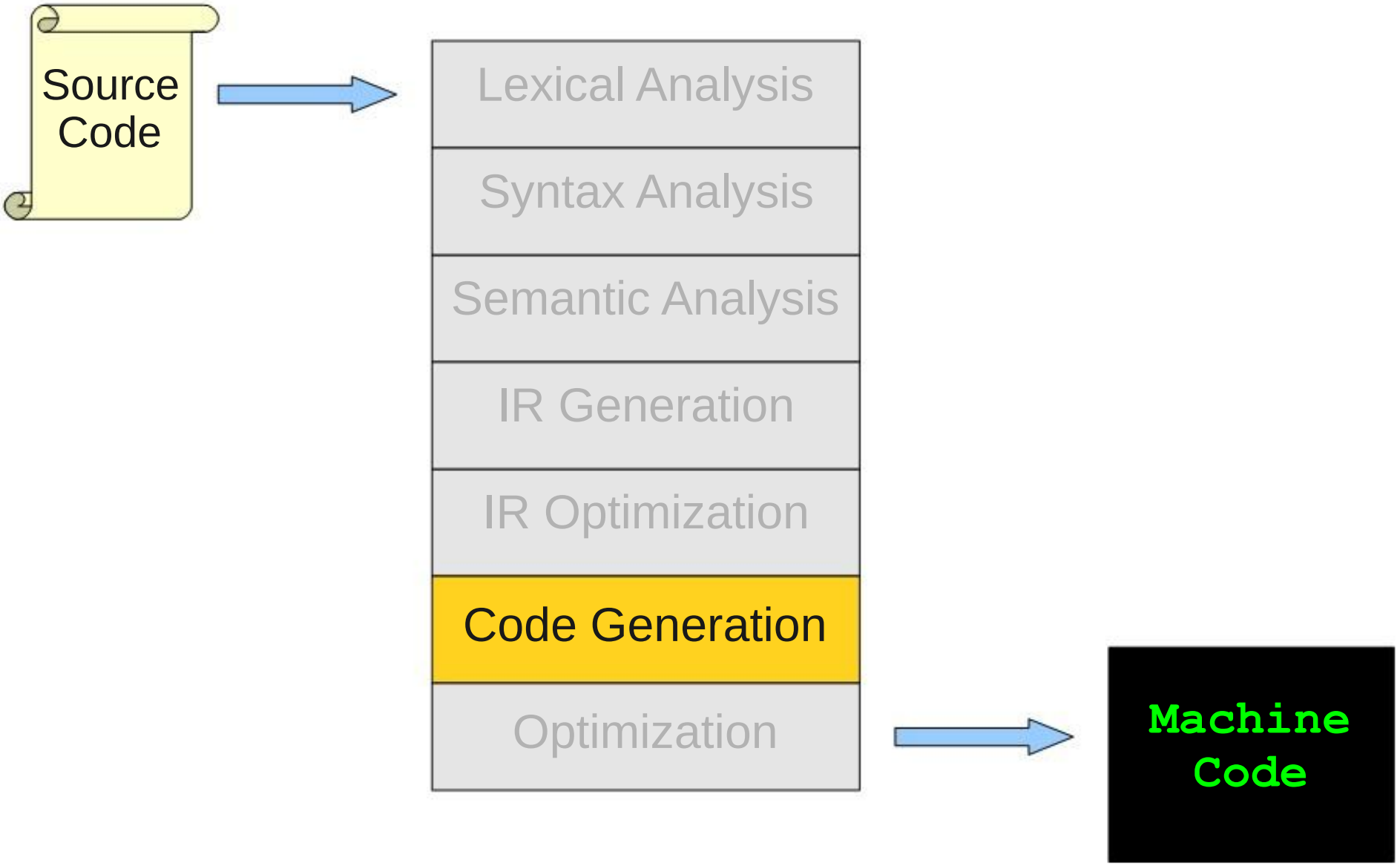
Garbage Collection

(continue)

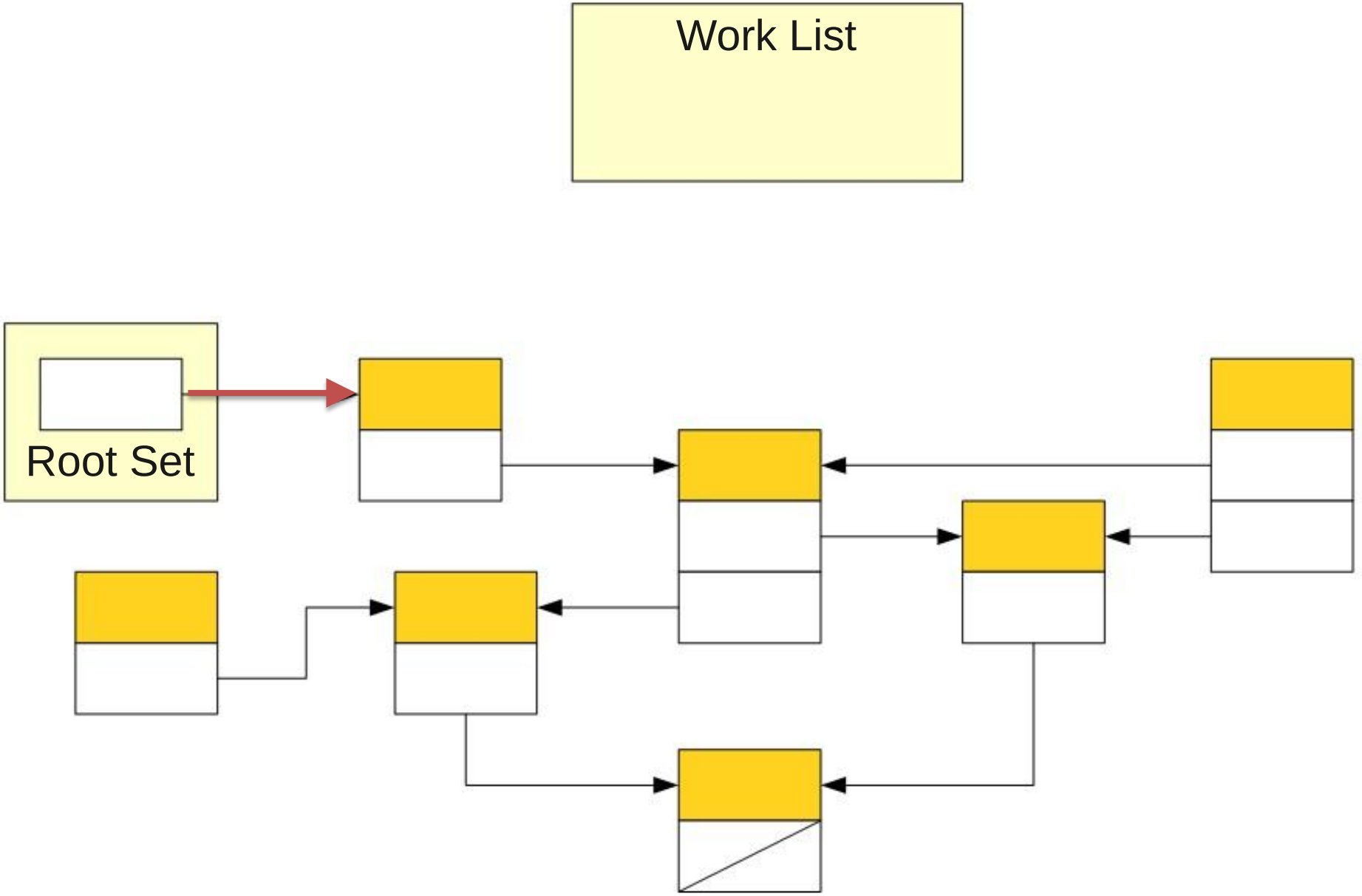
Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

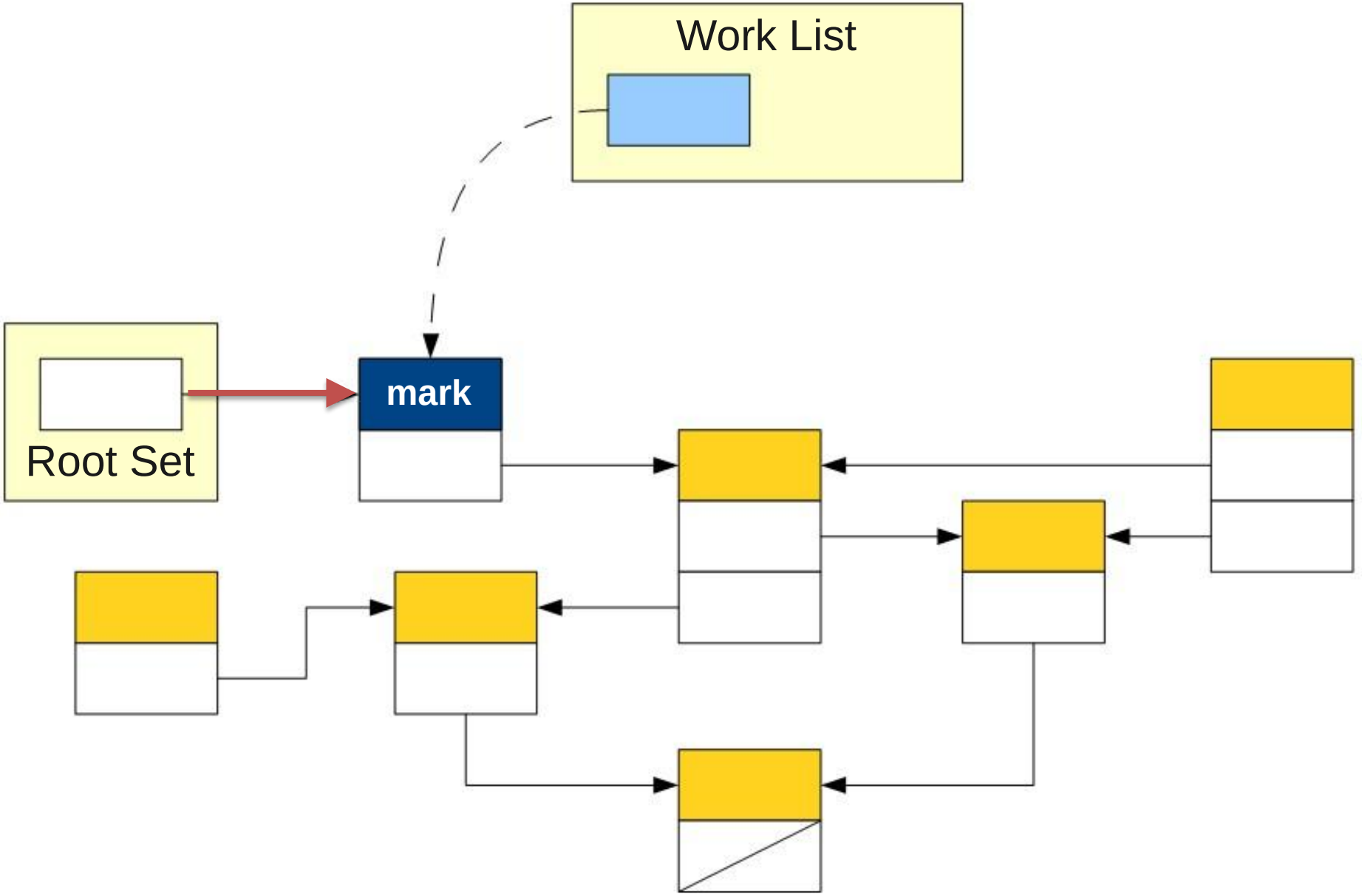
Mark-and-Sweep: The Intuition

- **Intuition**: find everything reachable in the program.
- The **root set** is the set of memory locations in the program that are known to be reachable.
- Do a **graph search** starting at the root set!

