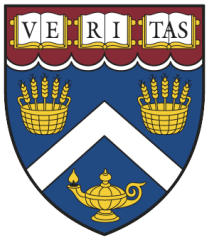
Compacting Collectors



Compacting Garbage Collection

- Copy reserve overhead can be significant
  - Later algorithms reduce it by a lot
- Can we get the benefits of copying without it?
- Compacting algorithms
  - Eliminate fragmentation by moving objects
  - Move objects within the same heap region

Compaction Algorithms

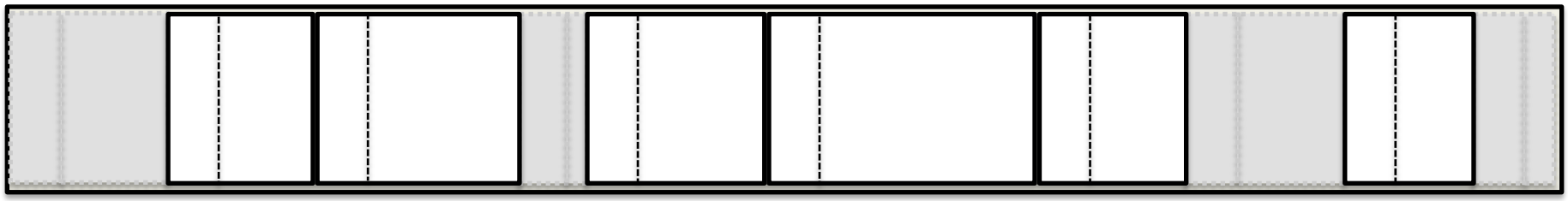
- Compacting happens in multiple passes
  - Figure out where objects will go
  - Update object references
  - Move the objects
- Need to track where the object goes after copy
  - Install forwarding pointers for copying GCs
  - We can overwrite headers because there is a copy
- Can't use forwarding pointers when compacting

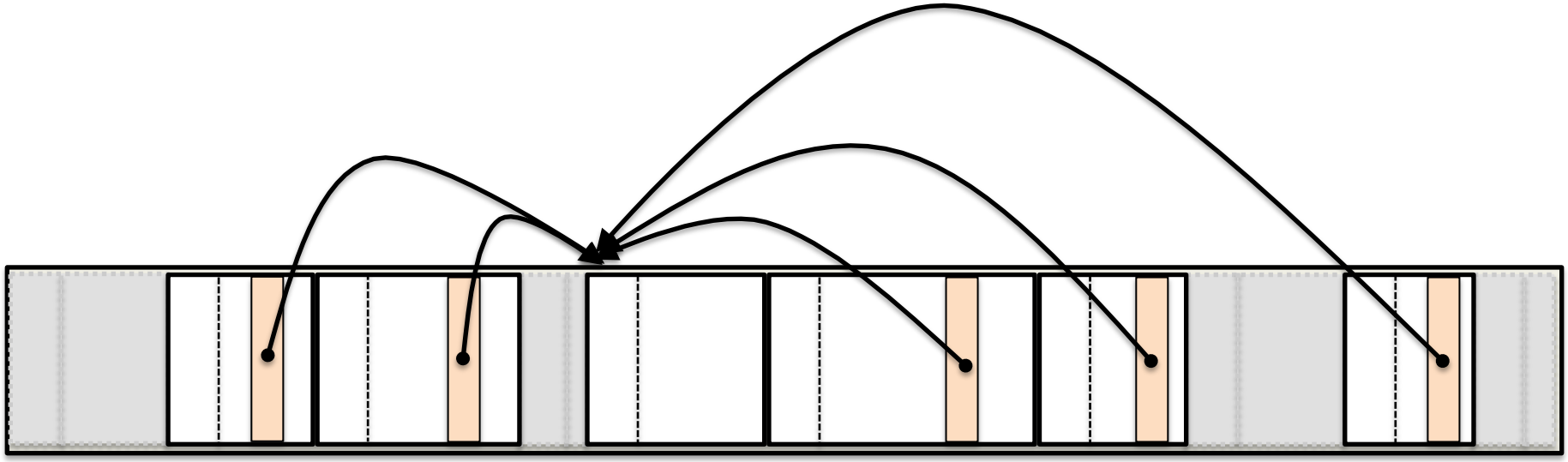
Tracking Forwarding Locations

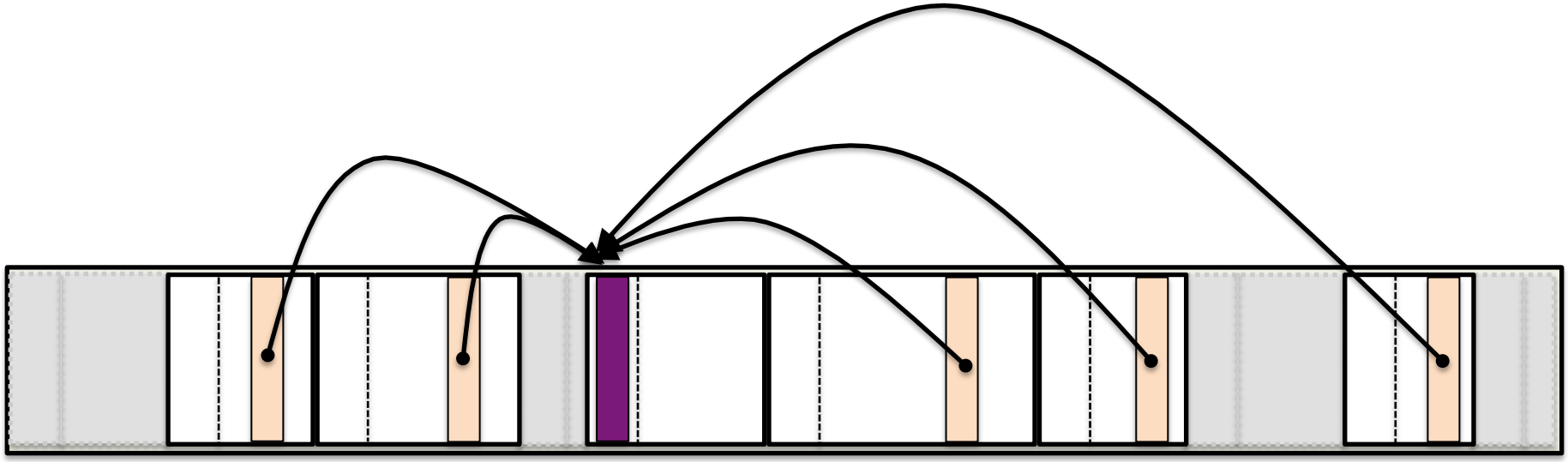
- Use an extra word per object
  - High per-object overhead
- Use a table off to the side
  - Table size scales with heap size
- Chain references

Reference Chaining

- Object graph is directional
  - Reference fields point to objects
  - No pointer in the other direction
- Can we reverse the direction of the graph?
  - Many-to-one problem
- We can build a linked list of references
  - Assume one word available in the header
  - Assume alignment gives us one free bit per pointer

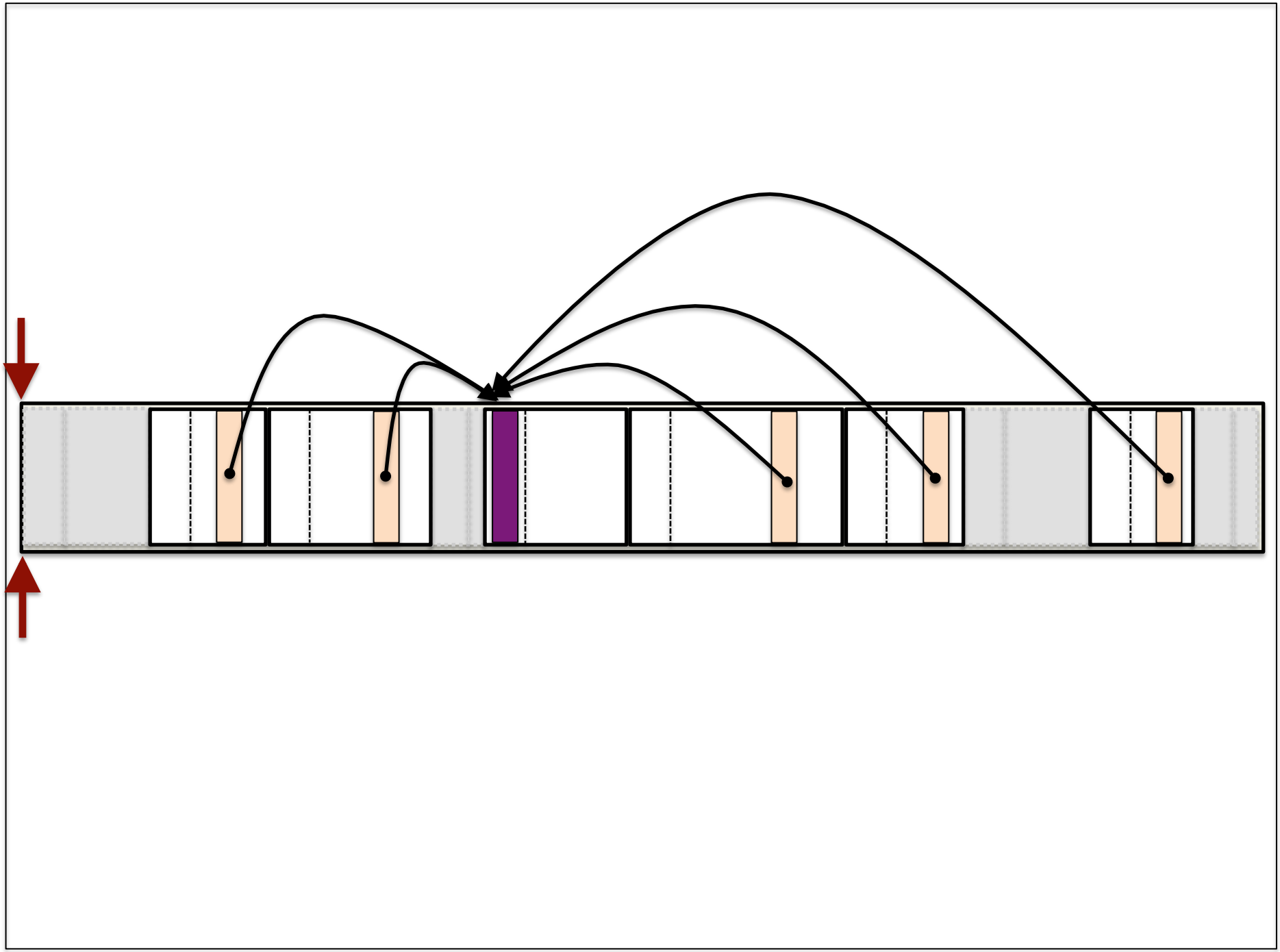


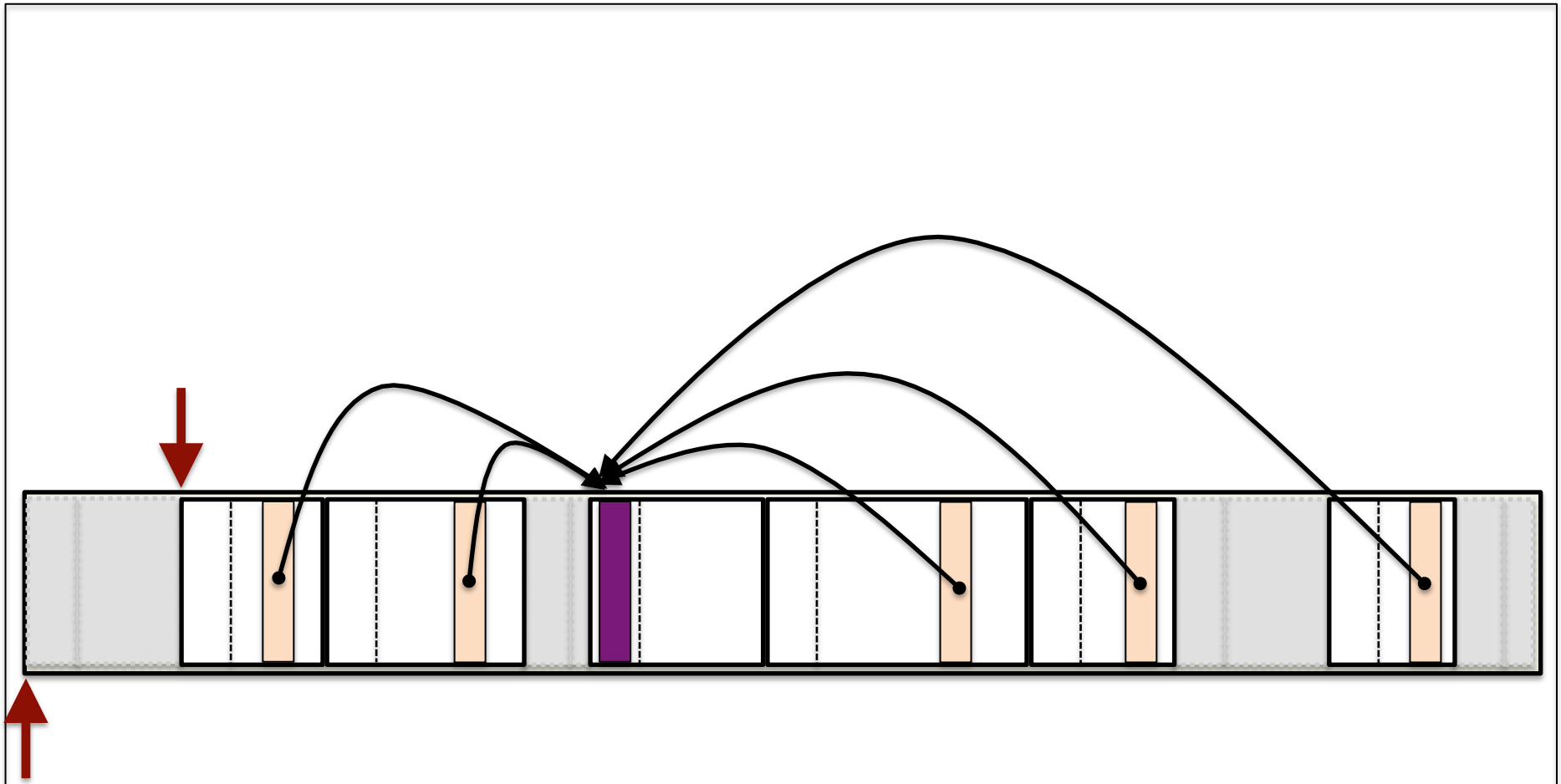


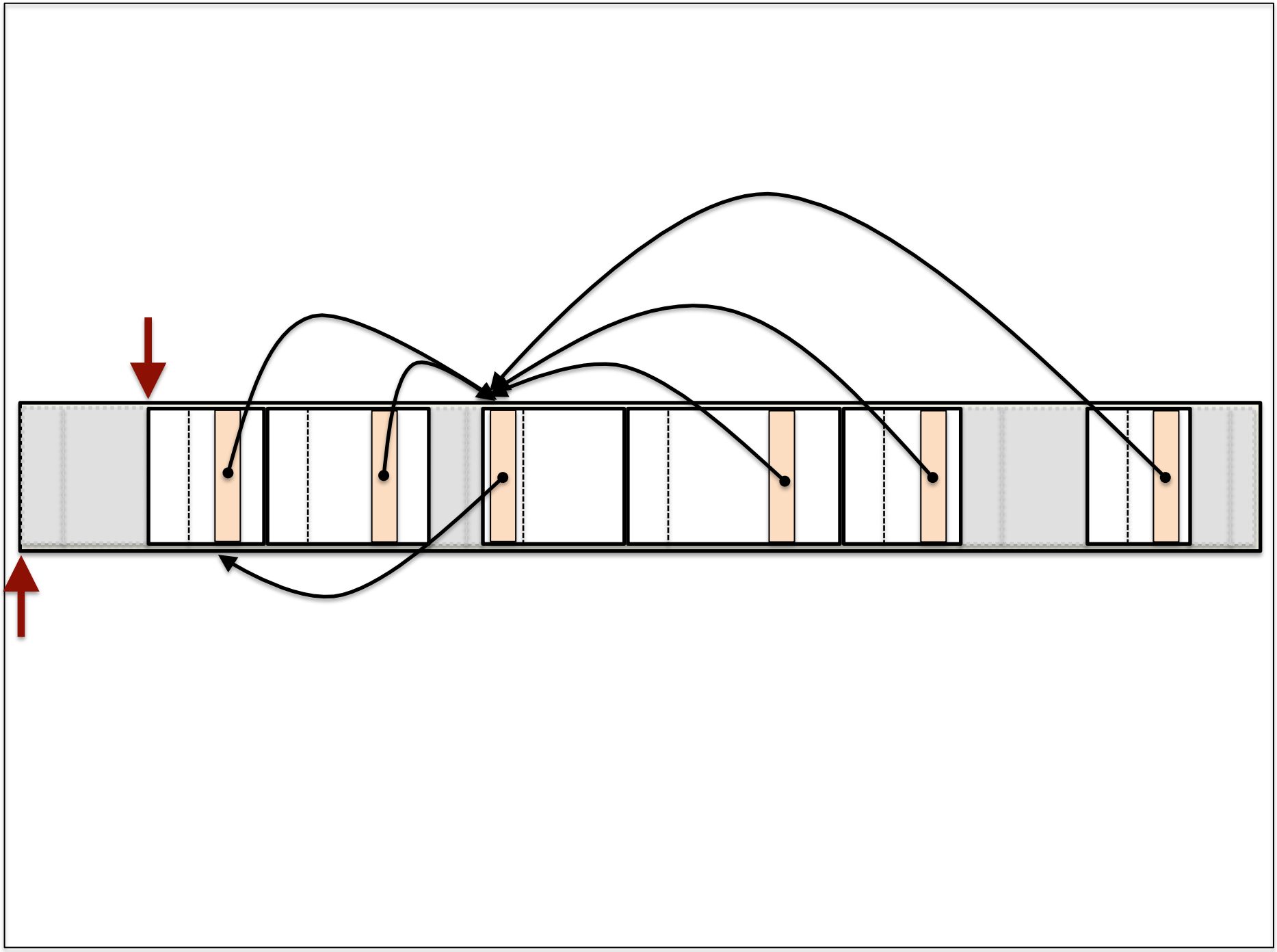


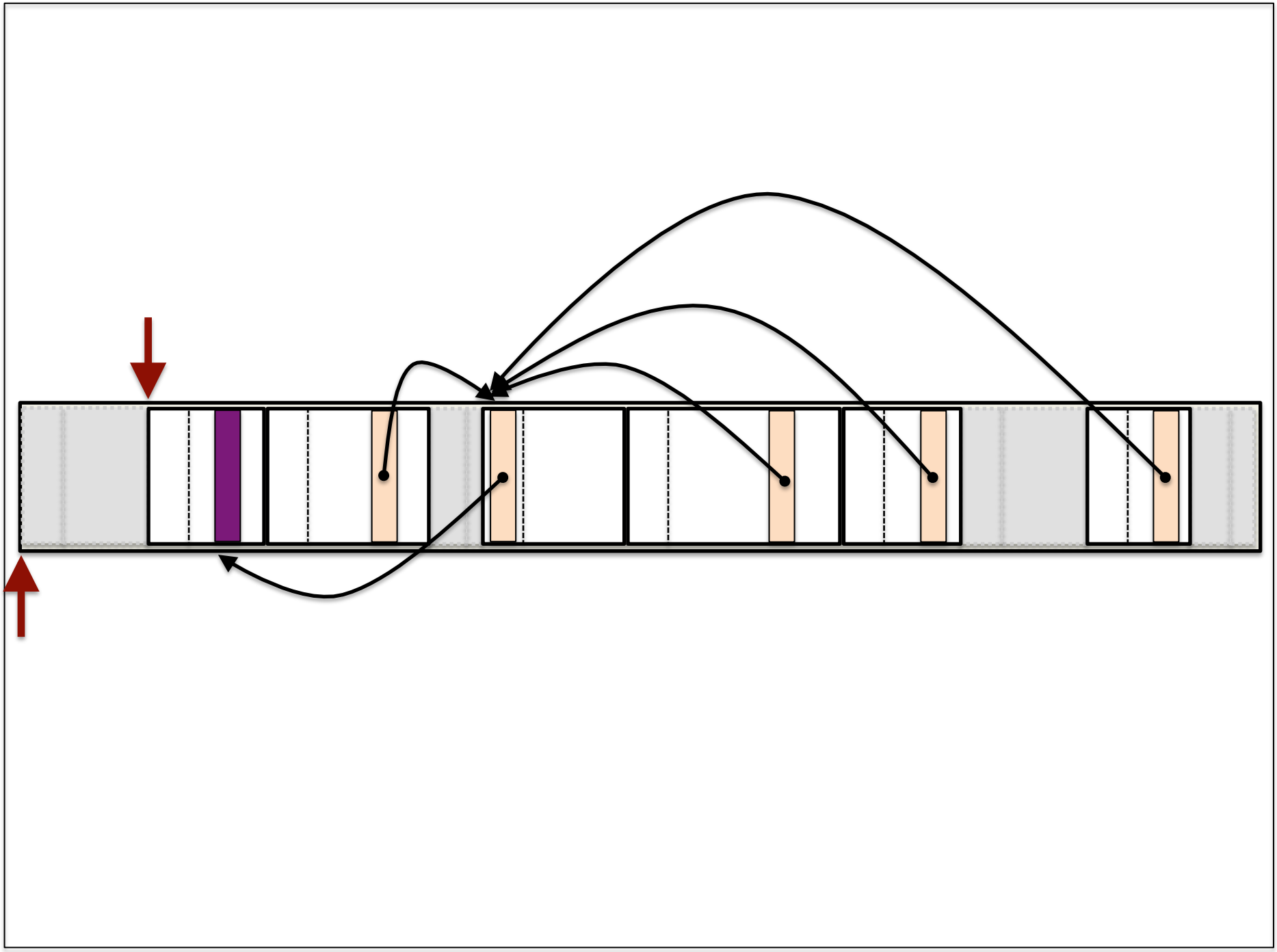
First Pass

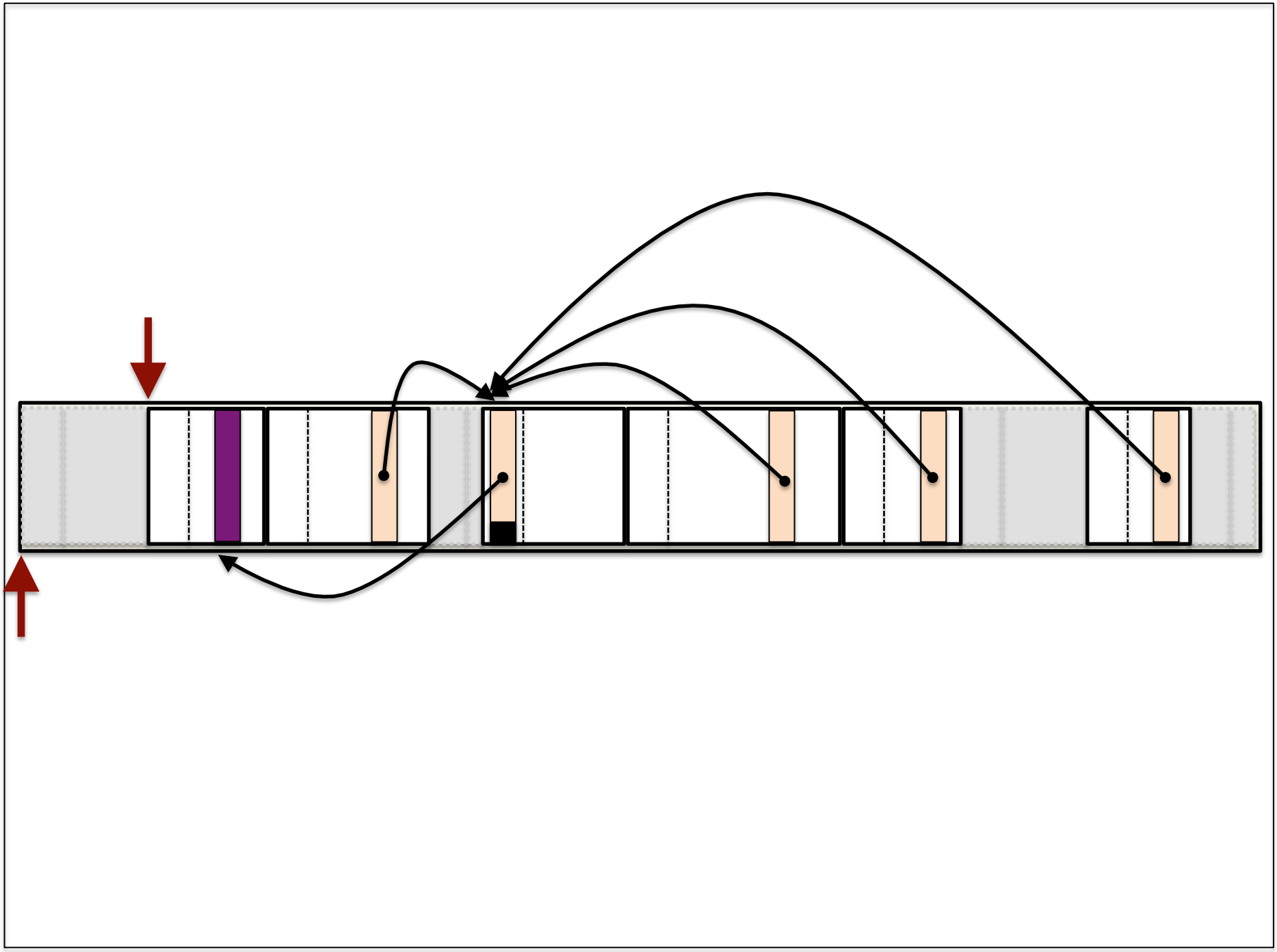
- Scan the heap linearly
  - We don't have to worry about holes
- Look only at references to live objects
  - Follows from tracing only marked objects
- Keep track of object sizes
- Reverse the direction of pointers
  - Use one bit as a flag to keep track

