CS 4102: Algorithms

Lecture 18: Greedy Algorithms

David Wu

Fall 2019

Warm-Up

Why is an algorithm’s space complexity important?

Why might a memory-intensive algorithm be undesirable?

Disadvantages of Large Memory Complexity

Using too much memory forces you to use slow memory

Memory is expensive

May have too little memory for the algorithm to even run

Lots of memory hinders parallelism

Contention for the memory

Memory $\leq$ time

Today’s Keywords

Greedy Algorithms

Choice Function

Cache Replacement

Hardware & Algorithms

CLRS Readings: Chapter 16

Homework

HW6 due **Tuesday, November 5, 11pm**

- Dynamic programming and greedy algorithms
- Written (use LaTeX!) – Submit <u>both</u> **zip** and **pdf** (two <u>separate</u> attachments)!

HW10A also due **Tuesday, November 5, 11pm**

- No late submissions allowed

Review: Huffman Encoding

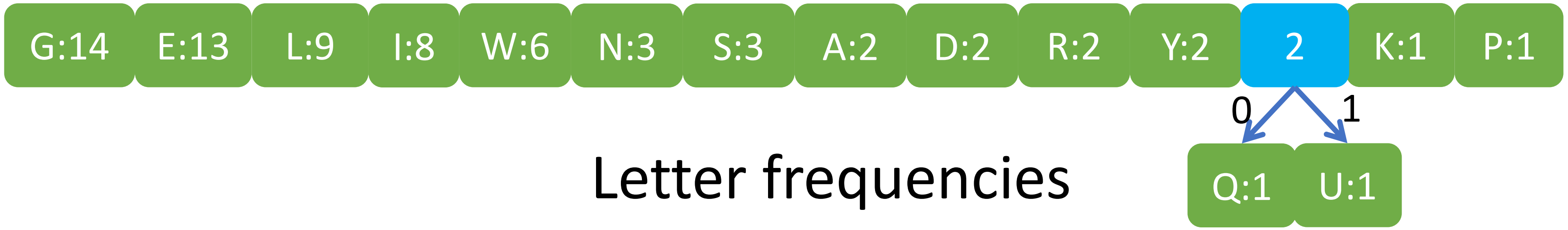
Choose the least frequent pair, combine into a subtree

G:14	E:13	L:9	I:8	W:6	N:3	S:3	A:2	D:2	R:2	Y:2	K:1	P:1	Q:1	U:1

Letter frequencies

Review: Huffman Encoding

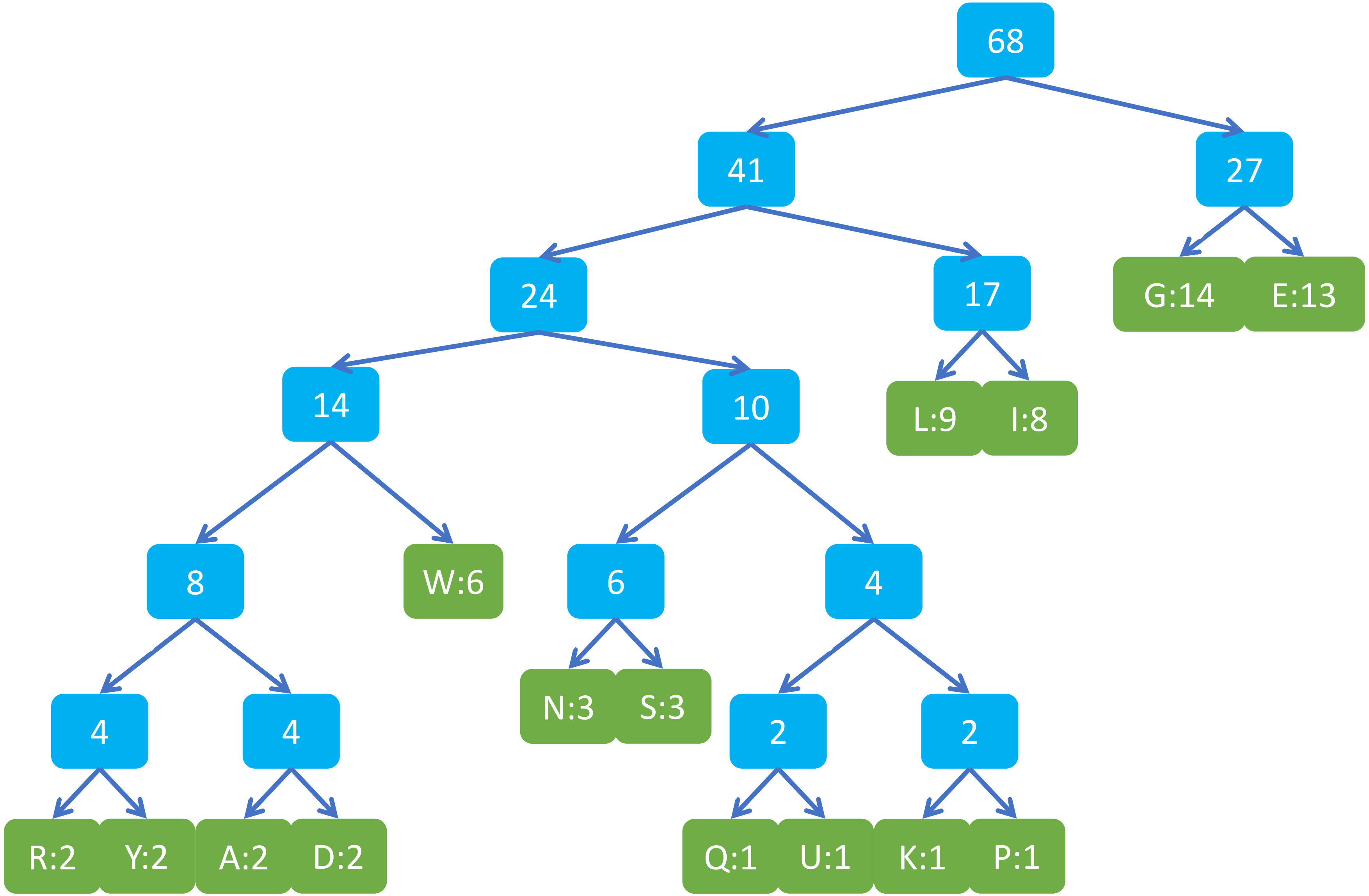
Choose the least frequent pair, combine into a subtree



Letter frequencies

Subproblem of size $n-1$!

Review: Huffman Encoding



Review: Optimality of Huffman Encoding

Proof Idea:

- Show that there is an optimal tree where the least frequent characters are siblings
  - Exchange argument

Greedy choice property

- Show that making them siblings and solving the new smaller sub-problem results in an optimal solution
  - Proof by contradiction

Optimal substructure

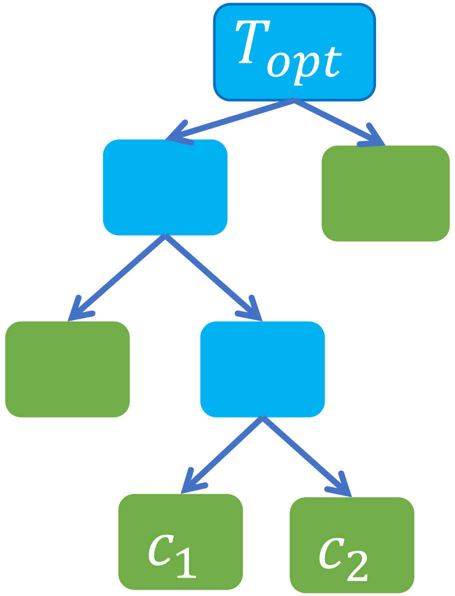
Huffman Analysis: Exchange Argument

Claim: if c_1, c_2 are the least-frequent characters, then there is an optimal prefix-free code where c_1, c_2 are siblings

- **Equivalently:** encodings of c_1, c_2 have the same length and differ only in their last bit

Proof. Consider some optimal tree T_{opt}

Case 1: Suppose c_1, c_2 are siblings in T_{opt}. Then claim holds



Huffman Analysis: Exchange Argument

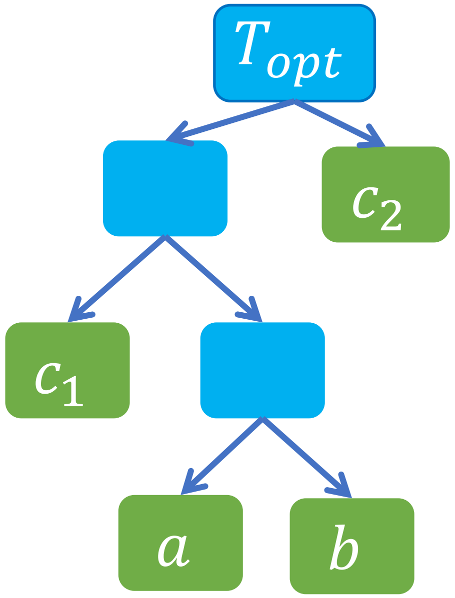
Claim: if c_1, c_2 are the least-frequent characters, then there is an optimal prefix-free code where c_1, c_2 are siblings

- **Equivalently:** encodings of c_1, c_2 have the same length and differ only in their last bit

Proof. Consider some optimal tree T_{opt}

Case 2: Suppose c_1, c_2 are not siblings in T_{opt}

Optimal tree must be full (every non-leaf node has two children); otherwise, can move a leaf node up and reduce the encoding size



Let a, b be sibling leaves of maximum depth

Why must this exist?

Exchange argument: Since $f_{c_1} \leq f_a$ and $f_{c_2} \leq f_b$, swapping c_1 with a (and c_2 with b) cannot increase the cost of the tree

Huffman Analysis: Optimal Substructure

Claim: An optimal solution for F involves finding an optimal solution for F', then adding c_1, c_2 as children to σ

Proof by contradiction: If there is a better solution for F, then can use that to obtain a better solution for F', which contradicts optimality of solution for F'

Caching Problem

Why is an algorithm's space complexity important?

Why might a memory-intensive algorithm be undesirable?

von Neumann Bottleneck

Reading from memory is <u>slow</u>

Big memory = slow memory

Solution: hierarchical memory

Takeaway for algorithms: More memory accesses means bigger runtime

Hopefully your data in here — CPU, registers — 0.2 ns

If not look here — L1 Cache — 1 ns

If not look here — L2 Cache — 4 ns

If not look here — L3 Cache — 20 ns

If not look here — Main Memory — 100 ns

Disk seek: 10 ms

Caching Problem

Cache misses are very expensive

When we load something new into cache, we must eliminate something already there

We want the best cache "schedule" to minimize the number of misses

Caching Problem Definition

Input:

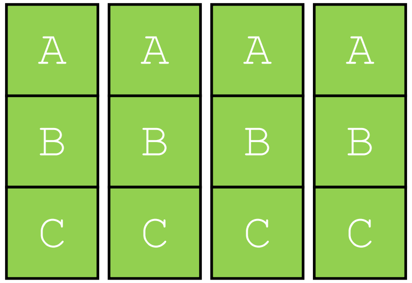
- k = size of the cache
- $M = [m_1, m_2, \dots m_n]$ = memory access pattern

Output:

- "Schedule" for the cache (list of items in the cache at each time) which minimizes cache misses

