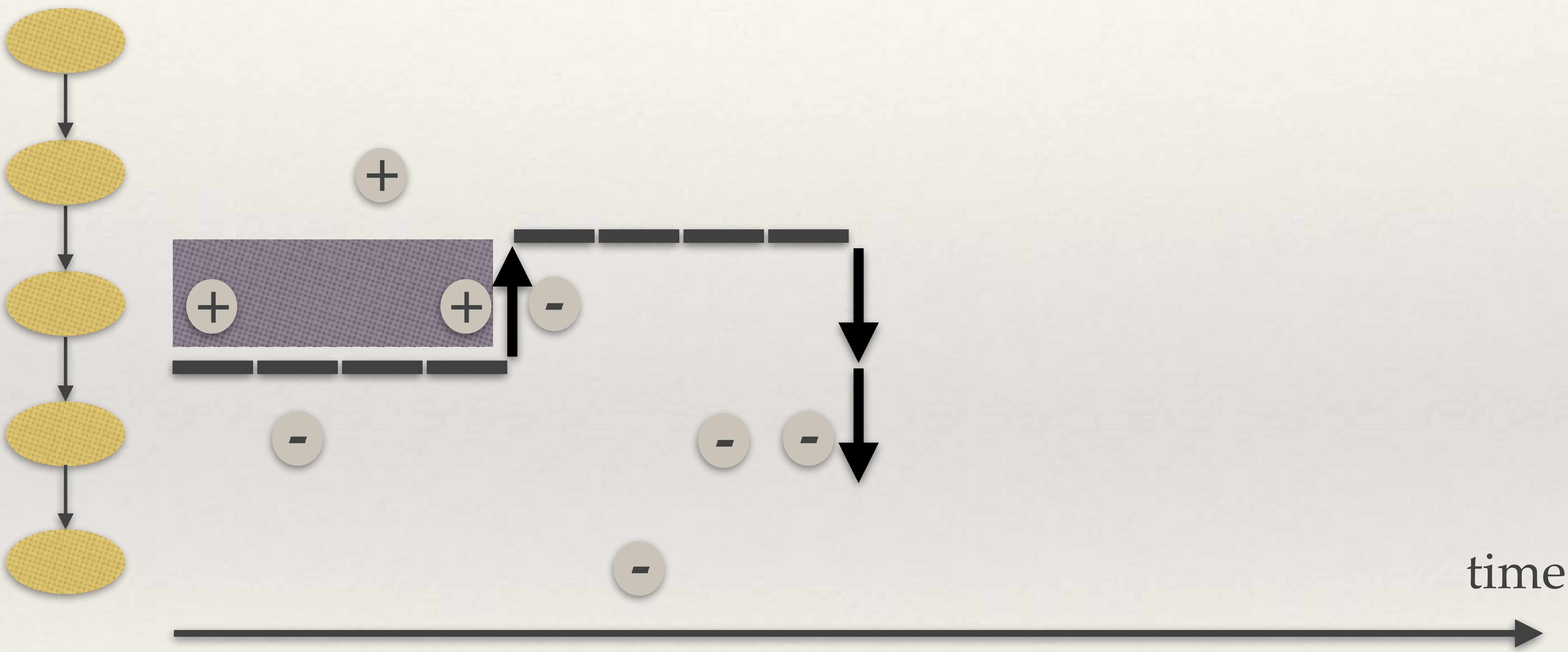
# Bounding cost of ALG

# Bounding cost of ALG

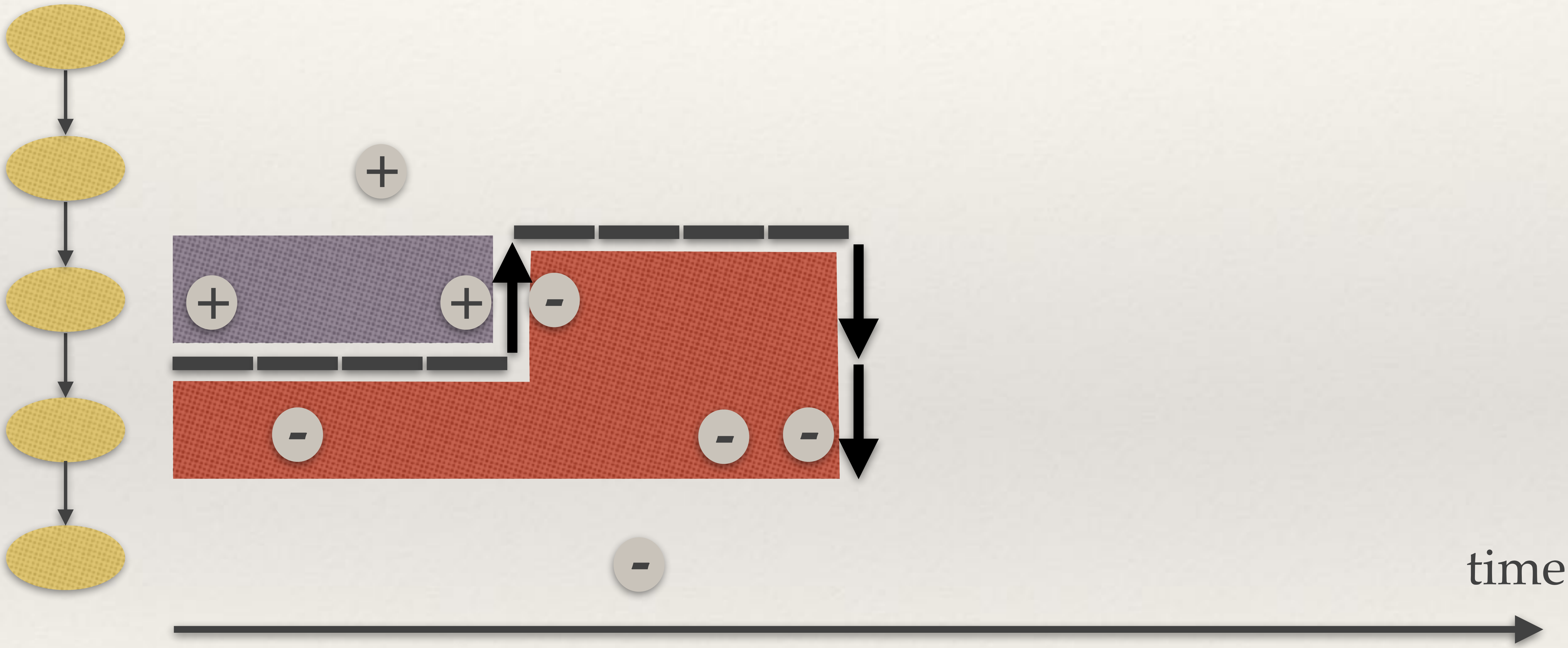
# Bounding cost of ALG

# Bounding cost of ALG

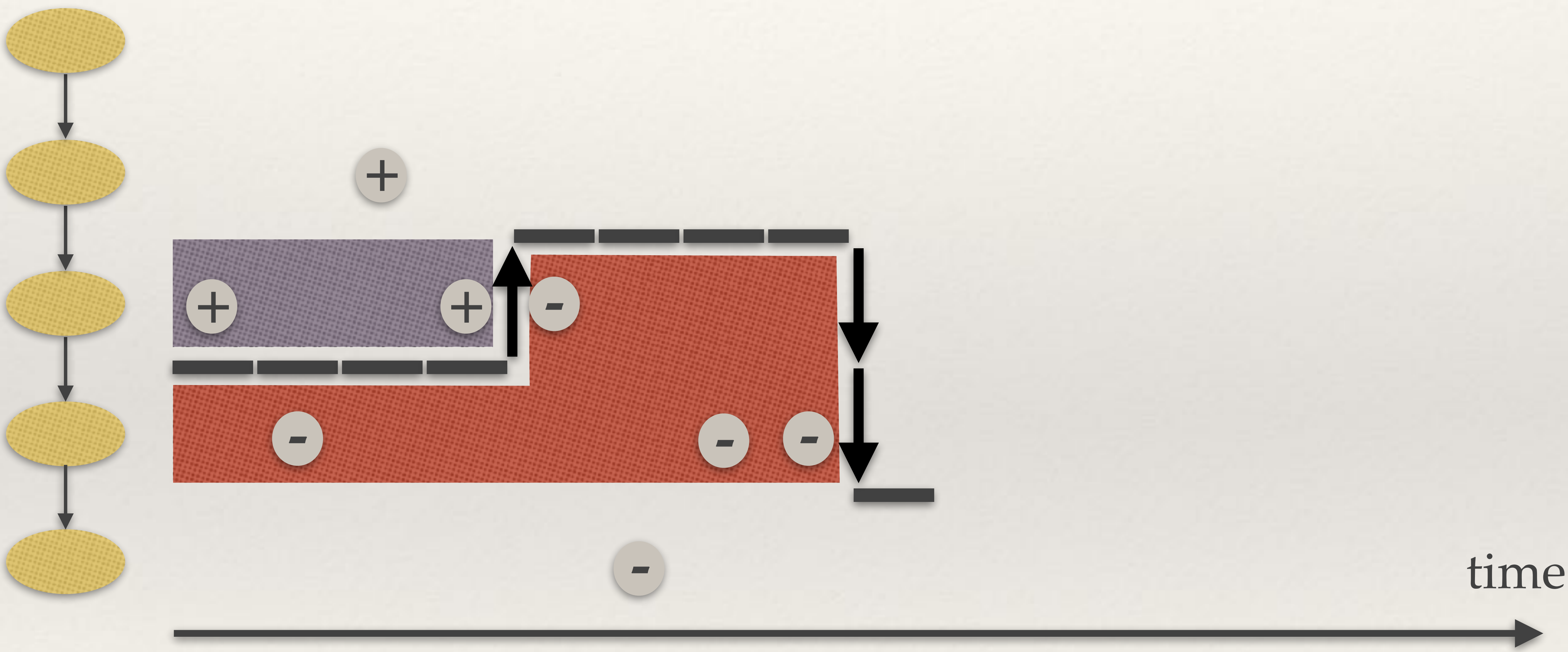
# Bounding cost of ALG



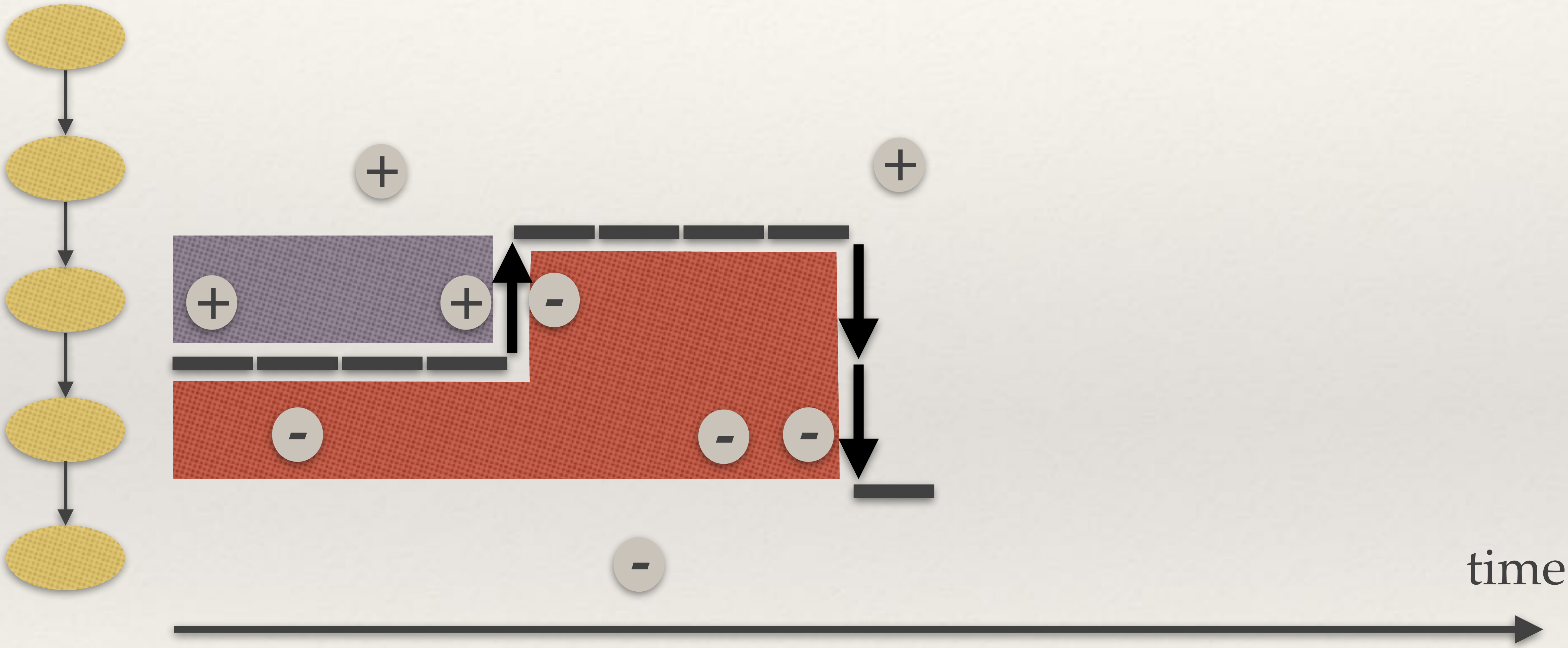