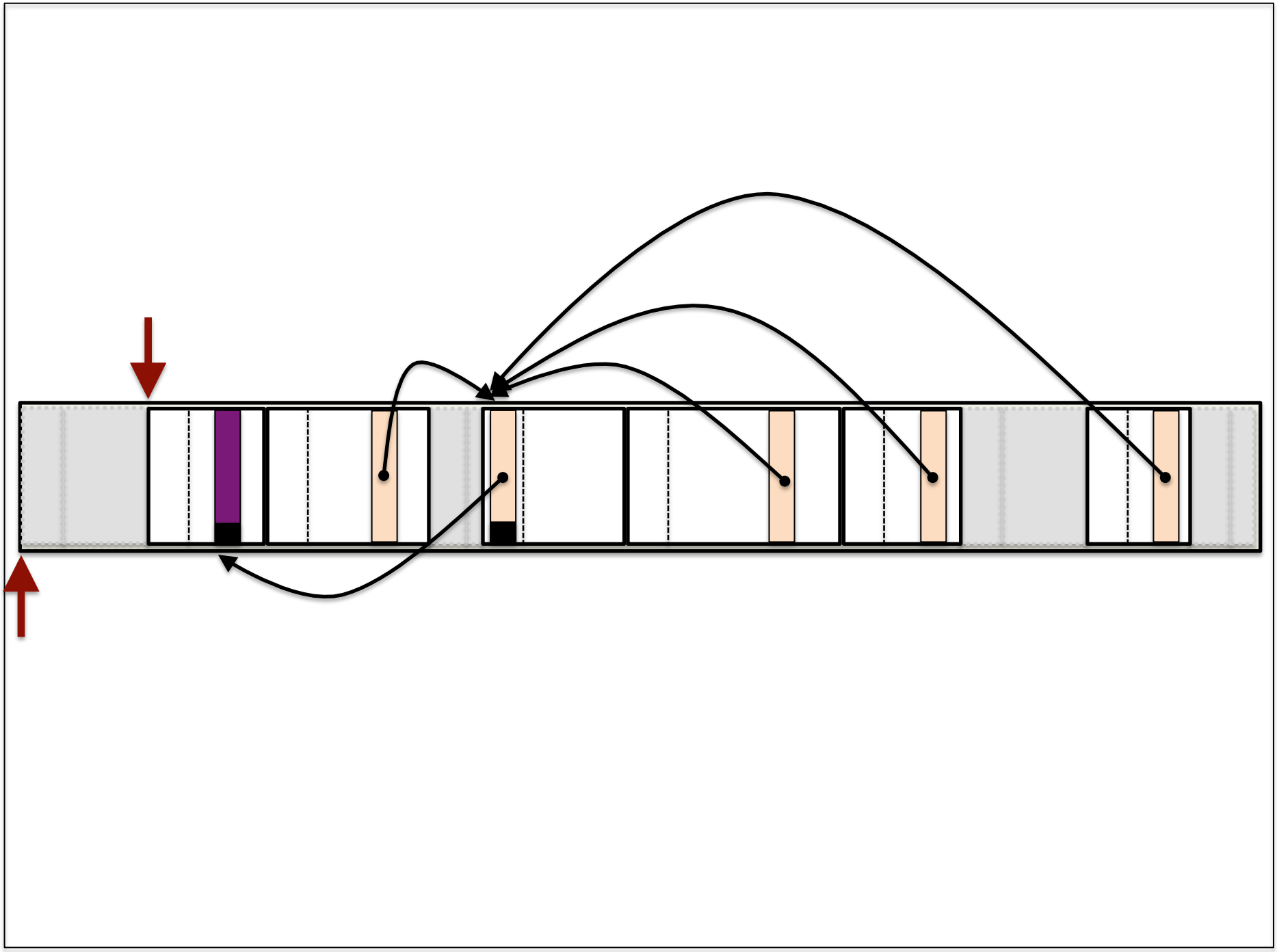


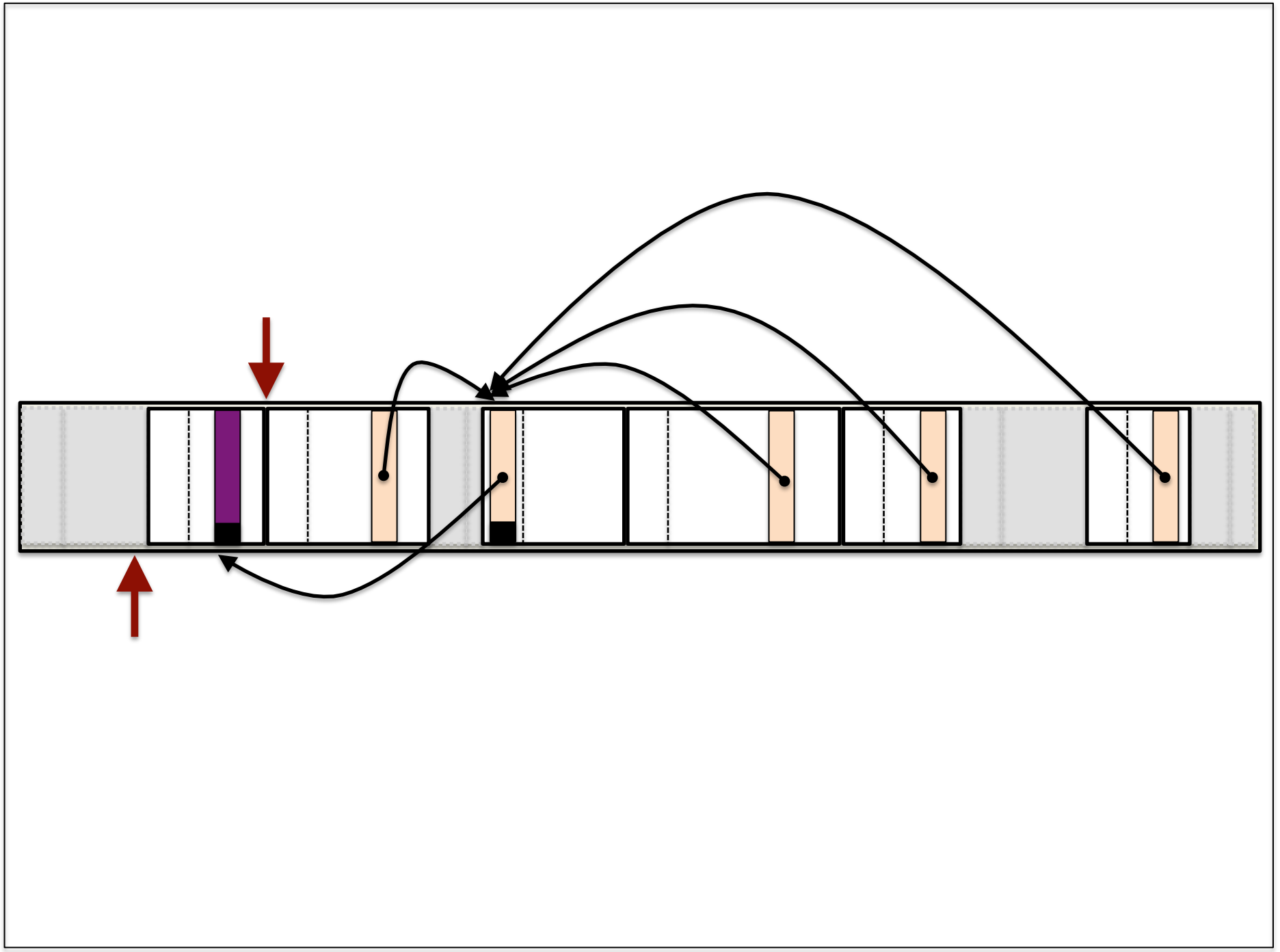


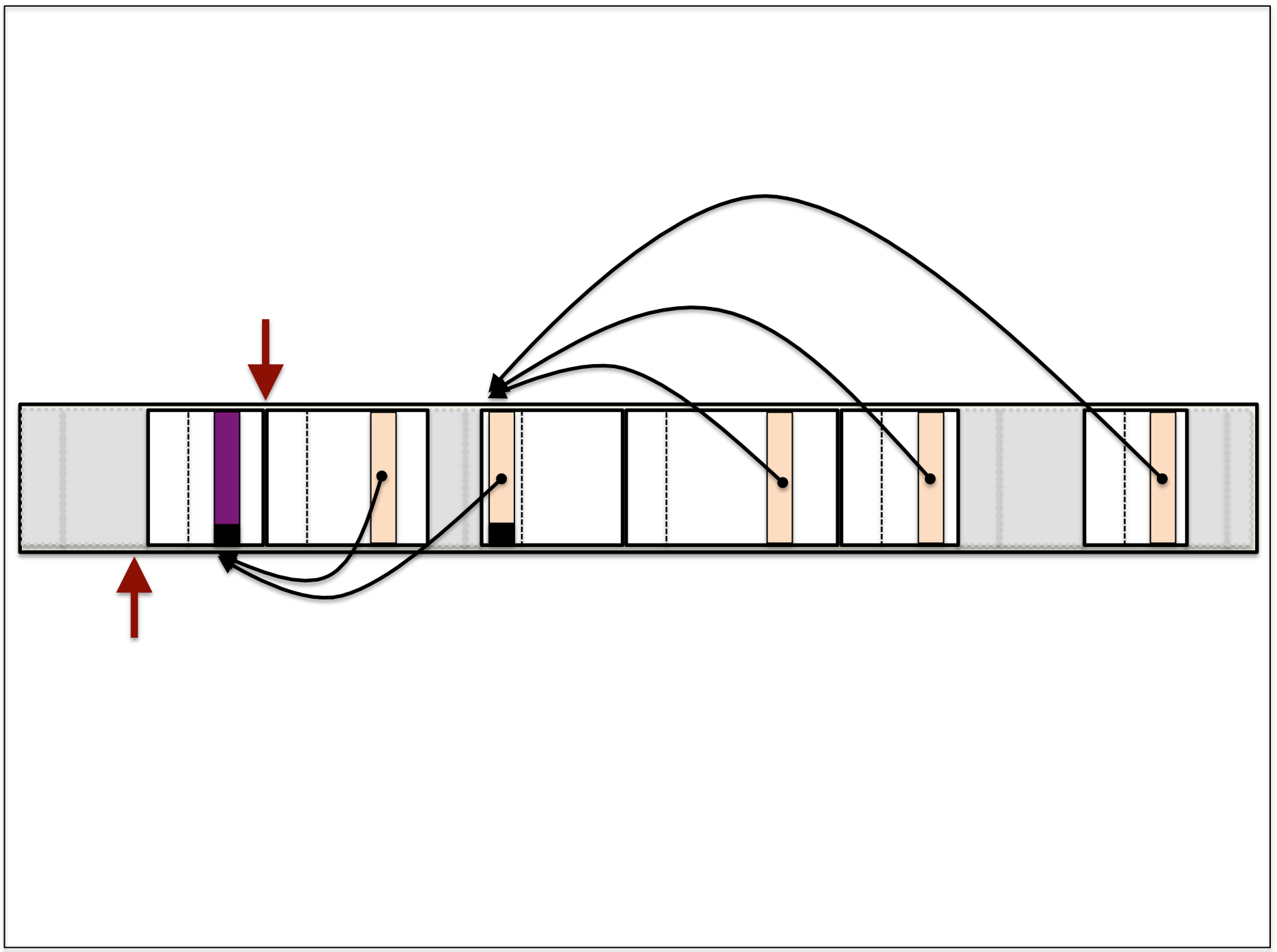


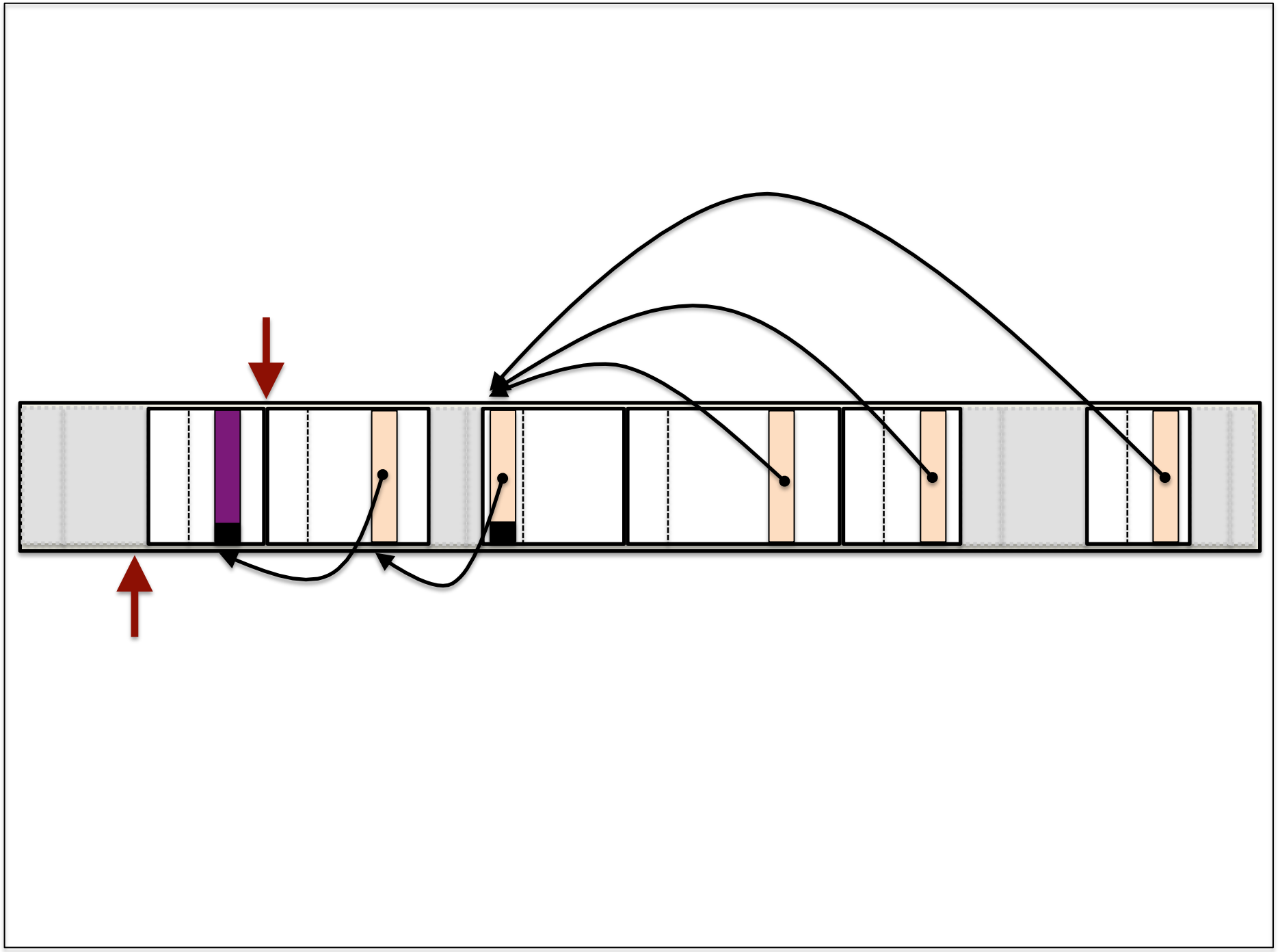


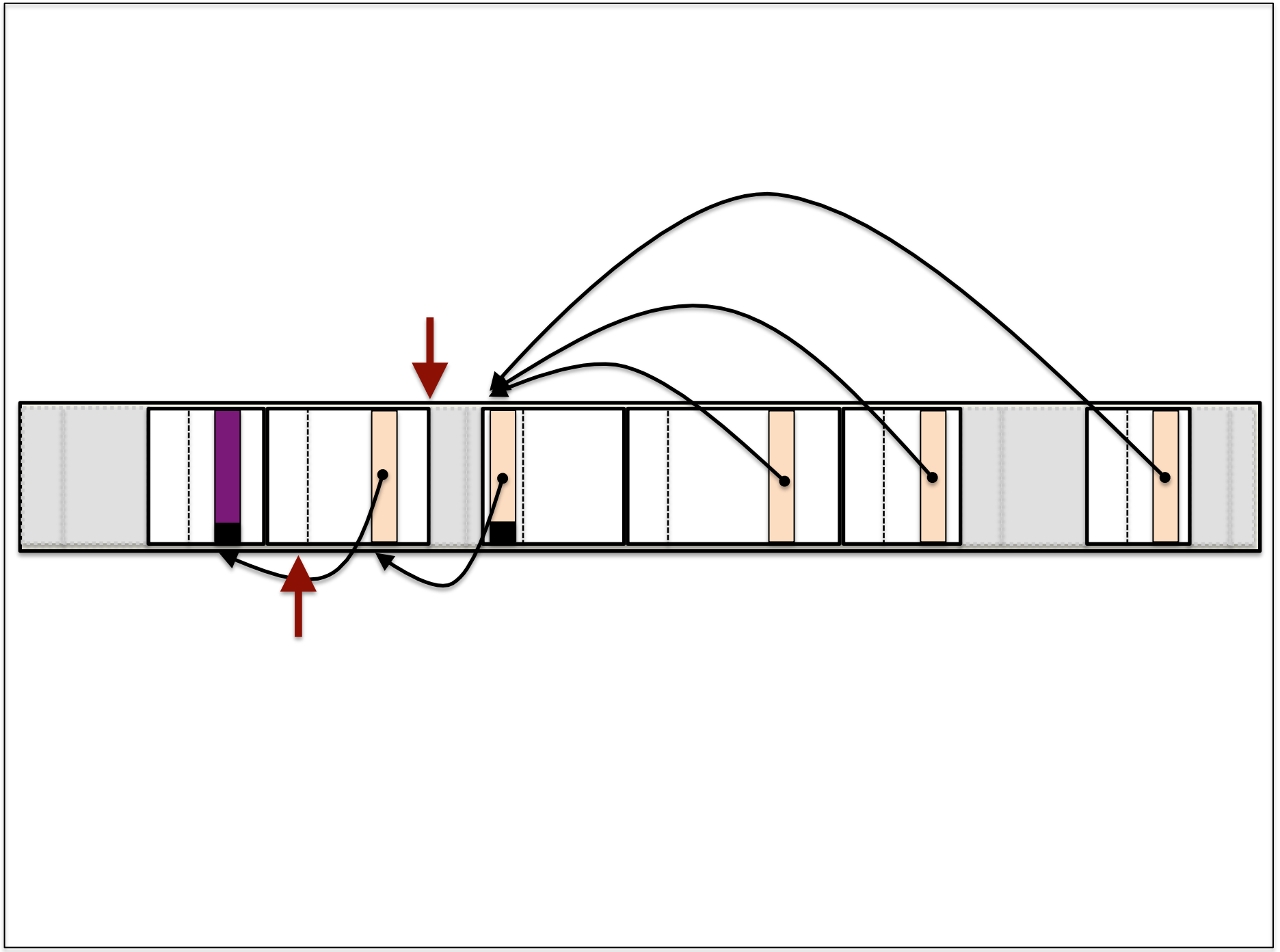


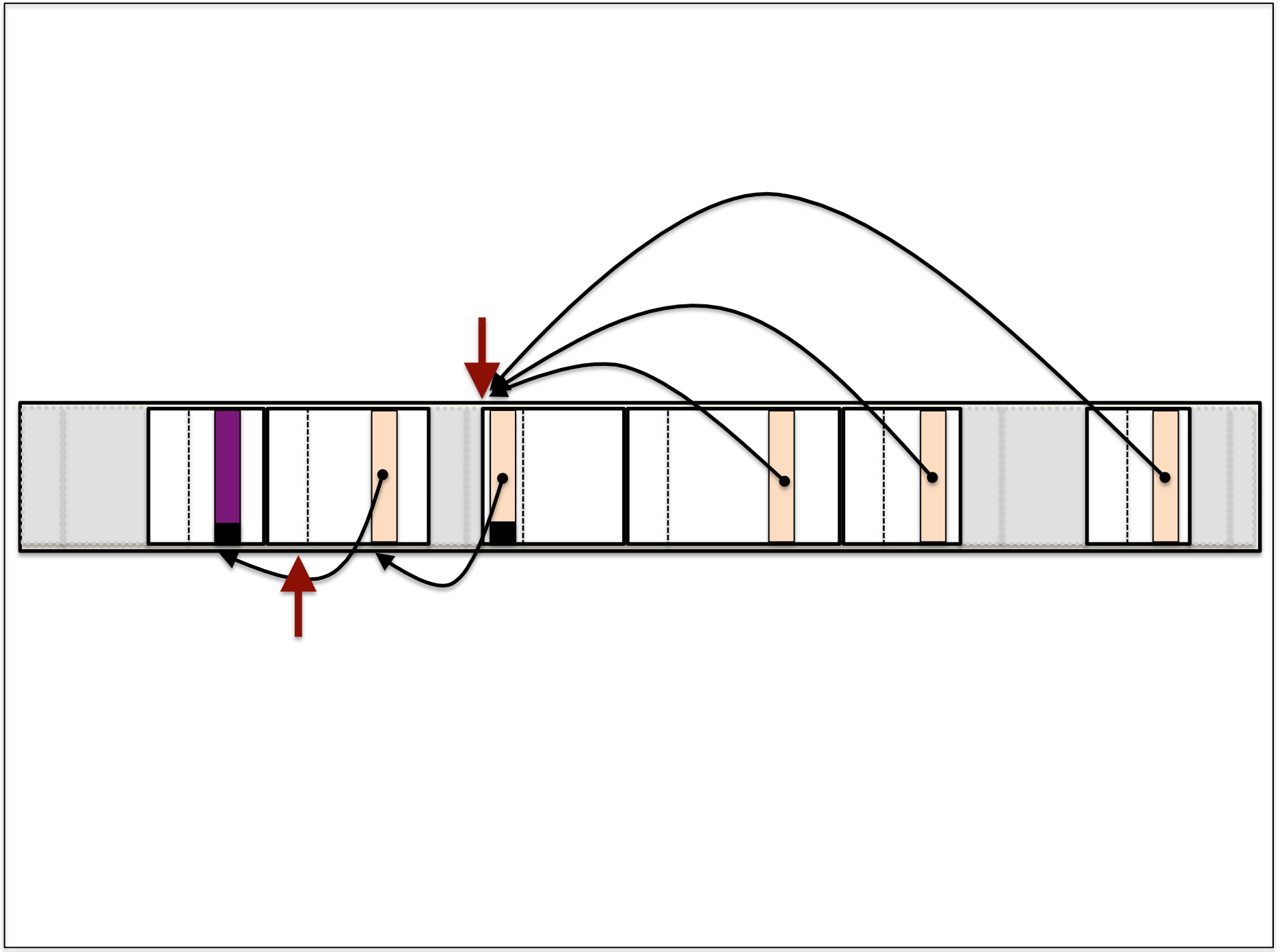






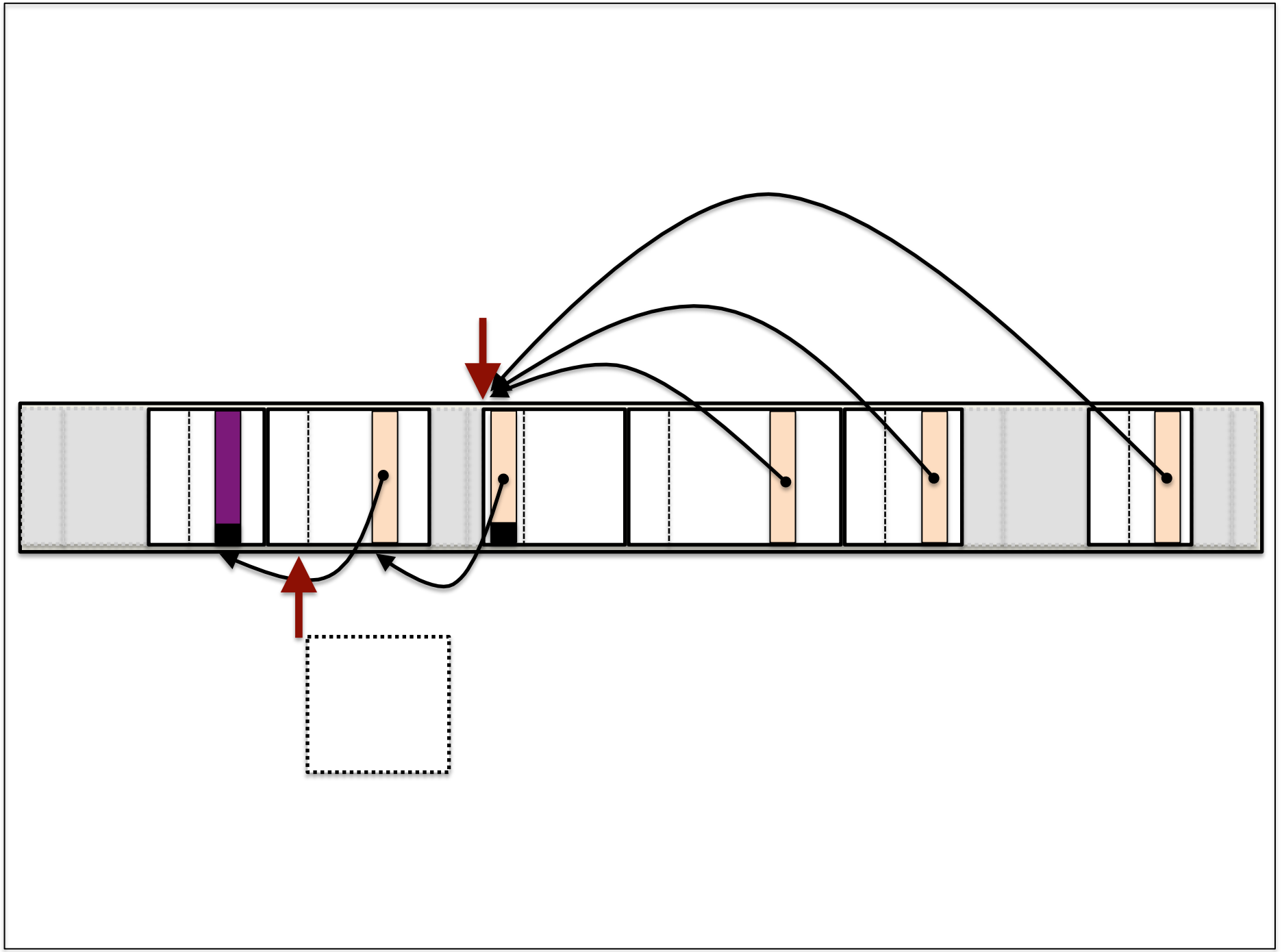


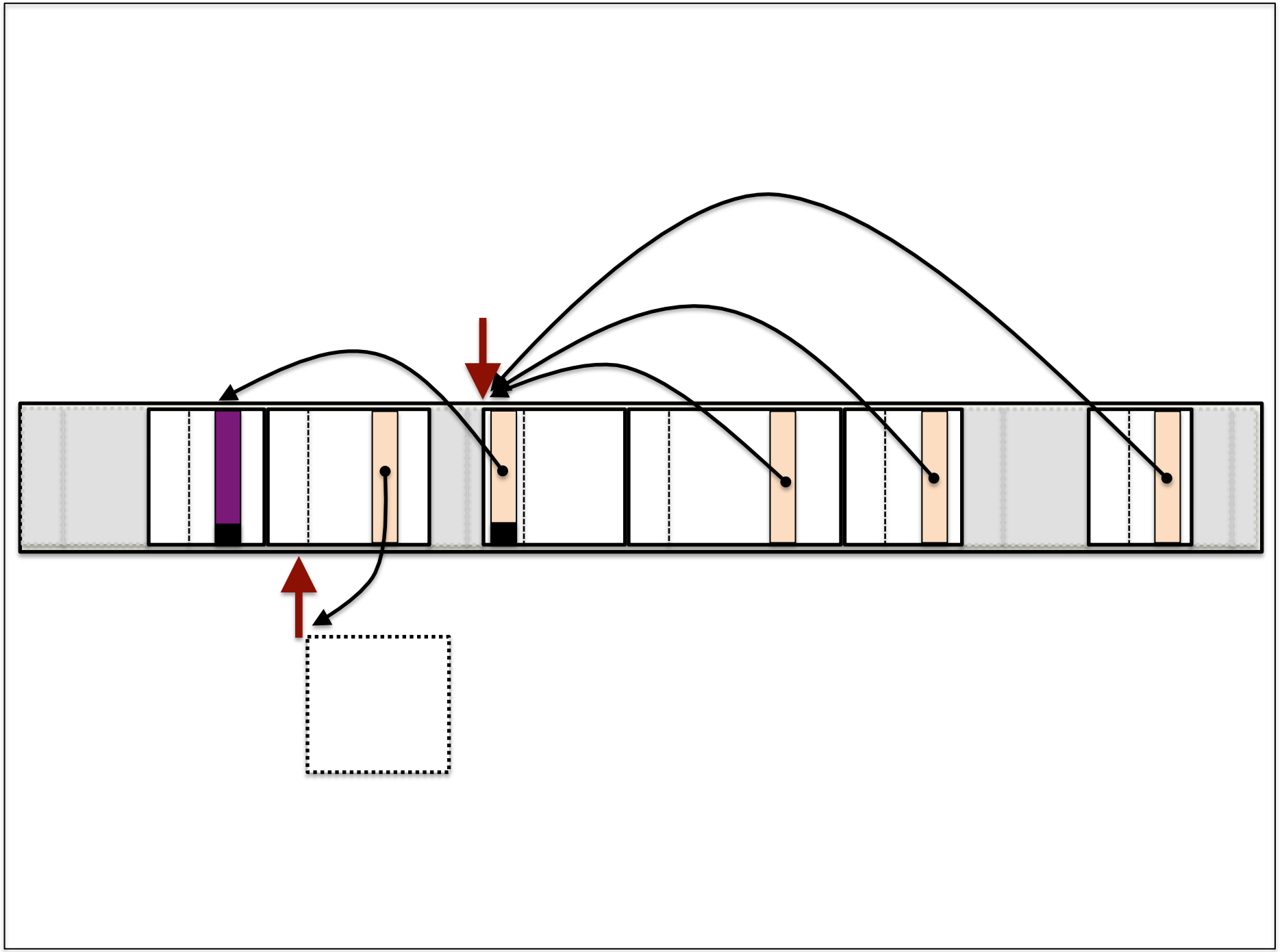


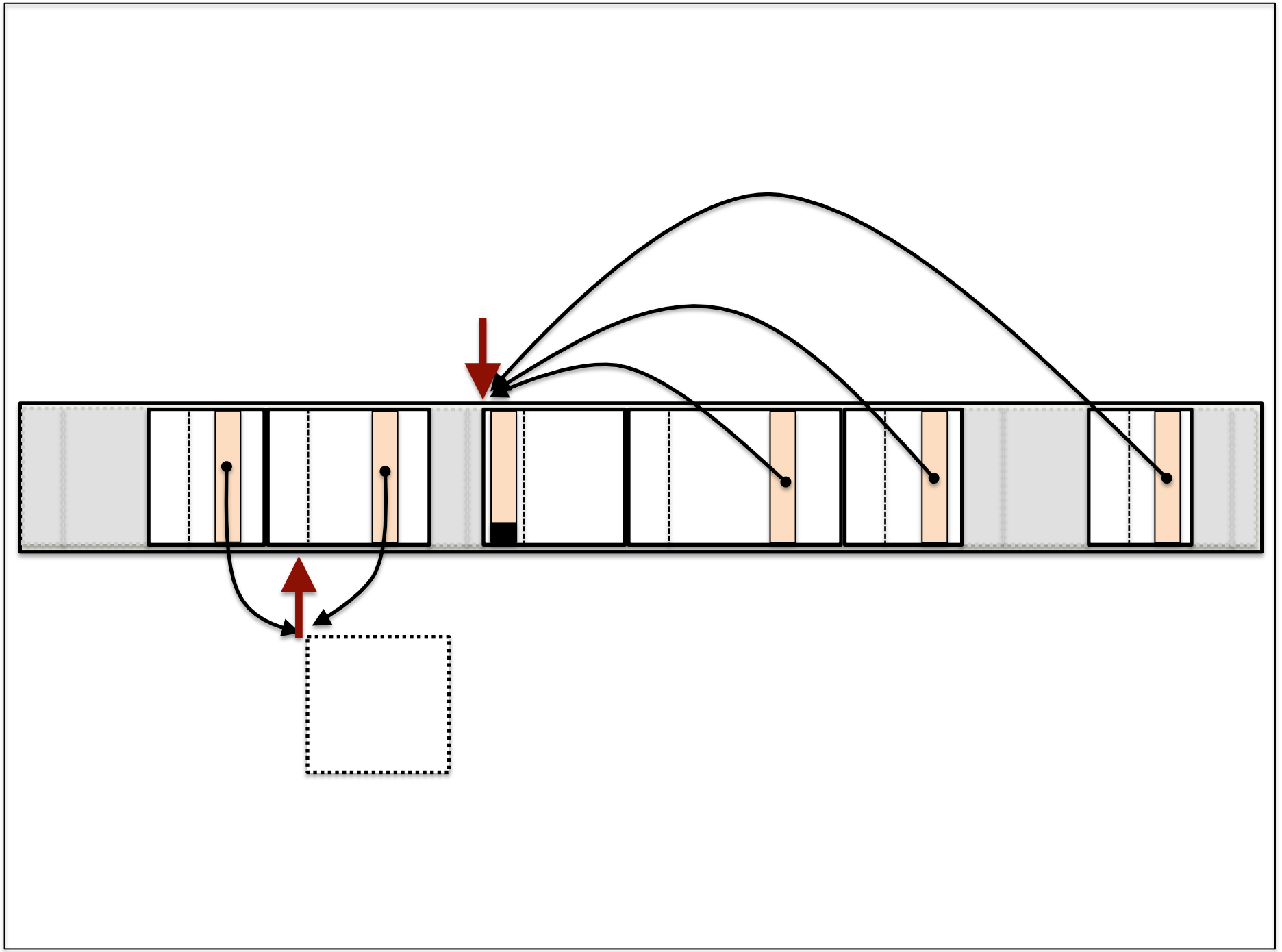