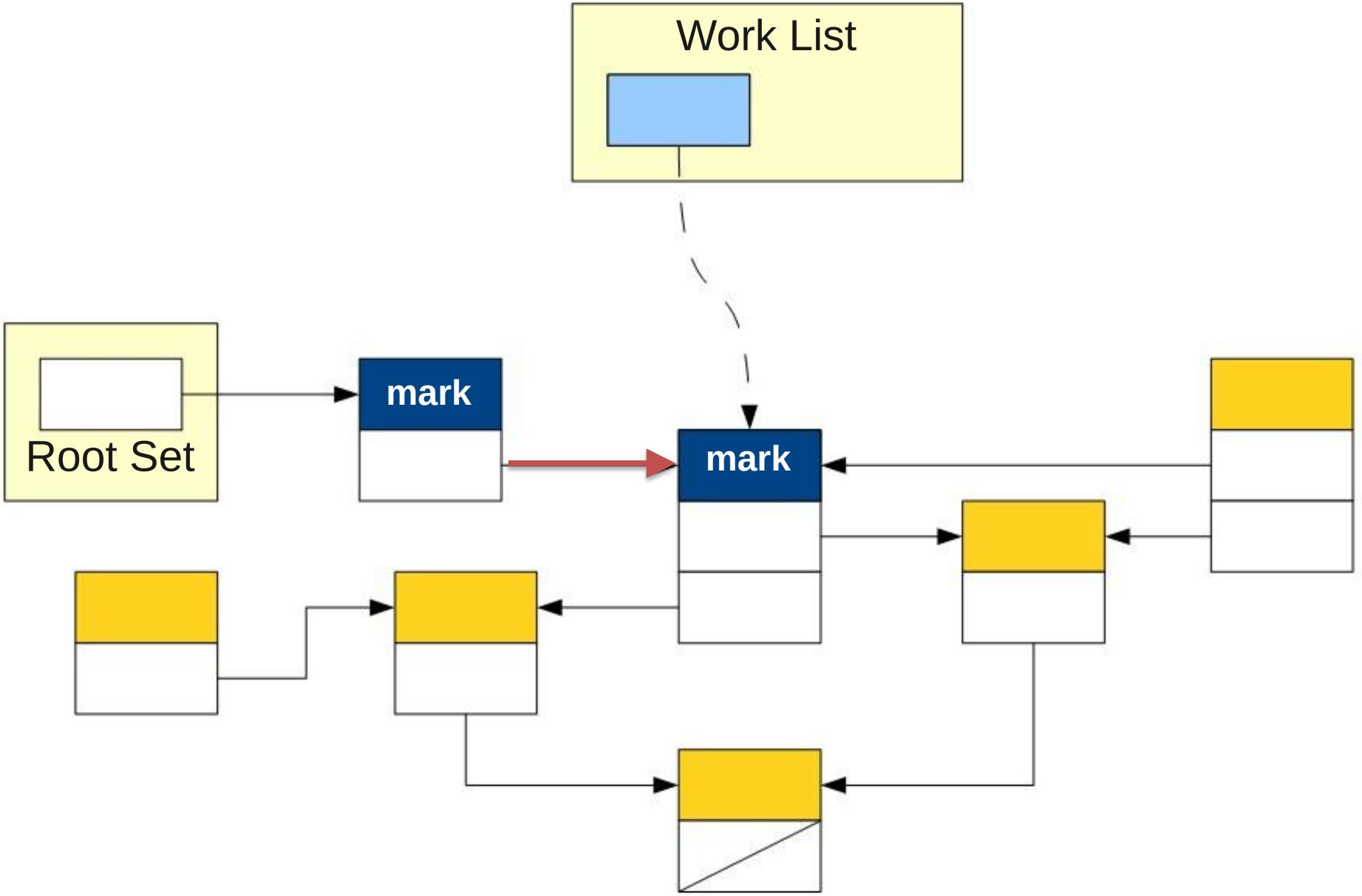
Mark-and-Sweep In Action



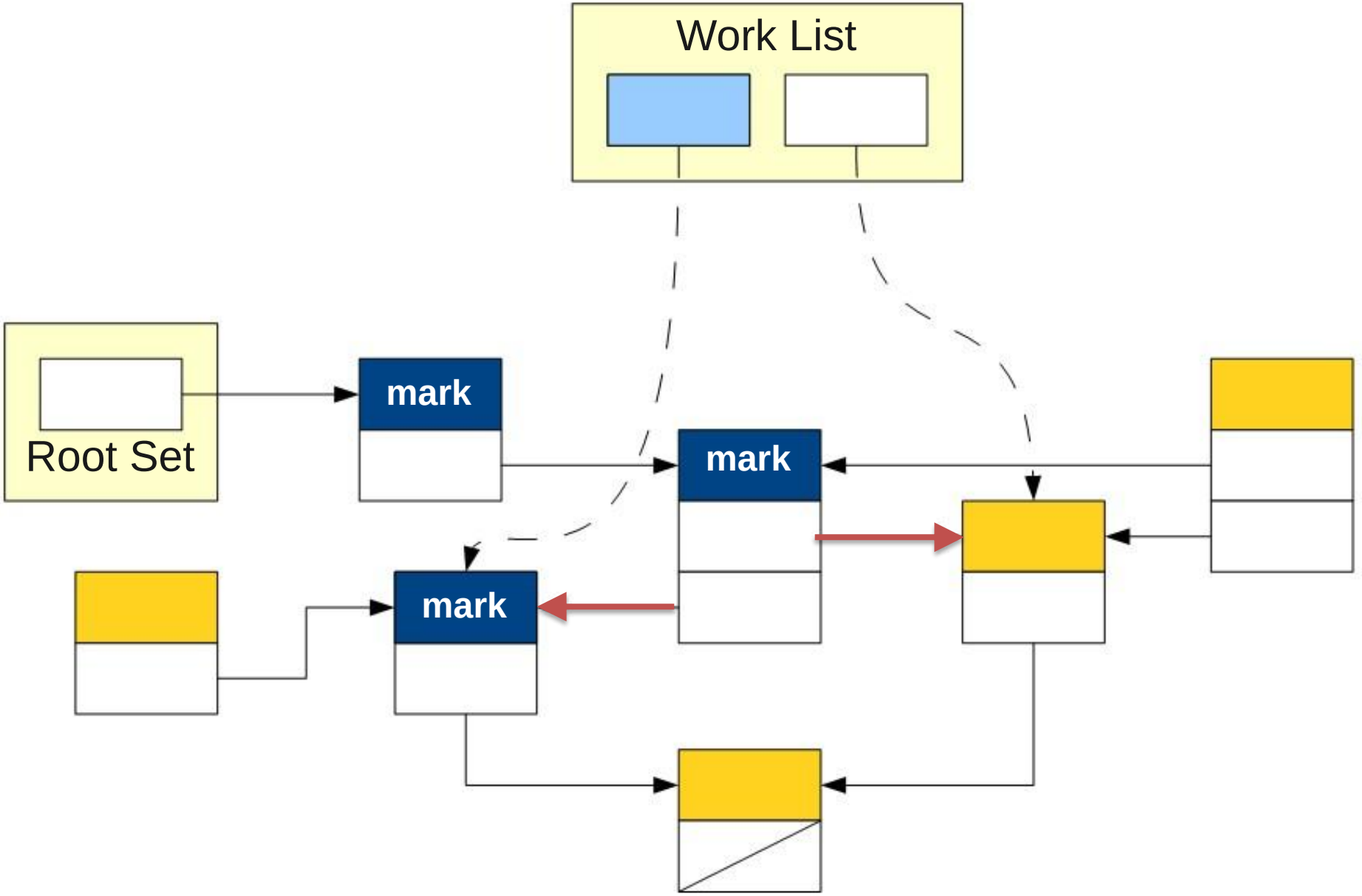
Mark-and-Sweep In Action



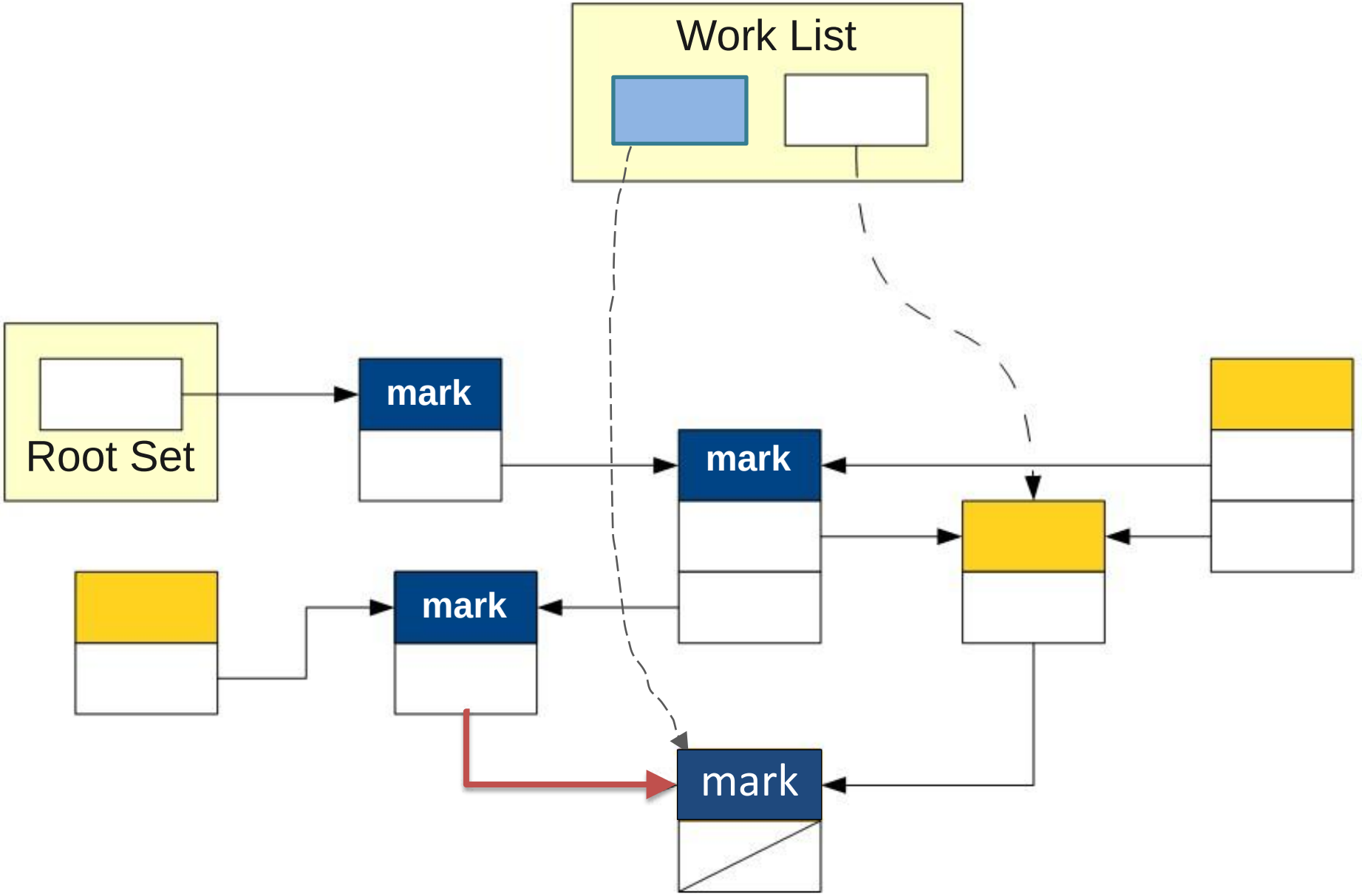
Mark-and-Sweep In Action



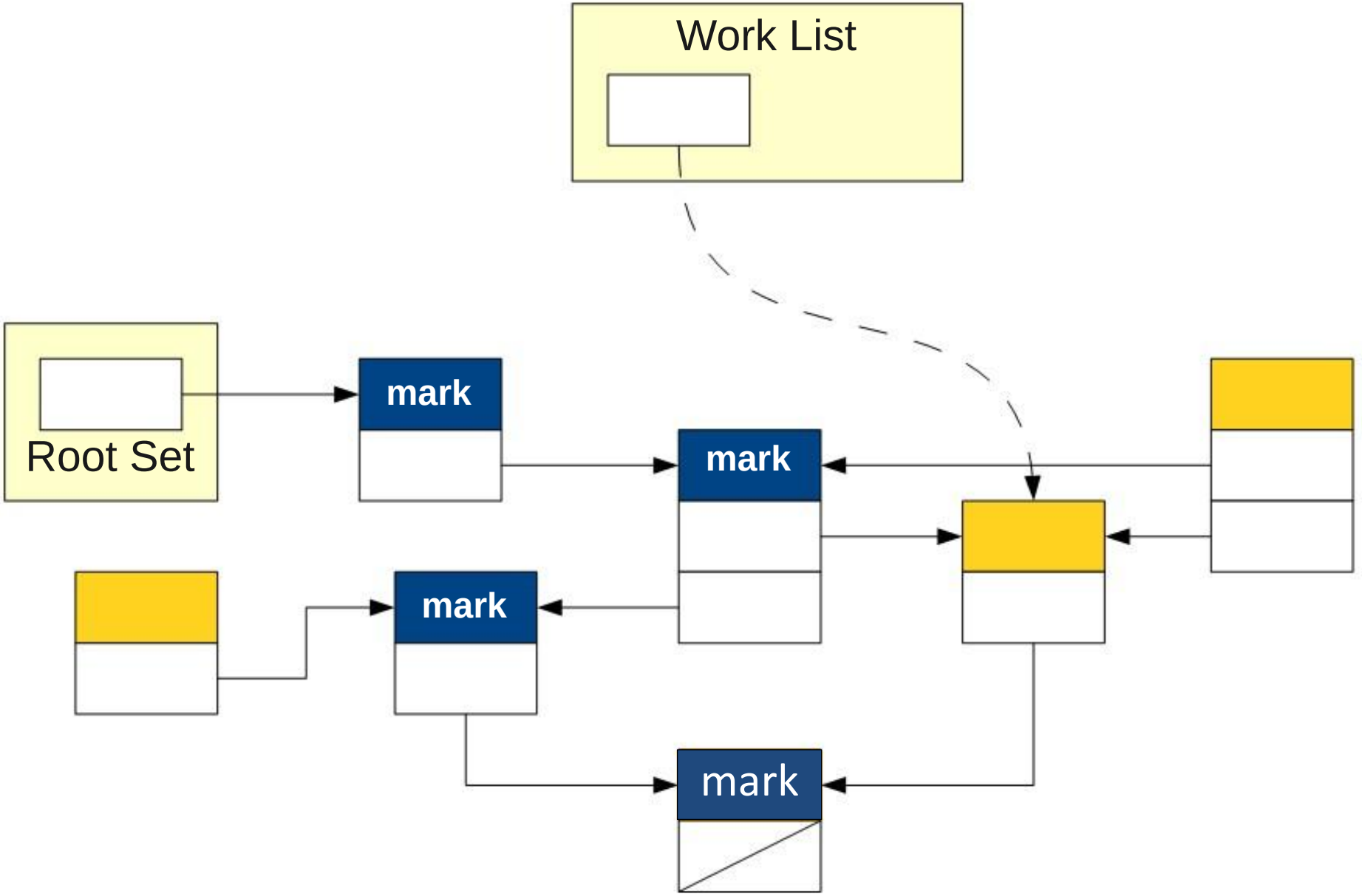
Mark-and-Sweep In Action



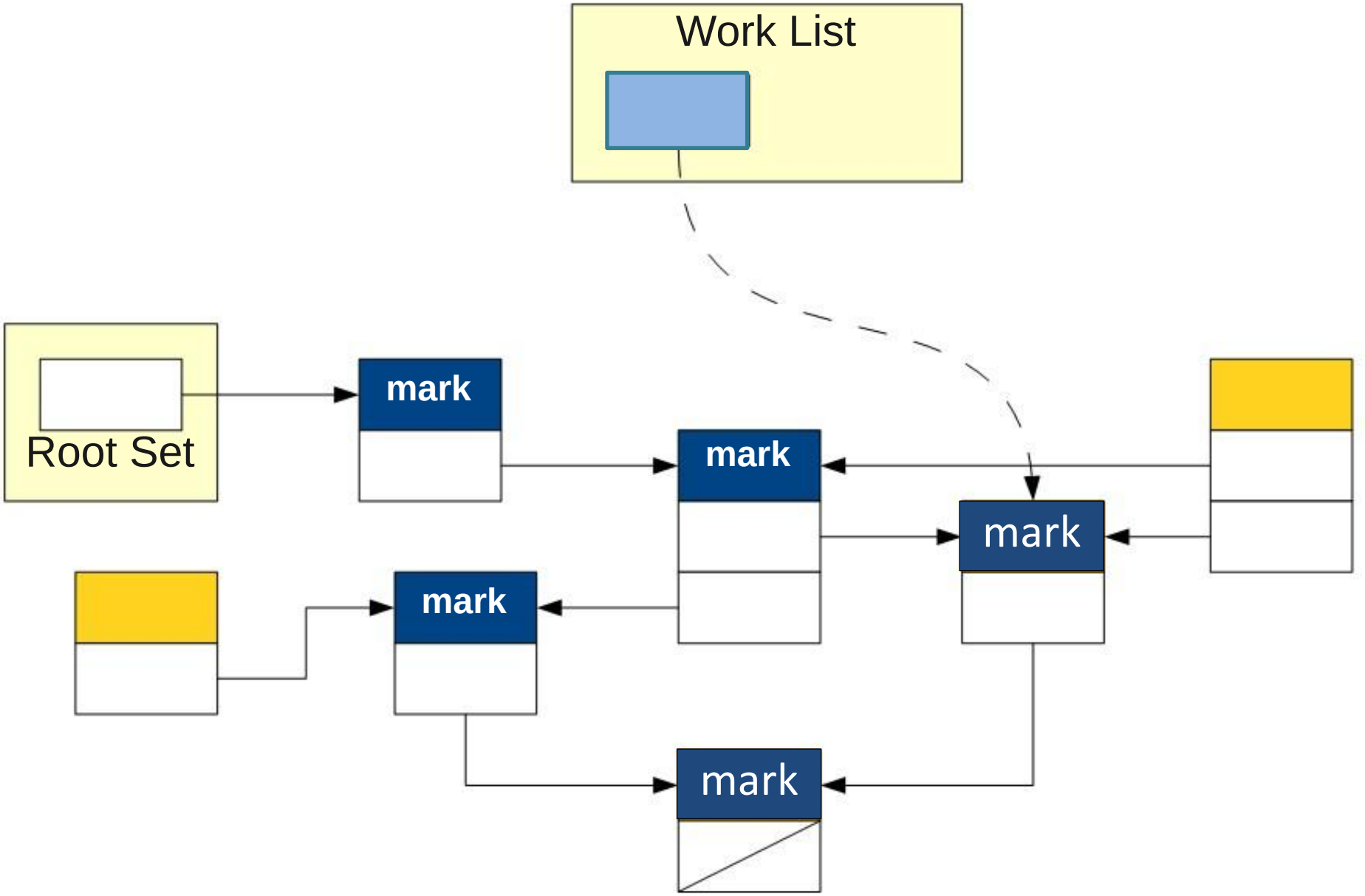
Mark-and-Sweep In Action



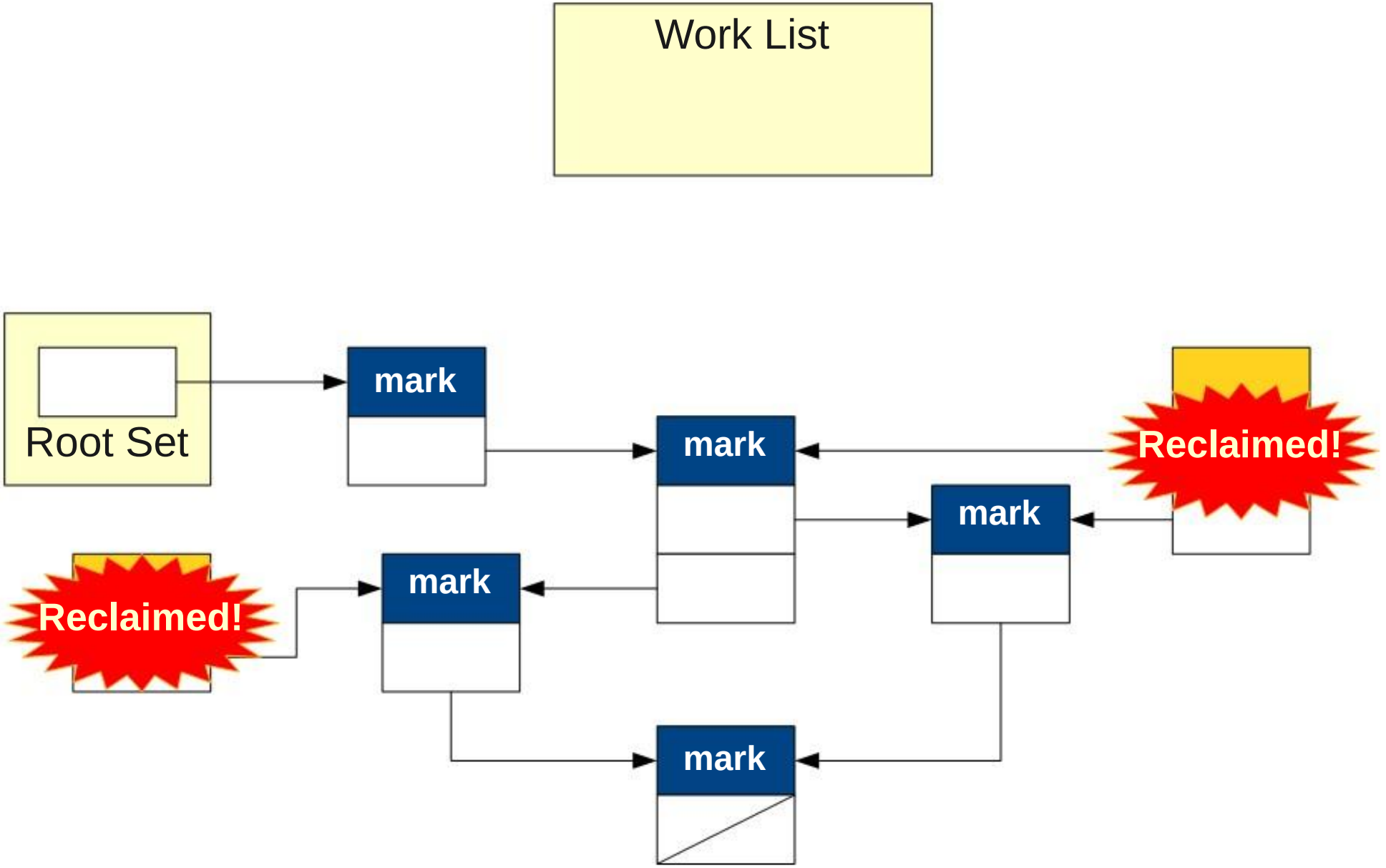
Mark-and-Sweep In Action



Mark-and-Sweep In Action



Mark-and-Sweep In Action



Problems of Mark-and-Sweep

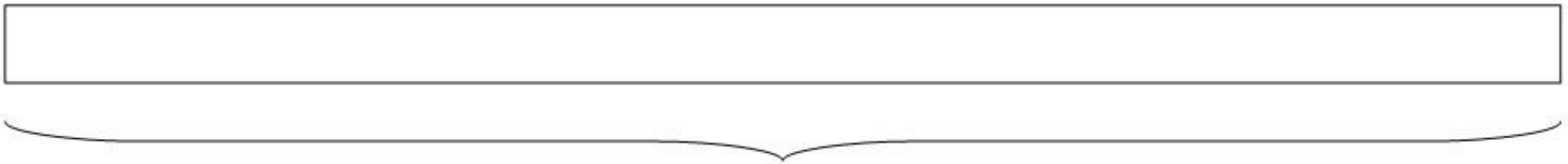
The mark-and-sweep algorithm has two serious problems.

- **Runtime proportional to number of allocated objects : O(n).**
  - Sweep phase visits all objects to free them or clear marks.
- **Work list requires lots of memory.**
  - Amount of space required could potentially be as large as all of memory.

Stop-and-Copy

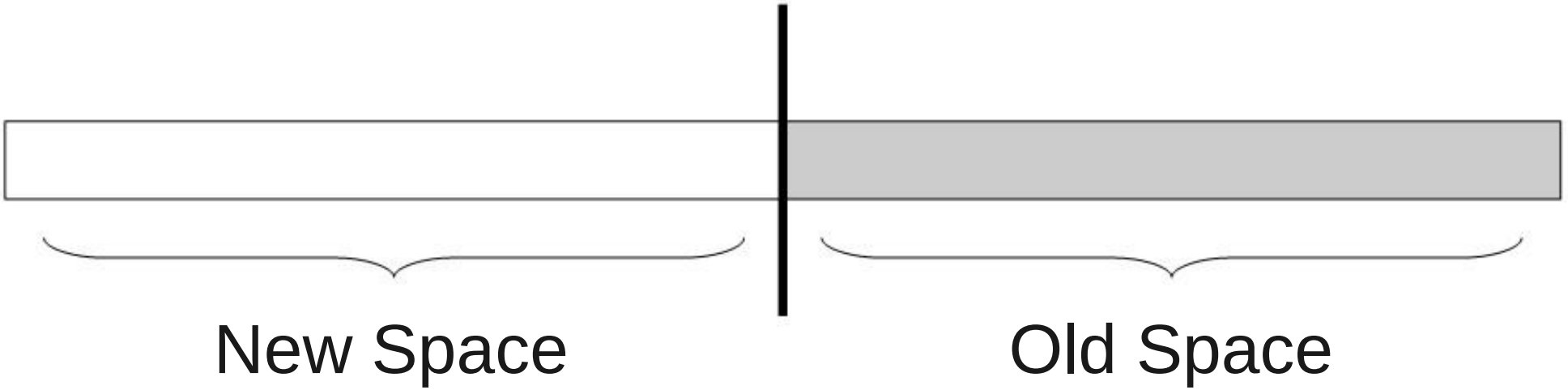
- **Idea**: When doing garbage collection, move all objects in memory so that they are adjacent to one another.
  - This is called **compaction**.
  - **Increasing locality** and **allocation speed**.

The Stop-and-Copy Collector

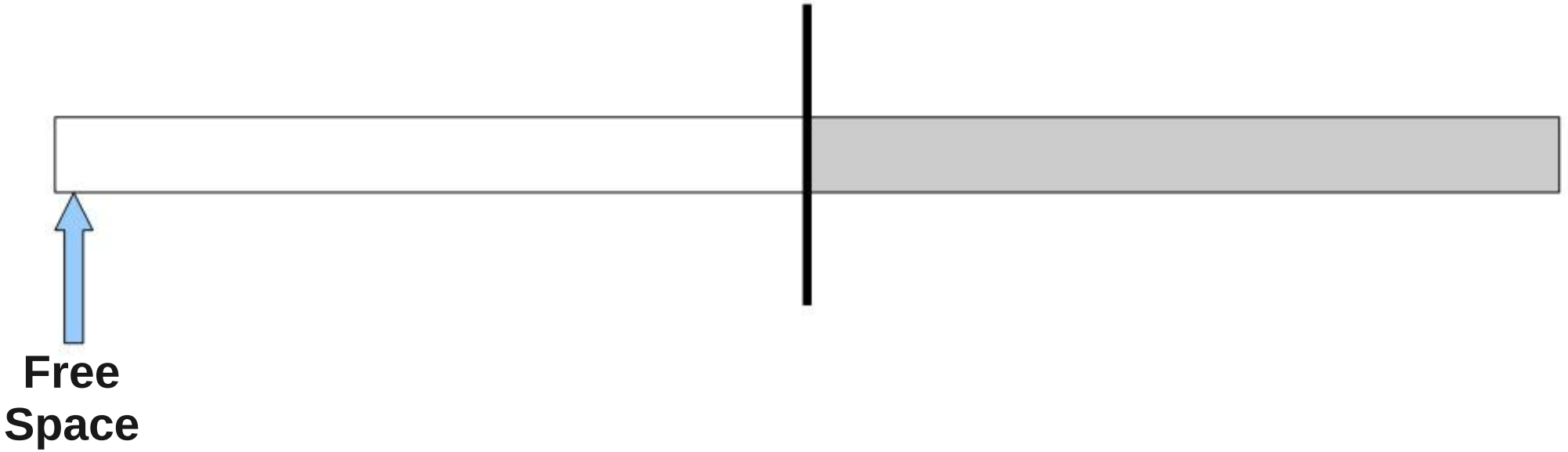


All of memory

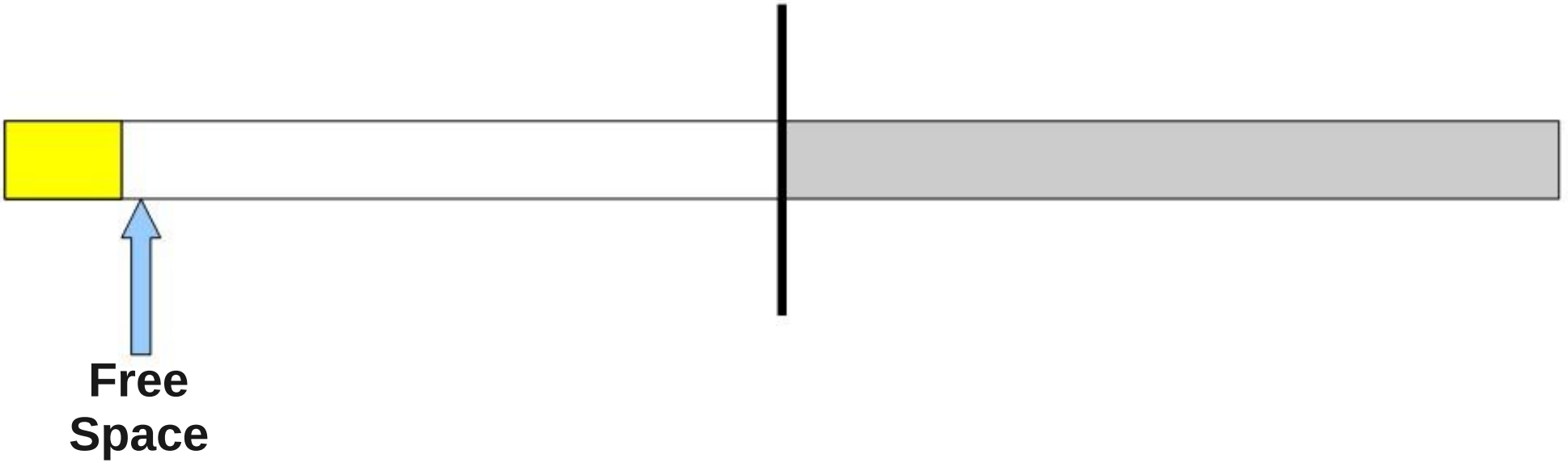
The Stop-and-Copy Collector



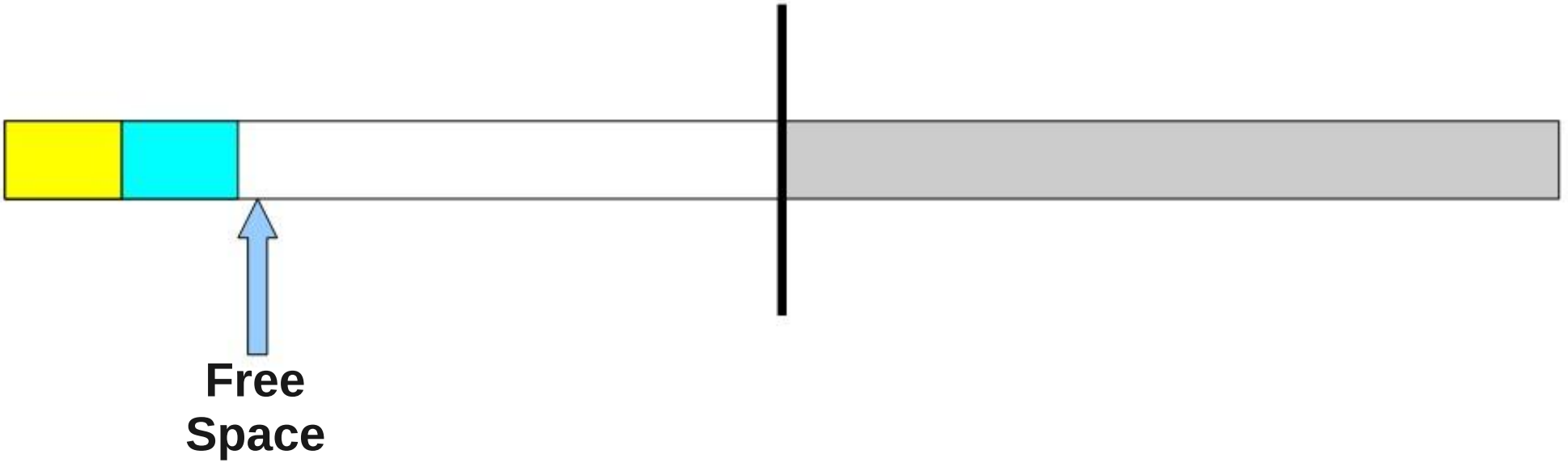
The Stop-and-Copy Collector



The Stop-and-Copy Collector

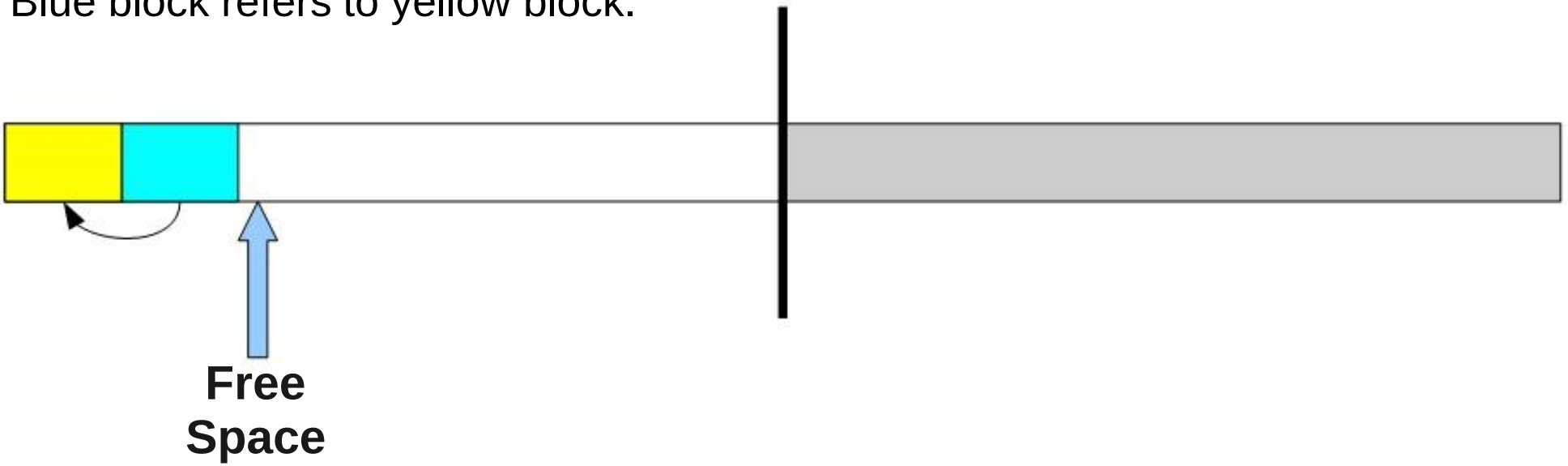


The Stop-and-Copy Collector

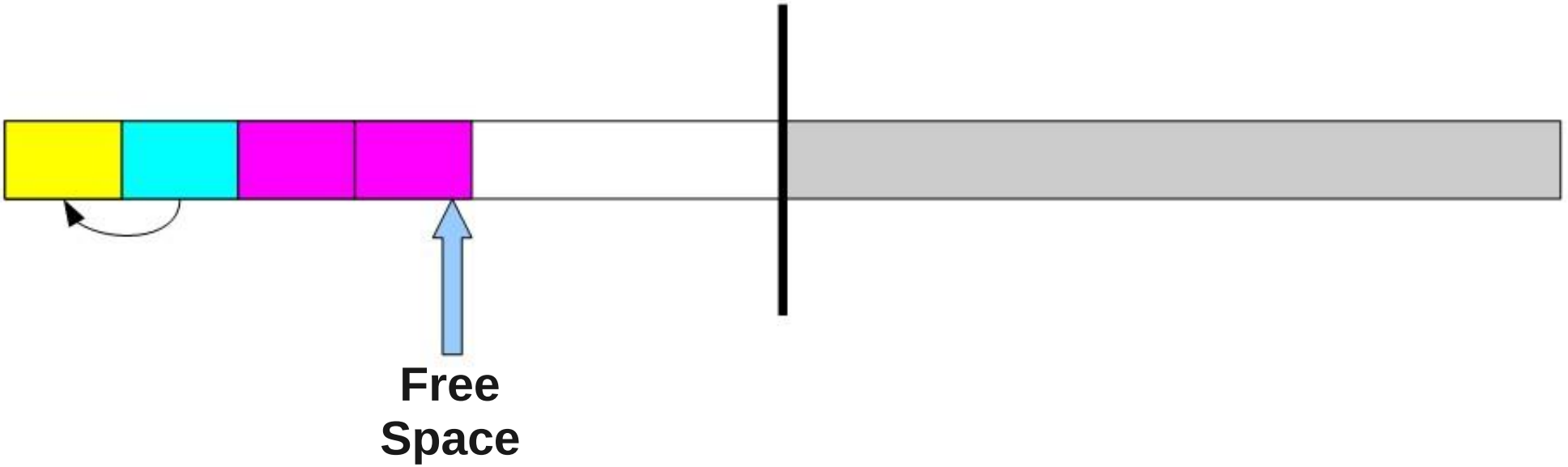


The Stop-and-Copy Collector

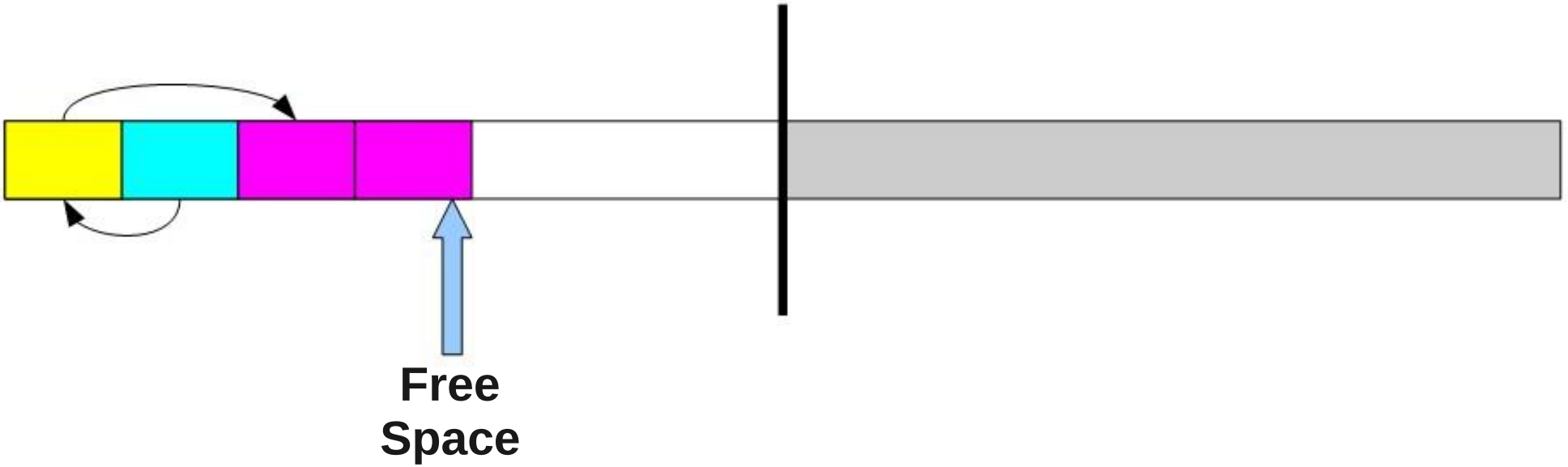
Blue block refers to yellow block.



