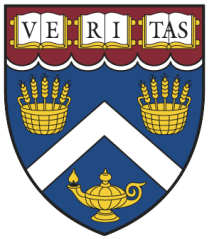
Uniprocessor GC Topics



Small Regions

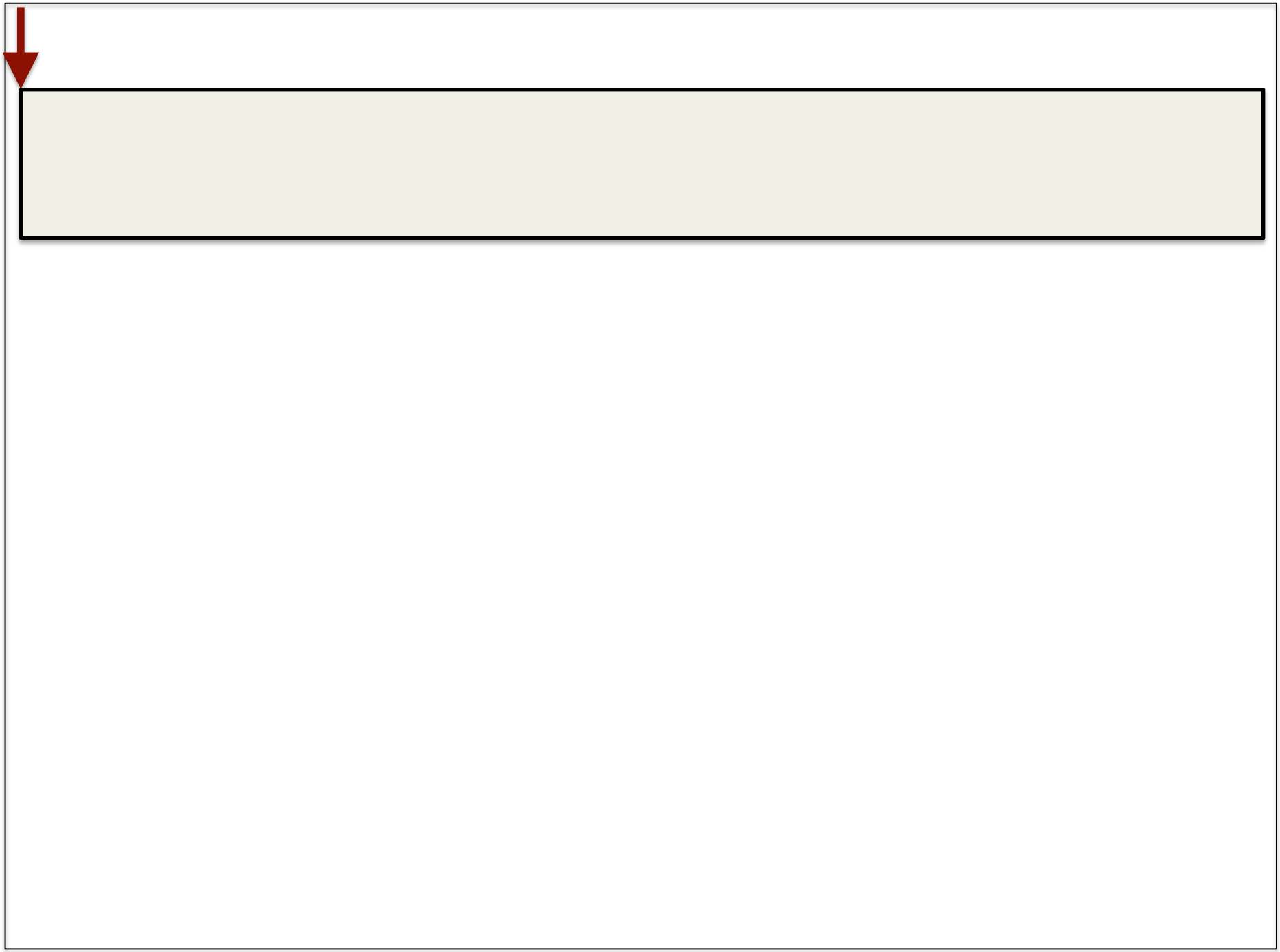
- Algorithms use regions differently
  - RC, M/S and compaction use one large region
  - SemiSpace uses two equal-sized regions
  - Generational uses two or more regions
- We don't need to limit ourselves
  - Use regions as a unit to group allocation
  - Extension of the generational GC concept

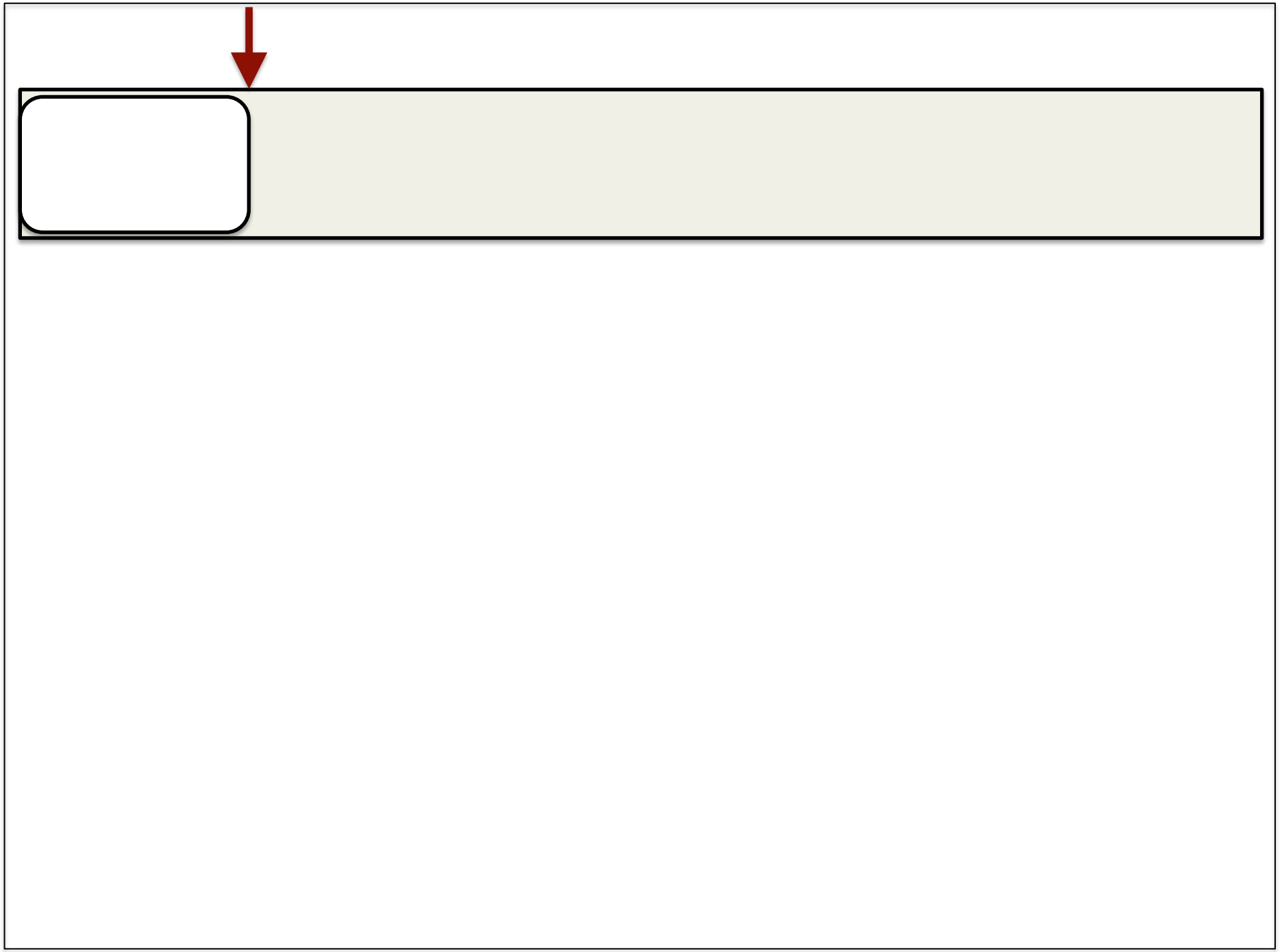
Region-Based Memory

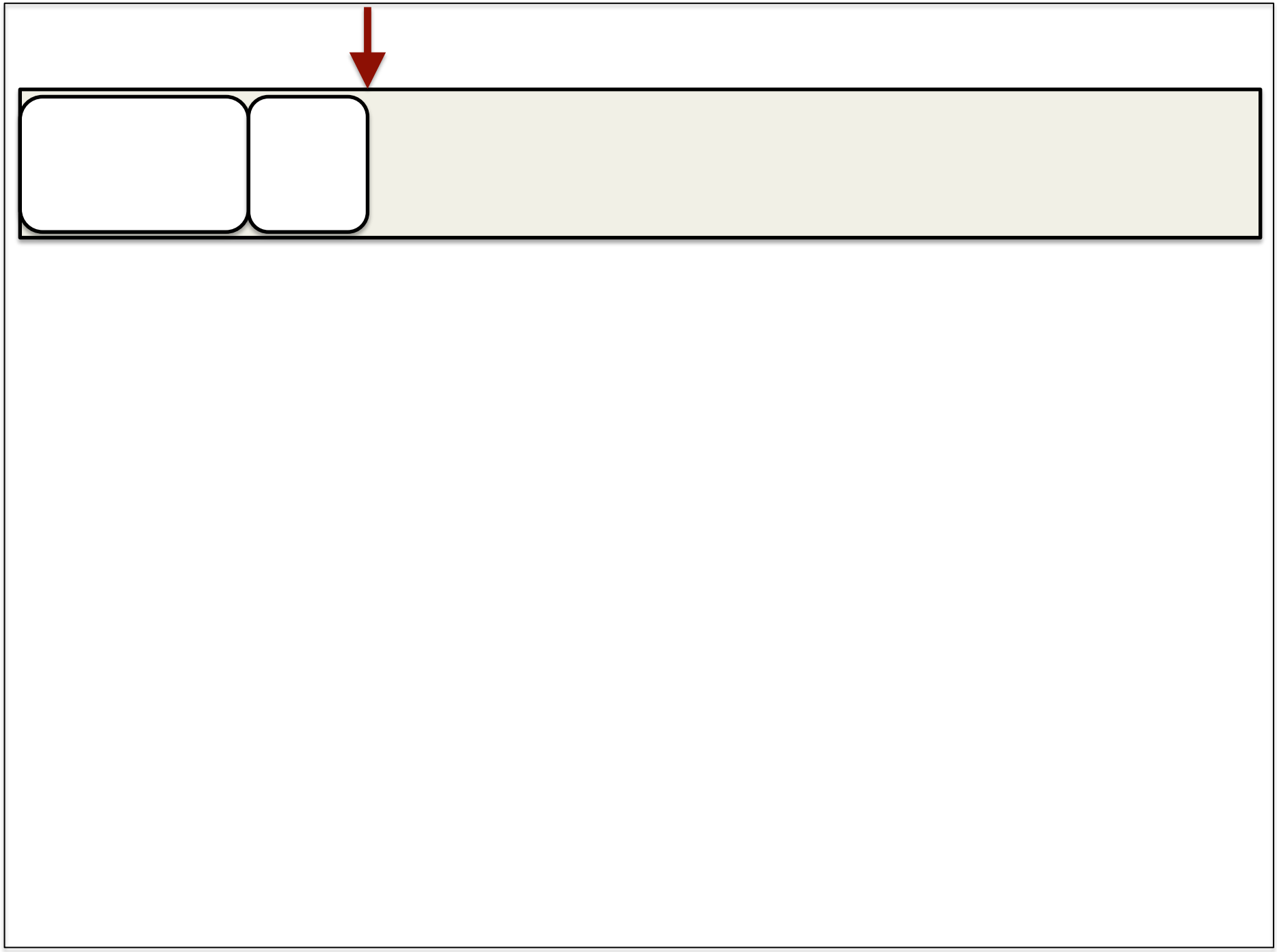
- Small regions improve incrementality
  - Don't need to collect the whole heap at once
- We can focus on regions with the most garbage
  - How do we know?
- More regions incur more overhead
  - Extension of generational remembered sets

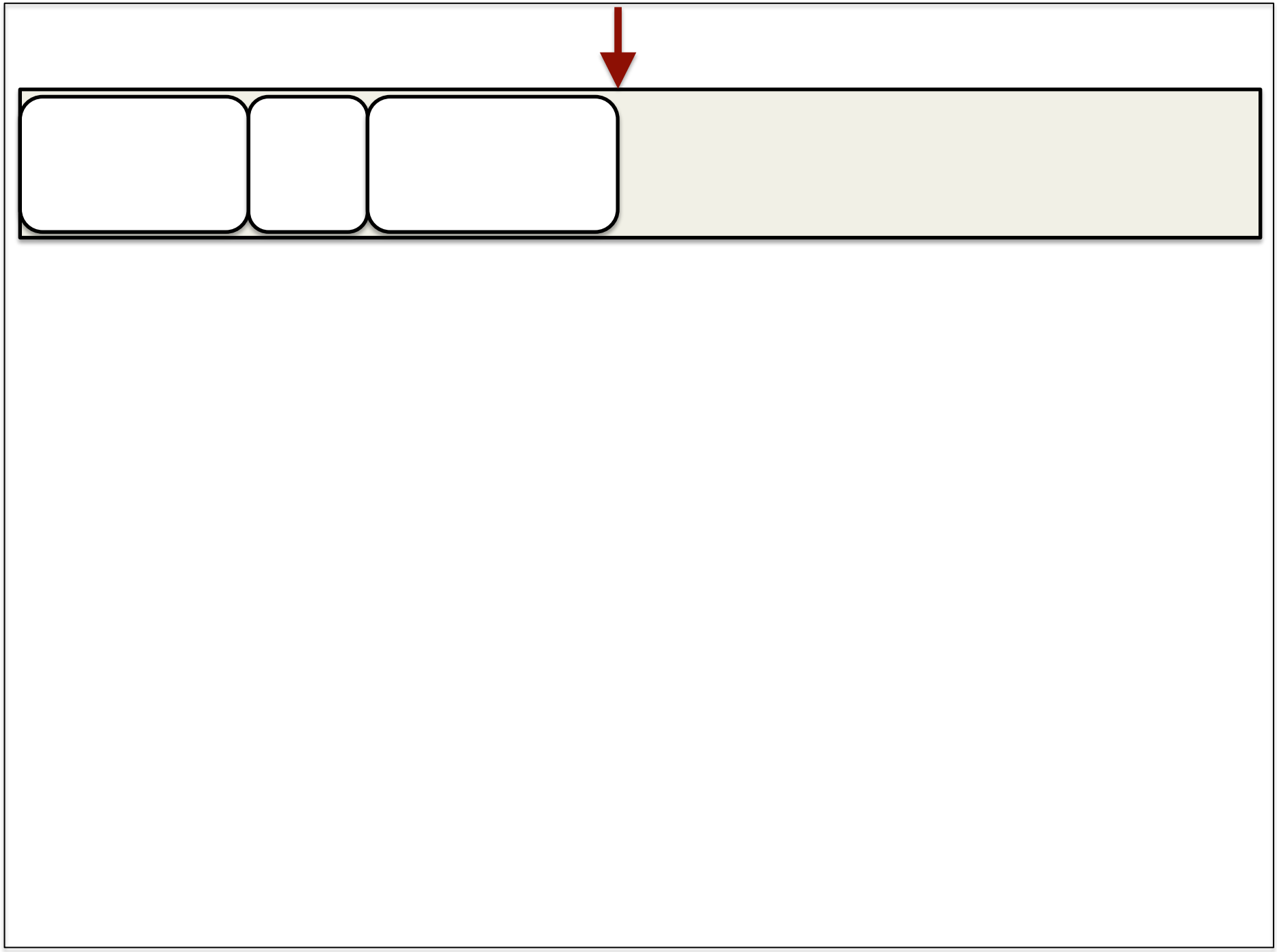
Scoped Memory

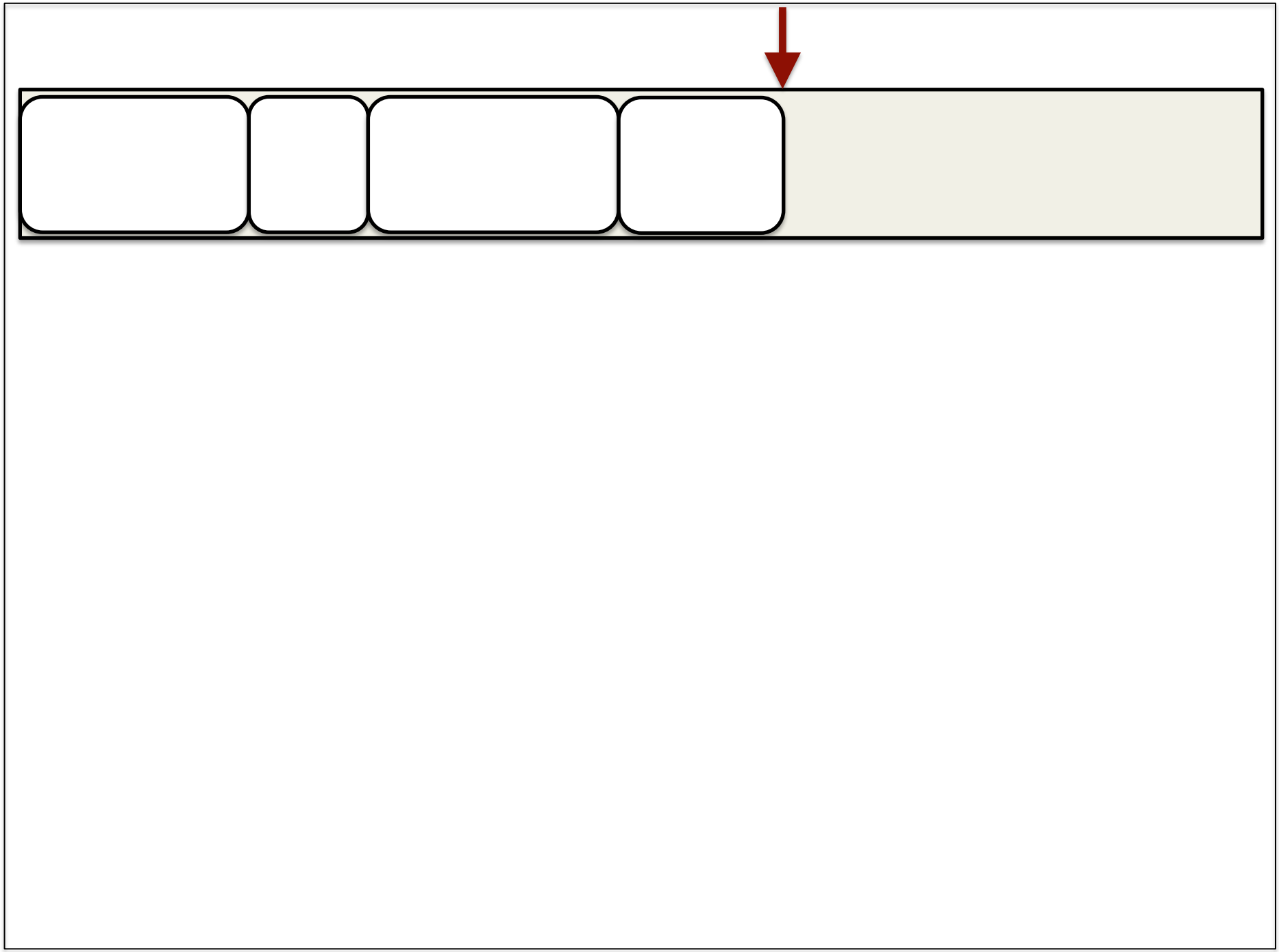
- Set up the heap as a stack of regions
  - Allocate objects in a single region
  - Discard when that memory is no longer needed
- Garbage collection overhead next to nothing
  - Don't track allocations, and don't trace the heap

