CS4102 Algorithms

Spring 2020

Today's Keywords

- Greedy Algorithms
- Choice Function
- Cache Replacement
- Hardware & Algorithms

CLRS Reading: Chapter 16

Caching Problem

time complexity
space complexity

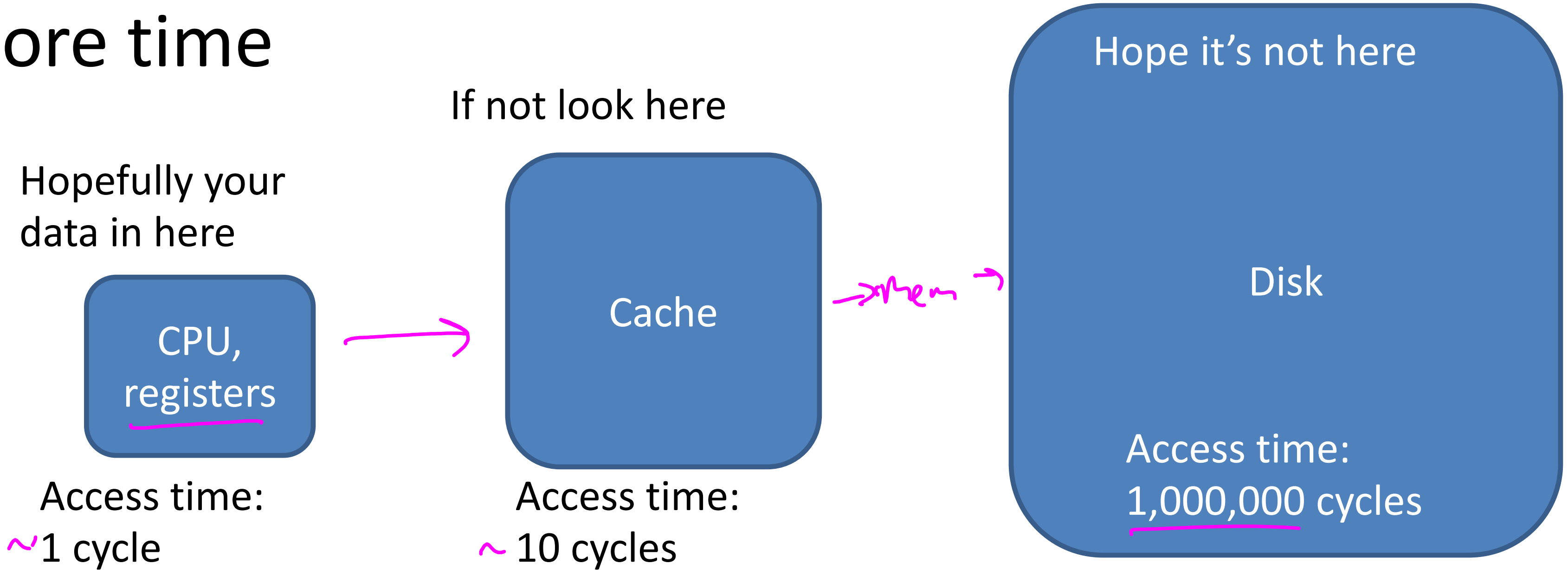
- Why is using too much memory a bad thing?
  - memory can be expensive
  - using too much memory forces us to use slower memory
  - time ≥ memory

Von Neumann Bottleneck

- Named for John von Neumann
- Inventor of modern computer architecture
- Other notable influences include:
  - Mathematics
  - Physics
  - Economics
  - Computer Science

Von Neumann Bottleneck

- Reading from memory is VERY slow
- Big memory = slow memory
- Solution: hierarchical memory
- Takeaway for Algorithms: Memory is time, more memory is a lot more time



Caching Problem

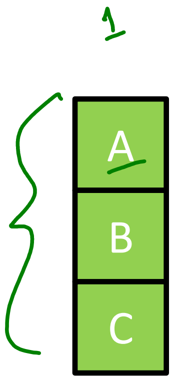
- Cache misses are very expensive
- When we load something new into cache, we must eliminate something already there
- We want the best cache "schedule" to minimize the number of misses

Caching Problem Definition

- Input:
  - k = size of the cache
  - $M = [m_1, m_2, \ldots m_n]$ = memory access pattern
- Output:
  - "schedule" for the cache (list of items in the cache at each time) which minimizes cache fetches