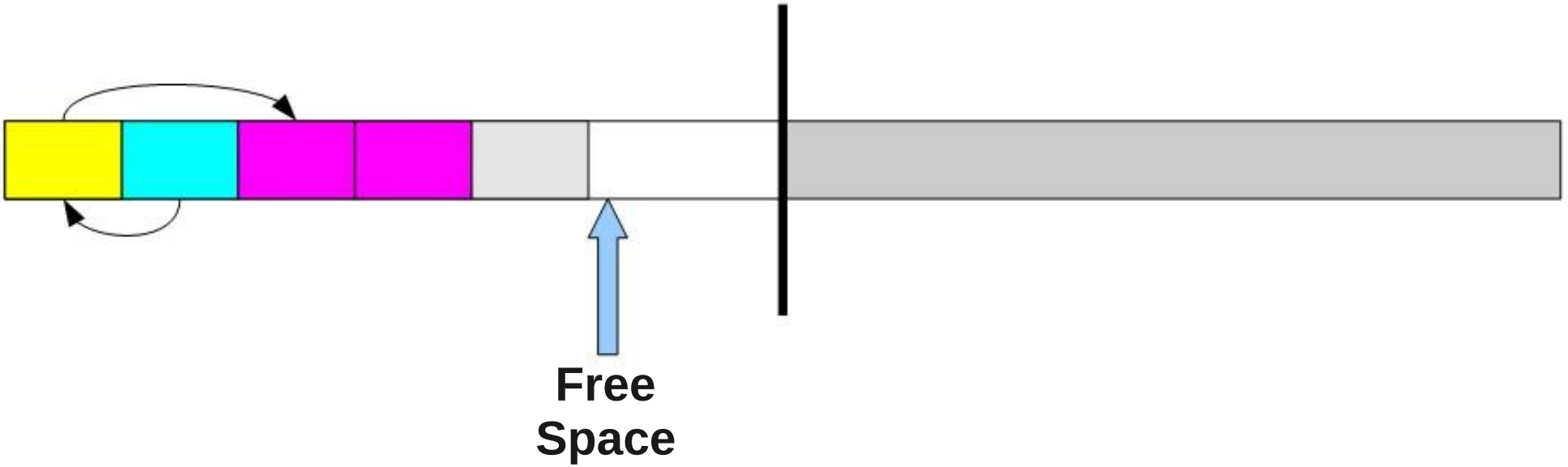
The Stop-and-Copy Collector



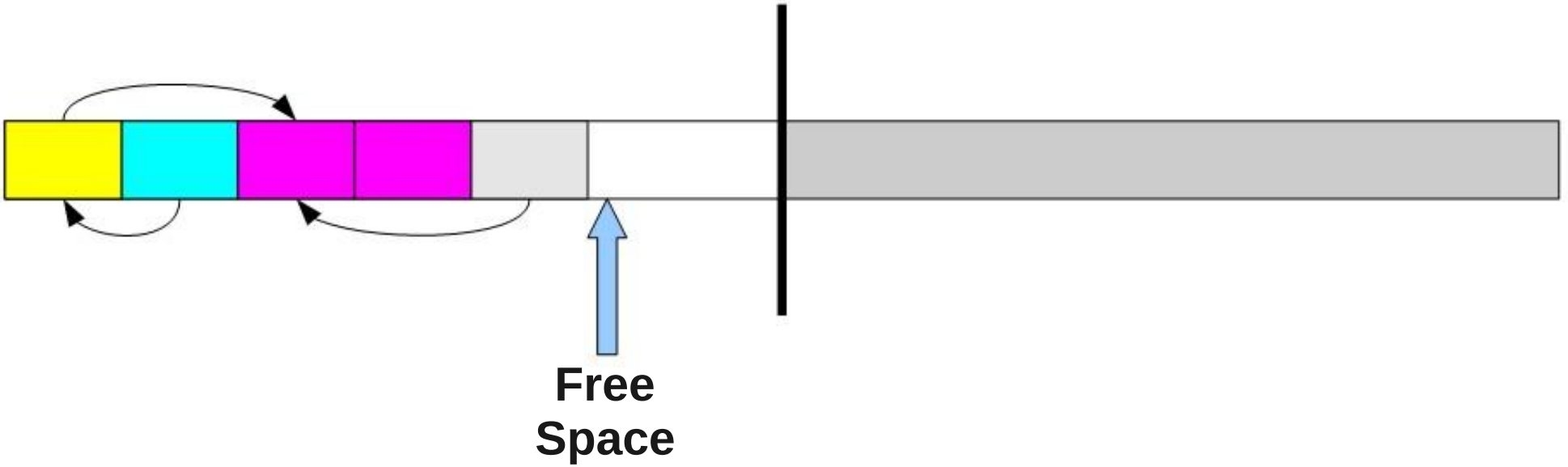
The Stop-and-Copy Collector



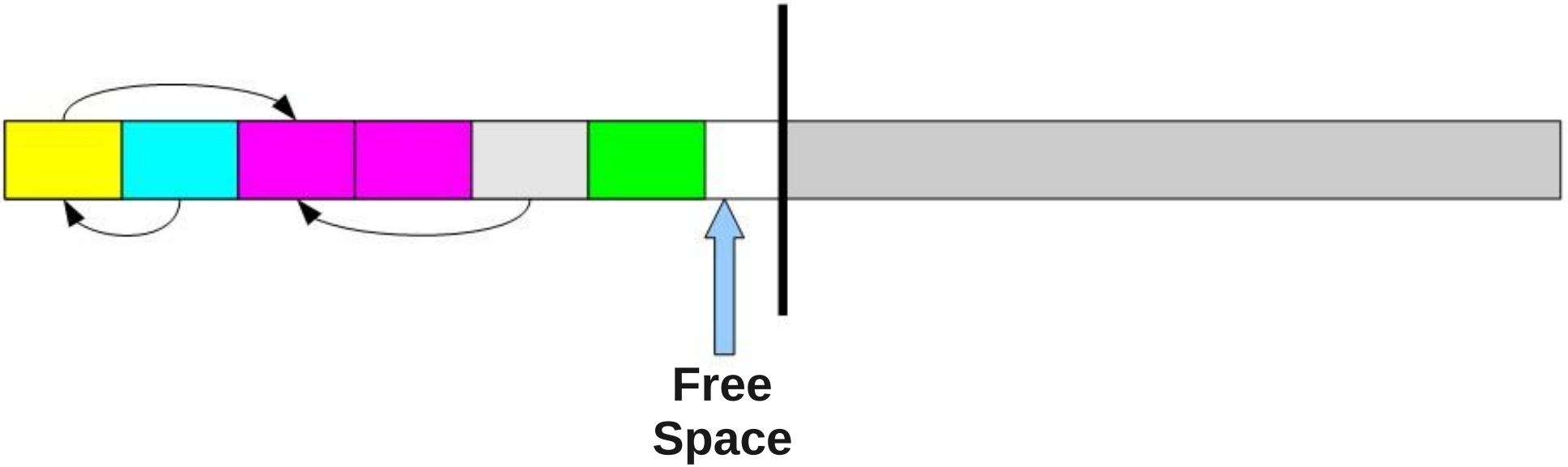
The Stop-and-Copy Collector



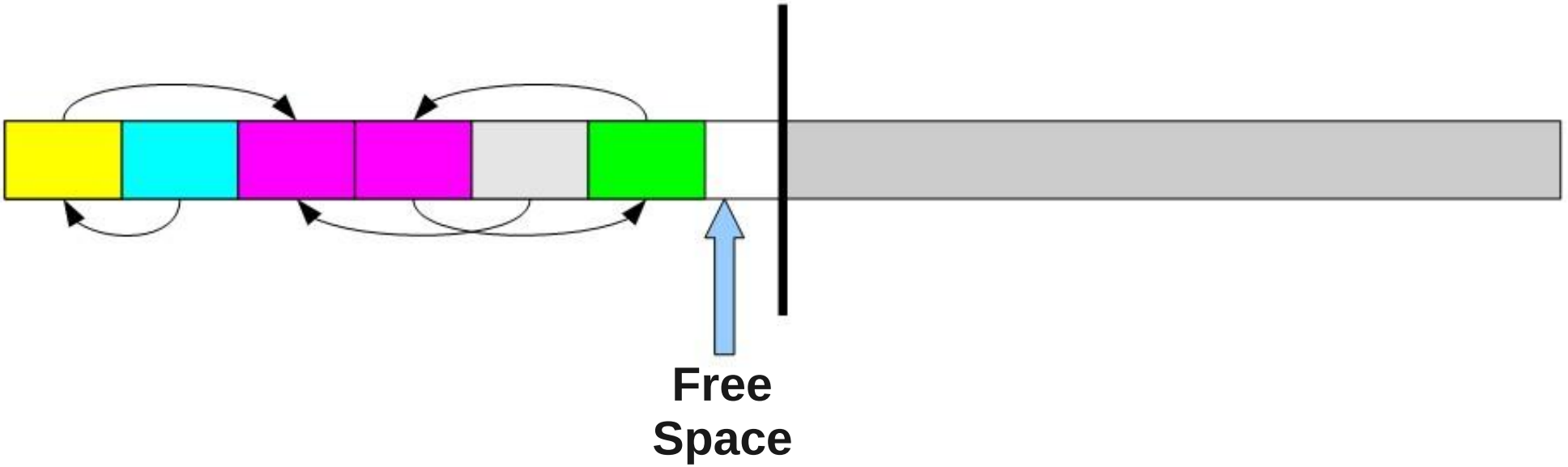
The Stop-and-Copy Collector



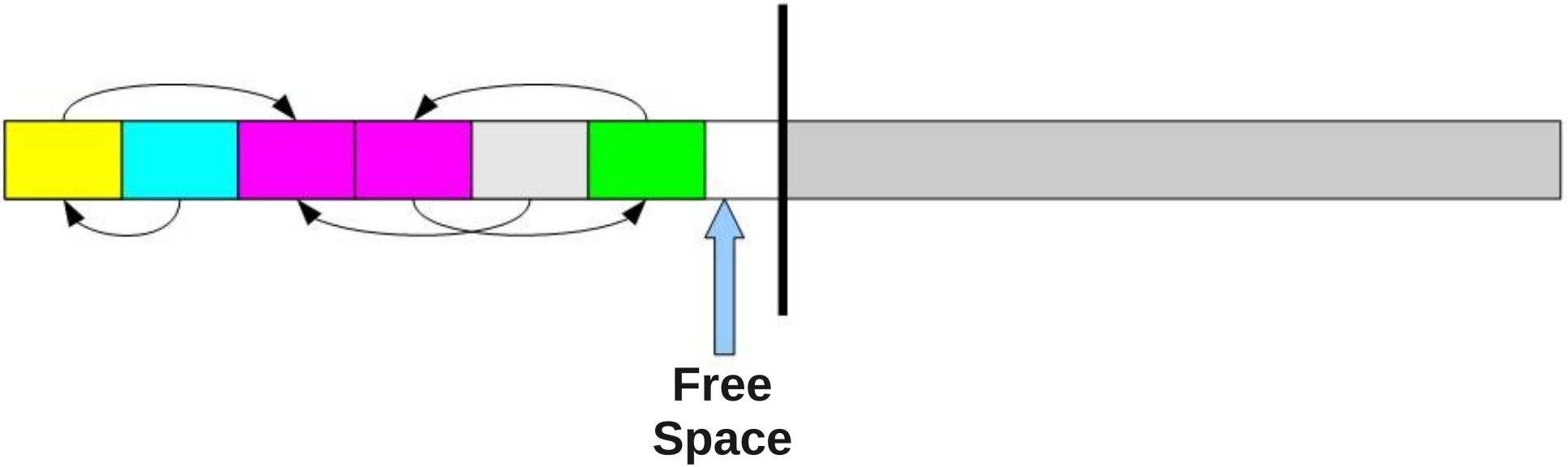
The Stop-and-Copy Collector



The Stop-and-Copy Collector

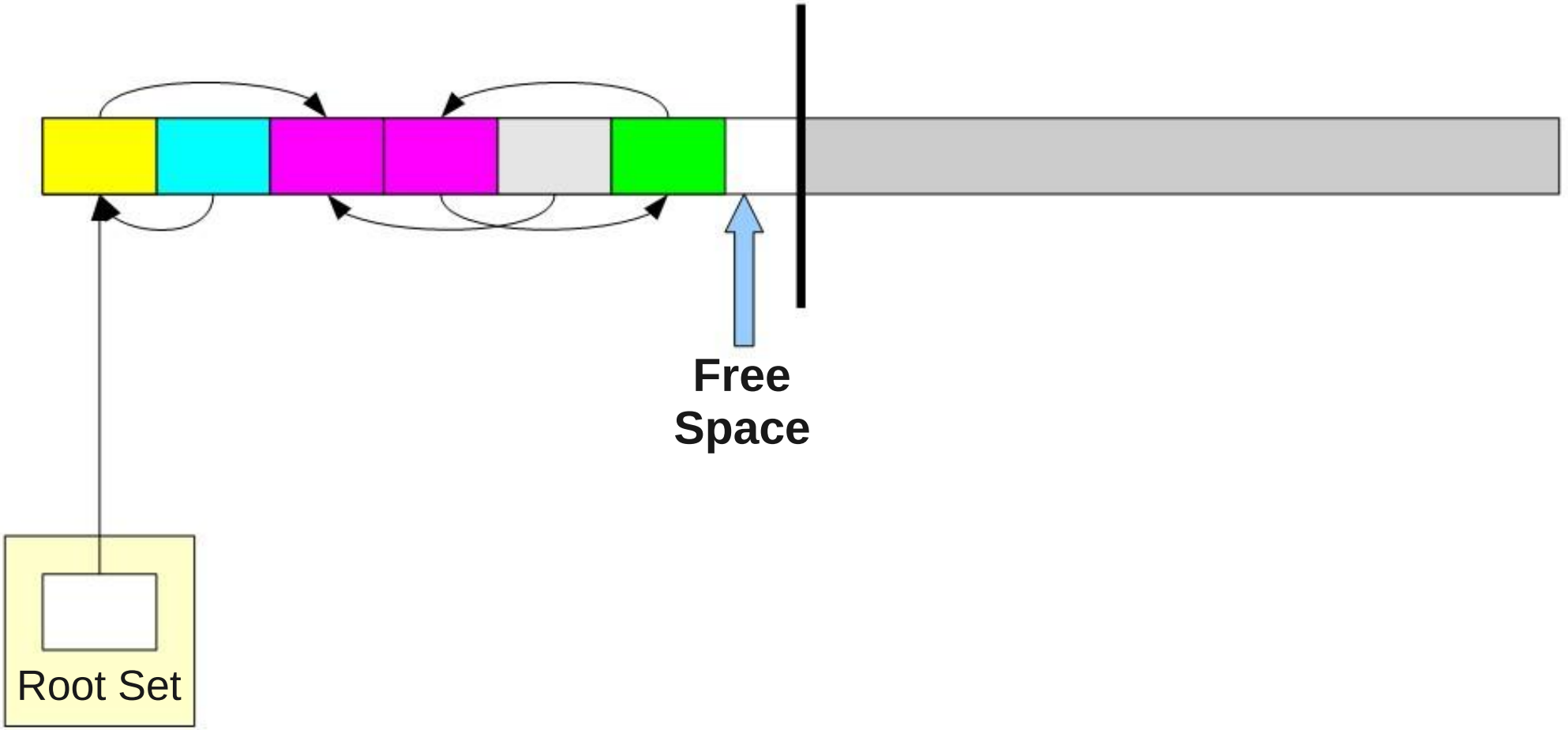


The Stop-and-Copy Collector

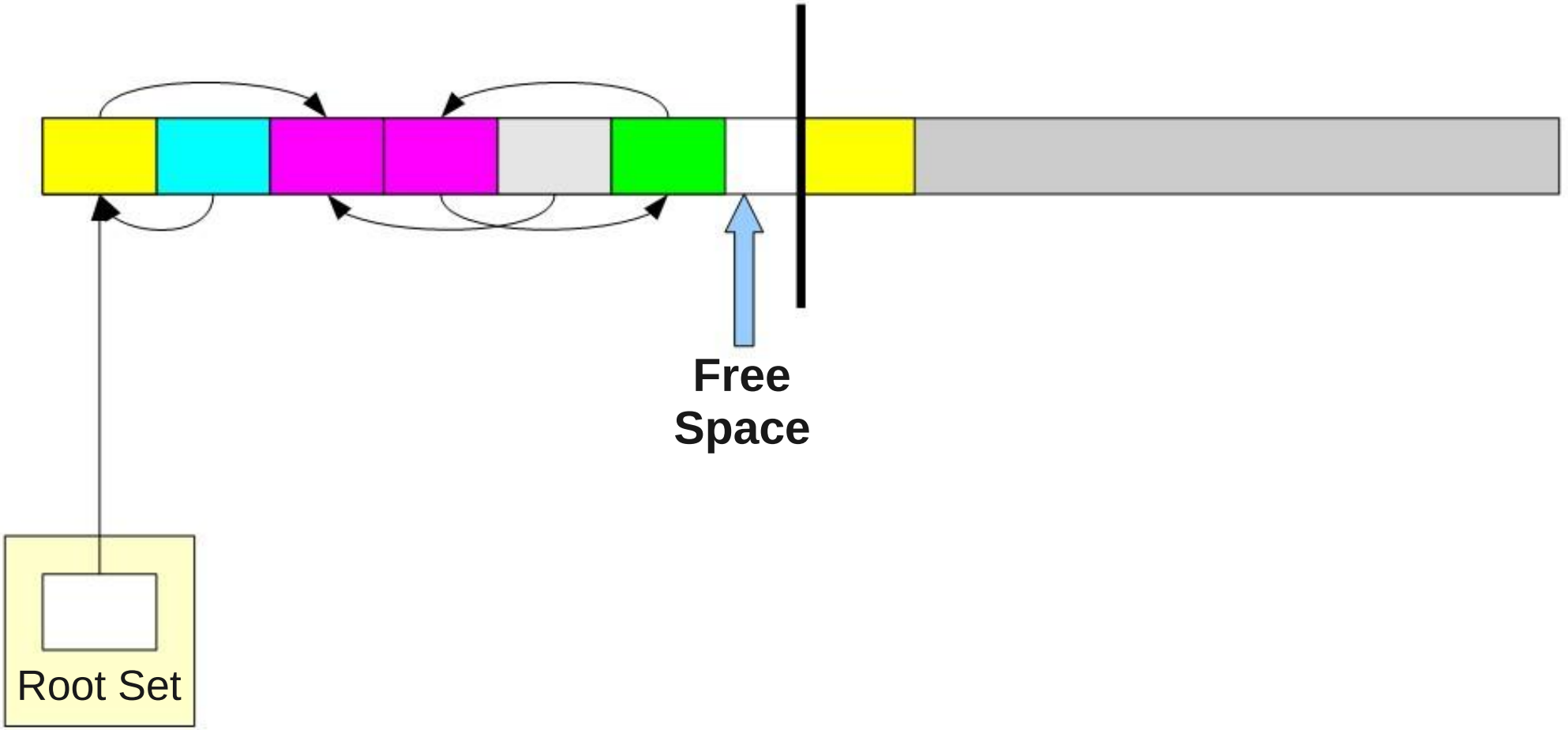


Out of space!