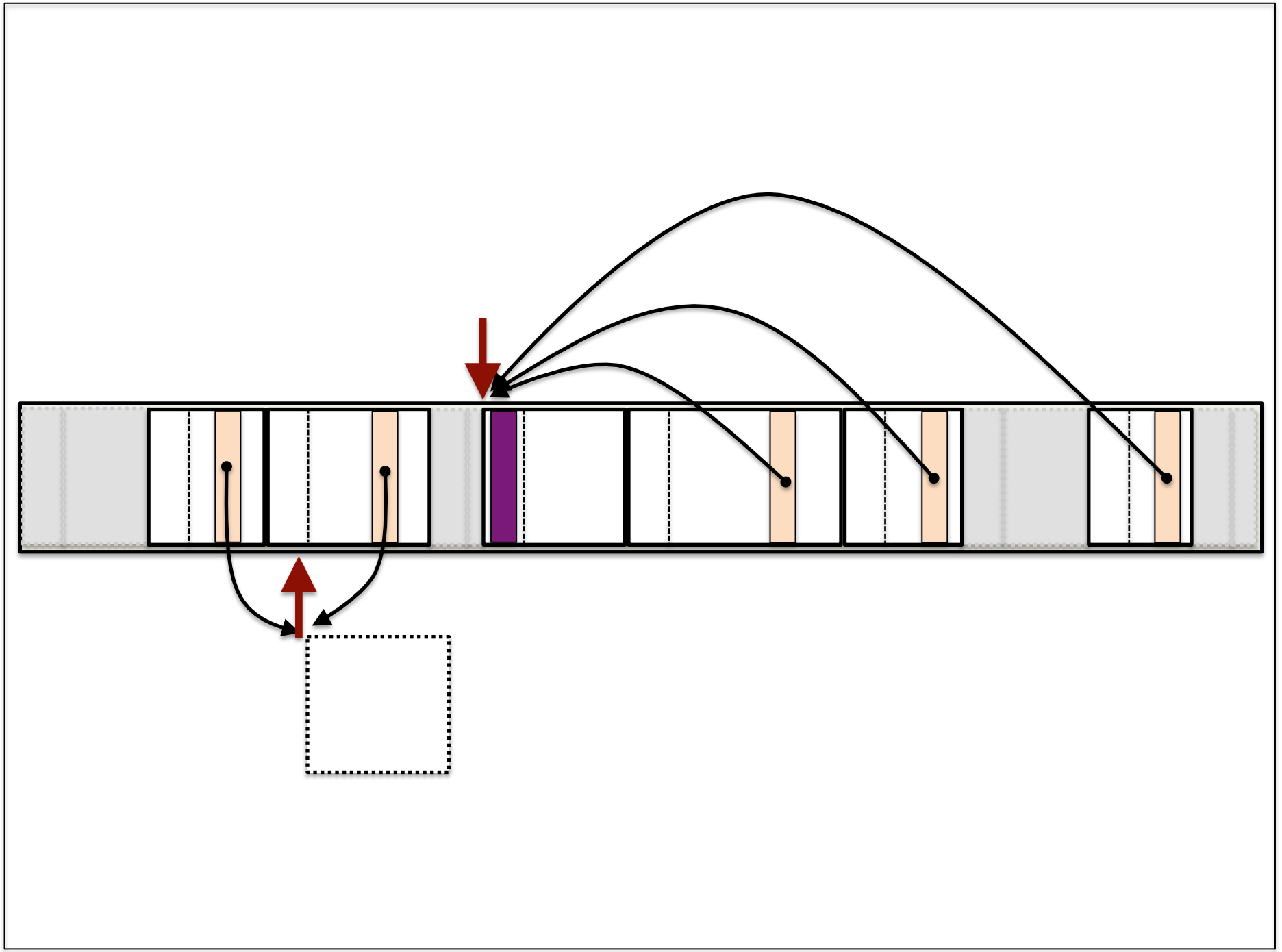
Compacted Offset

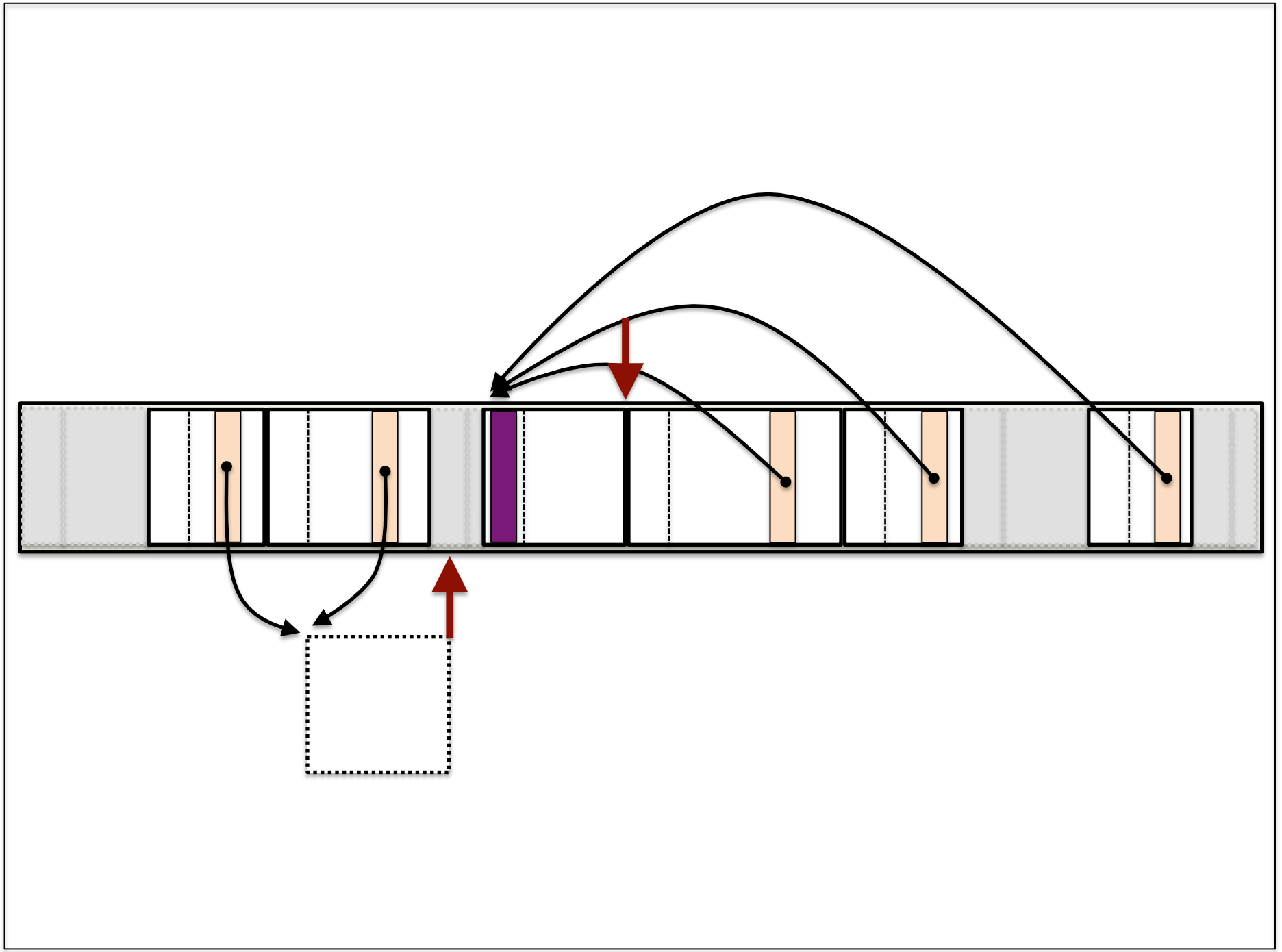
- We can compute where the object will go
  - We've scanned all objects lower in the heap
  - We can keep track of their sizes as we go
- Can't move the object just yet
  - Previous objects haven't been compacted
- Can't compact on first pass
  - There are still references that haven't been scanned