Caching Example

Cache contents



Must evict something to make room for D

A B C D A D E A D B A E C E A

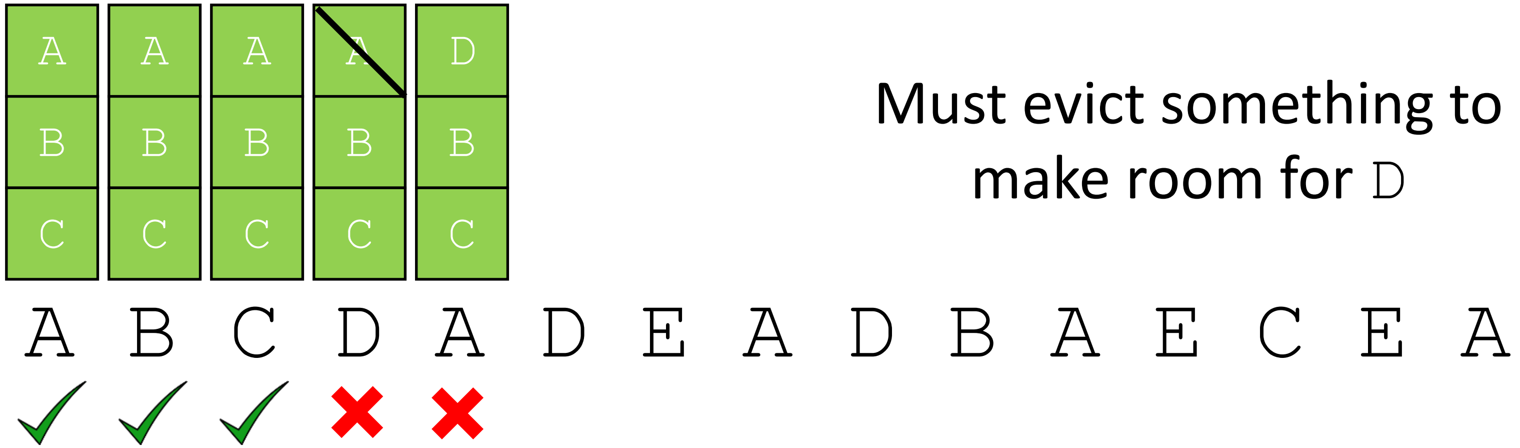
✓ ✓ ✓ ✗

Sequence of cache accesses

Caching Example

Suppose we evict A

Cache contents



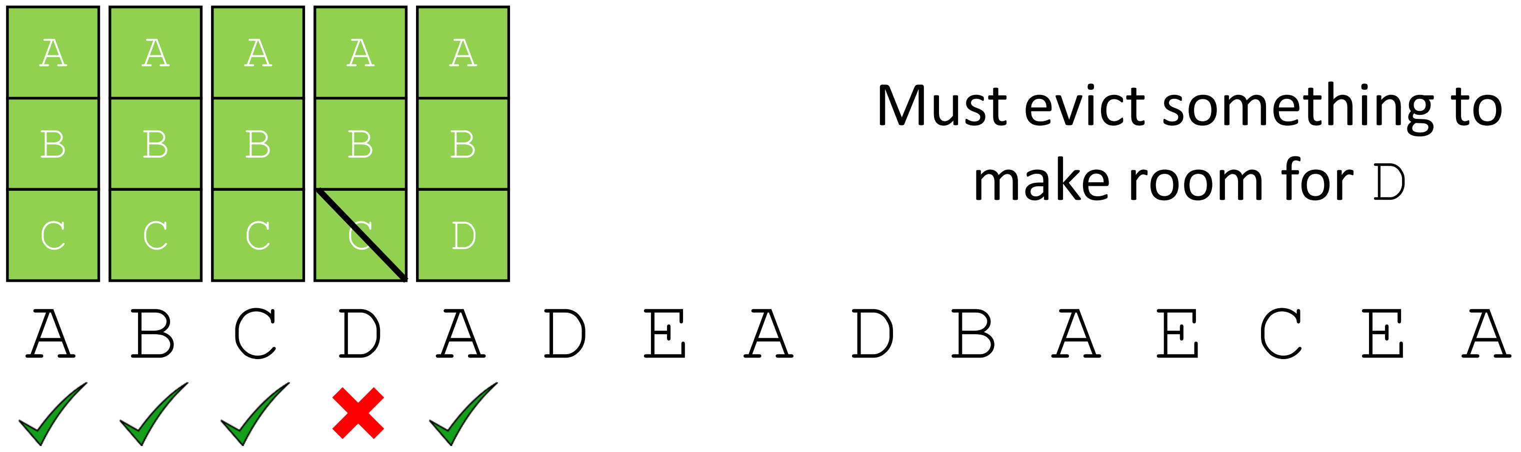
Must evict something to make room for D

Sequence of cache accesses

Caching Example

Suppose we evict C

Cache contents



Must evict something to make room for D

Sequence of cache accesses

Objective: Devise cache eviction strategy to minimize number of cache misses

Simplifying assumption: We know the entire sequence of accesses ahead of time

(valid assumption for data-oblivious computations)

Greedy Algorithms

Requires optimal substructure

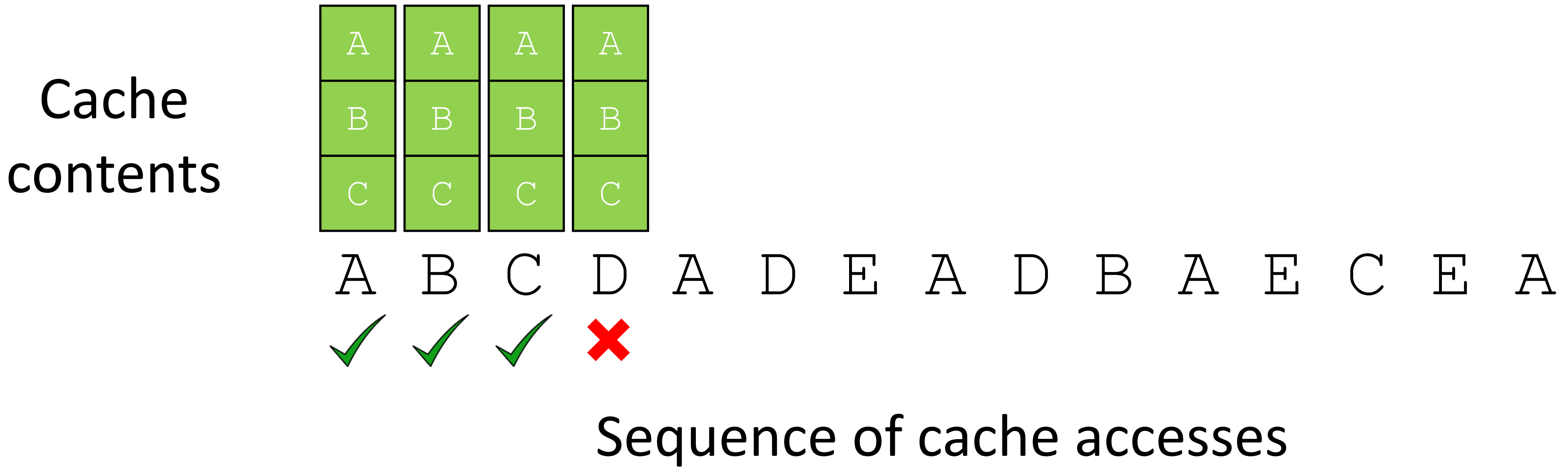
- Solution to larger problem contains the solution to a smaller one
- Only a single subproblem to consider

General Blueprint:

1. Identify a greedy choice property
   - Show that this choice is guaranteed to be included in some optimal solution
2. Repeatedly apply the choice property until no subproblems remain