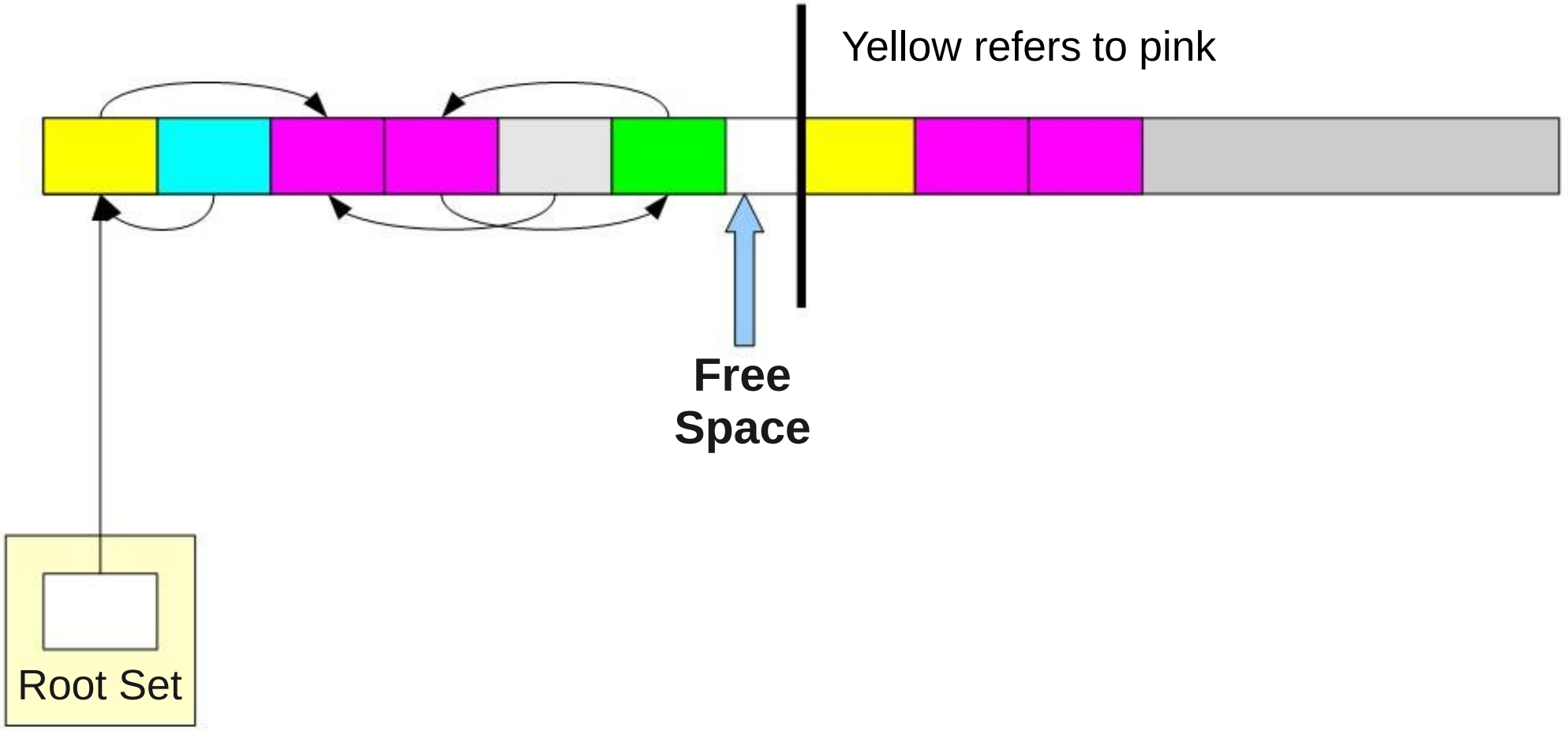
The Stop-and-Copy Collector



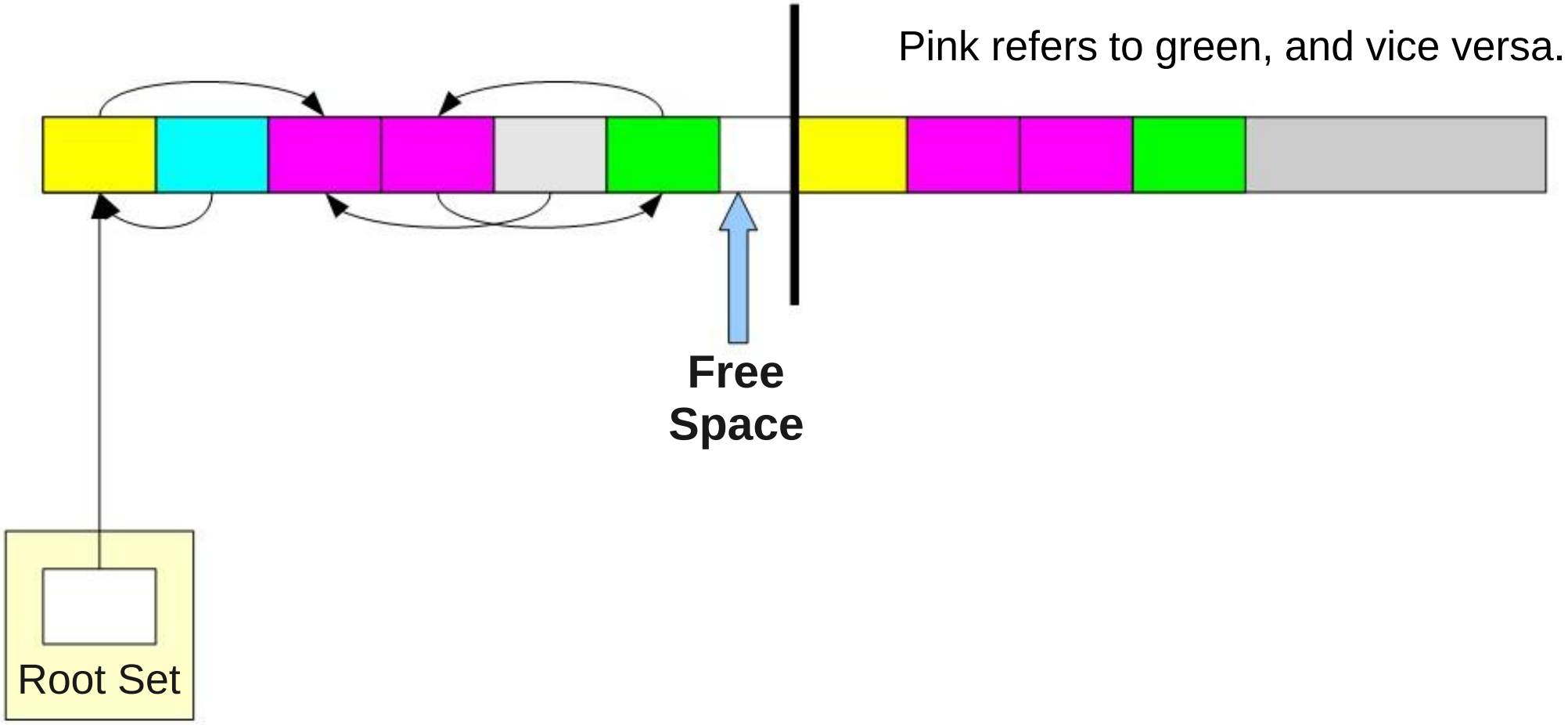
The Stop-and-Copy Collector



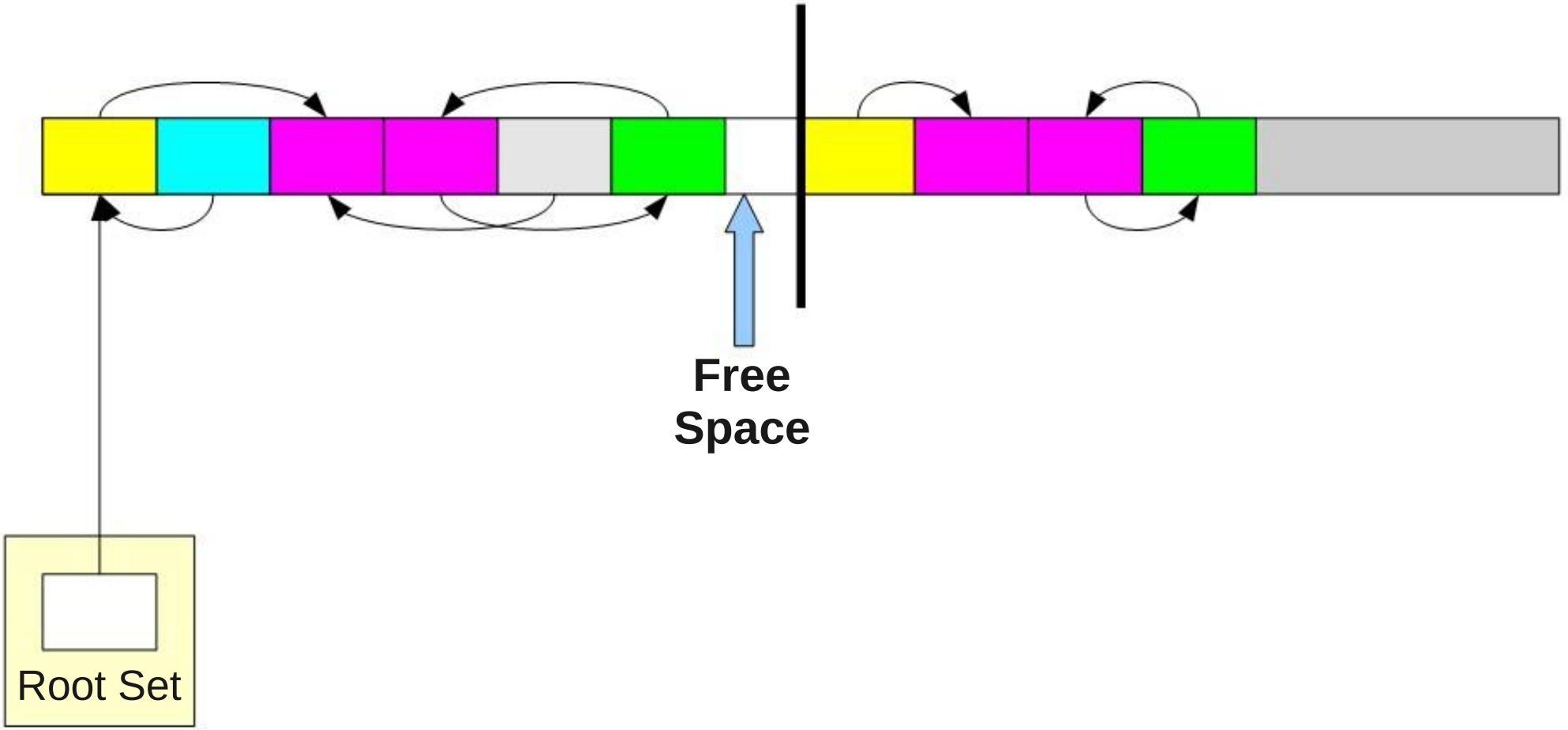
The Stop-and-Copy Collector



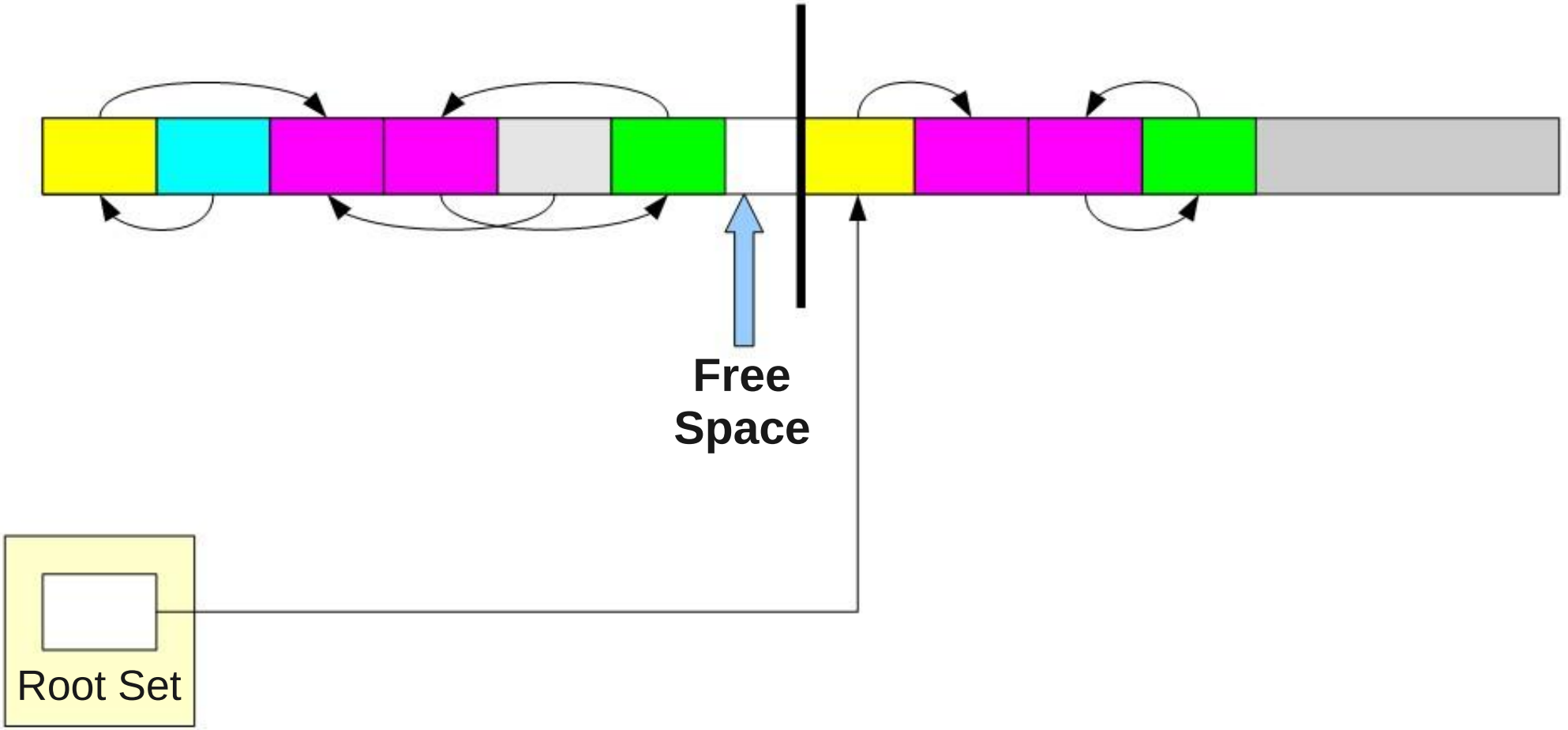
The Stop-and-Copy Collector



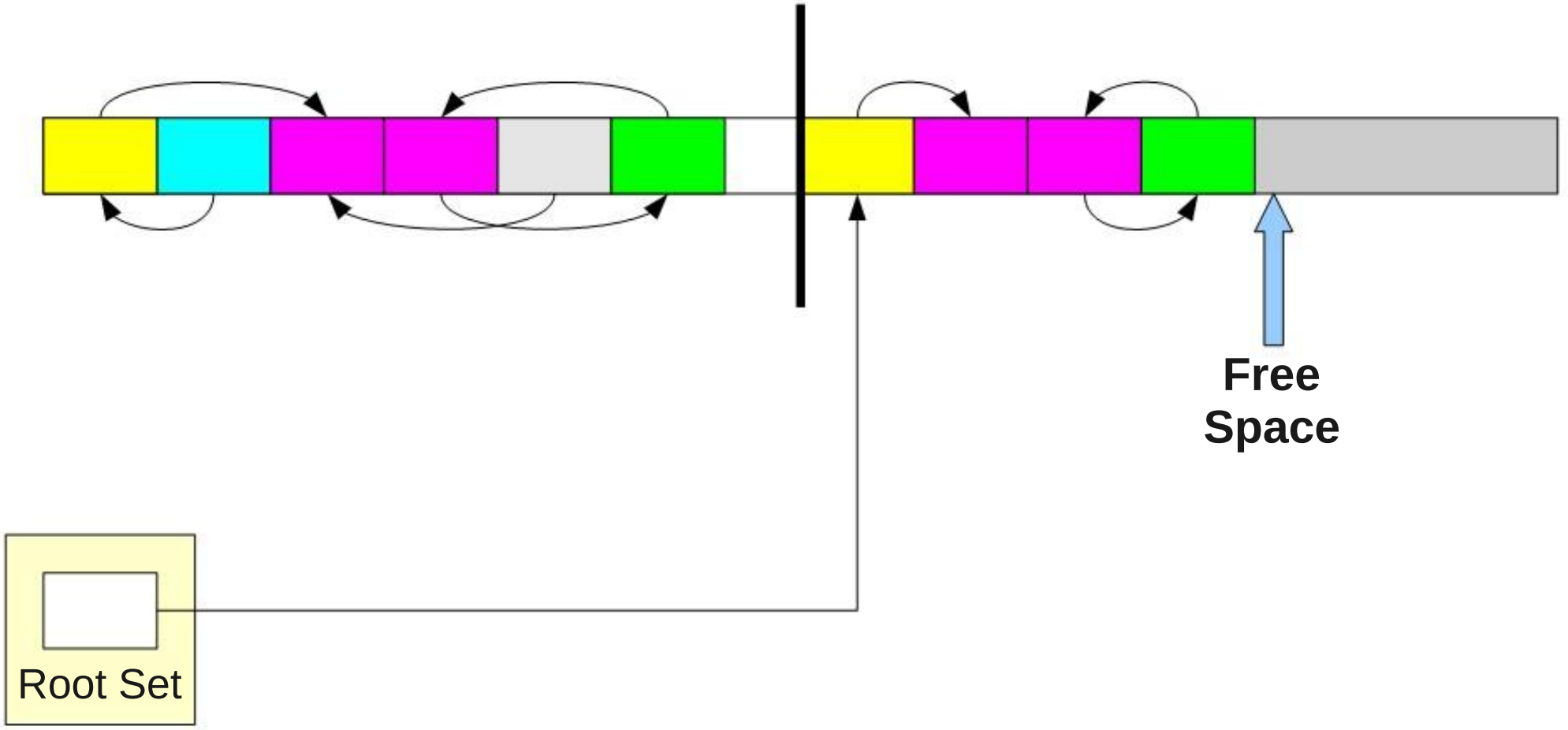
The Stop-and-Copy Collector



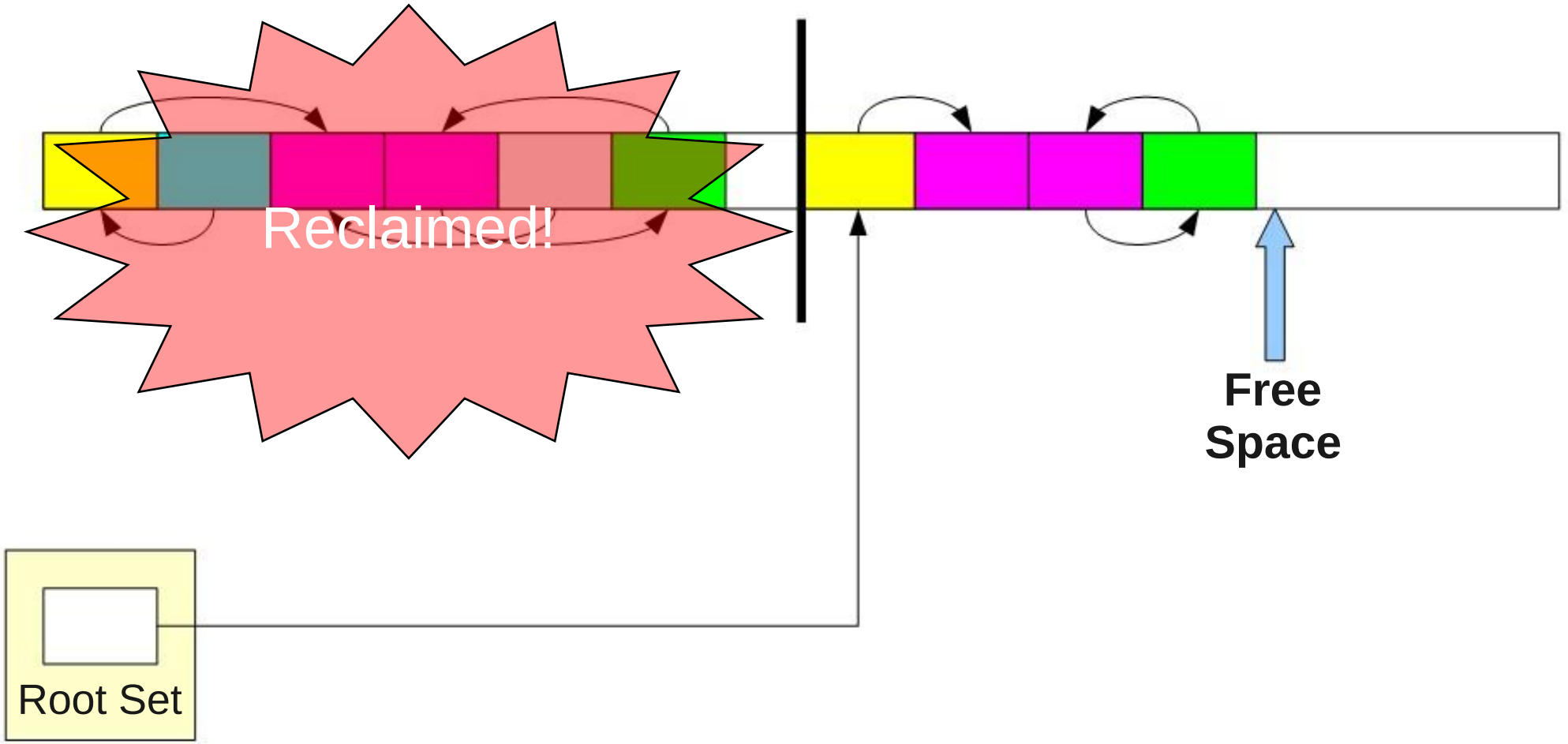
The Stop-and-Copy Collector



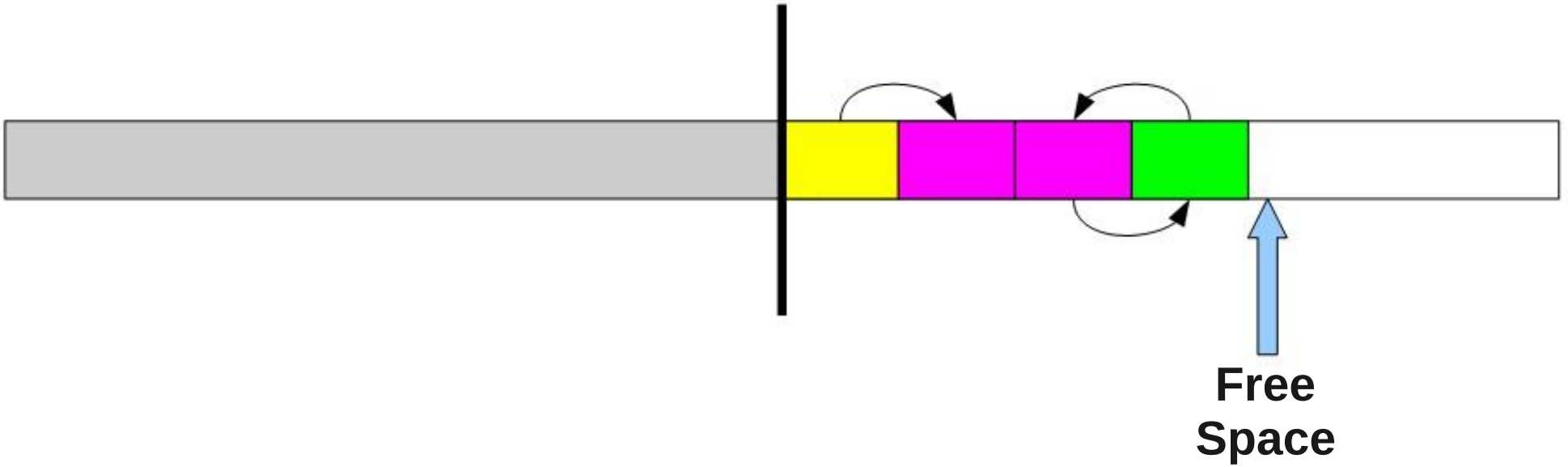
The Stop-and-Copy Collector



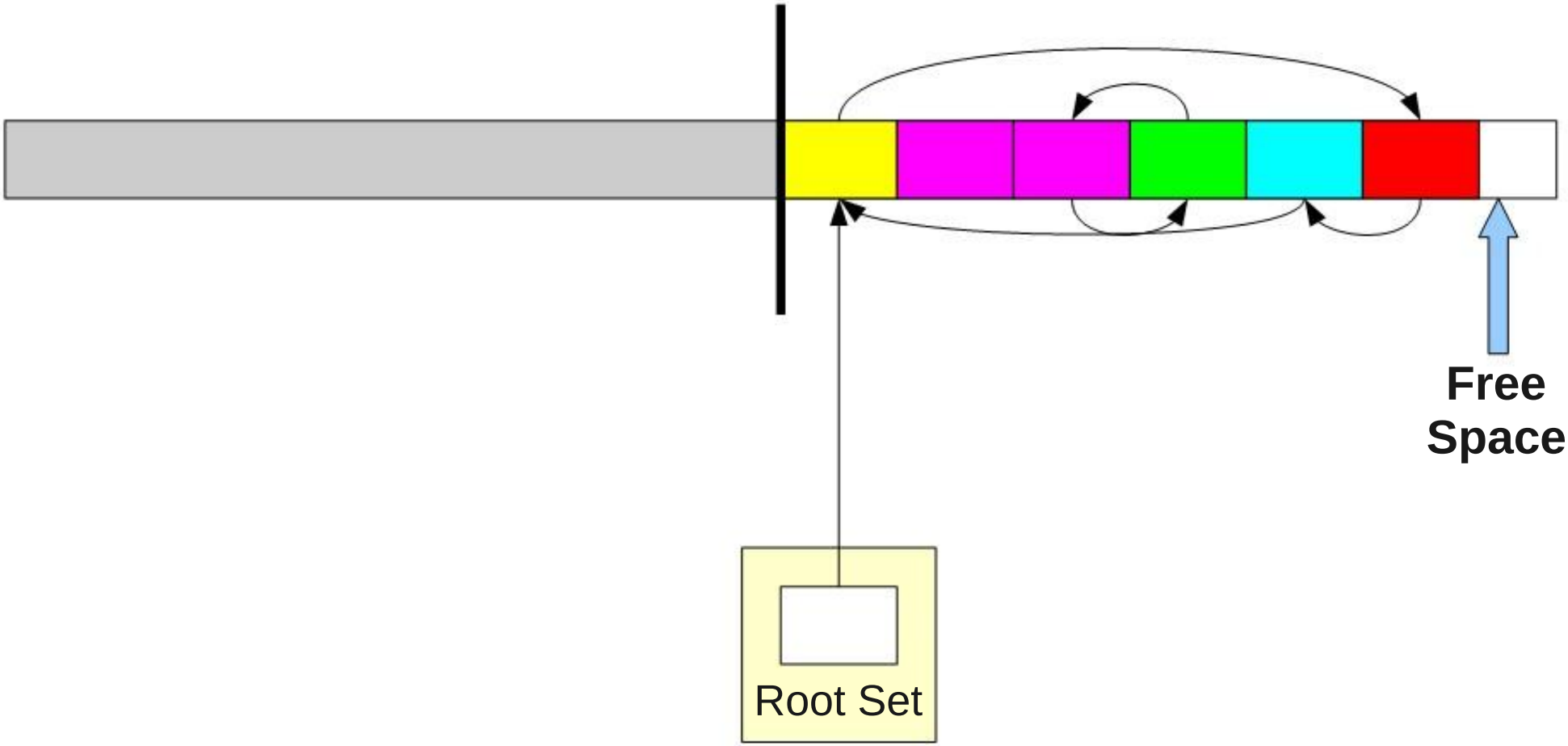
The Stop-and-Copy Collector



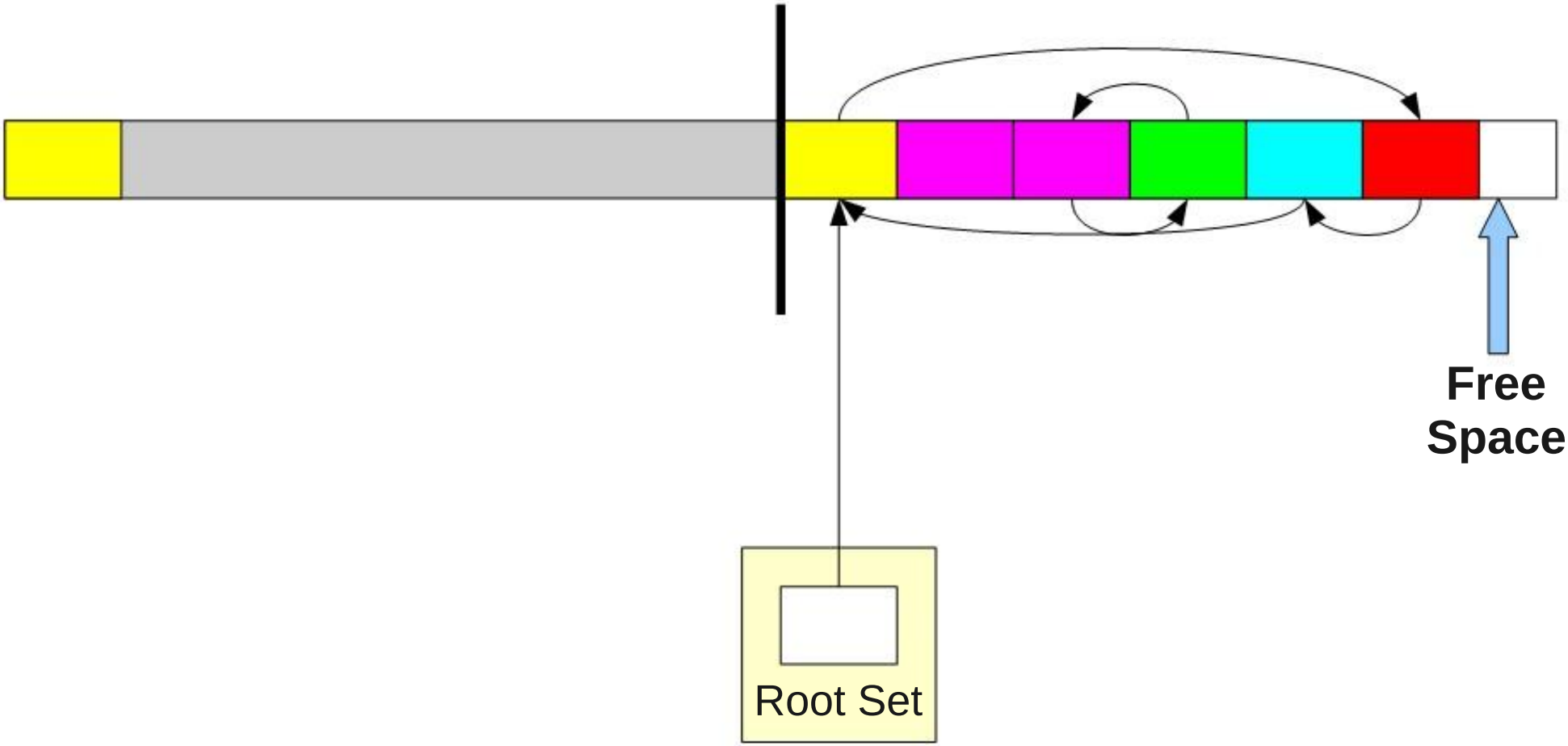
The Stop-and-Copy Collector



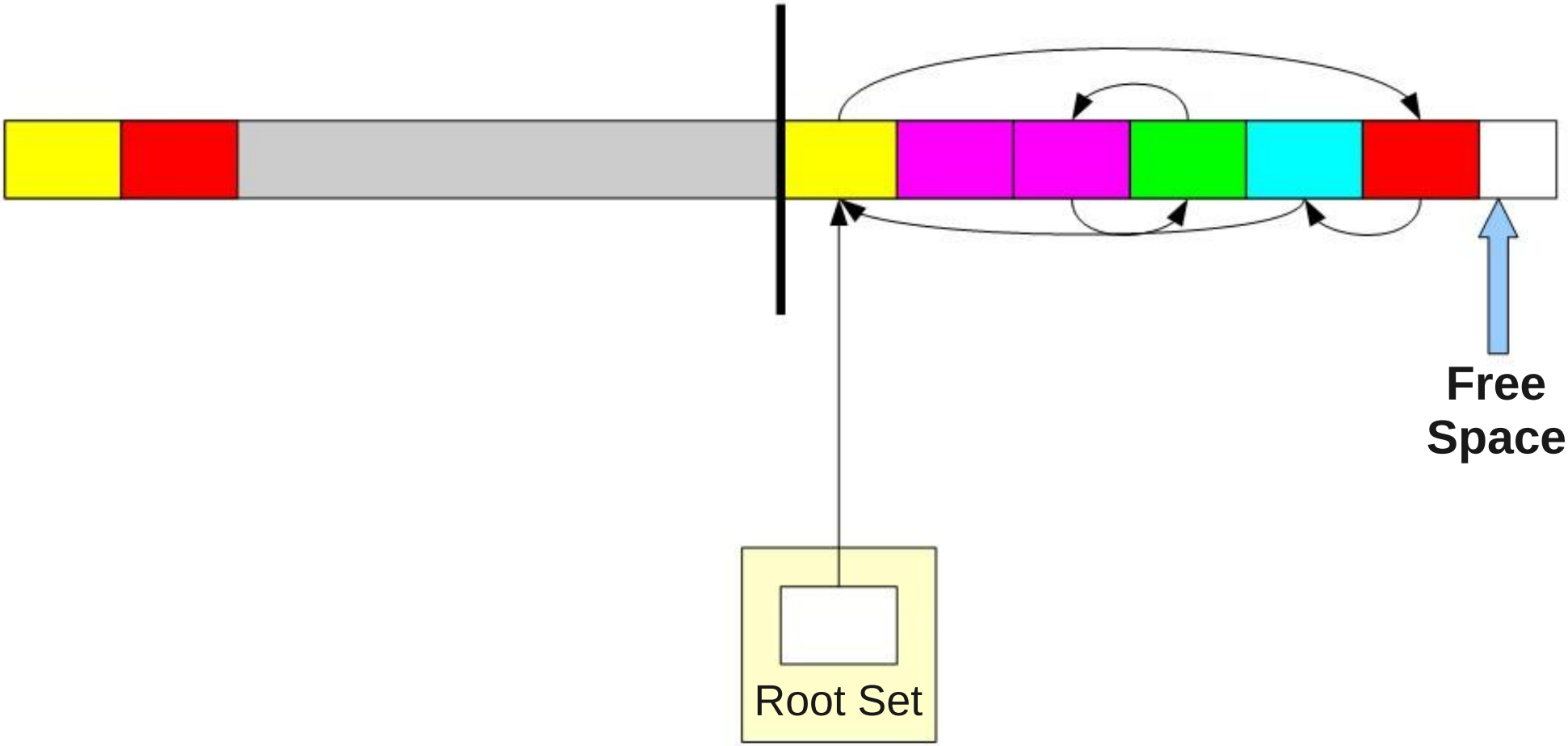
The Stop-and-Copy Collector



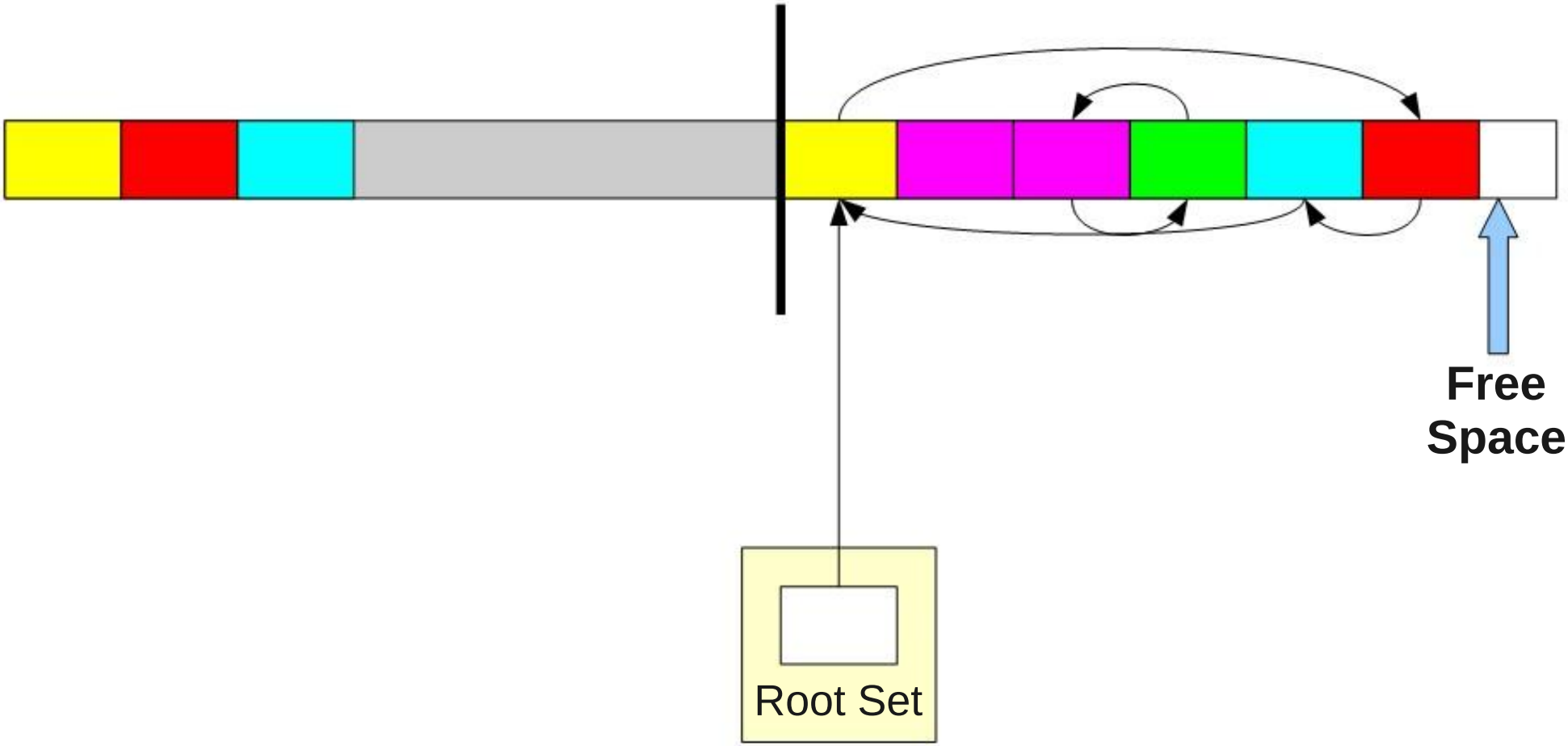
The Stop-and-Copy Collector



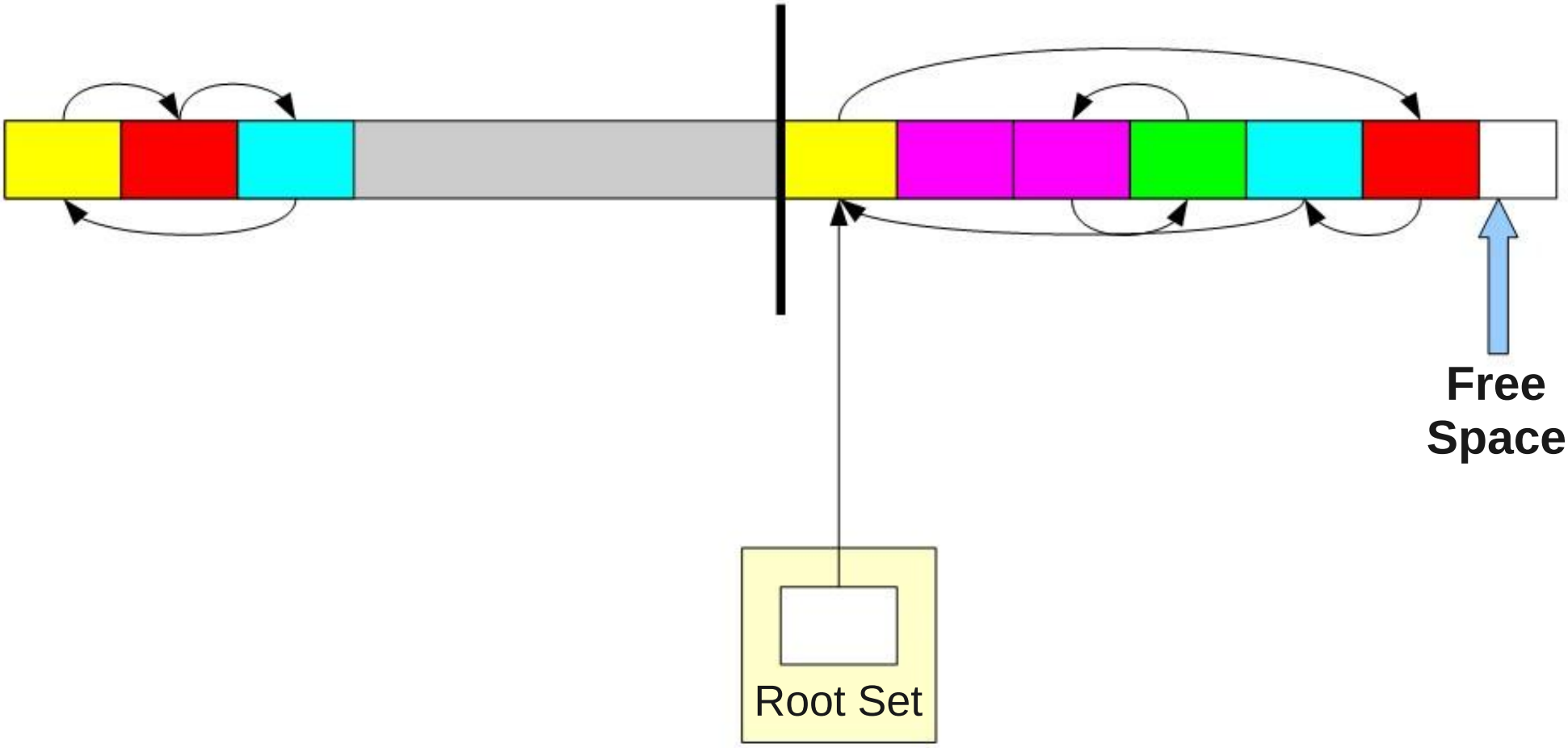
The Stop-and-Copy Collector



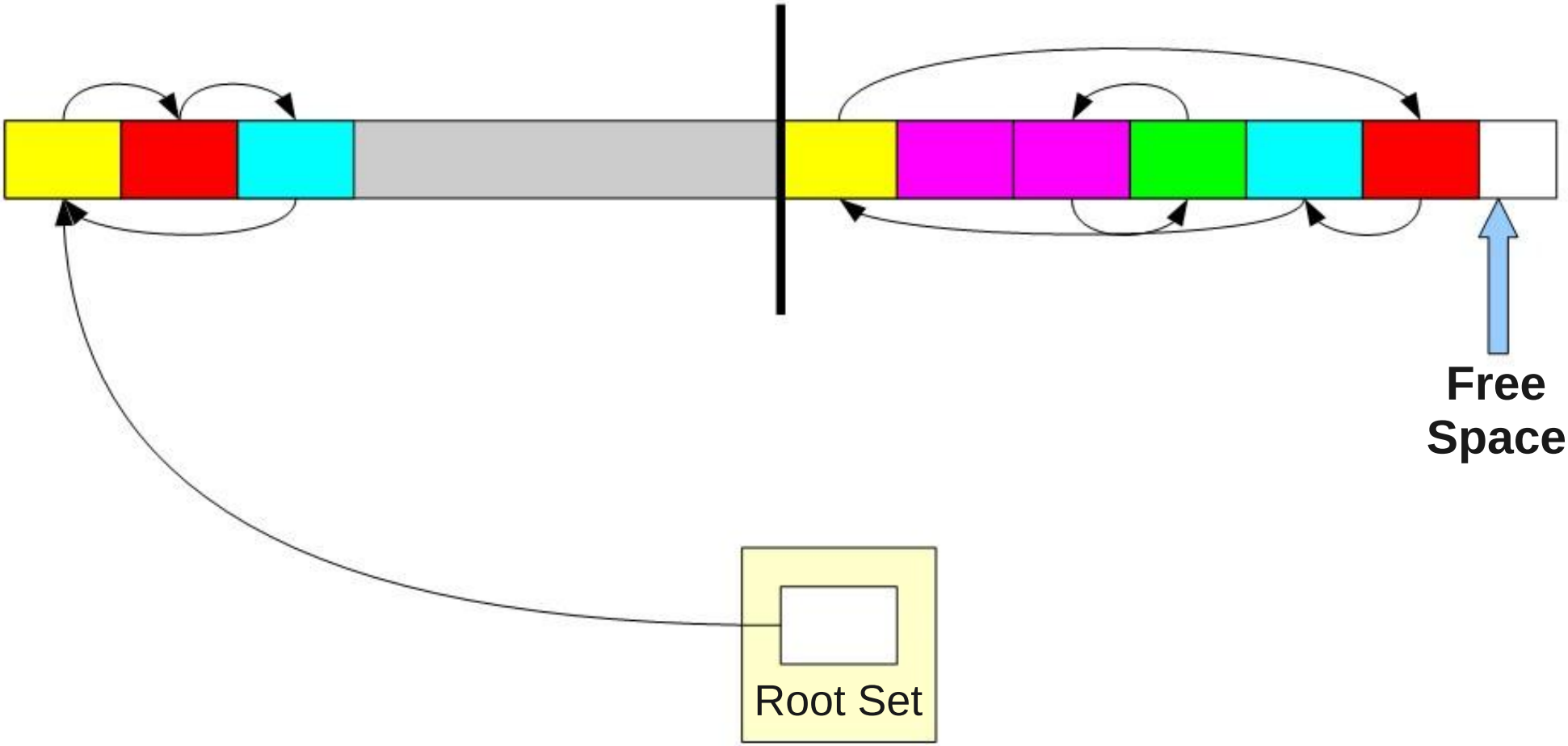
The Stop-and-Copy Collector



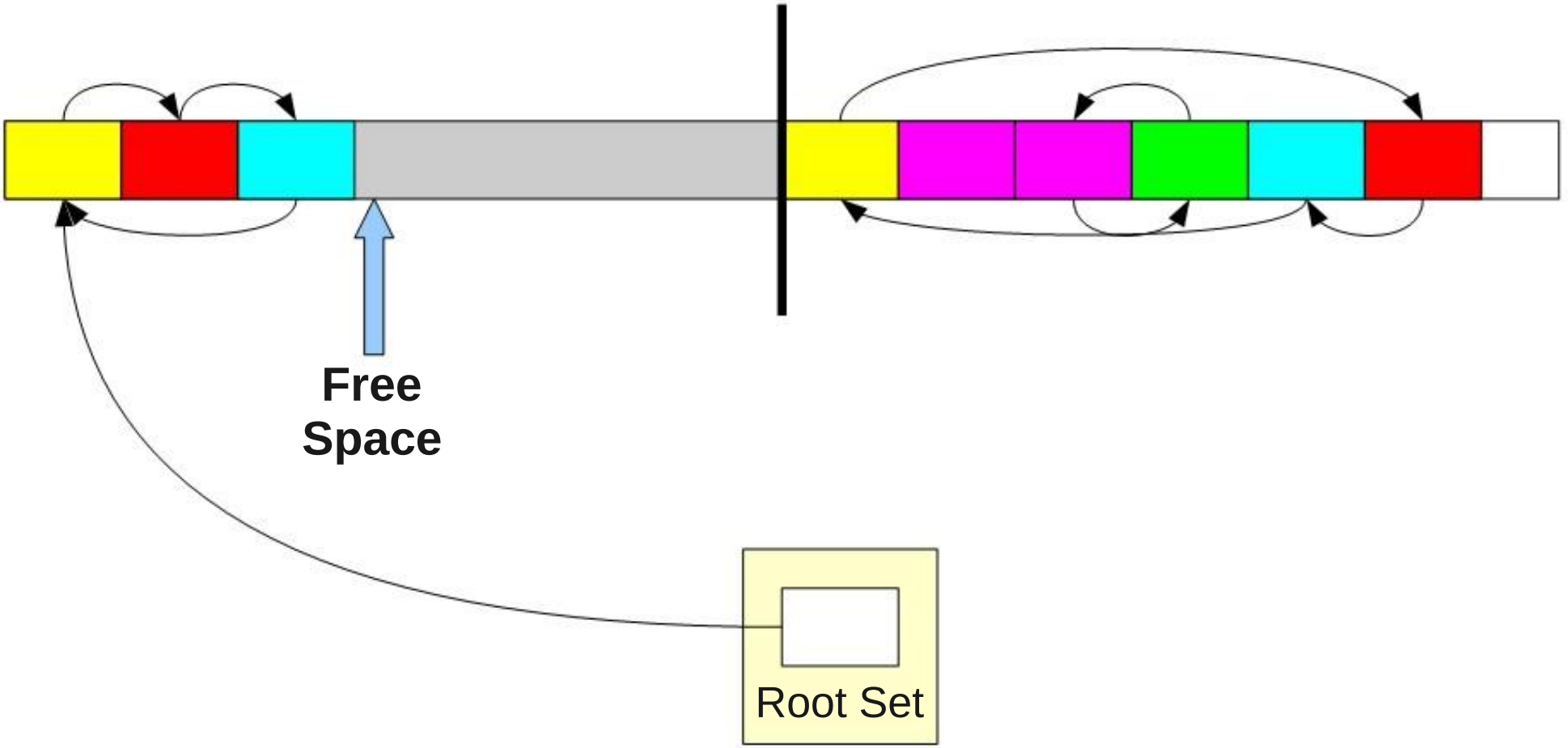
The Stop-and-Copy Collector



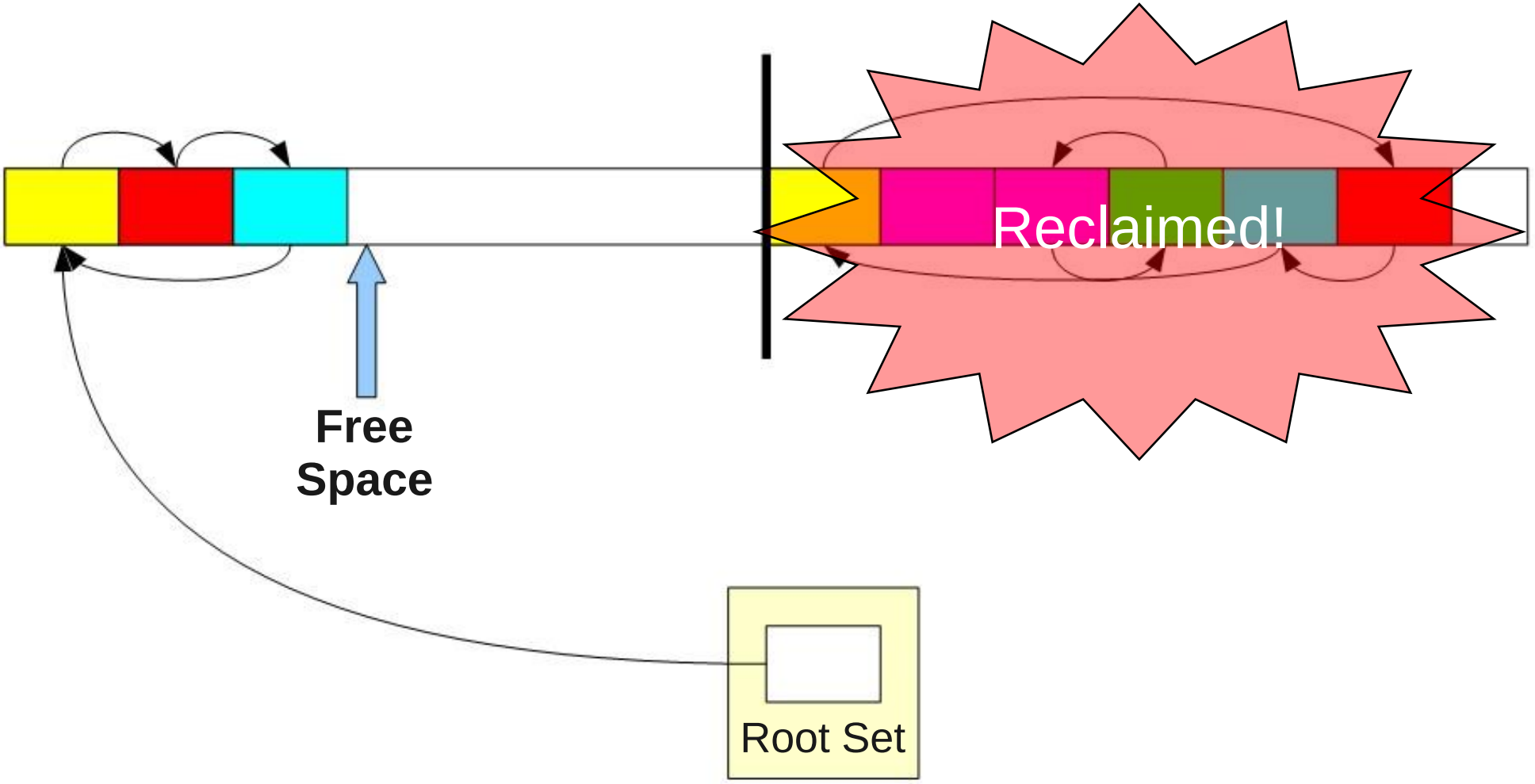
The Stop-and-Copy Collector



The Stop-and-Copy Collector



The Stop-and-Copy Collector



Stop-and-Copy in Detail

- Partition memory into two regions: the **old space** and the **new space**.
- Keep track of the next free address in the **new** space.
- To allocate **n** bytes of memory:
  - If **n** bytes space exist at the free space pointer, use those bytes and advance the pointer.
  - Otherwise, do a **copy** step.
- To execute a **copy** step:
  - For each object in the root set:
    - Copy that object over to the start of the **old** space.
    - Recursively copy over all objects reachable from that object.
  - Adjust the pointers in the **old** space and root set to point to new locations.
  - Exchange the roles of the **old** and **new** spaces.

Analysis of Stop-and-Copy

- **Advantages**:
  - Implementation **simplicity** (compared to mark-and-sweep).
  - **Fast and simple memory allocation**. (Sequentially)
  - **Excellent locality**; ordering of copied objects places similar objects near each other.
- **Disadvantages**:
  - Requires **half of memory to be free at all times**. May need to run the algorithm to reclaim memory frequently.