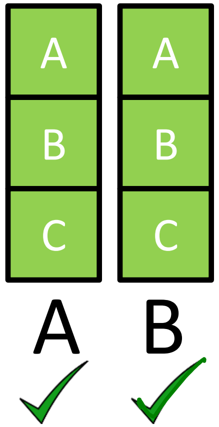
Example

k=3



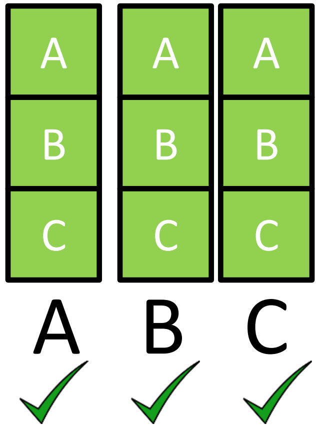
A B C D A D E A D B A E C E A

Example



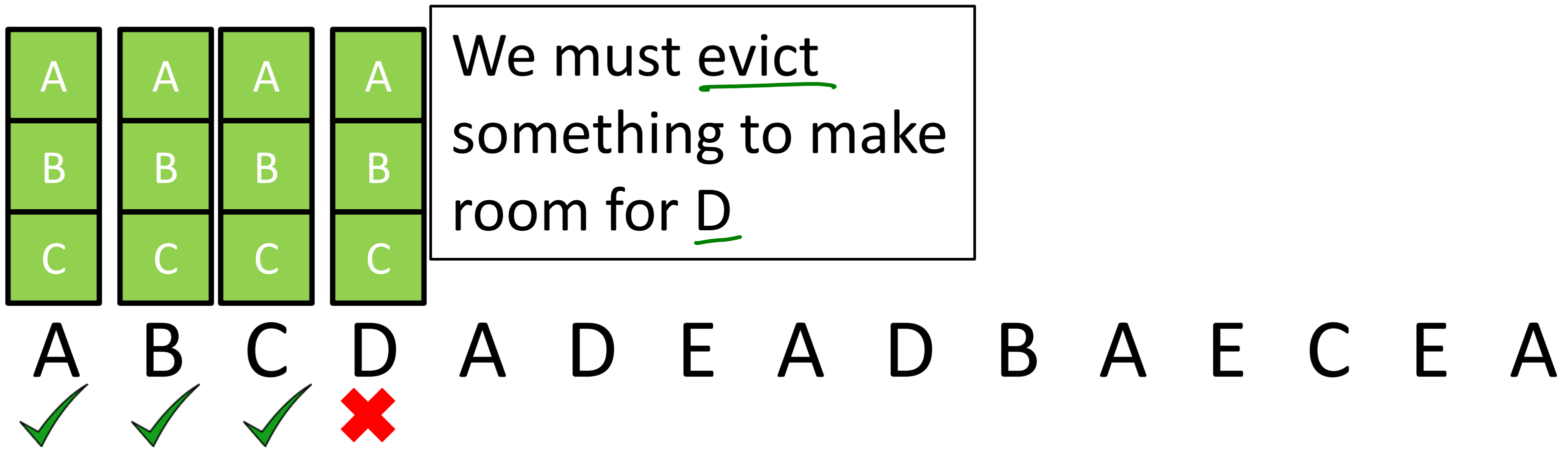
A B C D A D E A D B A E C E A

Example

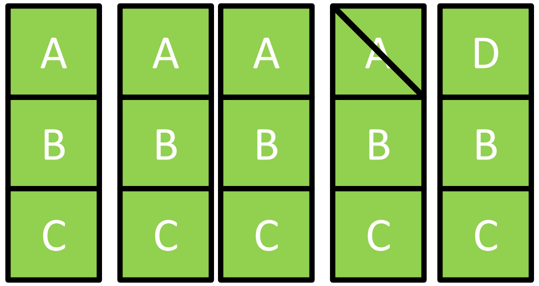


A B C D A D E A D B A E C E A

Example


A
B
C
We must evict something to make room for D
A B C D A D E A D B A E C E A

Example

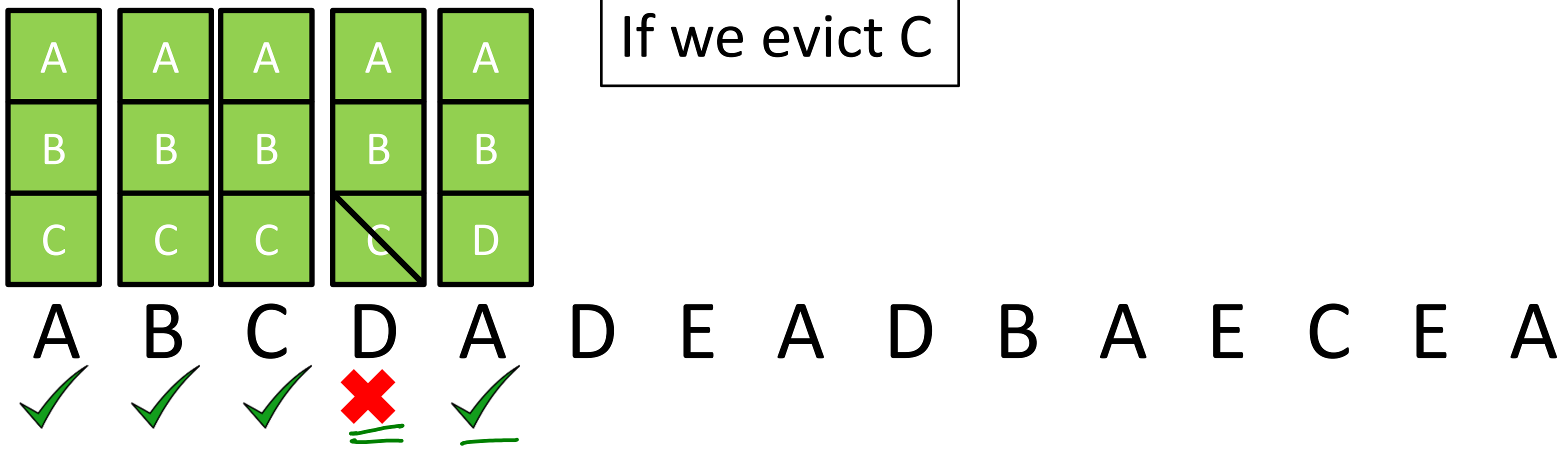


If we evict A

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

Example

If we evict C

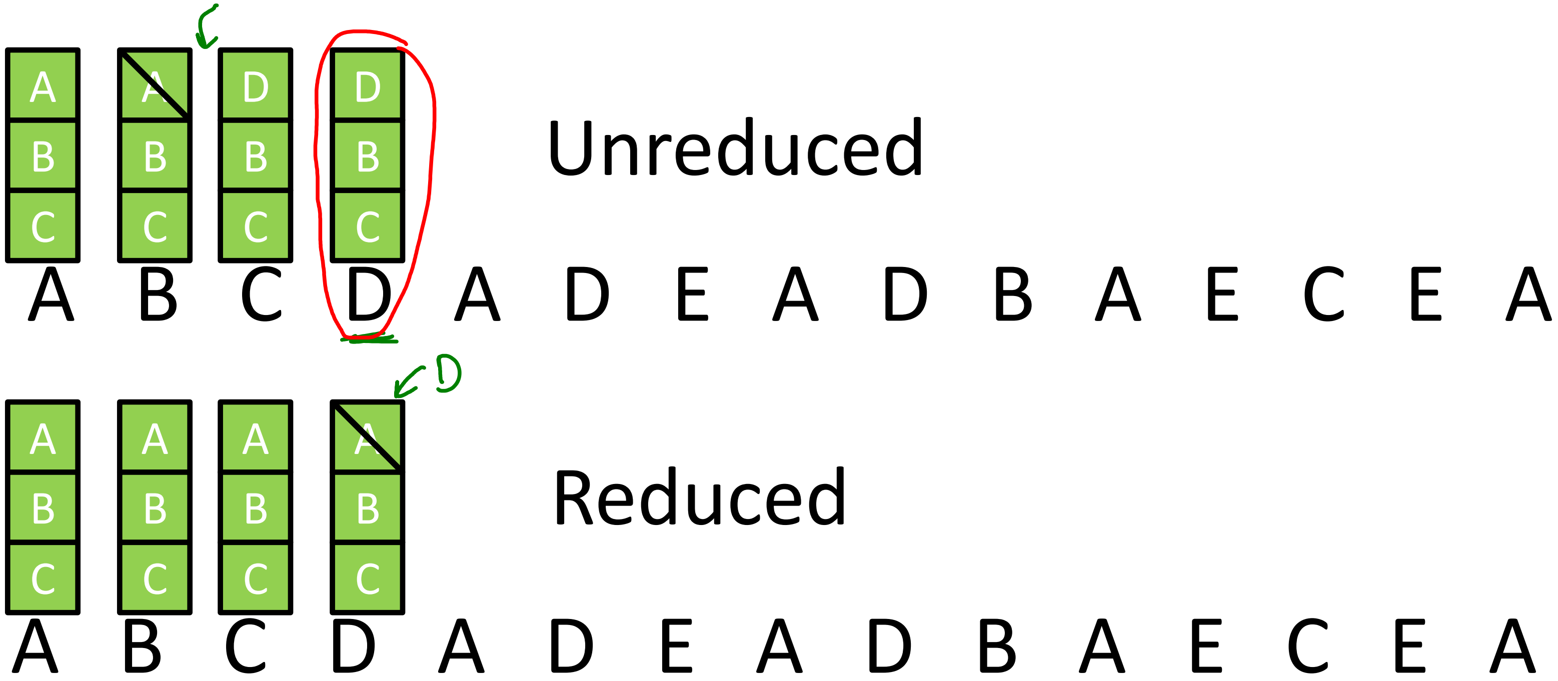


Our Problem vs Reality

- Assuming we know the entire access pattern
- Cache is Fully Associative – any memory address can go anywhere in the cache.
- Counting # of fetches (not necessarily misses)
- "Reduced" Schedule: Address only loaded on the cycle it's required
  - Reduced == Unreduced (by number of fetches)

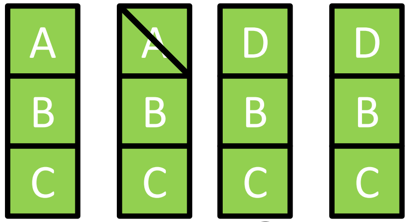
Our Problem vs Reality

- Assuming we know the entire access pattern
- Cache is Fully Associative
- Counting # of fetches (not necessarily misses)
- "Reduced" Schedule: Address only loaded on the cycle it's required
  - Reduced == Unreduced (by number of fetches)



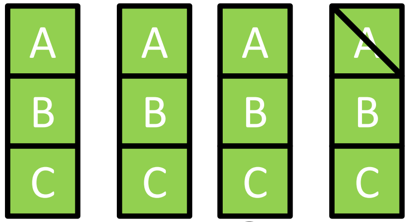
Our Problem vs Reality

- Assuming we know the entire access pattern
- Cache is Fully Associative
- Counting # of fetches (not necessarily misses)
- "Reduced" Schedule: Address only loaded on the cycle it's required
  - Reduced == Unreduced (by number of fetches)



Unreduced

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



Reduced

Leaving A in longer does not save fetches

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

Greedy Algorithms

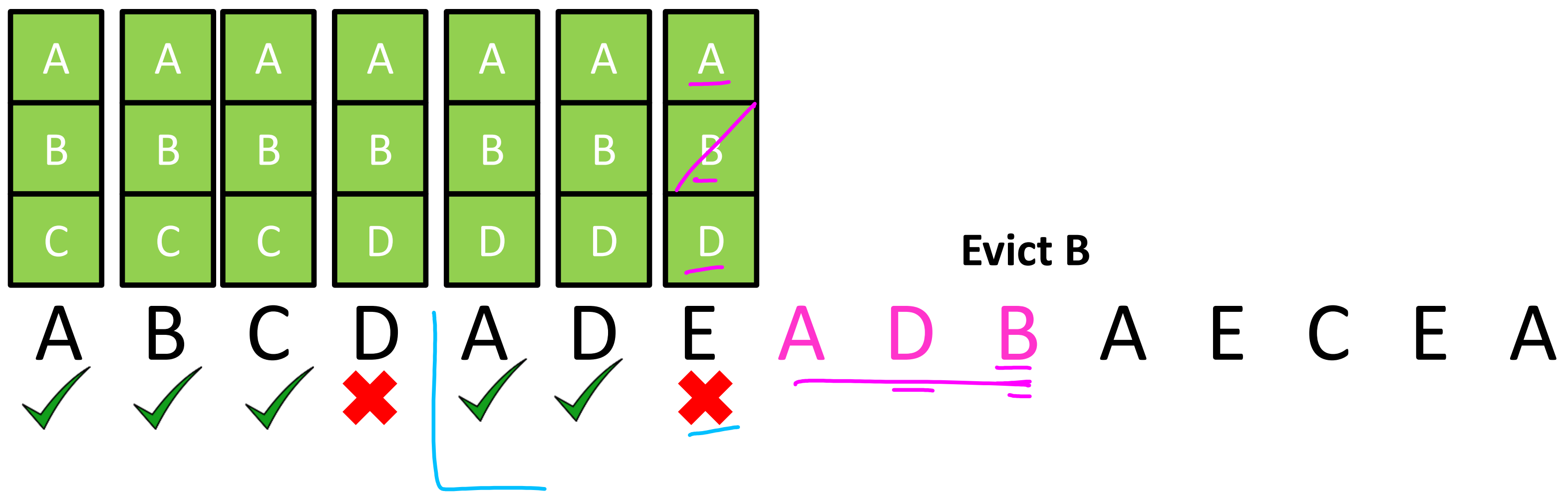
- Require Optimal Substructure
  - Solution to larger problem contains the solution to a smaller one
  - Only one subproblem to consider!
- Idea:
  1. Identify a greedy choice property
     - How to make a choice guaranteed to be included in some optimal solution
  2. Repeatedly apply the choice property until no subproblems remain