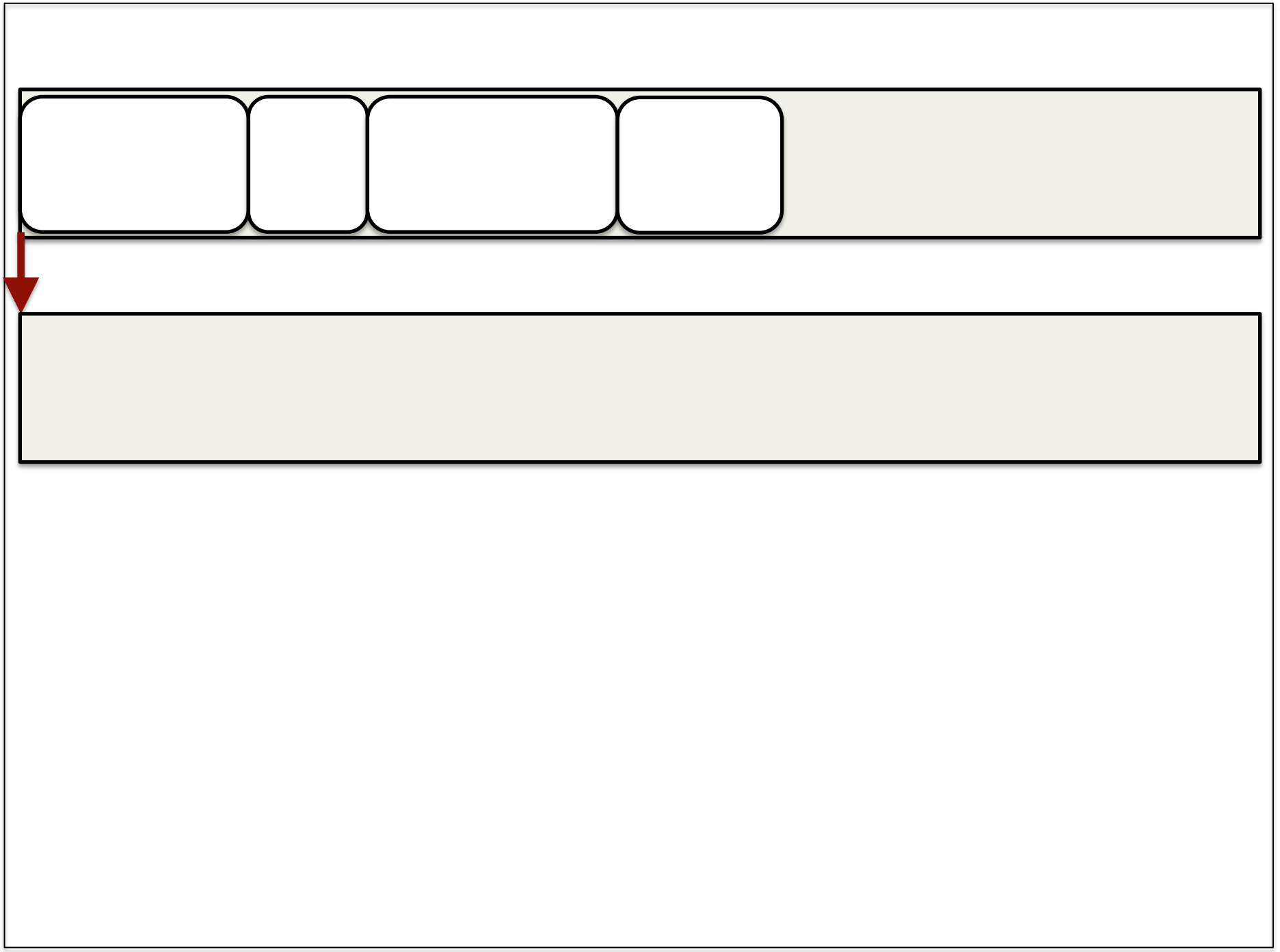


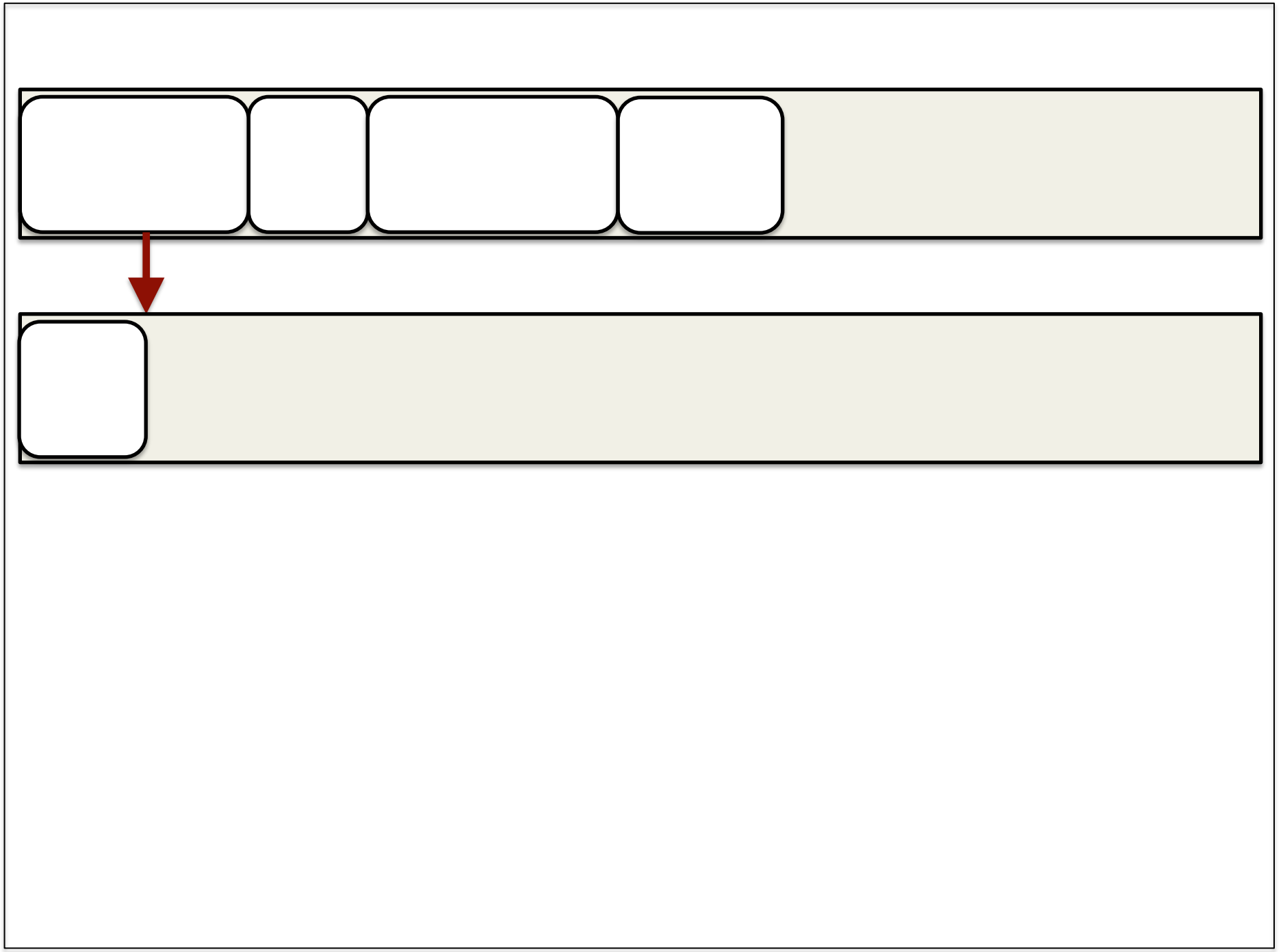


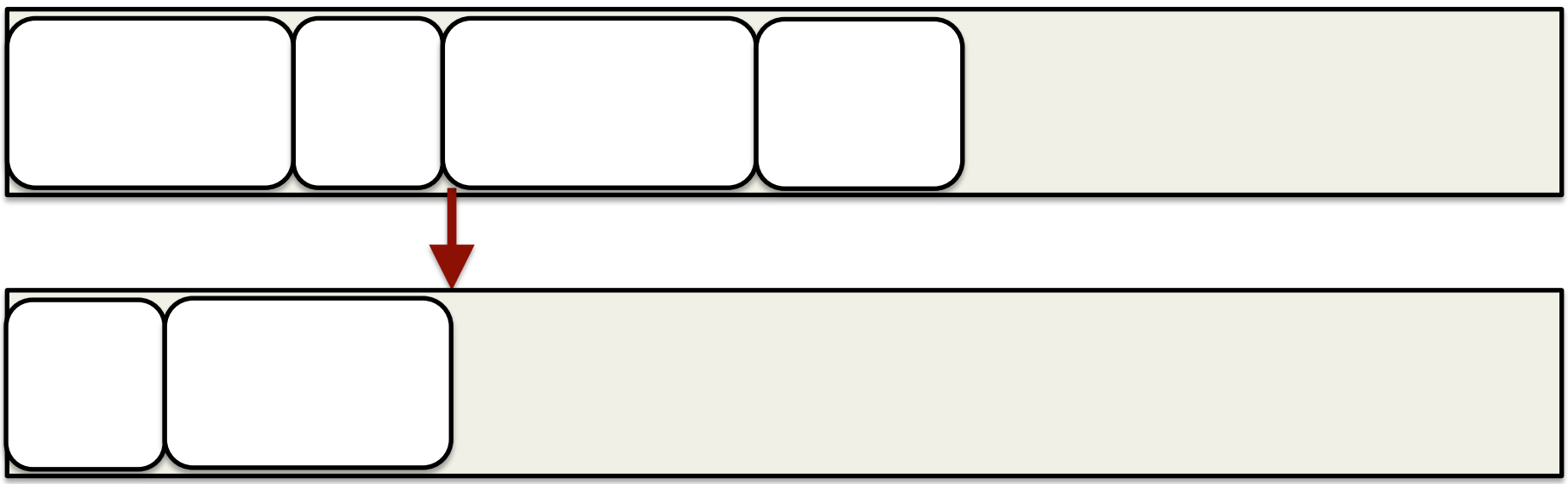


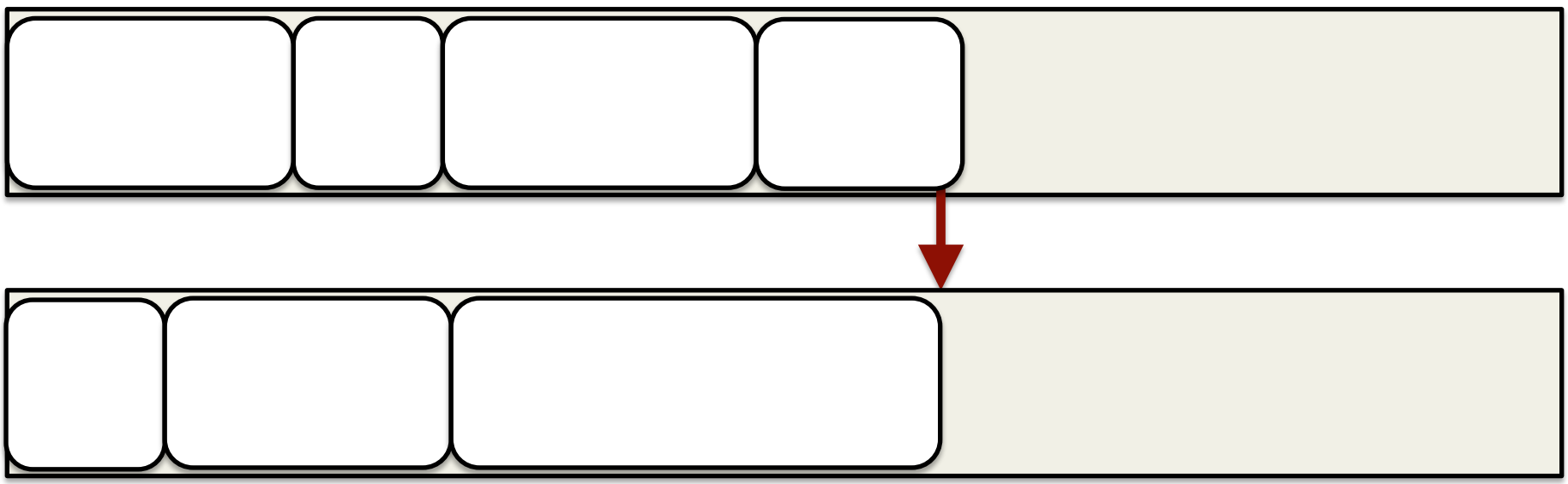


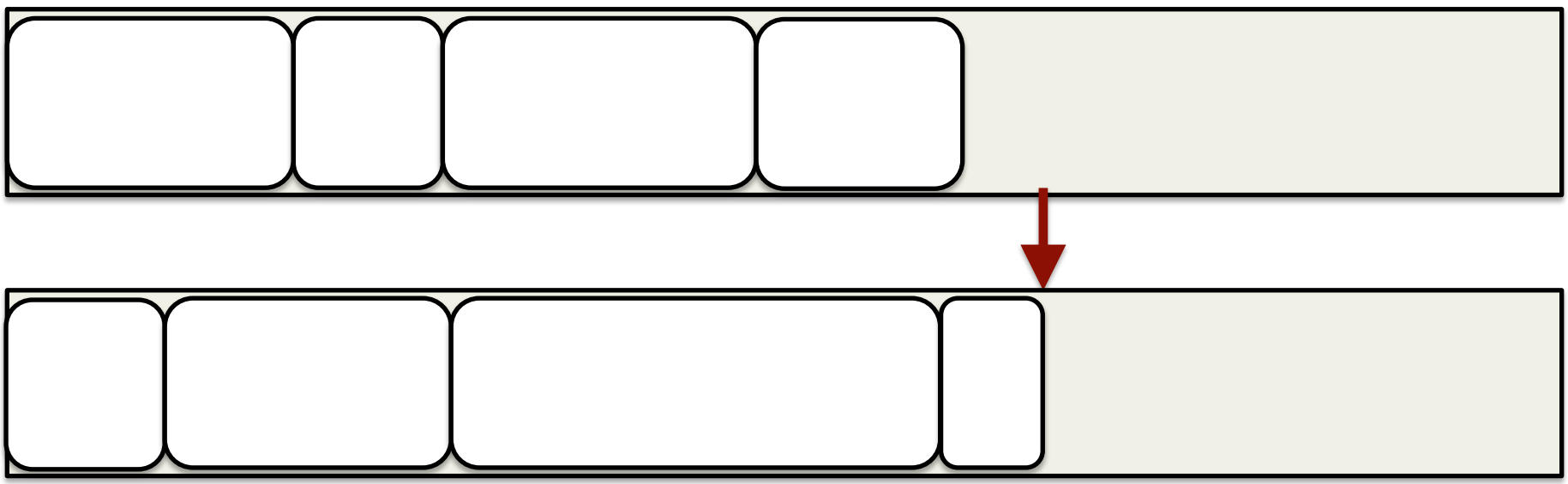


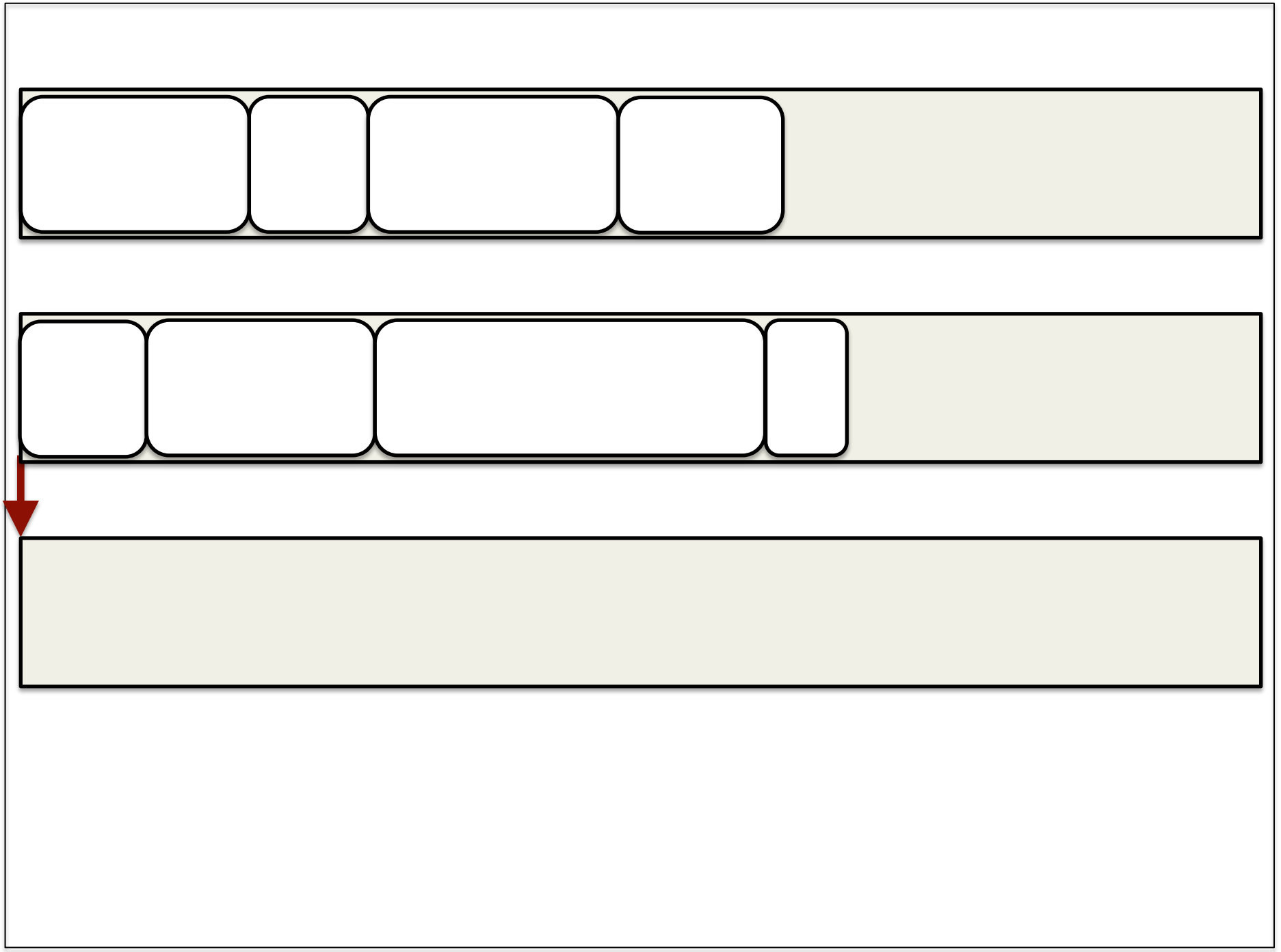


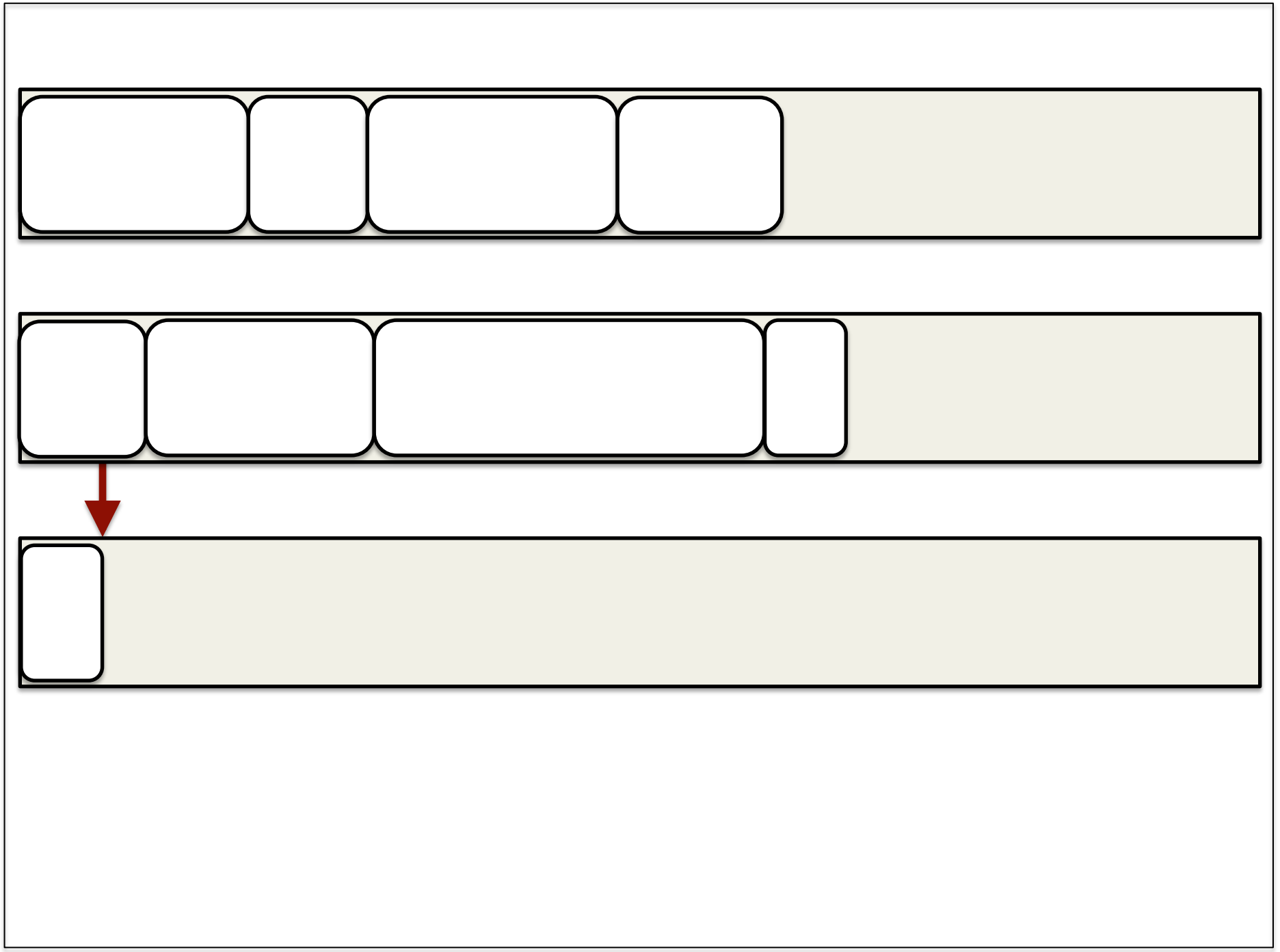


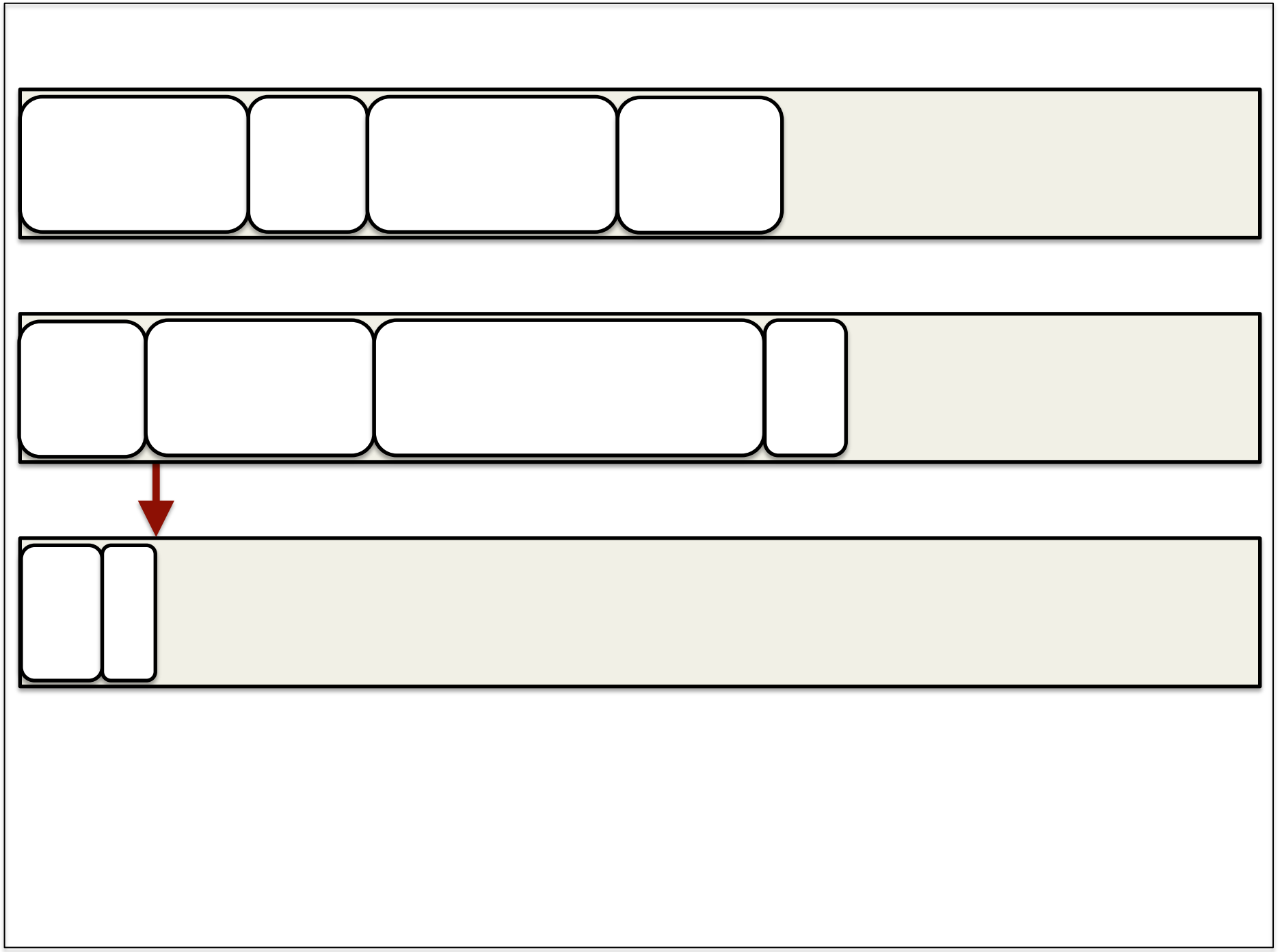


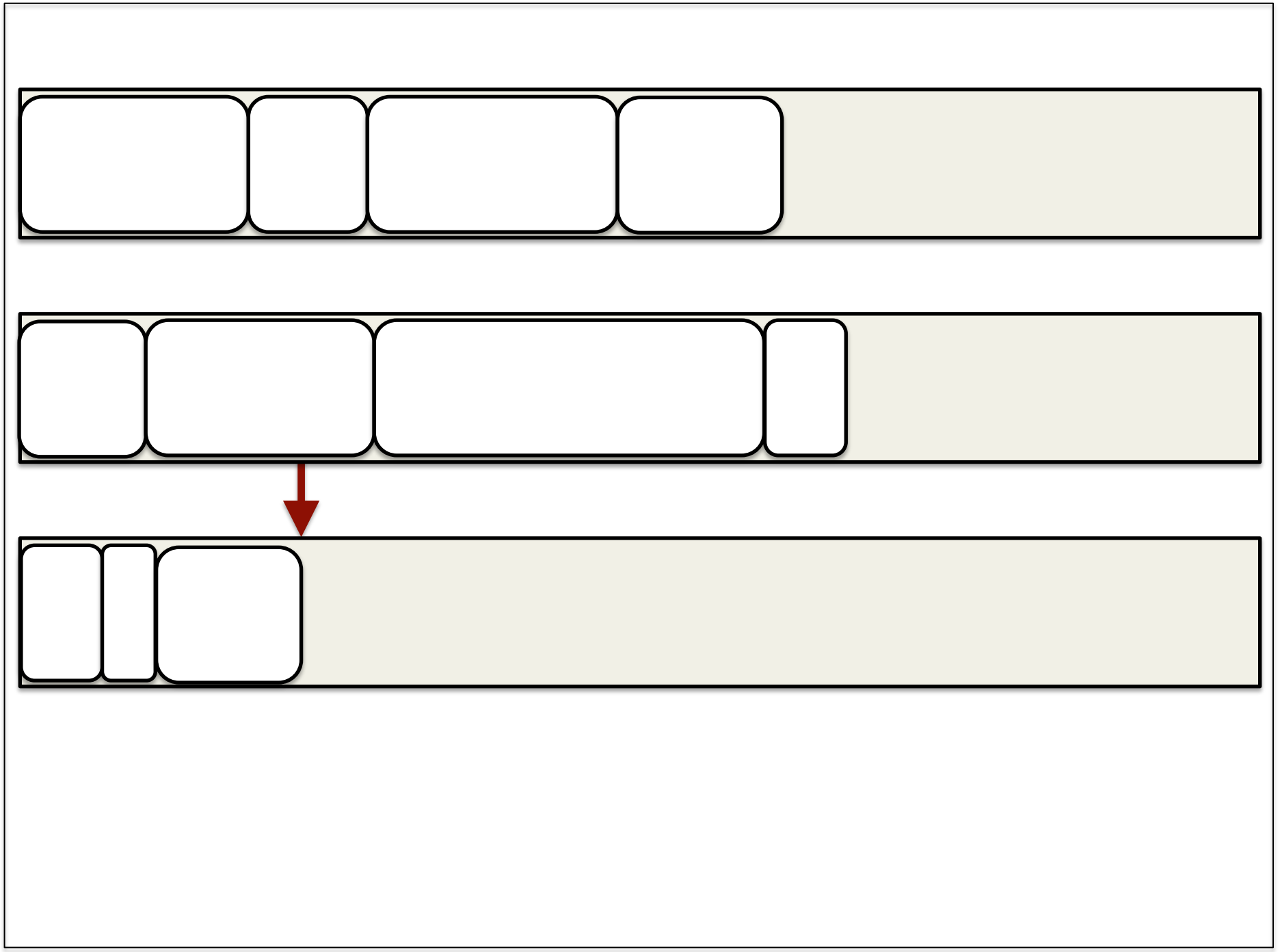


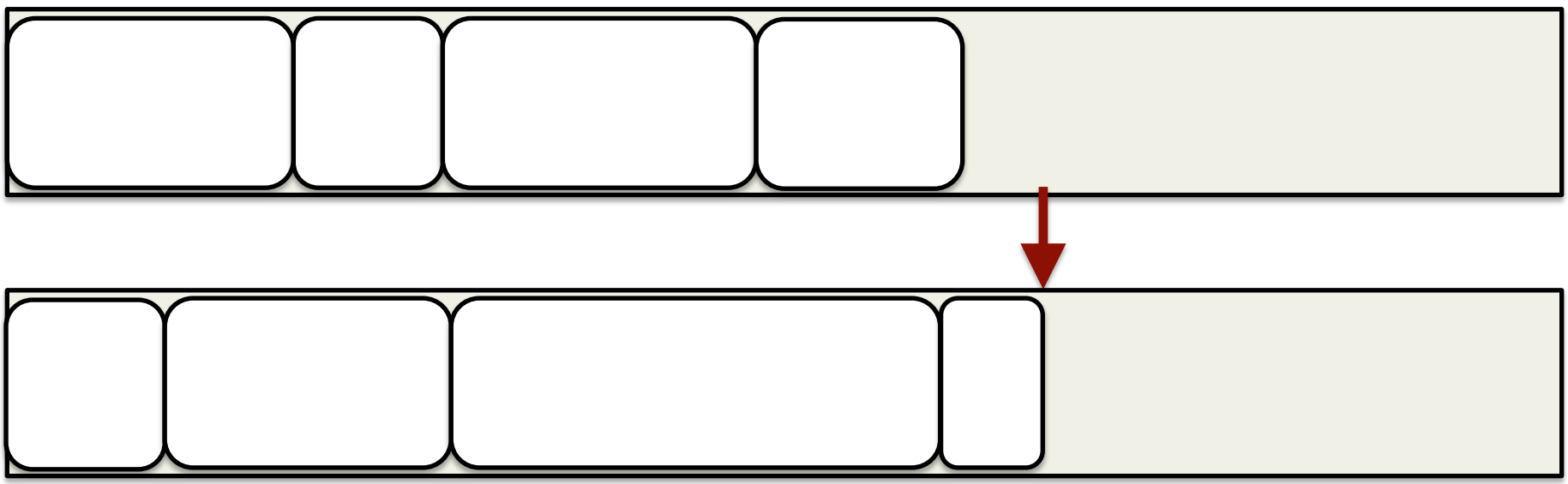


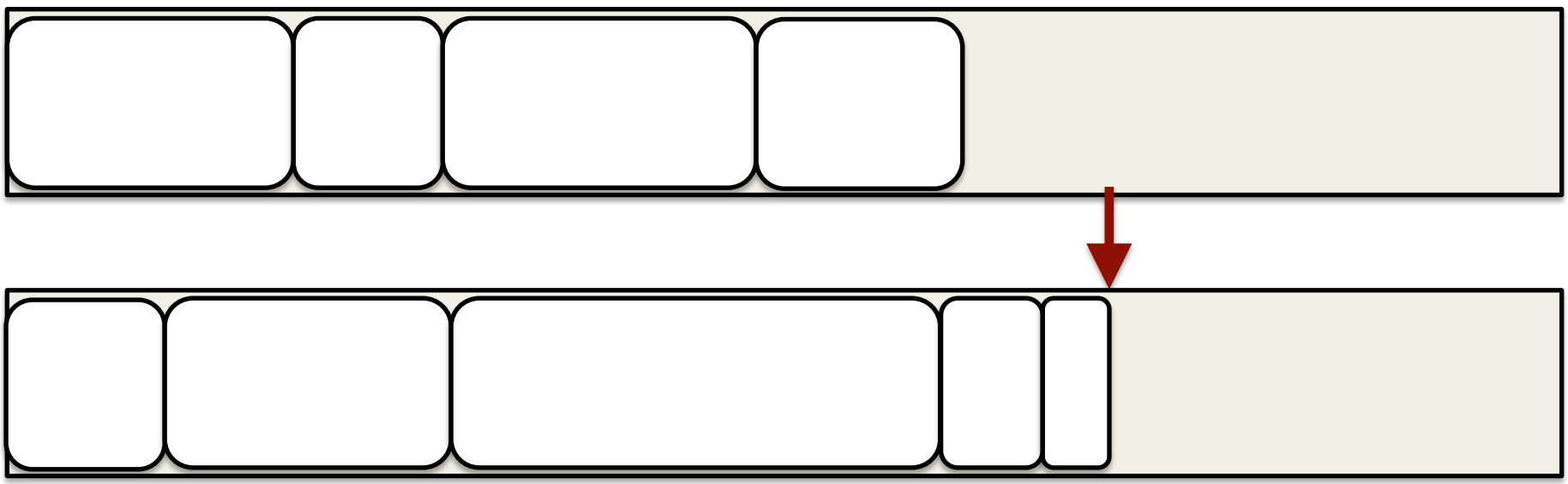


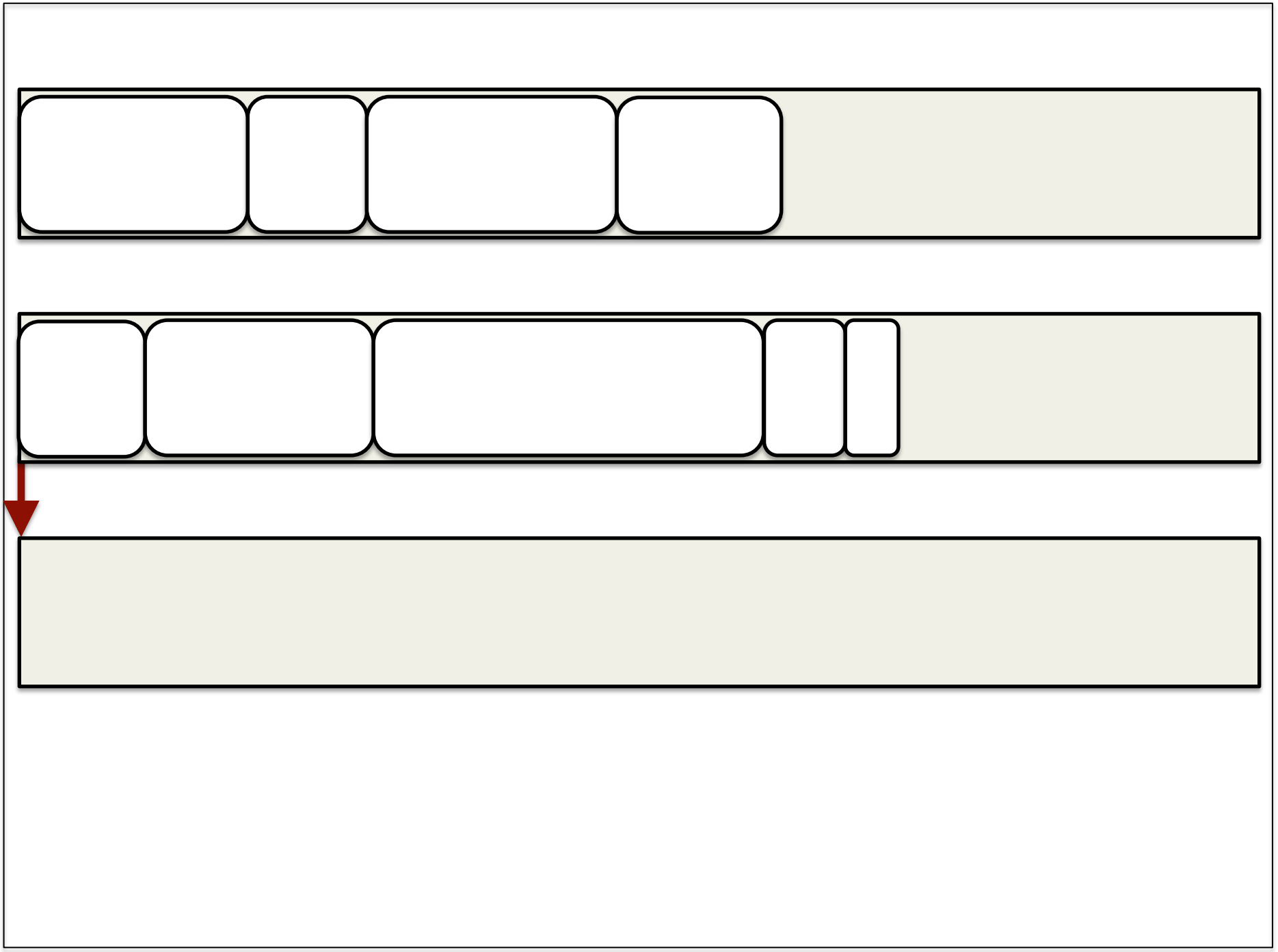


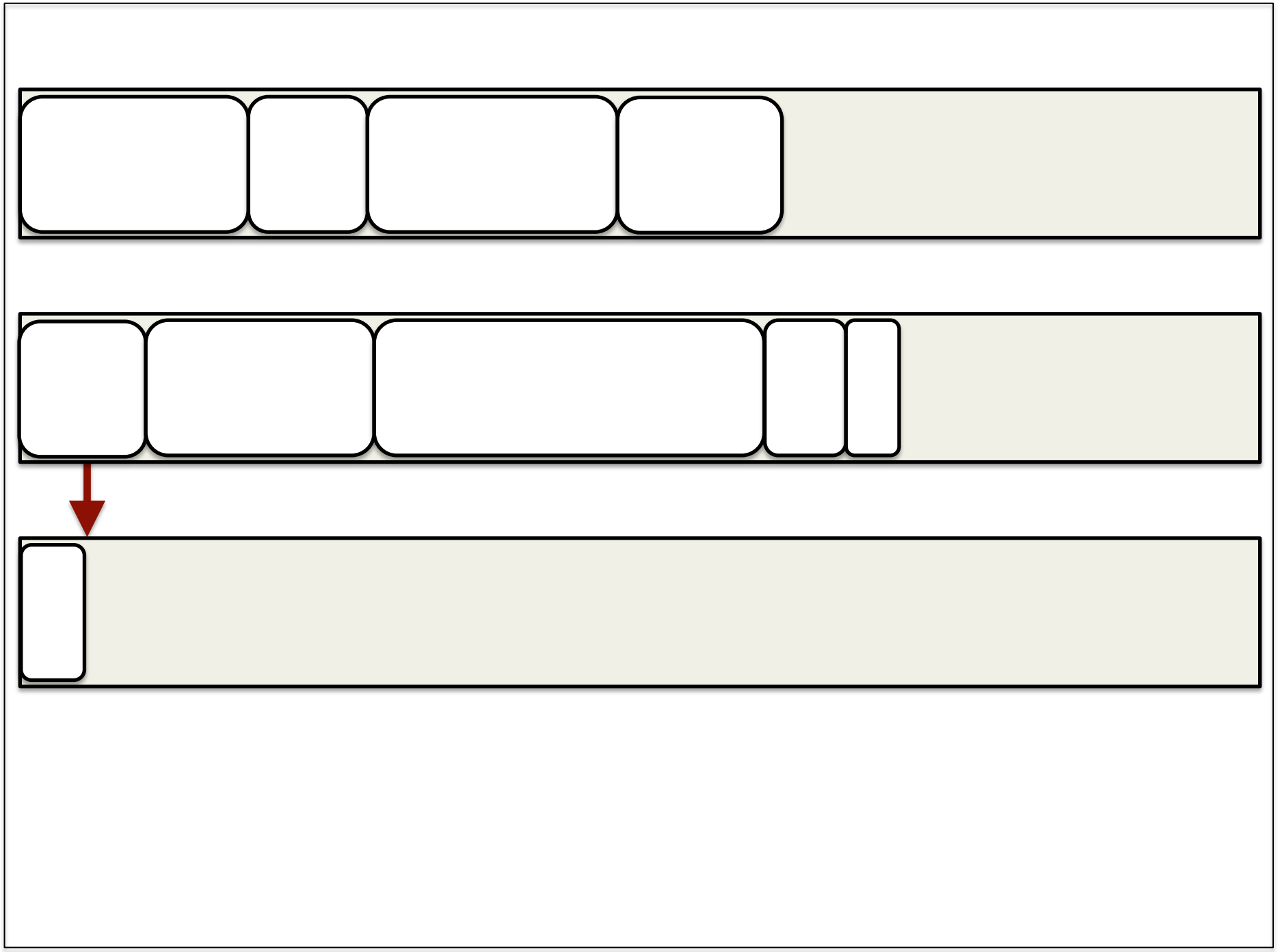


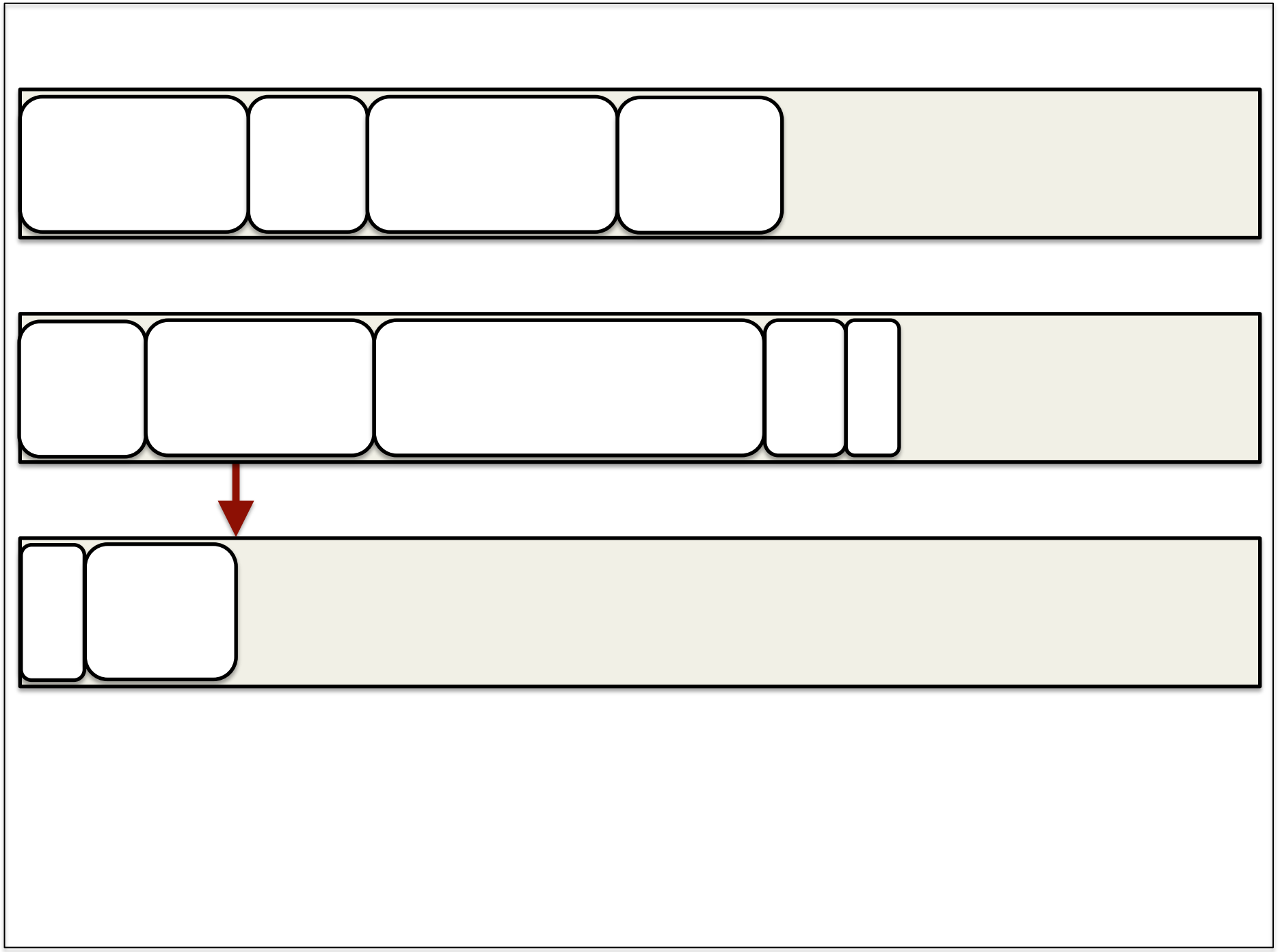


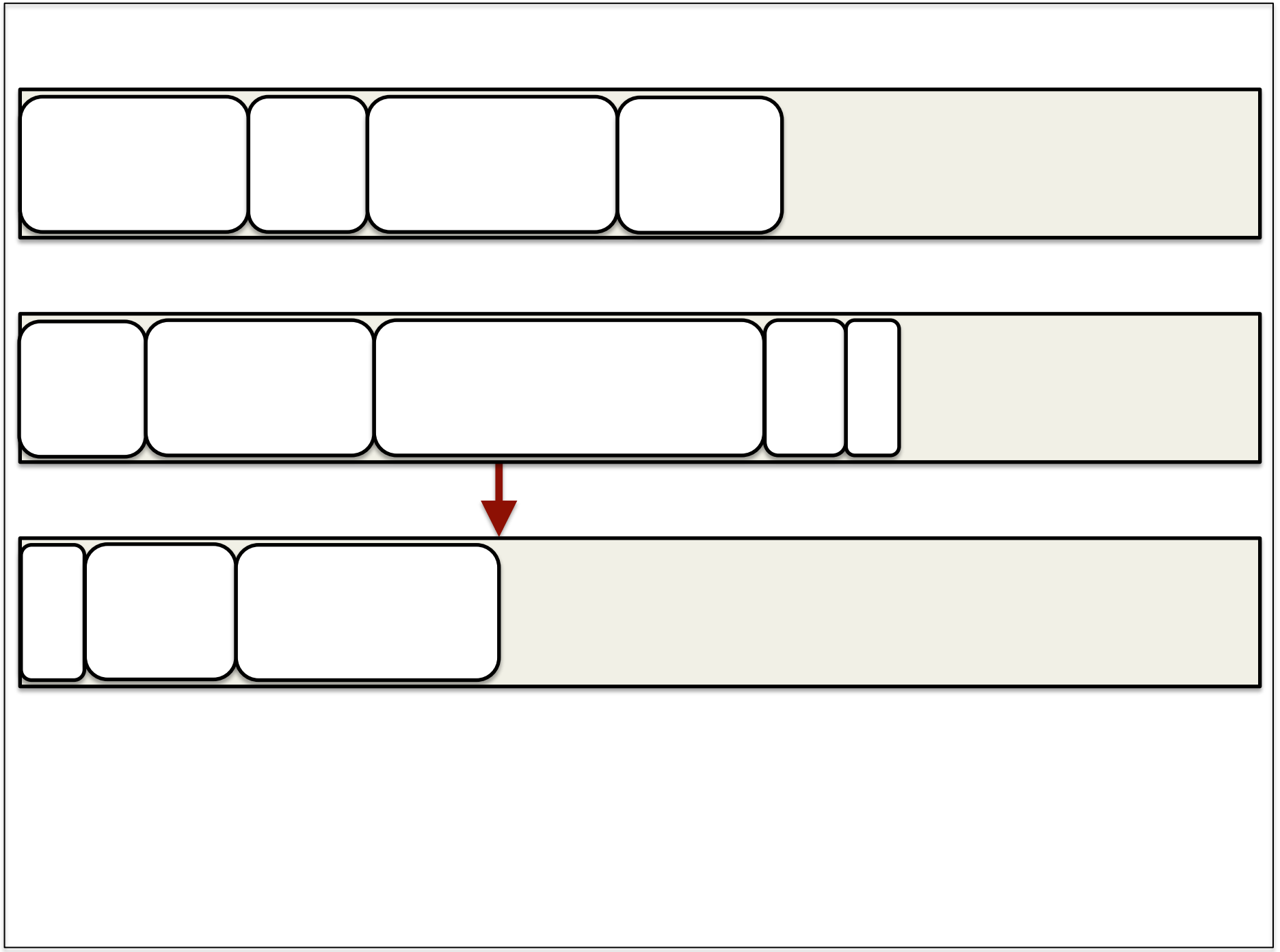


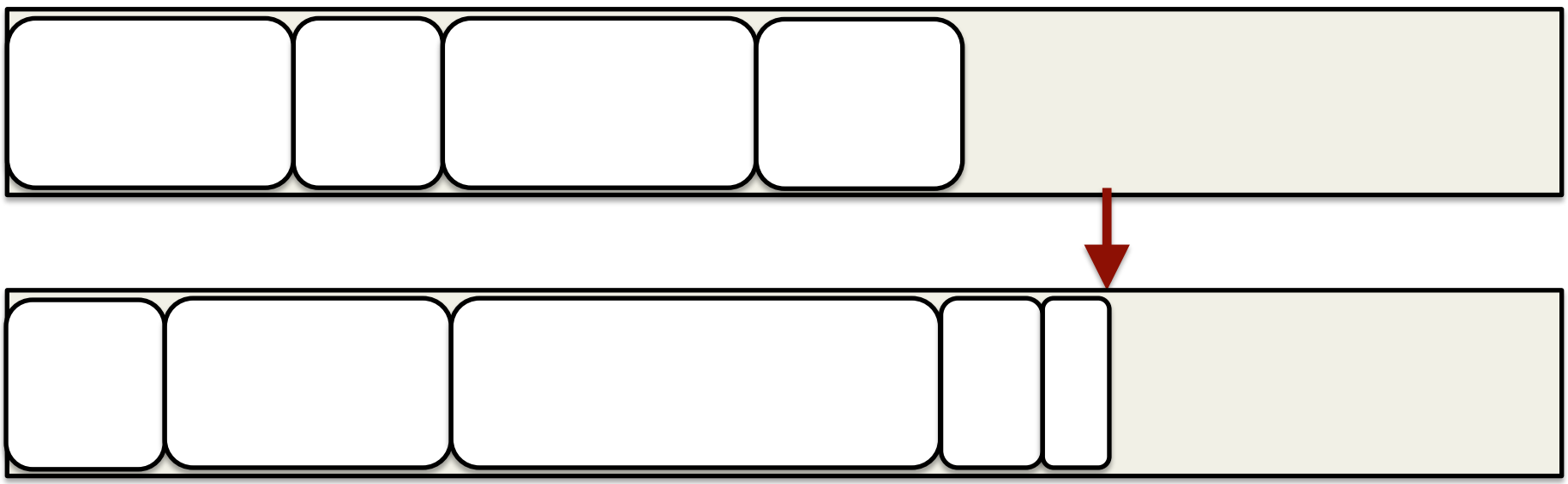


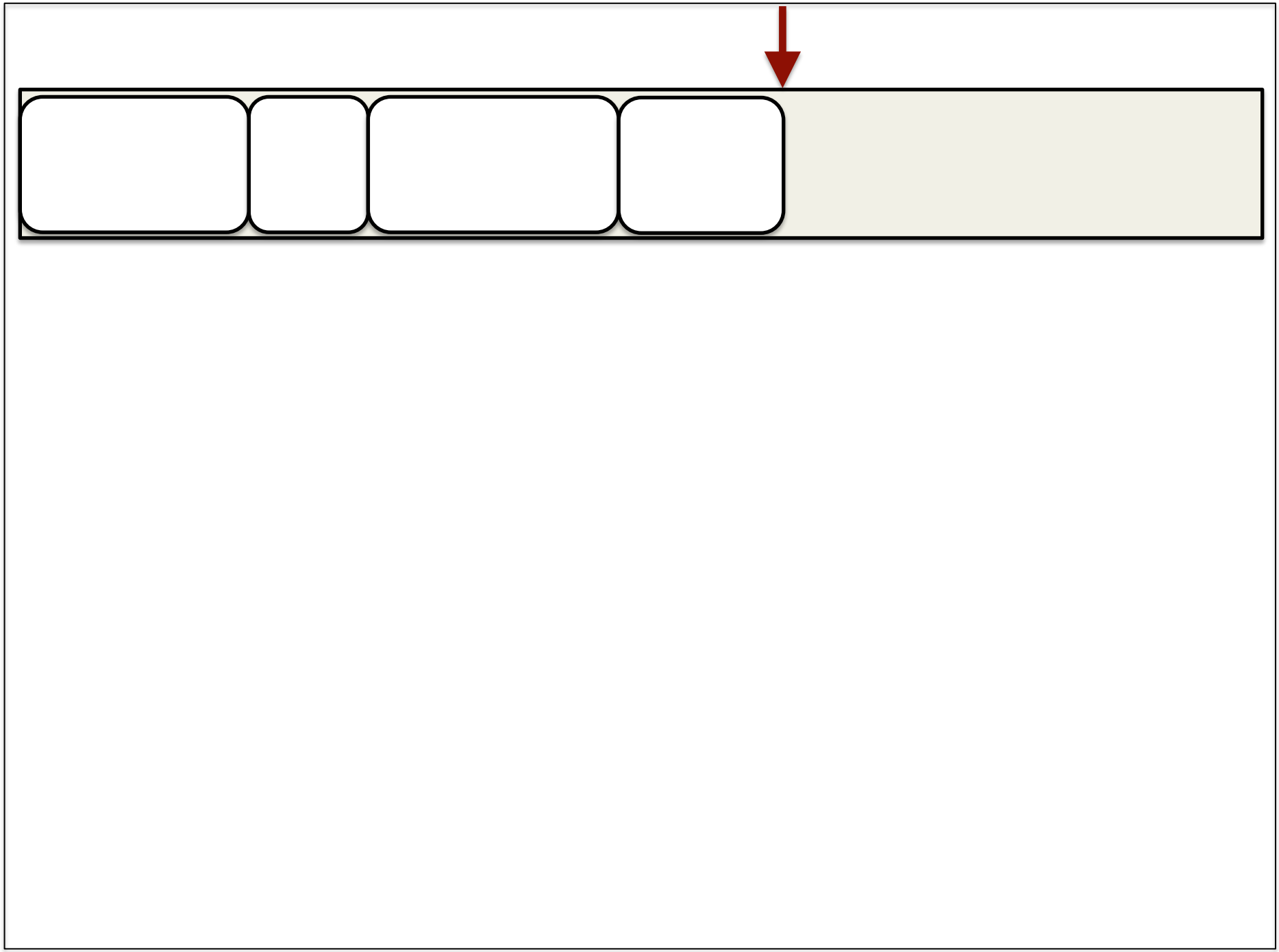


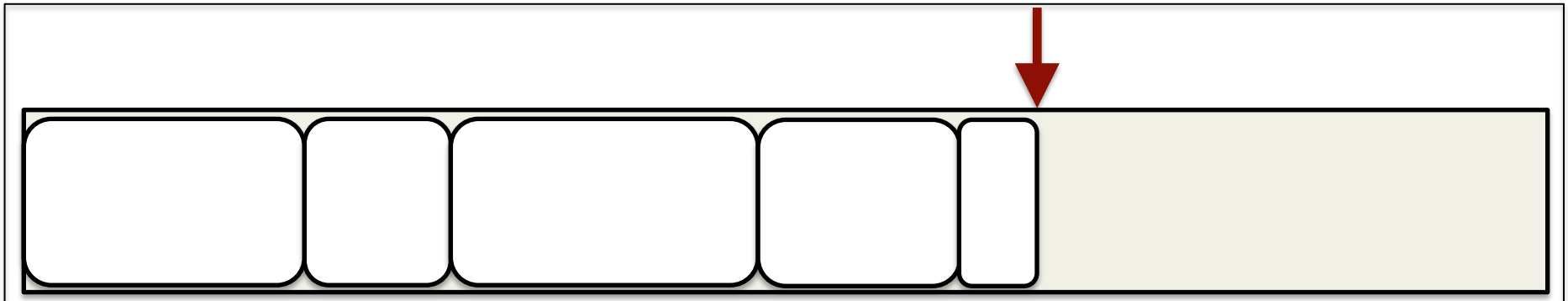












Scoped Memory Advantages