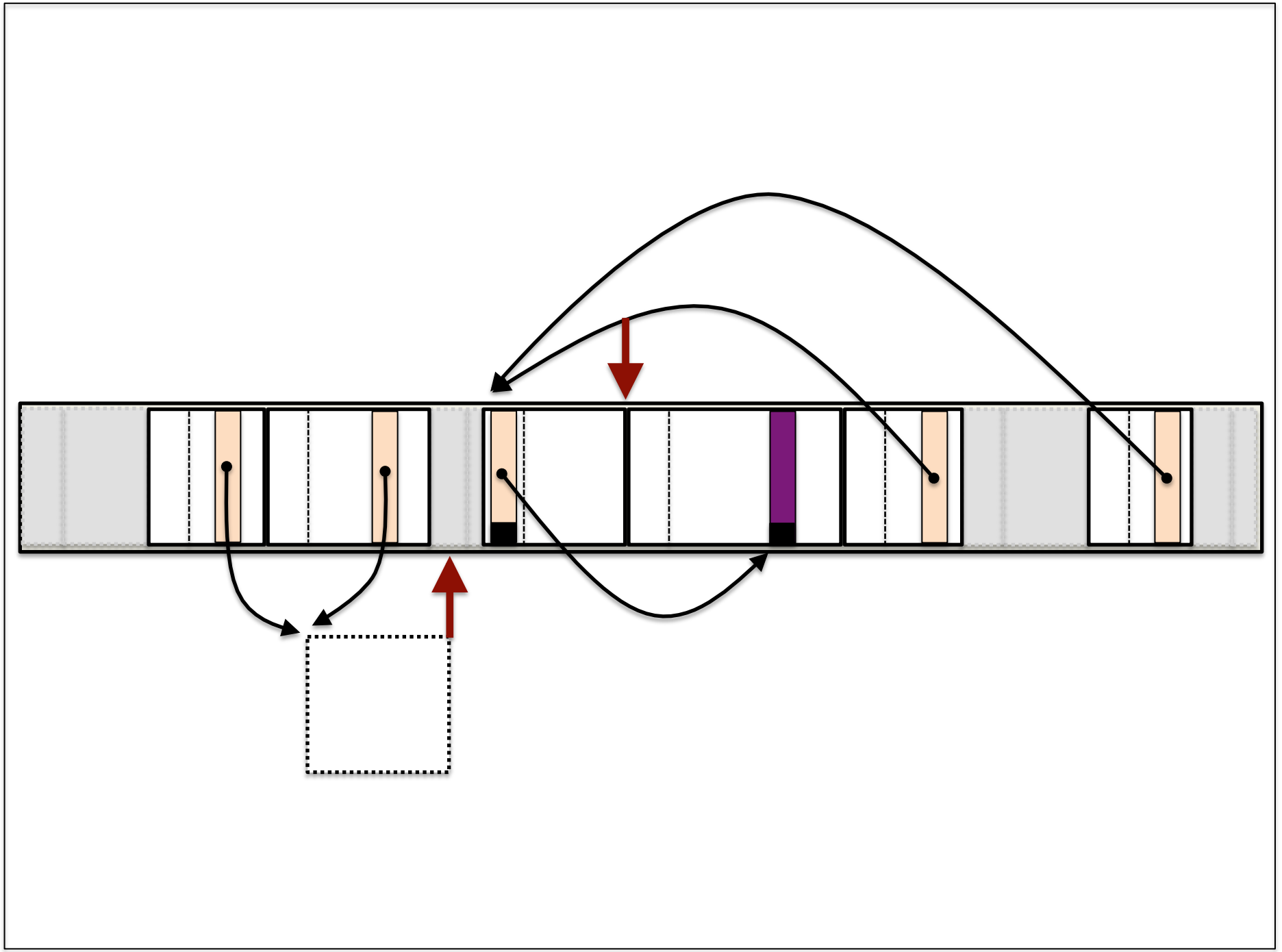


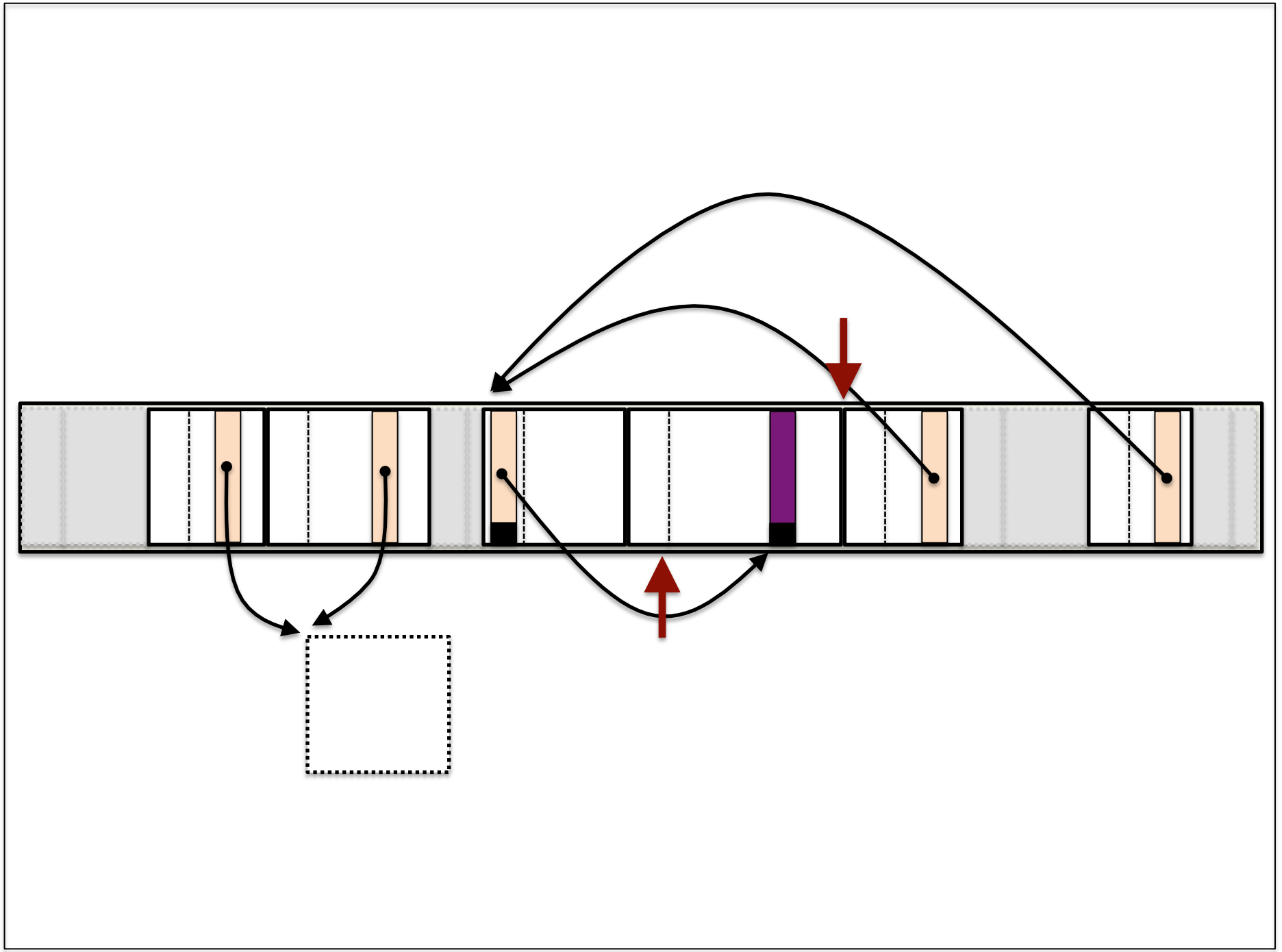


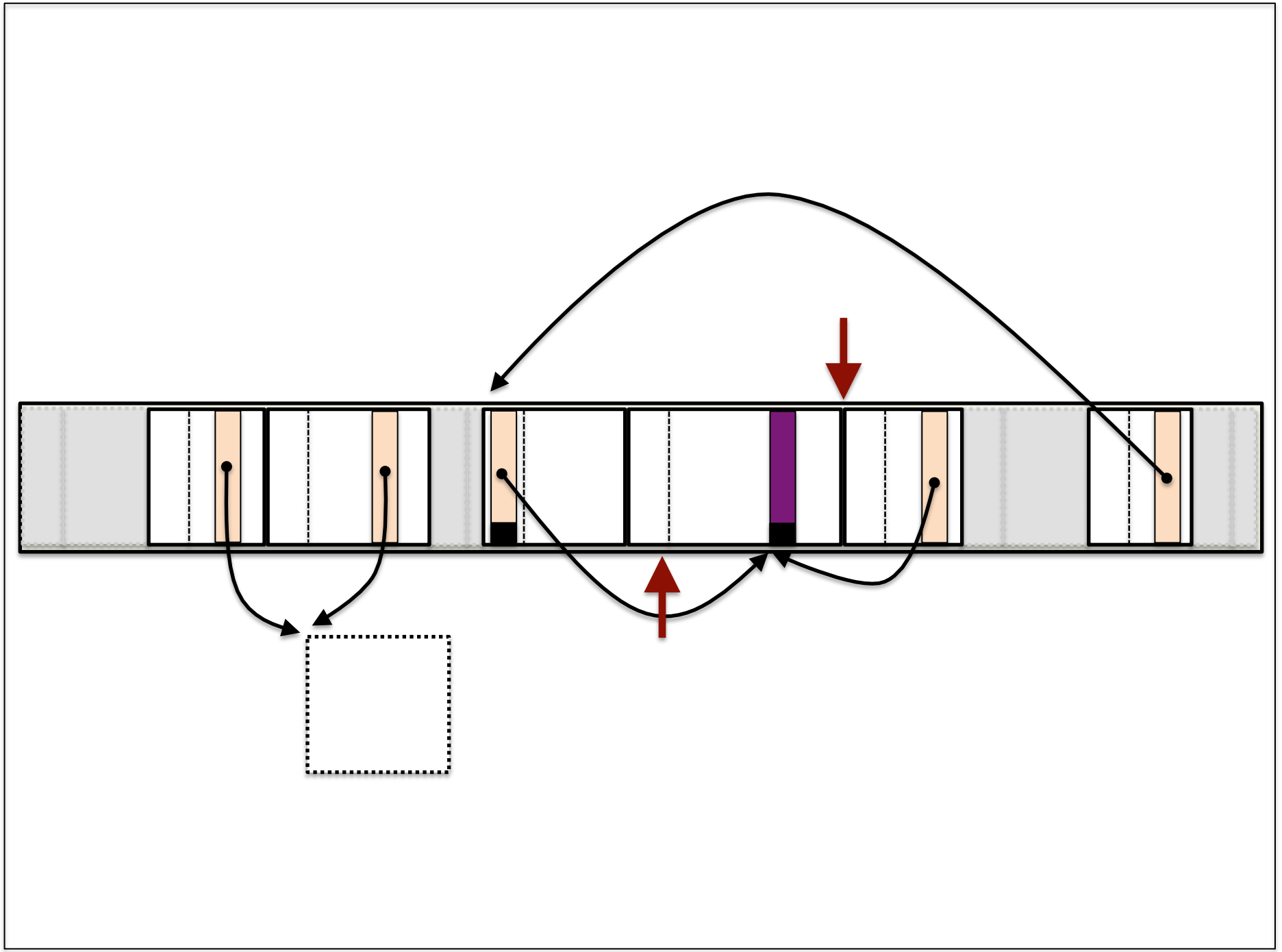


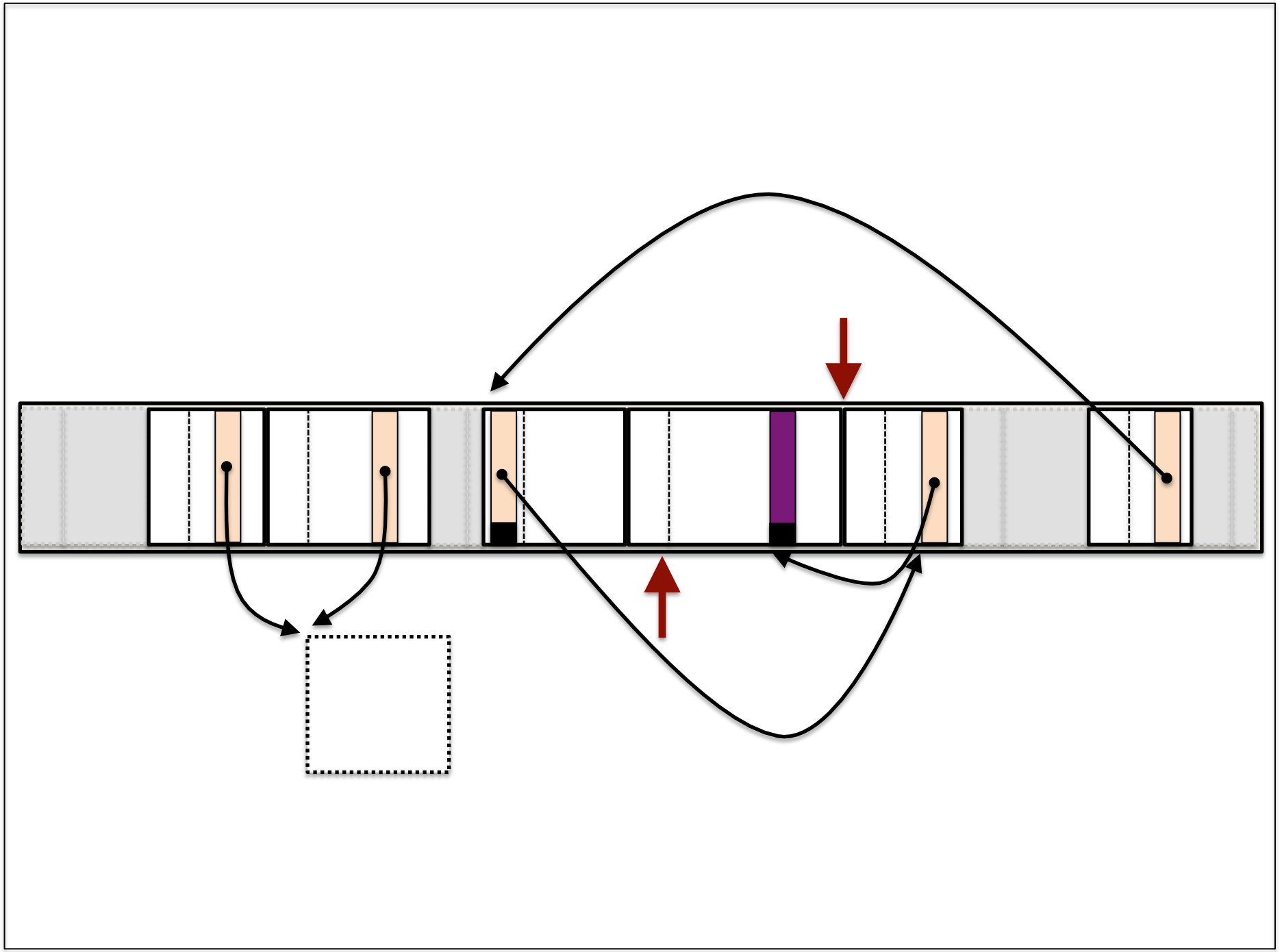


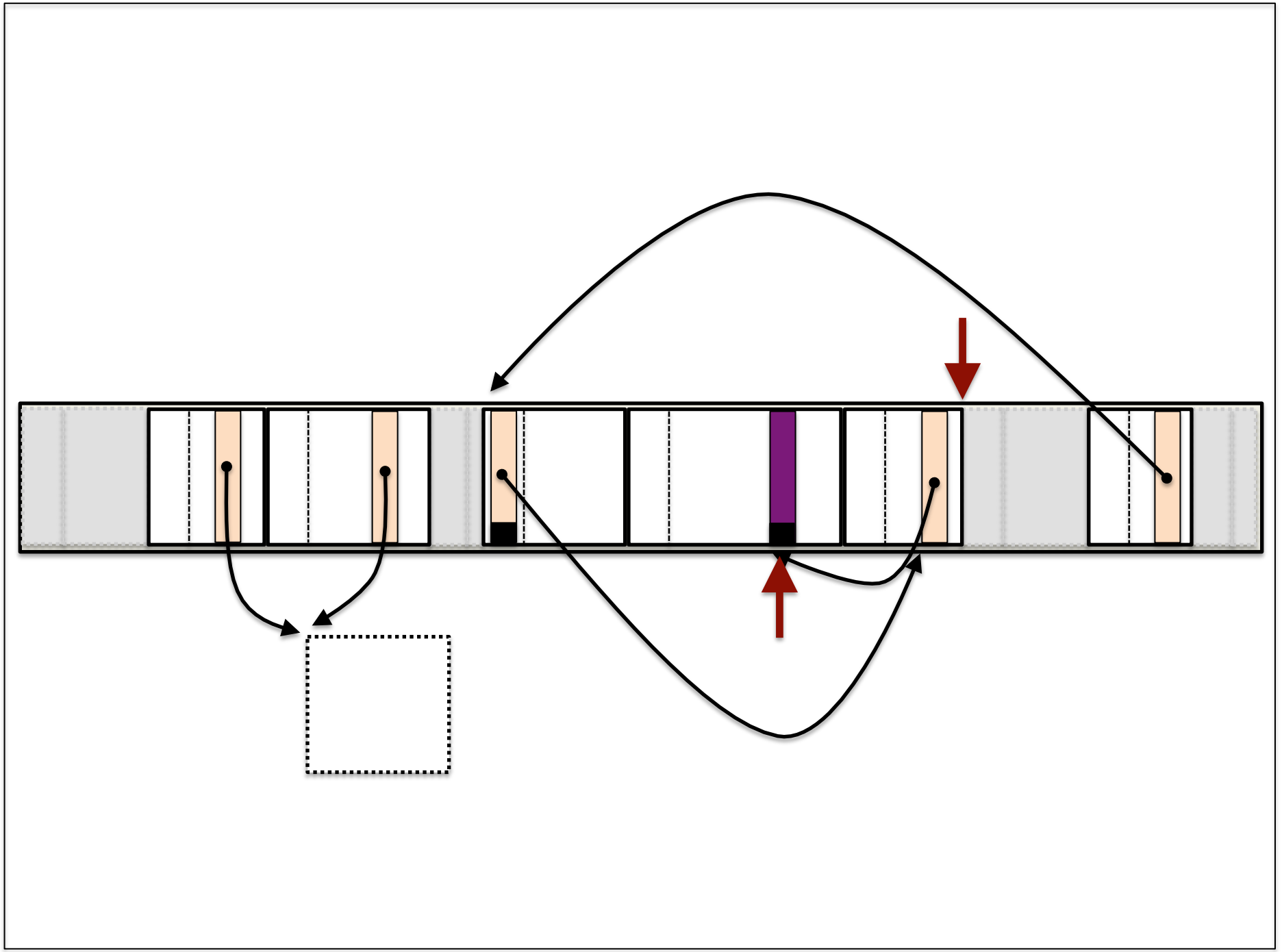


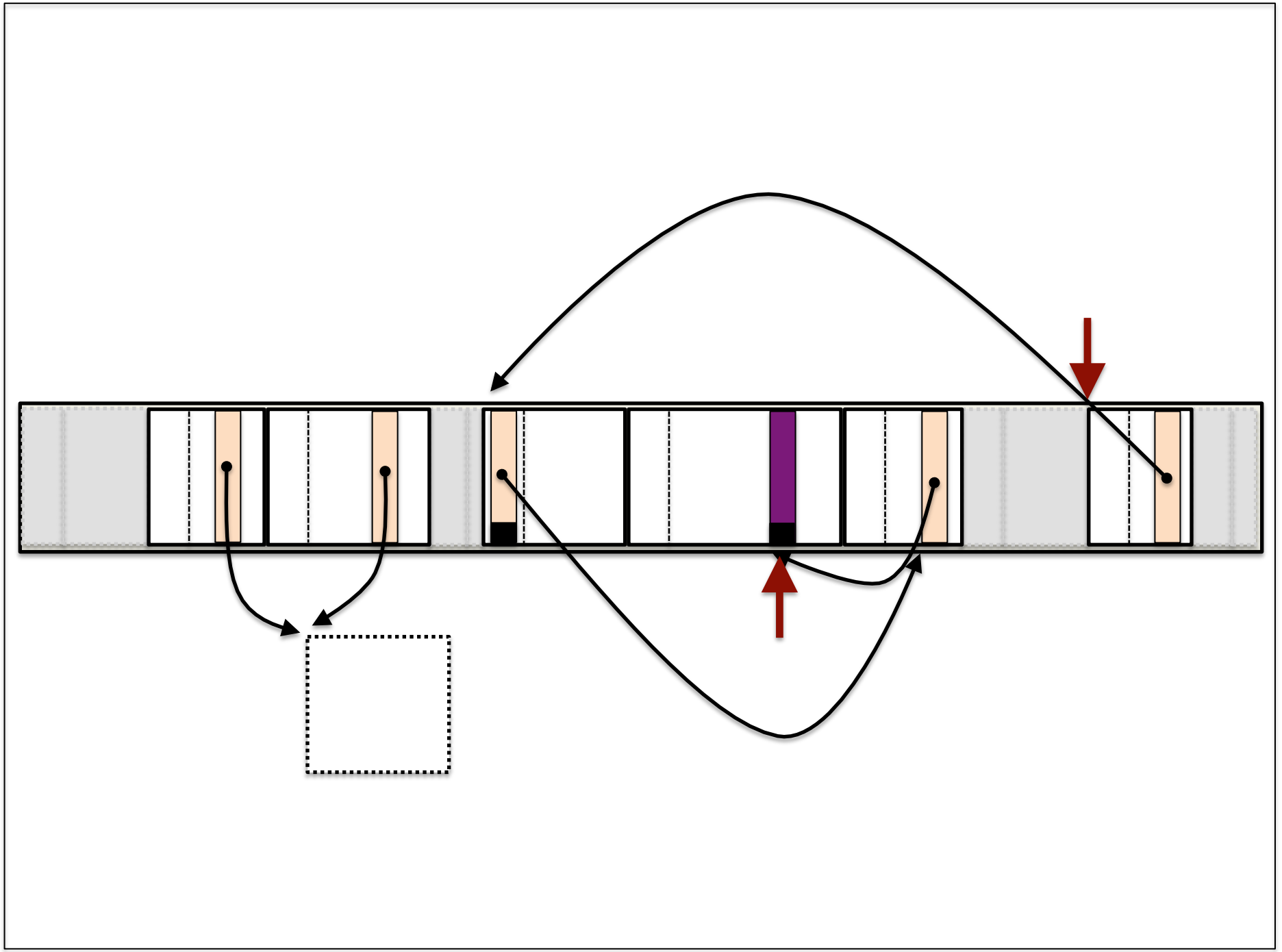


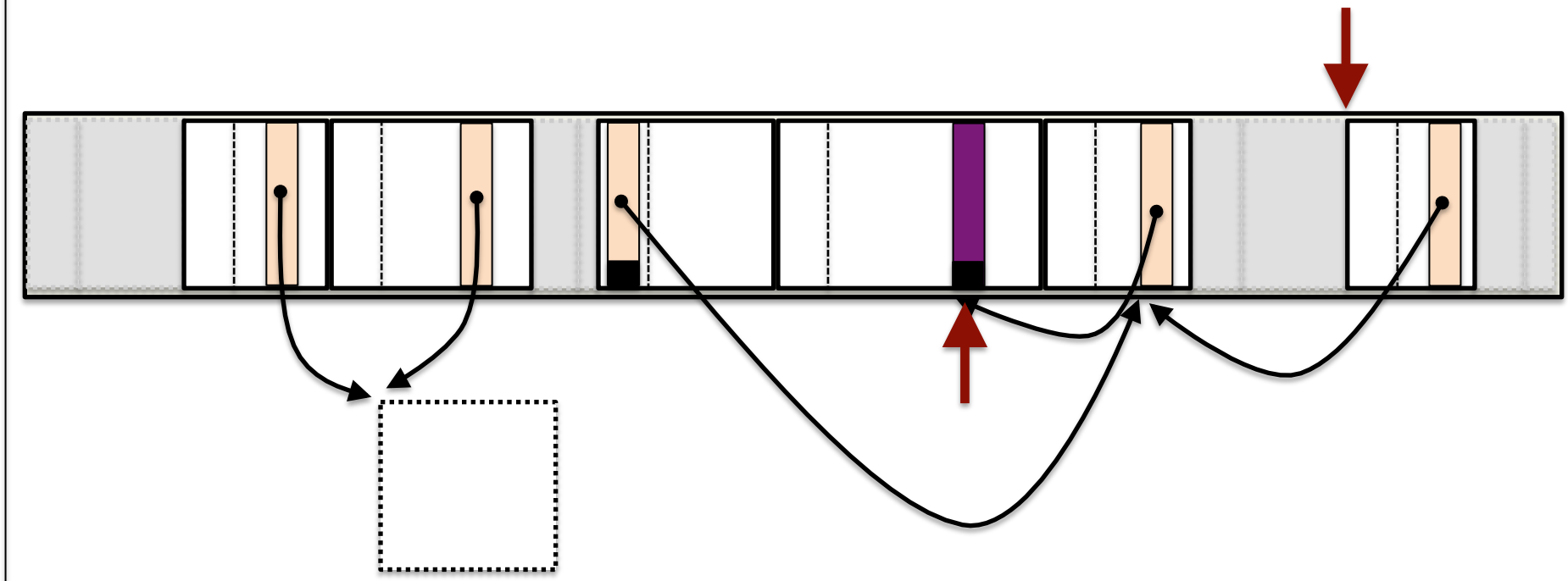


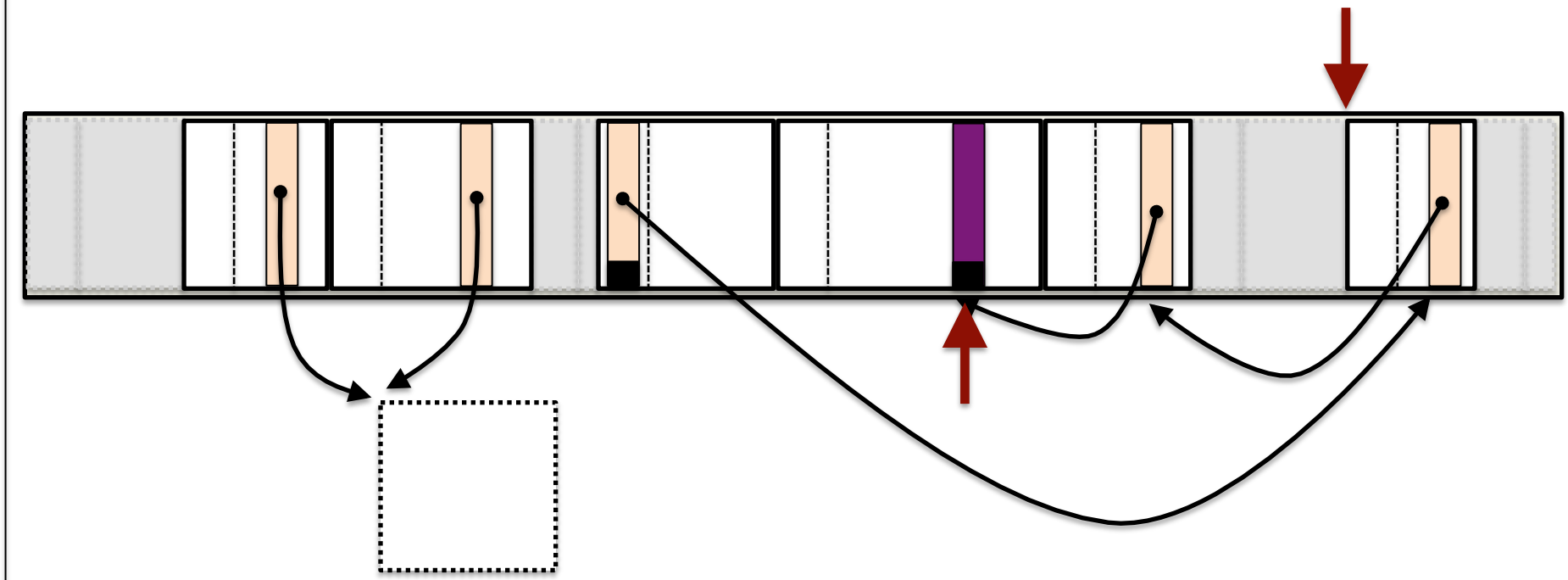


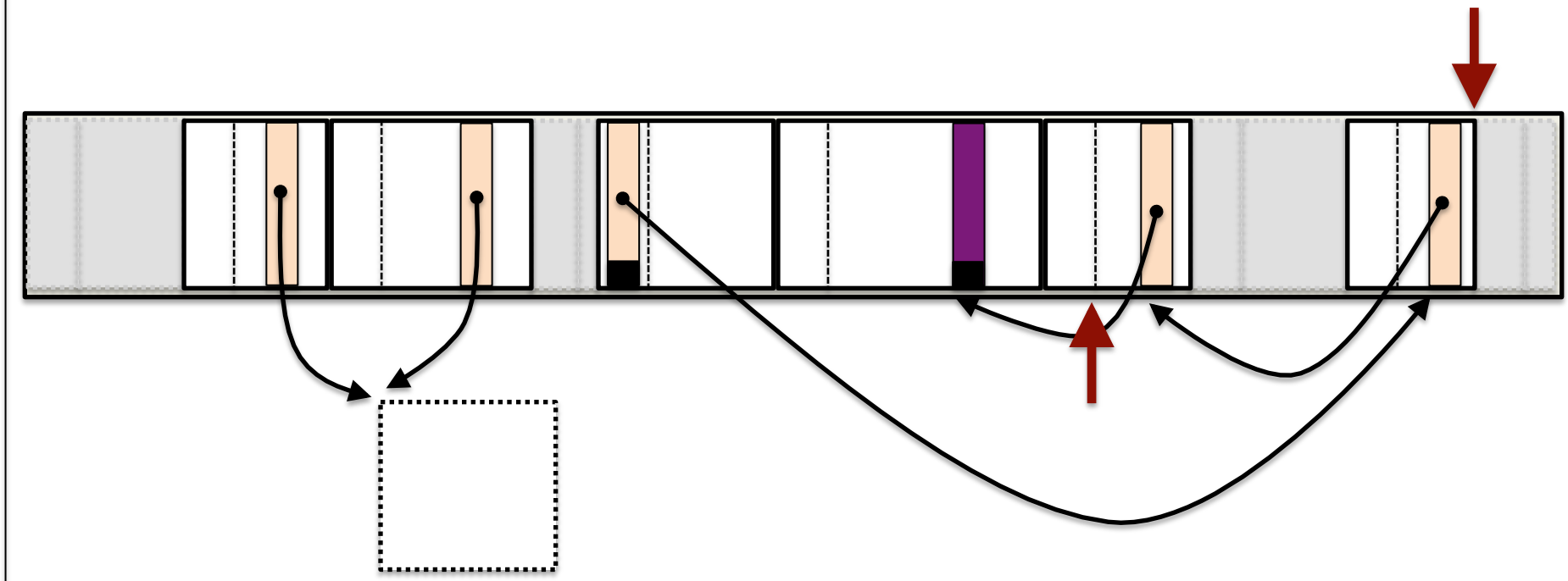


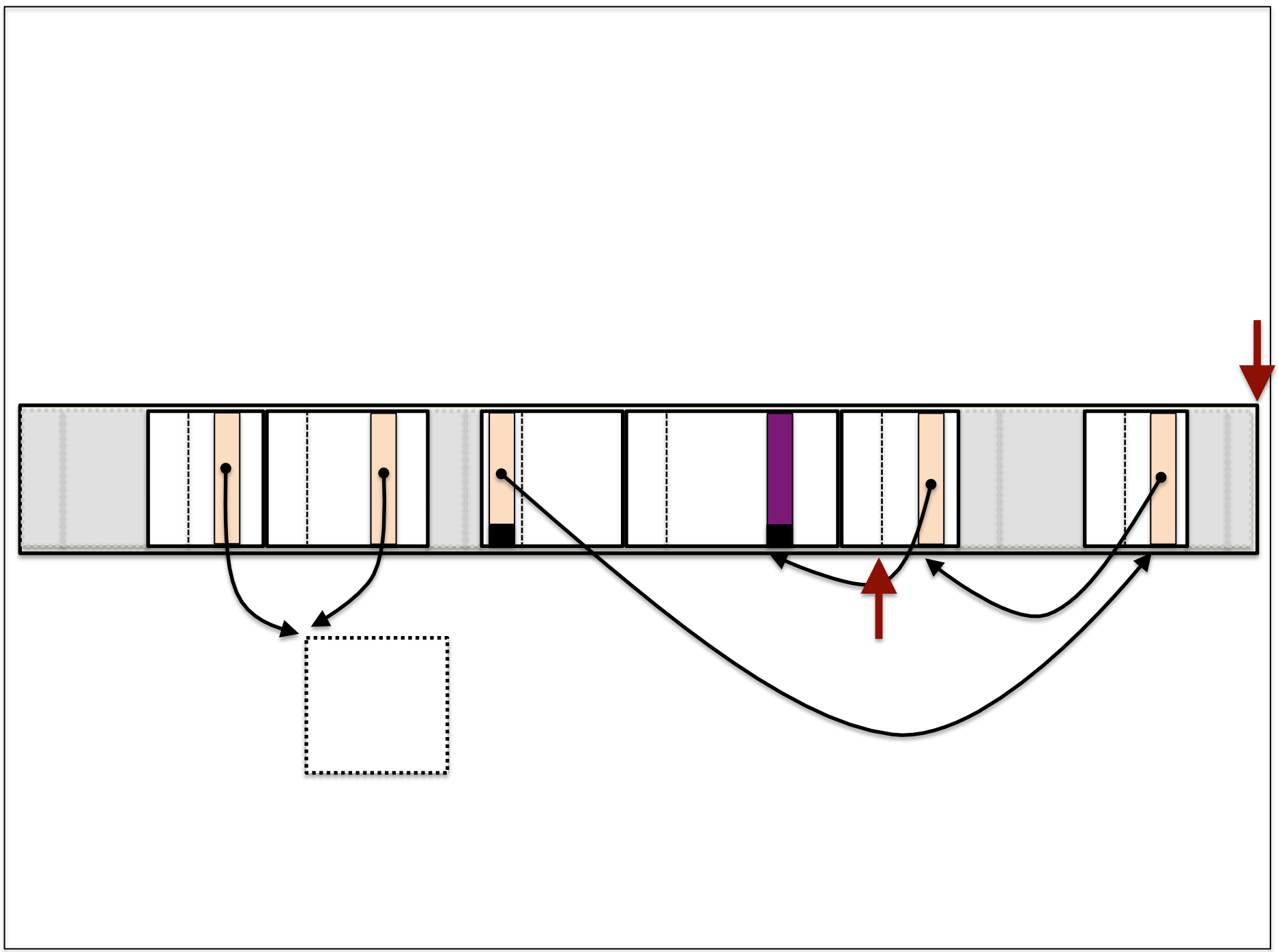