# Bounding cost of ALG



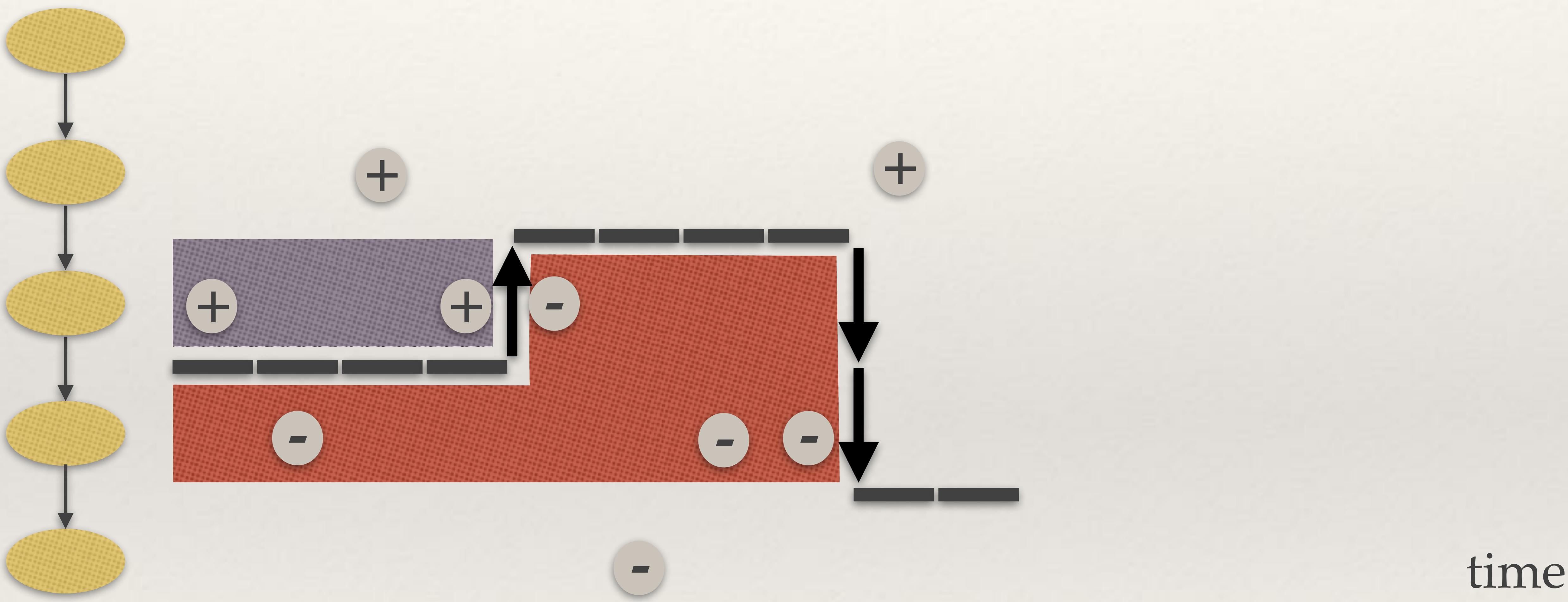
# Bounding cost of ALG



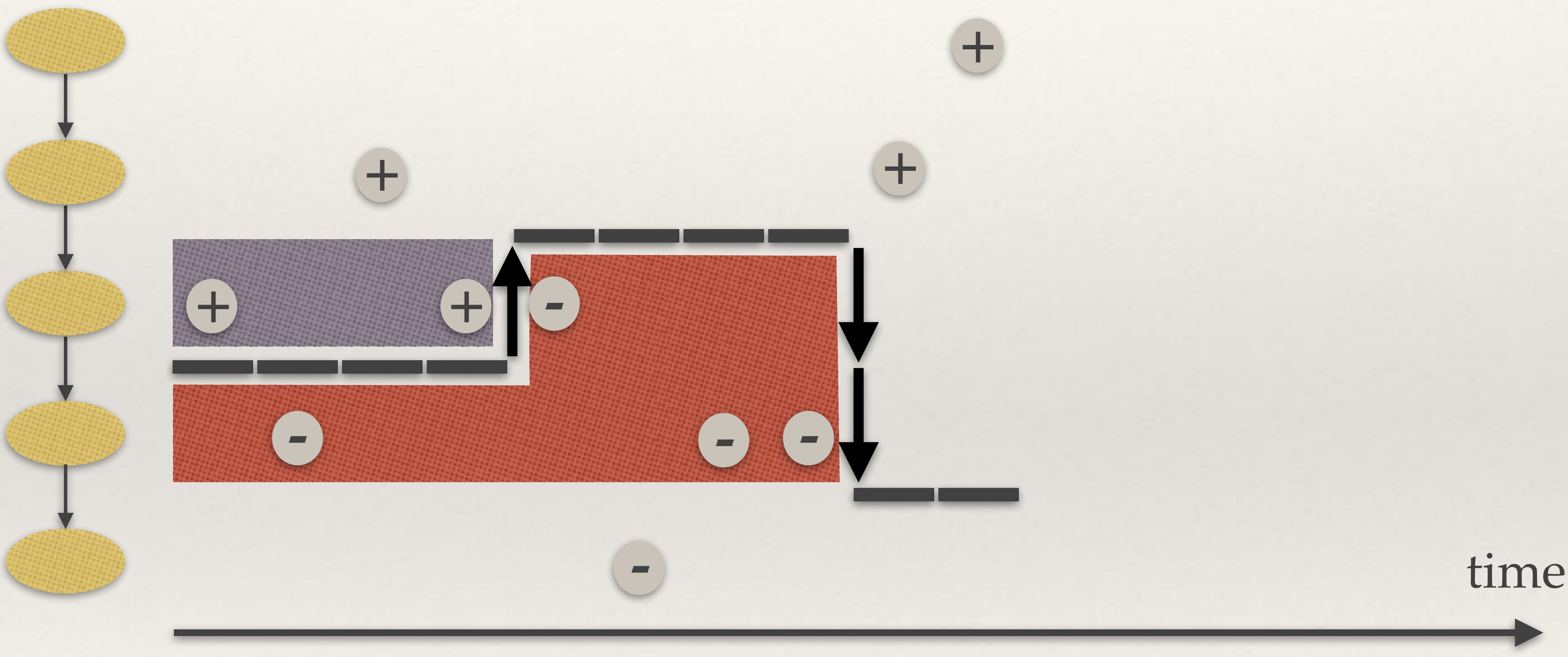
# Bounding cost of ALG



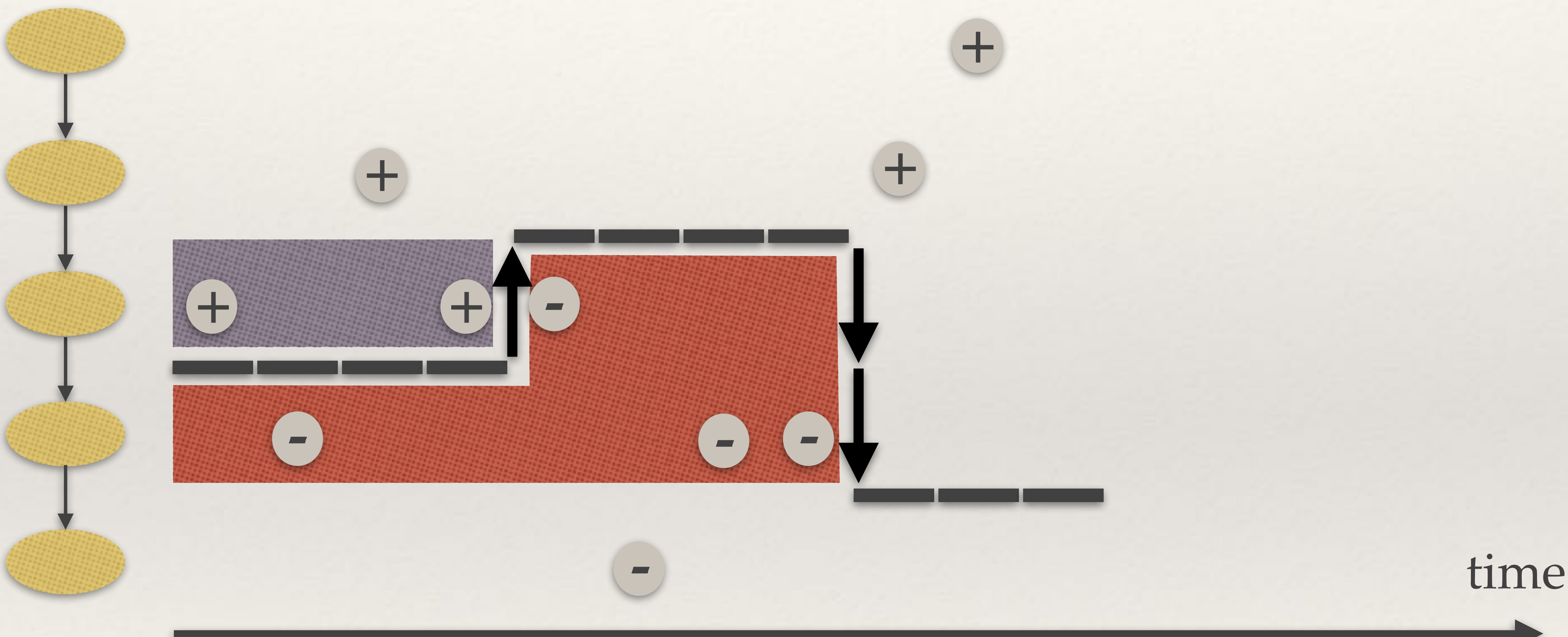
# Bounding cost of ALG



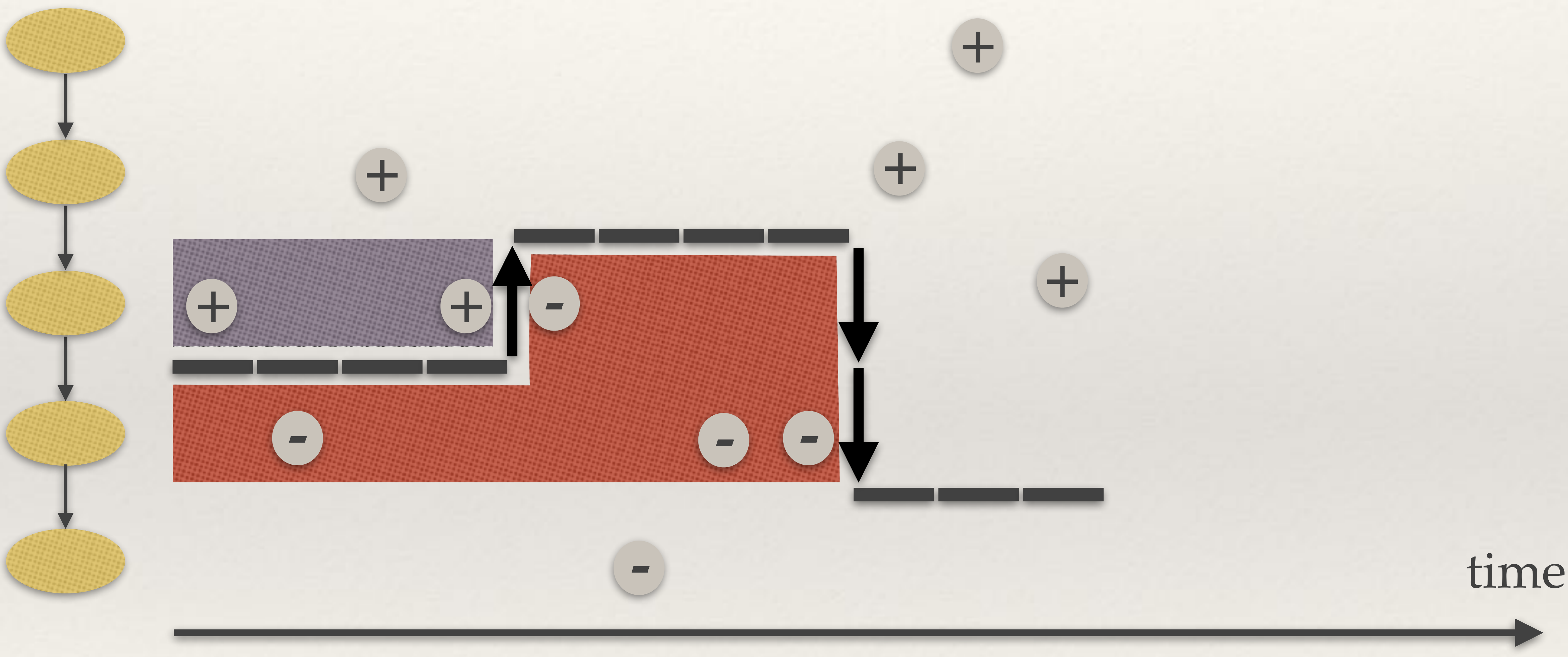
# Bounding cost of ALG



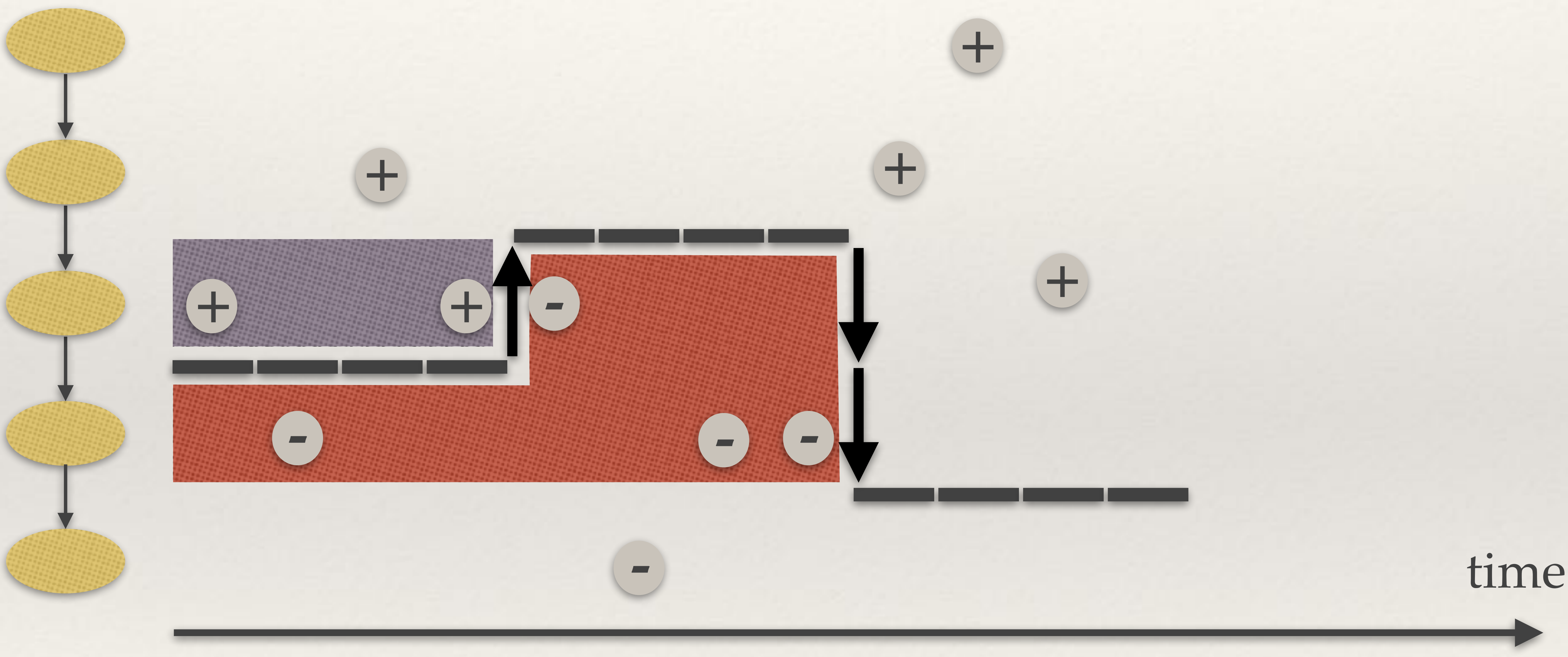