- Simple to understand
  - Basically how C/C++ stack allocation works
- Doesn't need runtime support
- Very efficient in the right circumstances
  - Very little allocation overhead
  - Deallocation happens at the region level

Scoped Memory Drawbacks

- Need to structure your code around it
- Inter-region references can only go to outer scope
  - Old to new pointers could be broken
  - Verification can be expensive
- Need to predict the size of regions
- Need a custom allocator
  - The system allocates at the region level
  - Custom allocator divides up that space

Mature Space Management

- We know how the nursery is managed
  - Not much room for variation
- The mature space is much more flexible

- Several drawbacks to mature space copying
  - Need to maintain a copy reserve
  - May end up copying objects frequently

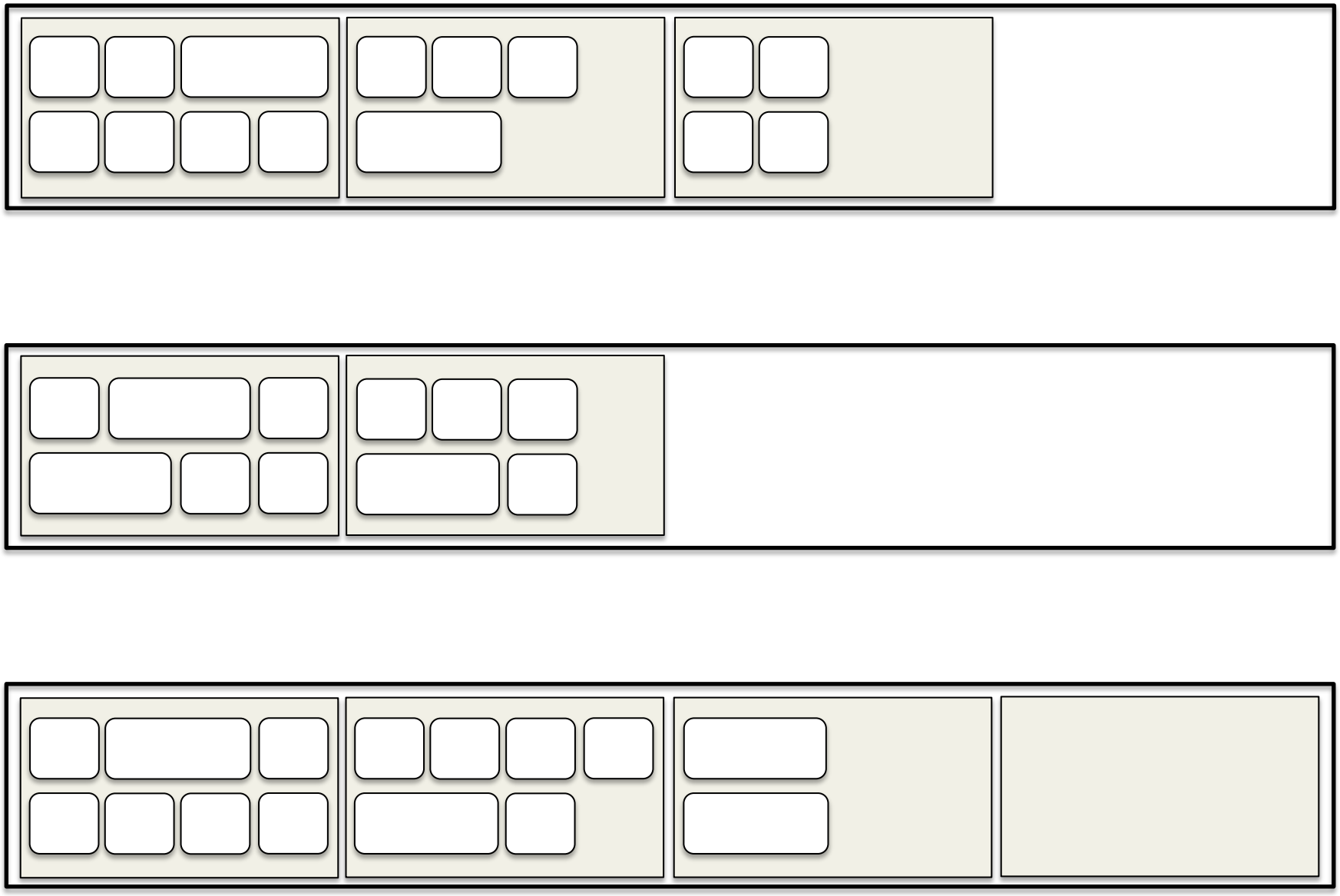
- We can cut down on this by using small regions

The Train Algorithm

- Also know as the Mature Object Space (MOS)
  - Focuses only on managing the mature space
  - The nursery works as we've seen already
- Never collects the entire heap
  - Splits the heap into many small regions
  - Do copy collection on one region at a time
- Copy reserve is no larger than the region size

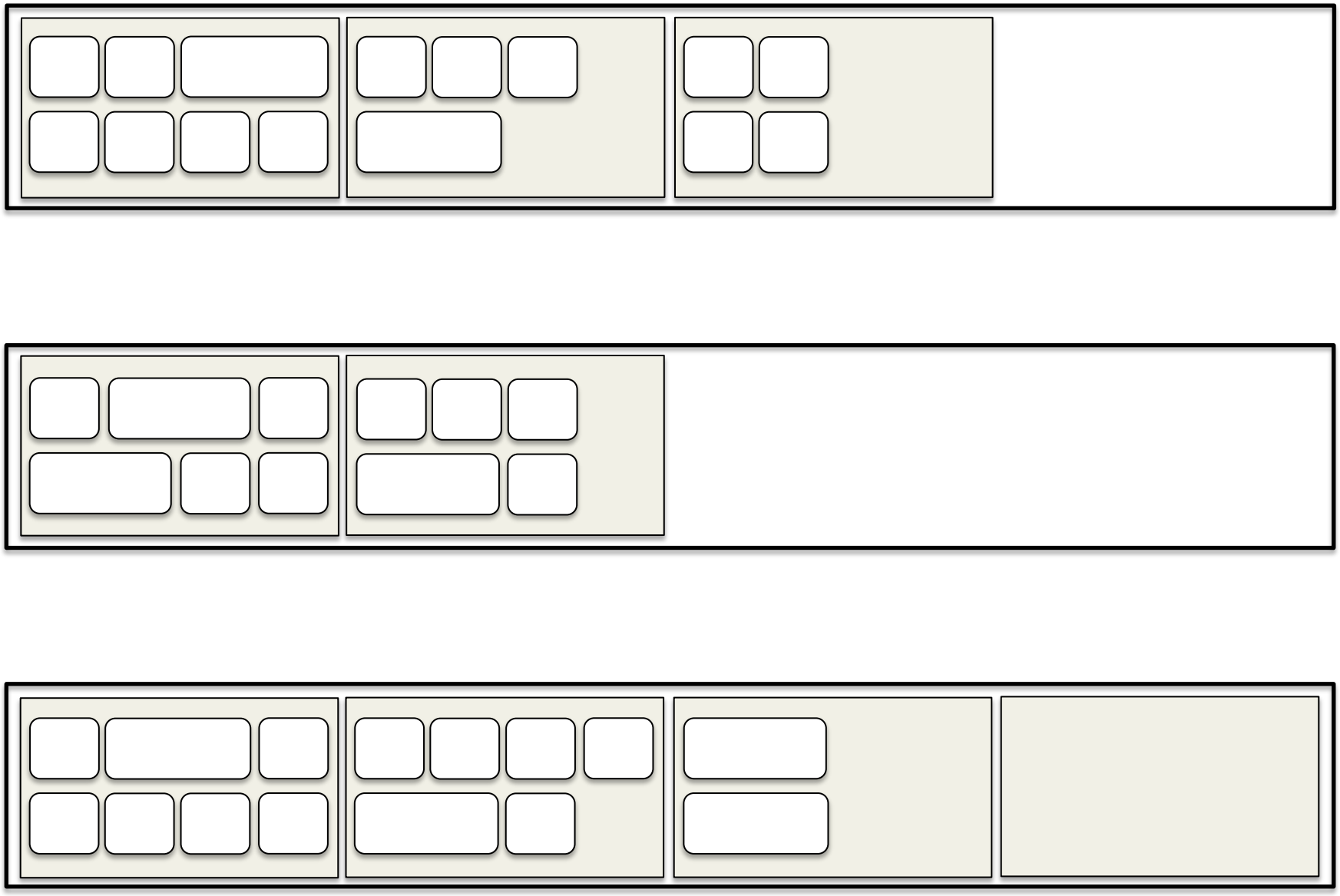
The Train Algorithm

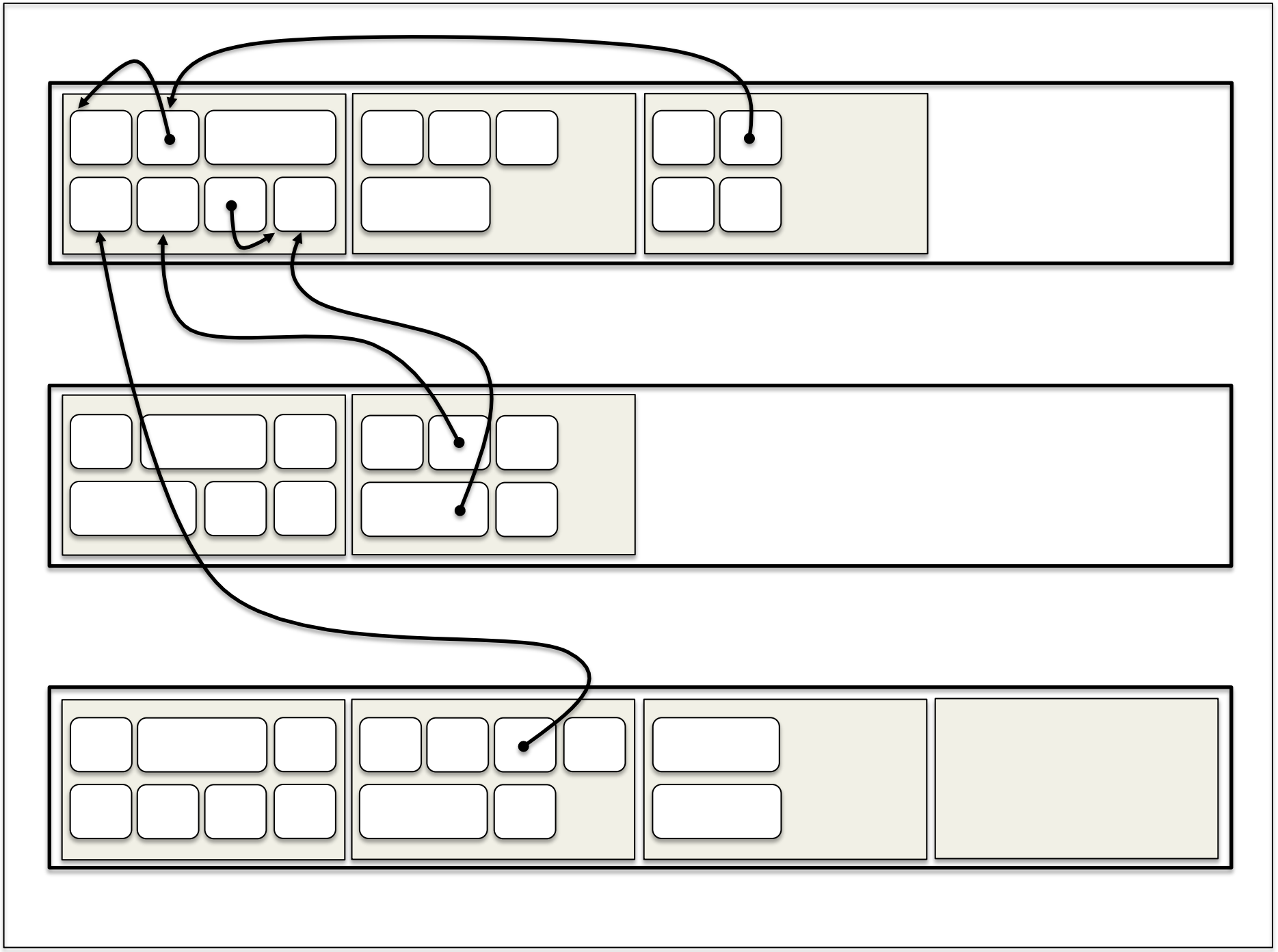
- Cars are fixed-sized regions
  - Size depends on architecture and tuning
  - Aligning on powers of two make barriers cheaper
- Trains are made up of one or more cars
  - No limit to train size
  - No limit to the number of trains
- Trains and cars have a natural ordering