Greedy Strategy

Belady eviction policy: Evict the item accessed farthest in the future



Greedy Strategy

Belady eviction policy: Evict the item accessed farthest in the future

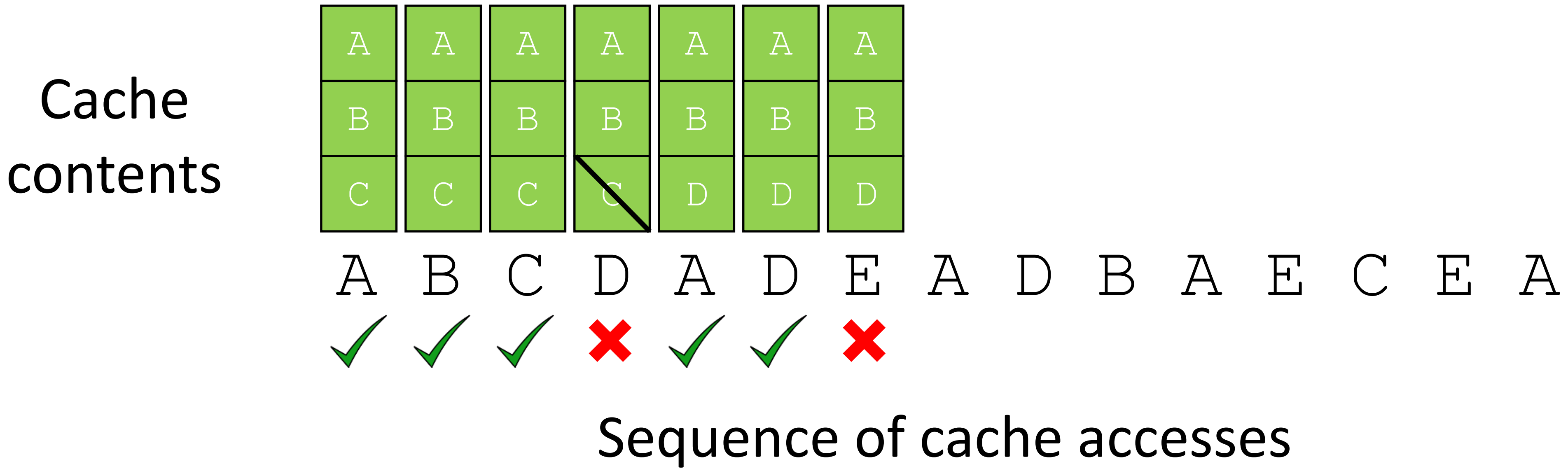
Cache contents

Sequence of cache accesses

Greedy choice: Evict C

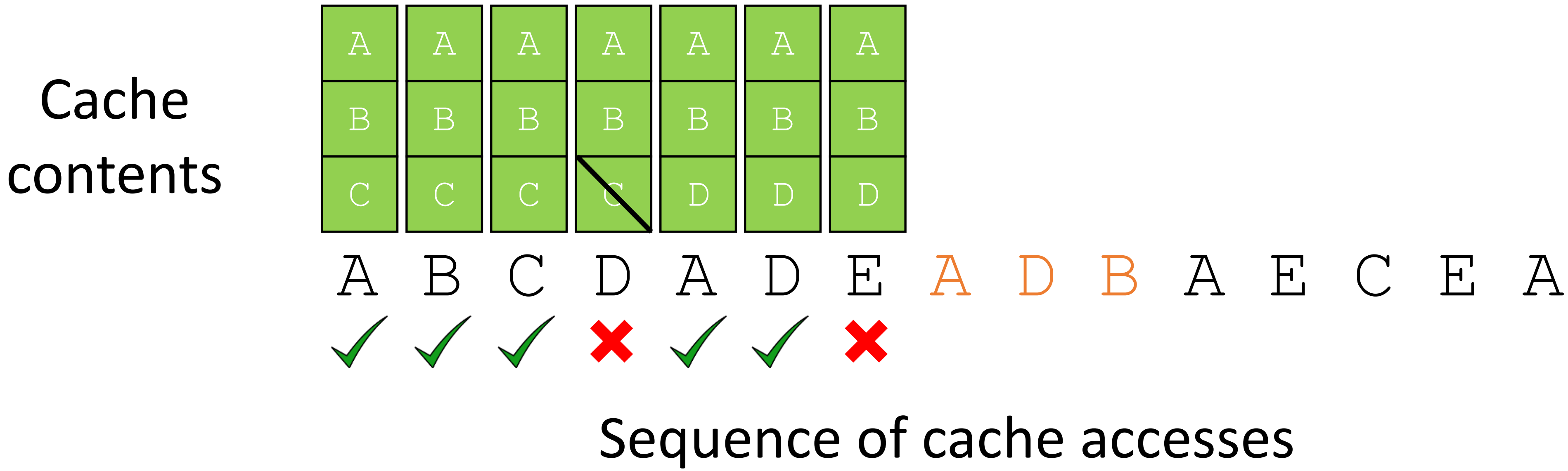
Greedy Strategy

Belady eviction policy: Evict the item accessed farthest in the future



Greedy Strategy

Belady eviction policy: Evict the item accessed farthest in the future

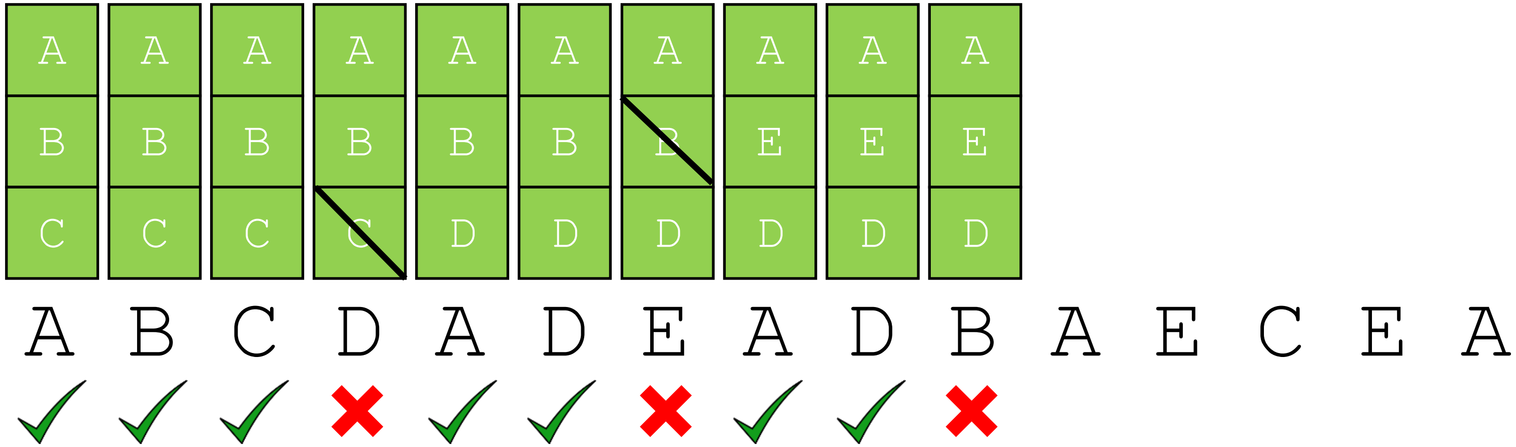


Greedy choice: Evict B

Greedy Strategy

Belady eviction policy: Evict the item accessed farthest in the future

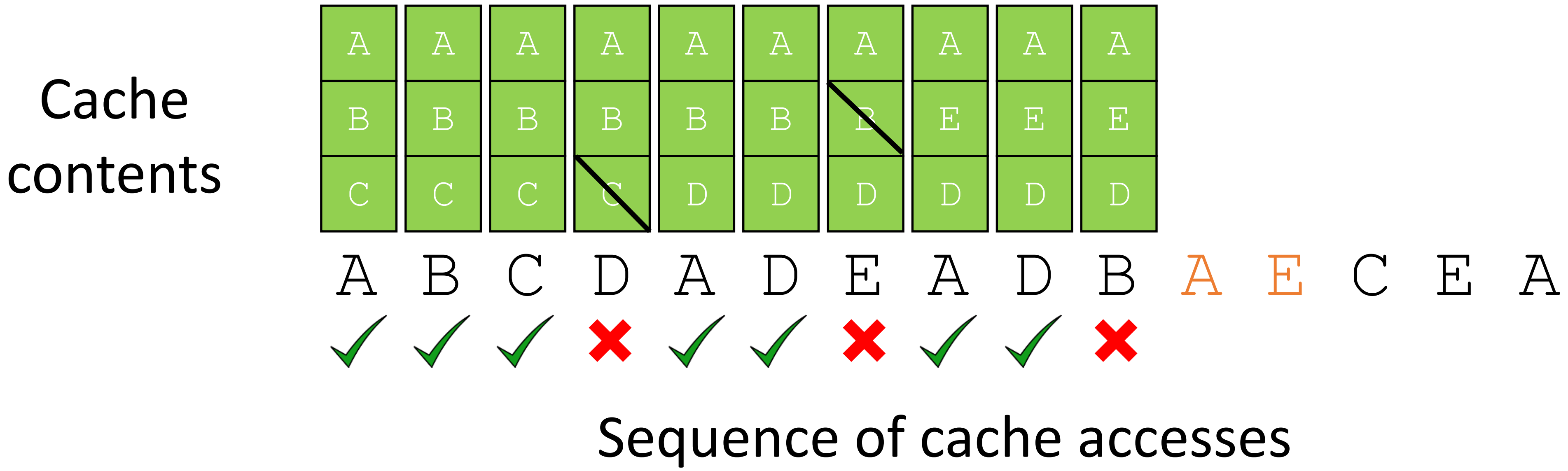
Cache contents



Sequence of cache accesses

Greedy Strategy

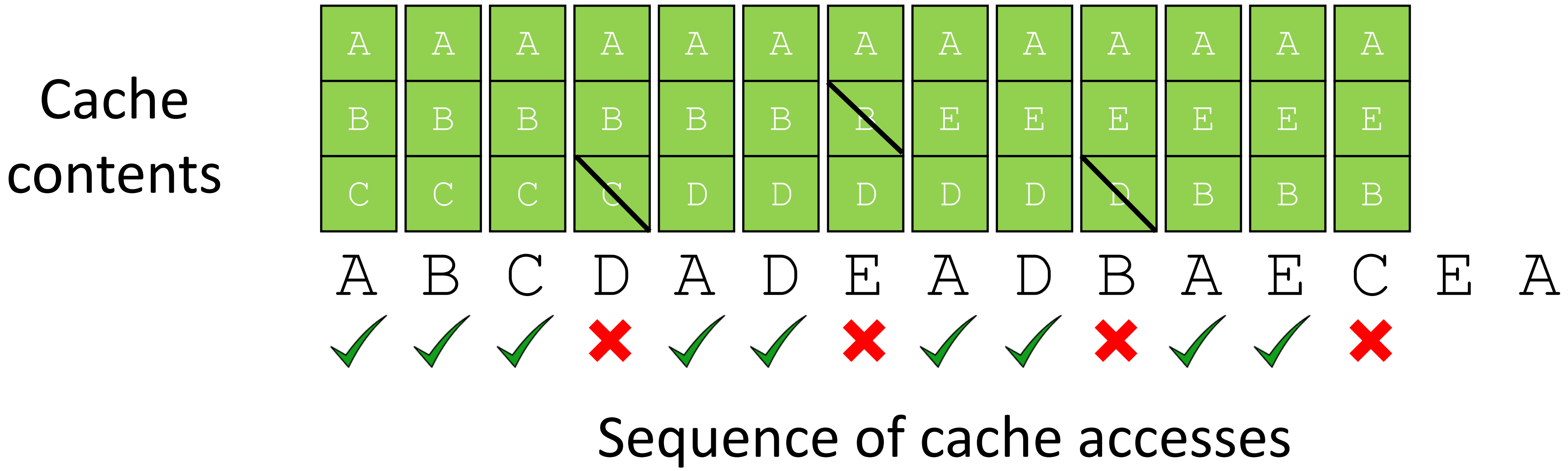
Belady eviction policy: Evict the item accessed farthest in the future



Greedy choice: Evict D

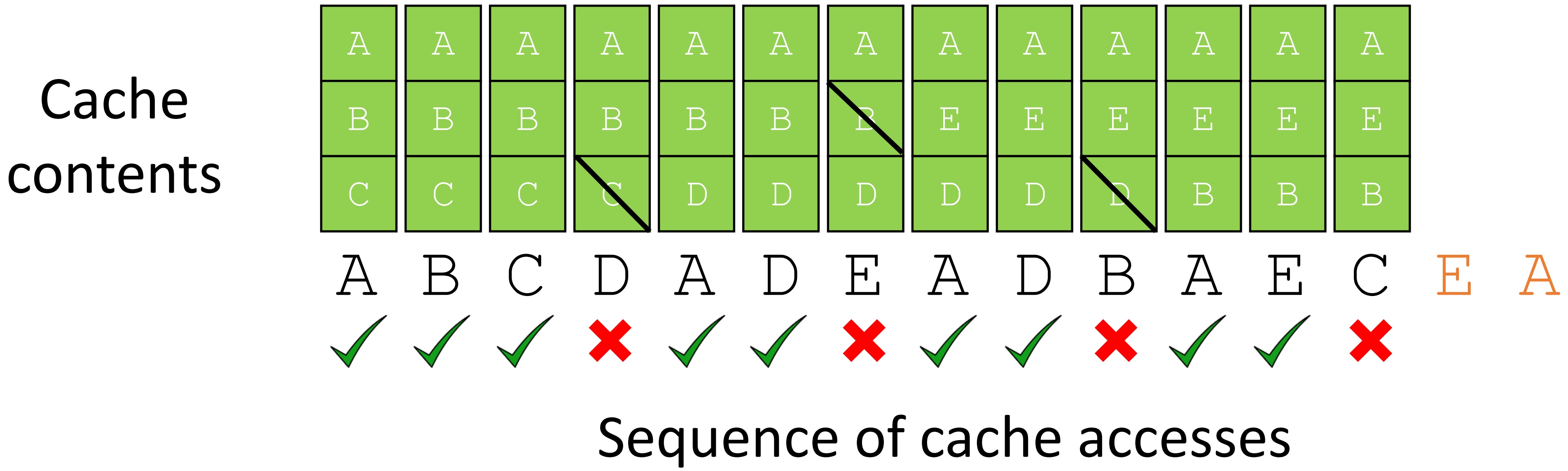
Greedy Strategy

Belady eviction policy: Evict the item accessed farthest in the future



Greedy Strategy

Belady eviction policy: Evict the item accessed farthest in the future

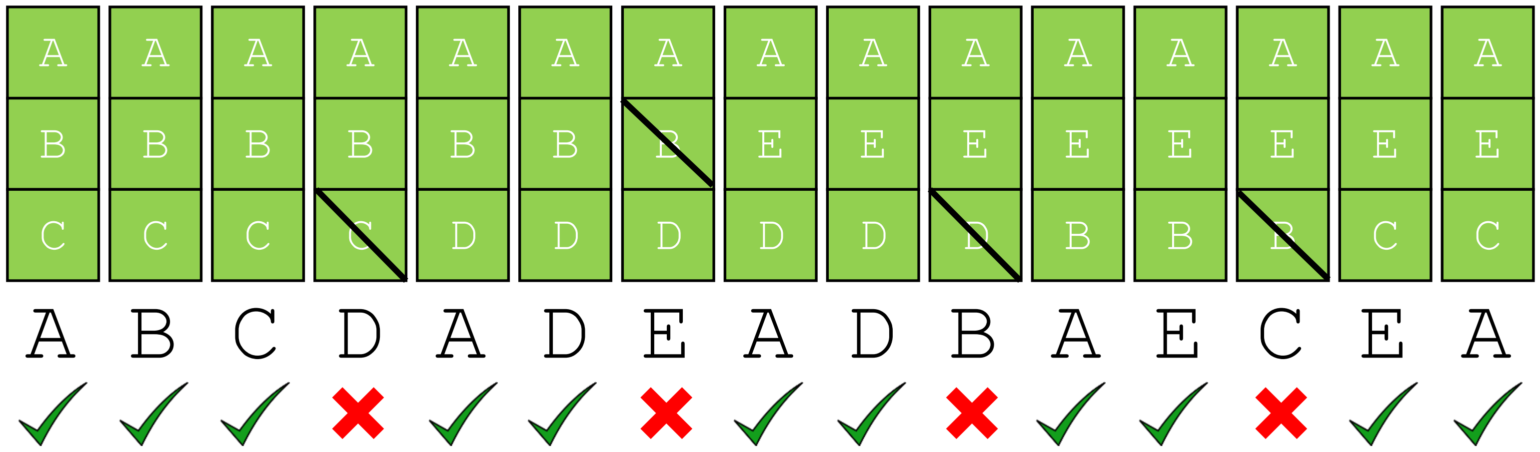


Greedy choice: Evict B

Greedy Strategy

Belady eviction policy: Evict the item accessed farthest in the future

Cache contents



Sequence of cache accesses

Total number of cache misses: 4

Greedy Algorithm

```
init cache = first k accesses
for each i = 1,...,n:
    if m[i] in cache:
        print cache
    else:
        z = furthest-in-future from cache
        evict z, add m[i] to cache
        print cache
```

`m[i]`: element accessed on i^{th} step

Running Time of Greedy Algorithm

`init cache = first k accesses`	$O(k)$
`for each i = 1,...,n:`	n times
`    if m[i] in cache:`	$O(k)$
`        print cache`	$O(k)$
`    else:`	
`        z = furthest-in-future from cache`	$O(kn)$
`        evict z, add m[i] to cache`	$O(1)$
`        print cache`	$O(k)$

Overall runtime: $O(kn^2)$

Proof of Correctness: Exchange Argument

Common technique to show correctness of a greedy algorithm

General idea: argue that at every step, the greedy choice is part of some optimal solution

Approach: Start with an arbitrary optimal solution and show that exchanging an item from the optimal solution with your greedy choice makes the new solution no worse (i.e., the greedy choice is as good as the optimal choice)

Exchange Argument for Belady Caching Algorithm

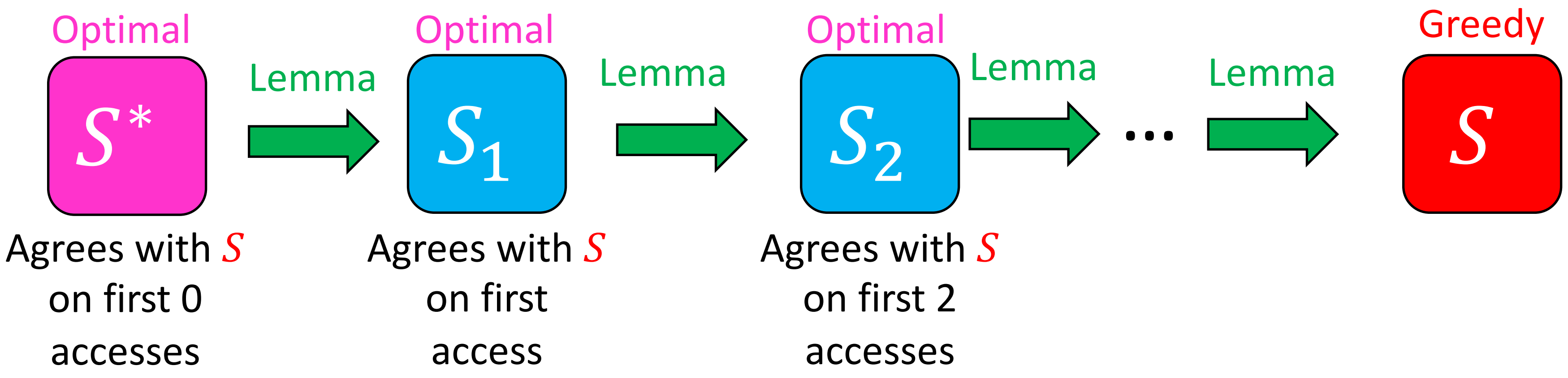
Let S be the schedule chosen by the greedy algorithm