# Bounding cost of ALG



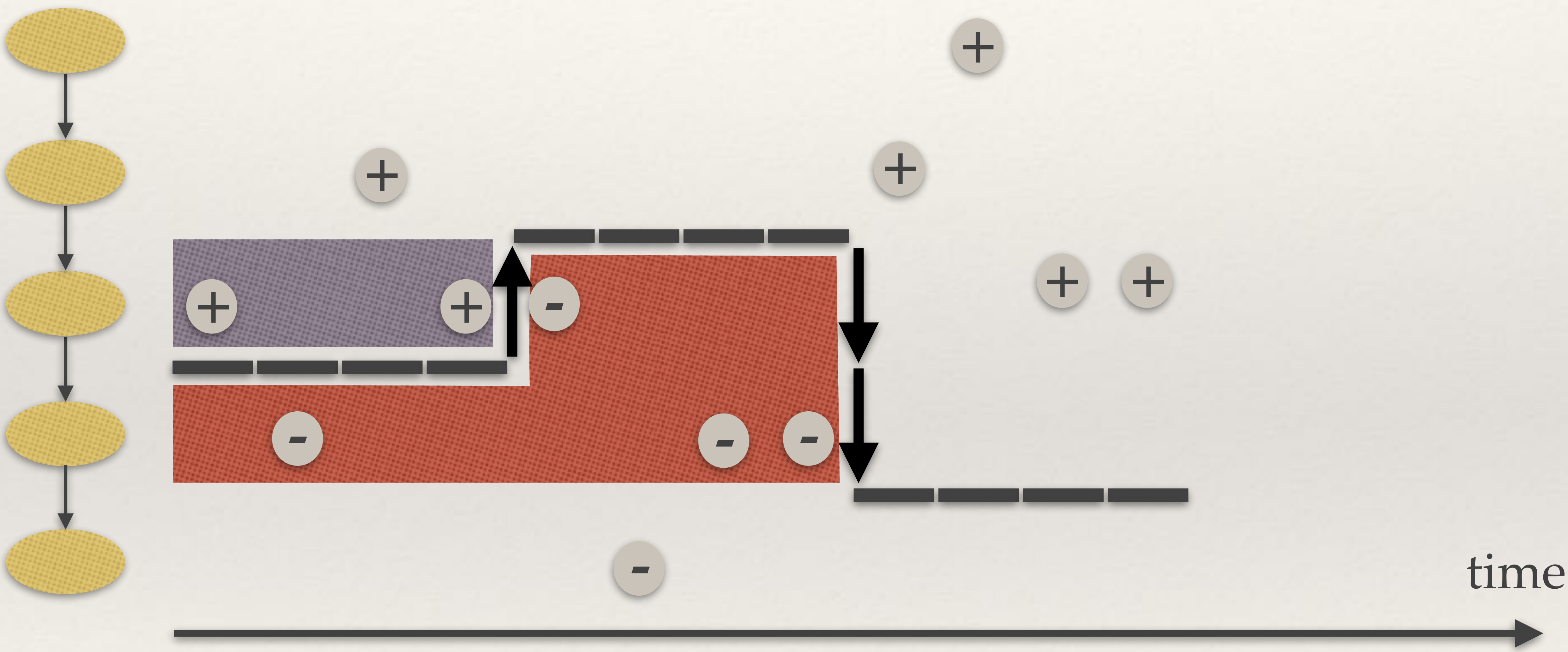
# Bounding cost of ALG



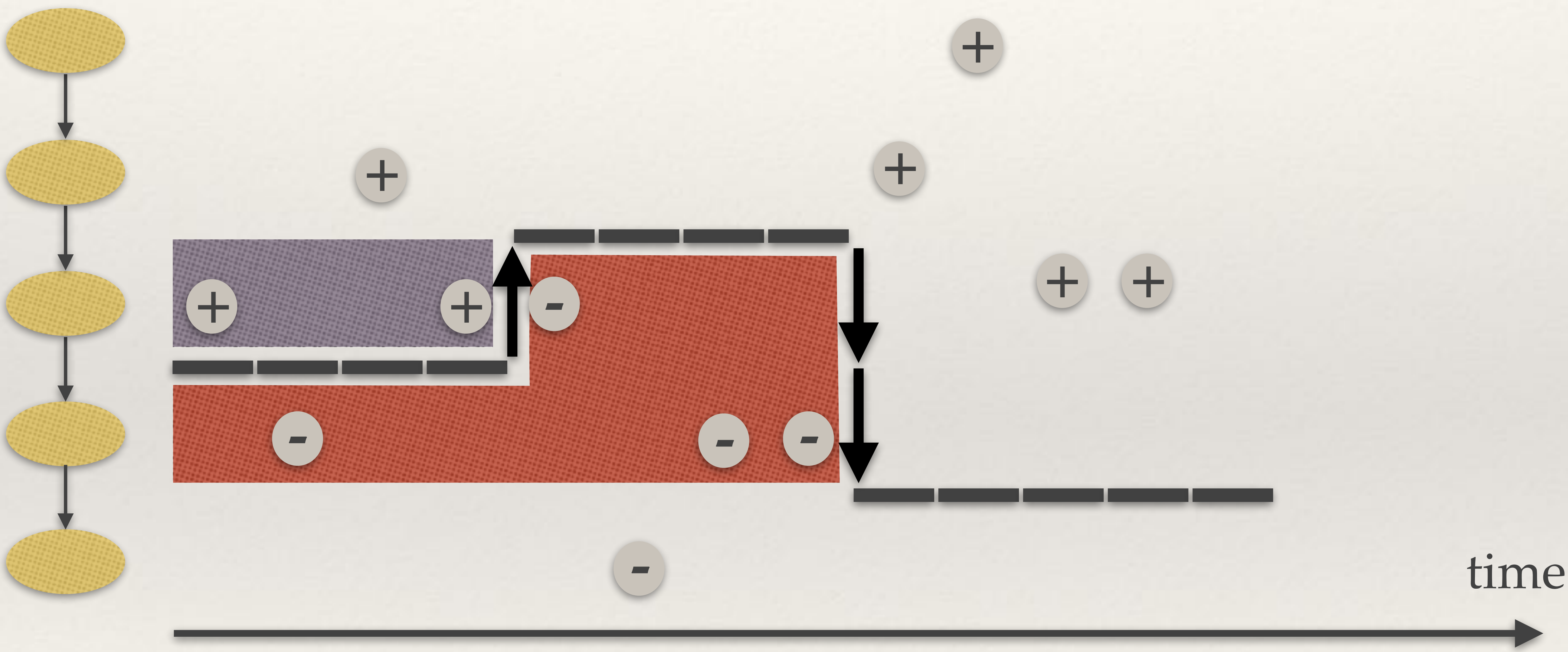
# Bounding cost of ALG



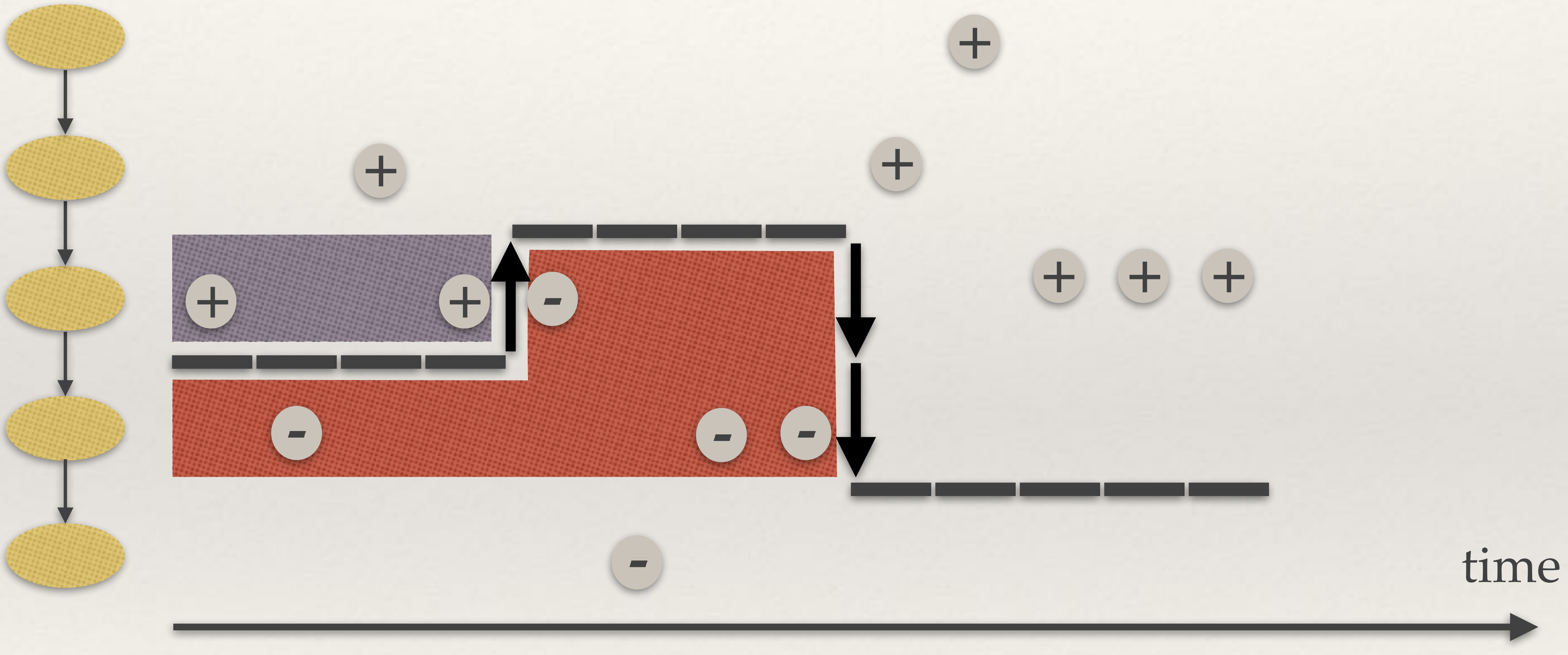
# Bounding cost of ALG



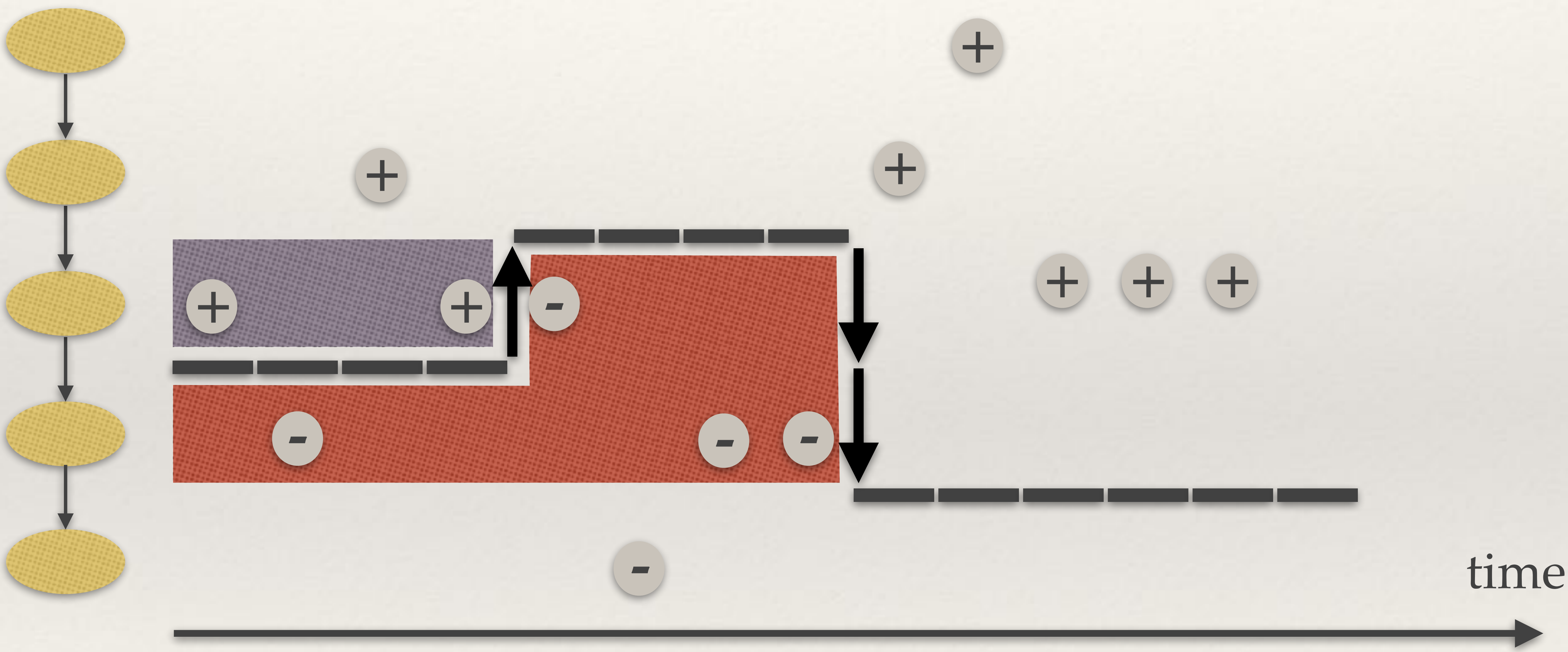