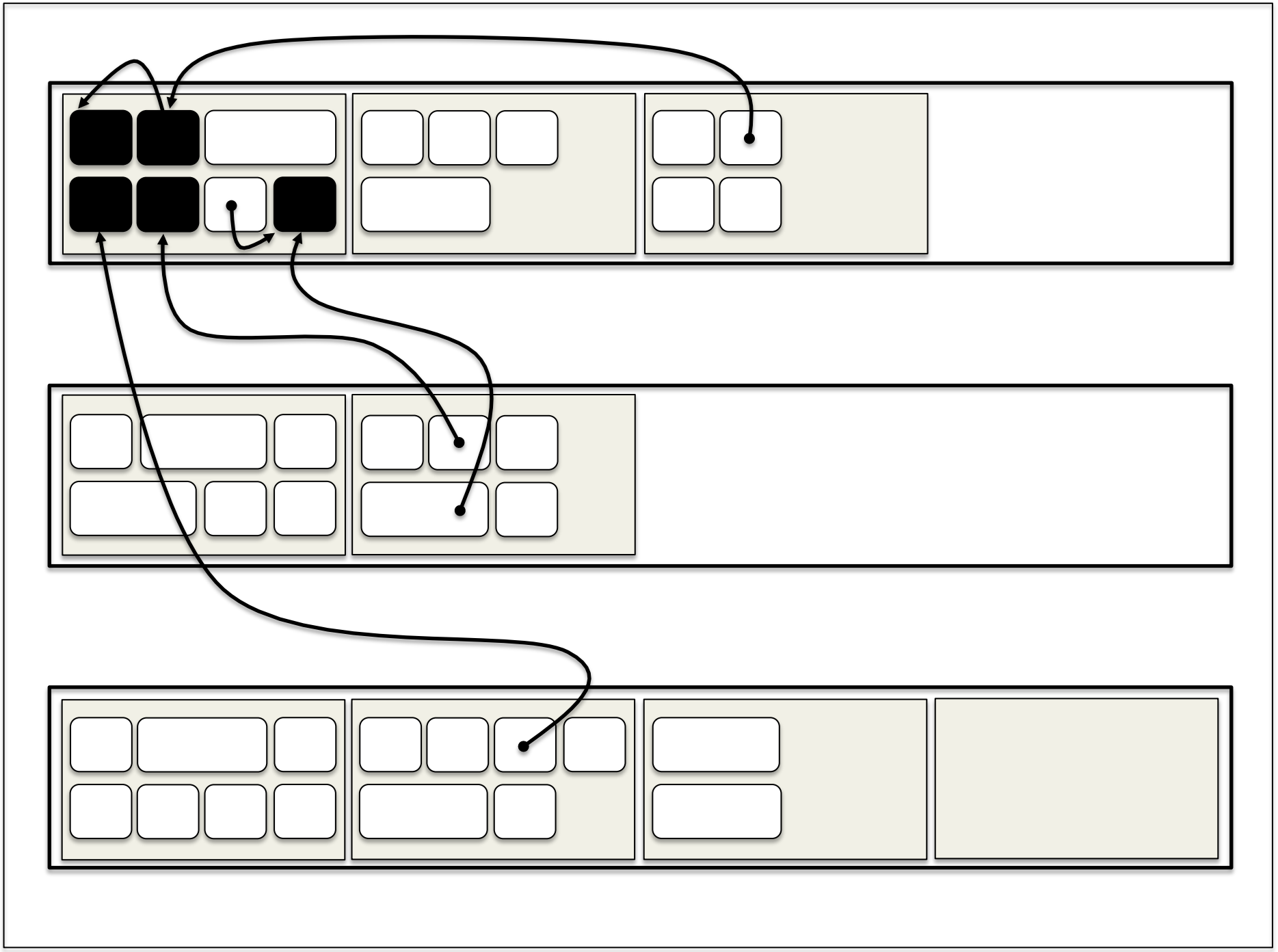


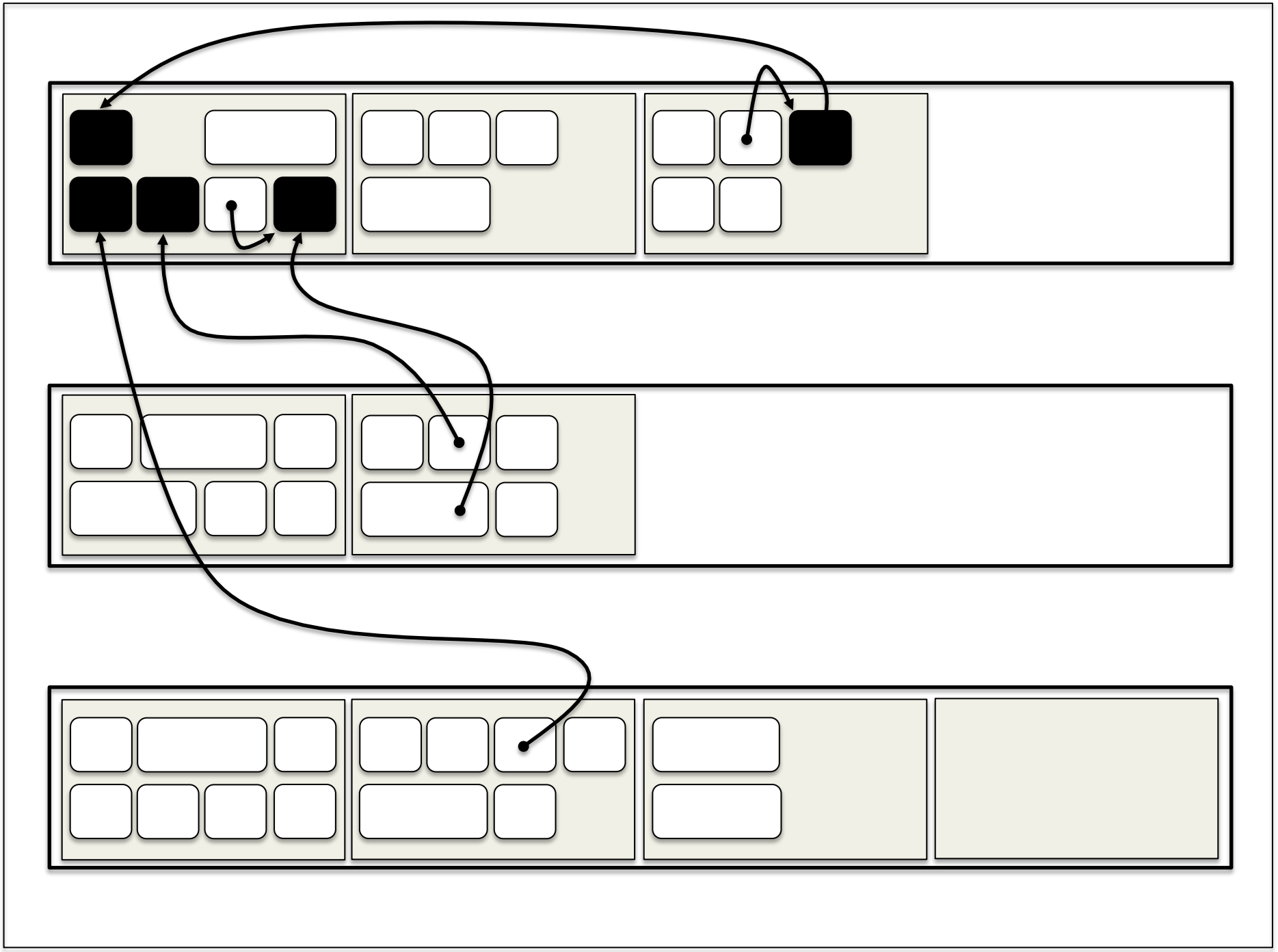
Remembered Set

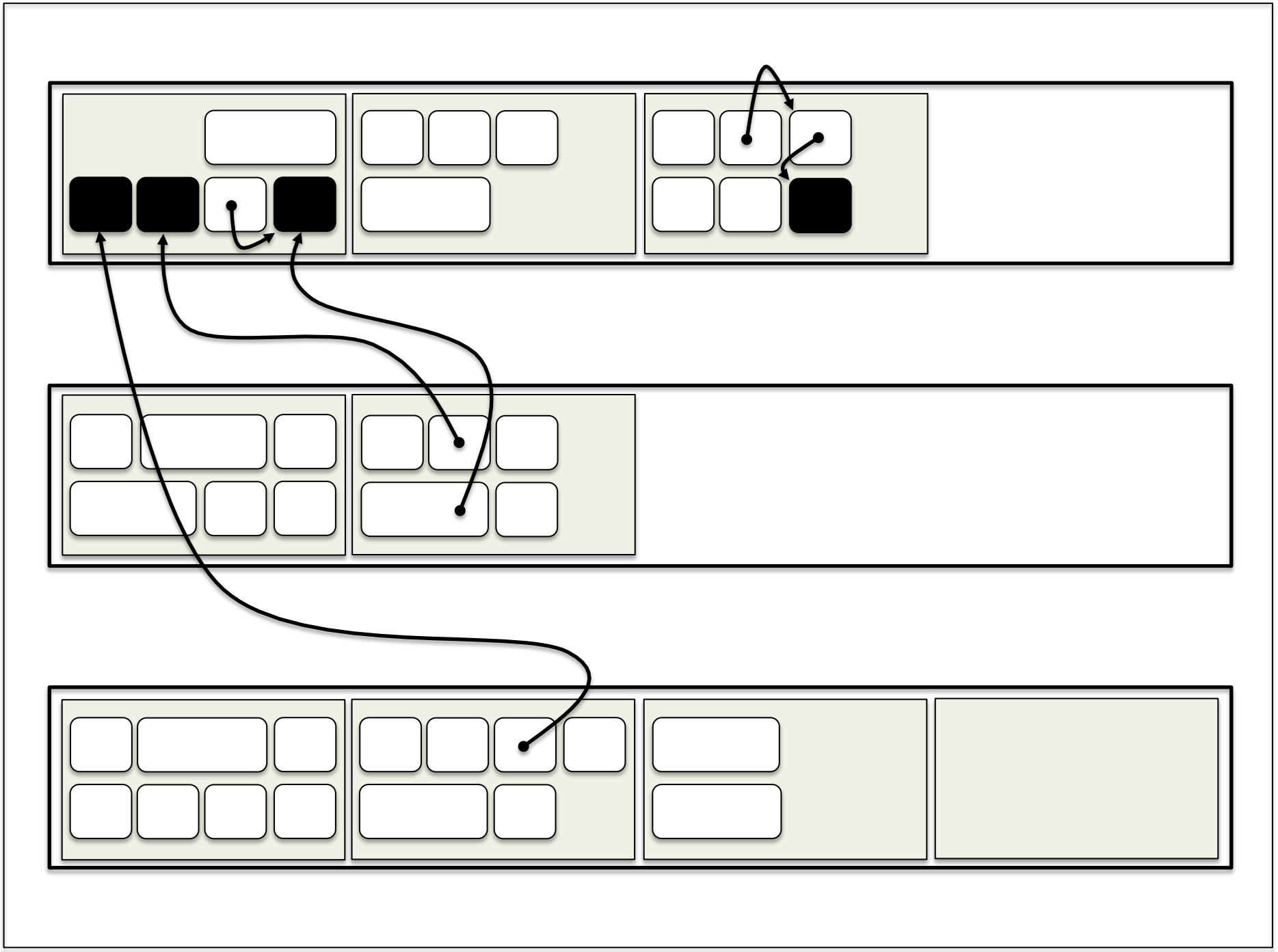
- Cars are collected individually
  - Don't want to scan the whole heap
  - Need to remember references into the car
- Each car has a remembered set
  - Indicates references to the car from outside
  - Interior references are not tracked
- Barrier more complex than for plain generational
  - Can reduce cost using some of the earlier techniques