Greedy choice property

- Belady evict rule:
  - Evict the item accessed farthest in the future

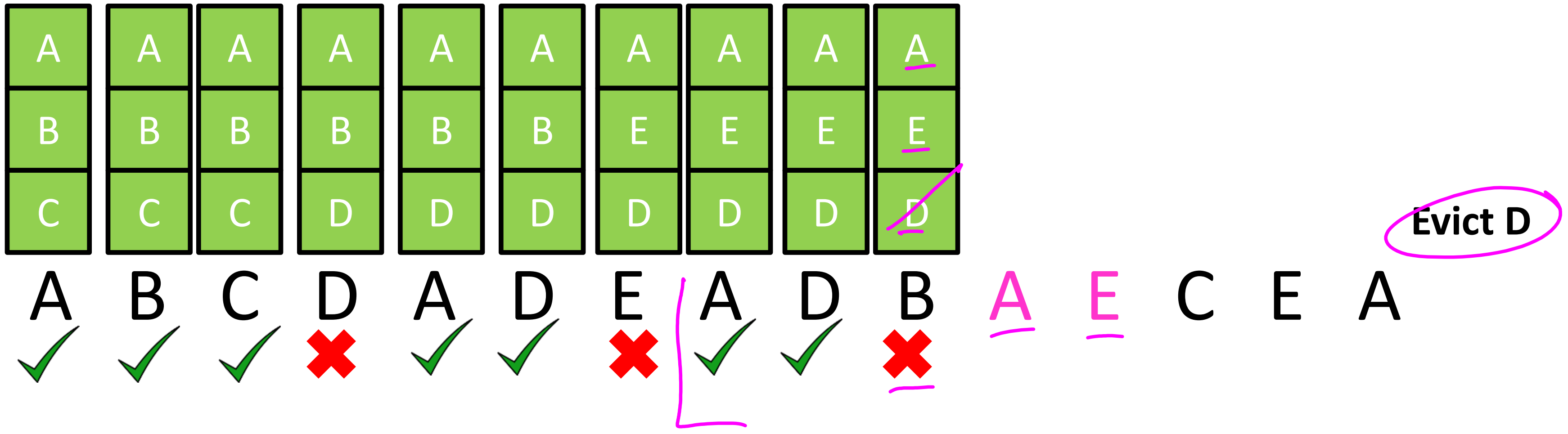
Greedy choice property

- Belady evict rule:
  - Evict the item accessed farthest in the future



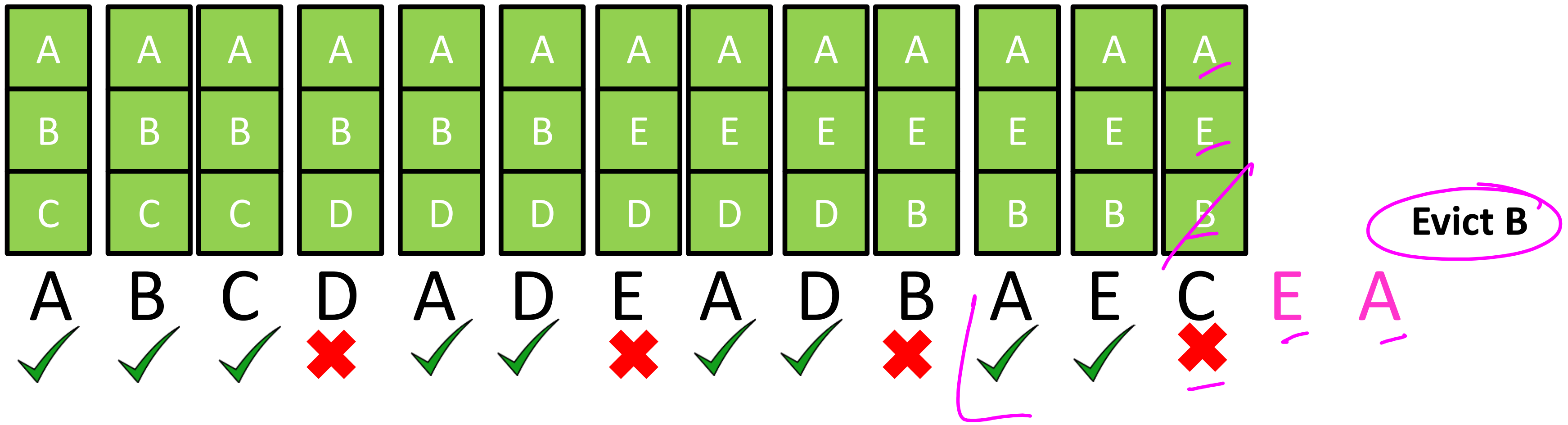
Greedy choice property

- Belady evict rule:
  - Evict the item accessed farthest in the future



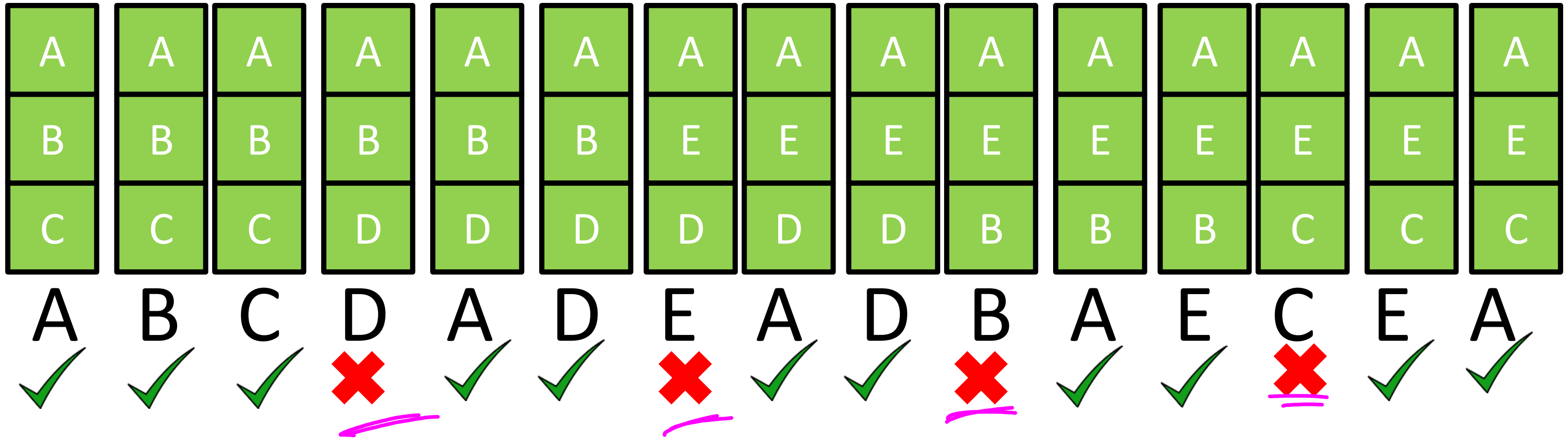
Greedy choice property

- Belady evict rule:
  - Evict the item accessed farthest in the future



Greedy choice property

- Belady evict rule:
  - Evict the item accessed farthest in the future



4 Cache Misses

Greedy Algorithms

- Require Optimal Substructure
  - Solution to larger problem contains the solution to a smaller one
  - Only one subproblem to consider!
- Idea:
  1. Identify a greedy choice property
     - How to make a choice guaranteed to be included in some optimal solution
  2. Repeatedly apply the choice property until no subproblems remain

Caching Greedy Algorithm

Initialize $cache$= first k accesses   O(k)

For each $m_i \in M$:   n times

    if $m_i \in cache$:   O(k)

        print $cache$ O(k)

    else:

        m = furthest-in-future from cache   O(kn)

        evict m, load m_i O(1)

        print $cache$ O(k)

O(kn²)

Exchange argument

- Shows correctness of a greedy algorithm
- Idea:
  - Show exchanging an item from an arbitrary optimal solution with your greedy choice makes the new solution no worse
  - How to show my sandwich is at least as good as yours:
    - Show: “I can remove any item from your sandwich, and it would be no worse by replacing it with the same item from my sandwich”

Belady Exchange Lemma

Let S_{ff} be the schedule chosen by our greedy algorithm - evict the mem location accessed farthest in the future.

Let S_i be a schedule which agrees with S_{ff} for the first i memory accesses.

We will show: there is a schedule S_{i+1} which agrees with S_{ff} for the first $i+1$ memory accesses, and has no more misses than S_i (i.e. $misses(S_{i+1}) \leq misses(S_i)$)

Belady Exchange Lemma

Let S_{ff} be the schedule chosen by our greedy algorithm

Let S_i be a schedule which agrees with S_{ff} for the first i memory accesses.

We will show: there is a schedule S_{i+1} which agrees with S_{ff} for the first $i + 1$ memory accesses, and has no more misses than S_i

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

S_0

Optimal

S^*

Agrees with S_{ff} on first 0 accesses

Belady Exchange Lemma

Let S_{ff} be the schedule chosen by our greedy algorithm

Let S_i be a schedule which agrees with S_{ff} for the first i memory accesses.

We will show: there is a schedule S_{i+1} which agrees with S_{ff} for the first $i+1$ memory accesses, and has no more misses than S_i

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

Optimal

S^*

Agrees with S_{ff} on first 0 accesses

Greedy

S_{ff}

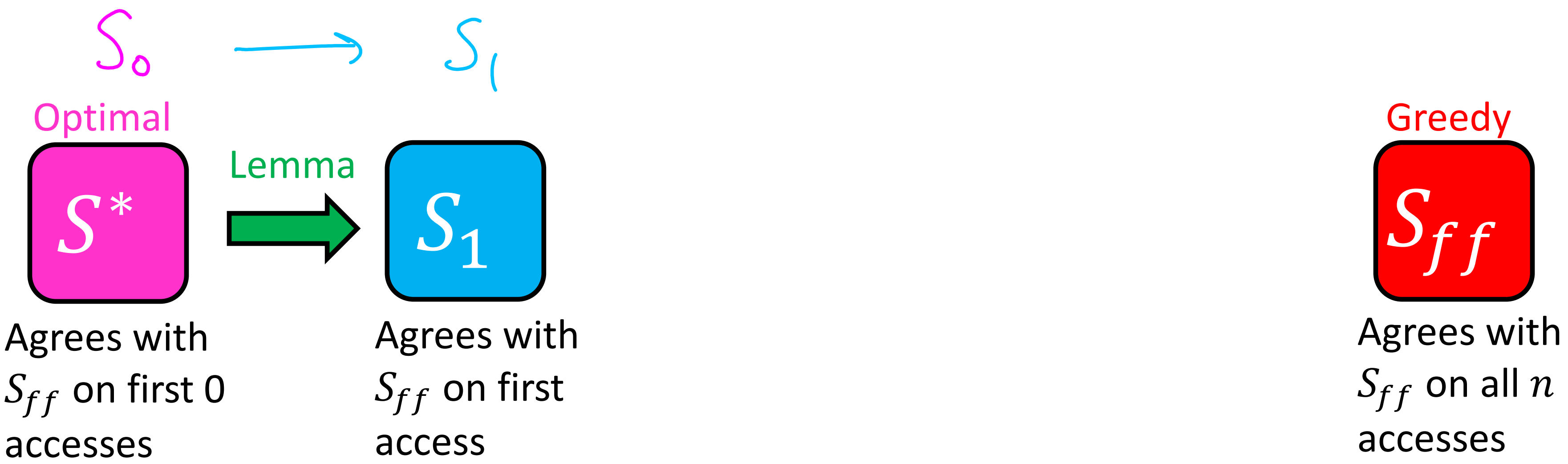
Agrees with S_{ff} on all n accesses

Belady Exchange Lemma

Let S_{ff} be the schedule chosen by our greedy algorithm

Let S_i be a schedule which agrees with S_{ff} for the first i memory accesses.