Let S^* be any optimal schedule (that minimizes the number of cache misses)

Lemma: If S_i and S agree on the first i accesses, then there is a schedule S_{i+1} that agrees with S on the first $i+1$ accesses such that

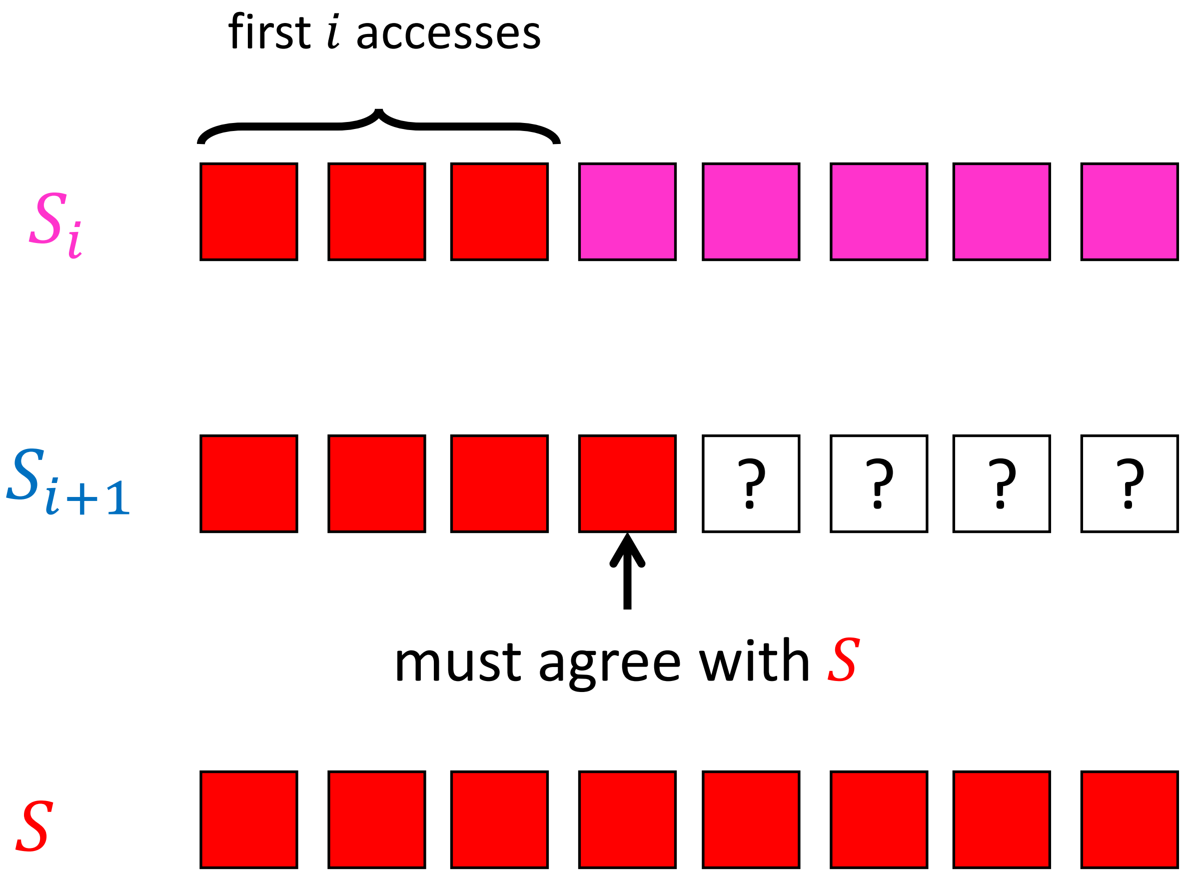
$$\text{misses}(S_{i+1}) \le \text{misses}(S_i)$$

Correctness then follows by induction:



$$\text{misses}(S) = \text{misses}(S_n) \le \text{misses}(S_{n-1}) \le \cdots \le \text{misses}(S_0) = \text{misses}(S^*)$$

Exchange Argument for Belady Caching Algorithm

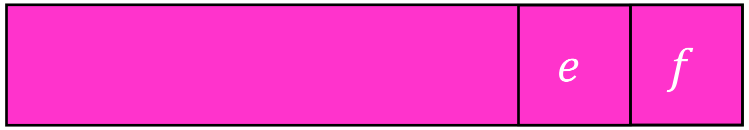


Goal: Need to fill in the rest of S_{i+1} to have no more cache misses than S_i

Exchange Argument for Belady Caching Algorithm

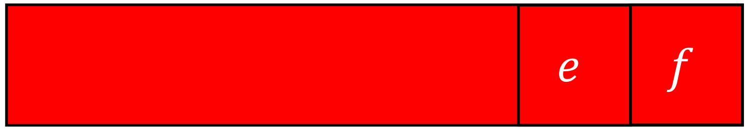
Lemma: If S_i and S agree on the first i accesses, then there is a schedule S_{i+1} that agrees with S on the first $i+1$ accesses such that $\text{misses}(S_{i+1}) \leq \text{misses}(S_i)$

Since S_i agrees with S for the first i accesses, the state of the cache at access $i+1$ will be identical



Cache after i accesses with policy S_i

=



Cache after i accesses with policy S

Consider access $m_{i+1} = d$

Case 1: d is in the cache. Then, neither S_i nor S need to evict an element so we can use the same cache for S_{i+1}

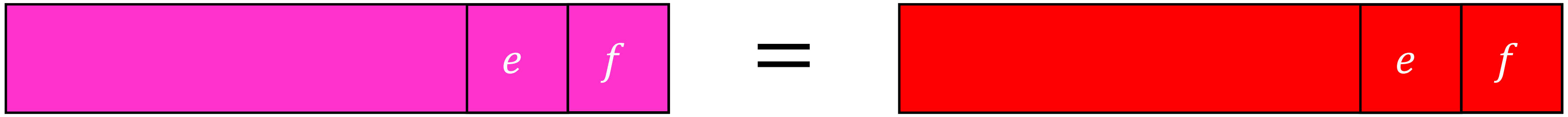
Cache after i accesses with policy S_{i+1}

Remaining evictions will follow S_i:
$\text{misses}(S_{i+1}) = \text{misses}(S_i)$

Exchange Argument for Belady Caching Algorithm

Lemma: If S_i and S agree on the first i accesses, then there is a schedule S_{i+1} that agrees with S on the first $i+1$ accesses such that $\text{misses}(S_{i+1}) \leq \text{misses}(S_i)$

Since S_i agrees with S for the first i accesses, the state of the cache at access $i+1$ will be identical



Cache after i accesses with policy S_i

Cache after i accesses with policy S

Consider access $m_{i+1} = d$

Case 2: d is not in the cache and both S_i and S evict the same element (e.g., f) from the cache. In this case, we can use the same cache for S_{i+1}

Cache after i accesses with policy S_{i+1}

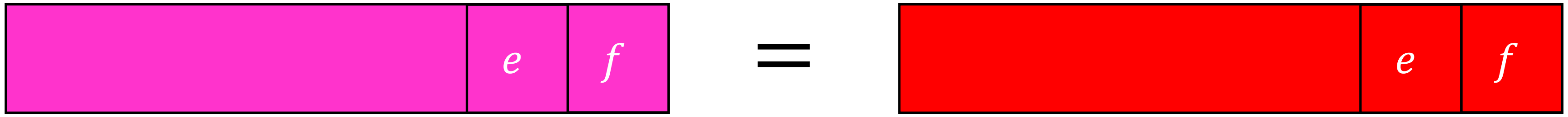
Remaining evictions will follow S_i:

$$\text{misses}(S_{i+1}) = \text{misses}(S_i)$$

Exchange Argument for Belady Caching Algorithm

Lemma: If S_i and S agree on the first i accesses, then there is a schedule S_{i+1} that agrees with S on the first $i+1$ accesses such that $\text{misses}(S_{i+1}) \leq \text{misses}(S_i)$

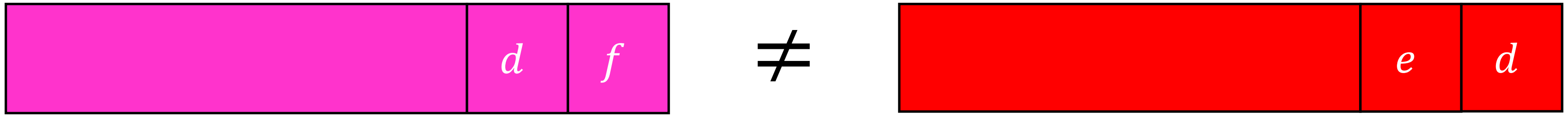
Since S_i agrees with S for the first i accesses, the state of the cache at access $i+1$ will be identical



Cache after i accesses with policy S_i | Cache after i accesses with policy S

Consider access $m_{i+1} = d$

Case 3: d is not in the cache and S_i and S evict different elements (e.g., S_i evicts e and S evicts f)

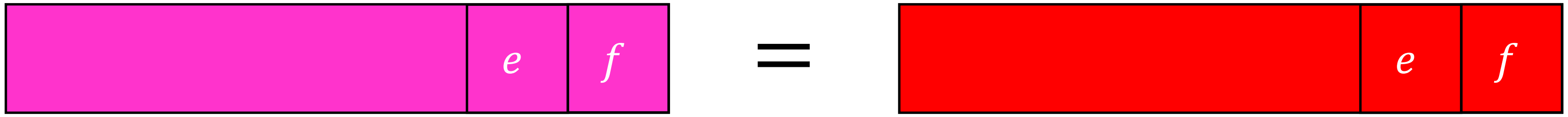


Cache after $i+1$ accesses with policy S_i | Cache after $i+1$ accesses with policy S

Exchange Argument for Belady Caching Algorithm

Lemma: If S_i and S agree on the first i accesses, then there is a schedule S_{i+1} that agrees with S on the first $i+1$ accesses such that $\text{misses}(S_{i+1}) \leq \text{misses}(S_i)$

Since S_i agrees with S for the first i accesses, the state of the cache at access $i+1$ will be identical