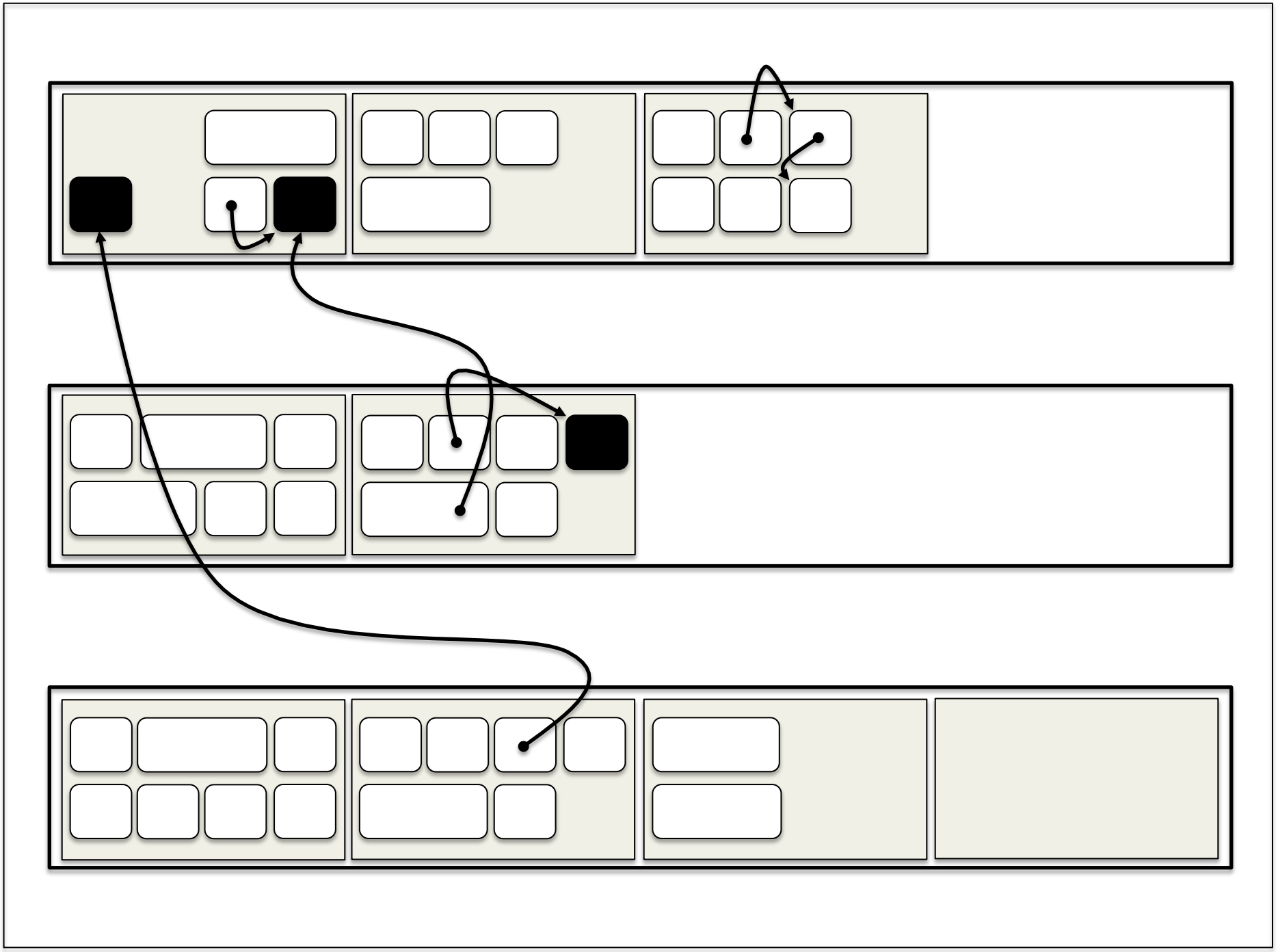


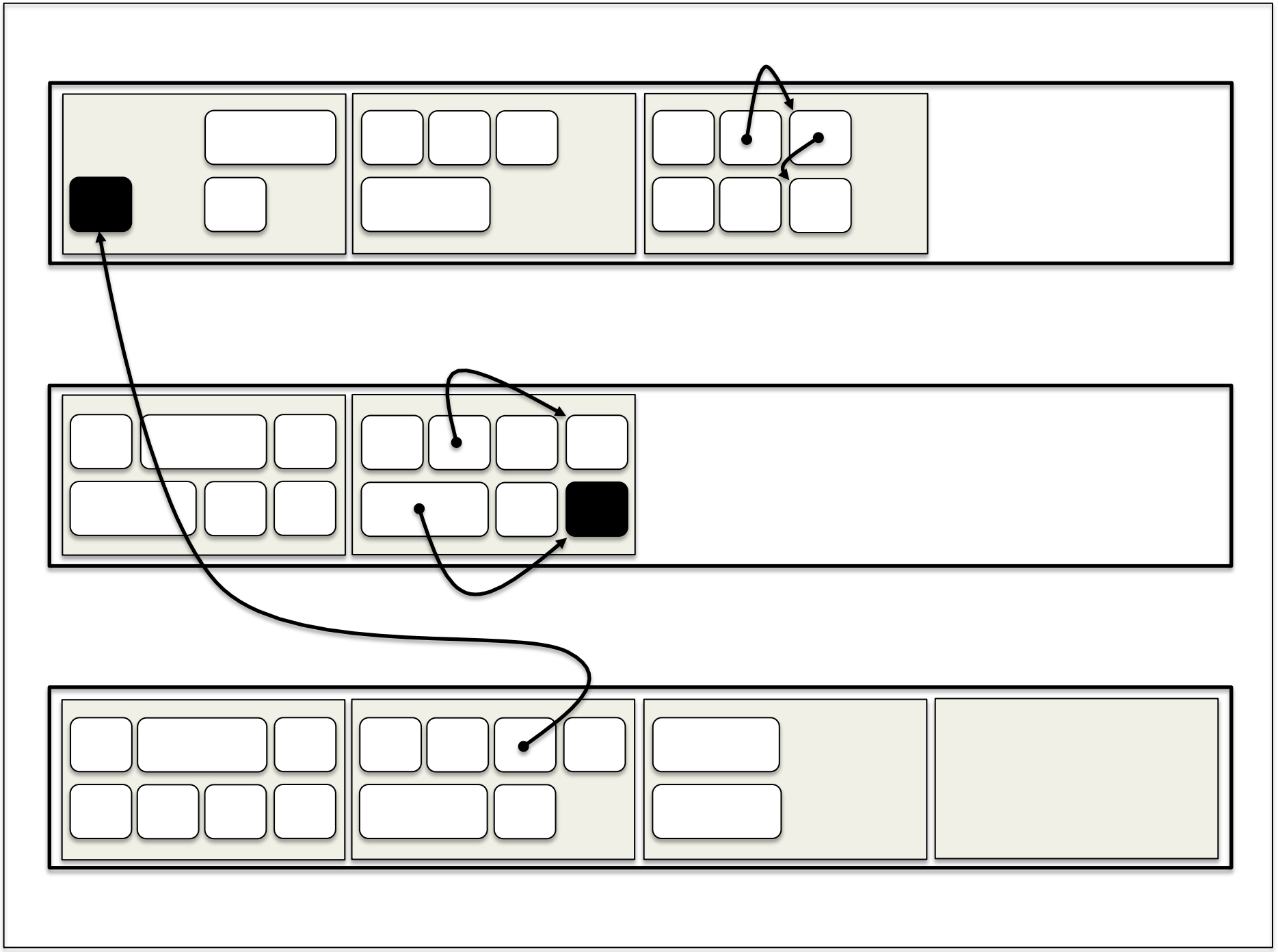


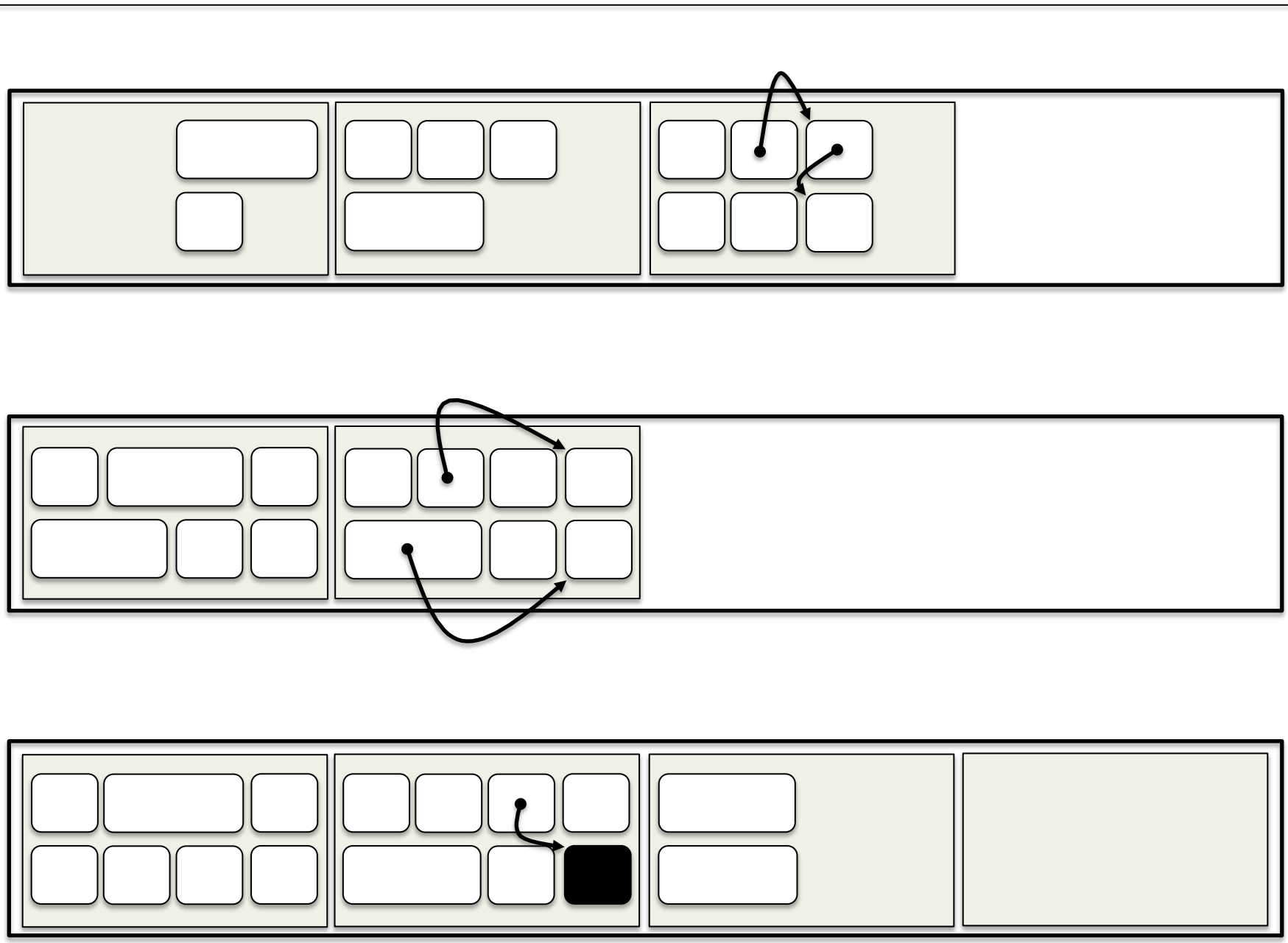


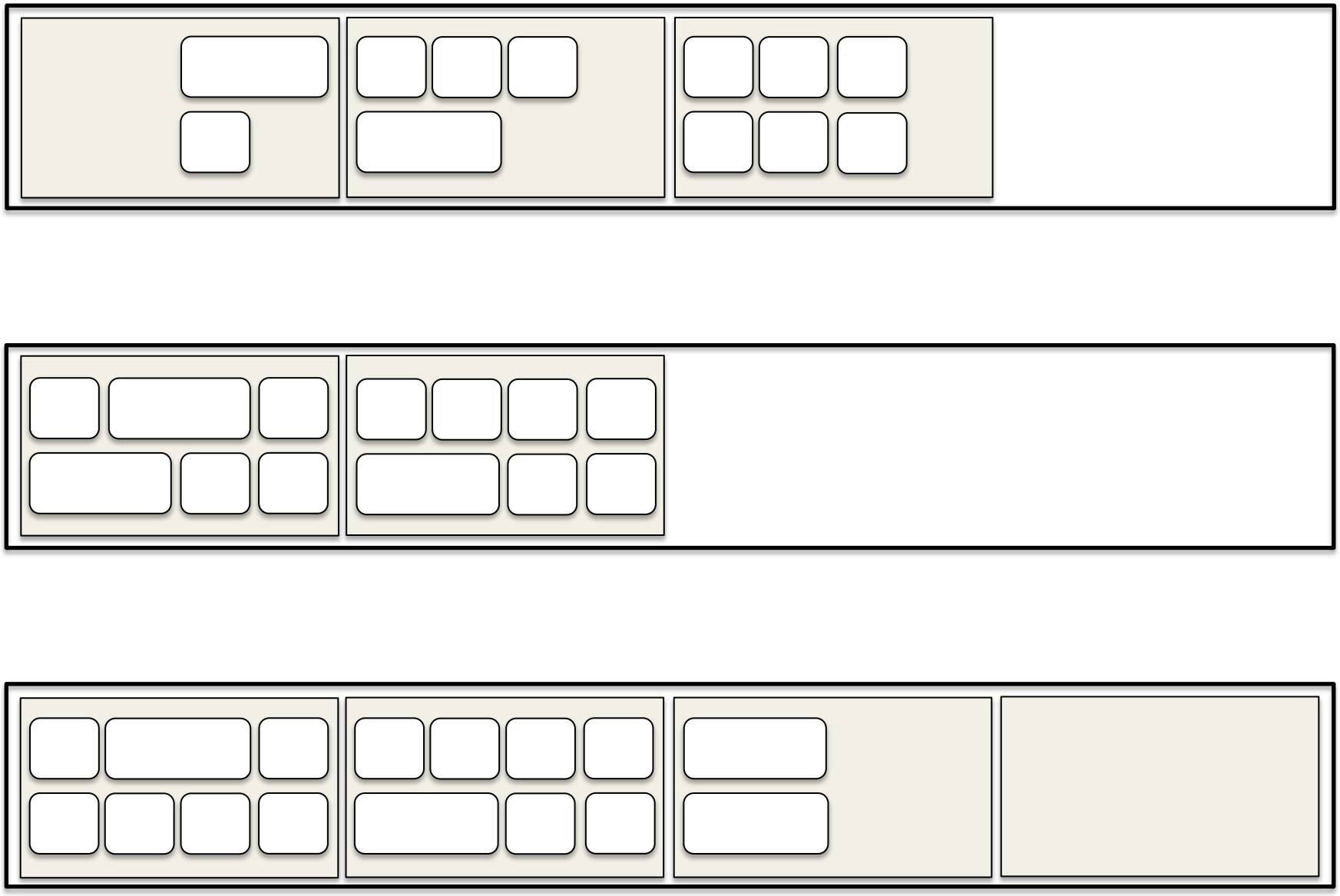


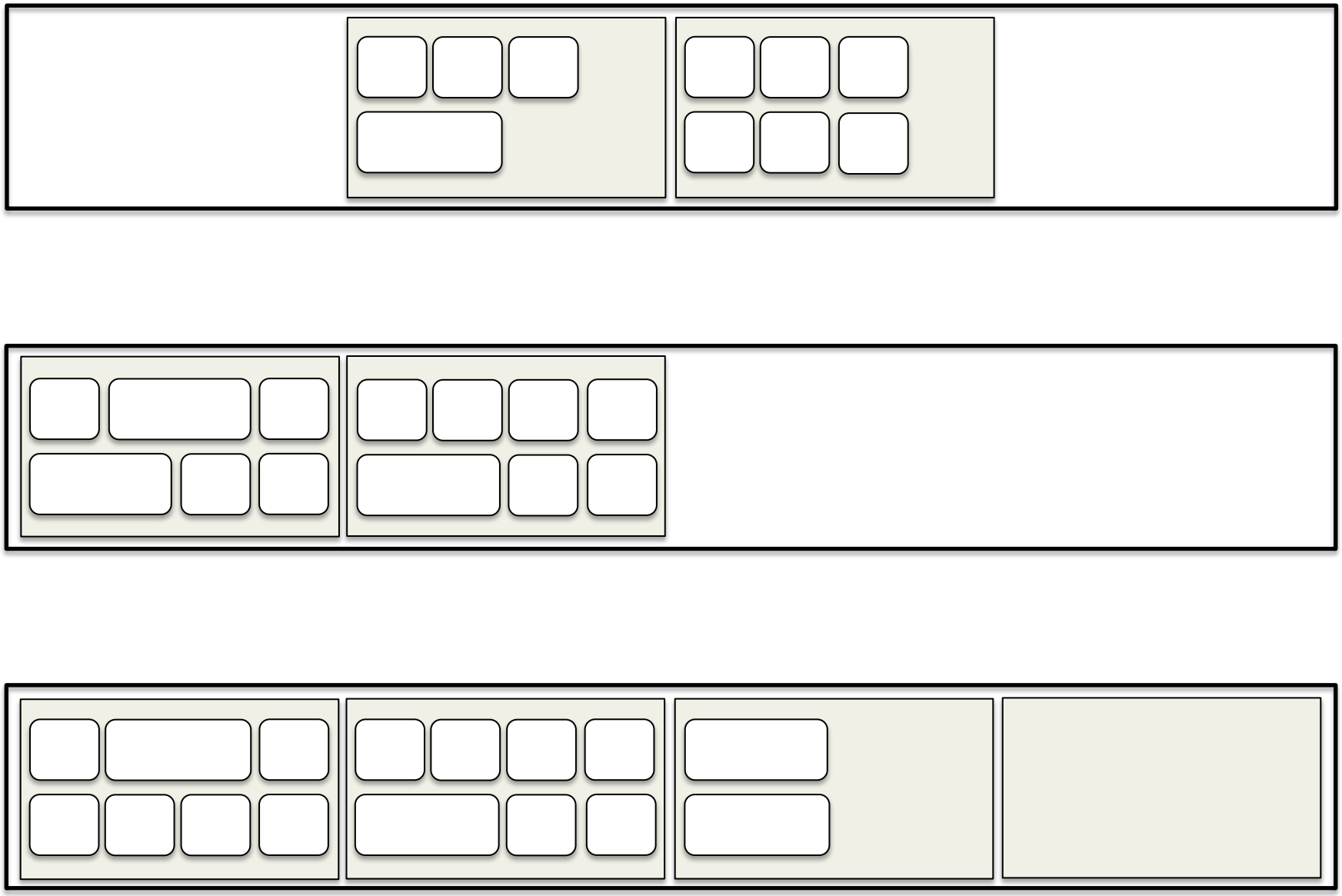


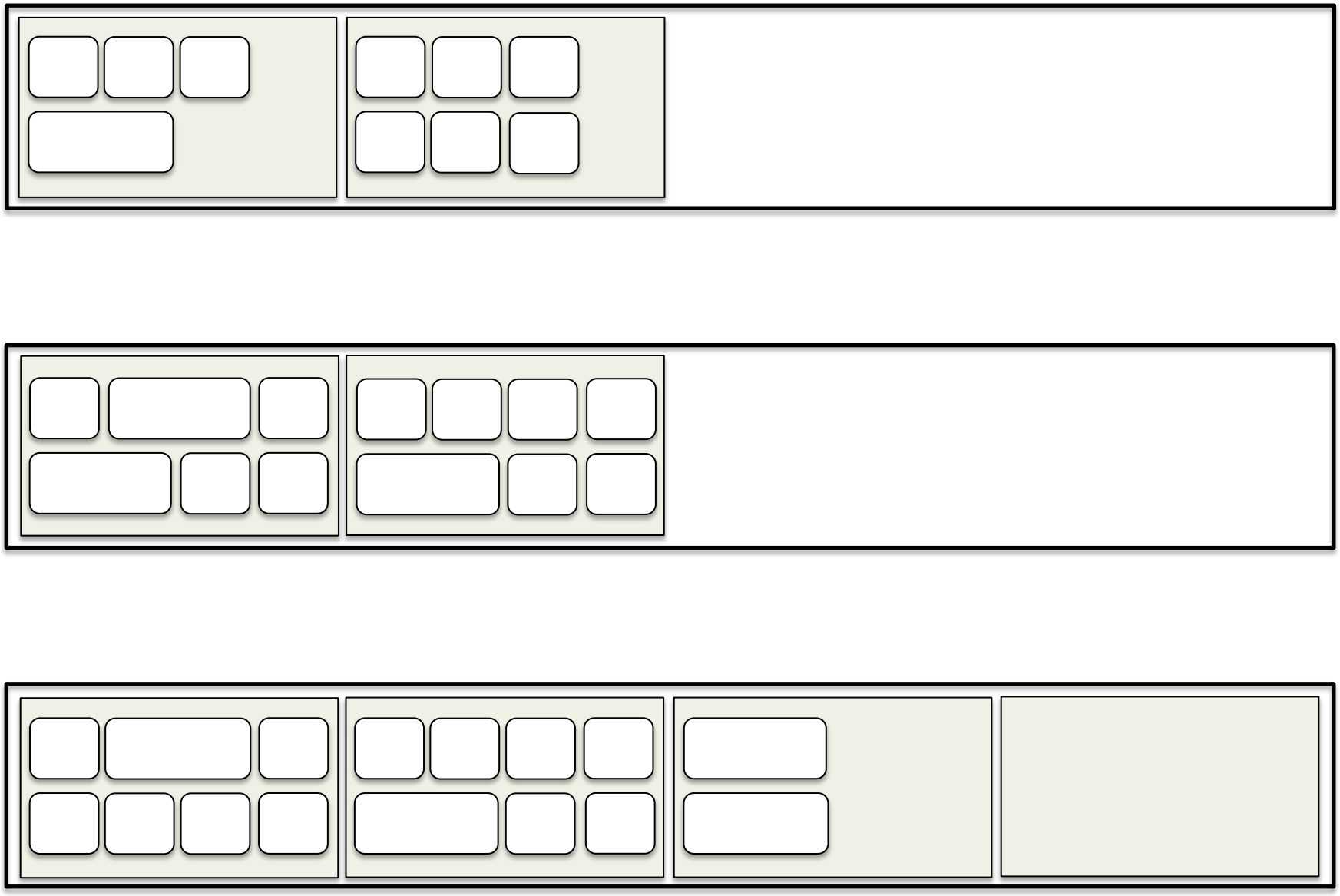






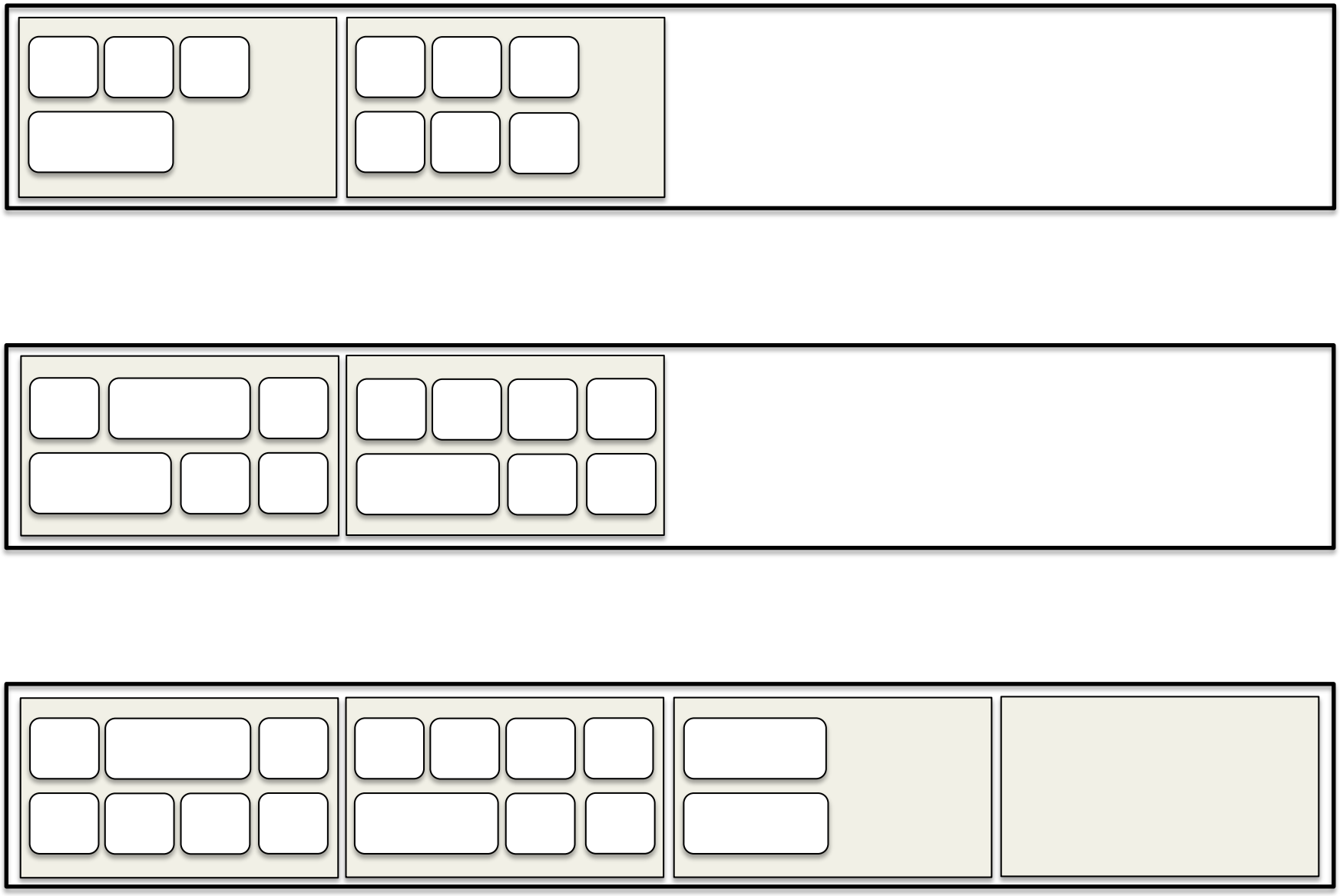


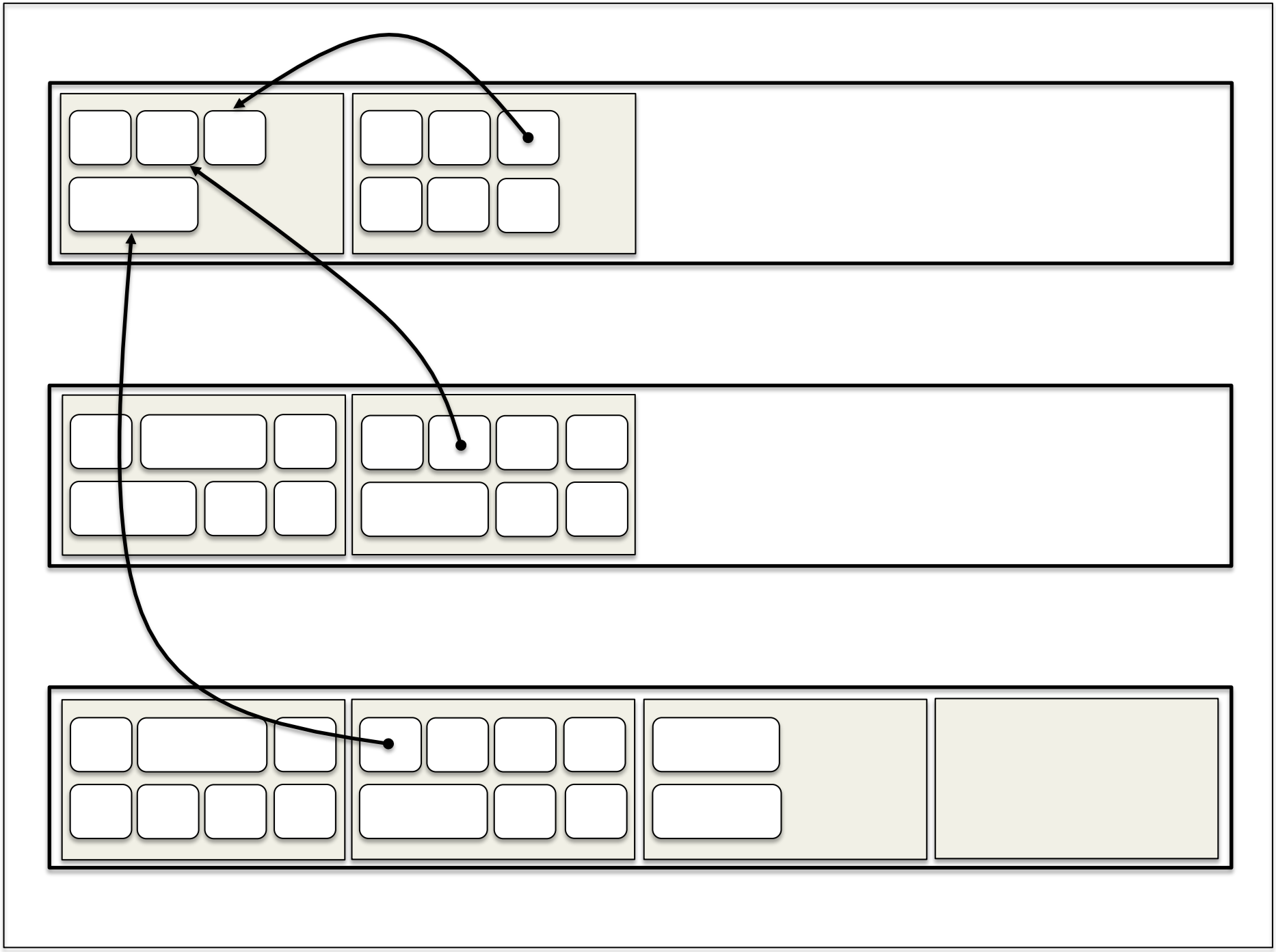


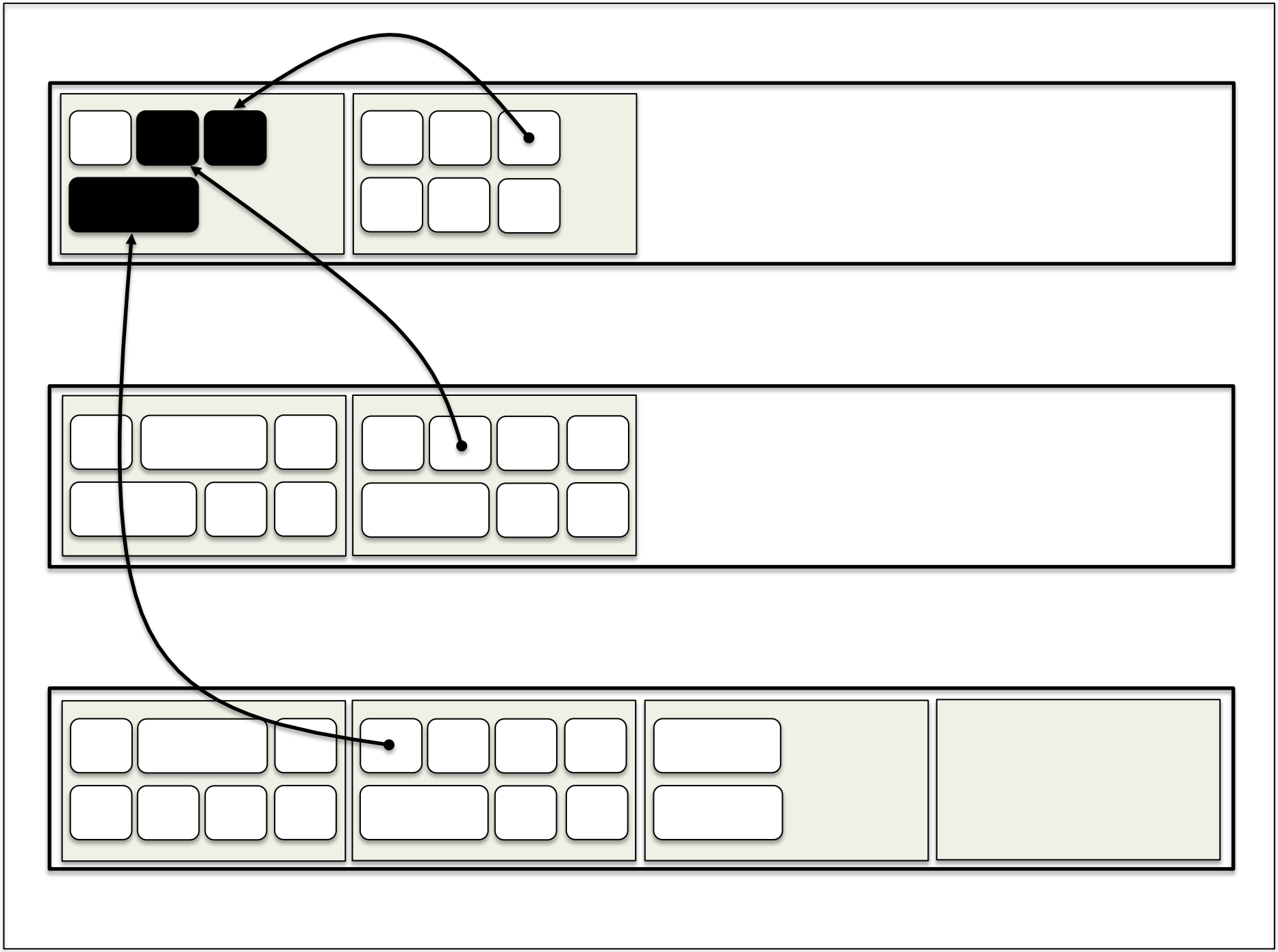