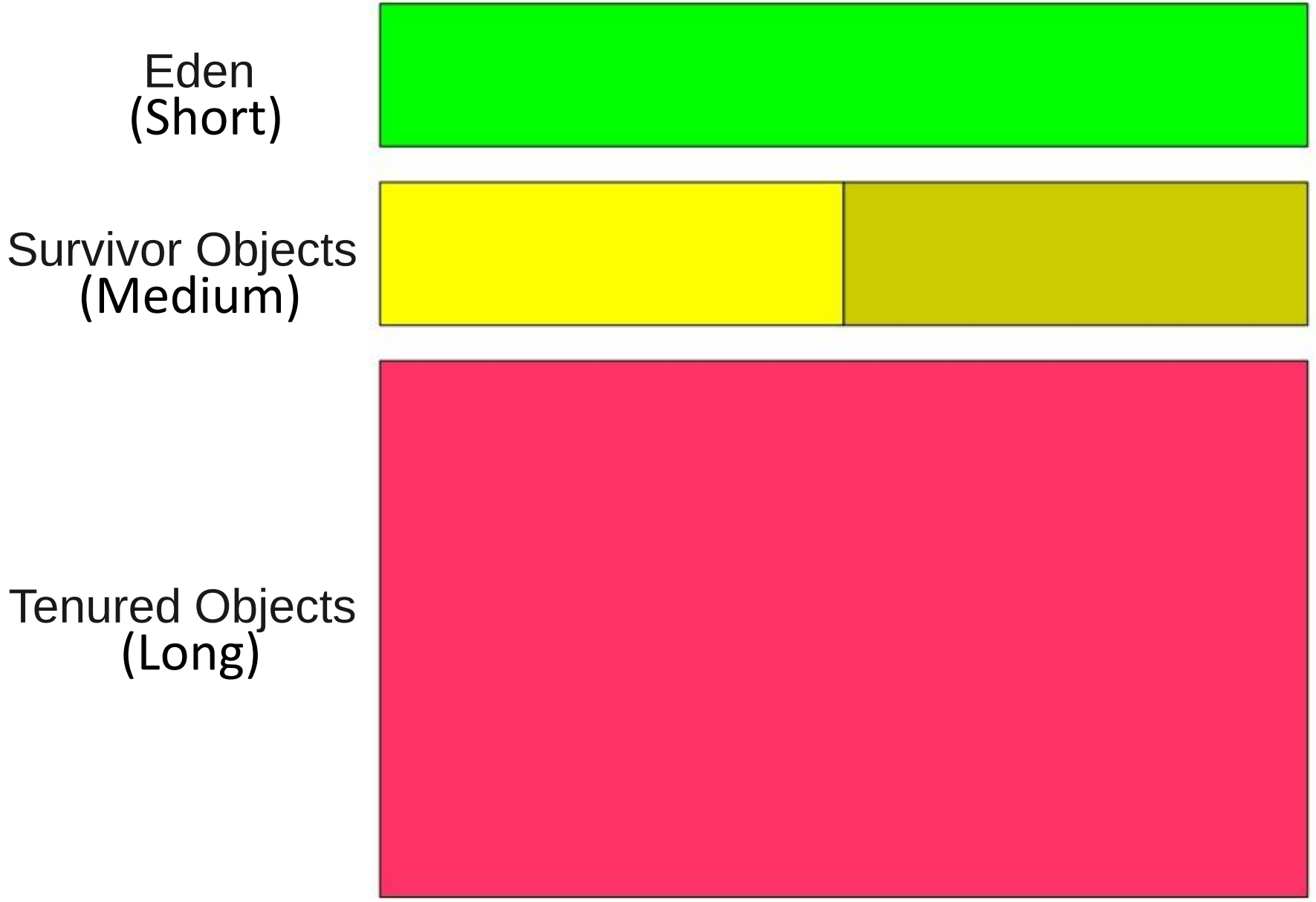
Generational Garbage Collection

- A method to reinforce the existing garbage collection techniques, e.g. Reference Counting, Mark-and-Sweep, and Stop-and-Copy

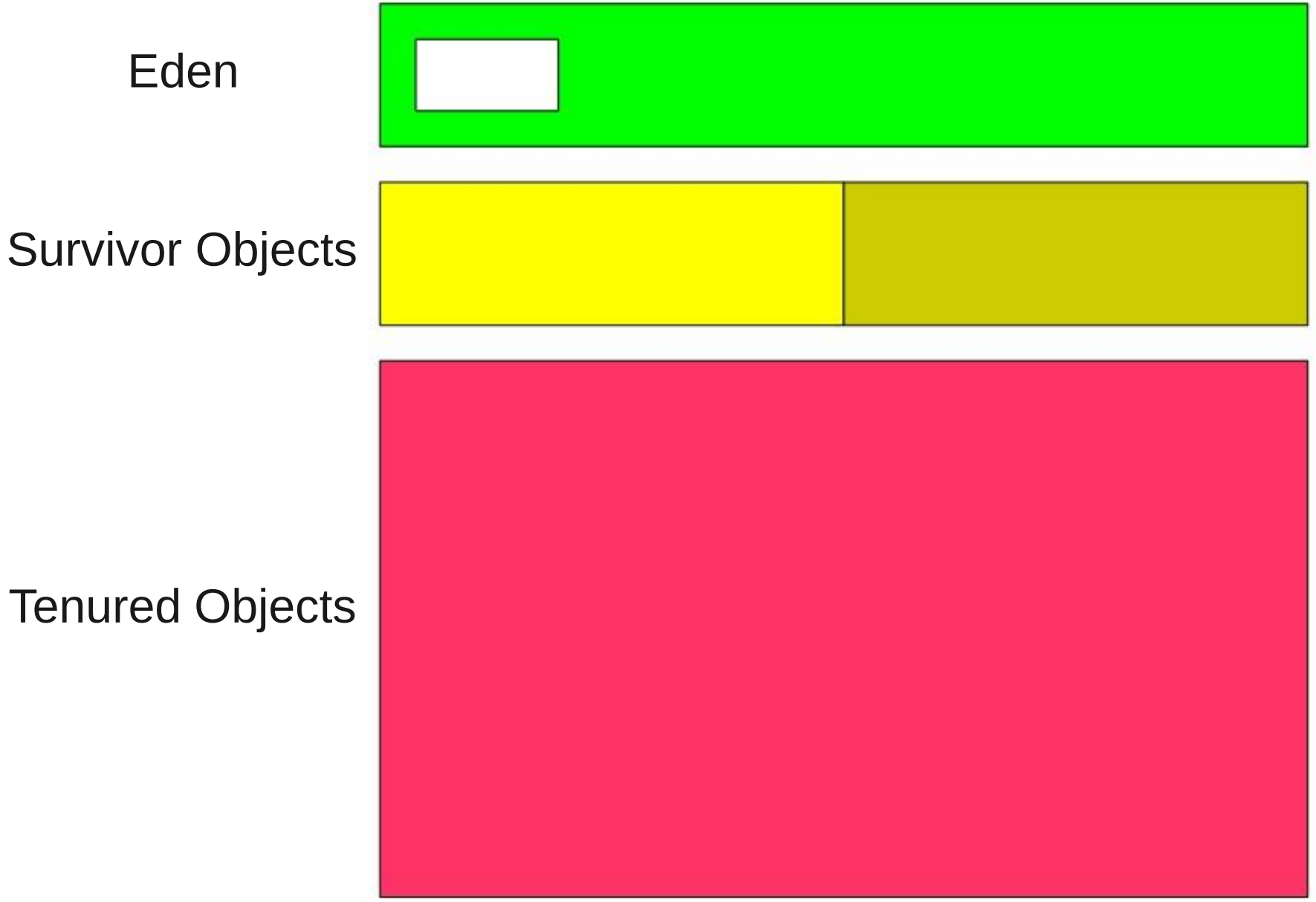
- The Motto of Garbage Collection: **Objects Die Young**.

- Most objects have extremely short lifetimes.
  - Temporary objects used to construct larger objects.
- Optimize garbage collection to reclaim young objects while spending less time on older objects.