# Bounding cost of ALG



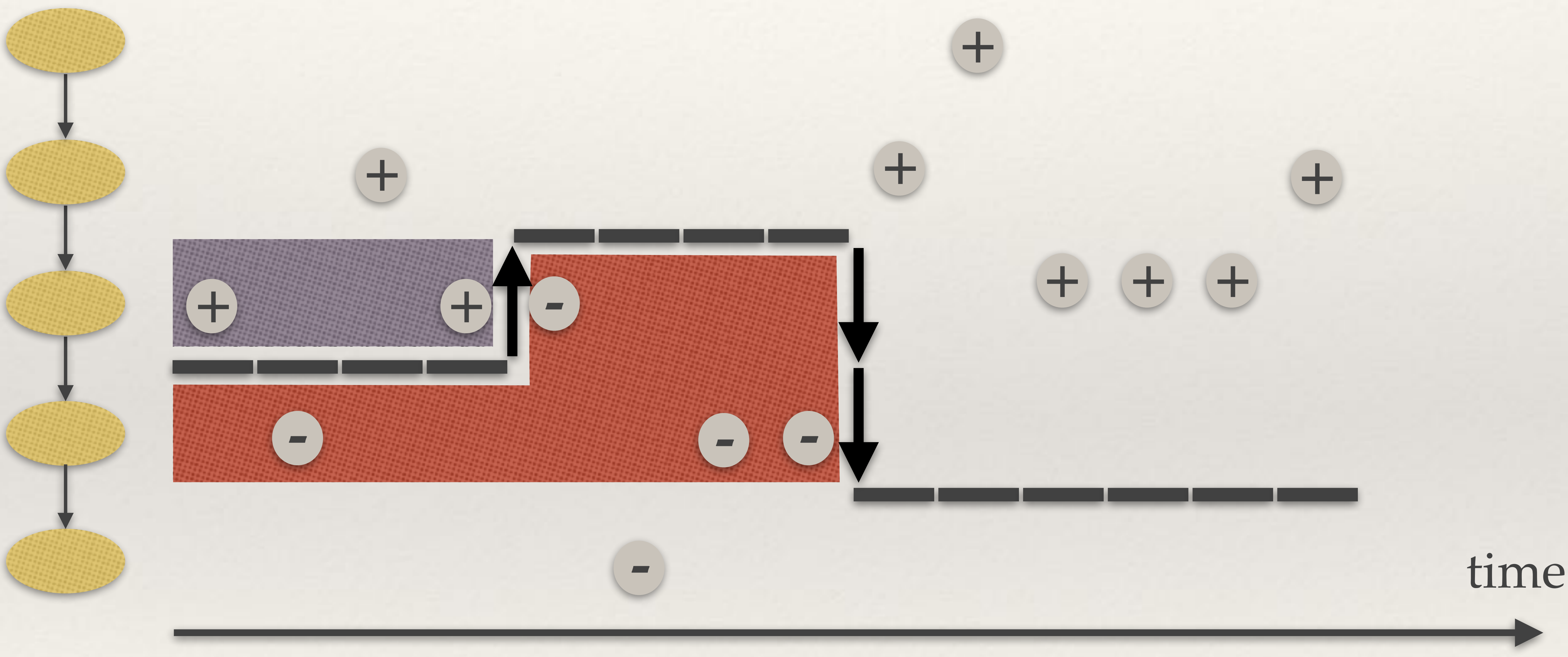
# Bounding cost of ALG



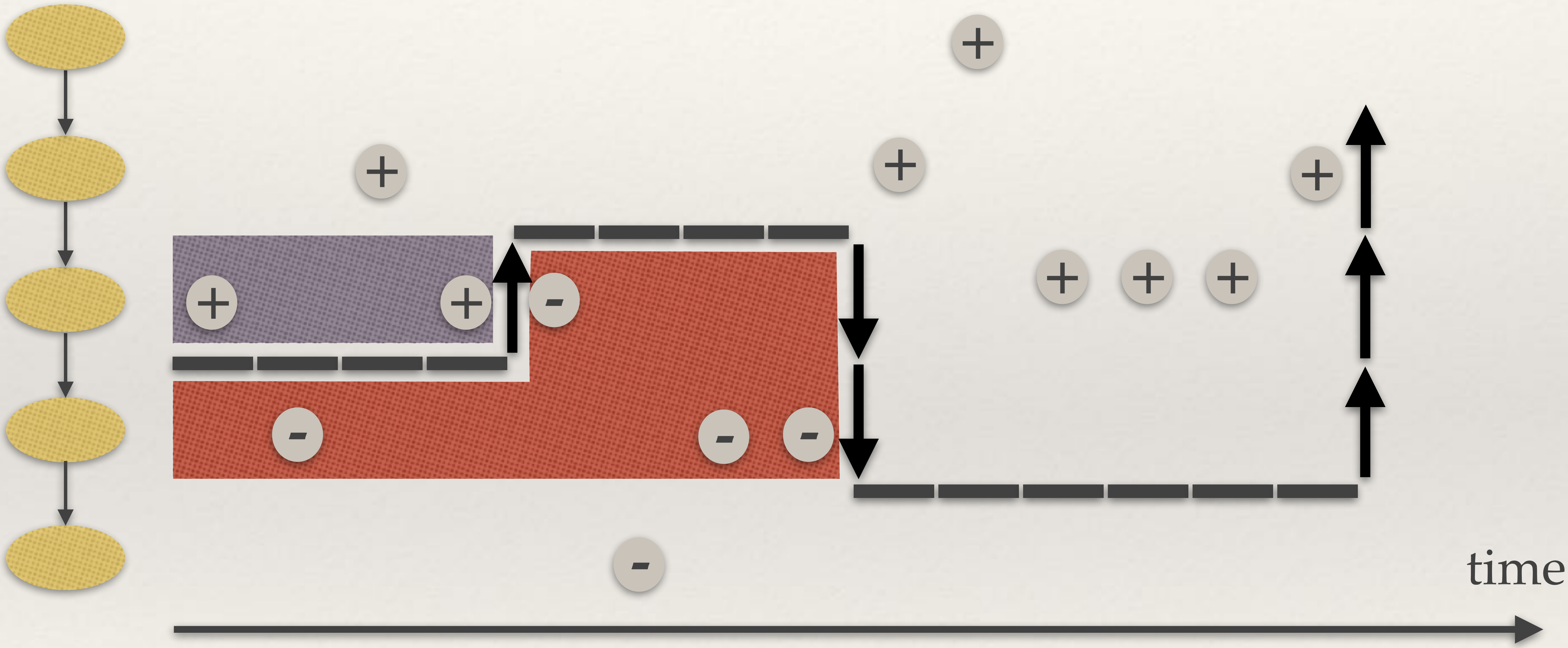
# Bounding cost of ALG



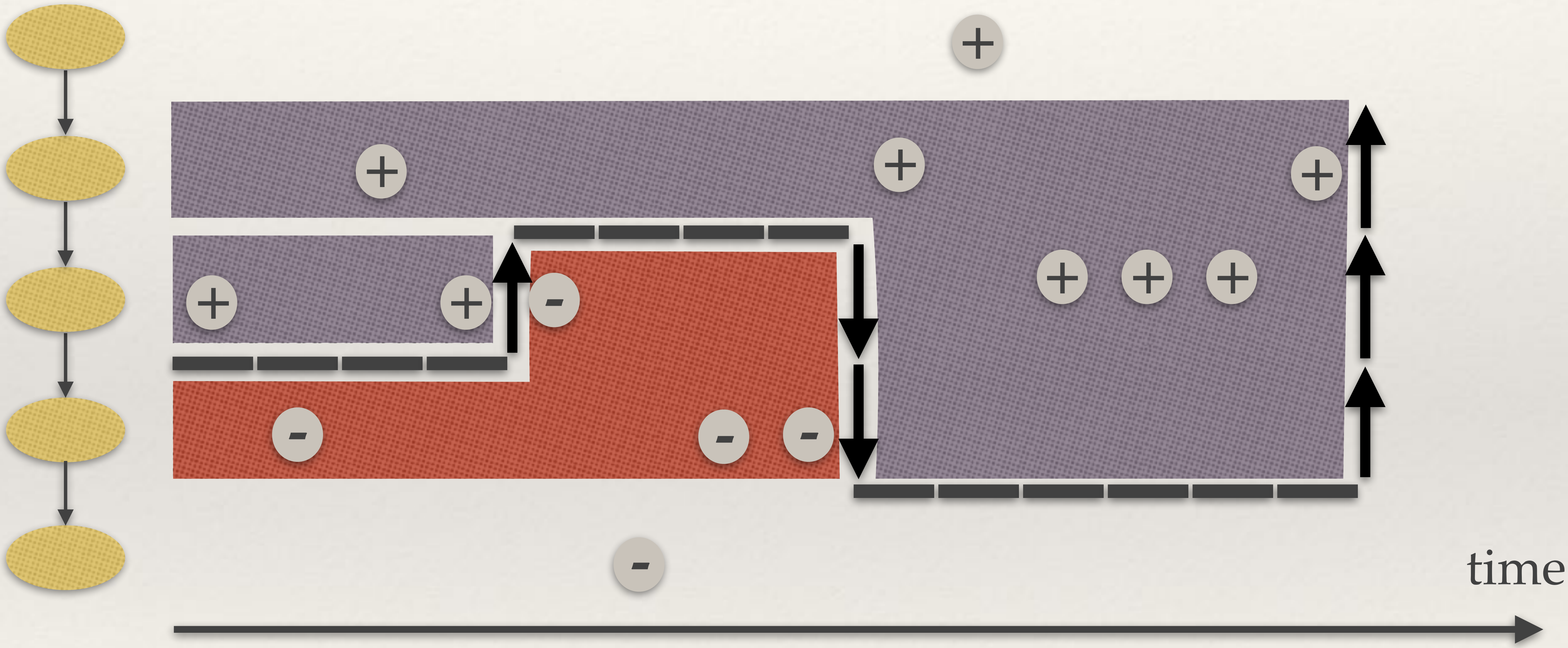
# Bounding cost of ALG



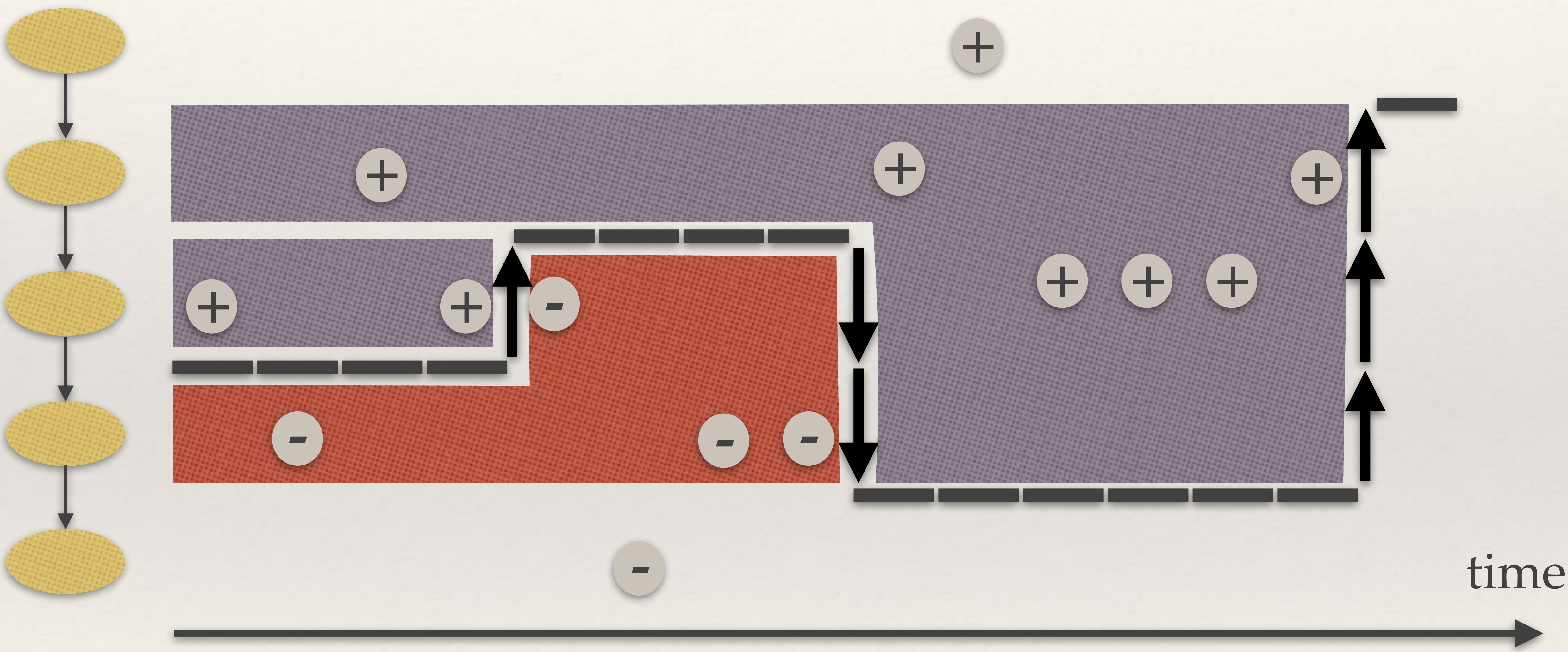
# Bounding cost of ALG



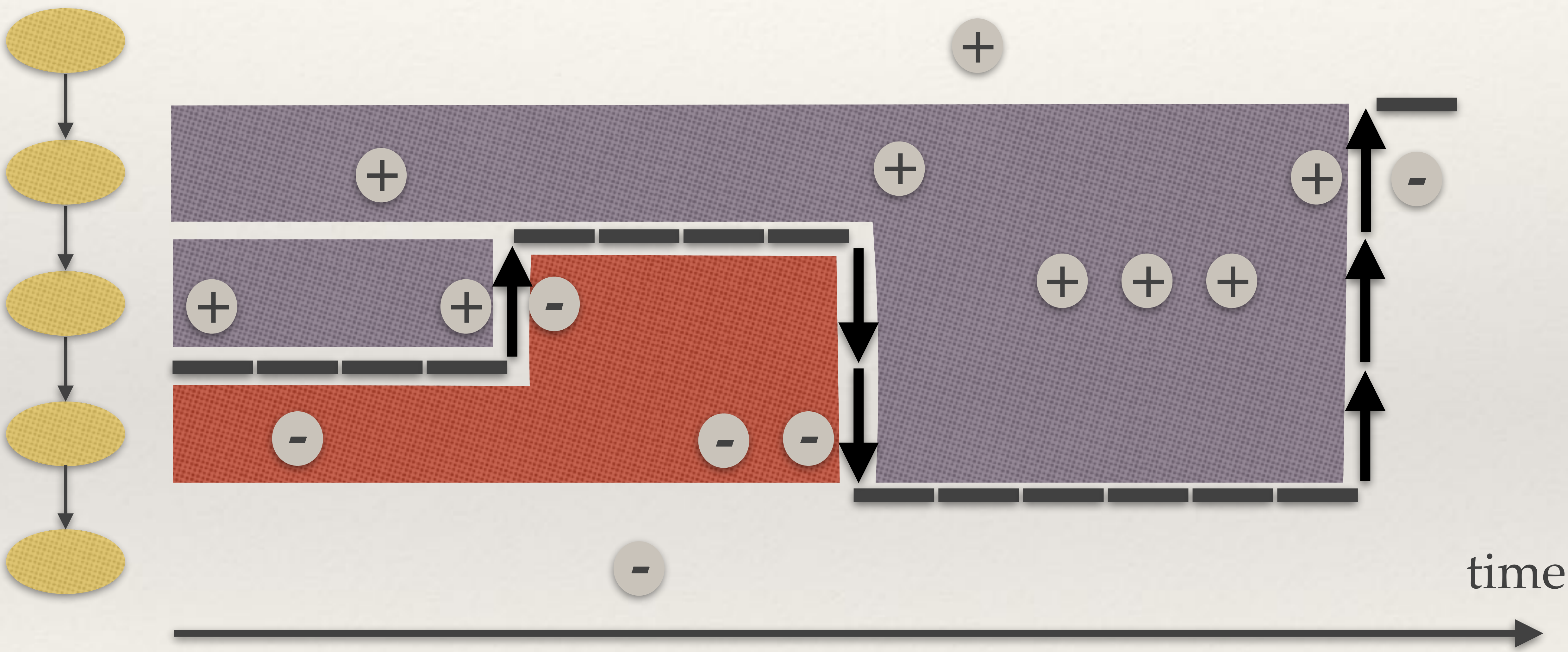