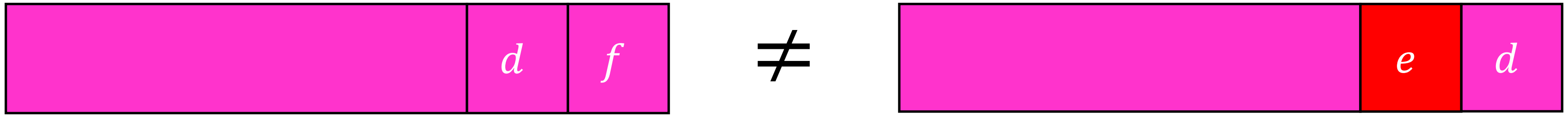


Cache after i accesses with policy S_i Cache after i accesses with policy S

Consider access $m_{i+1} = d$

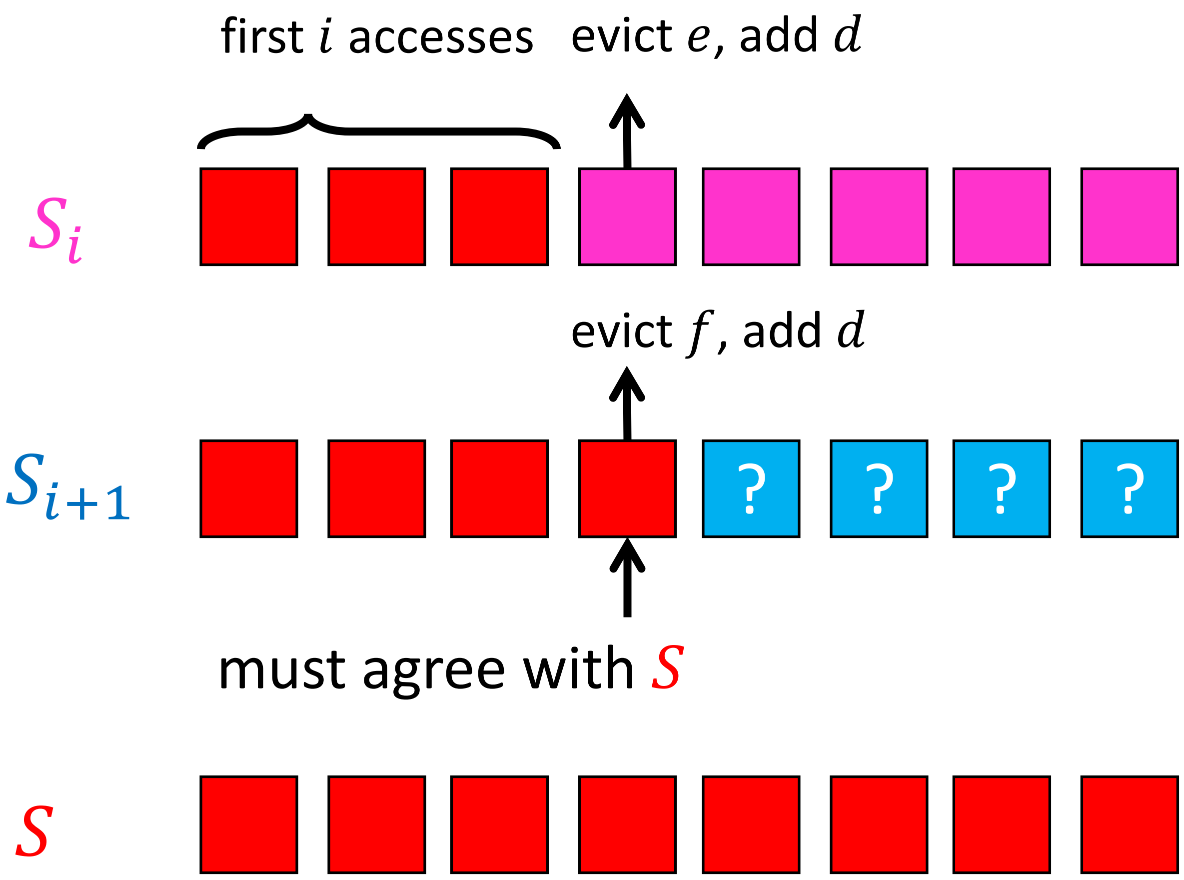
Case 3: d is not in the cache and S_i and S evict
and S evicts f)

Key observation: caches only differ in a single element



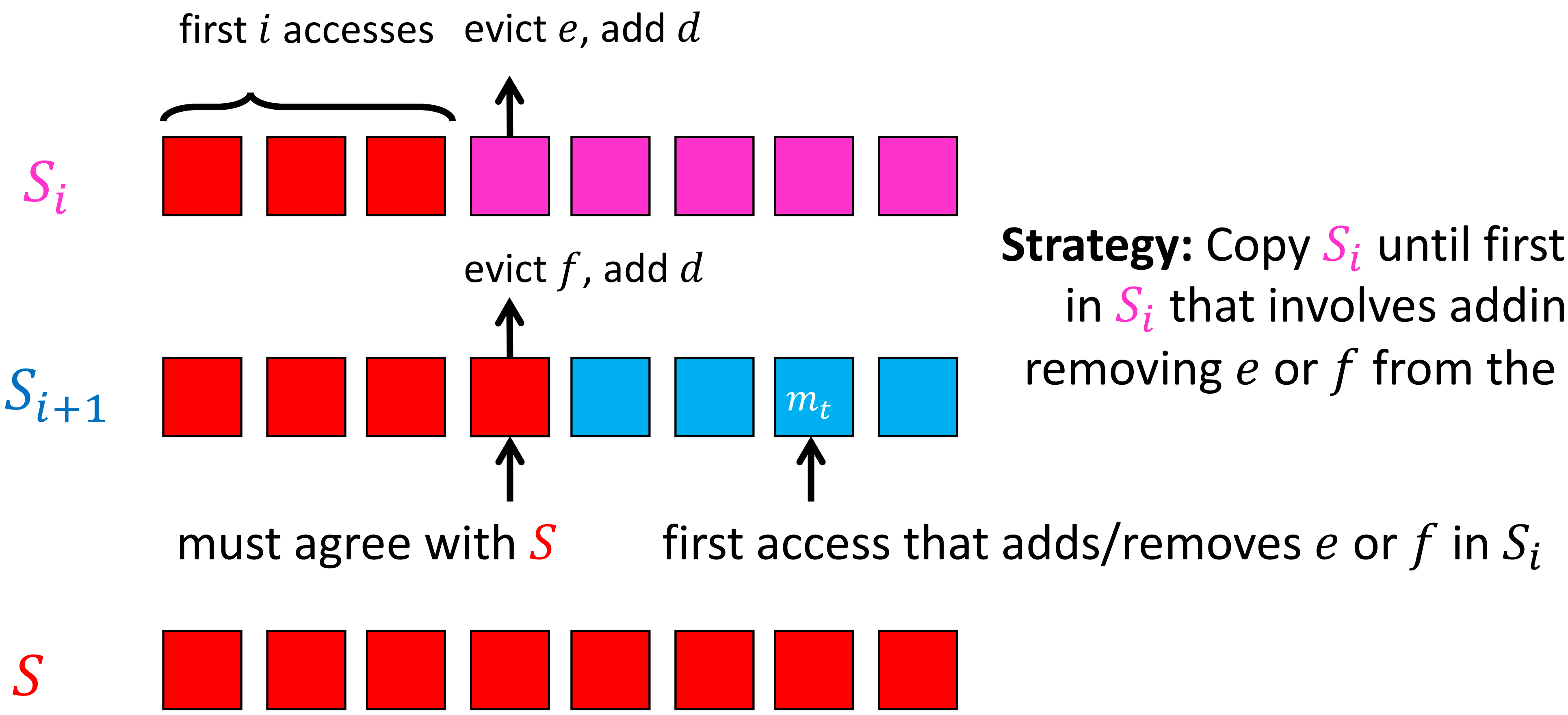
Cache after $i+1$ accesses with policy S_i Cache after $i+1$ accesses with policy S

Exchange Argument for Belady Caching Algorithm



Objective: Need to fill in the rest of S_{i+1} to have <u>no more</u> cache misses than S_i

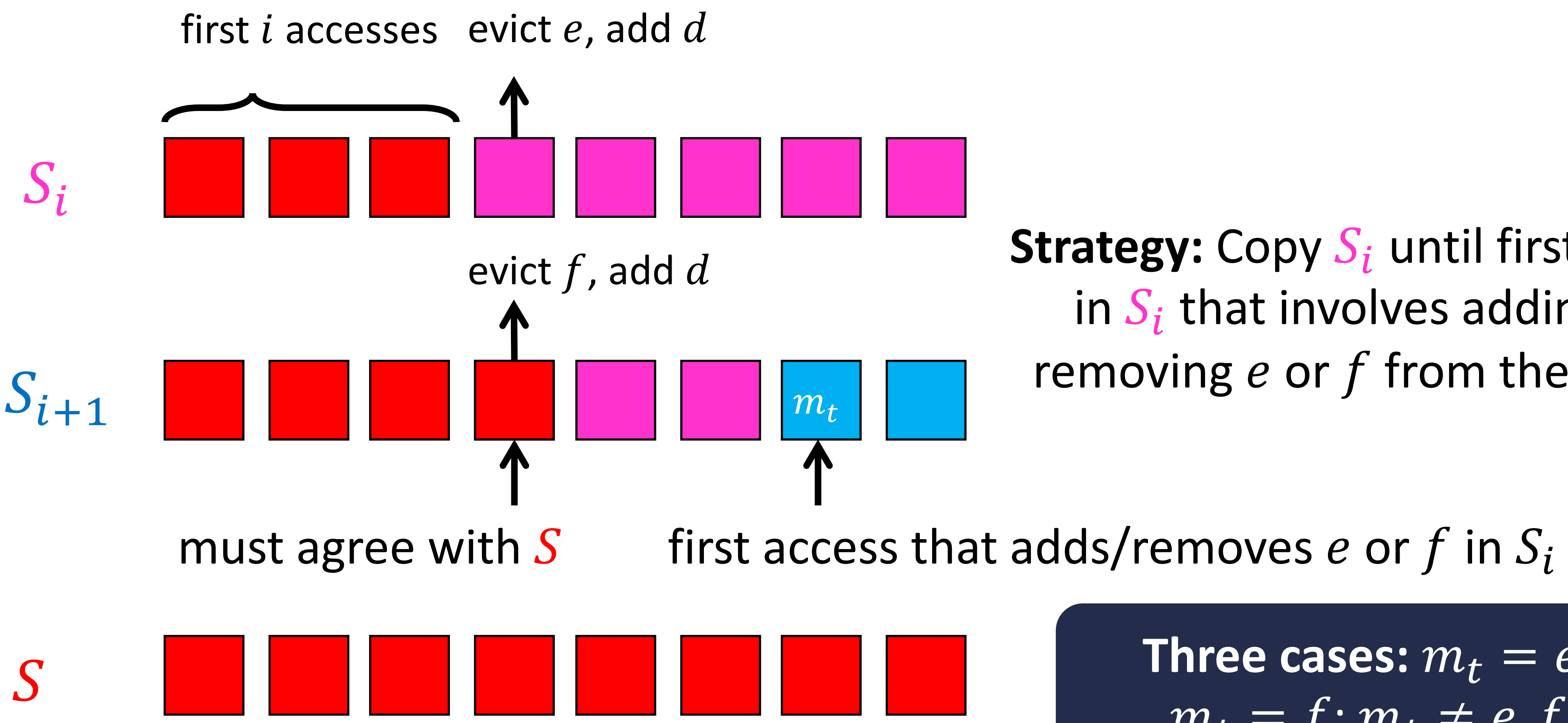
Exchange Argument for Belady Caching Algorithm



Strategy: Copy S_i until first access in S_i that involves adding or removing e or f from the cache

Objective: Need to fill in the rest of S_{i+1} to have <u>no more</u> cache misses than S_i

Exchange Argument for Belady Caching Algorithm



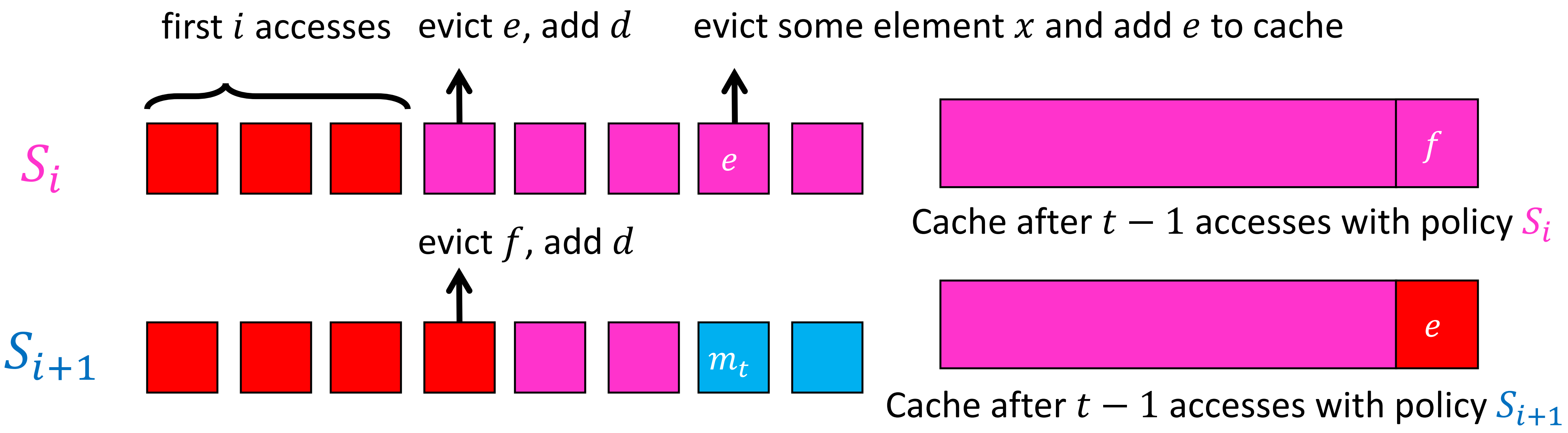
Strategy: Copy S_i until first access in S_i that involves adding or removing e or f from the cache

Three cases: $m_t = e$; $m_t = f$; $m_t \neq e, f$

Objective: Need to fill in the rest of S_{i+1} to have <u>no more</u> cache misses than S_i