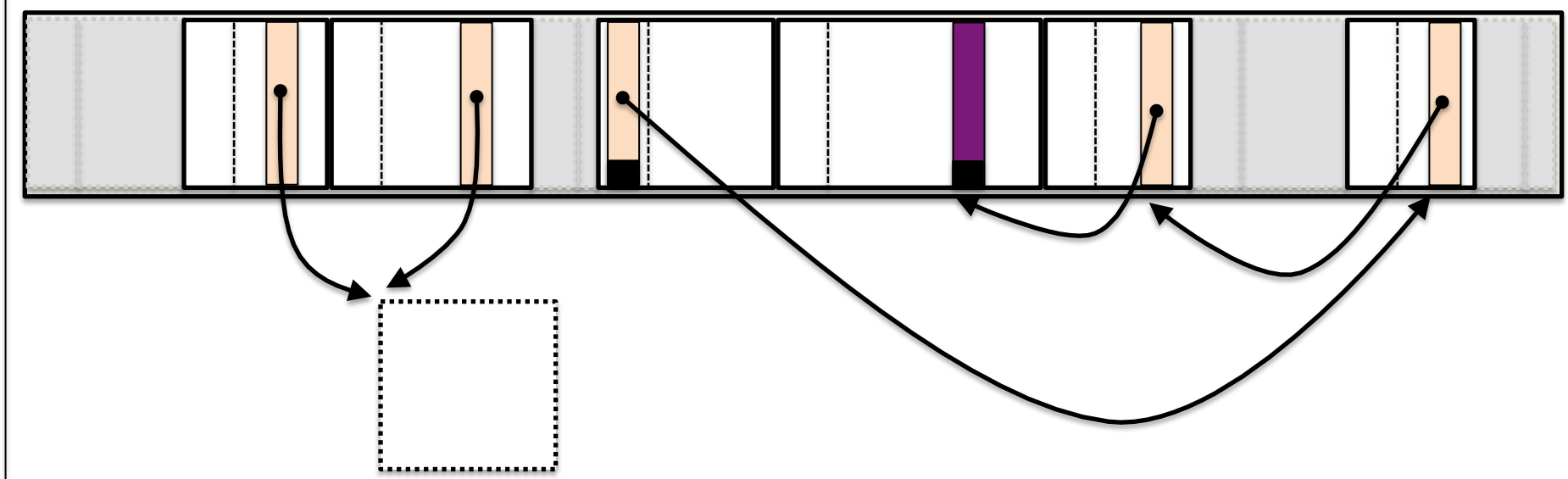


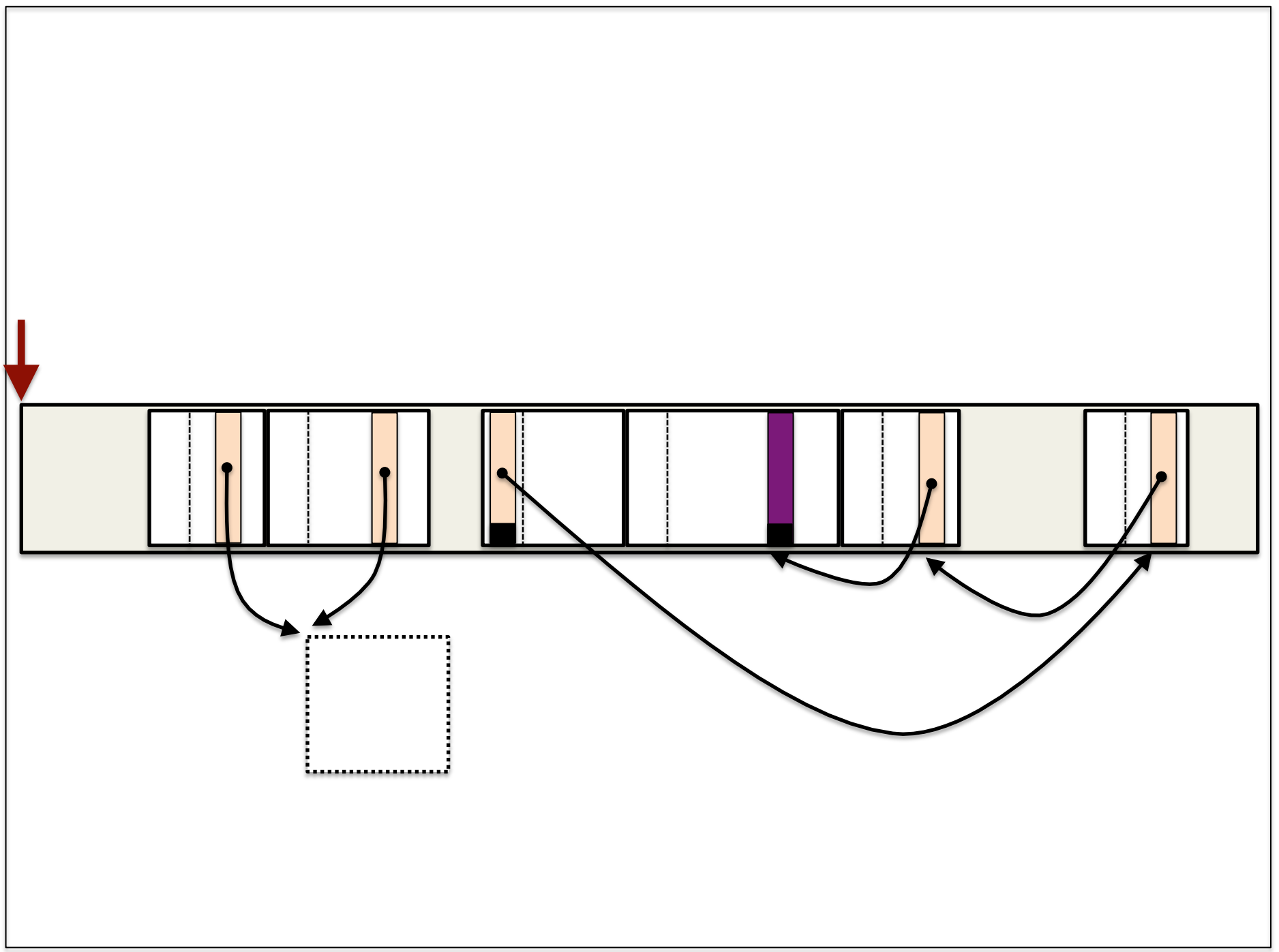


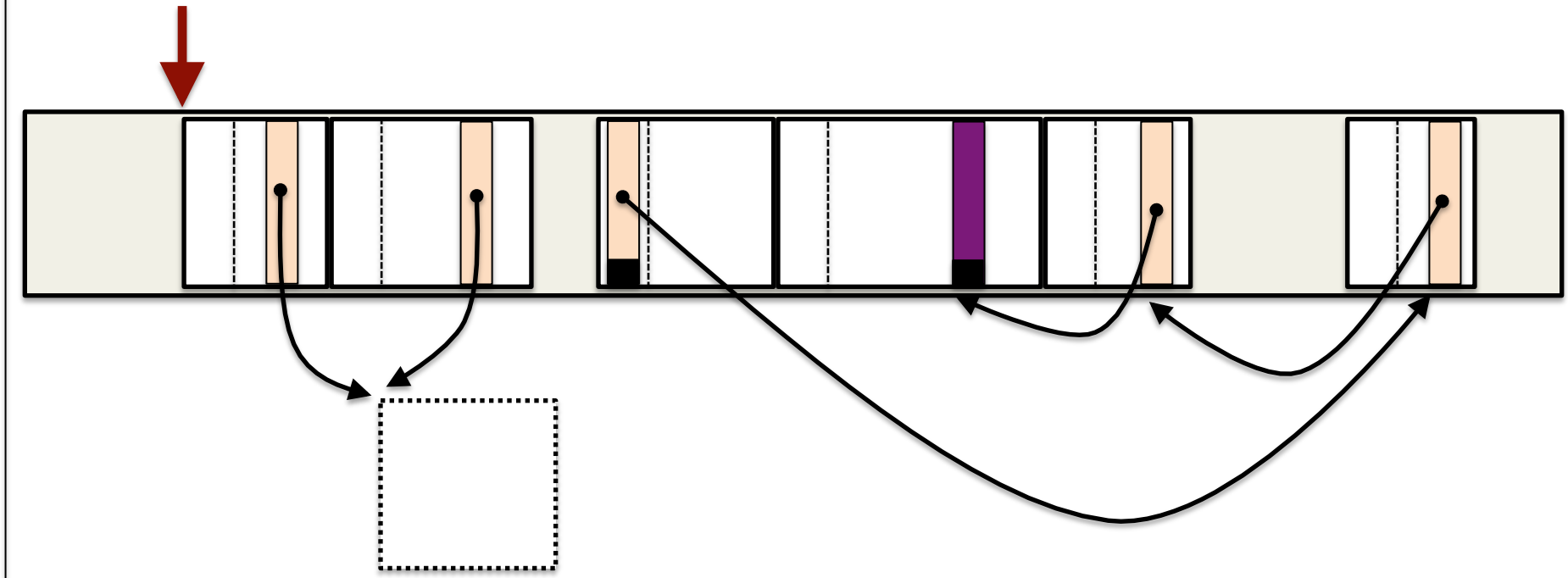


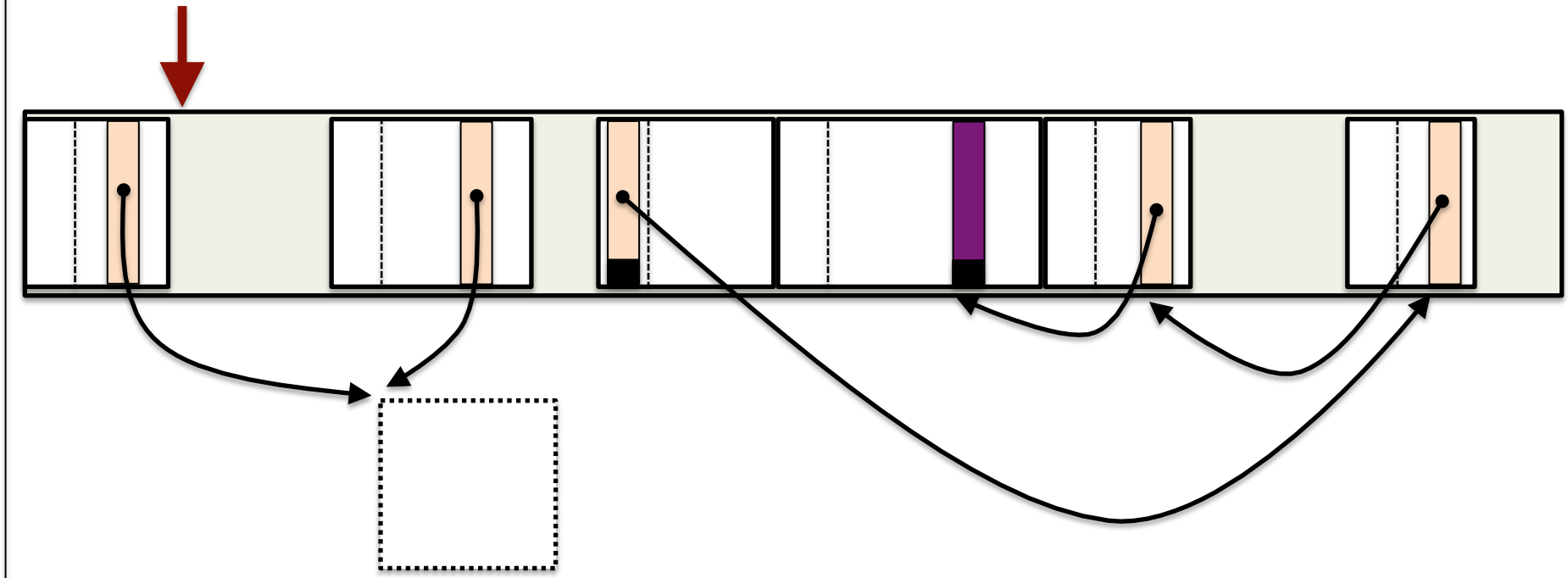
Second Pass

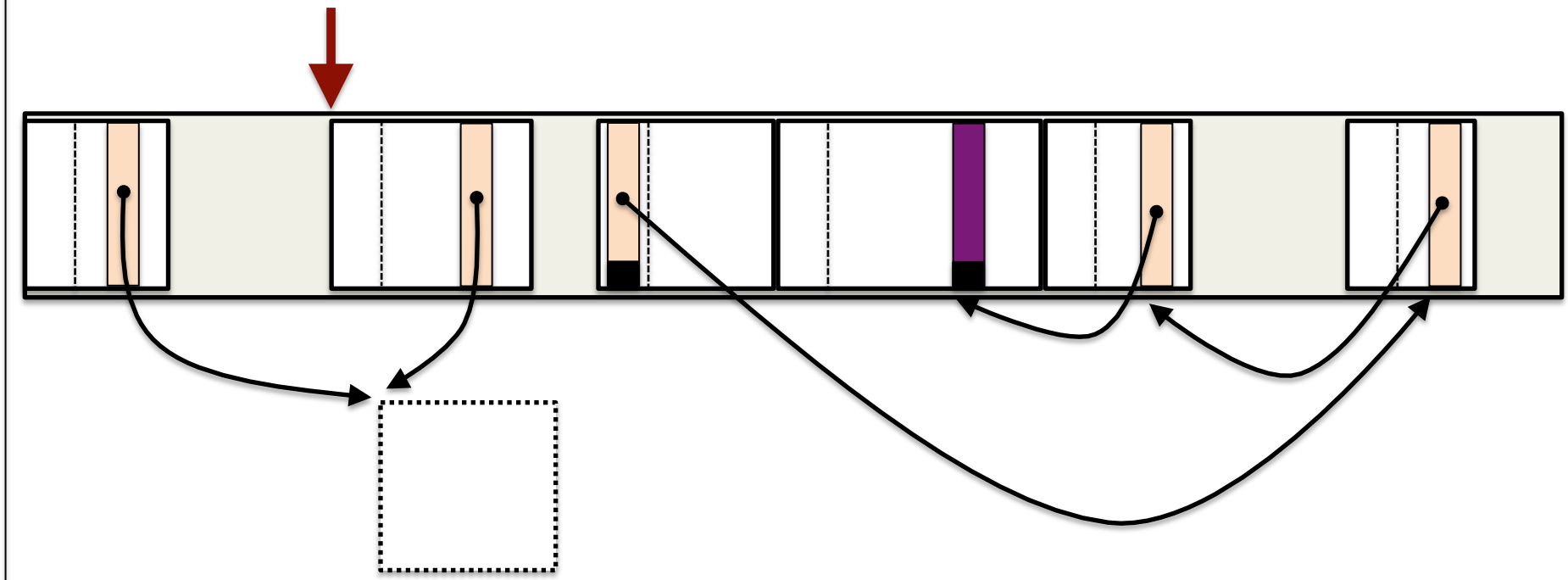
- After the first pass we know two things
  - All forward references point to new address
  - All backward references are chained
- Second pass completes the collection
  - Moves objects to their new locations
  - Updates chained backward references