We will show: there is a schedule S_{i+1} which agrees with S_{ff} for the first $i+1$ memory accesses, and has no more misses than S_i (i.e. $misses(S_{i+1}) \leq misses(S_i)$)



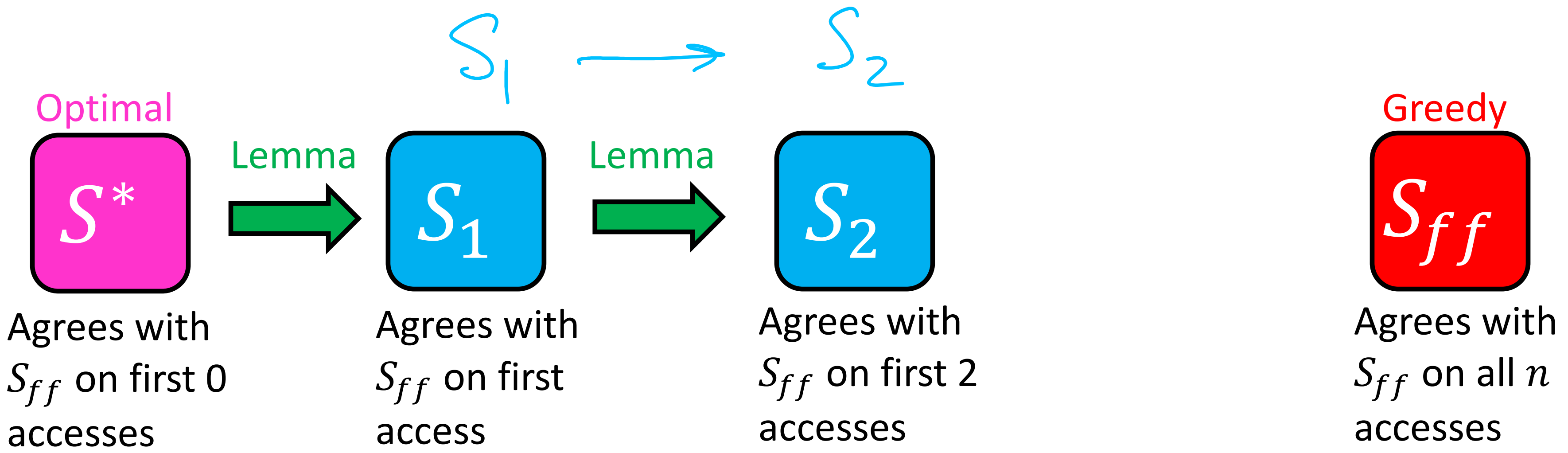
Belady Exchange Lemma

Let S_{ff} be the schedule chosen by our greedy algorithm

Let S_i be a schedule which agrees with S_{ff} for the first i memory accesses.

We will show: there is a schedule S_{i+1} which agrees with S_{ff} for the first $i+1$ memory accesses, and has no more misses than S_i

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

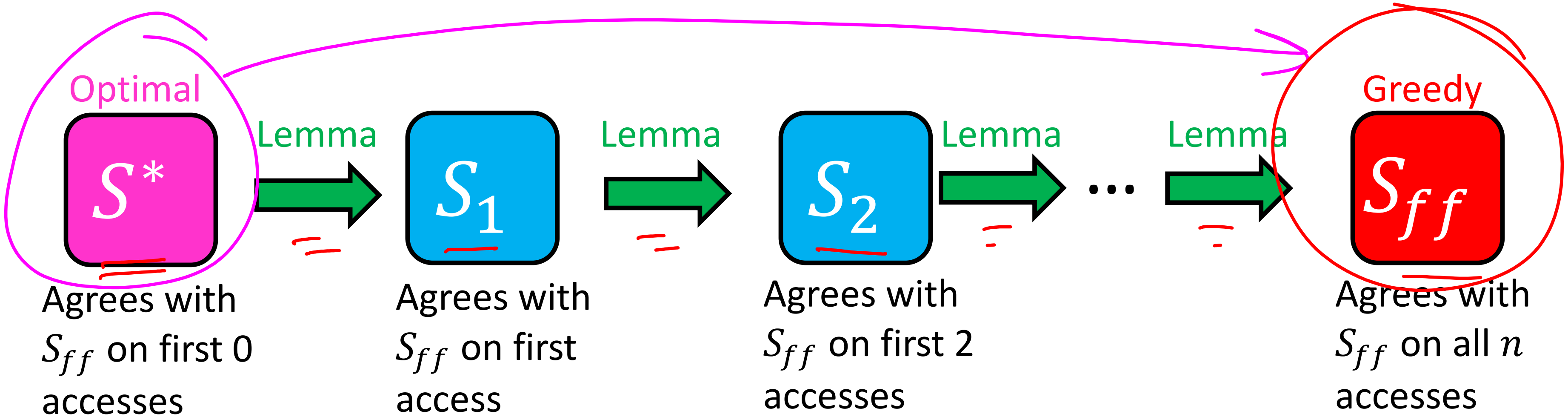


Belady Exchange Lemma

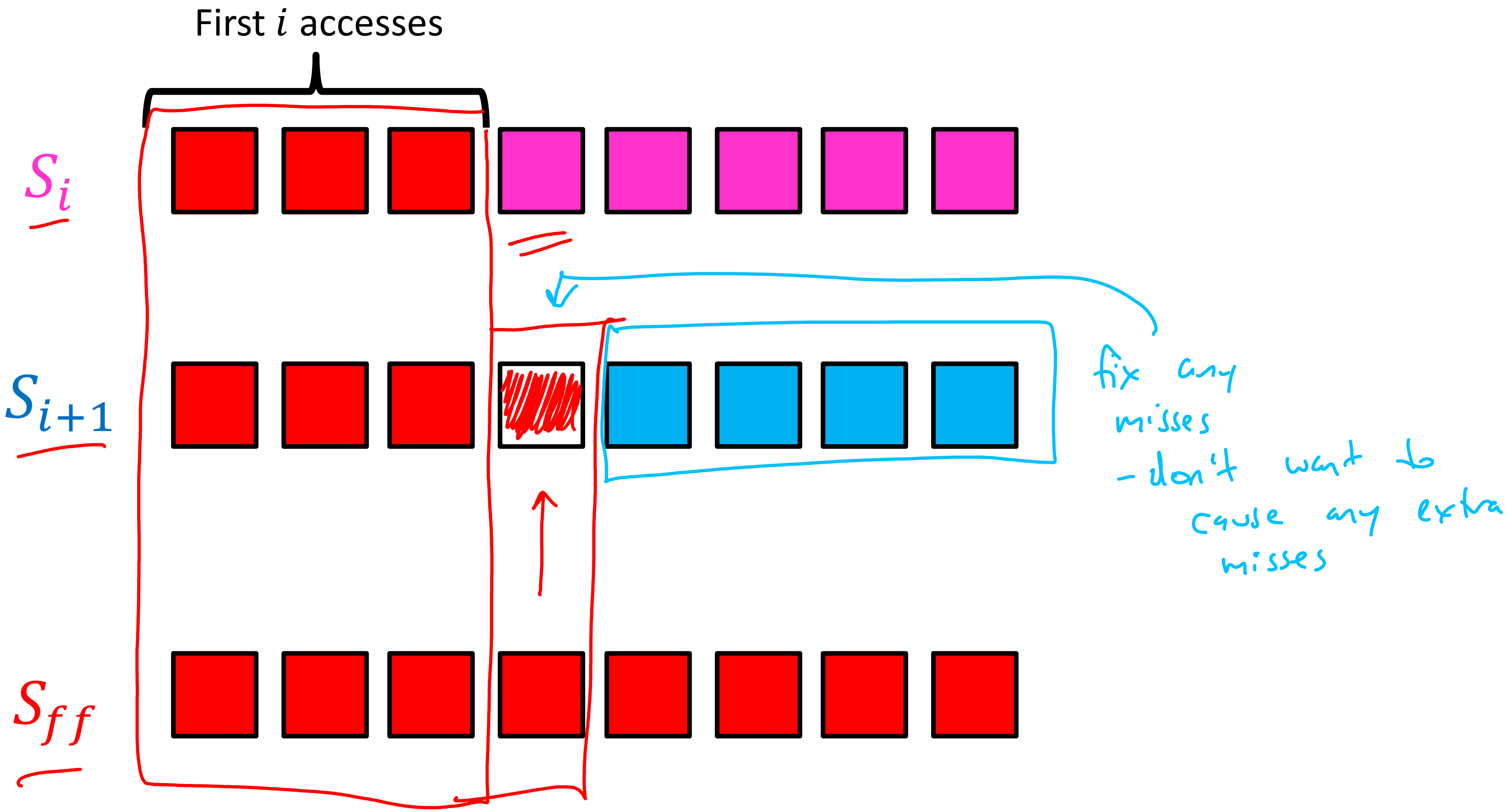
Let S_{ff} be the schedule chosen by our greedy algorithm

Let S_i be a schedule which agrees with S_{ff} for the first i memory accesses.