# Bounding cost of ALG



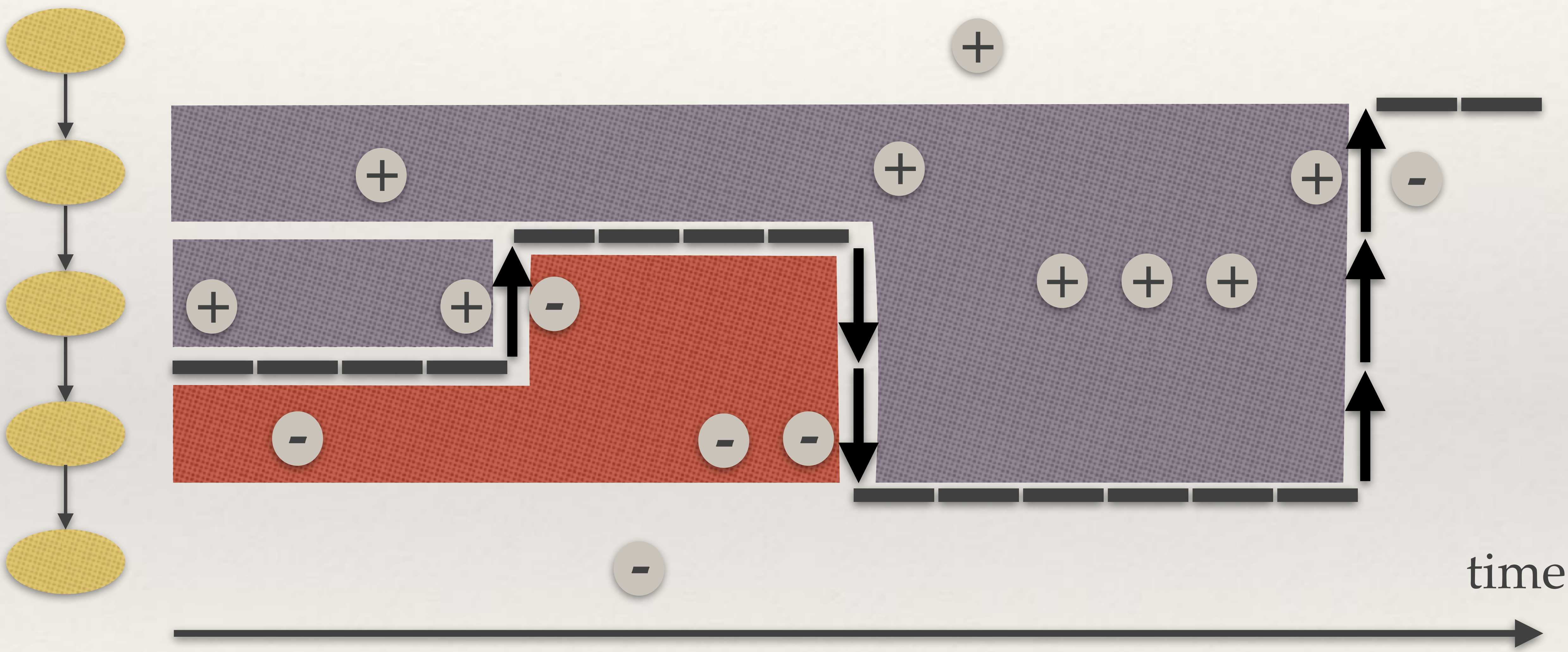
# Bounding cost of ALG



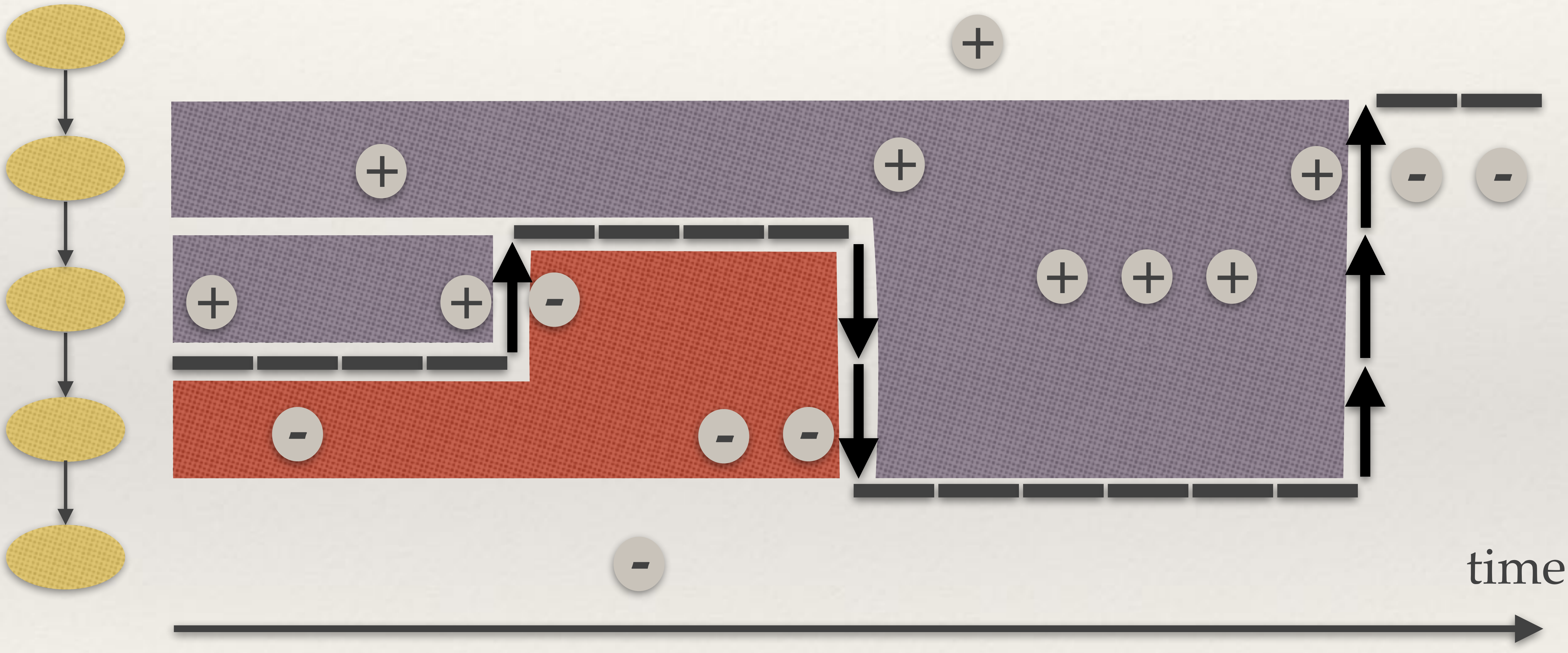
# Bounding cost of ALG



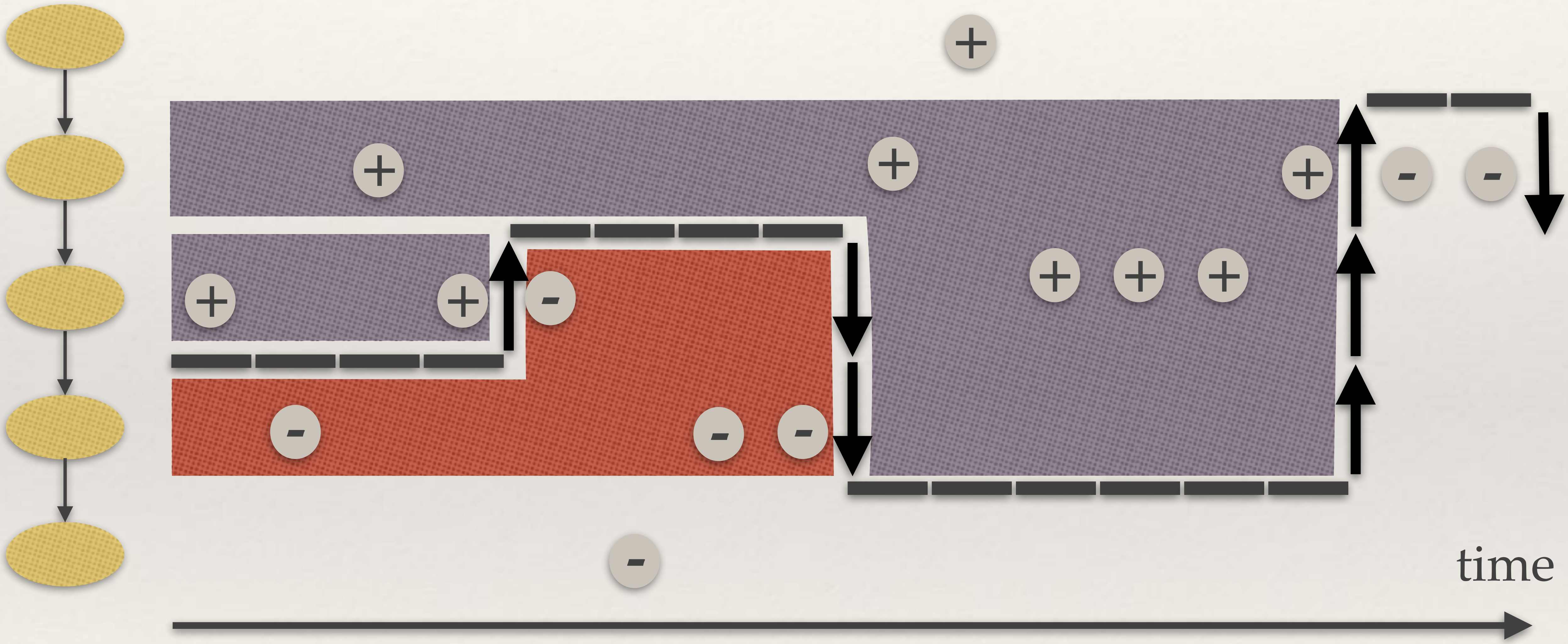
# Bounding cost of ALG



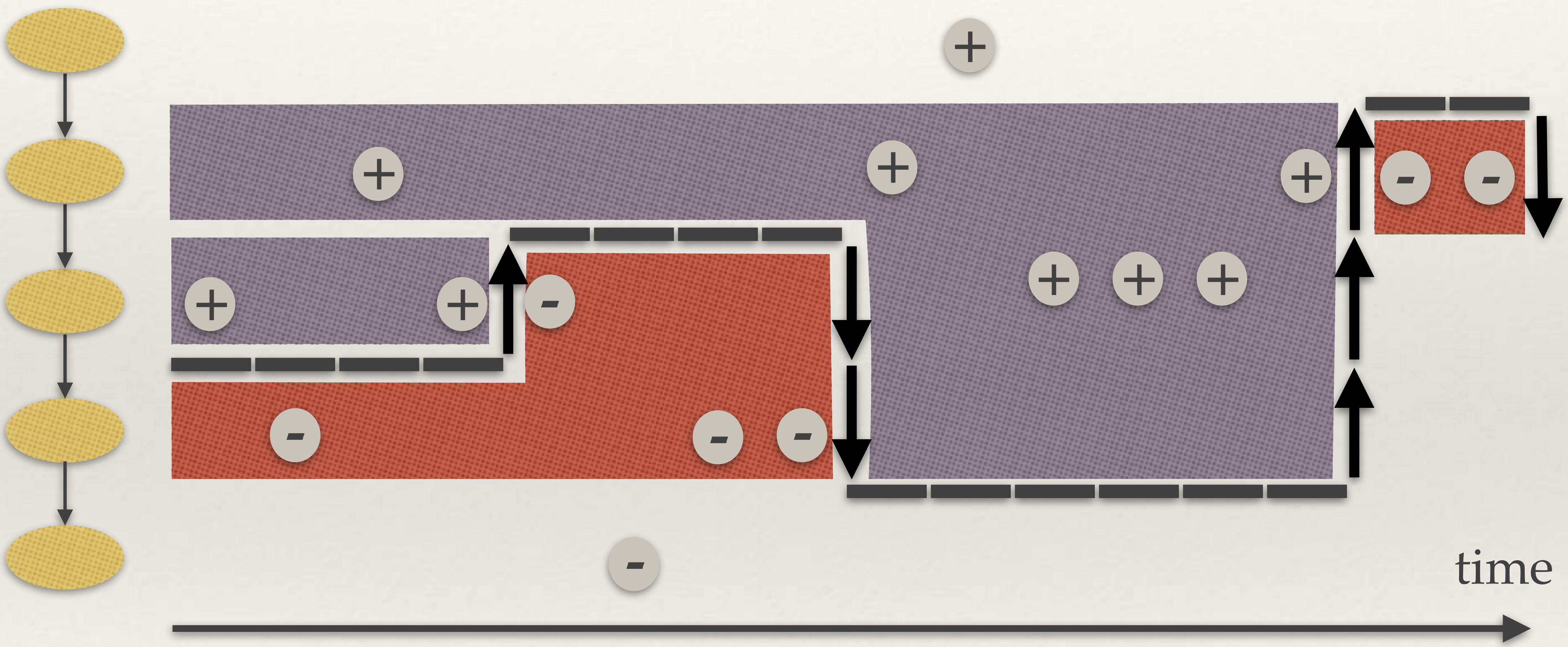