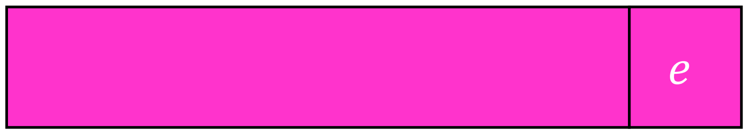
Exchange Argument for Belady Caching Algorithm

Case 1:
$m_t = e$



Two possibilities:

- $x = f$. Then, cache after t accesses with policy S_i is identical to that with policy S_{i+1}:



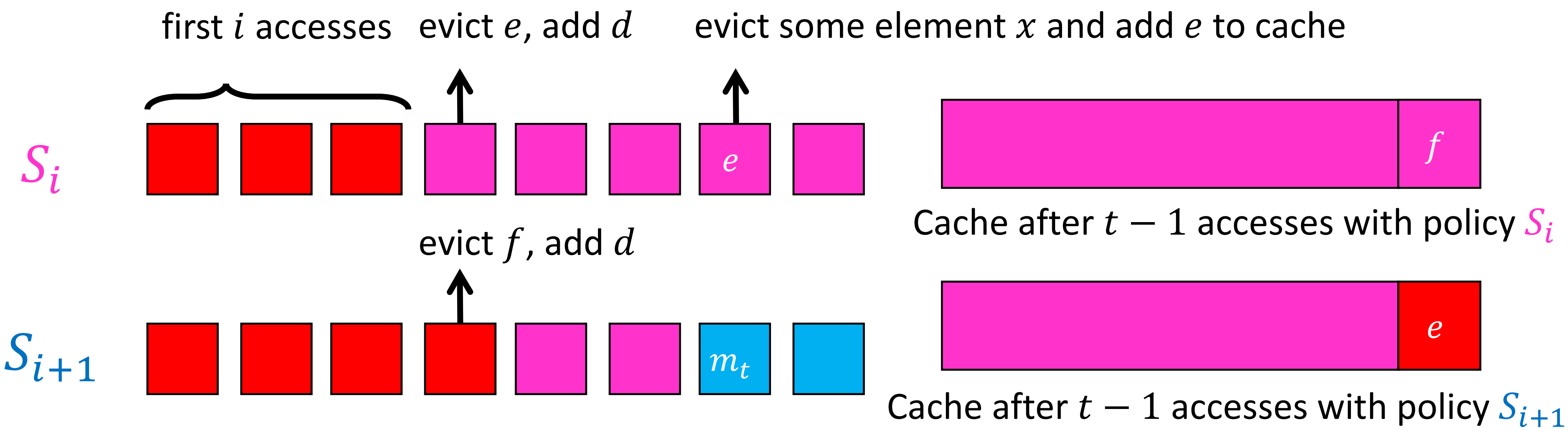
Cache after t accesses with policy S_i

Remaining evictions will follow S_i:

$$\text{misses}(S_{i+1}) < \text{misses}(S_i)$$

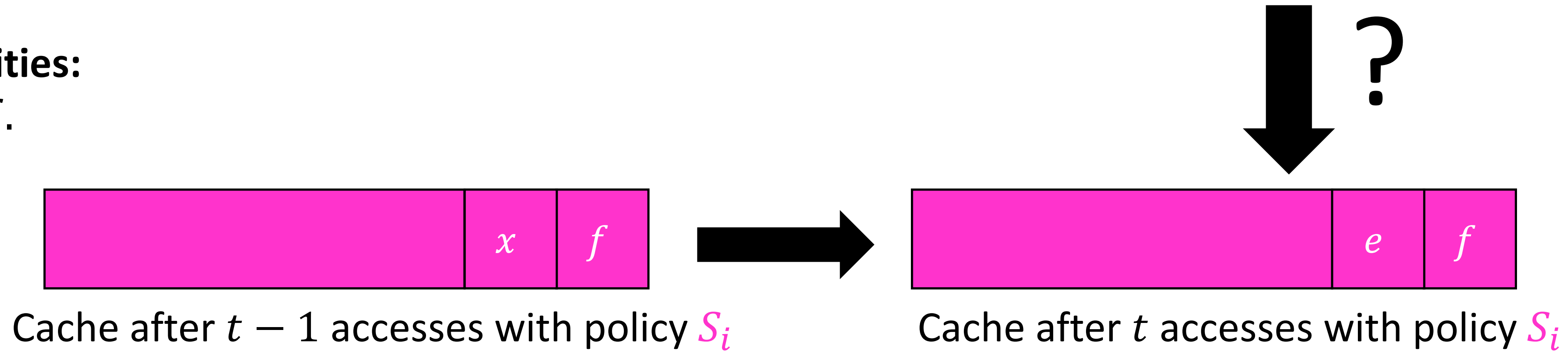
Exchange Argument for Belady Caching Algorithm

Case 1:
$m_t = e$



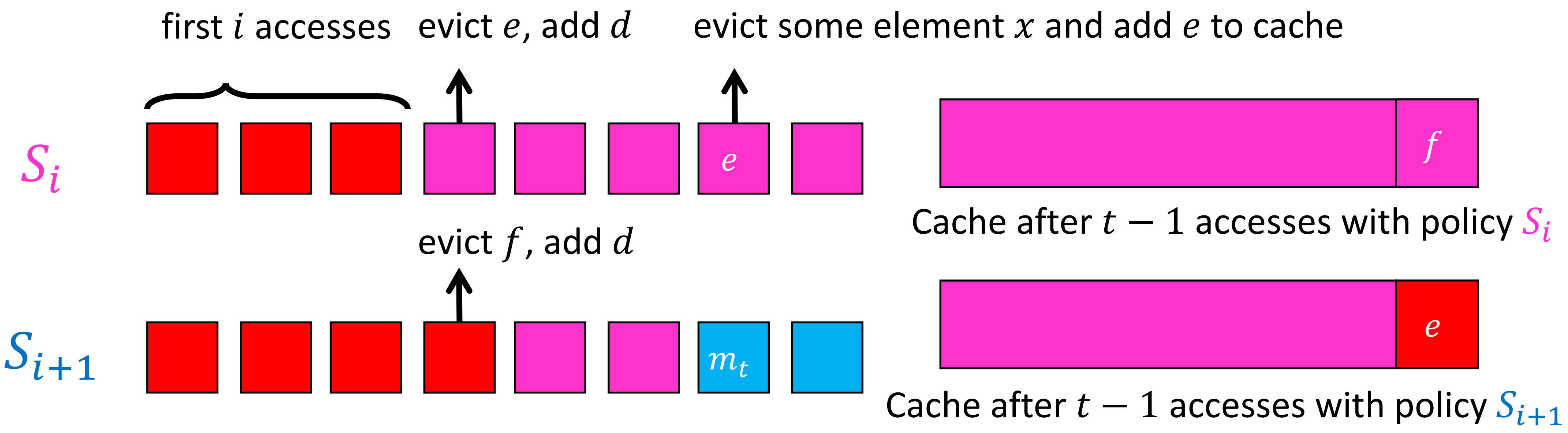
Two possibilities:

- $x \neq f$.



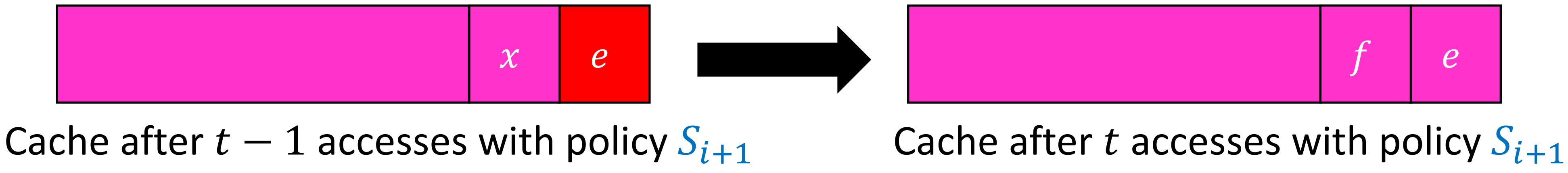
Exchange Argument for Belady Caching Algorithm

Case 1:
$m_t = e$



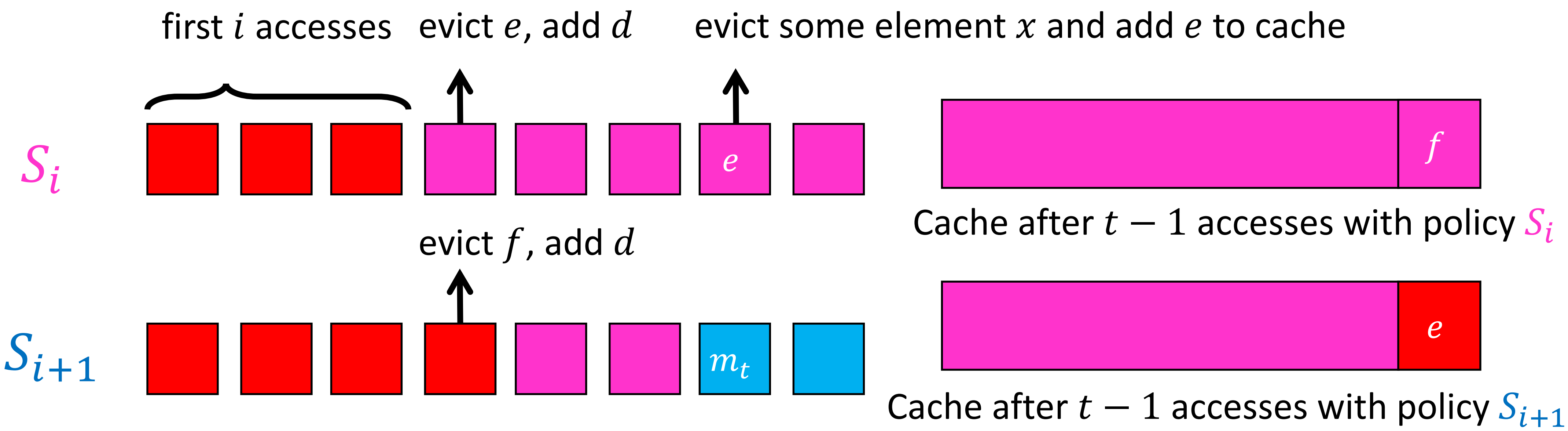
Two possibilities:

- $x \neq f$. S_{i+1} will also evict x from the cache and load f, so caches now match.



Exchange Argument for Belady Caching Algorithm

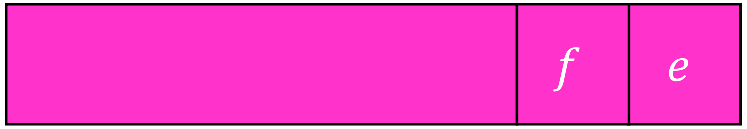
Case 1:
$m_t = e$



Two possibilities:

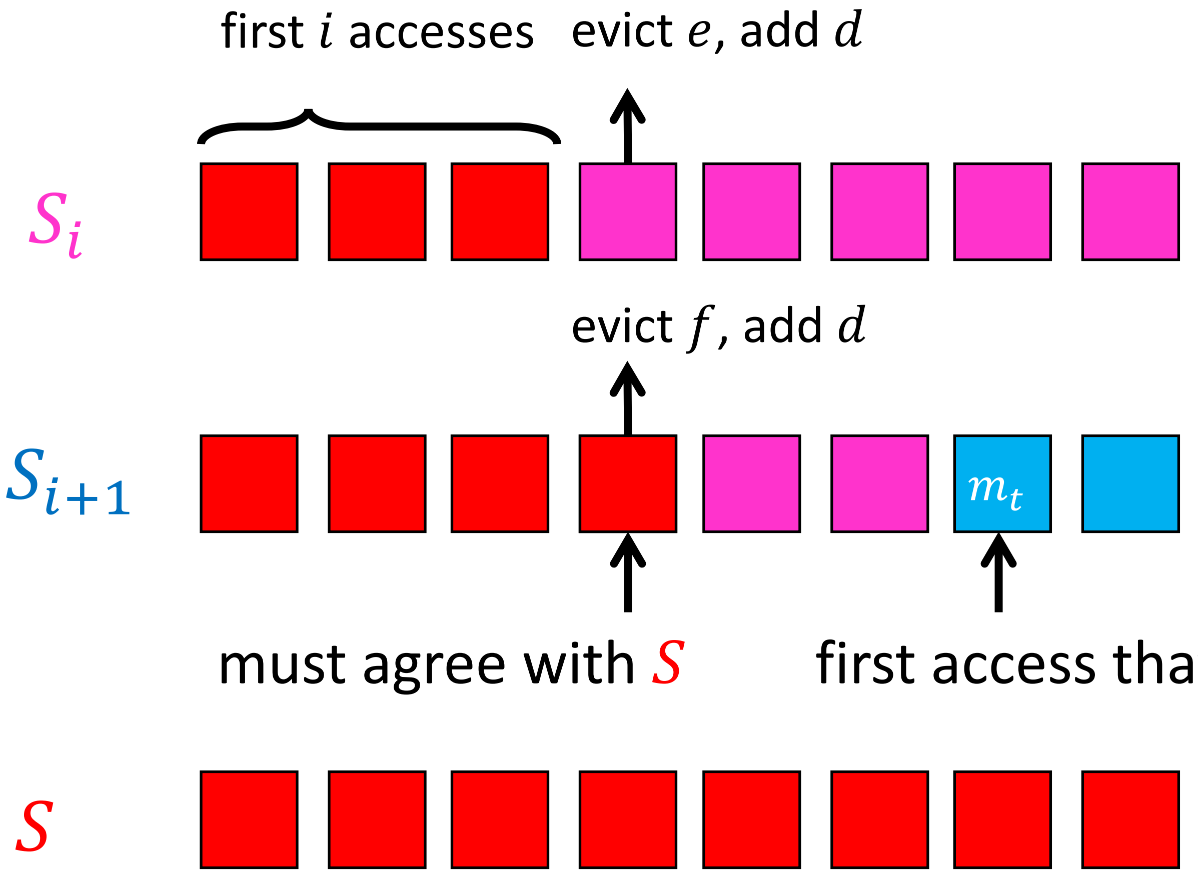
- $x \neq f$. S_{i+1} will also evict x from the cache and load f, so caches now match.