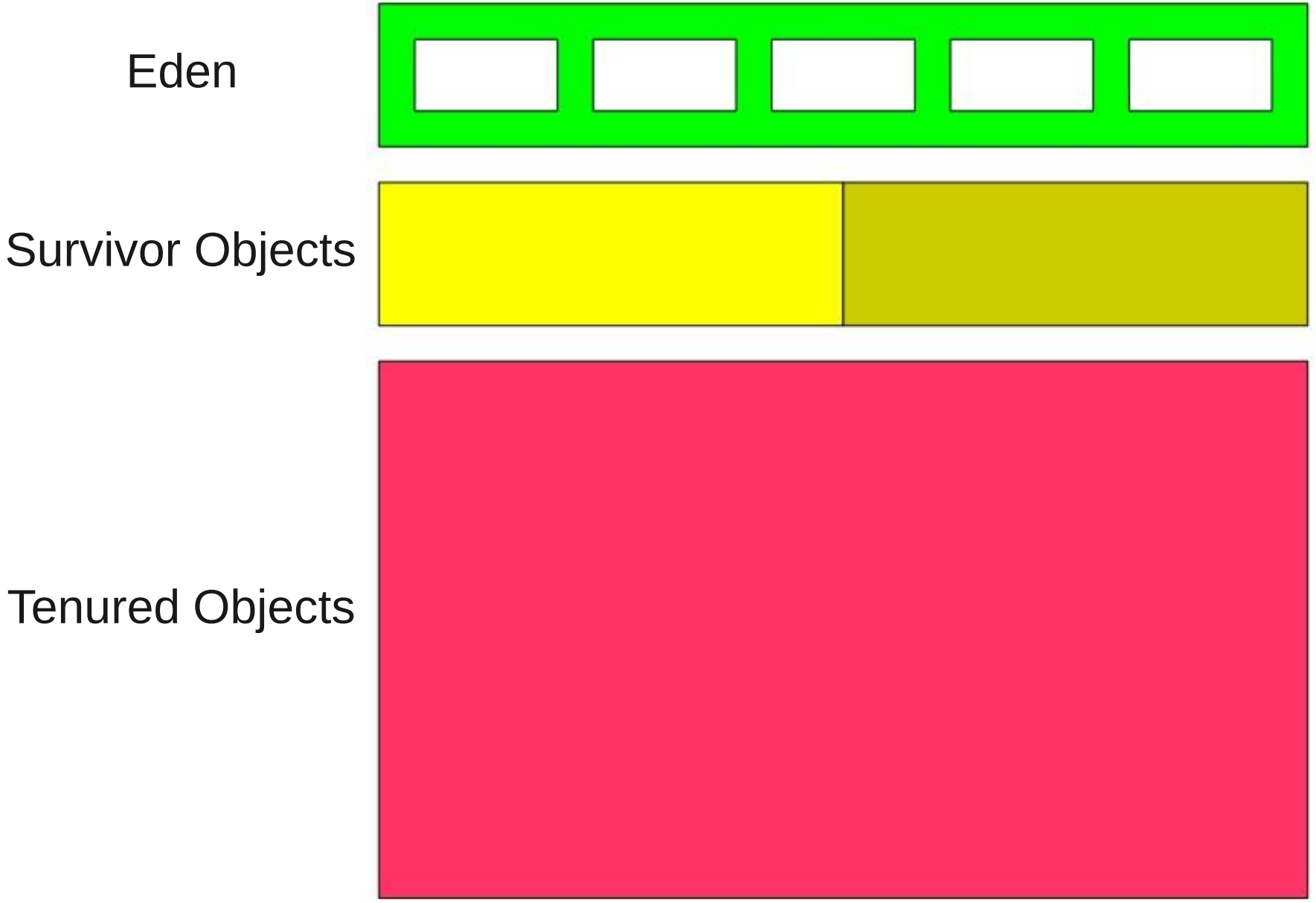
Garbage Collection in Java



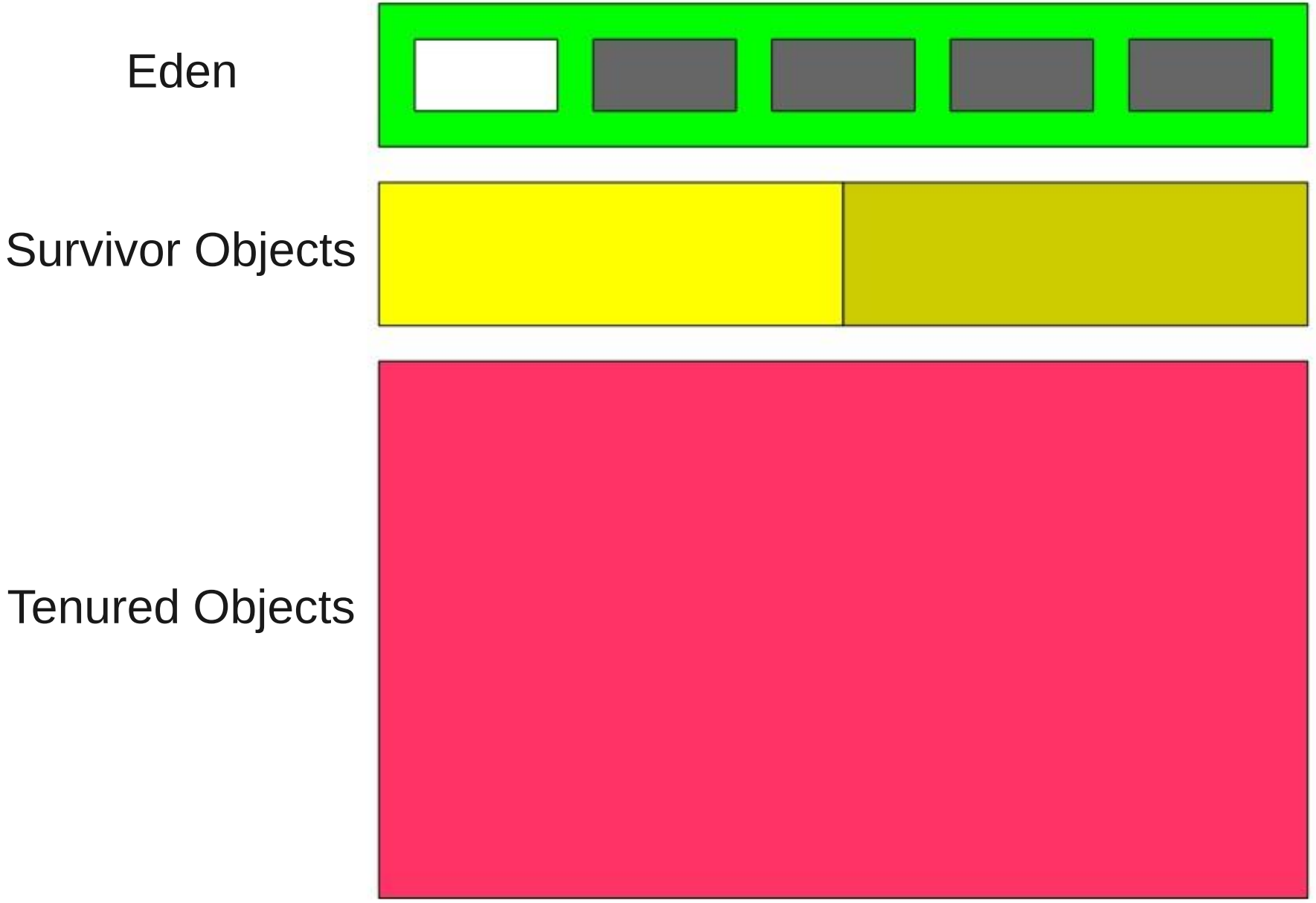
Garbage Collection in Java



Garbage Collection in Java

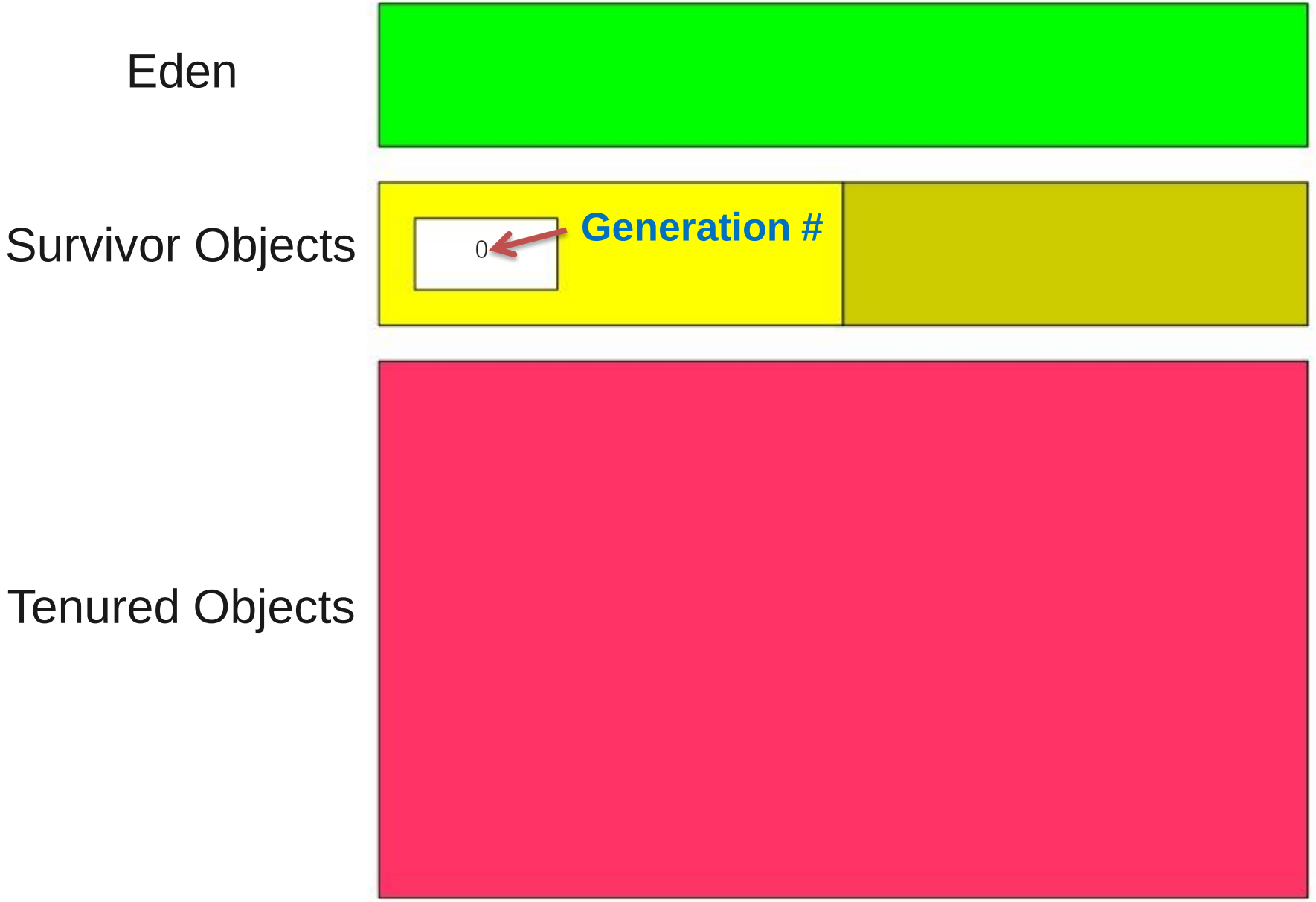


Garbage Collection in Java

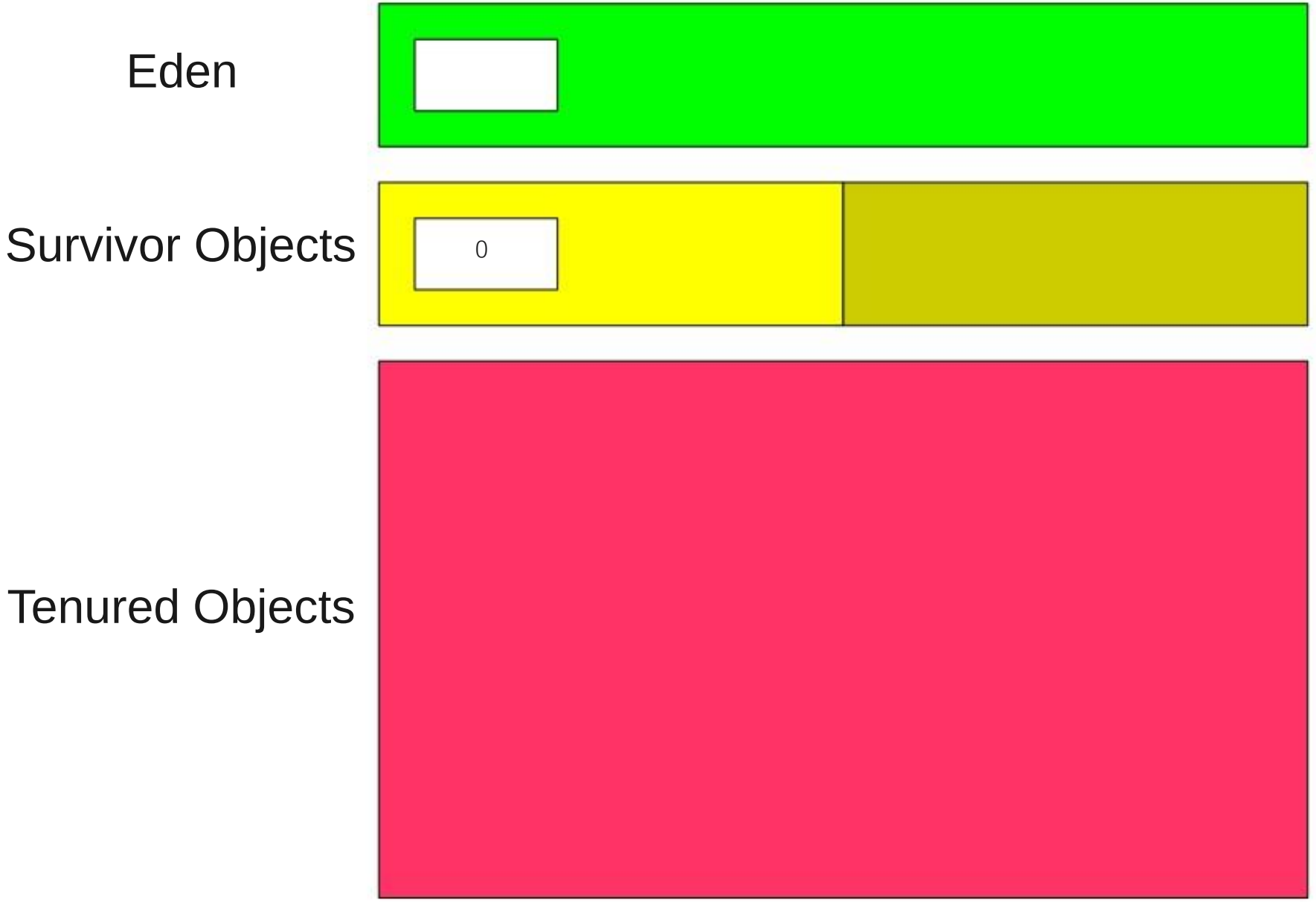


Garbage Collection in Java

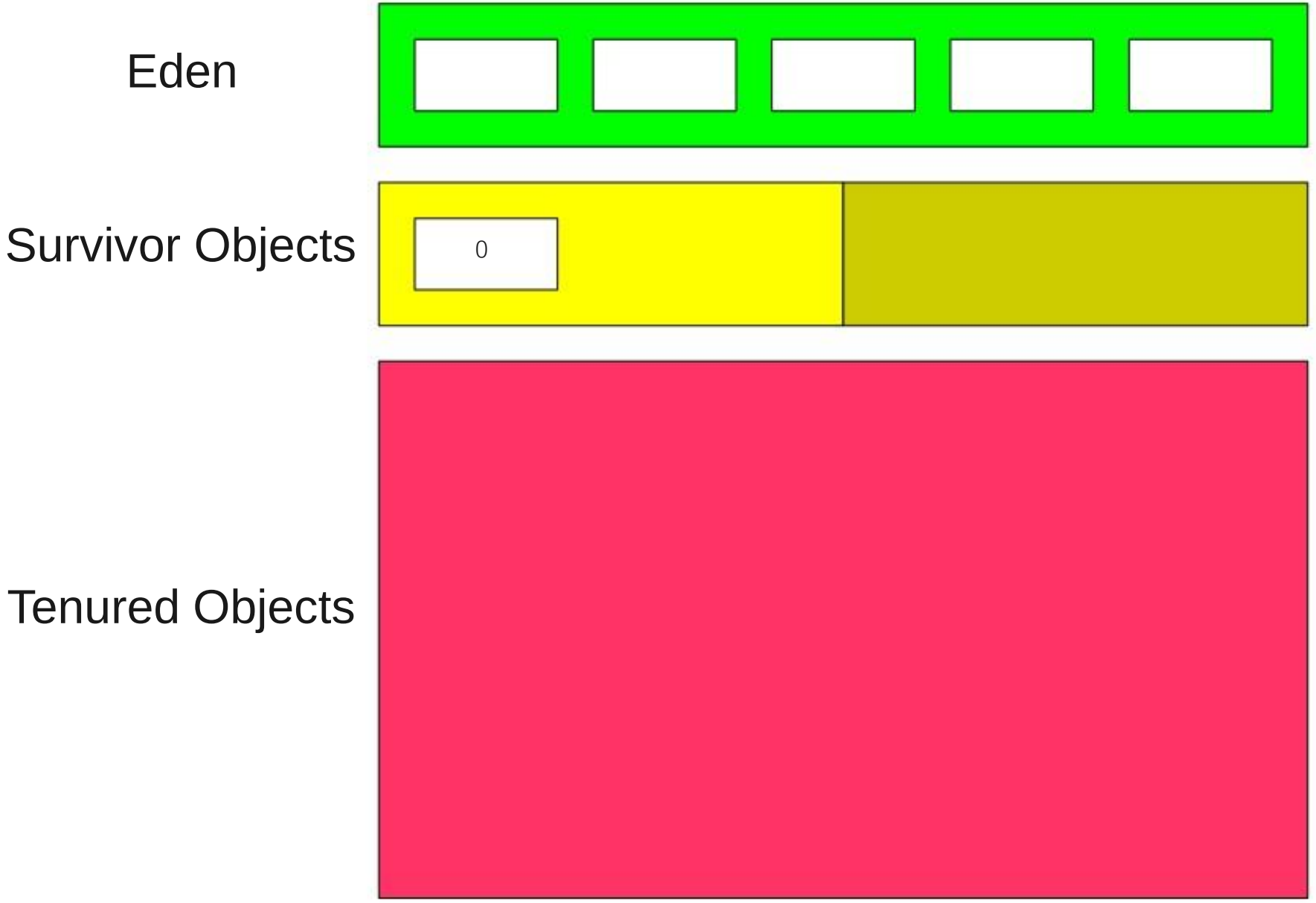
Garbage Collection in Java



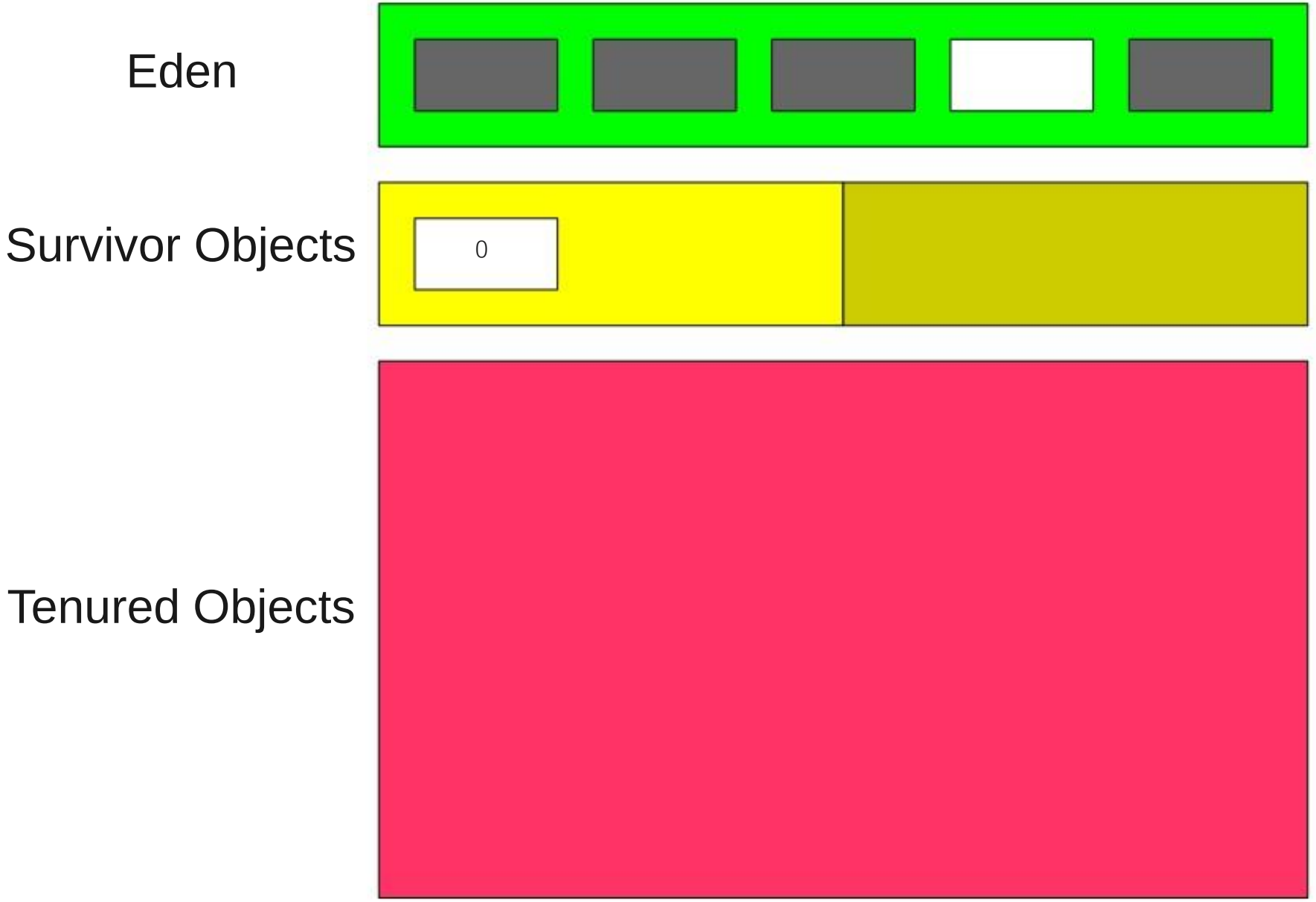
Garbage Collection in Java



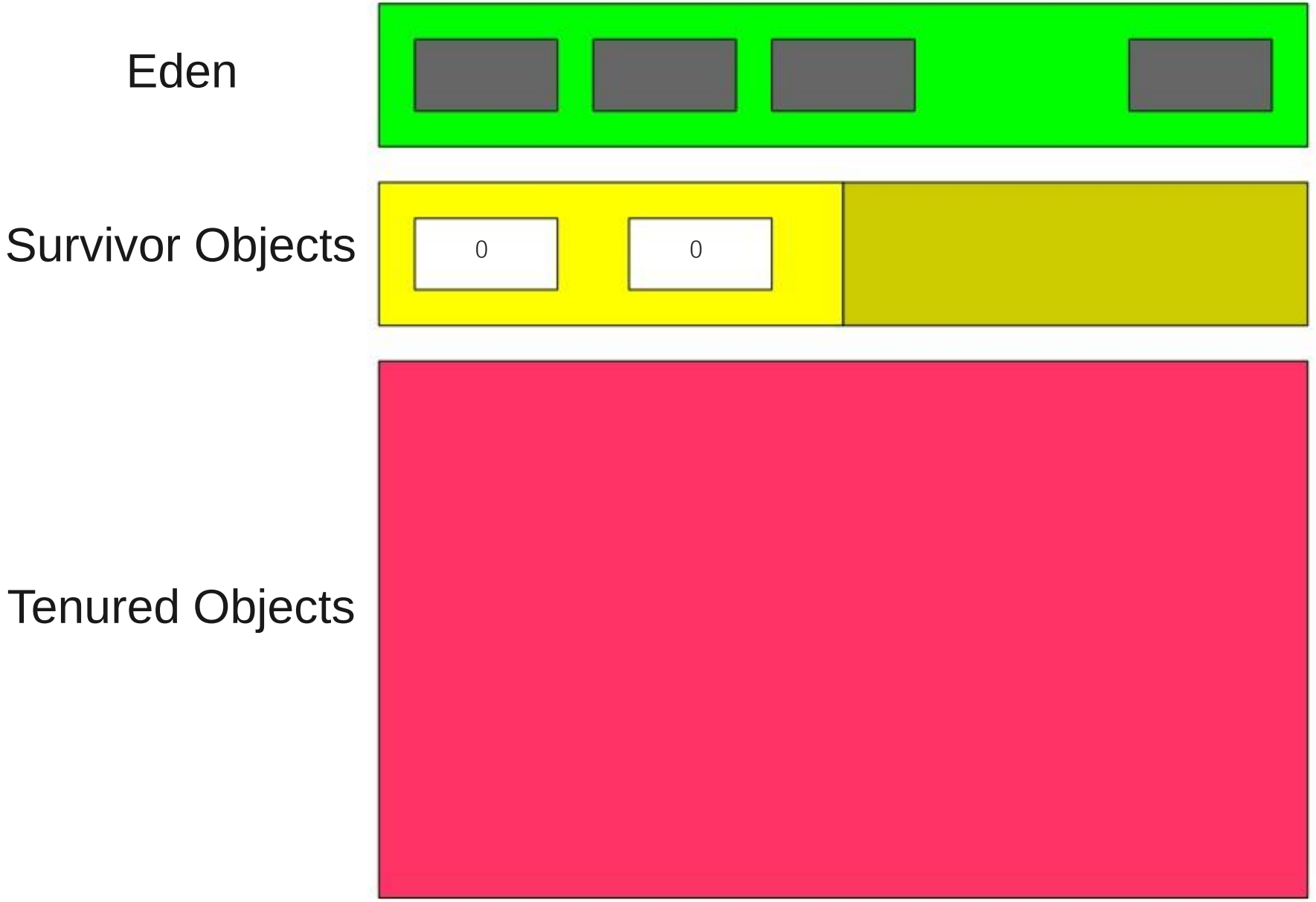
Garbage Collection in Java



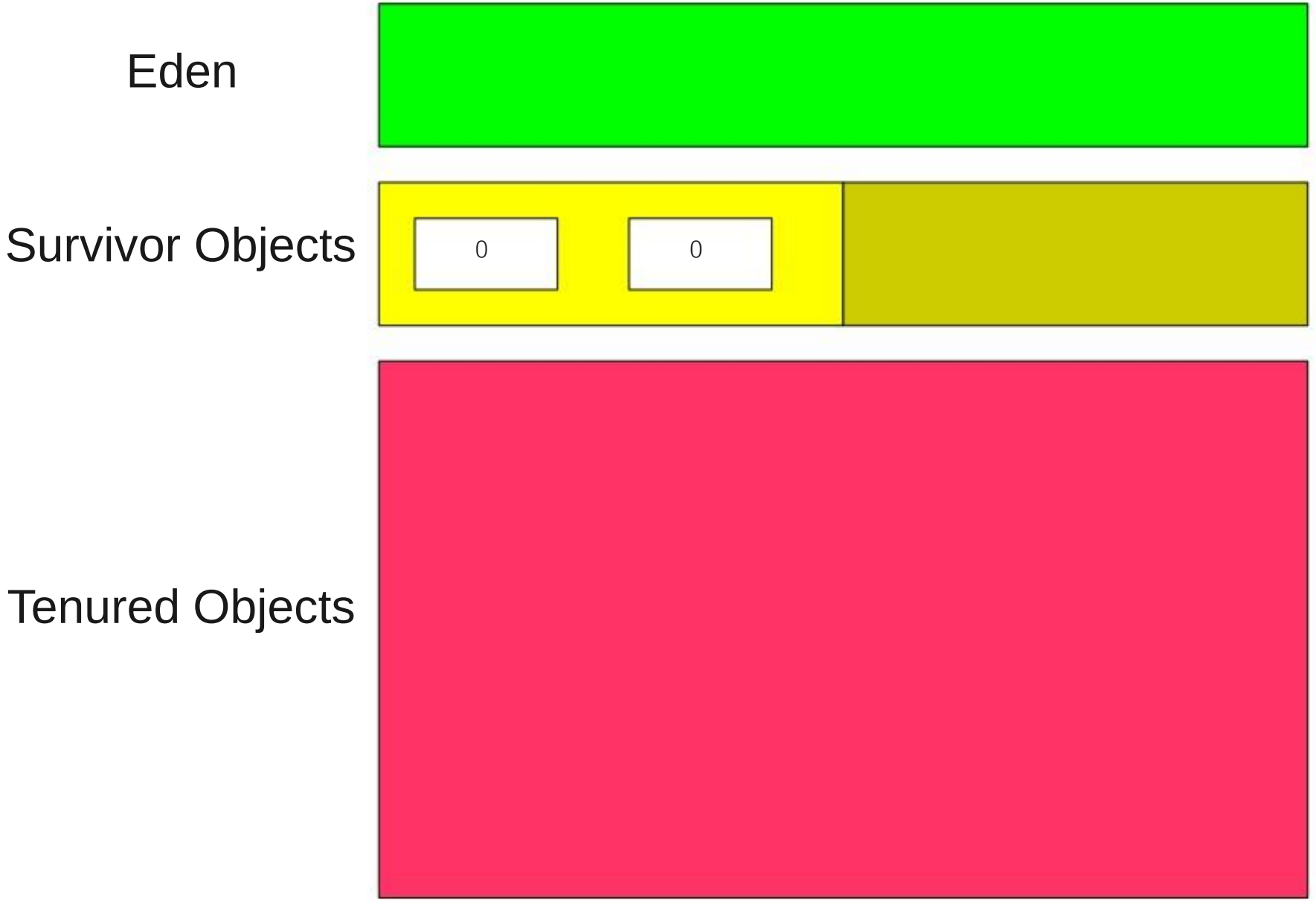
Garbage Collection in Java



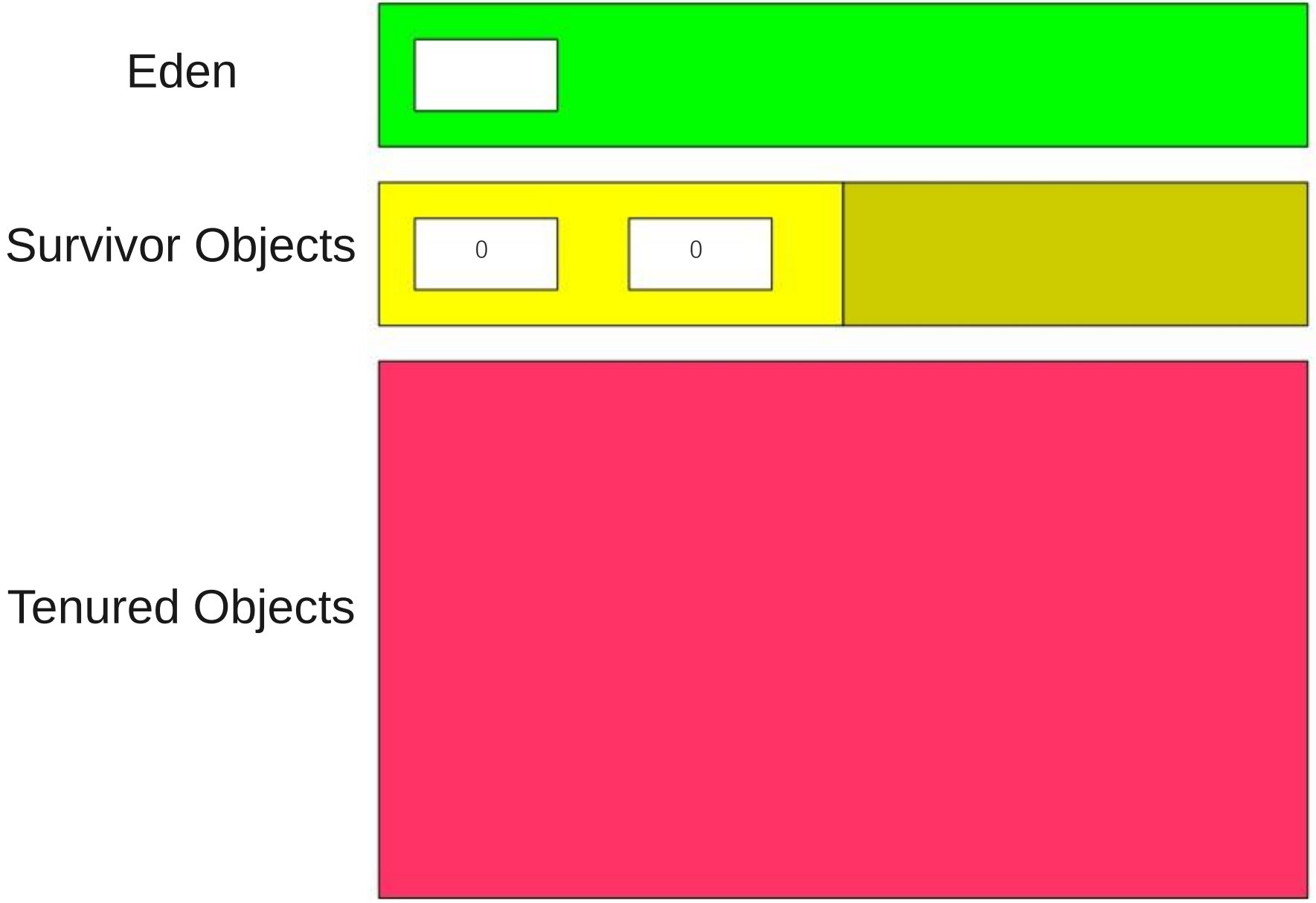
Garbage Collection in Java