# Bounding cost of ALG



# Bounding cost of ALG



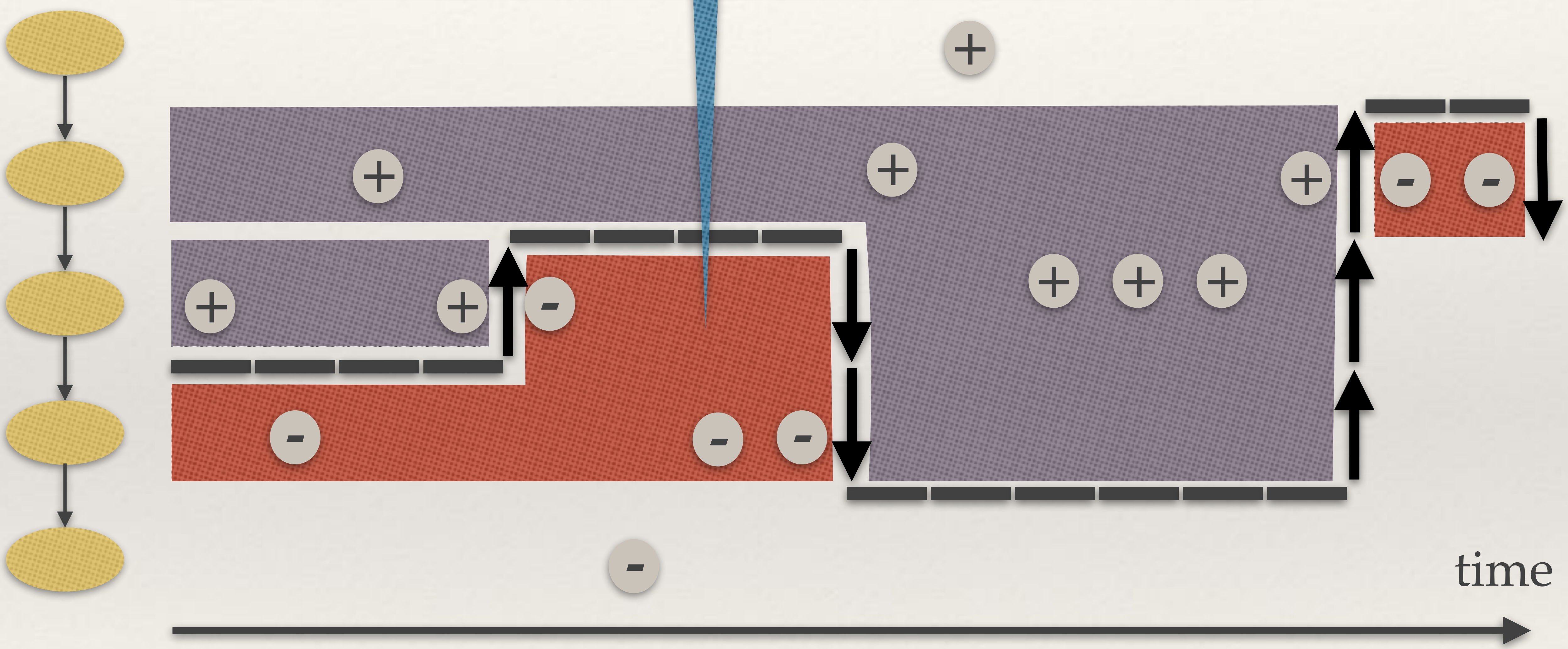
# Bounding cost of ALG



# Bounding cost of ALG

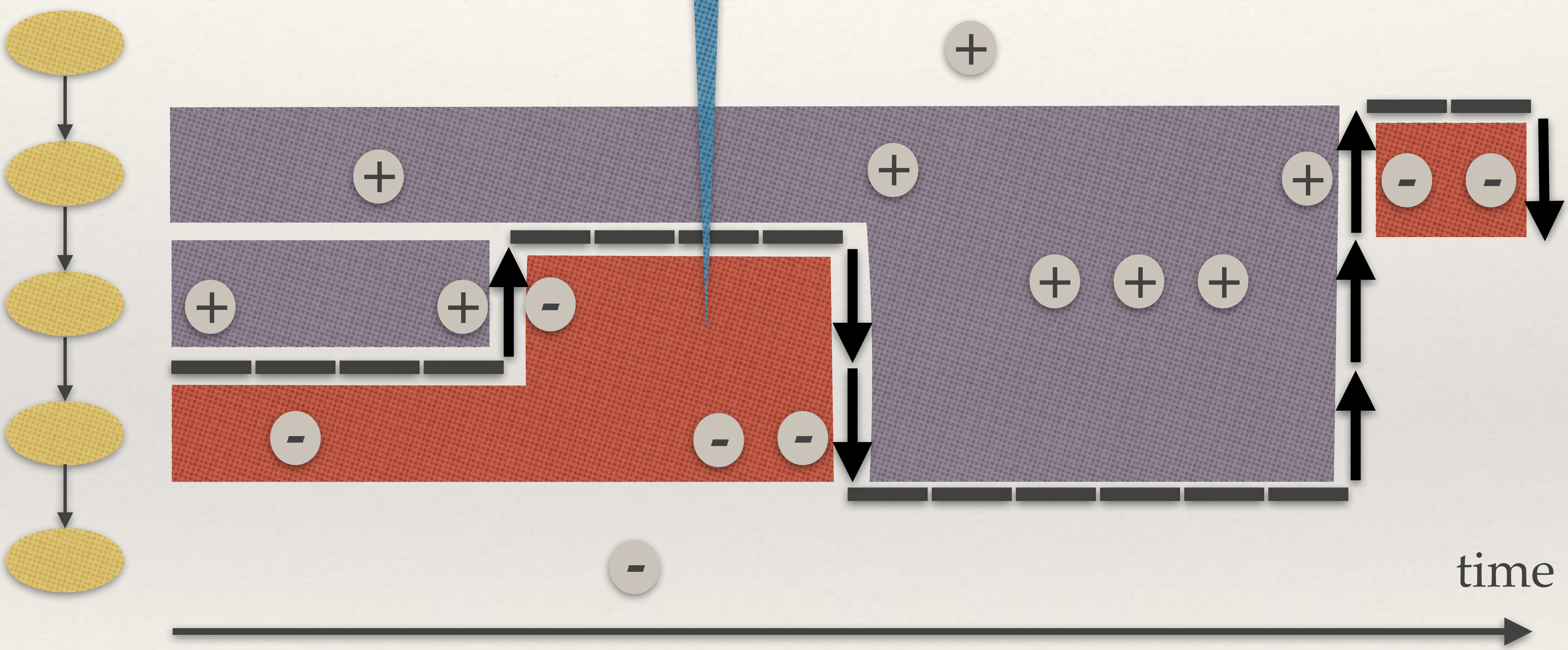
**For any field $F$, ALG pays:**

* $height(F) \cdot \alpha$ **for requests inside** $F$ **and**
* $height(F) \cdot \alpha$ **for cache change at the end of** $F$