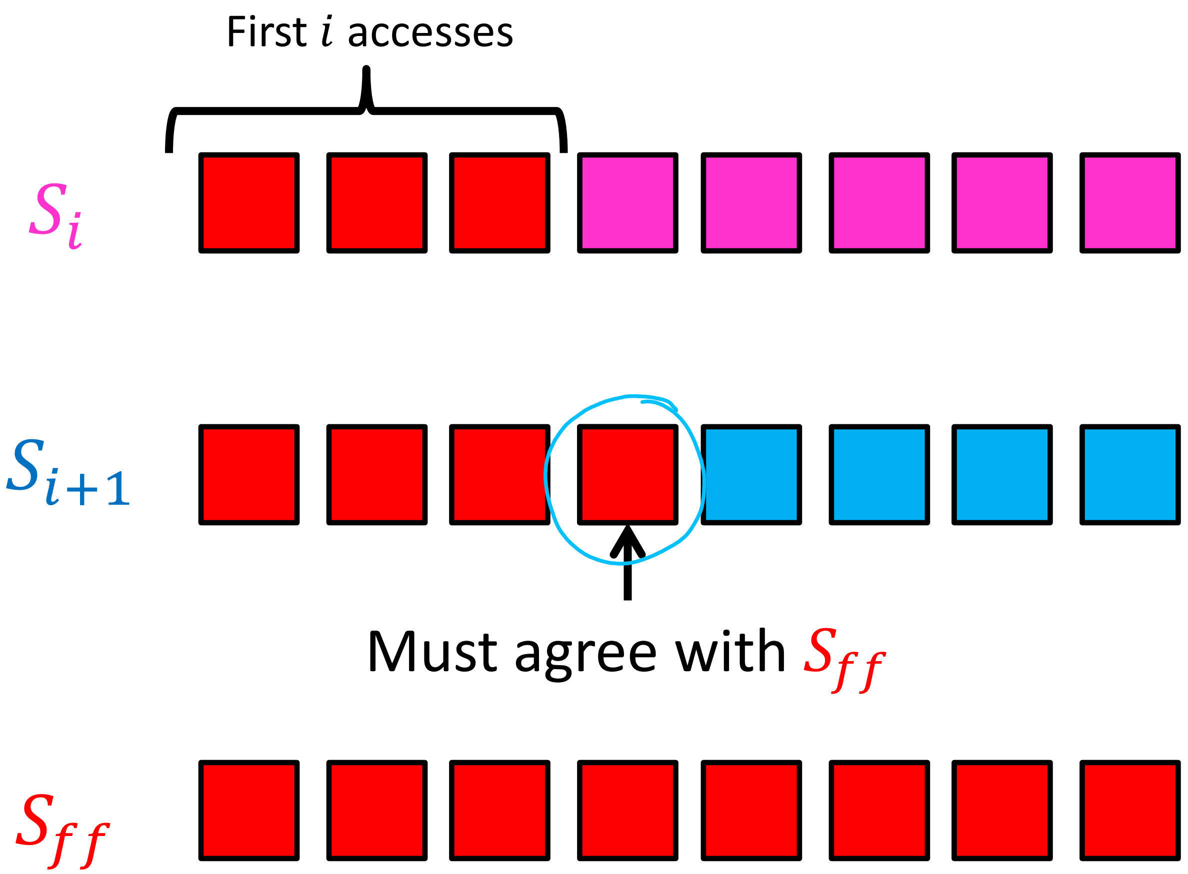
We will show: there is a schedule S_{i+1} which agrees with S_{ff} for the first $i+1$ memory accesses, and has no more misses than S_i (i.e. $misses(S_{i+1}) \leq misses(S_i)$)



Belady Exchange Proof Idea



Belady Exchange Proof Idea



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

Proof of Lemma

Goal: find S_{i+1} s.t. $misses(S_{i+1}) \leq misses(S_i)$

Since S_i agrees with S_{ff} for the first i accesses, the state of the cache at access $i+1$ will be the same

S_i Cache after i	d	e	f

=

S_{ff} Cache after i	d	e	f

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

Case 1: if d is in the cache, then neither S_i nor S_{ff} evict from the cache, use the same cache for S_{i+1}

S_i =

S_{i+1} Cache after i	d	e	f

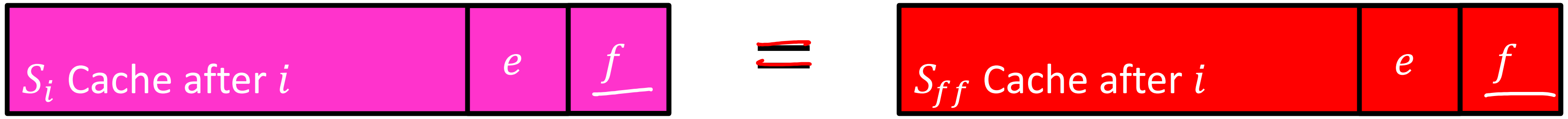
= S_{ff}

$m_{i+2} \to m_n$ S_{i+1} will do what S_i did

Proof of Lemma

Goal: find S_{i+1} s.t. $misses(S_{i+1}) \le misses(S_i)$

Since S_i agrees with S_{ff} for the first i accesses, the state of the cache at access $i+1$ will be the same



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

miss

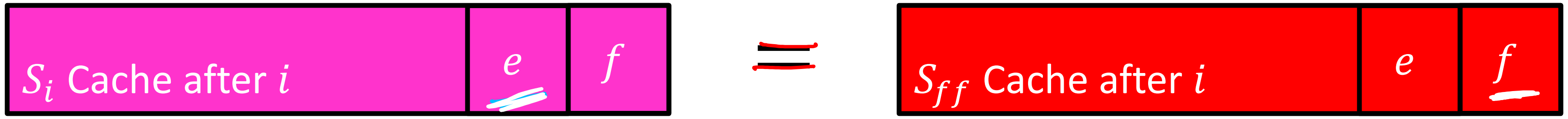
Case 2: if d isn't in the cache, and both S_i and S_{ff} evict f from the cache, evict f for d in S_{i+1}

S_{i+1} will do what S_i did for the rest of the accesses. => same # of misses

Proof of Lemma

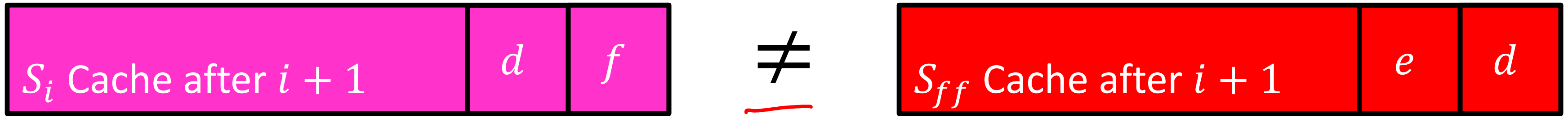
Goal: find S_{i+1} s.t. $misses(S_{i+1}) \leq misses(S_i)$

Since S_i agrees with S_{ff} for the first i accesses, the state of the cache at access $i+1$ will be the same