Remaining evictions will follow S_i:
$\text{misses}(S_{i+1}) = \text{misses}(S_i)$



Cache after t accesses with policy S_{i+1}

Exchange Argument for Belady Caching Algorithm



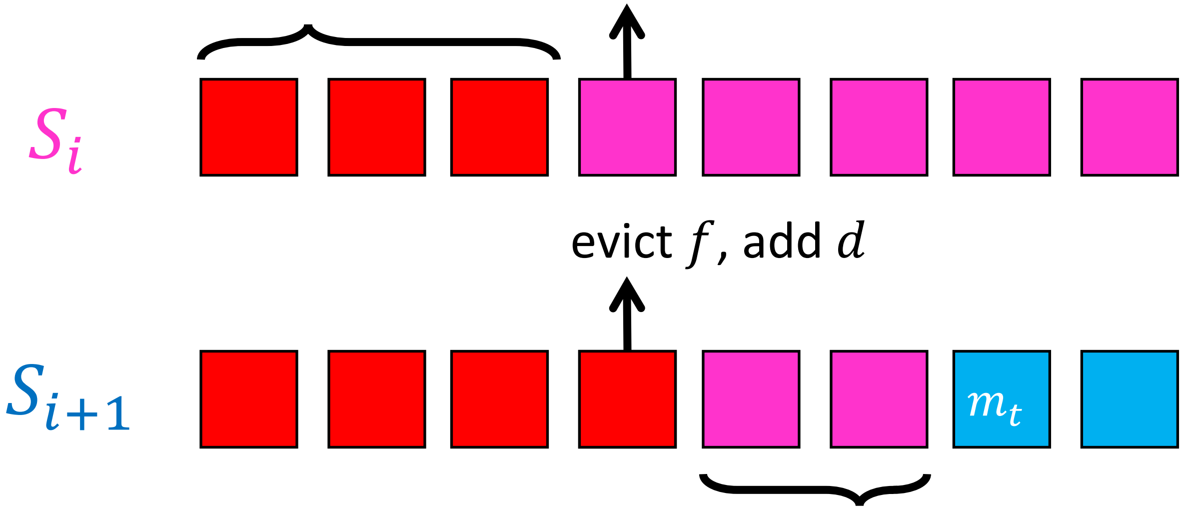
Strategy: Copy S_i until first access in S_i that involves adding or removing e or f from the cache

Three cases: $m_t = e$; $m_t = f$; $m_t \neq e, f$

Objective: Need to fill in the rest of S_{i+1} to have <u>no more</u> cache misses than S_i

Exchange Argument for Belady Caching Algorithm

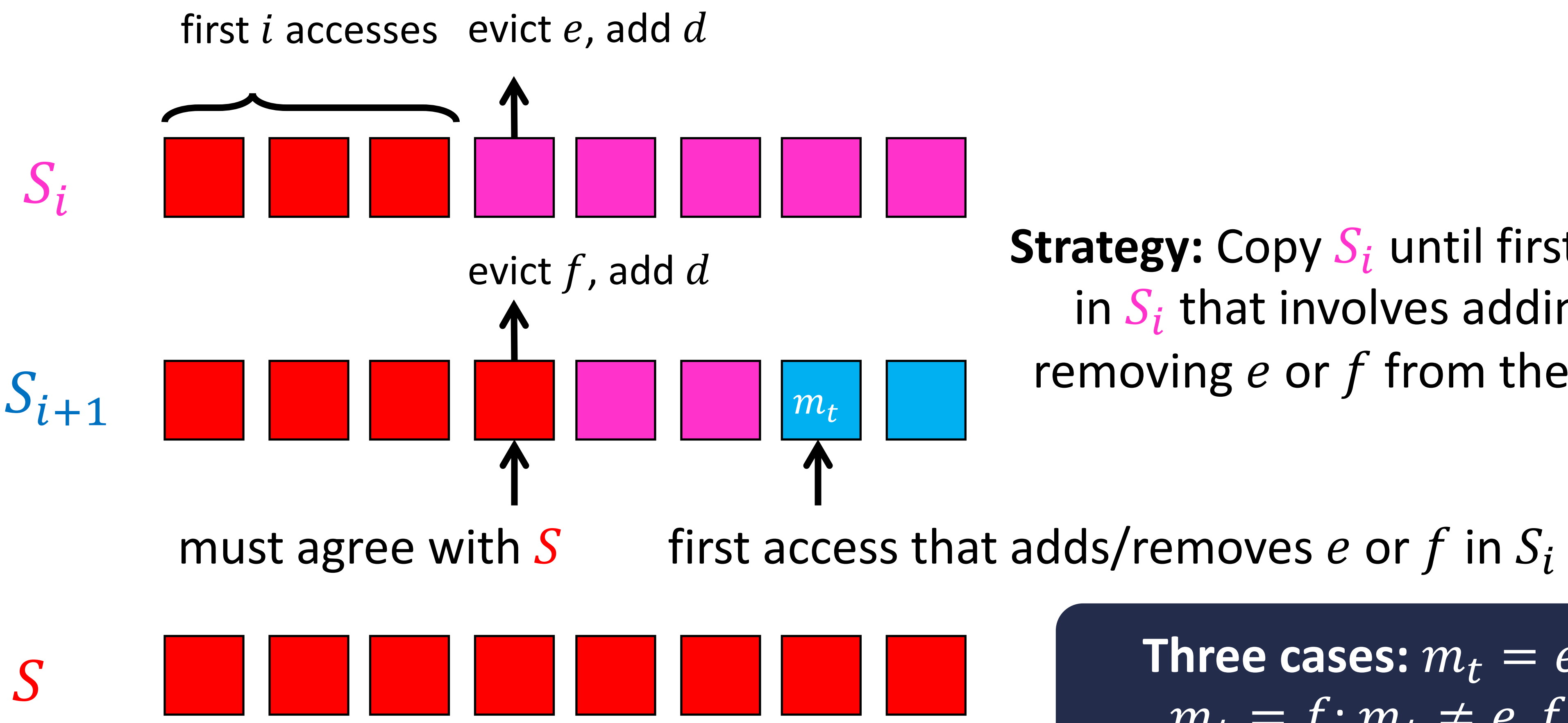
Case 2:
$m_t = f$



Contradiction: greedy choice is to evict element that is furthest in future, but element f is used before element e

Conclusion: this case cannot happen

Exchange Argument for Belady Caching Algorithm



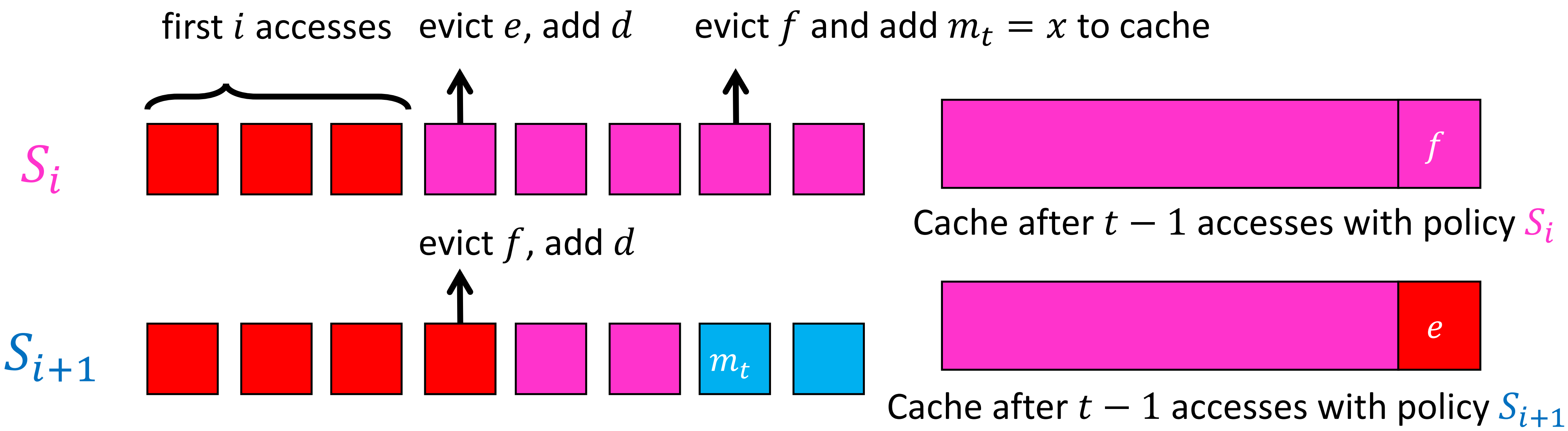
Strategy: Copy S_i until first access in S_i that involves adding or removing e or f from the cache

Three cases: $m_t = e$; $m_t = f$; $m_t \neq e, f$

Objective: Need to fill in the rest of S_{i+1} to have no more cache misses than S_i

Exchange Argument for Belady Caching Algorithm

Case 3:
$m_t \neq e, f$



Observation: not loading e, f so must be evicting either e or f

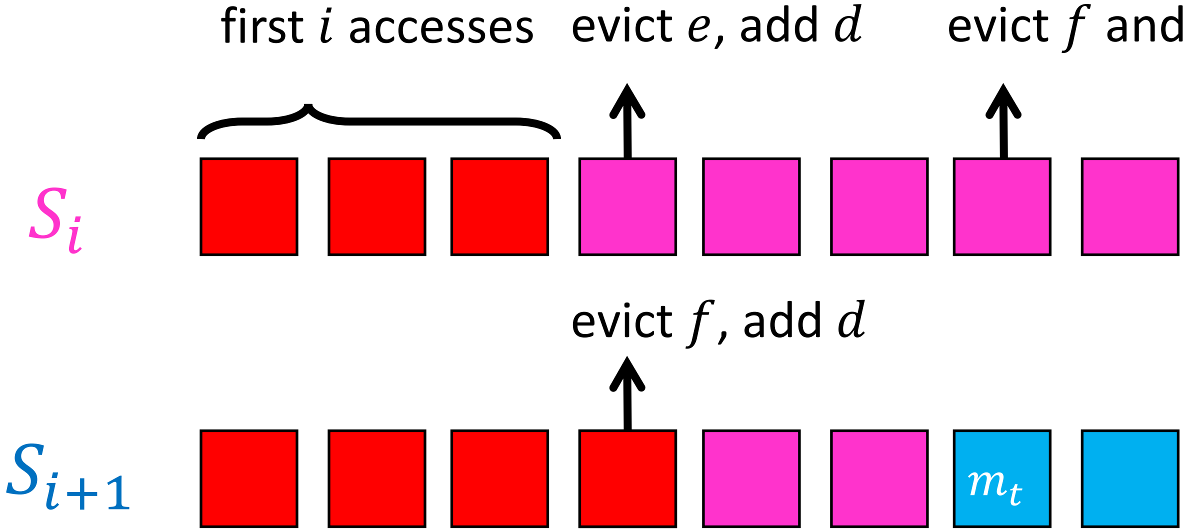
In S_i, e was already evicted and has not been loaded (by definition of m_t)

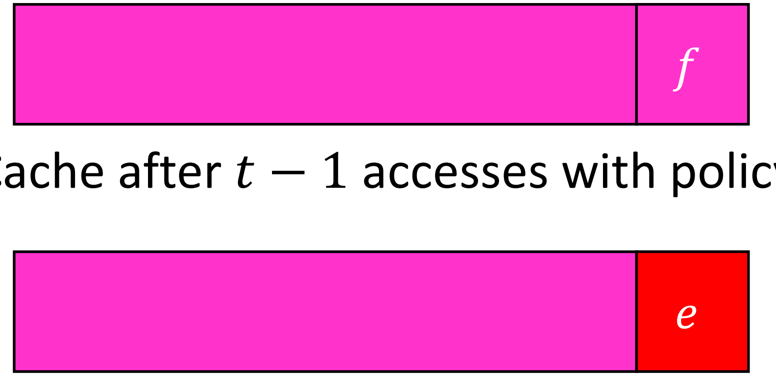
Only option is for S_i to evict f

Cache after $t-1$ accesses with policy S_i Cache after t accesses with policy S_i

Exchange Argument for Belady Caching Algorithm

Case 3:
$m_t \neq e, f$





Cache after $t-1$ accesses with policy S_i

Evict e and add x in S_{i+1} will yield <u>same</u> cache state!