# Bounding cost of ALG

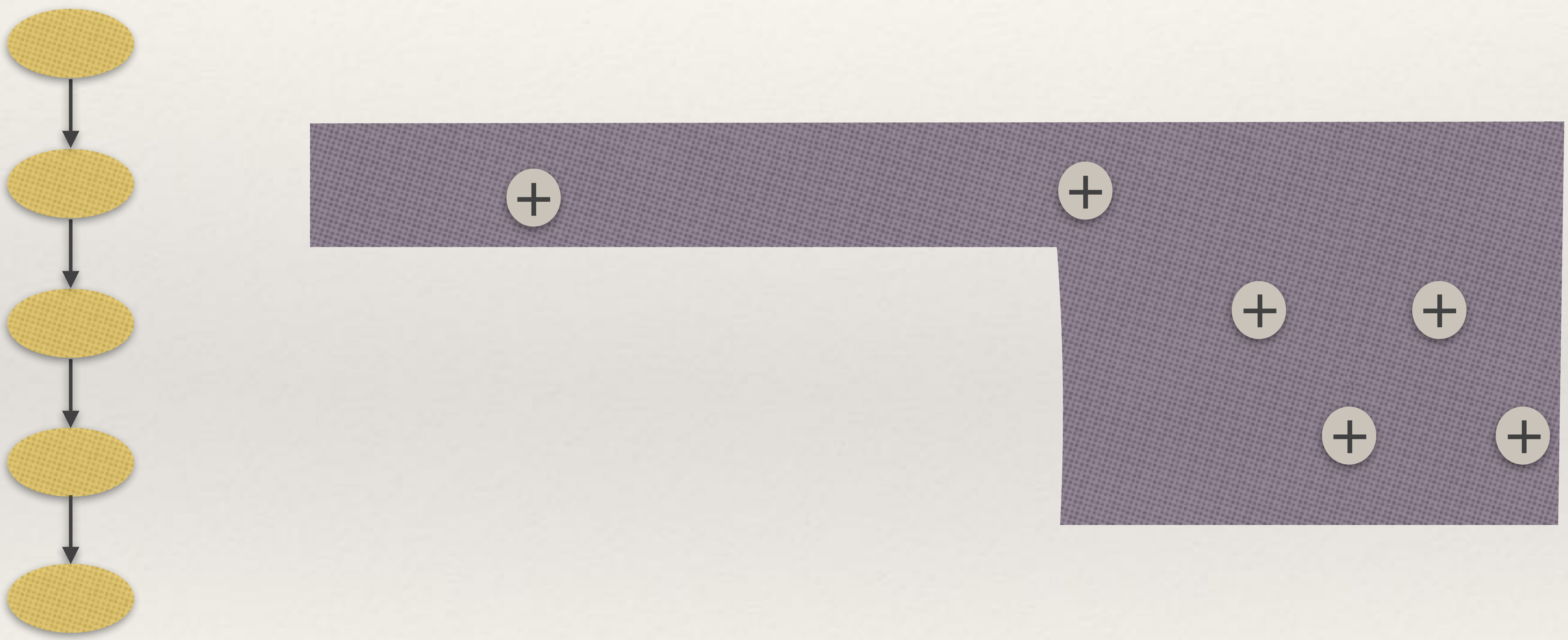
**For any field *F*, ALG pays:**
*** *height(F)* $\cdot\ \alpha$ for requests inside *F* and**
*** *height(F)* $\cdot\ \alpha$ for cache change at the end of *F***



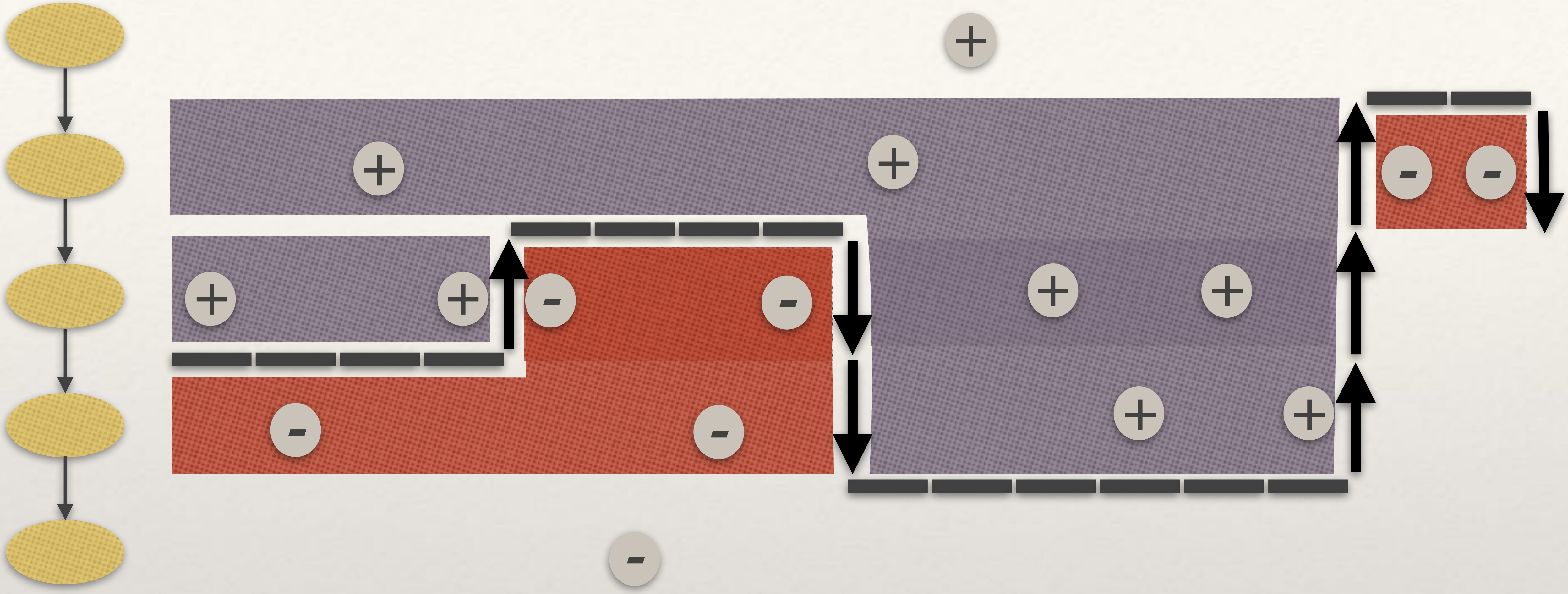

**On average: ALG pays $O(\alpha)$ for each arrow ( ↑ or ↓ )**

# Ideal fields

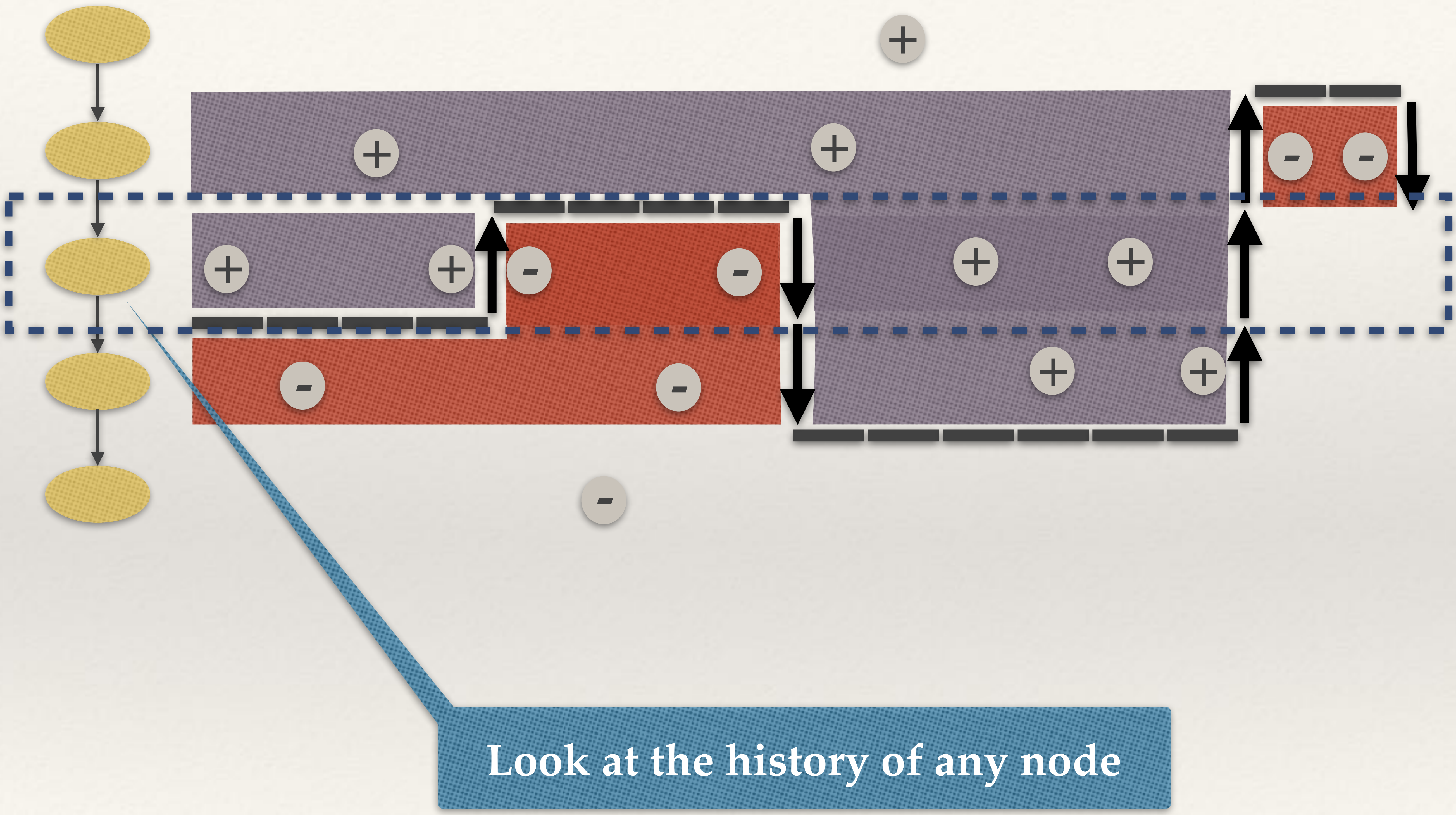
**Ideal field = each node receives exactly $\alpha$ requests.**