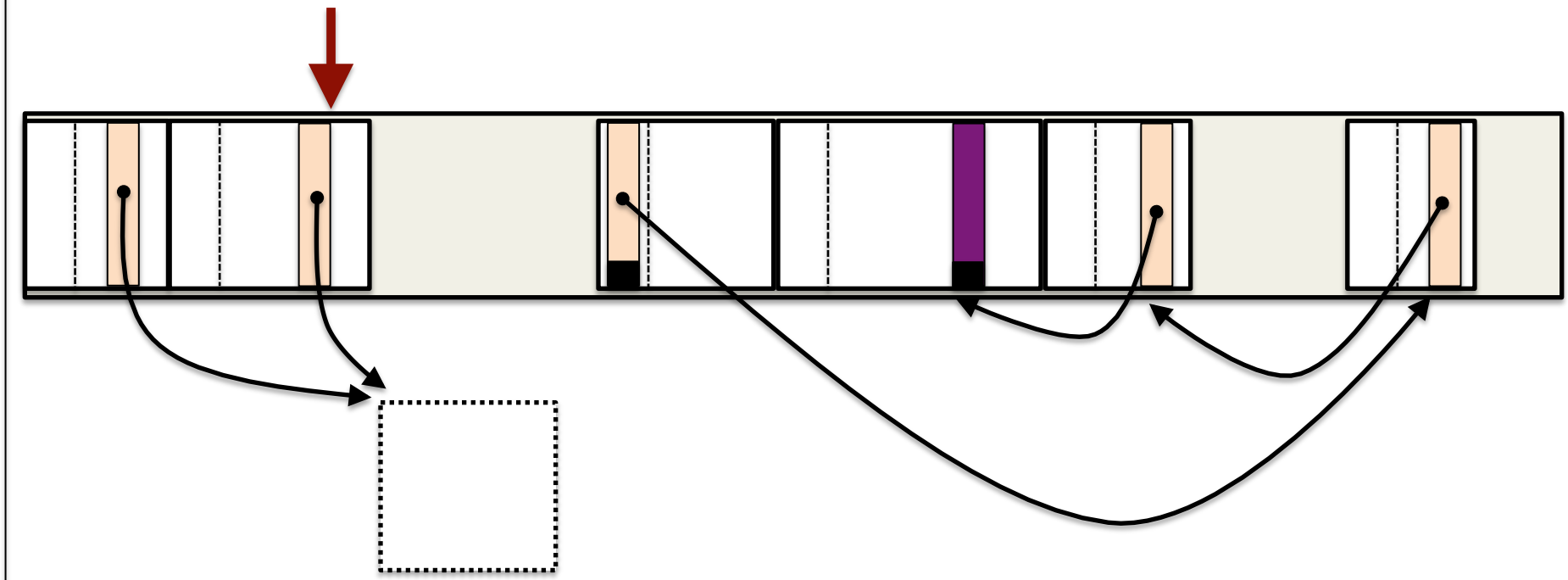


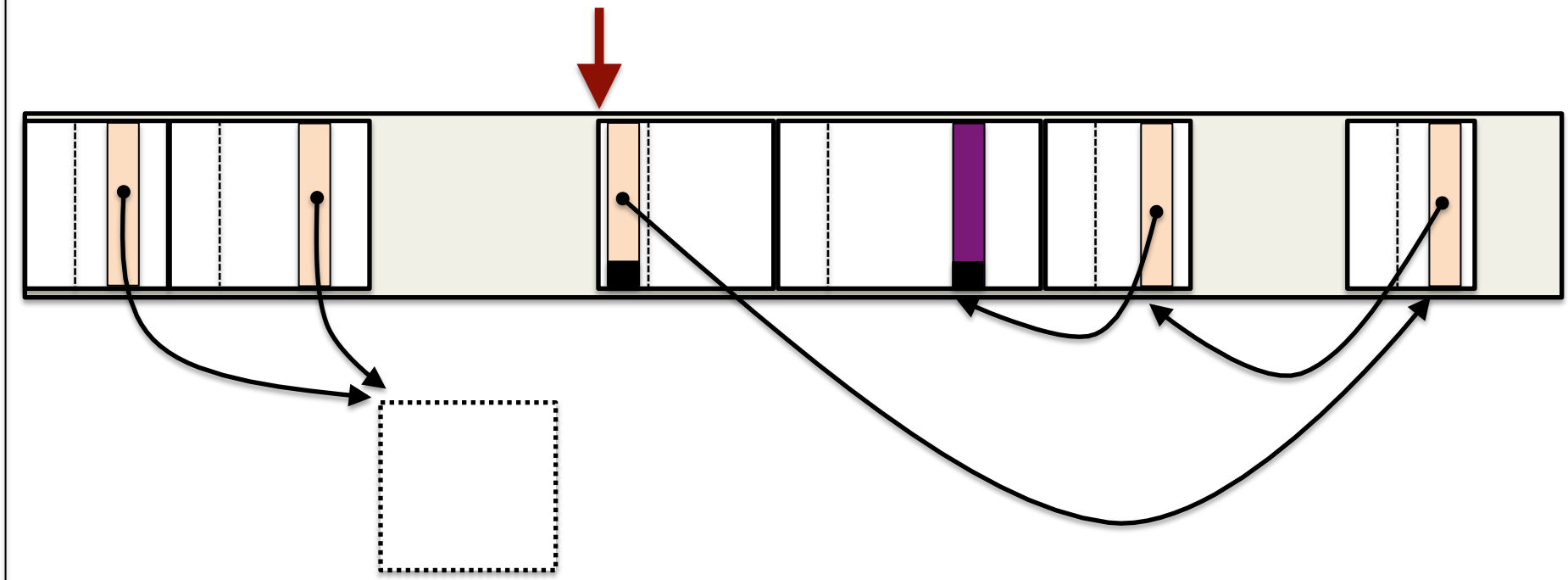


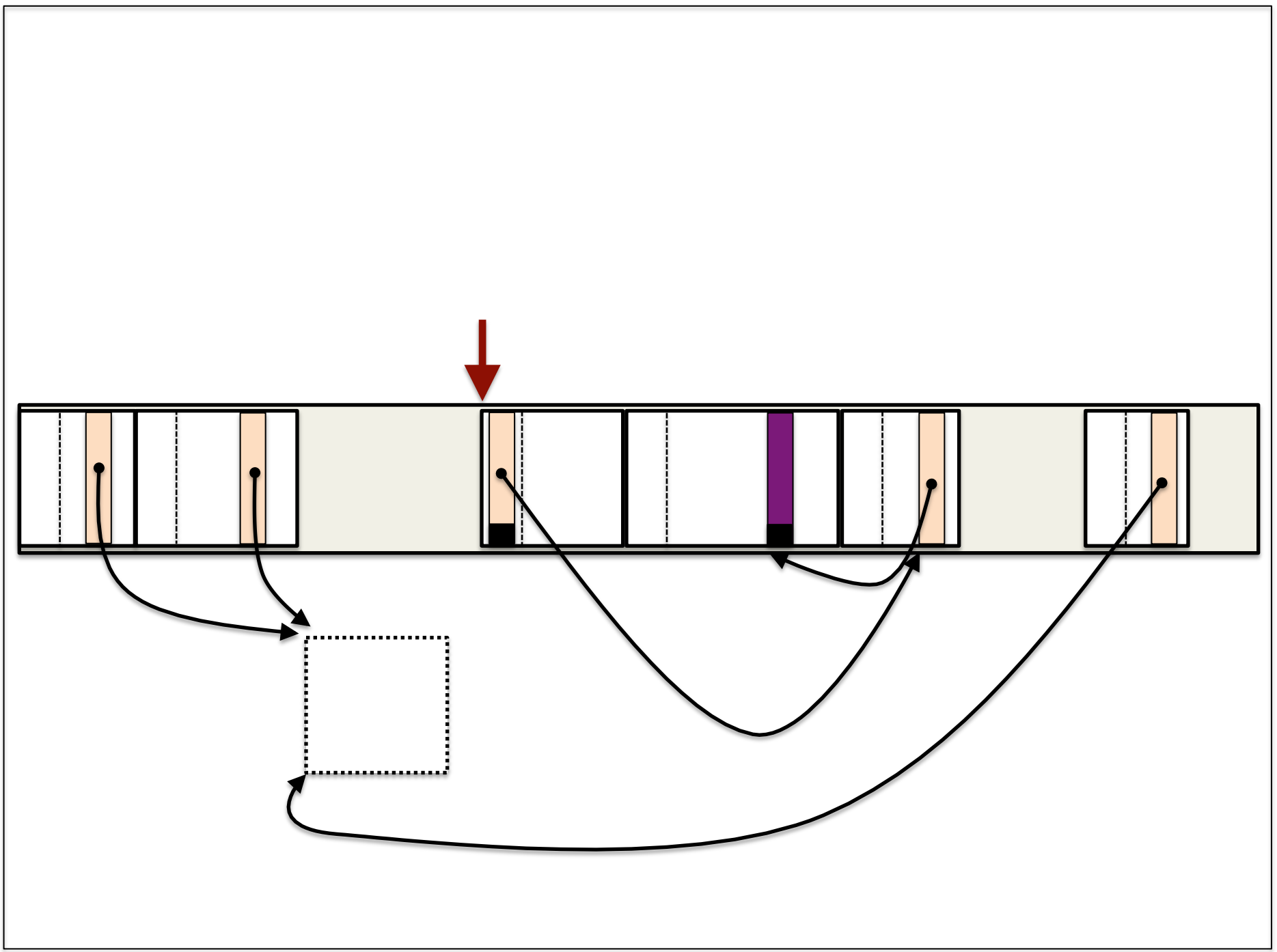


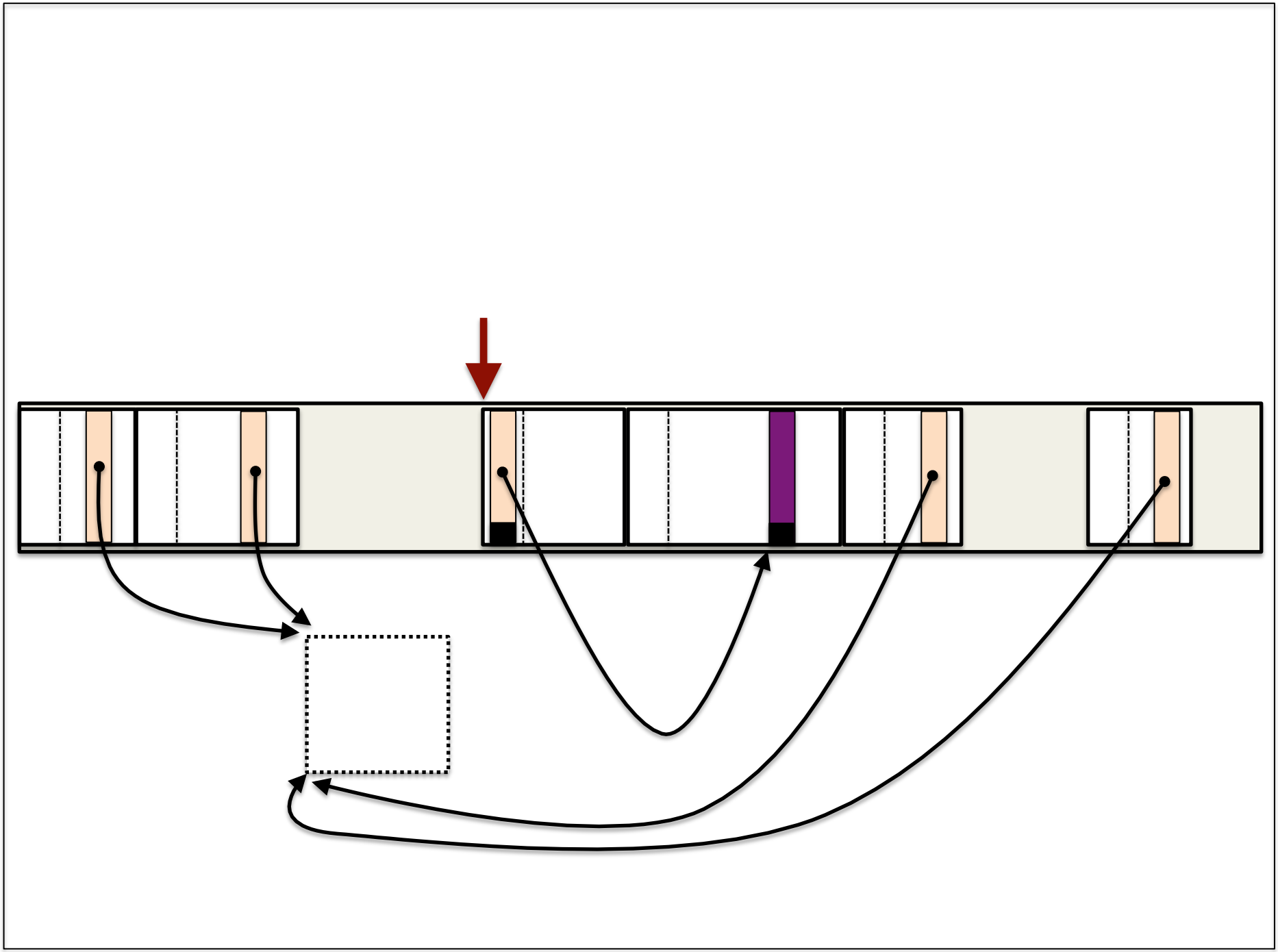


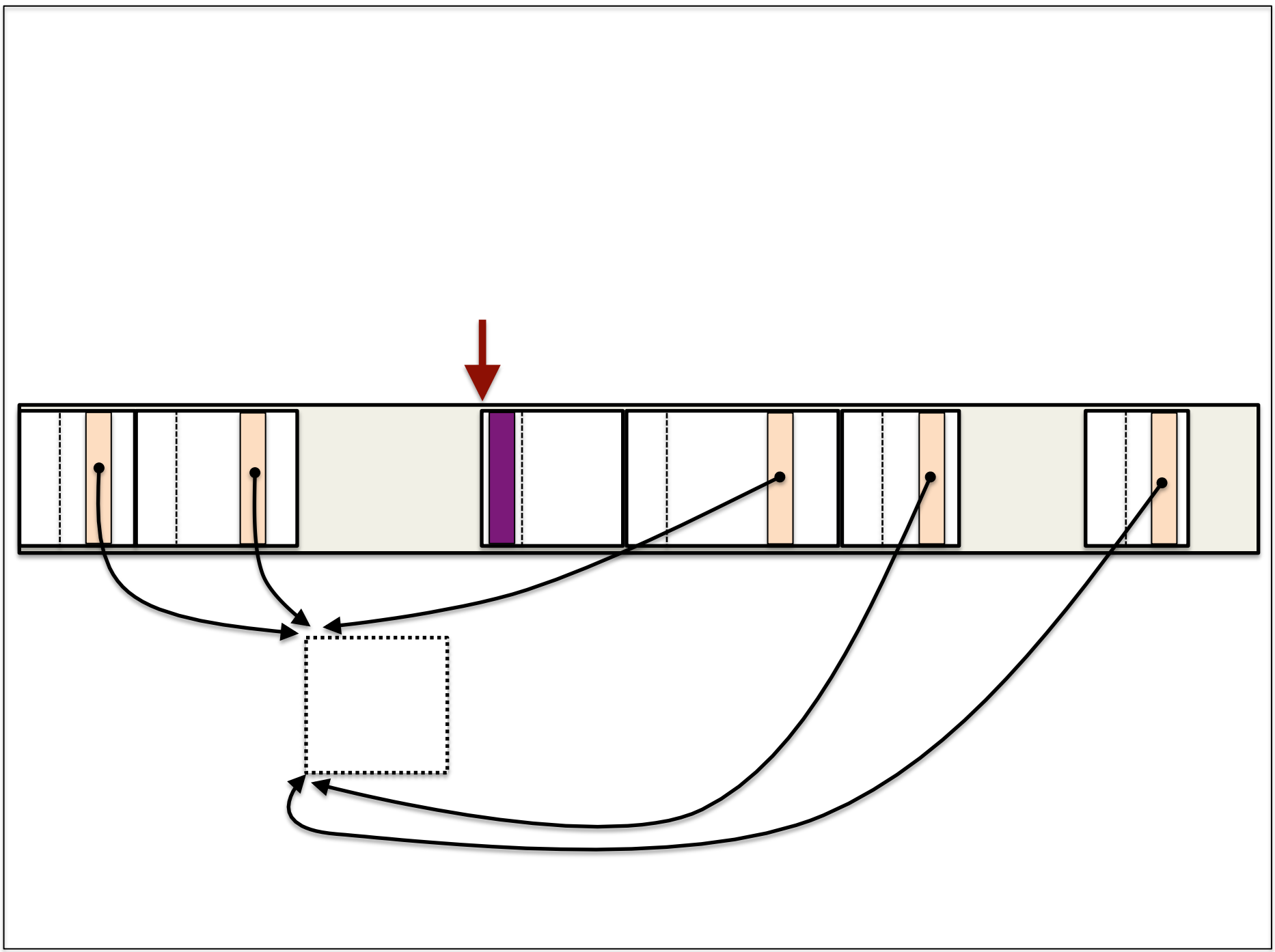


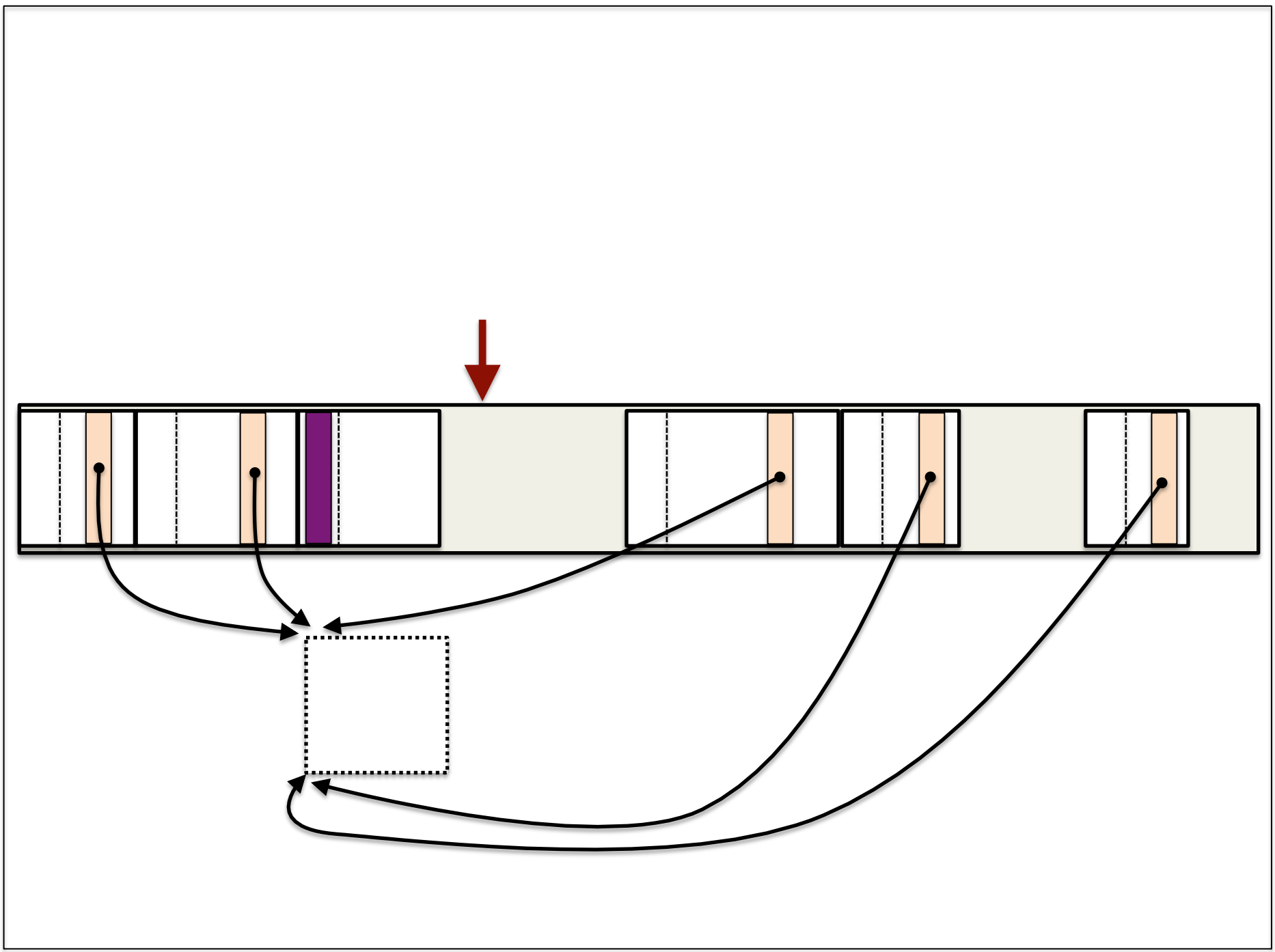


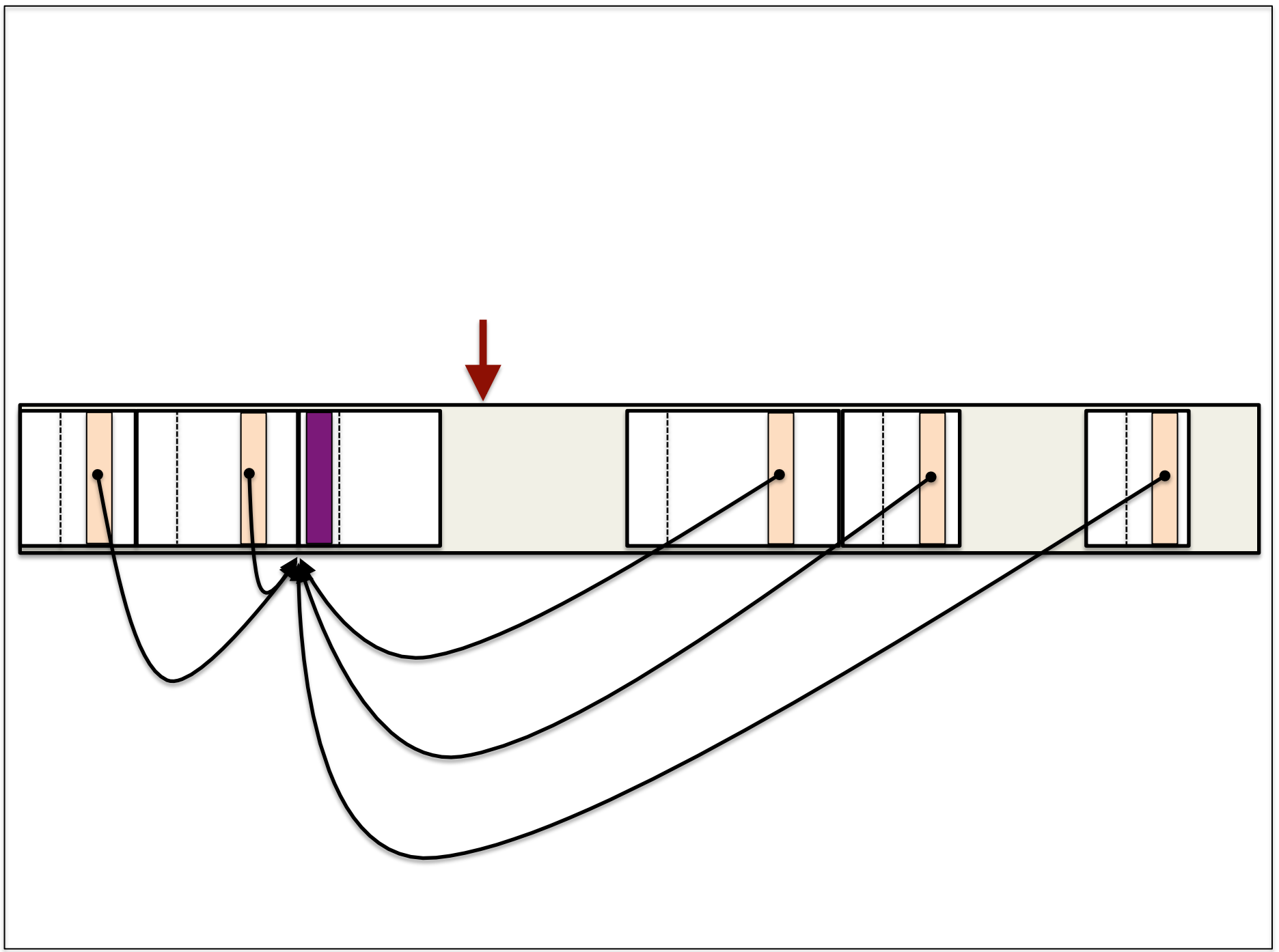


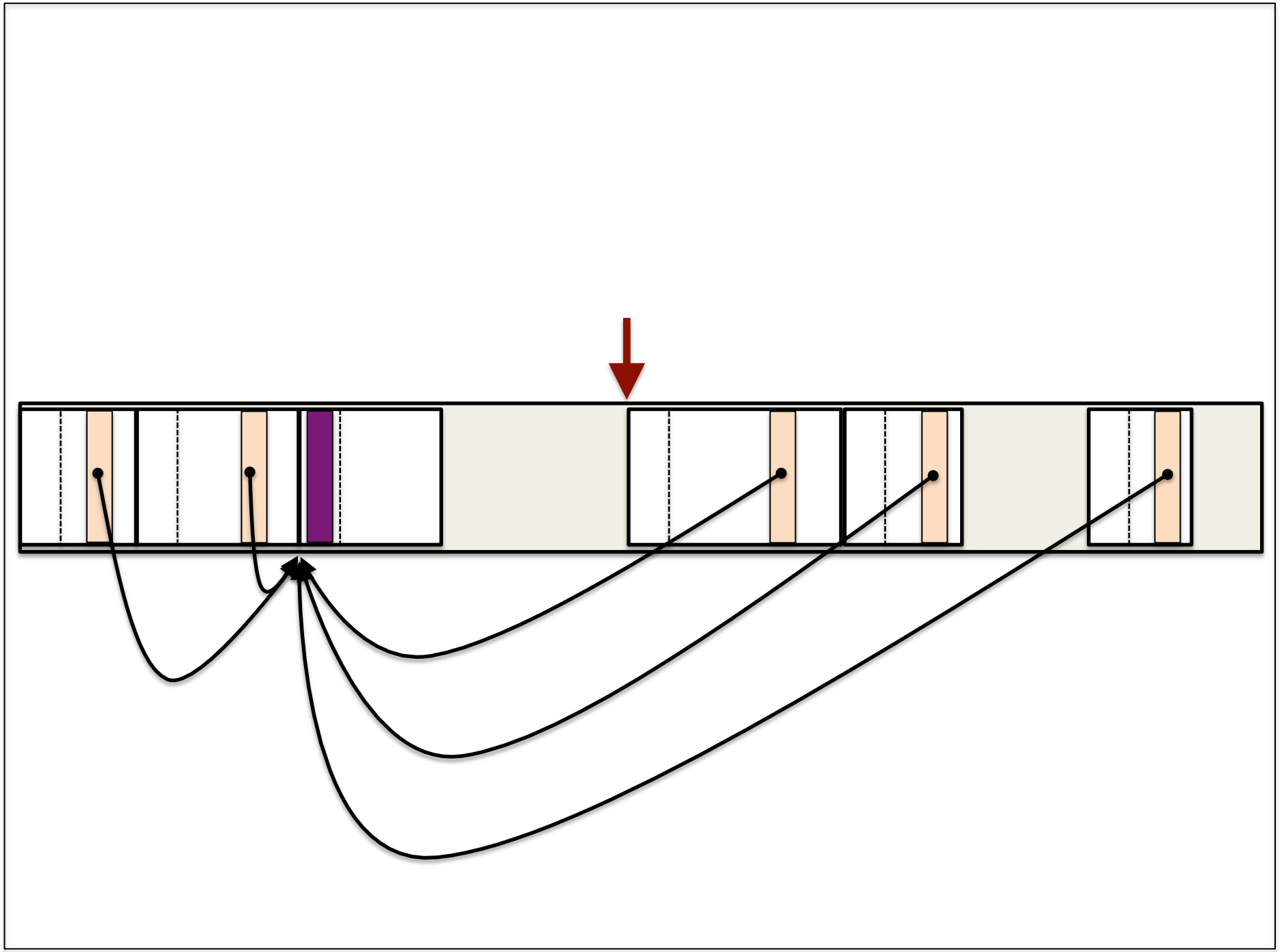


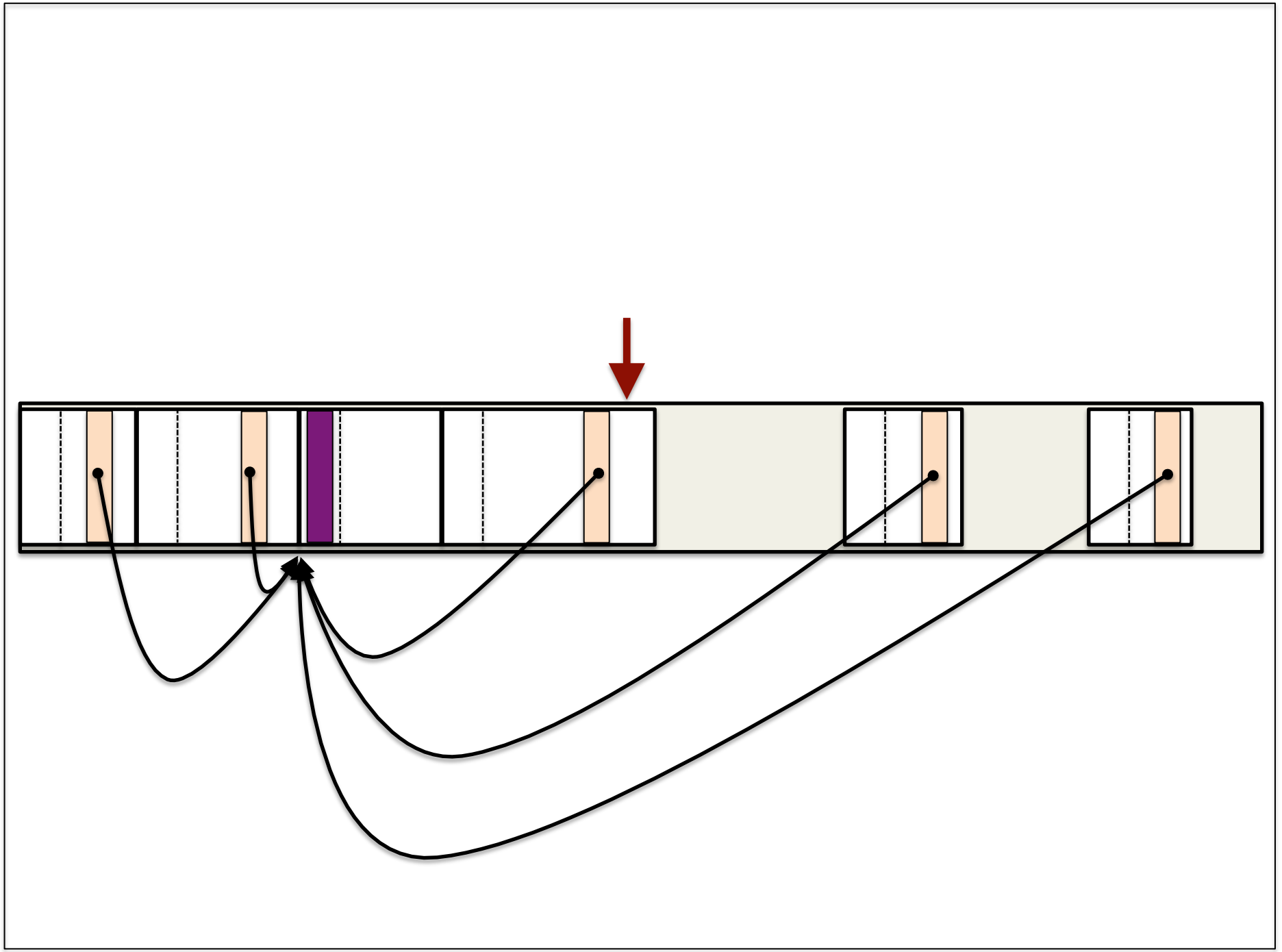


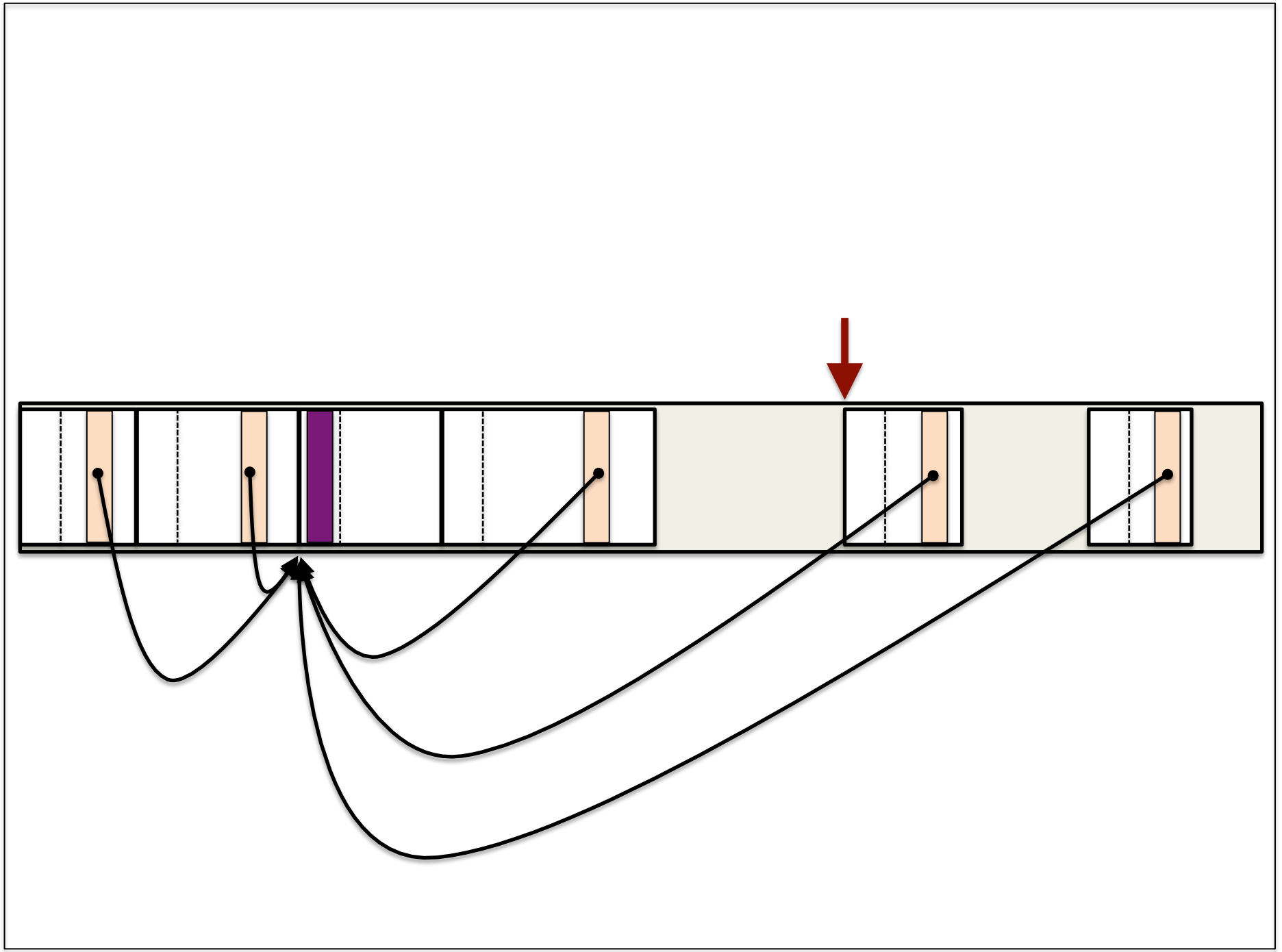


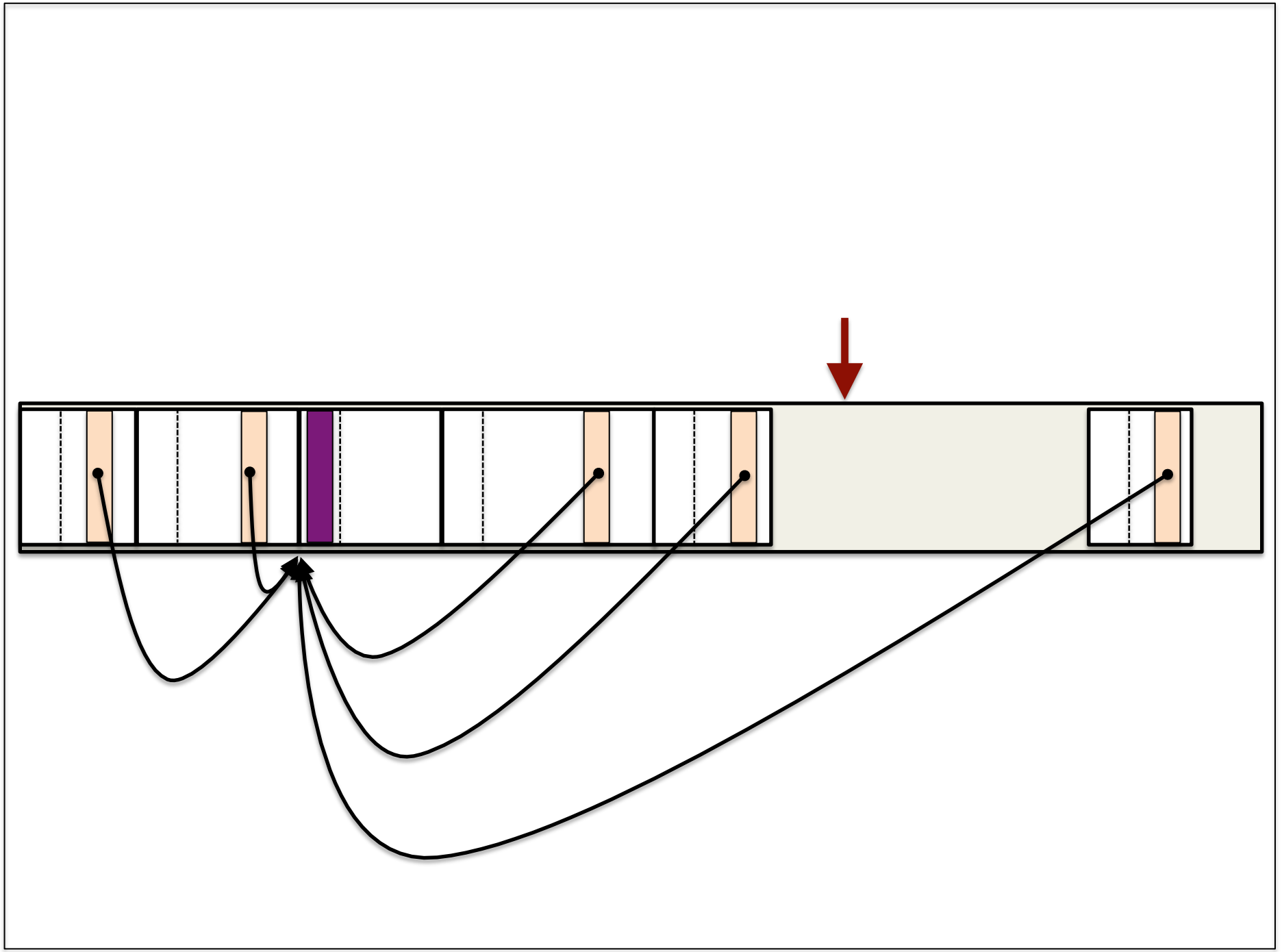


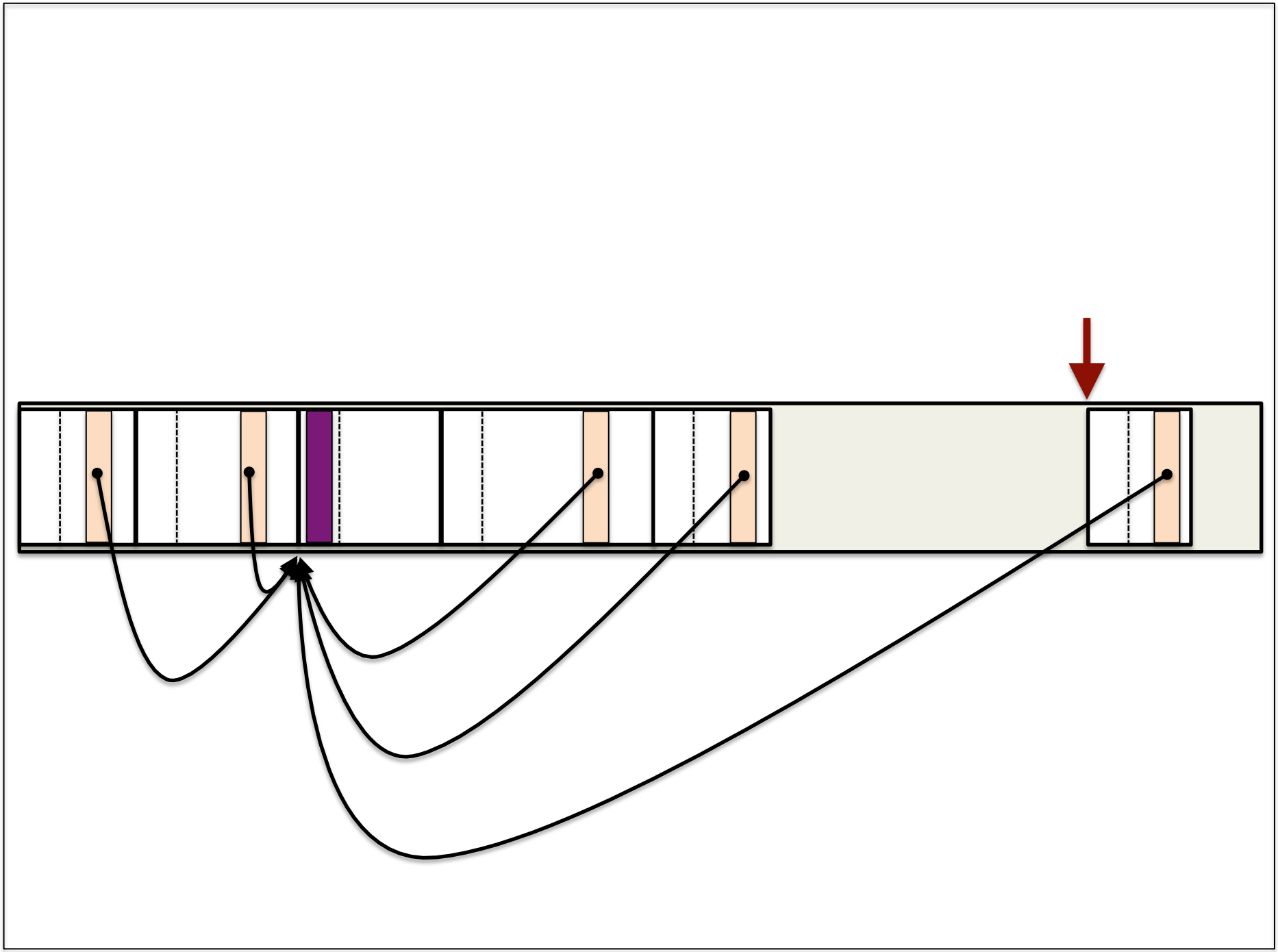


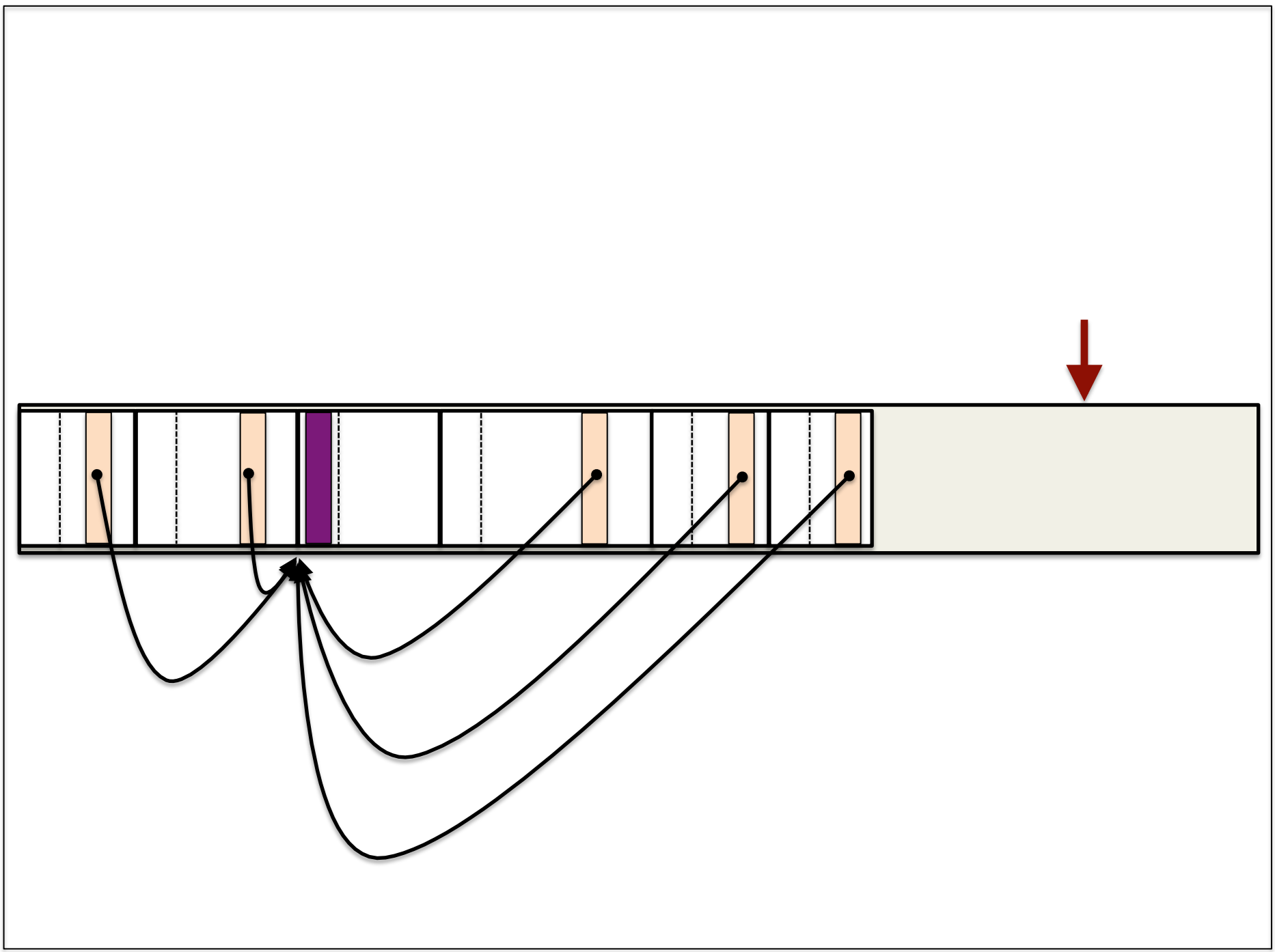


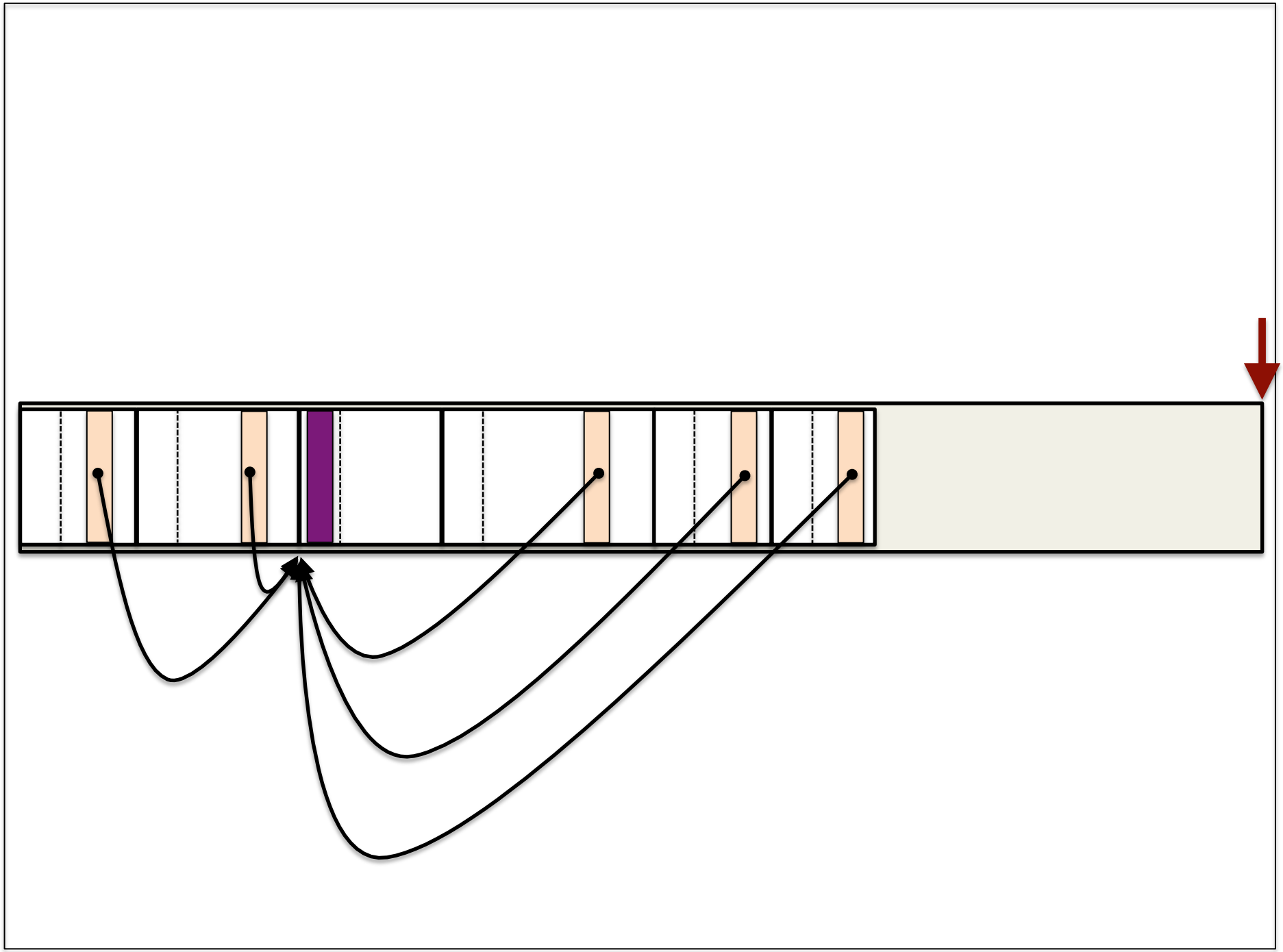












Compacting Collectors

- Compacting good for constrained architectures
  - No copy reserve needed
- Need one available bit
  - Memory is normally aligned at least that much
- Need one available word
  - Can generally use existing header structures

Compacting Collectors

- Complicated
  - Implementation trickier than other algorithms
- Requires multiple passes of the heap
  - At least two linear scans
  - Jumping around memory to chain pointers
- Hard to make concurrent
  - The heap is inconsistent for a long time

Comparing Garbage Collectors

	Reference Counting	Mark and Sweep	Semi-Space Copying	Compacting

Comparing Garbage Collectors

	Reference Counting	Mark and Sweep	Semi-Space Copying	Compacting
Incrementality	High	Low	Low	Low

Comparing Garbage Collectors

	Reference Counting	Mark and Sweep	Semi-Space Copying	Compacting
Incrementality	High	Low	Low	Low
Tracing	No	Yes	Yes	Yes

Comparing Garbage Collectors

	Reference Counting	Mark and Sweep	Semi-Space Copying	Compacting
Incrementality	High	Low	Low	Low
Tracing	No	Yes	Yes	Yes
Allocation Overhead	High	High	Low	Low

Comparing Garbage Collectors

	Reference Counting	Mark and Sweep	Semi-Space Copying	Compacting
Incrementality	High	Low	Low	Low
Tracing	No	Yes	Yes	Yes
Allocation Overhead	High	High	Low	Low
Heap Utilization	Reasonable	Reasonable	Poor	Good

Comparing Garbage Collectors

	Reference Counting	Mark and Sweep	Semi-Space Copying	Compacting
Incrementality	High	Low	Low	Low
Tracing	No	Yes	Yes	Yes
Allocation Overhead	High	High	Low	Low
Heap Utilization	Reasonable	Reasonable	Poor	Good
Complexity	Low	Medium	Medium	High

Comparing Garbage Collectors

	Reference Counting	Mark and Sweep	Semi-Space Copying	Compacting
Incrementality	High	Low	Low	Low
Tracing	No	Yes	Yes	Yes
Allocation Overhead	High	High	Low	Low