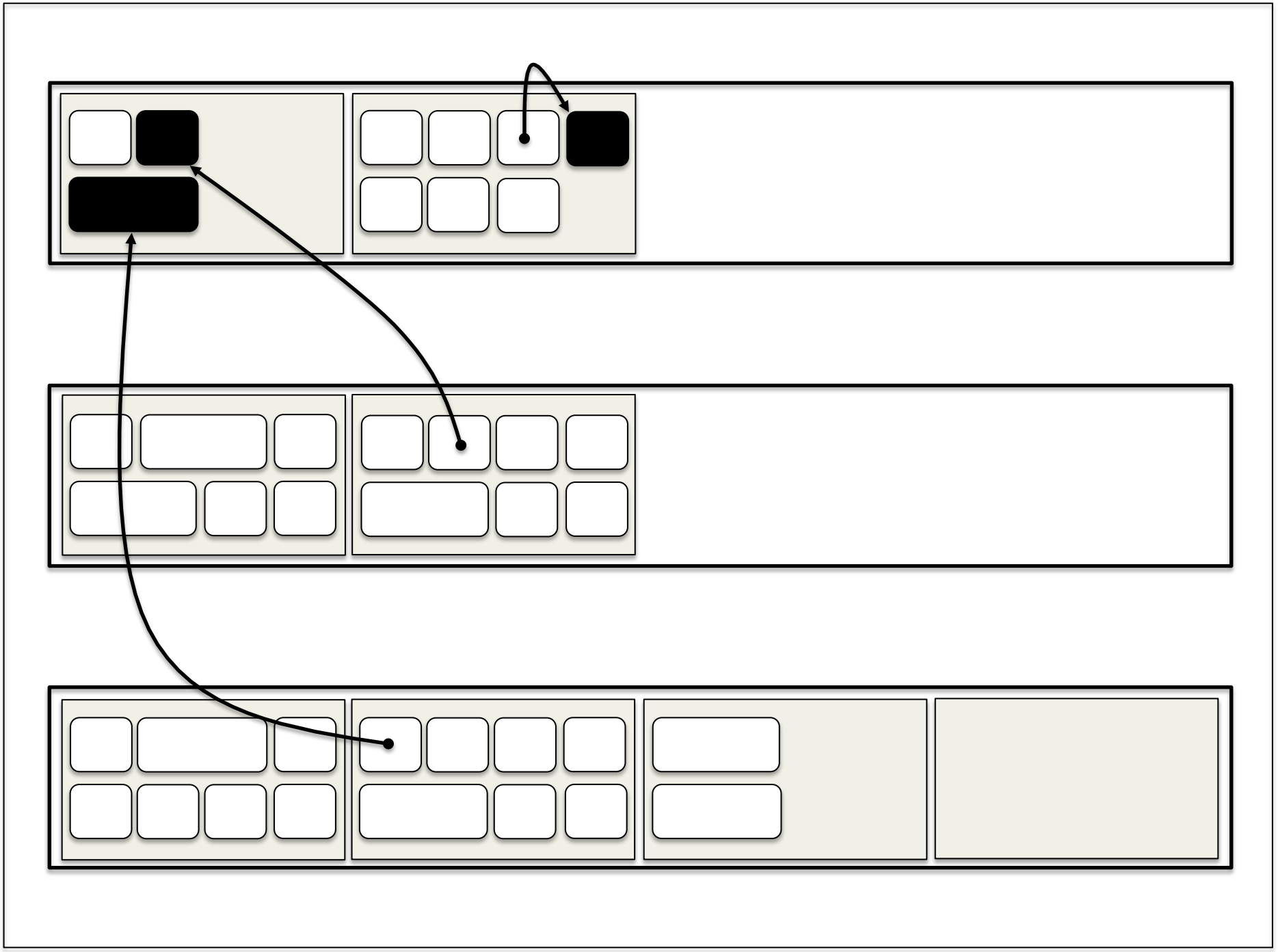
Evacuating Objects

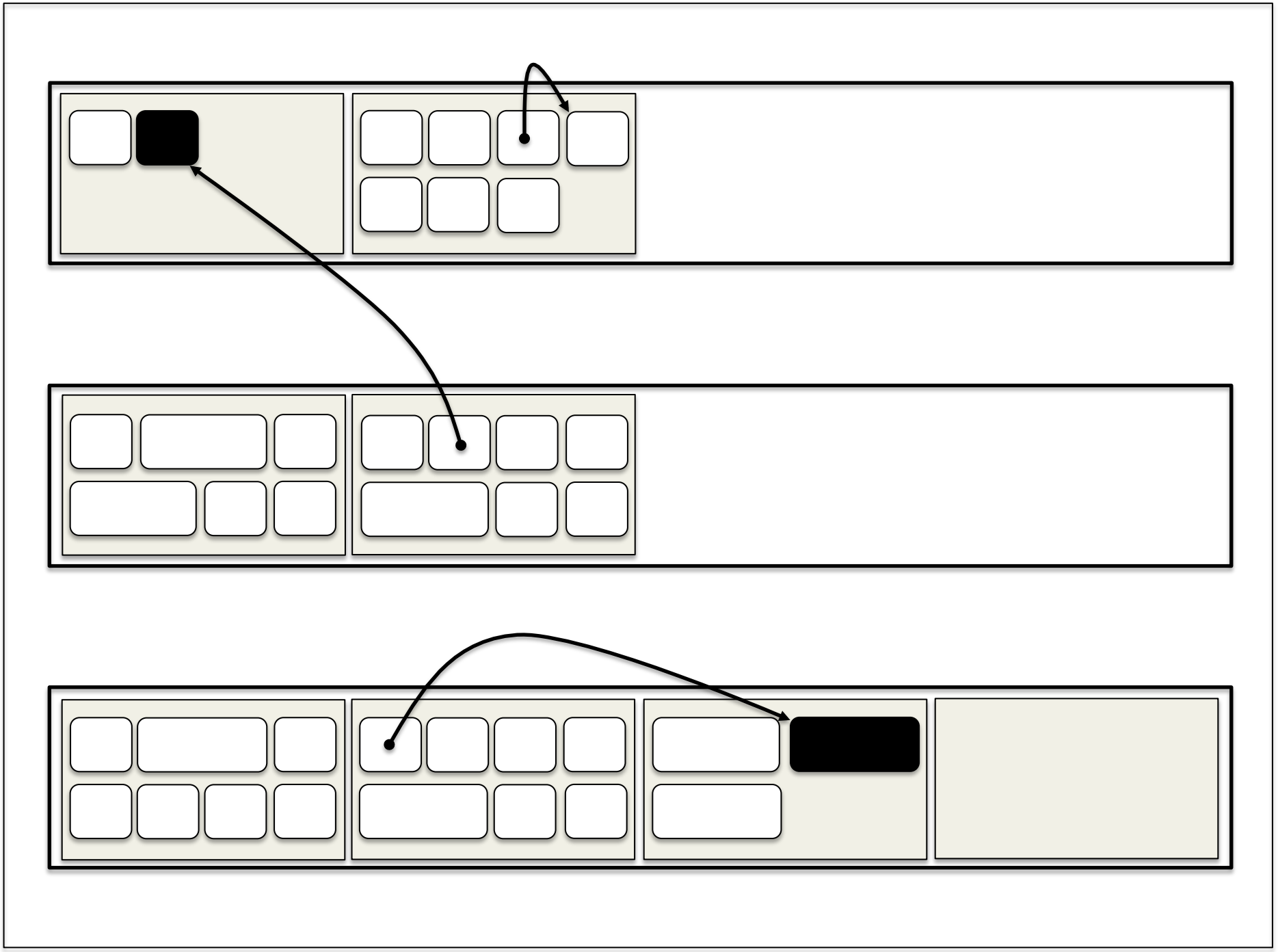
- Two goals for copying collection
  - Collect as much garbage as possible
  - Move referenced objects close together
- We can optimize by choosing where to copy
  - If object referenced from another train, move there
  - If not, move to another car in the same train
    - Try to move to a car that references the object