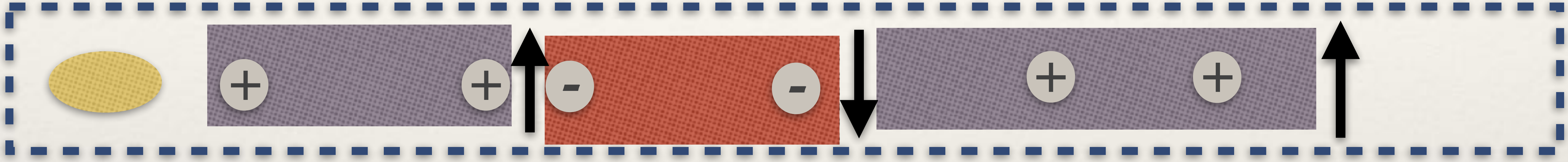
# OPT cost for ideal fields



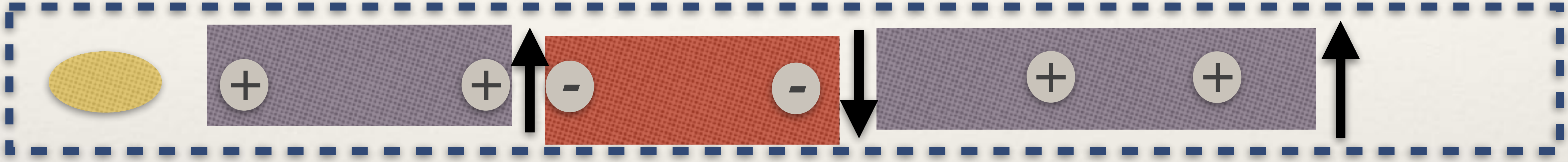
# OPT cost for ideal fields



# OPT cost for ideal fields

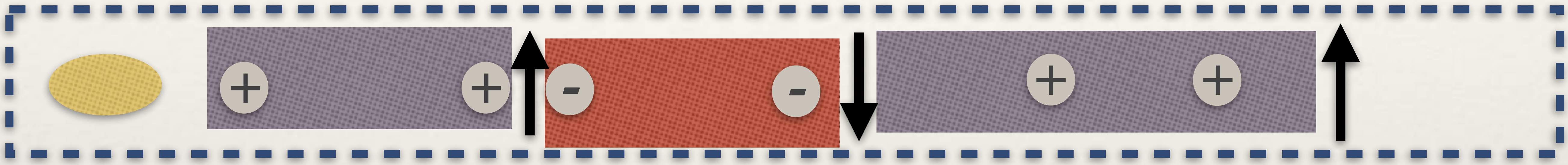


# OPT cost for ideal fields



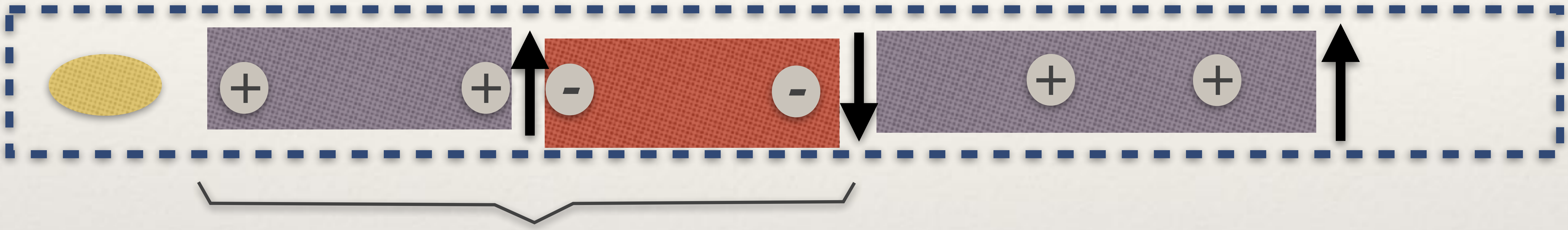
# OPT cost for ideal fields

- Recall: ALG pays $O(\alpha)$ for each arrow ( $\uparrow$ or $\downarrow$ )



# OPT cost for ideal fields

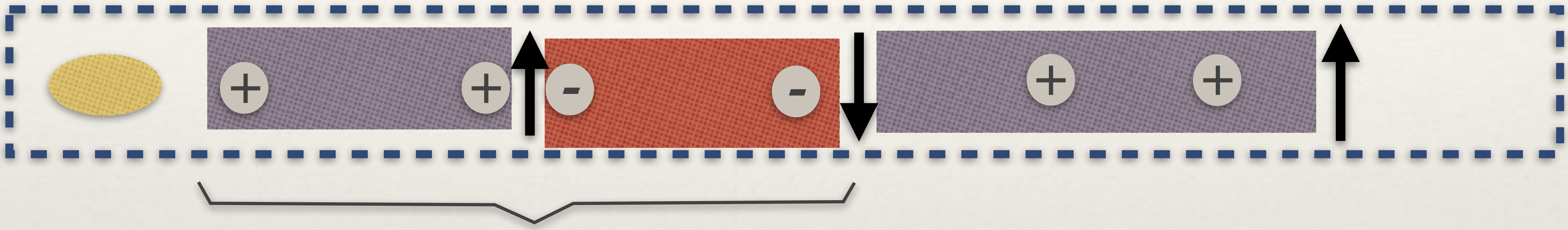
- Recall: ALG pays $O(\alpha)$ for each arrow ( $\uparrow$ or $\downarrow$ )




- OPT pays at least $\alpha$ here
  (for fetch, eviction, for positive or for negative requests)

# OPT cost for ideal fields

- Recall: ALG pays $O(\alpha)$ for each arrow ( $\uparrow$ or $\downarrow$ )



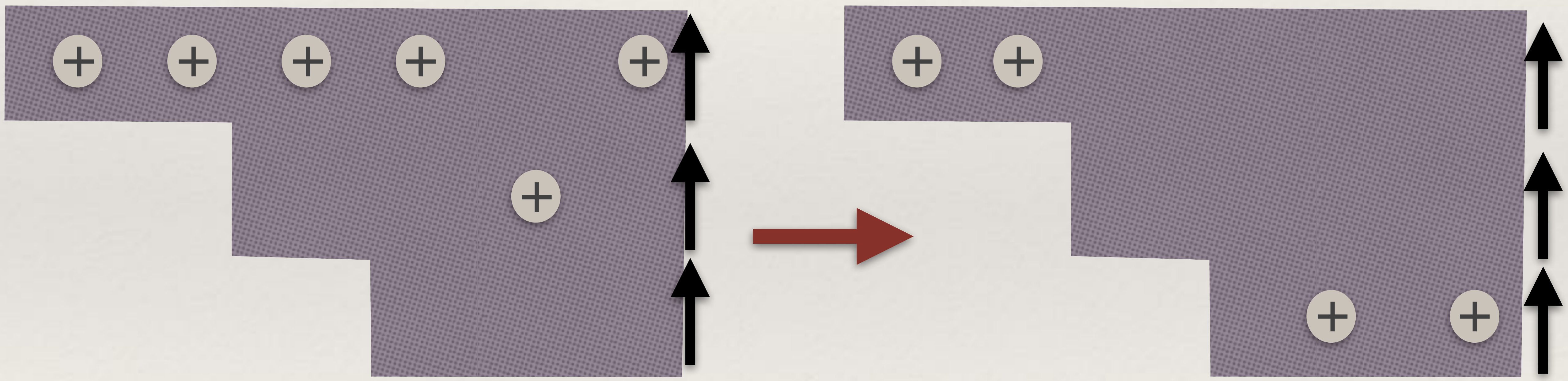

- OPT pays at least $\alpha$ here
  (for fetch, eviction, for positive or for negative requests)

**ALG is $O(1)$-competitive for input that induces ideal fields.**

# Arbitrary (non-ideal) fields

It is possible to **shift requests in each field**, so that:

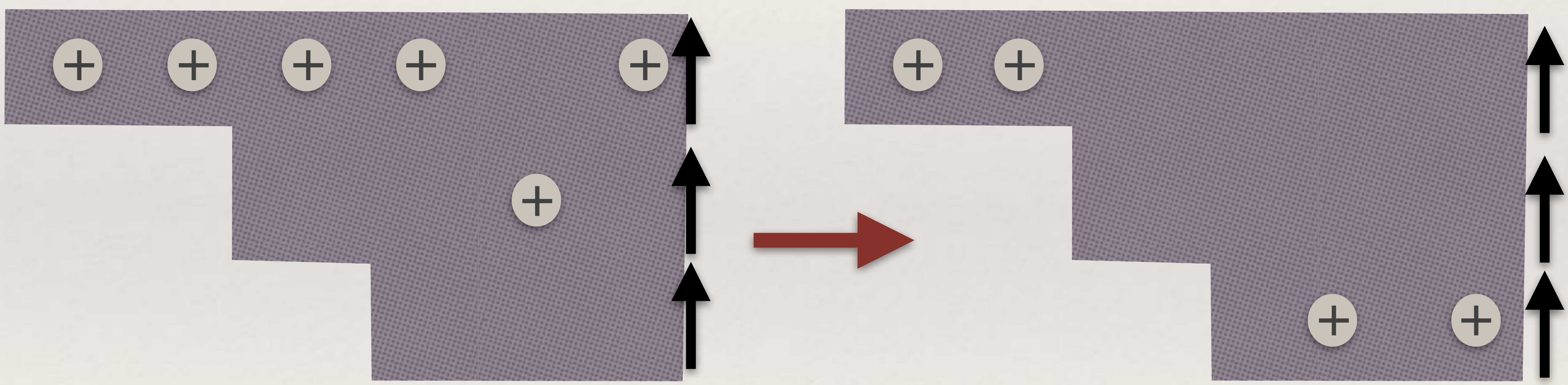
- the resulting sequence is not more difficult for OPT,
- the resulting field is ``**almost ideal**", i.e.,
  $\mathbf{\Omega(1\,/\,\mathit{height}(T))}$ of all nodes have $\Omega(\alpha)$ requests.



# Arbitrary (non-ideal) fields

It is possible to **shift requests in each field**, so that:

- the resulting sequence is not more difficult for OPT,
- the resulting field is ``**almost ideal**", i.e.,
  **$\Omega$(1 / *height*(*T*))** of all nodes have $\Omega(\alpha)$ requests.



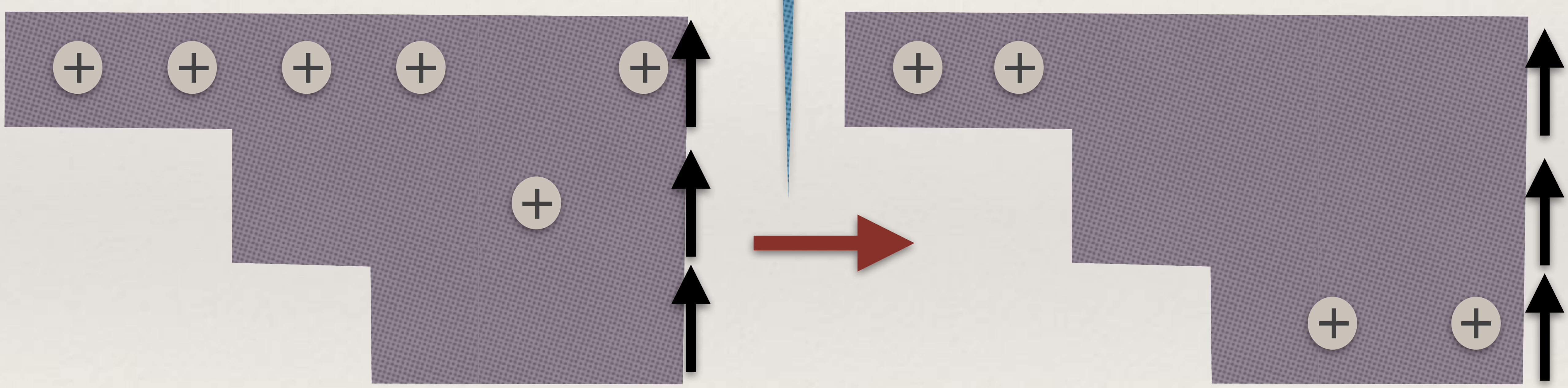
**ALG is *O*(*height*(*T*))-competitive for any input**

# Arbitrary (non-ideal) fields

It is possible to **shift requ**

“Requests density“ is higher on the top of the field. We shift requests down.

- the resulting sequence is not more difficult for OPT,
- the resulting field is ``**almost ideal**”, i.e., **Ω(1 / *height*(*T*))** of all nodes have Ω(*α*) requests.



**ALG is *O*(*height*(*T*))-competitive for any input**

# Outlook

- Tree caching problem = abstraction for FIB offloading.
- Simple, competitive counter-based algorithm.
- Algorithm can be implemented efficiently at the controller.

Thank you!

# Alternative solution: FIB compression

- Replacing the set of rules by **smaller and equivalent set**.

  **Draves, King, Venkatachary, Zill (INFOCOM ’99);**
  **Suri, Sandholm, Warkhede (Algorithmica ’03)**

# Alternative solution: FIB compression

- Replacing the set of rules by **smaller and equivalent set**.

  **Draves, King, Venkatachary, Zill (INFOCOM '99);**
  **Suri, Sandholm, Warkhede (Algorithmica '03)**

- **Problematic in presence of updates** (thousands rule updates / sec.)
  - Systems-oriented approaches

    **Liu, Zhao, Nam, Wang, Zhang (GLOBECOM '10); Zhao, Liu, Wang, Zhang (INFOCOM '10); Uzmi, Nebel, Tariq, Jawad, Chen, Shaikh, Wang, Francis (CoNEXT '11); Karpilovsky, Caesar, Rexford, Shaikh, Merwe (Trans. Netw Serv. Manag. '12); Liu, Zhang, Wang (INFOCOM '13); Luo, Xie, Salamatian, Uhlig, Mathy, Xie (INFOCOM '13); Rétvári, Tapolcai, Korösi, Majdán, Heszberger (SIGCOMM '13), …**

  - Analytic (competitive-ratio based) approaches

    **B., Schmid (SIROCCO '13); B., Sarrar, Schmid, Uhlig. (ICDCS '14)**