Heap Utilization	Reasonable	Reasonable	Poor	Good
Complexity	Low	Medium	Medium	High
Garbage Object Overhead	High	High	Low	Low

Comparing Garbage Collectors

	Reference Counting	Mark and Sweep	Semi-Space Copying	Compacting
Incrementality	High	Low	Low	Low
Tracing	No	Yes	Yes	Yes
Allocation Overhead	High	High	Low	Low
Heap Utilization	Reasonable	Reasonable	Poor	Good
Complexity	Low	Medium	Medium	High
Garbage Object Overhead	High	High	Low	Low
Live Object Overhead	None	Low	High	High

Live to Garbage Ratio

- Amount of data that survives a collection
  - Important metric when choosing collector
  - Very widely studied in practice
- Moving collectors pay cost per live object
  - Copying overhead
- Others pay cost per garbage object
  - Freeing objects individually

Cache Behavior

- Memory latency a major factor in performance
  - CPU architecture tries to minimize its effect
  - Best way to do this is through the cache
- We're interested in cache behavior in two places
  - During garbage collector
  - During mutator operation
- We can use GC to optimize mutator performance

Principal of Locality

- Also known as Locality of Reference
- Suggests that cache behavior can be predicted
  - For some value of predicted
  - Knowing the future is hard
- Processors are designed with locality in mind
  - Principal developed over decades of observation
  - Consistently found to be accurate

Temporal Locality

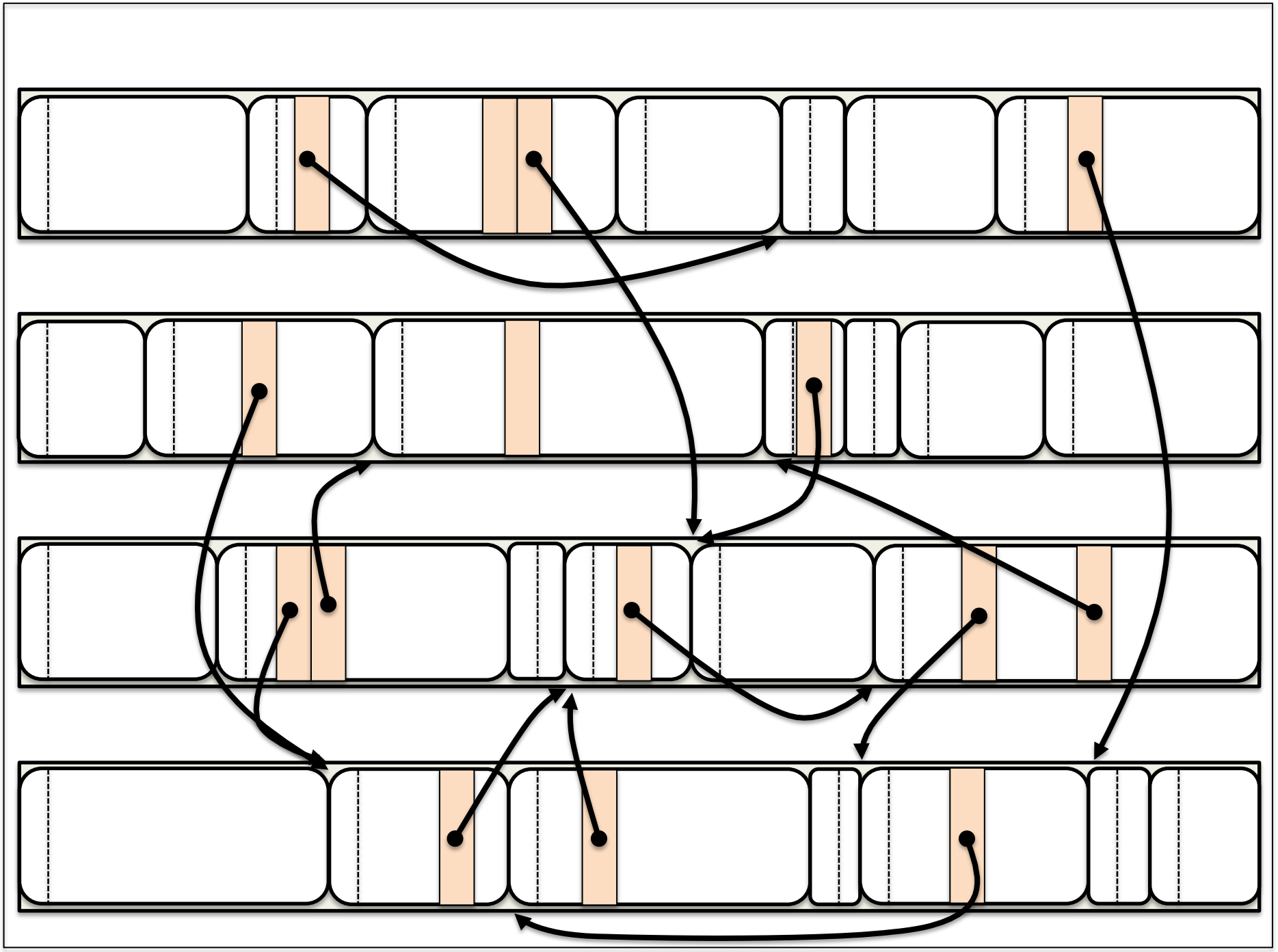
- Accesses to values tend to be bursty
  - Once you use a value, you likely use it again soon
  - Afterwards it may be a long time before using it again
- Hardware caches retain data after use
  - Replacement policies favor recently-used data

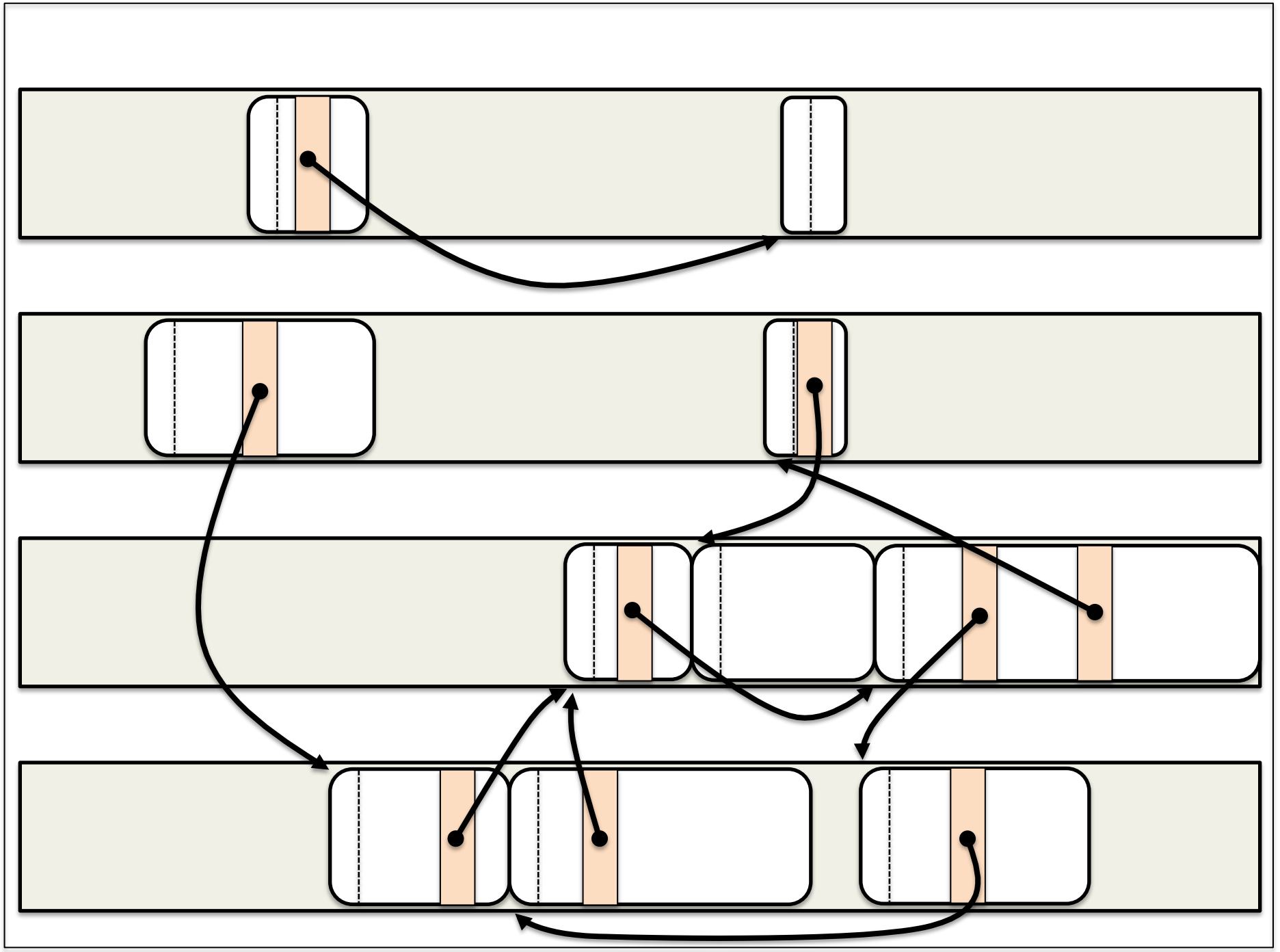
Spatial Locality

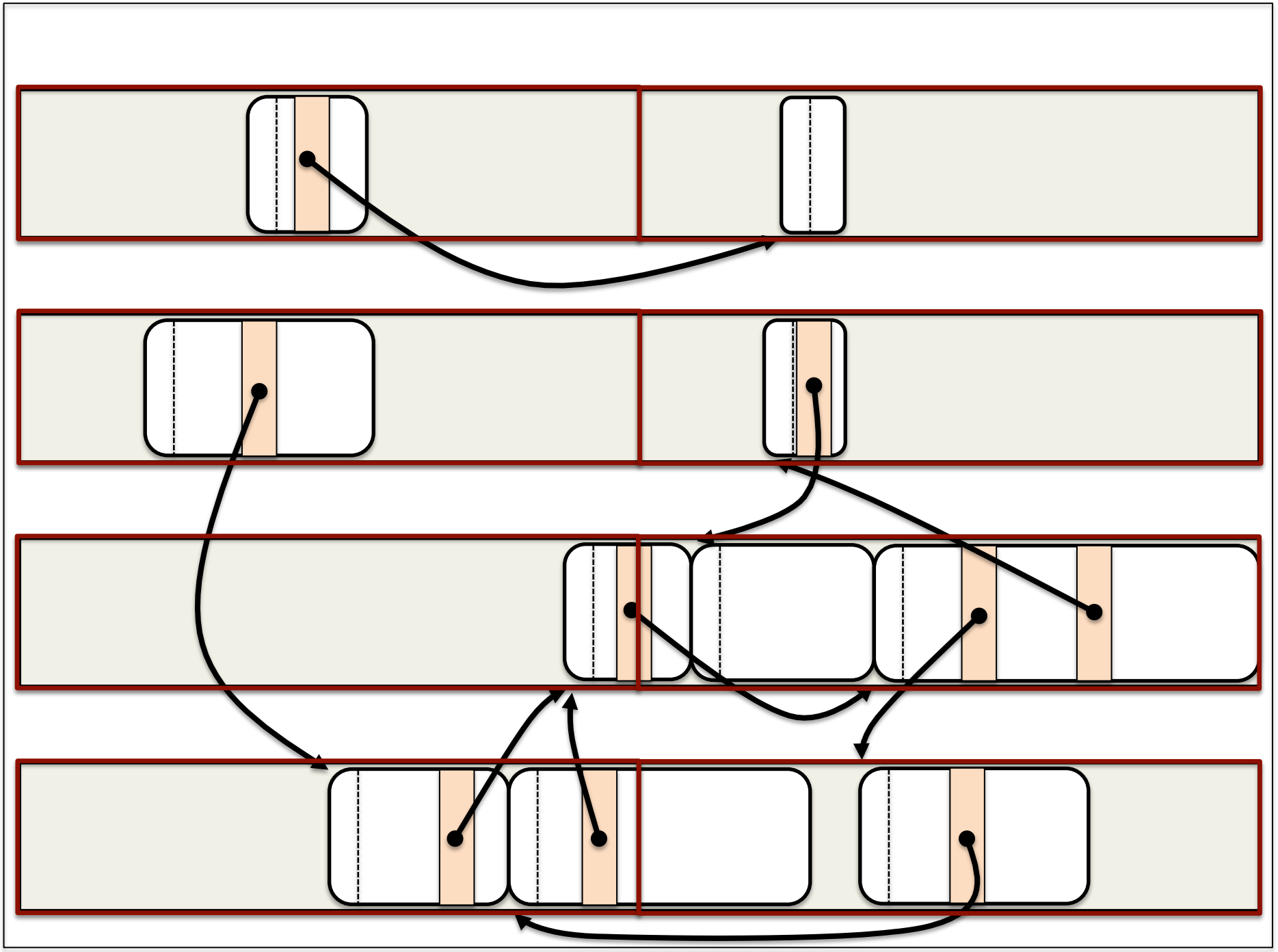
- Nearby objects tend to be used together
  - Think of scanning through an array
  - Accessing fields in an object
- Hardware caches read data in lines
  - Bring in the requested address plus nearby data
  - Likely to have the next object already in the cache
- Caches best at going in a straight line

Maximizing Locality

- Garbage collection is an artificial process
  - Doesn't necessarily follow locality principle
  - Touches much of the heap at once
- Design of a collector can make a big difference
  - Linear scan better than random access
  - Clustering updates close in time
- Compacting collectors bring live objects together