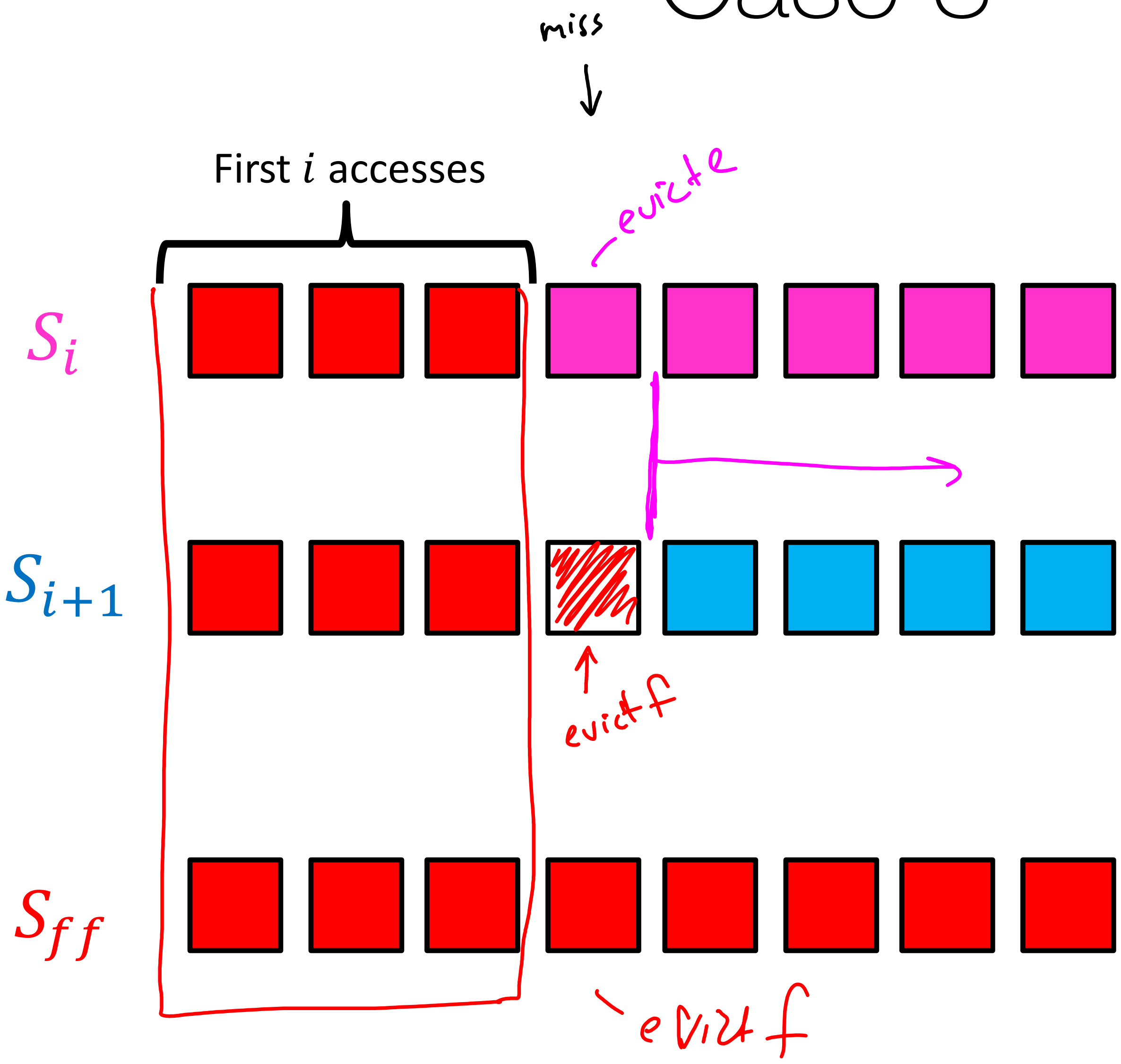


Consider access $m_{i+1} = d$

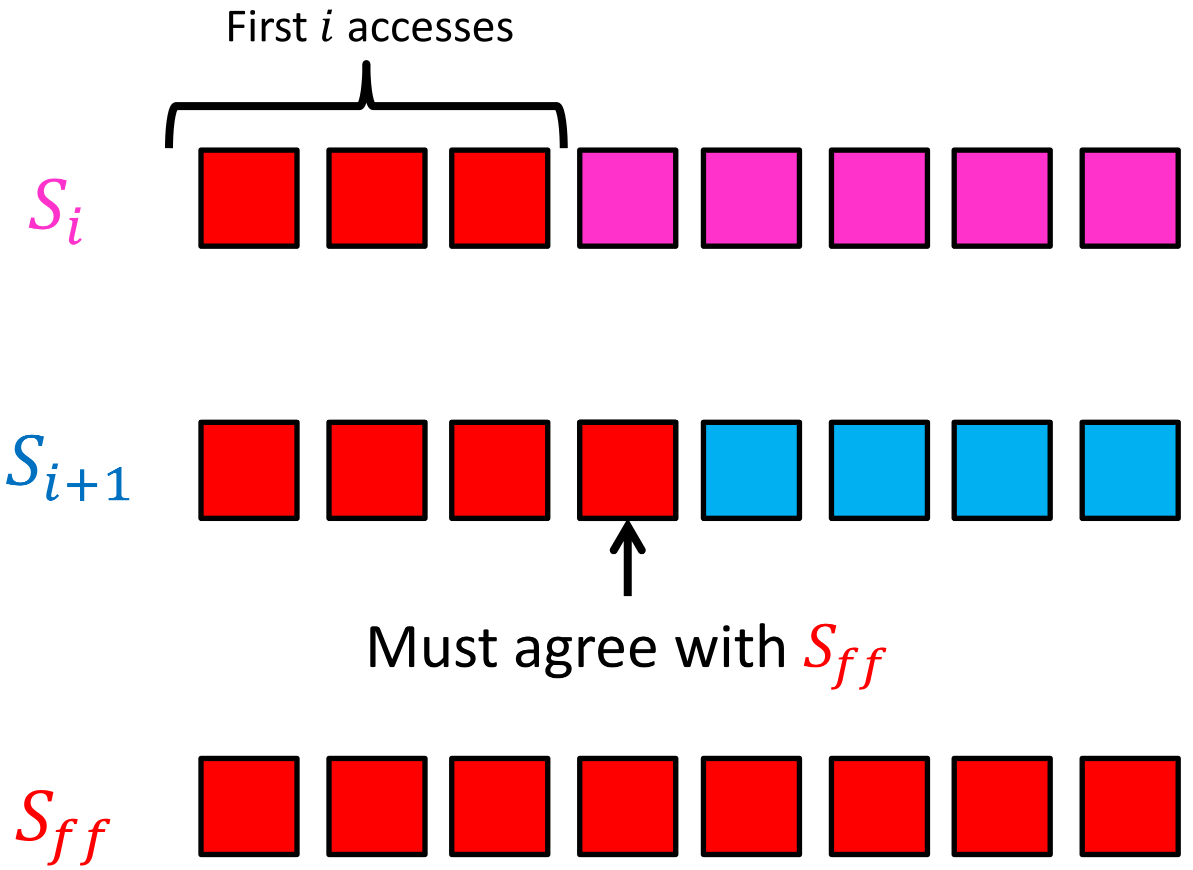
Case 3: if d isn't in the cache, S_i evicts e and S_{ff} evicts f from the cache