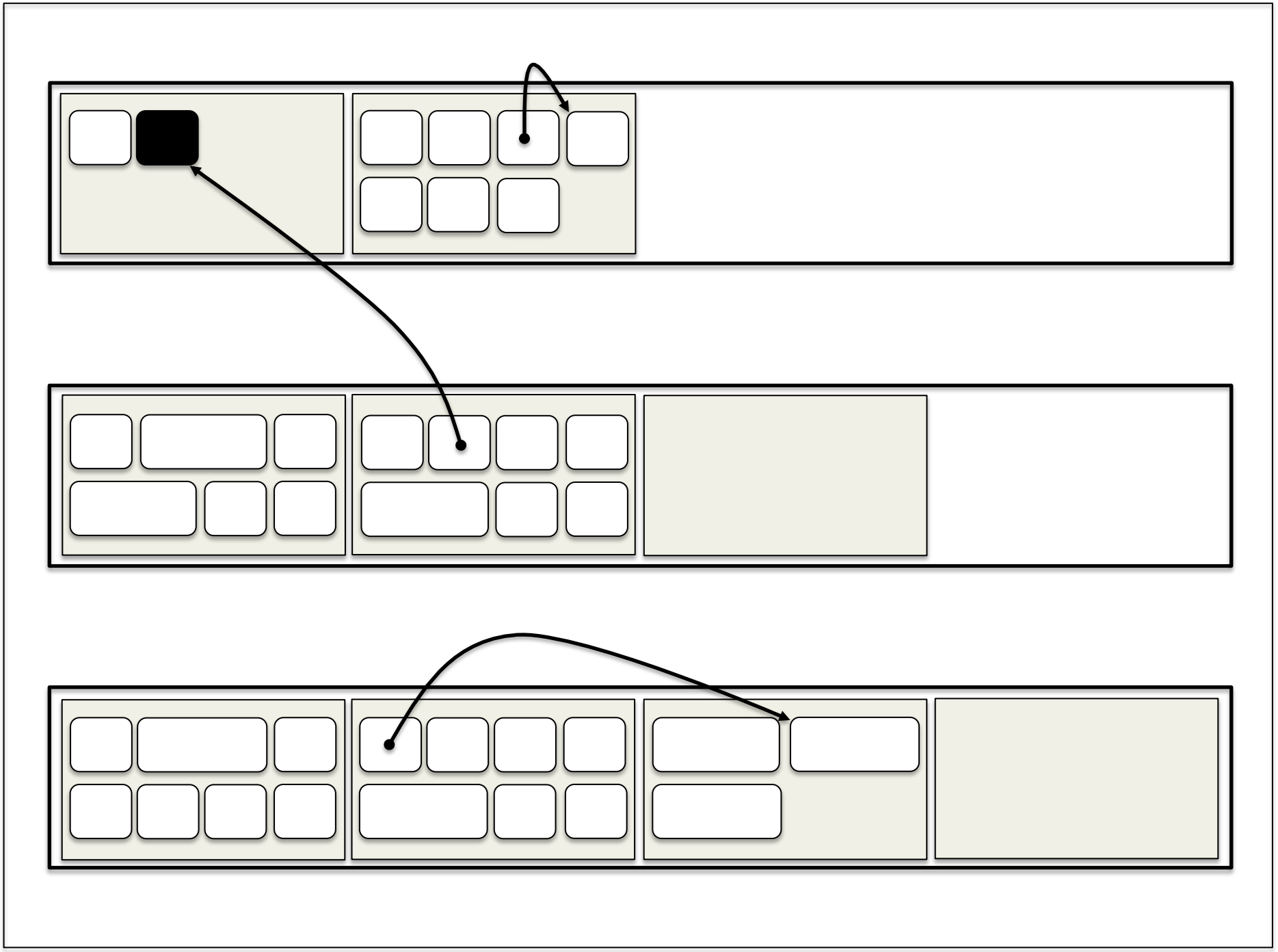


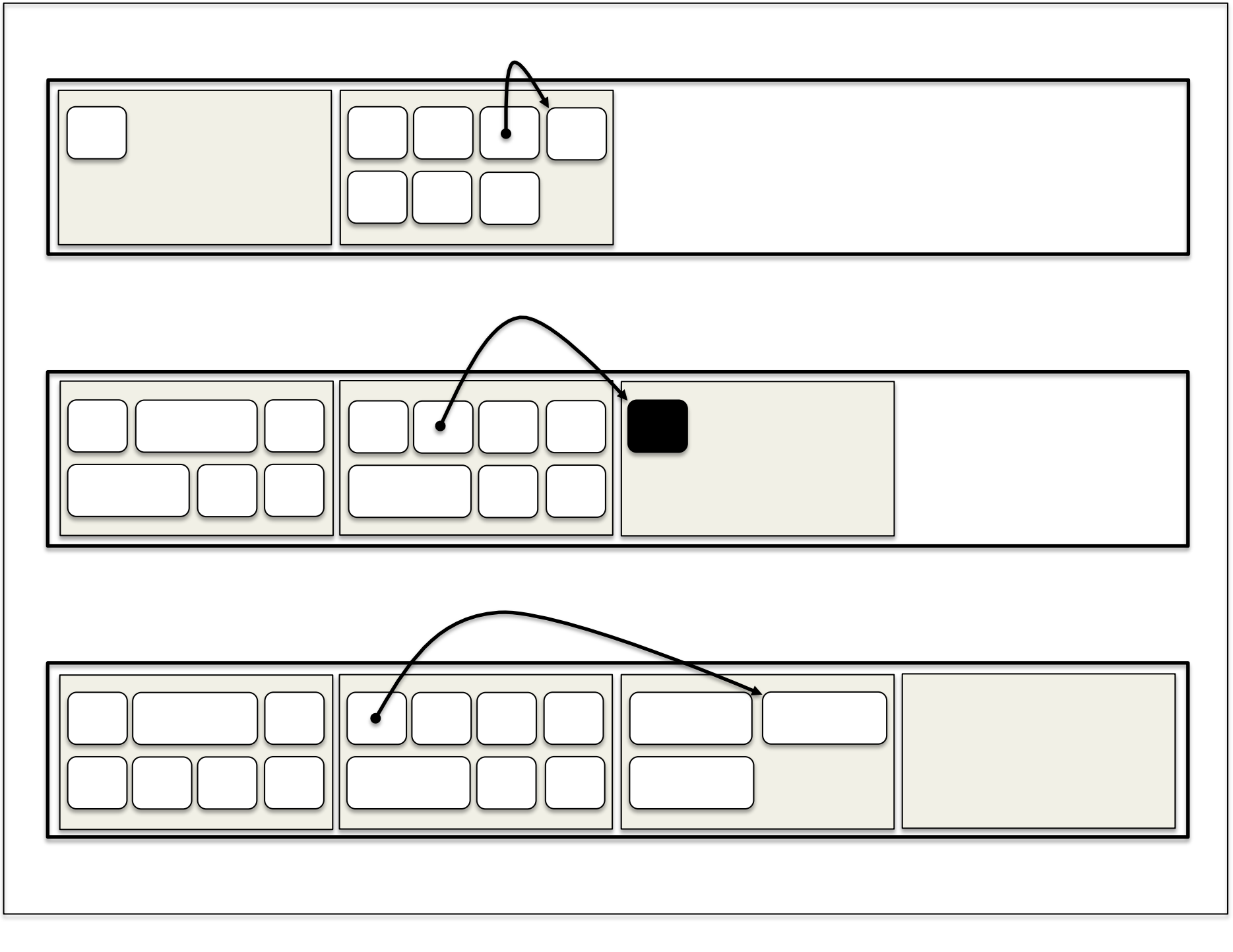


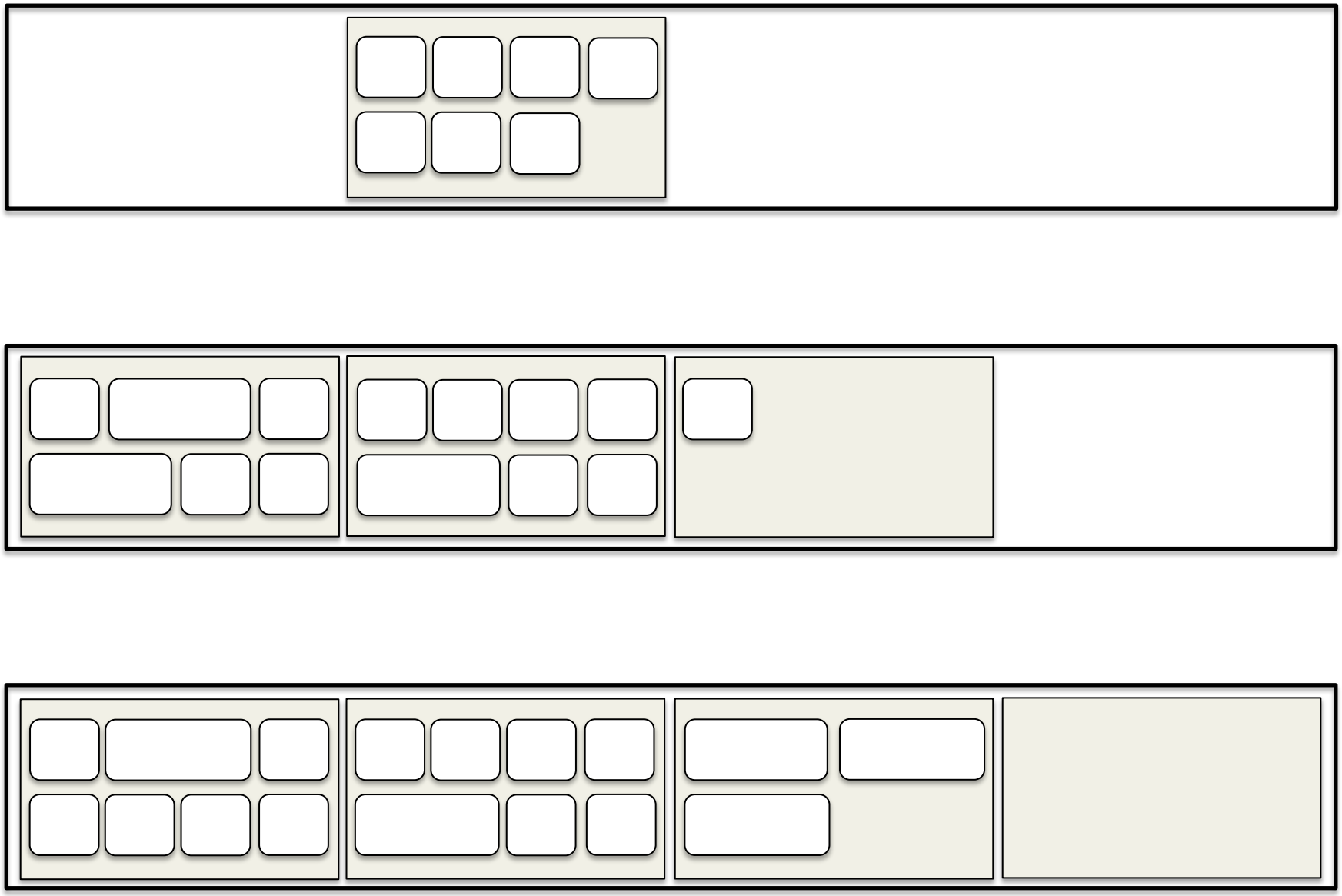


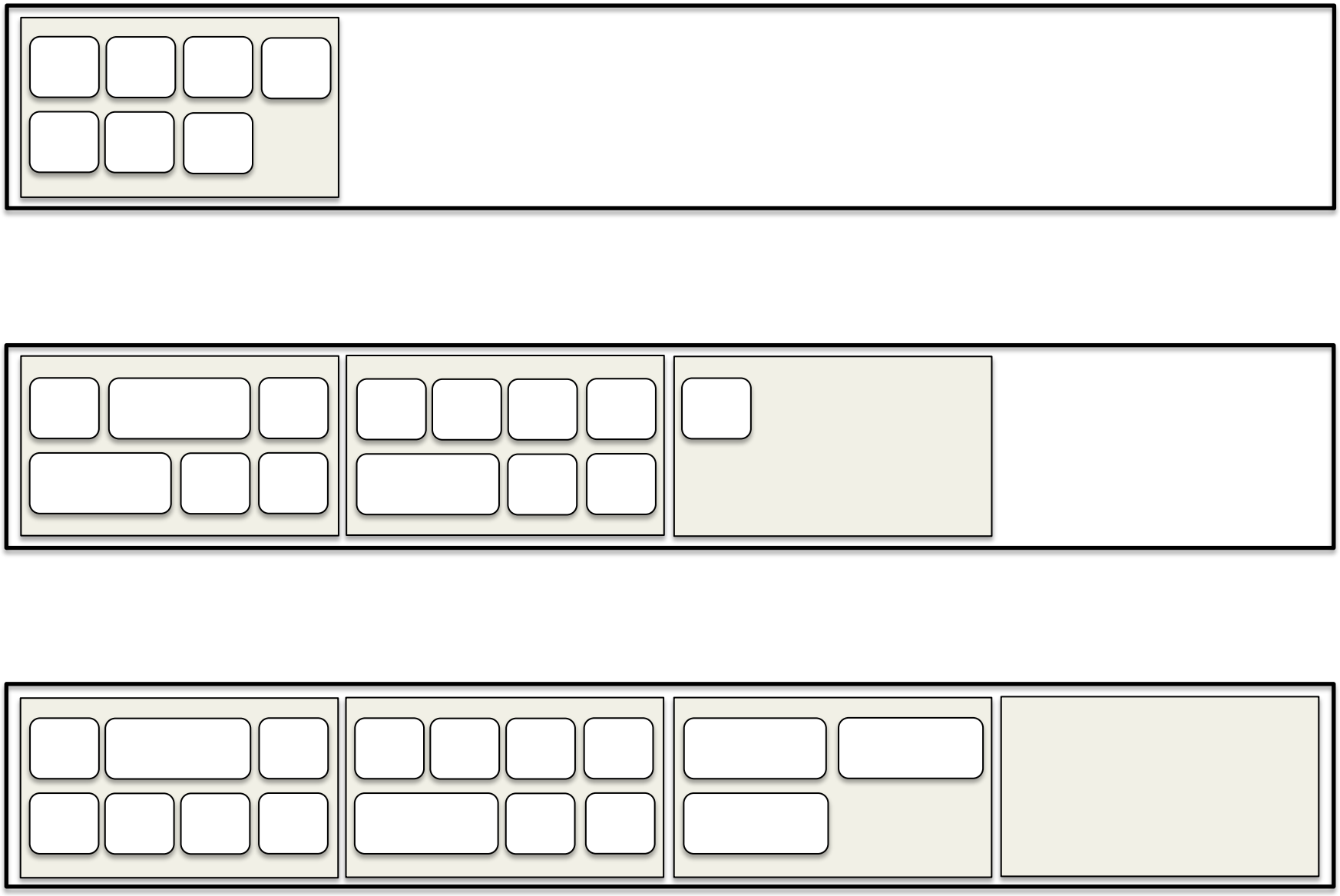






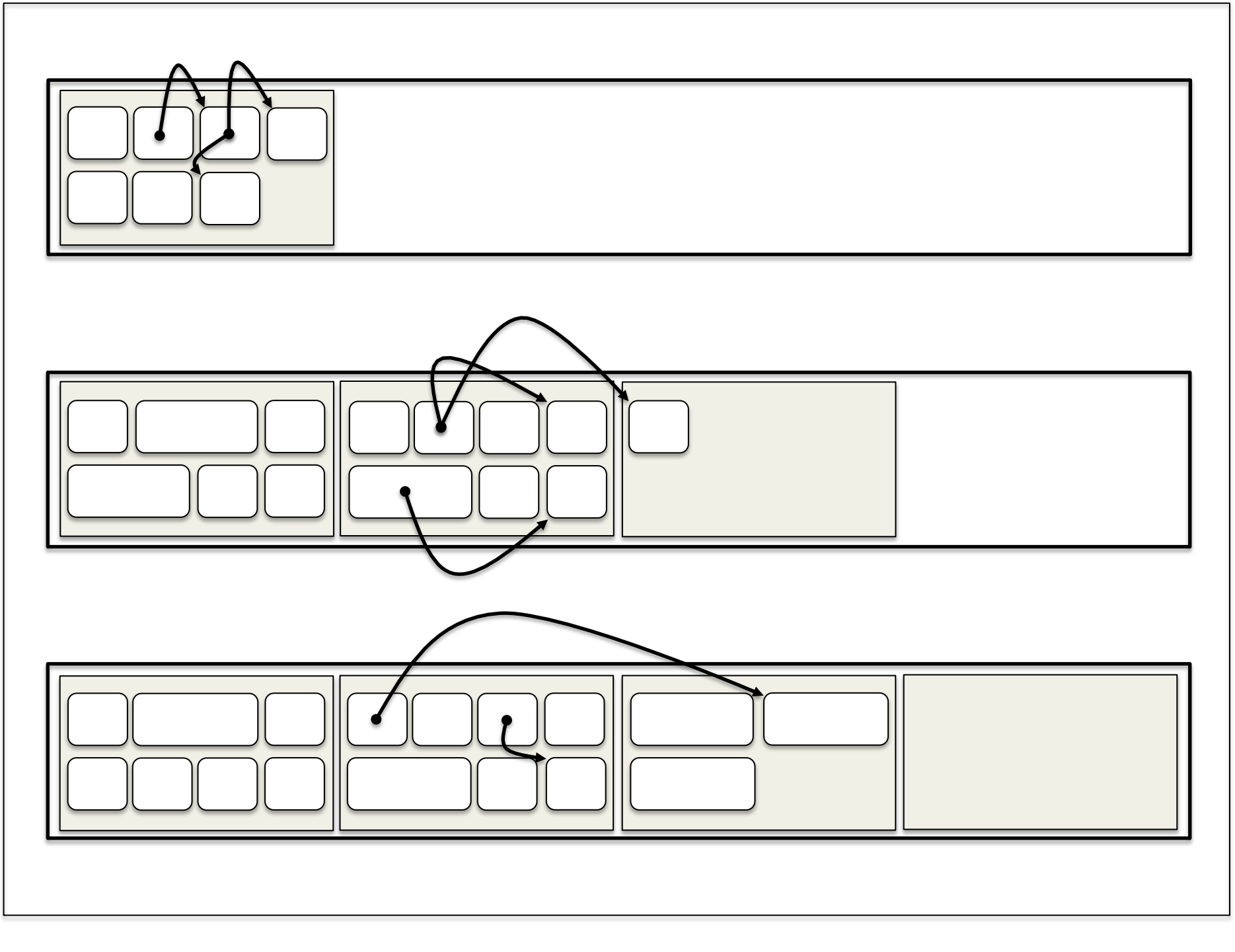


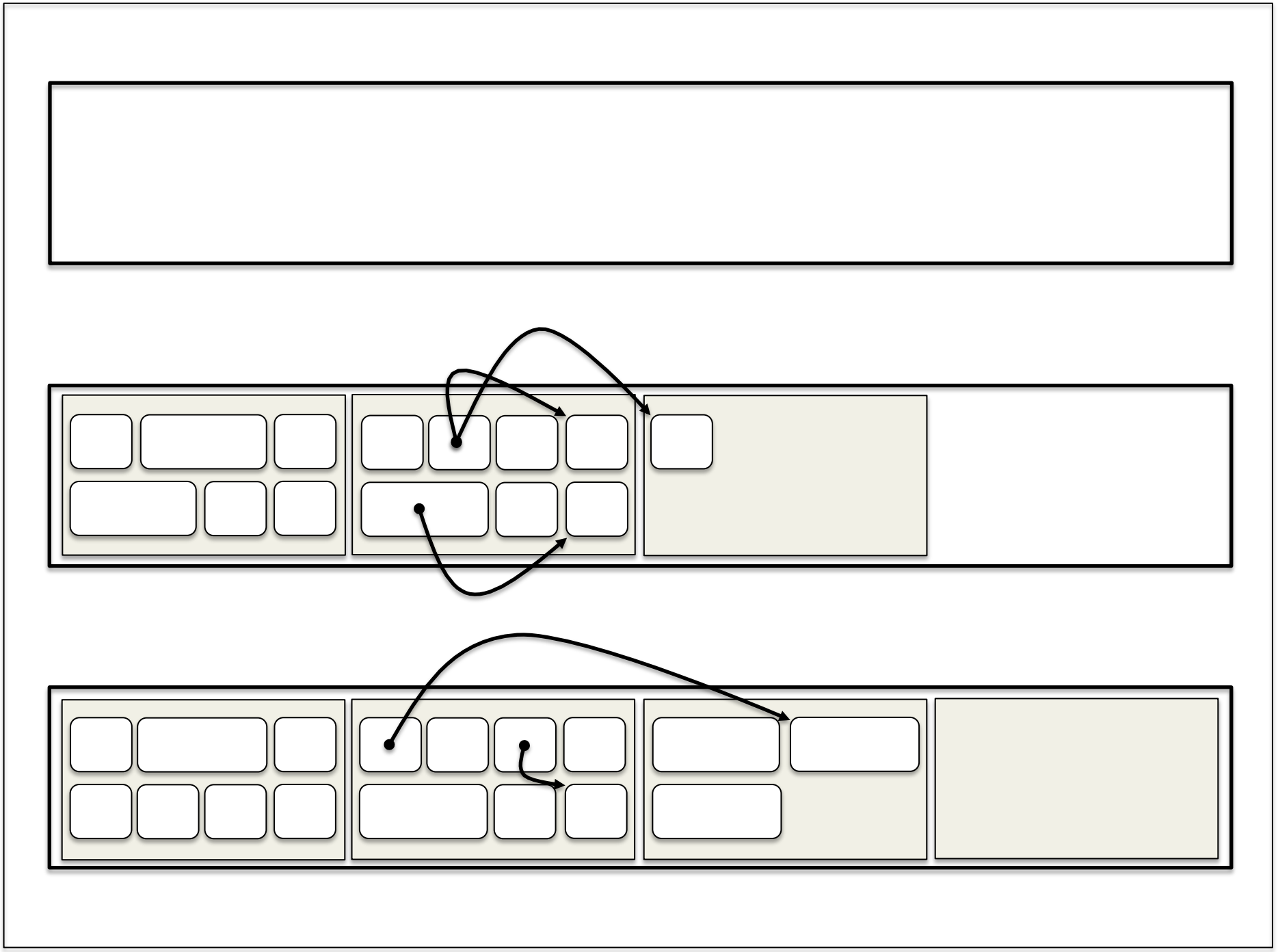


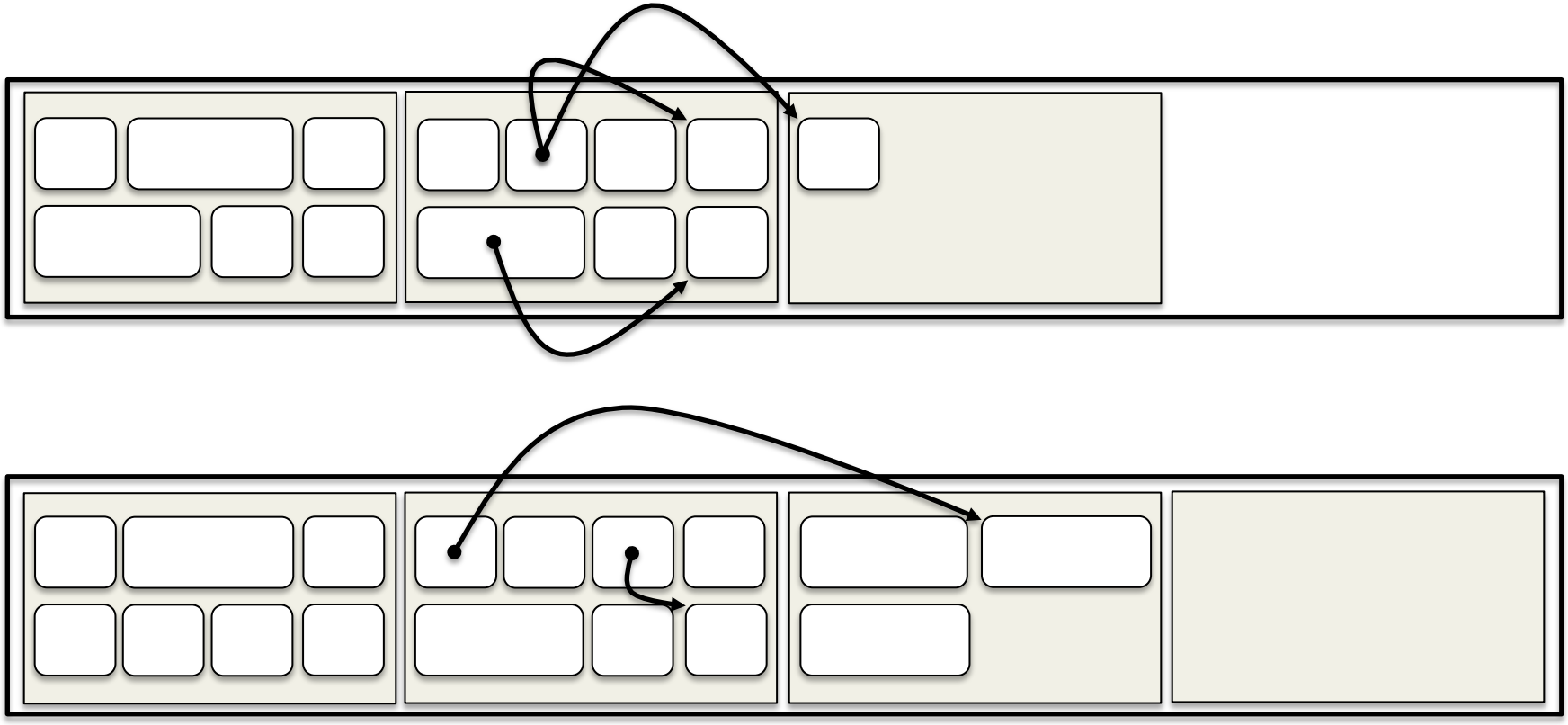


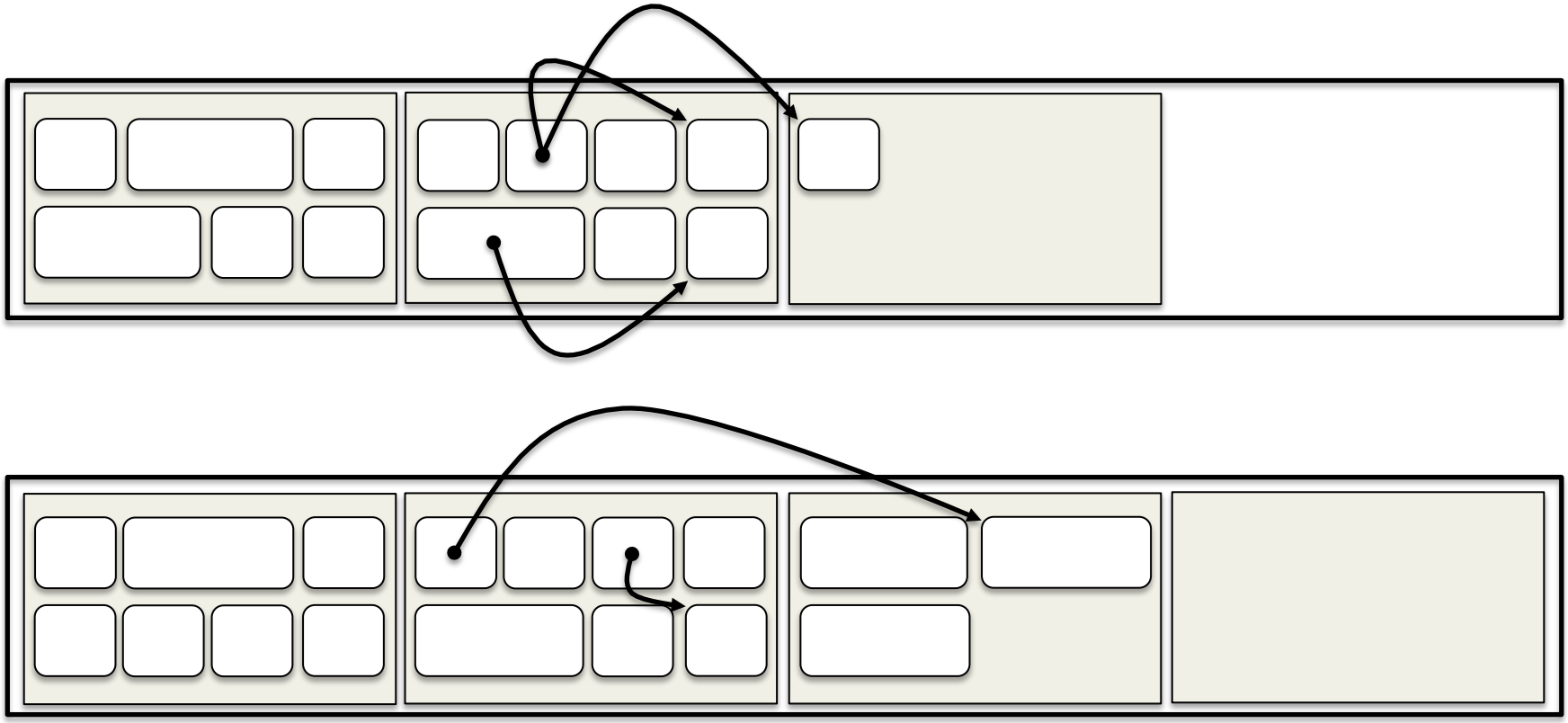
Clustering Objects

- Objects in a car can be pointed to from outside
  - System root locations
  - Remembered sets
- Move connected objects to the same train
  - To the same car if possible
- More likely that all references are local
  - We can collect the whole car with no extra work



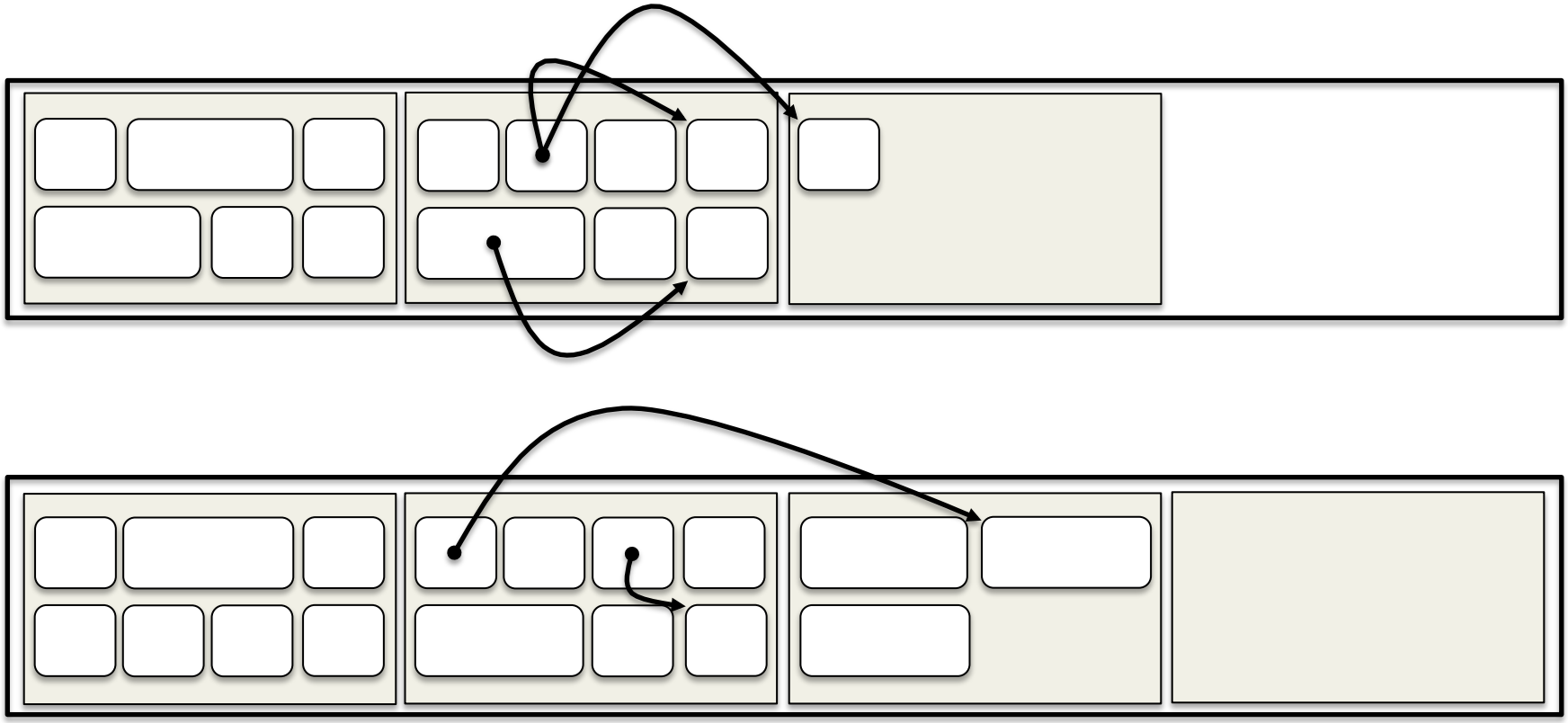


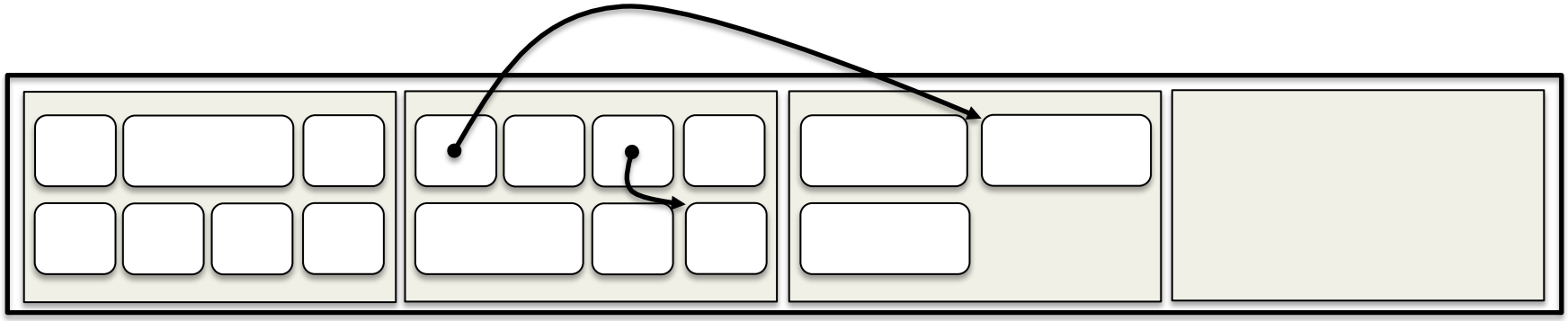


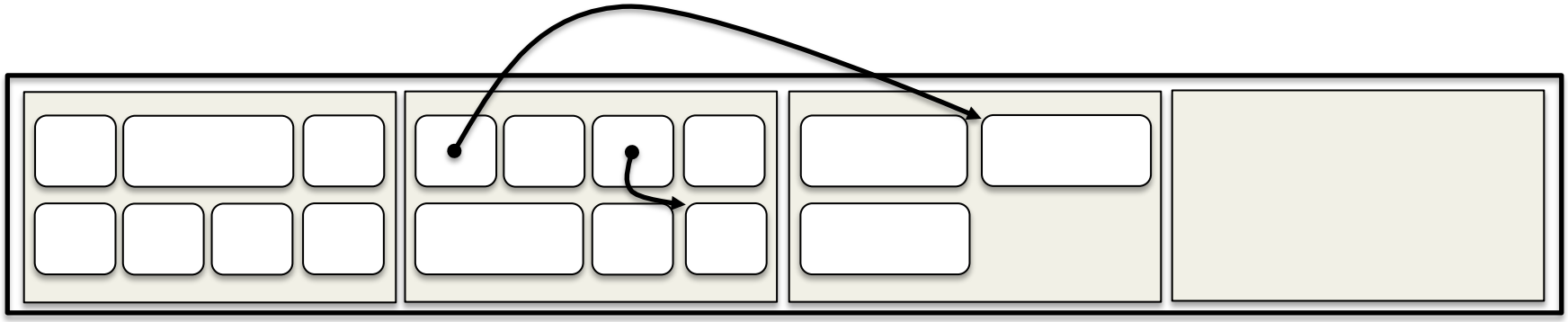


Collecting Trains

- The same thing holds for trains
  - We know all external references to a train
  - Sometimes a whole train becomes unreachable
  - We can then collect the entire car
- We want to encourage this behavior
  - Advantage of moving referenced objects together
  - Higher chance of large structures becoming garbage







Algorithm Details

- We know that all cars will be collected
  - We process trains and cars in order
  - Every train eventually becomes the first
- Create new trains based on allocation rate
- Some objects may not fit in a single car
  - Maintain a separate large object space
- Enforces an upper bound on pause time
  - But may not guarantee that space is freed

Train Algorithm Characteristics

- Maximizes dynamic locality
  - Clusters related objects together
  - Cycles tend to stay within a given train
- Implementation is quite complex
  - Managing barrier overhead is difficult
  - Lots of remembered sets to manage
- Algorithm elements show up elsewhere

Distributed MOS

- Consider some of the train algorithm's benefits
  - Related objects are moved close together
  - Reference between trains minimized
  - Copying generally happens within a train
- Excellent characteristics for a distributed system
  - References between machines very expensive
  - Local overhead less important
- Train algorithm a basis for distributed GC

Copying Overhead

- The train algorithm doesn't eliminate copying
  - All cars are evacuated in order
  - All objects are copied or discarded
- Do we have to process all cars in order?
  - Some cars may never be garbage collected
  - Do we care?

Garbage First

- Basis of Hotspot's cutting-edge collector
- We know how many references point to a region
  - Focus on the regions with the fewest references
  - Likely to have the most garbage
- G1 gets more accurate profiling data
  - Uses concurrent mark process to estimate live ratio

Card Marking

- Remembered set costs vary with pointer density
  - Lots of old-new pointers make for large sets
  - Slow path taken more often
- Card marking is a way to fix overheads
  - Divide heap up into equally-sized cards
  - Maintain a bit (or byte) map off to the side
  - On any pointer write, mark the corresponding card

Card Marking Advantages

- Bounded memory use
  - Depends on the size of the card
  - Also on whether bit or byte addressed
- Faster barrier
  - Don't bother checking source and target pointers
  - Unconditional update to the card table

Card Marking Issues

- May use more space
  - Depends on pointer density
  - Need to align objects on card boundaries
- Takes more time during GC
  - Need to scan all objects in the card
  - Could scan a lot of the heap for a few references

GC Safepoints

- The GC can't run at any time
  - Some operations put the stack in a bad state
- Consider the PutField bytecode
  - Pops a reference from the stack
  - Pops a value from the stack
  - Writes the value to the object at the reference
- What happens if GC happens at step 2

GC Safepoints

- Less important in the single-threaded case
  - GC triggered when an allocation fails
  - Must be in the NEW or NEWARRAY instructions
- Very important when multiple threads running
  - Need to stop the world at a safe point
- Safepoint locations implementation dependent

Safepoints in Hotspot

- Same process as for JIT compilation
  - Poison page algorithm
- Safepoint location depends on execution mode
  - Interpreter can pause at bytecode boundaries
  - JIT pauses at method return and back branches
- Code may be optimized between safe points
  - Particularly in the JIT

Stack and Local References

- Exact pointer information needed for most GC
  - Need to find references in stack and locals
- Easy to track in the interpreter
  - We know what the stack layout is at all safepoints
  - Can pre-calculate where references will be
- Much harder in JIT compiled code
  - Register allocation may put references anywhere

Stack Maps

- Stack maps allow JIT code to support GC
  - JIT generates information about reference location
  - GC can use the map to find all references
- Stack maps generated at each safe point
  - Expensive in time and space
  - JIT safe points are less frequent
- Stack maps very platform dependent

Read Barriers

- Generational GC requires a write barrier
  - Code runs whenever a reference is written
- Some implementations also use read barriers
  - Side-effect of bit stealing
  - Other bookkeeping tasks
- Reads are much more common than writes

Read Barriers

- Unconditional
  - Mask out stolen bits
    - 8% / 5% / 1% overhead for AMD/Pentium/Power
- Conditional
  - Test whether bits are stolen
    - 20% / 16% / 7% for AMD/Pentium/Power
- Overhead very platform specific
  - Power family of processors very cheap
  - AMD and Intel much more expensive