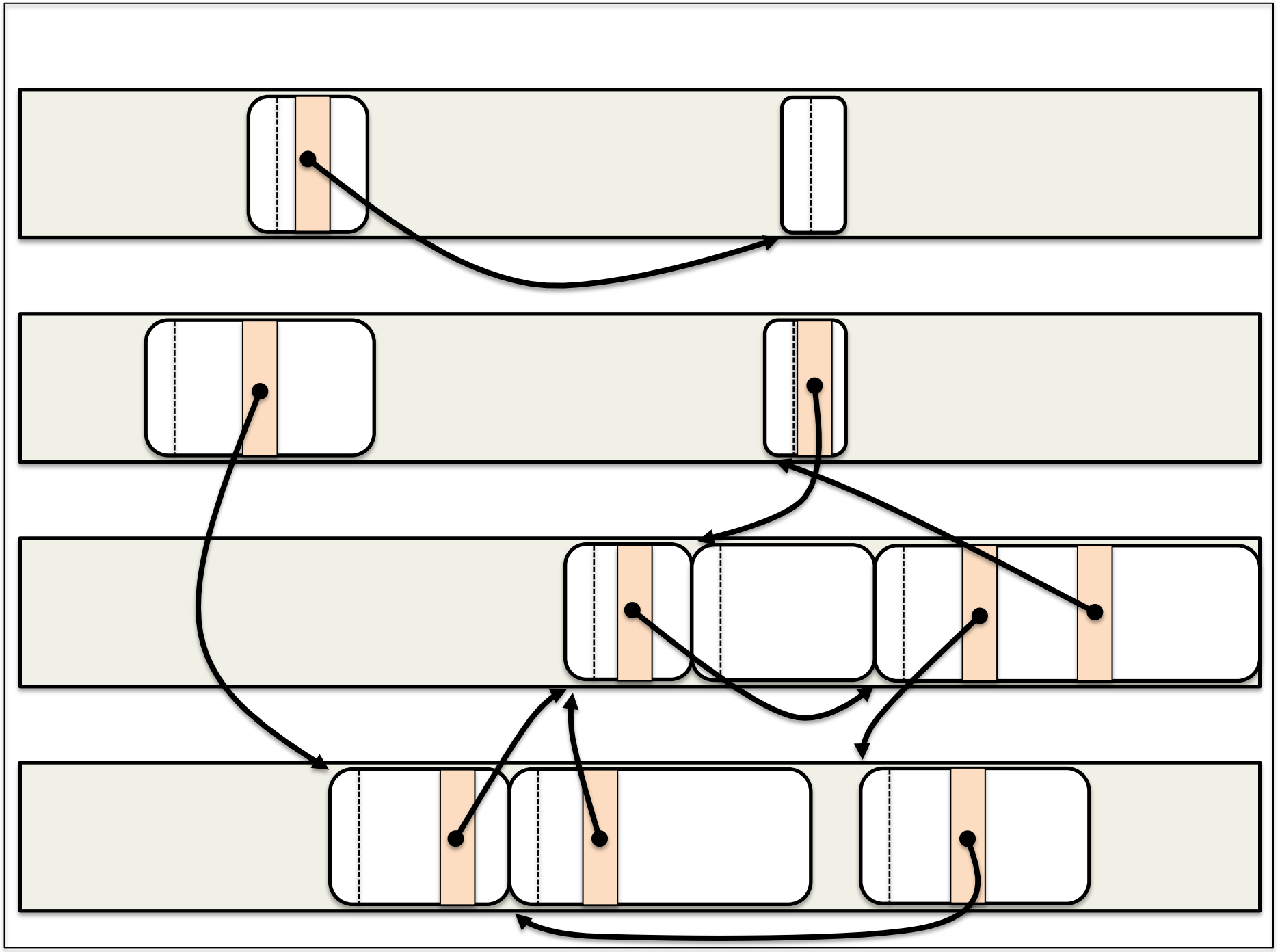


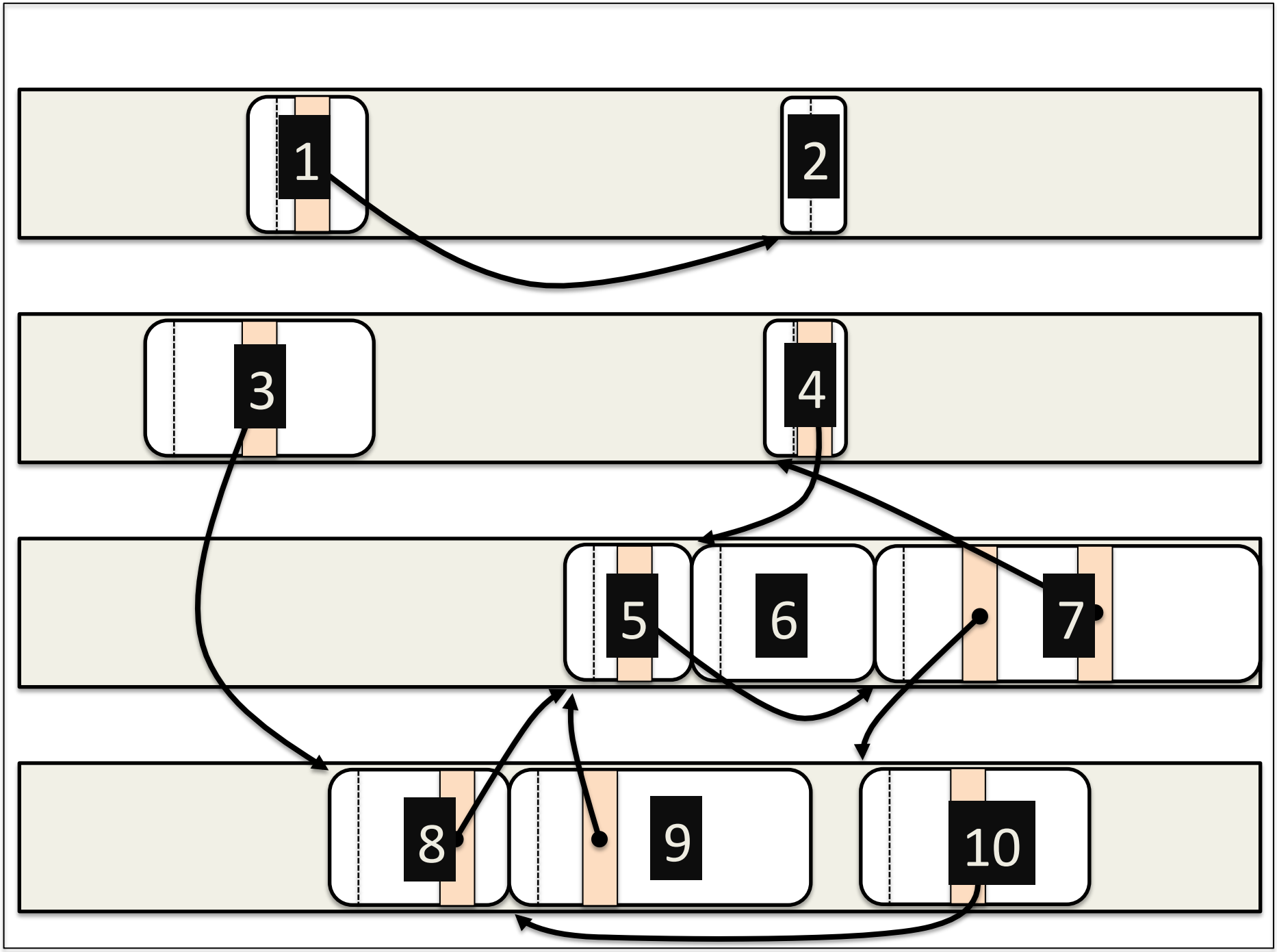


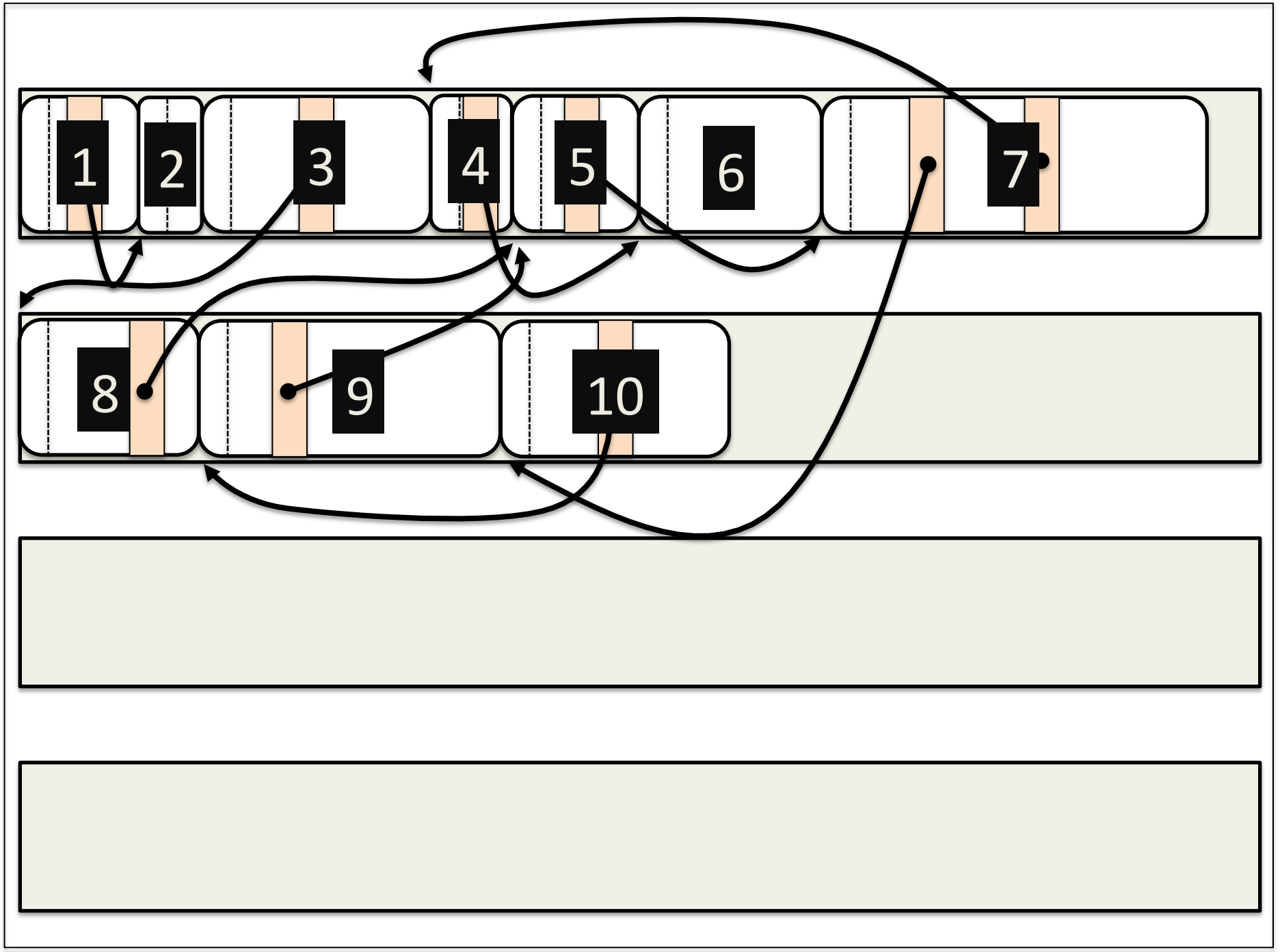


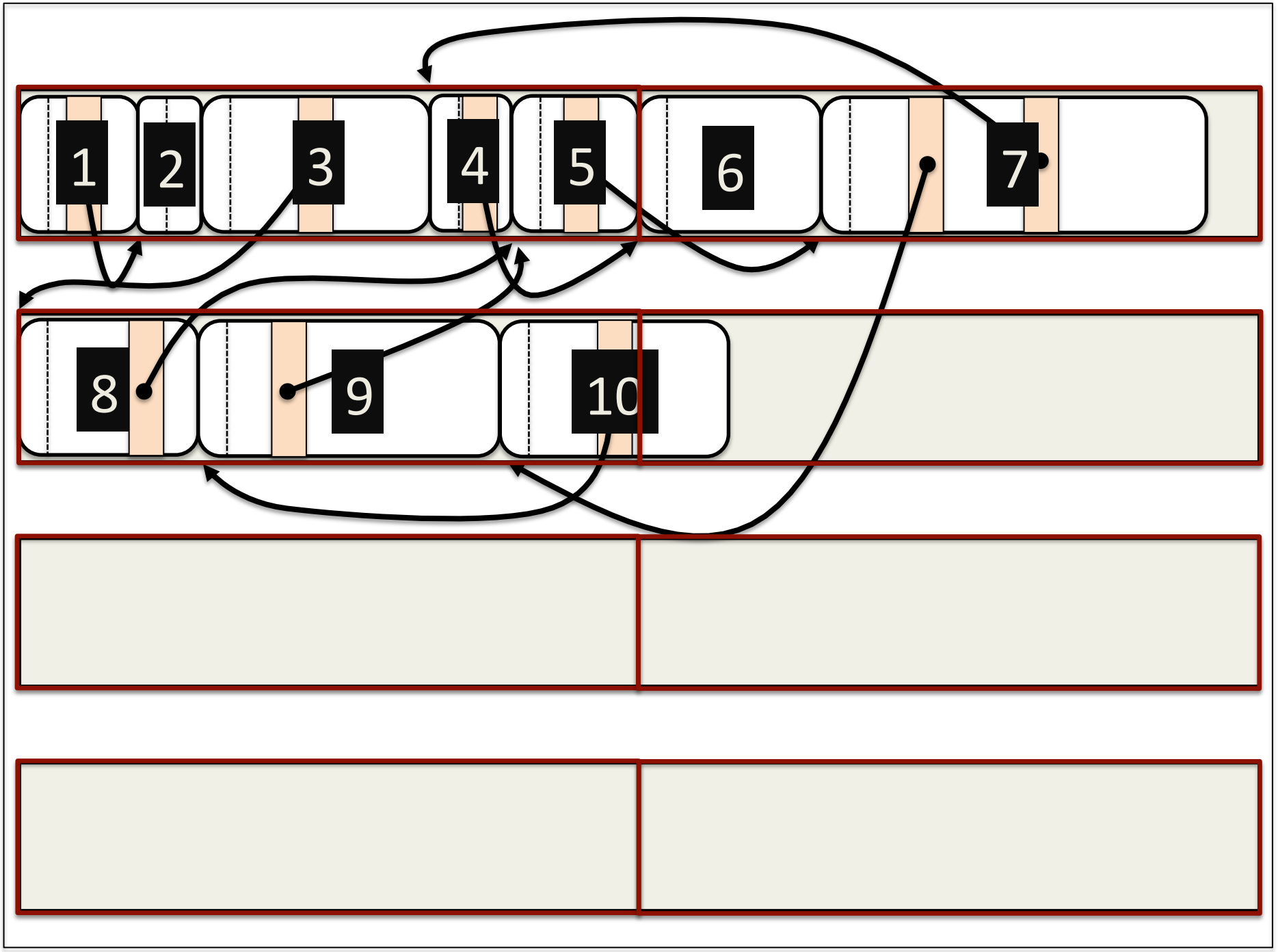
Locality in Mark and Sweep

- Fragmentation affects cache behavior
  - Cache blocks contain empty space
  - Unrelated data fills in gaps
  - We'll see next week why high fragmentation is usual




1
2
3
4
5
6
7
8
9
10


1
2
3
4
5
6
7
8
9
10

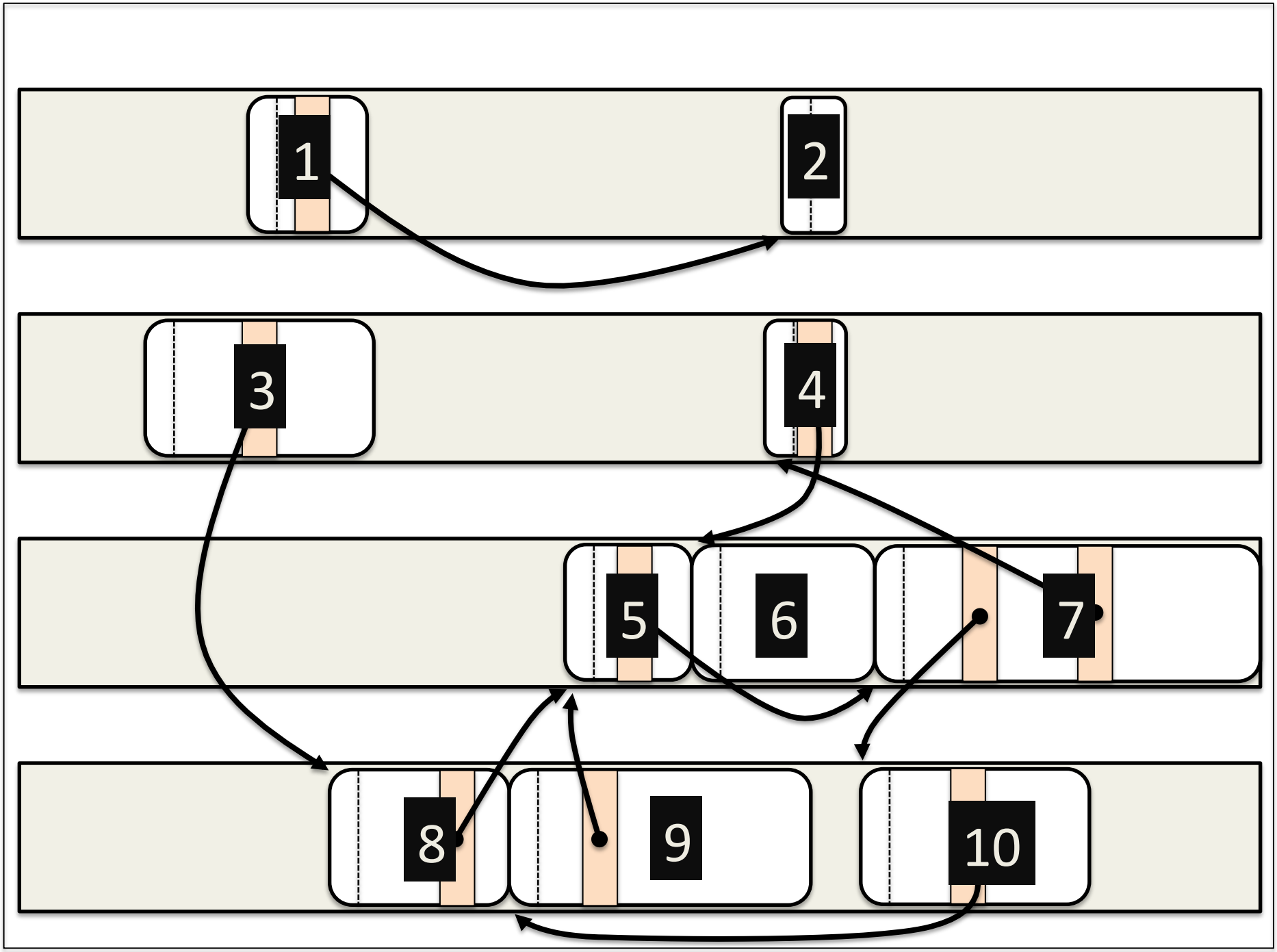

1
2
3
4
5
6
7
8
9
10

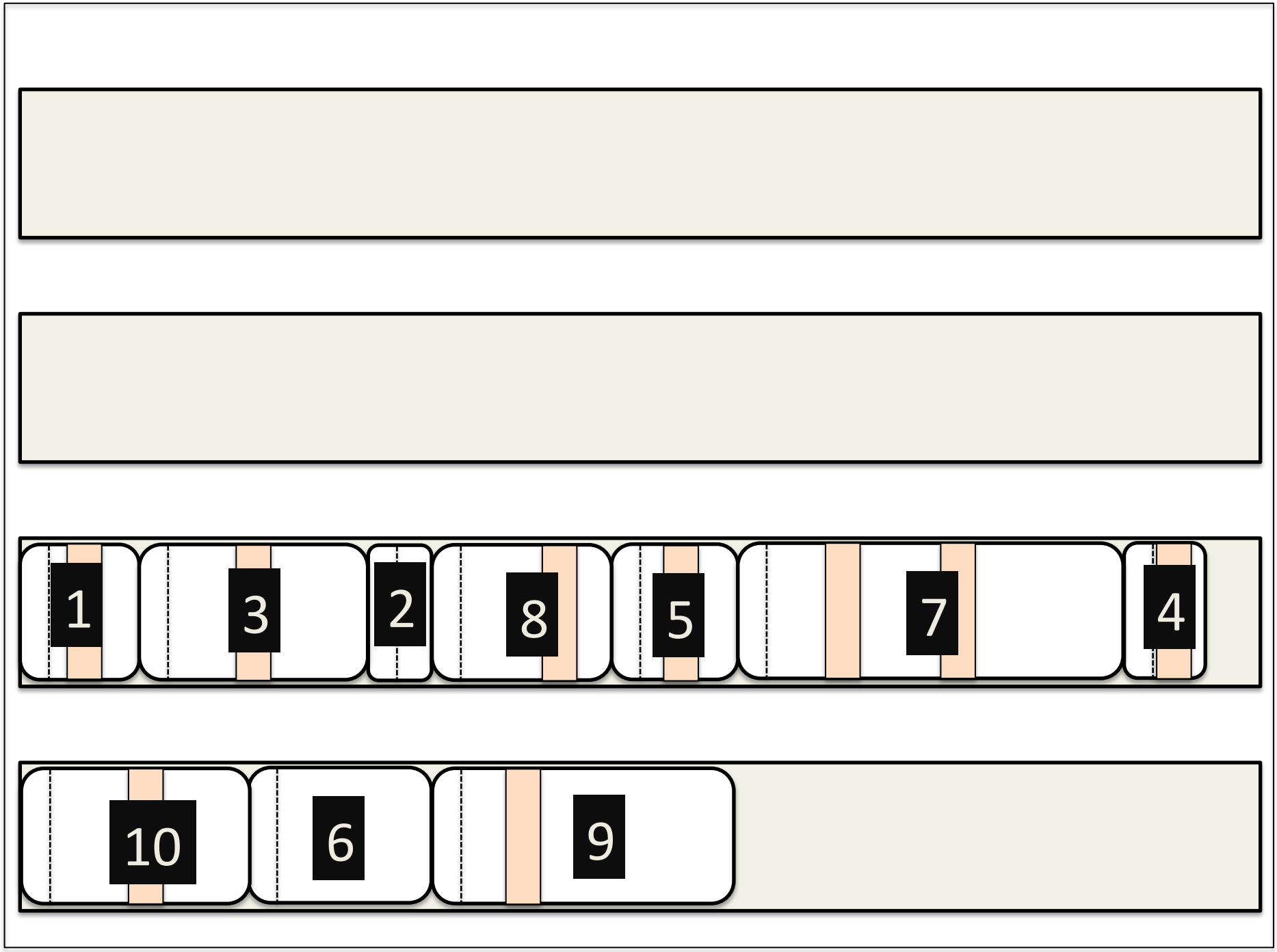
Locality in Compacting Collectors

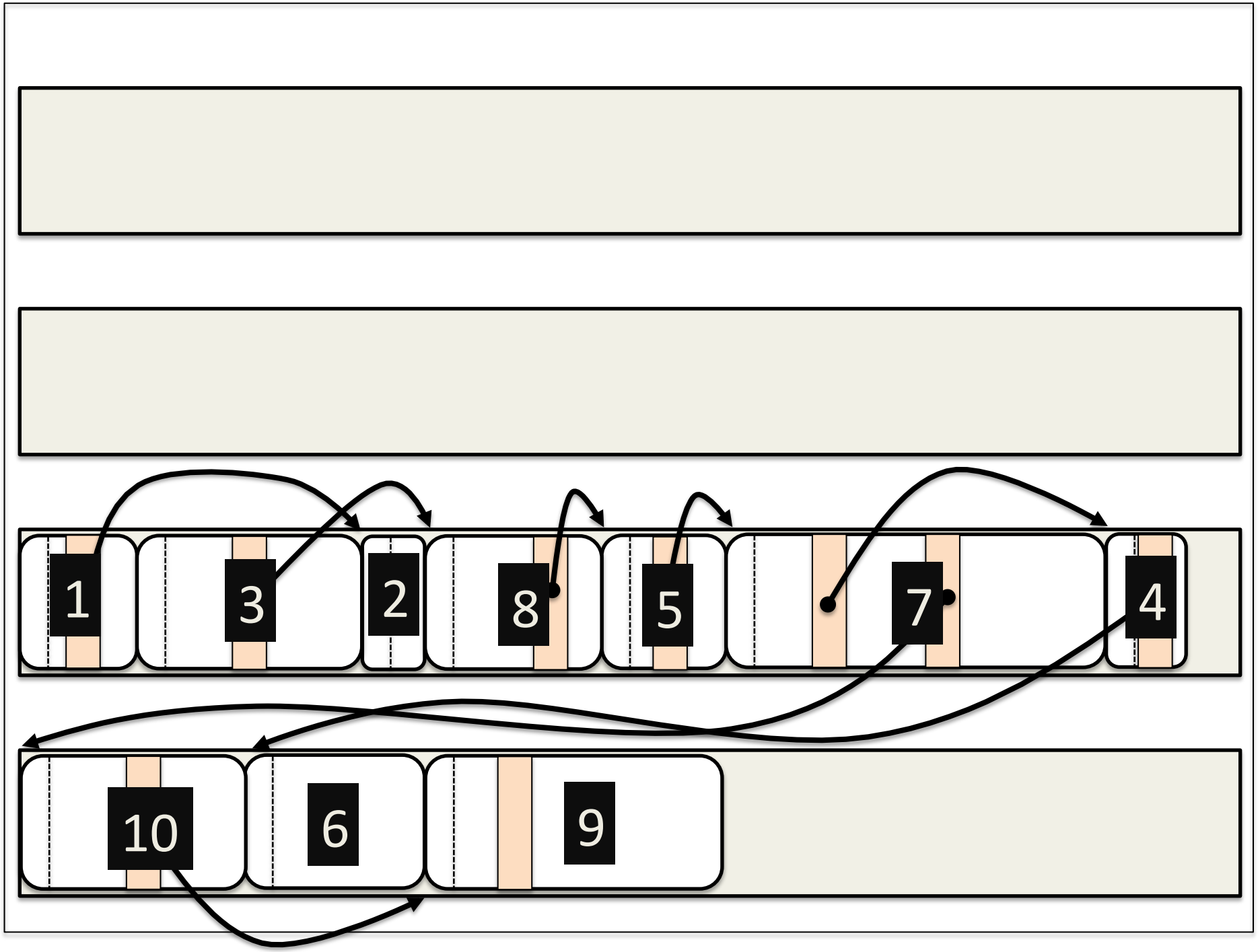
- Compaction pass eliminates fragmentation
  - No holes in the cache line
- Objects maintain allocation order
  - Good spatial locality
- Is there a better ordering?

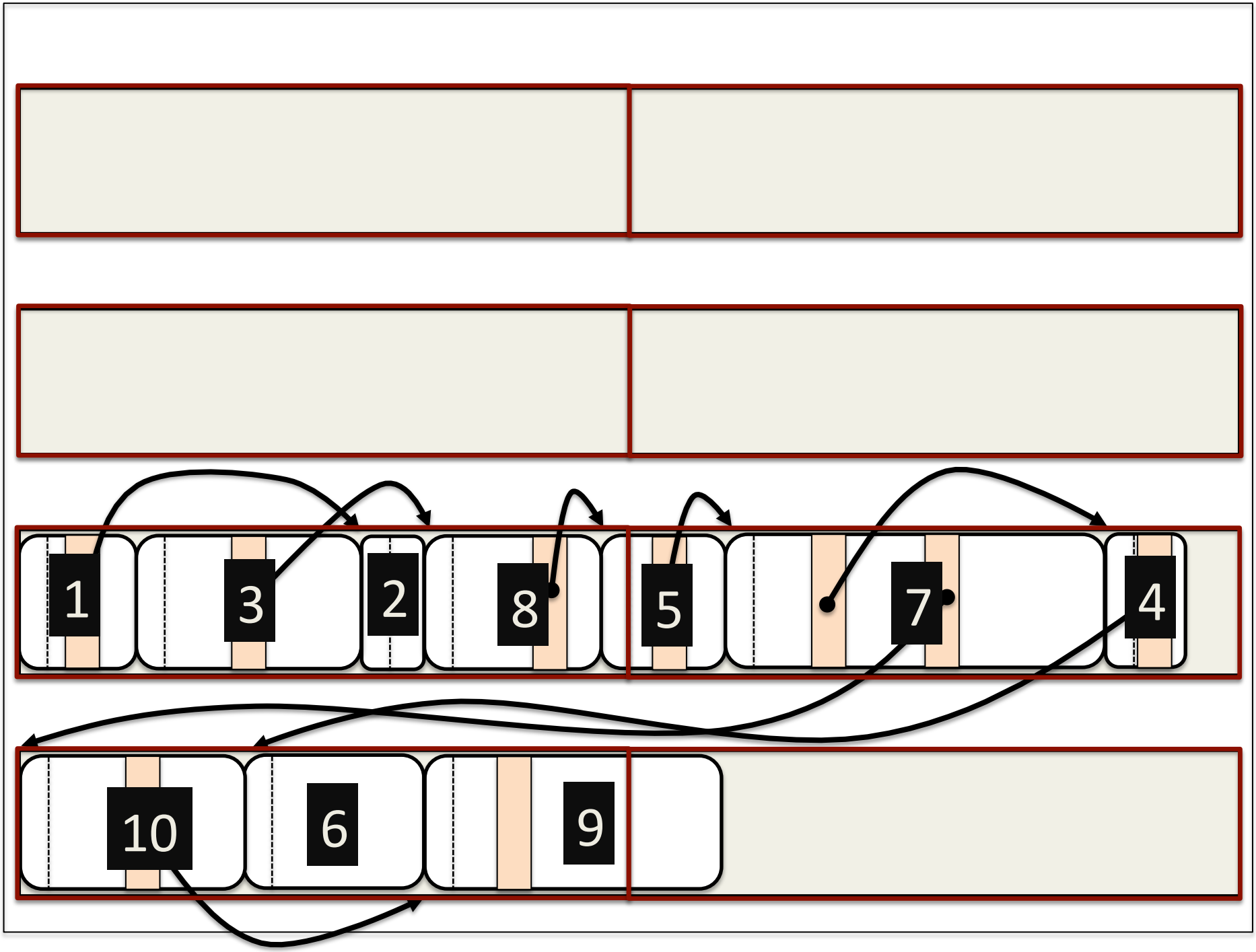
Object Oriented Languages

- Spatial locality extends to references
  - Objects that point to one another accessed together
- References between objects can change
  - Object graph mutated over time
  - Allocation order may no longer be optimal
- Mark Sweep and Compaction can't exploit changes
  - Objects stay in allocation order


1
2
3
4
5
6
7
8
9
10


1
3
2
8
5
7
4
10
6
9


1
3
2
8
5
7
4
10
6
9


1
3
2
8
5
7
4
10
6
9

Dynamic Locality

- Appel-style collectors copy in breadth-first order
  - Objects are copied as references are found
- Resulting order reflects connectivity
- Later collections can reorder objects
  - Can be useful if access patterns shift
- Generally get the most effect the first time
  - Cost of moving dominates application improvement

Conservative GC Revisited

- Conservative GCs can't move objects
  - Don't know for certain if a value is a pointer
  - May update an integer
- Most high-performing GCs do some copying
  - Sometimes a hybrid approach
- Imposes a limit on GC performance