Case 3

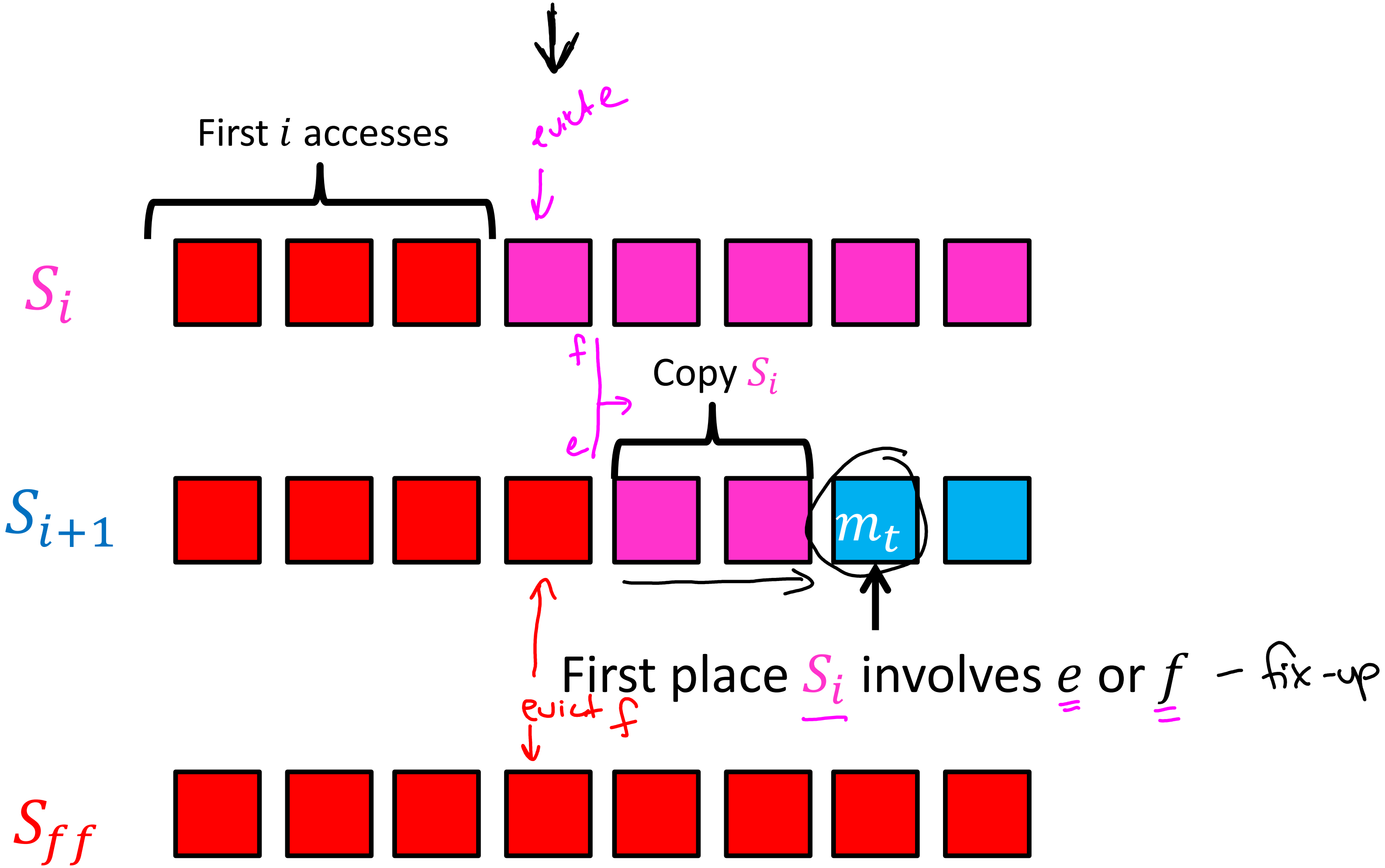


Case 3



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

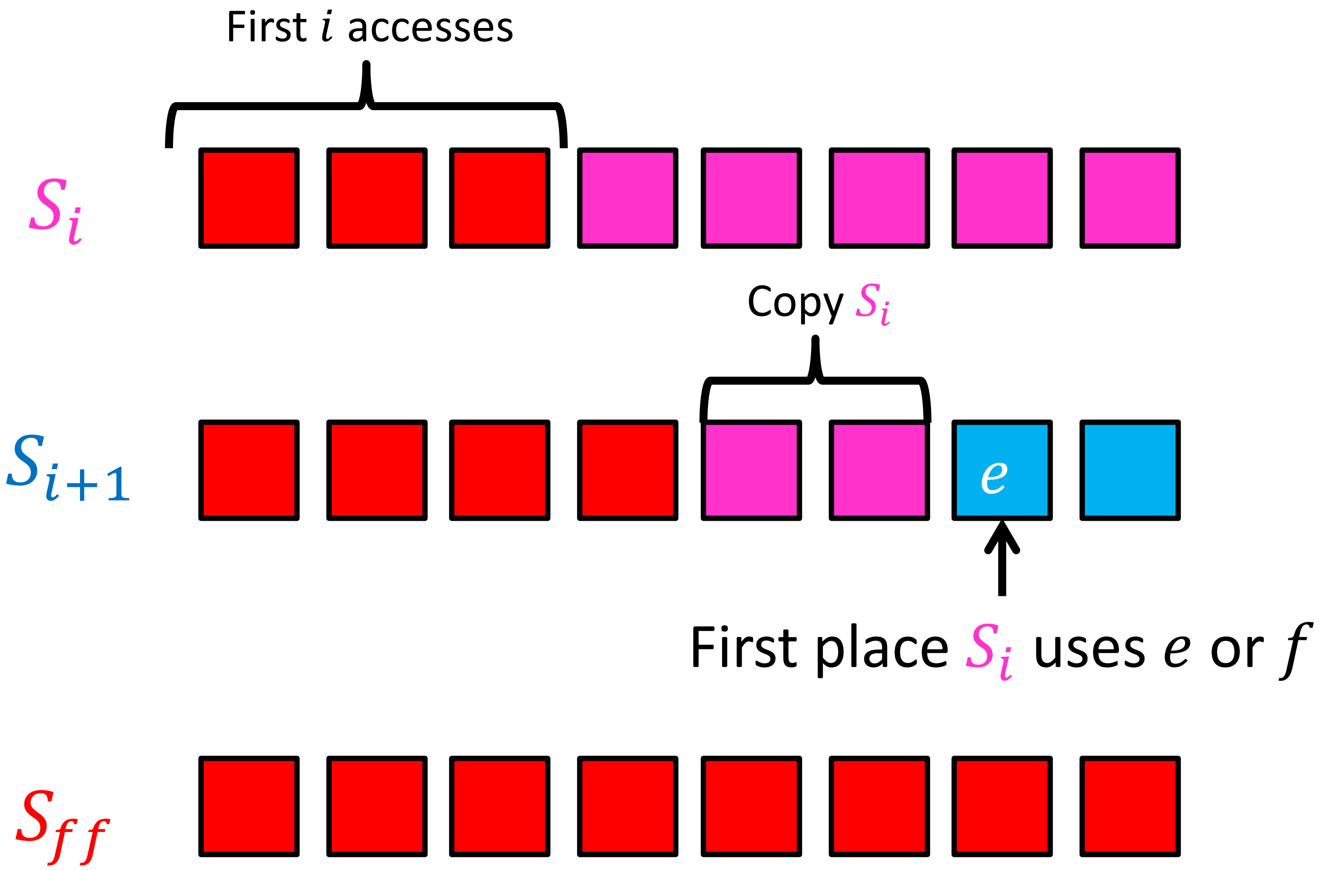
Case 3



m_t = the first access after $i + 1$ in which S_i deals with e or f

3 options: $\boldsymbol{m_t = e}$ or $\boldsymbol{m_t = f}$ or $\boldsymbol{m_t = x \neq e, f}$

Case 3, $m_t = e$

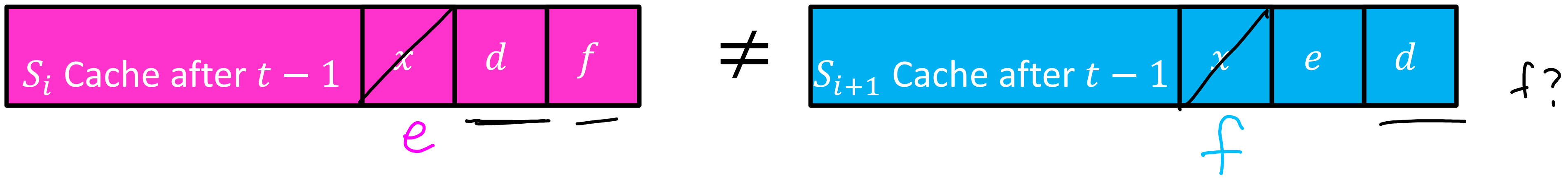


m_t = the first access after $i+1$ in which S_i deals with e or f

3 options: $\underline{\boldsymbol{m_t = e}}$ or $\boldsymbol{m_t = f}$ or $\boldsymbol{m_t = x \neq e, f}$

Case 3, $m_t = e$

Goal: find S_{i+1} s.t. $misses(S_{i+1}) \leq misses(S_i)$



S_i must load e into the cache, assume it evicts x

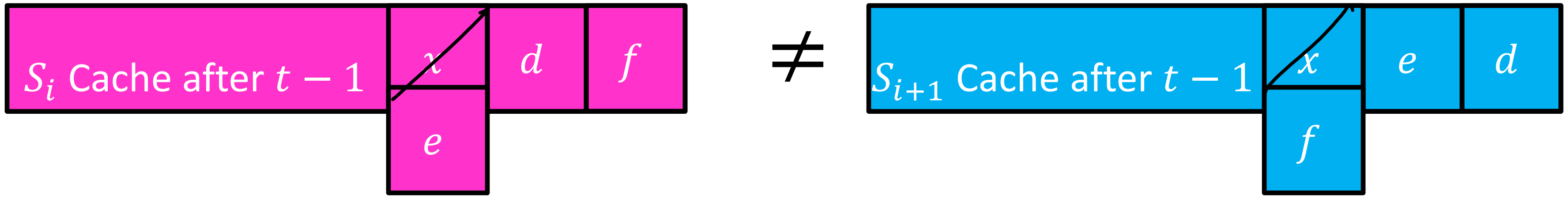
- cache miss

S_{i+1} will load f into the cache, evict x

now caches match!

Case 3, $m_t = e$

Goal: find S_{i+1} s.t. $misses(S_{i+1}) \leq misses(S_i)$



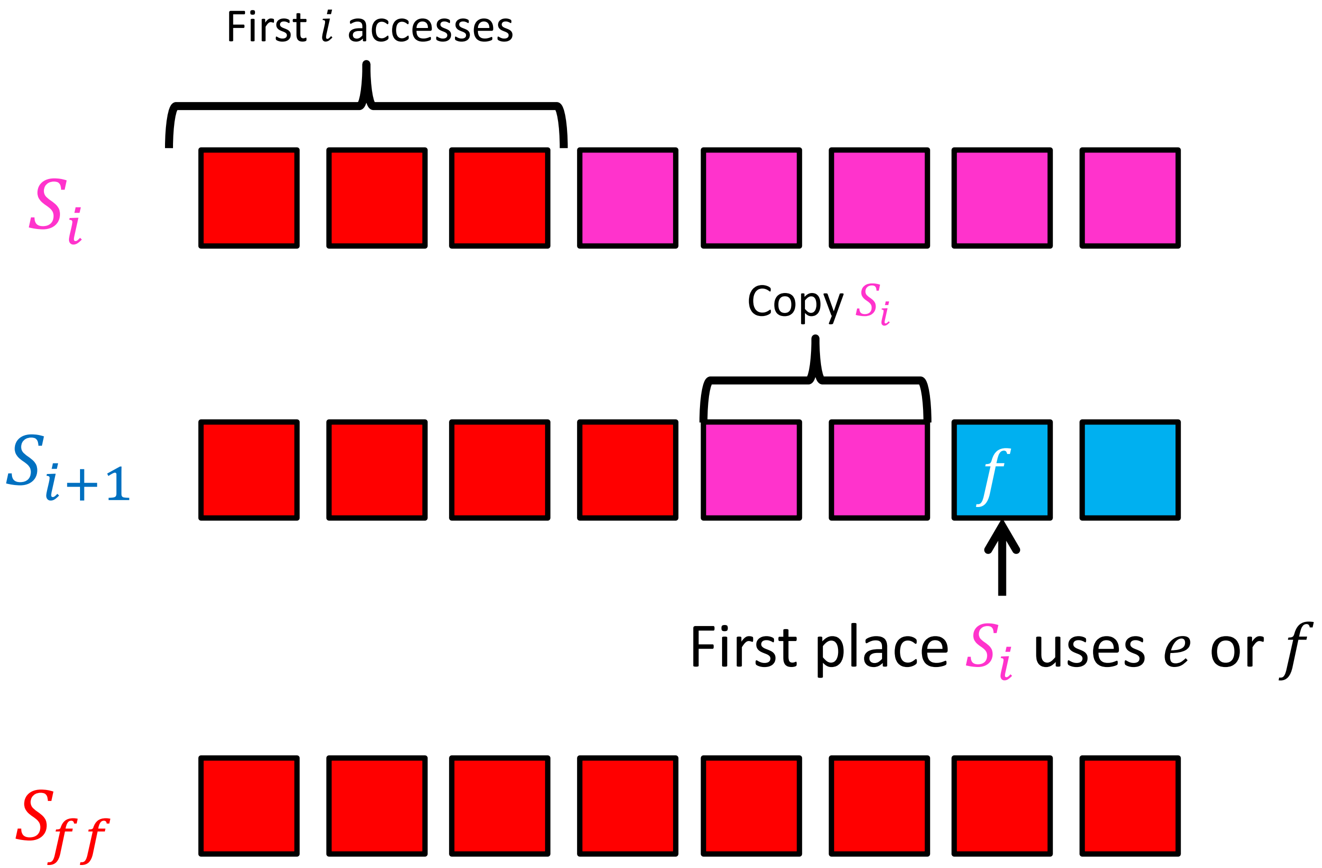
S_i must load e into the cache, assume it evicts x

S_{i+1} will load f into the cache, evicting x

The caches now match!