Observation: not loading e, f so must be evicting either e or f

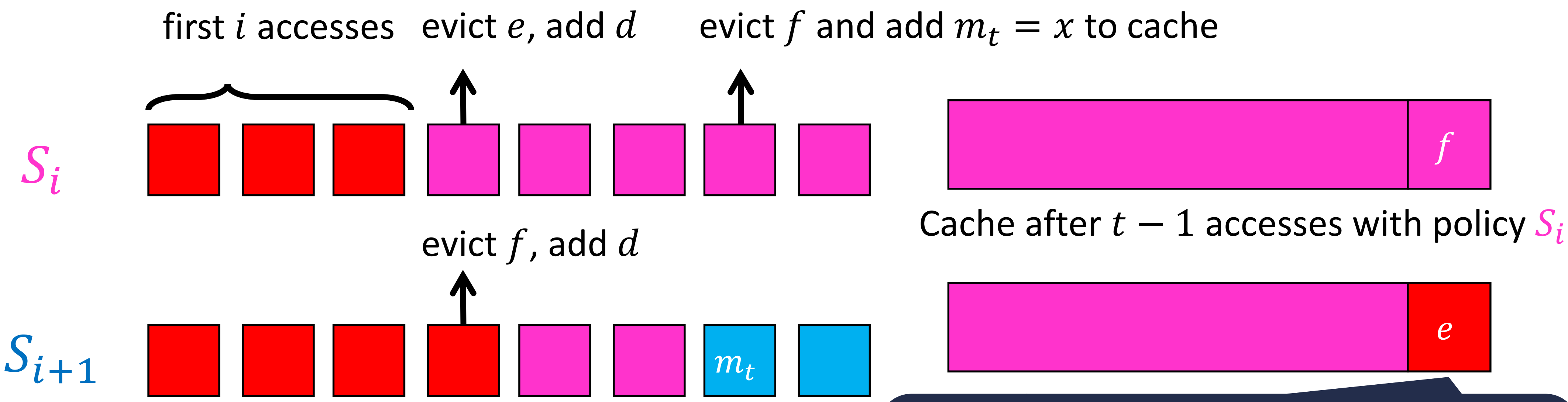
In S_i, e was already evicted and has not been loaded (by defi

Only option is for S_i to evict f

Cache after $t-1$ accesses with policy S_i → Cache after t accesses with policy S_i

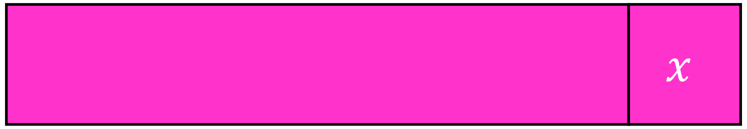
Exchange Argument for Belady Caching Algorithm

Case 3:
$m_t \neq e, f$



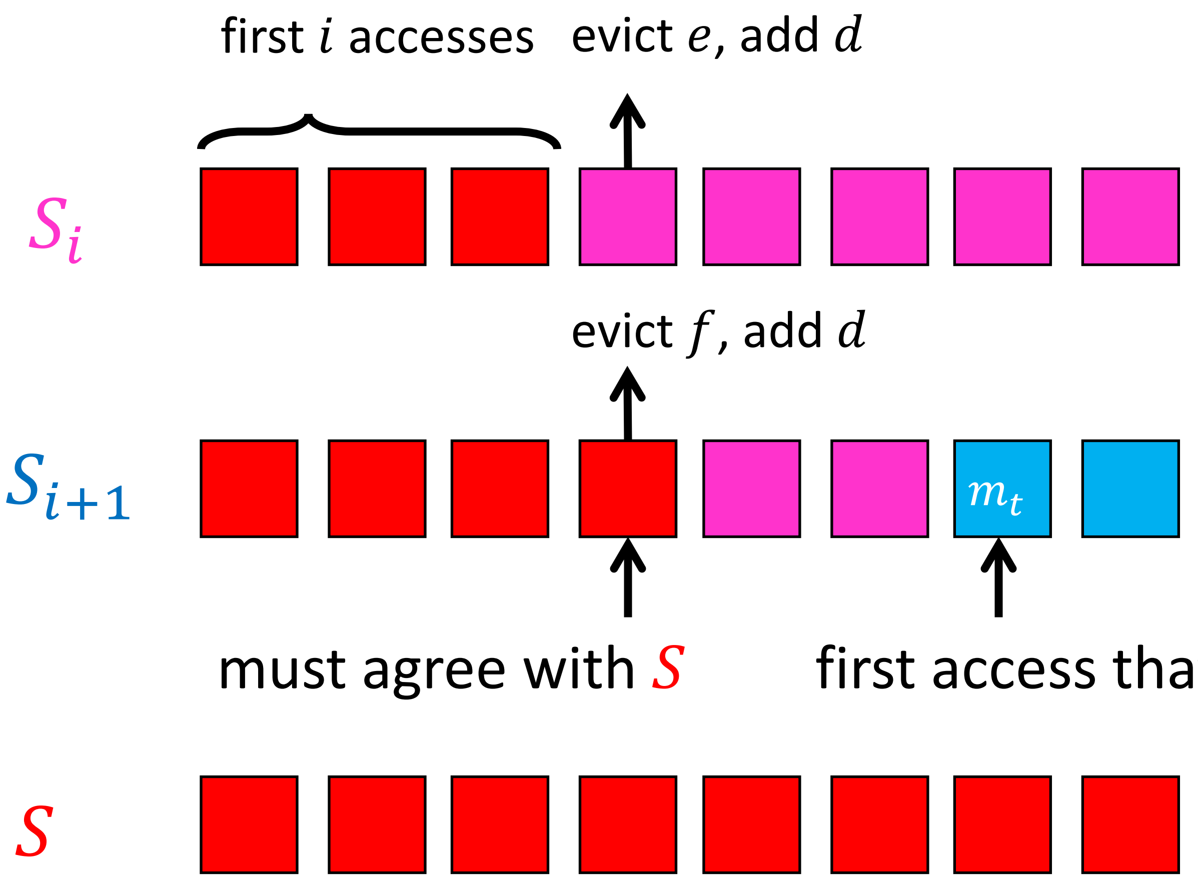
Evict e and add x in S_{i+1} will yield <u>same</u> cache state!

Remaining evictions will follow S_i:
$\text{misses}(S_{i+1}) = \text{misses}(S_i)$



Cache after t accesses with policy S_{i+1}

Exchange Argument for Belady Caching Algorithm



Strategy: Copy S_i until first access in S_i that involves adding or removing e or f from the cache

Three cases: $m_t = e$; $m_t = f$; $m_t \neq e, f$

Conclusion: In all three cases, we can construct a strategy where

$$\text{misses}(S_{i+1}) \leq \text{misses}(S_i)$$

Exchange Argument for Belady Caching Algorithm

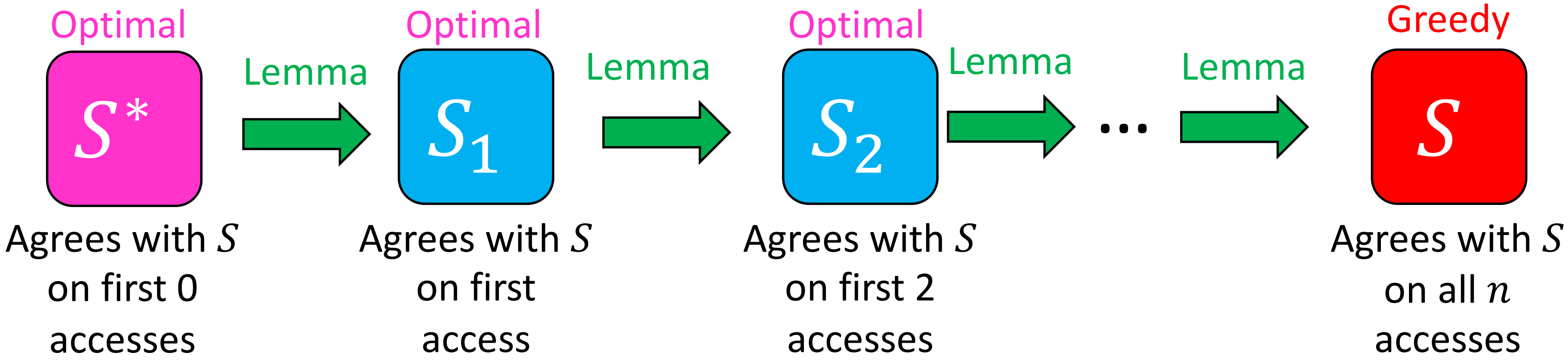
Let S be the schedule chosen by the greedy algorithm

Let S^* be any optimal schedule (that minimizes the number of cache misses)

Lemma: If S_i and S agree on the first i accesses, then there is a schedule S_{i+1} that agrees with S on the first $i+1$ accesses such that

$$\text{misses}(S_{i+1}) \leq \text{misses}(S_i)$$

Correctness then follows by induction:

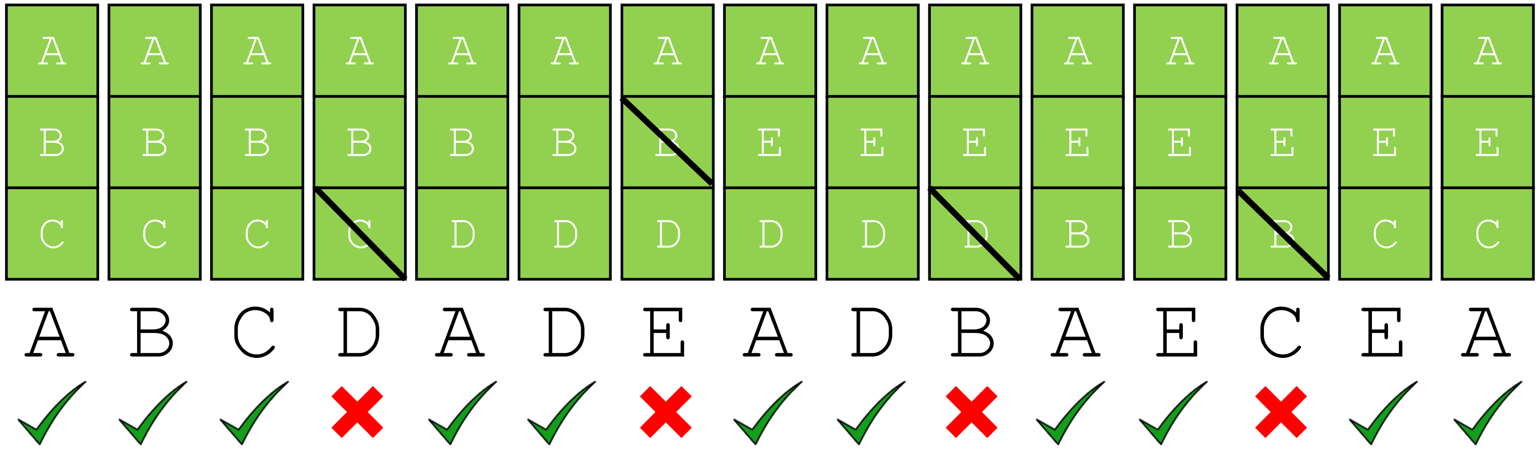


$$\text{misses}(S) = \text{misses}(S_n) \leq \text{misses}(S_{n-1}) \leq \cdots \leq \text{misses}(S_0) = \text{misses}(S^*)$$

Belady Caching

Belady eviction policy: Evict the item accessed farthest in the future

Cache contents



Sequence of cache accesses

In online settings, we do not know exact sequence of memory accesses, so cannot compute farthest future access

Heuristic: past access pattern is a good predictor for future

Strategy: evict the least-recently used item (LRU caching)