S_{i+1} behaved exactly the same as S_i between $i+1$ and t, and has the same cache after t, therefore $misses(S_{i+1}) = misses(S_i)$

Case 3, $m_t = f$

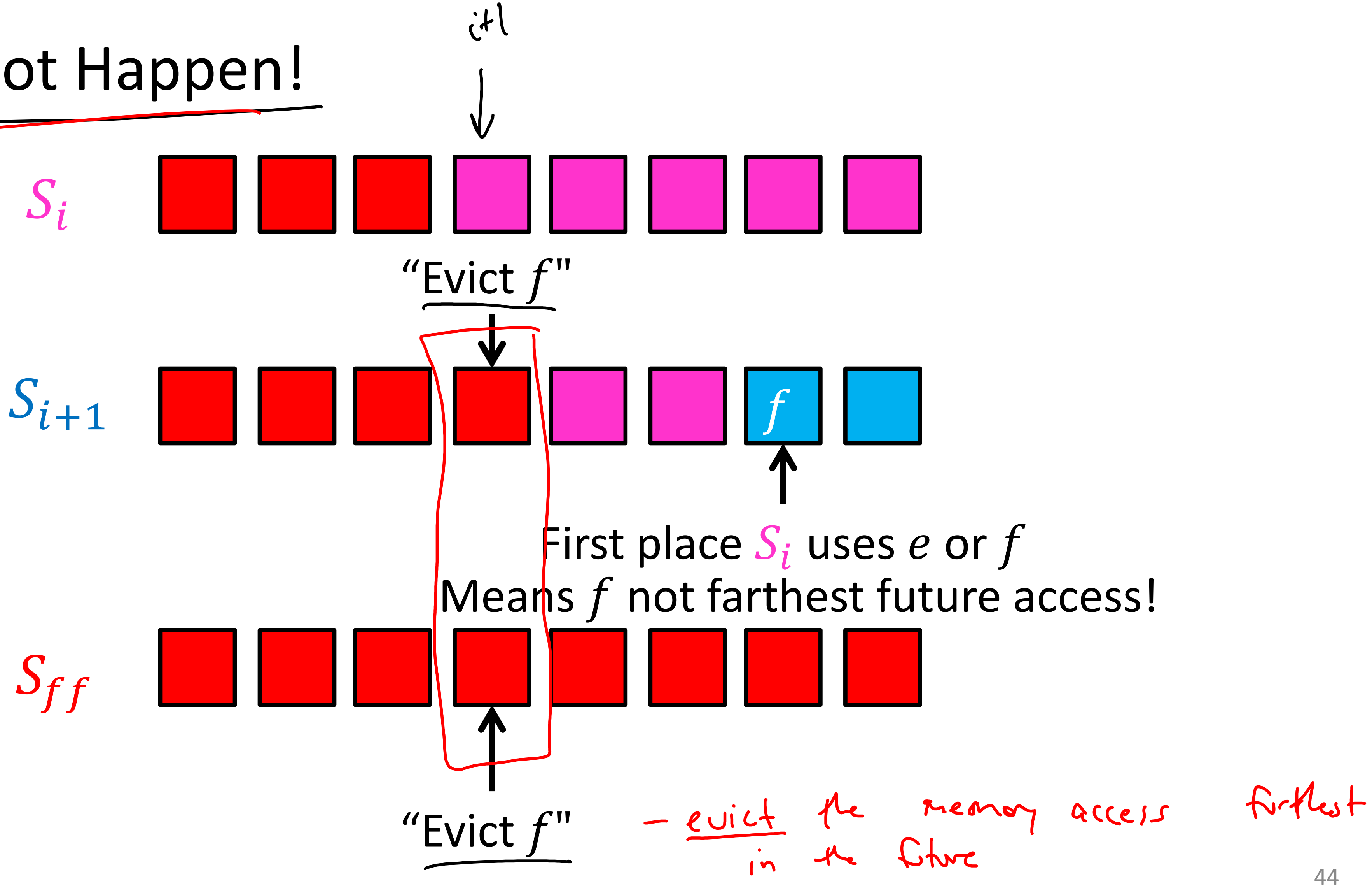


m_t = the first access after $i + 1$ in which S_i deals with e or f

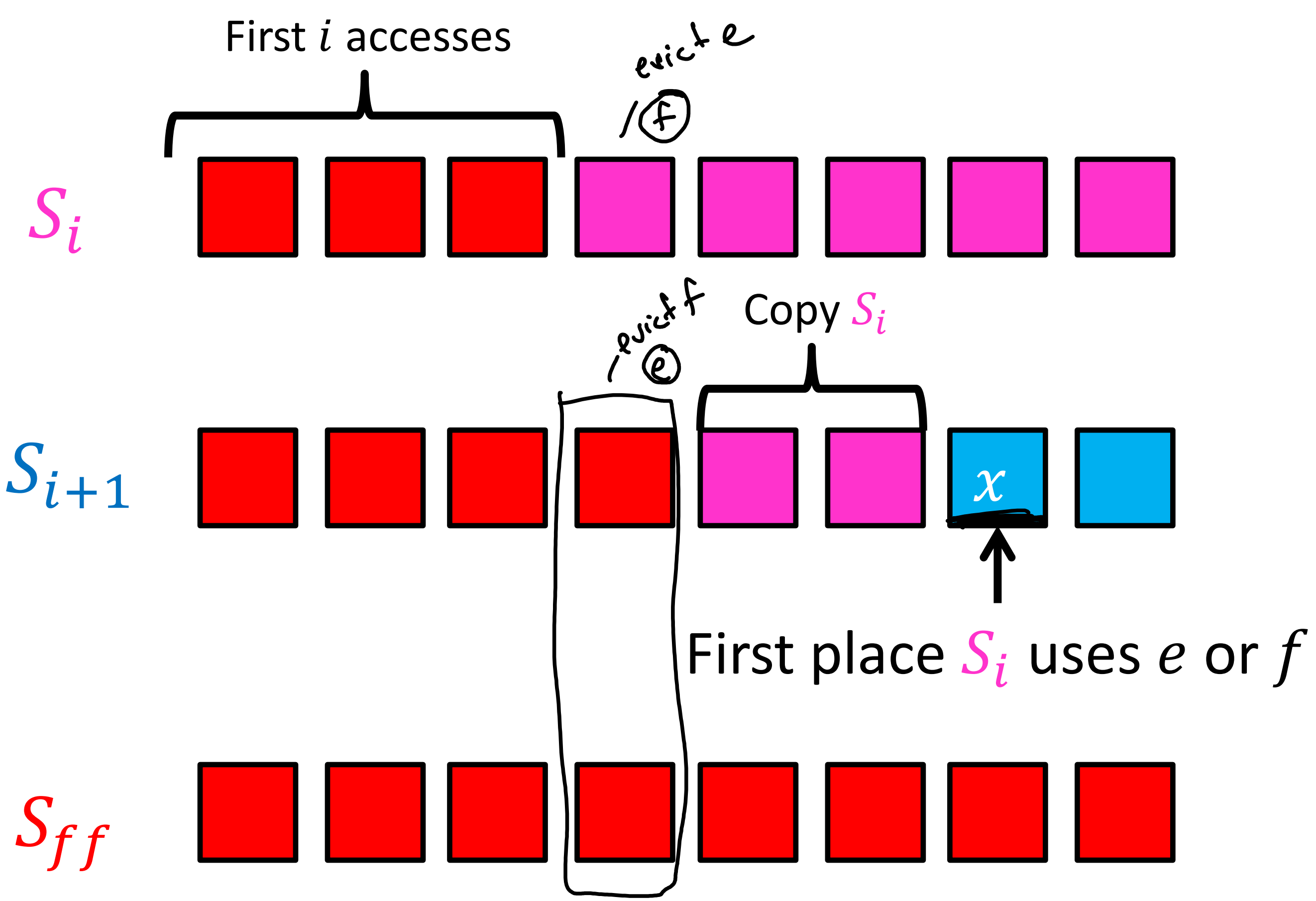
3 options: $m_t = e$ or $\boldsymbol{m_t = f}$ or $m_t = x \neq e, f$

Case 3, $m_t = f$

Cannot Happen!



Case 3, $m_t = x \neq e, f$

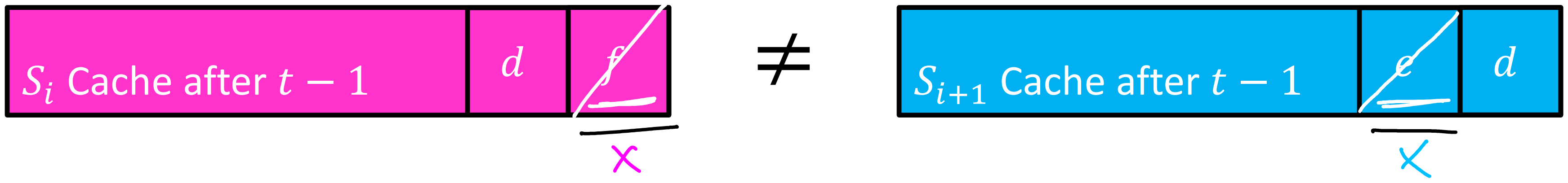


m_t = the first access after $i+1$ in which S_i deals with e or f

3 options: $m_t = e$ or $m_t = f$ or $\boldsymbol{m_t = x \neq e, f}$

Case 3, $m_t = x \neq e, f$

Goal: find S_{i+1} s.t. $misses(S_{i+1}) \leq misses(S_i)$



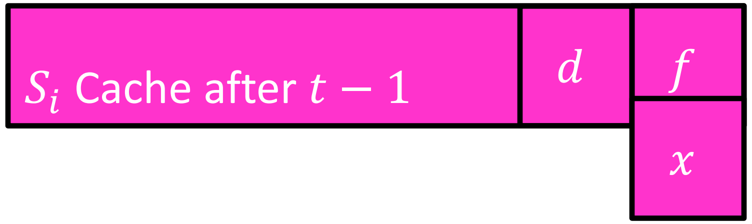
S_i loads x into the cache, it must be evicting f – cache miss

S_{i+1} also has cache miss for x, evict e, load x

caches now match!

Case 3, $m_t = x \neq e, f$

Goal: find S_{i+1} s.t. $misses(S_{i+1}) \leq misses(S_i)$



$\neq$

S_i loads x into the cache, it must be evicting f

S_{i+1} will load x into the cache, evicting e

The caches now match!

S_{i+1} behaved exactly the same as S_i between $i+1$ and t, and has the same cache after t, therefore $misses(S_{i+1}) = misses(S_i)$

Use Lemma to show Optimality