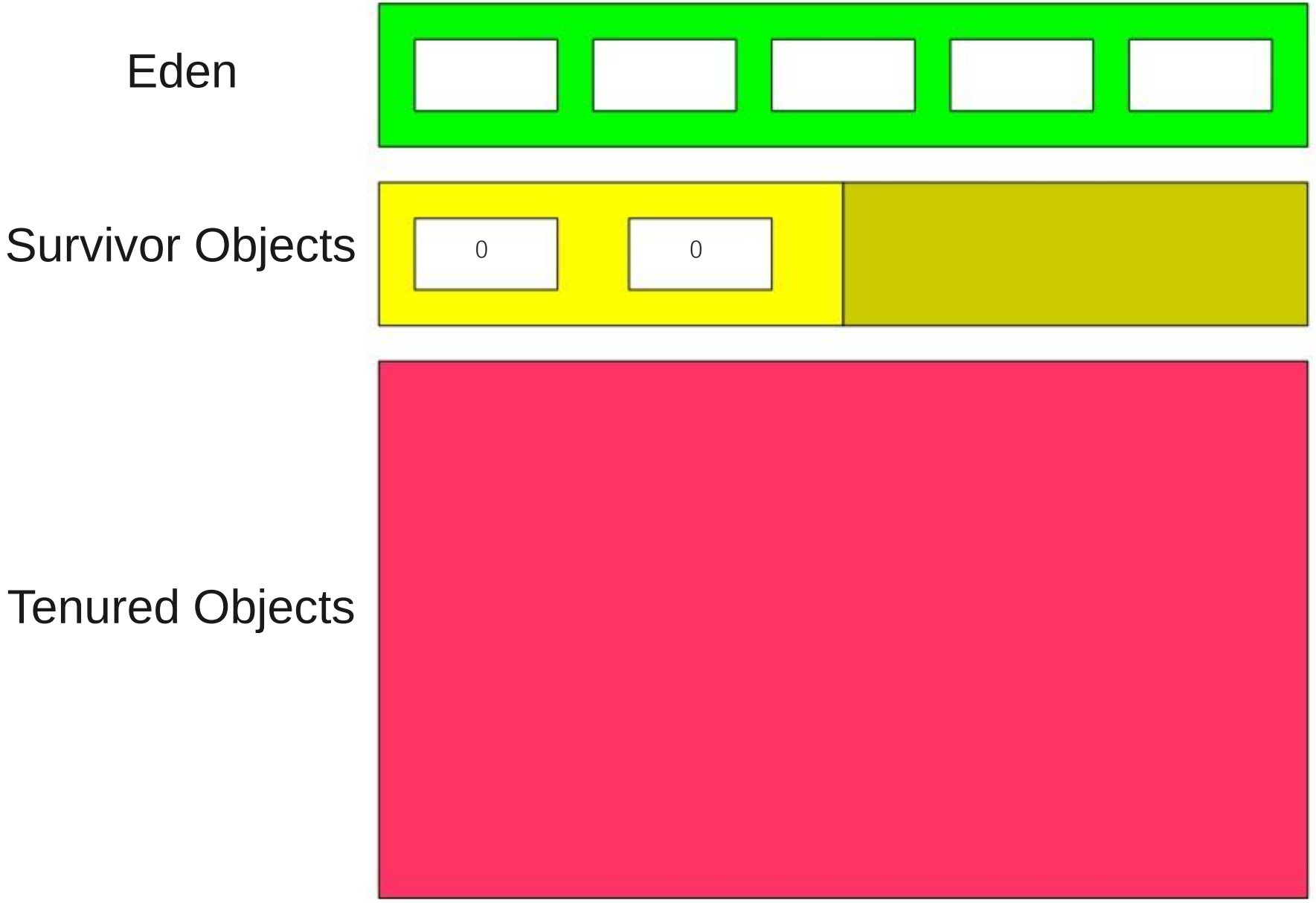
Garbage Collection in Java



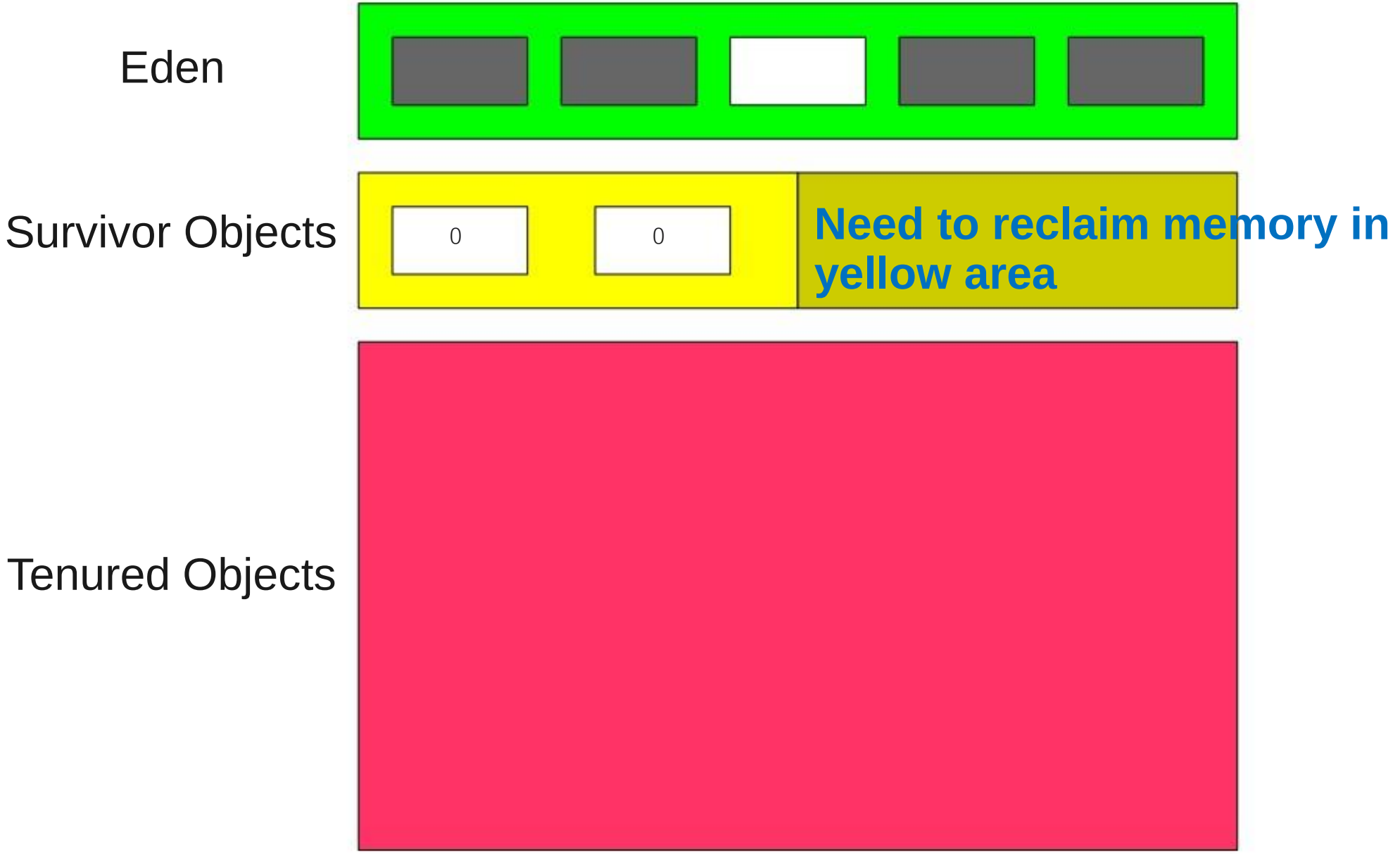
Garbage Collection in Java



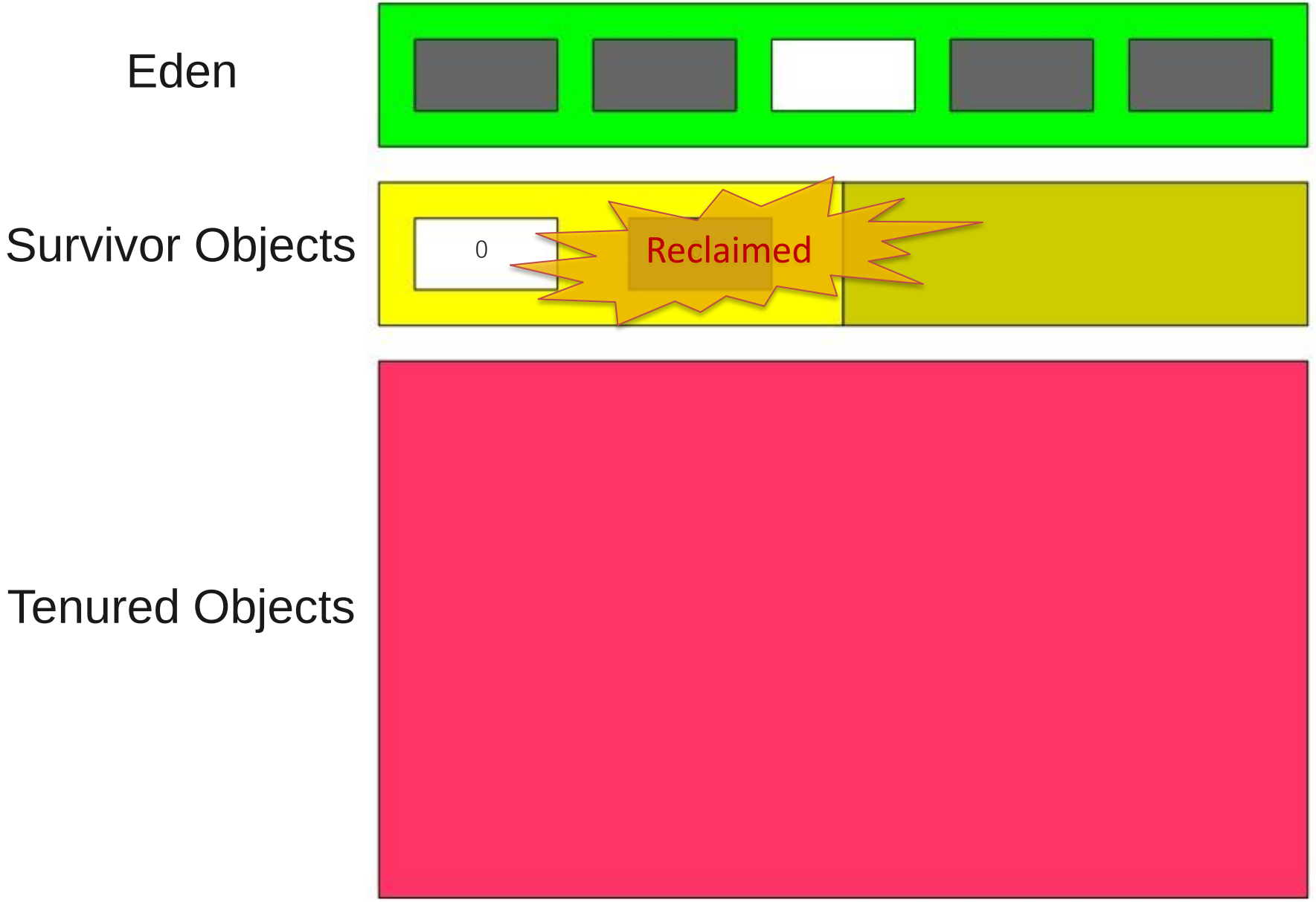
Garbage Collection in Java



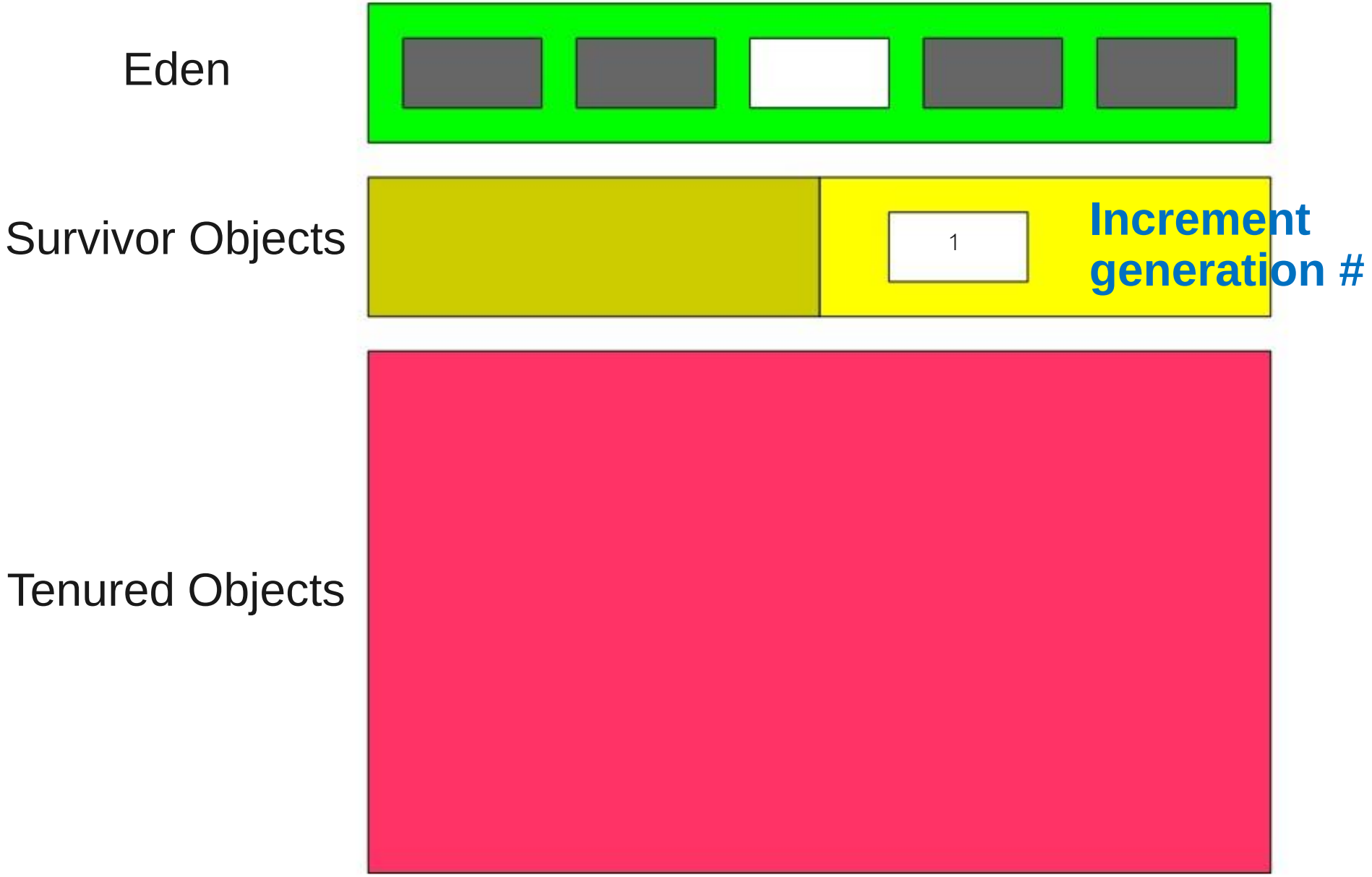
Garbage Collection in Java



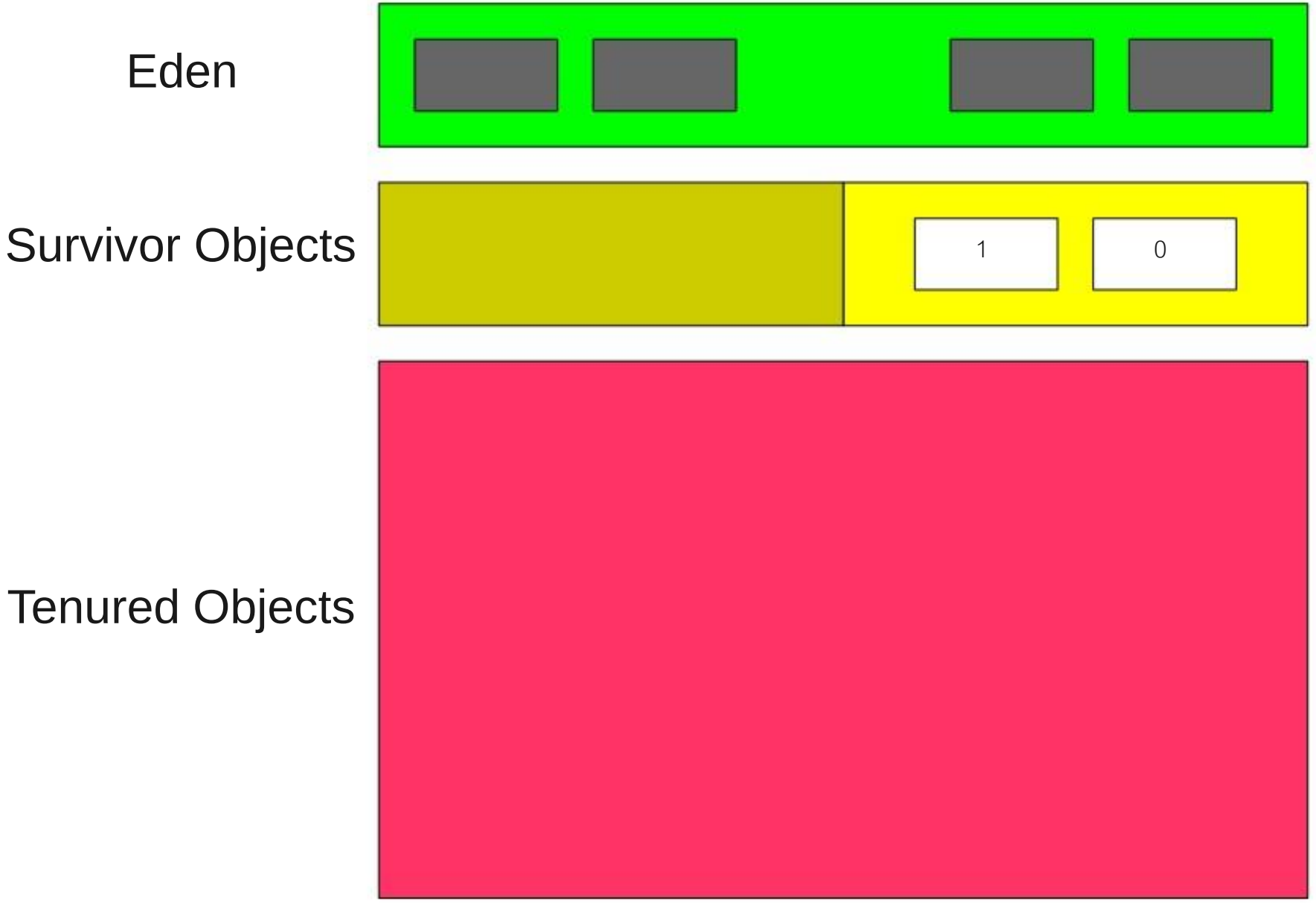
Garbage Collection in Java



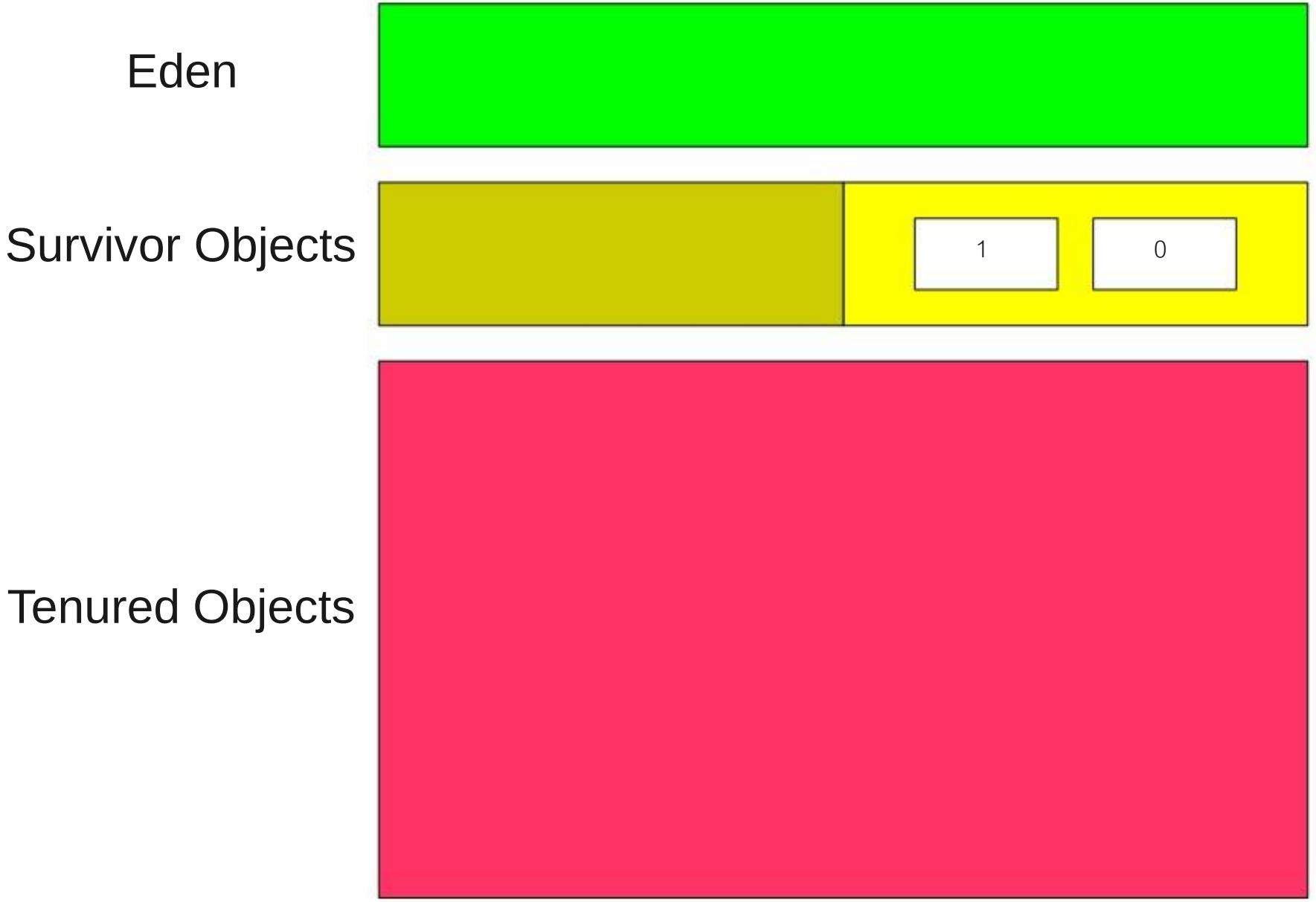
Garbage Collection in Java



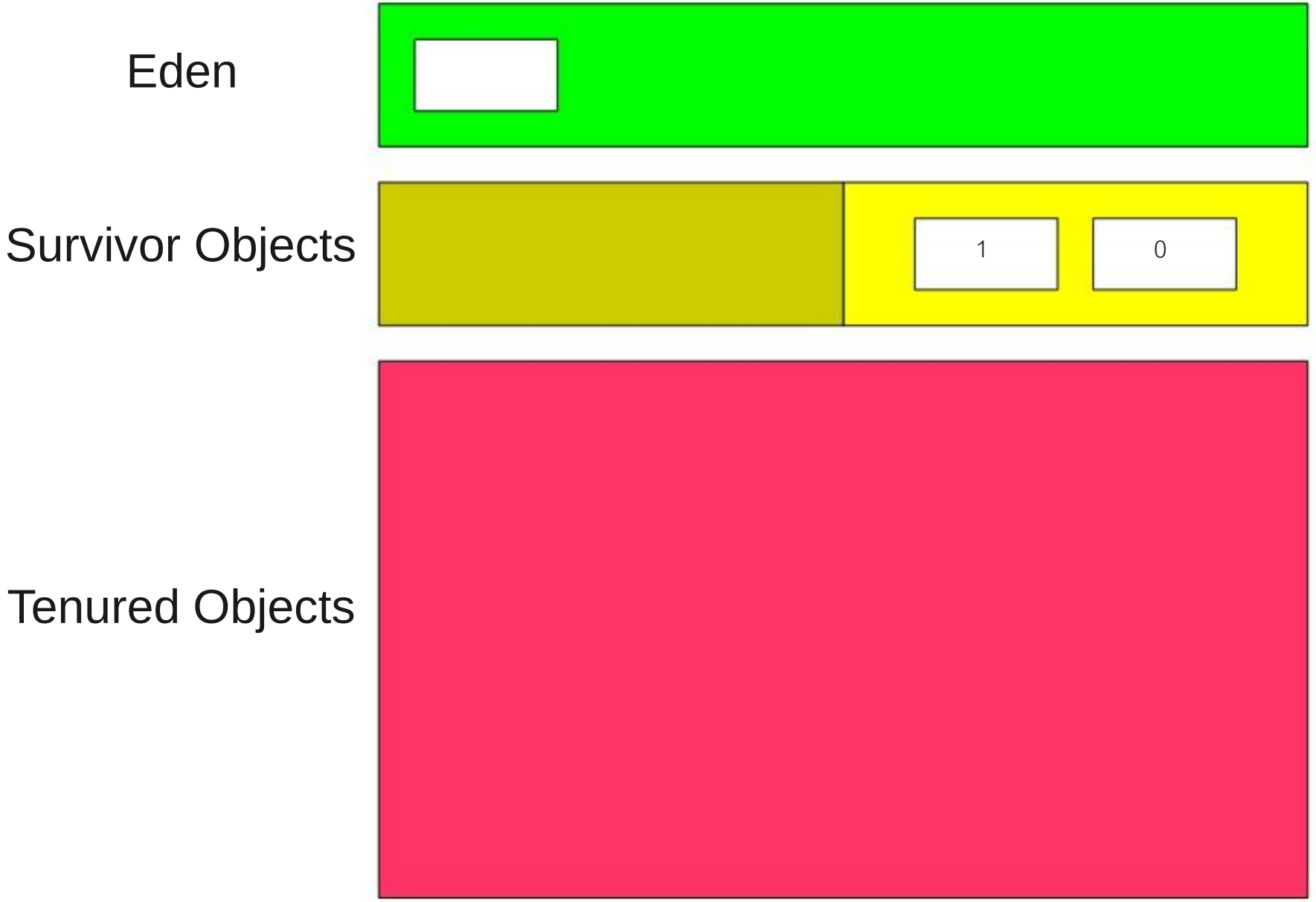
Garbage Collection in Java



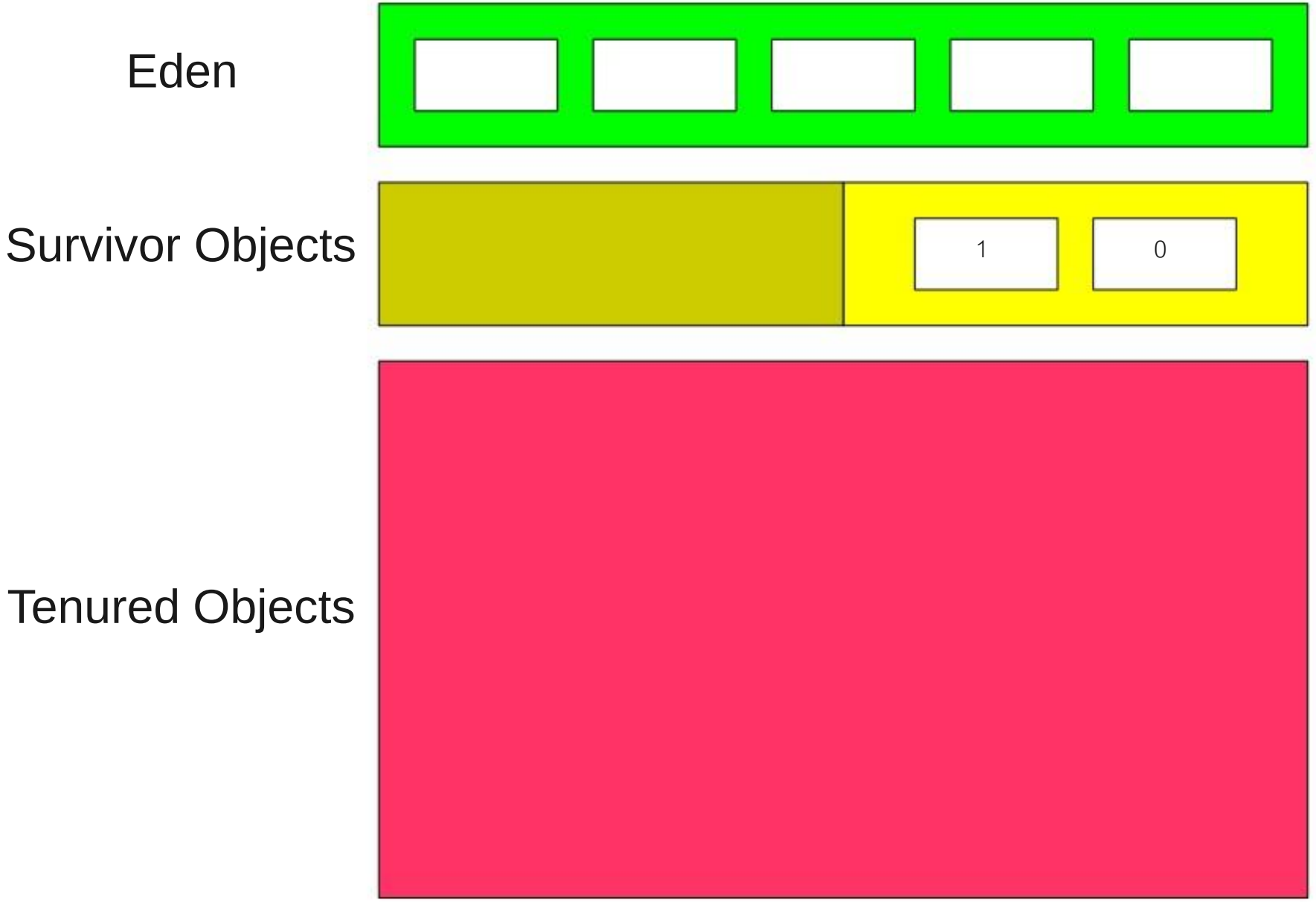
Garbage Collection in Java



Garbage Collection in Java

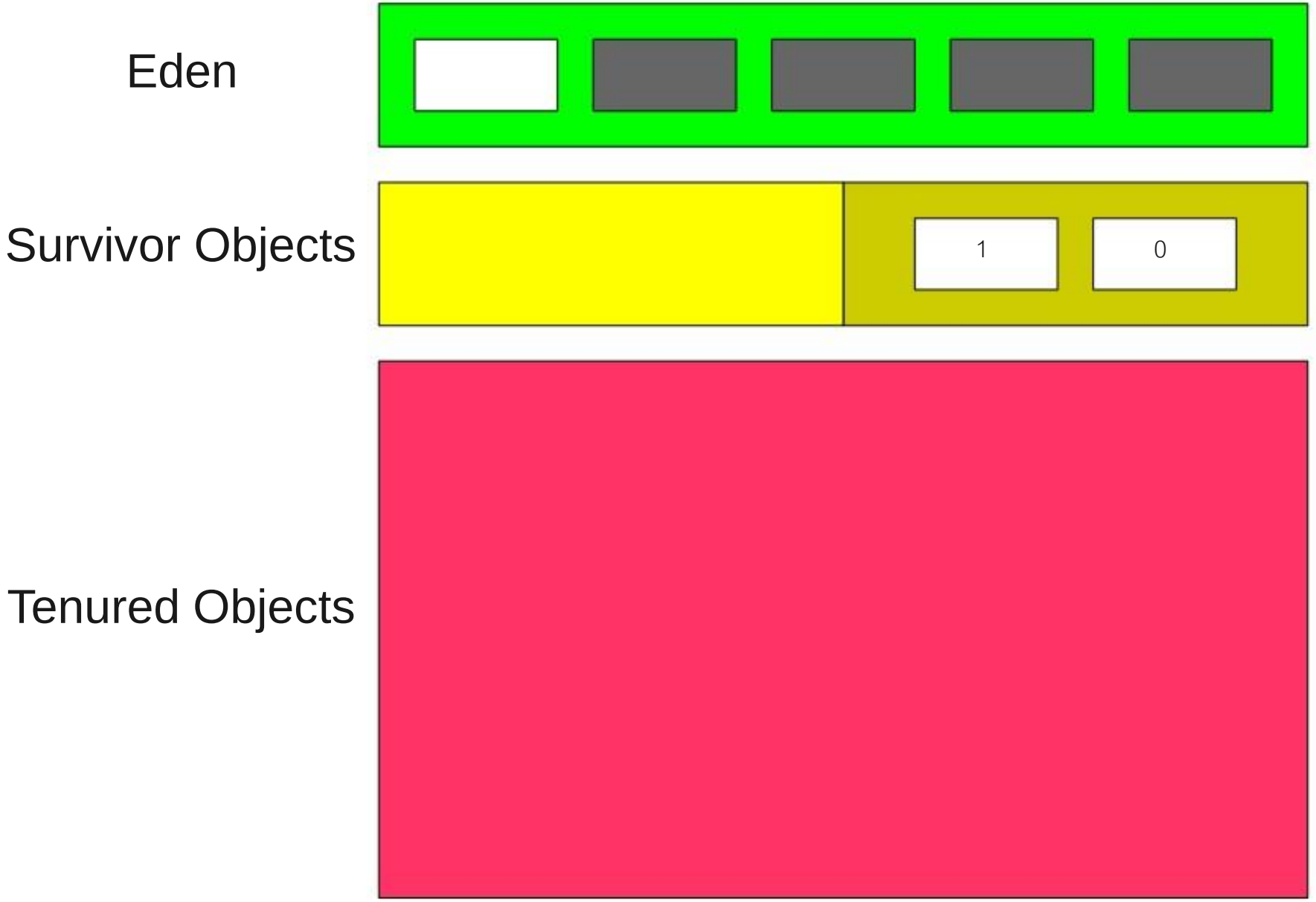


Garbage Collection in Java

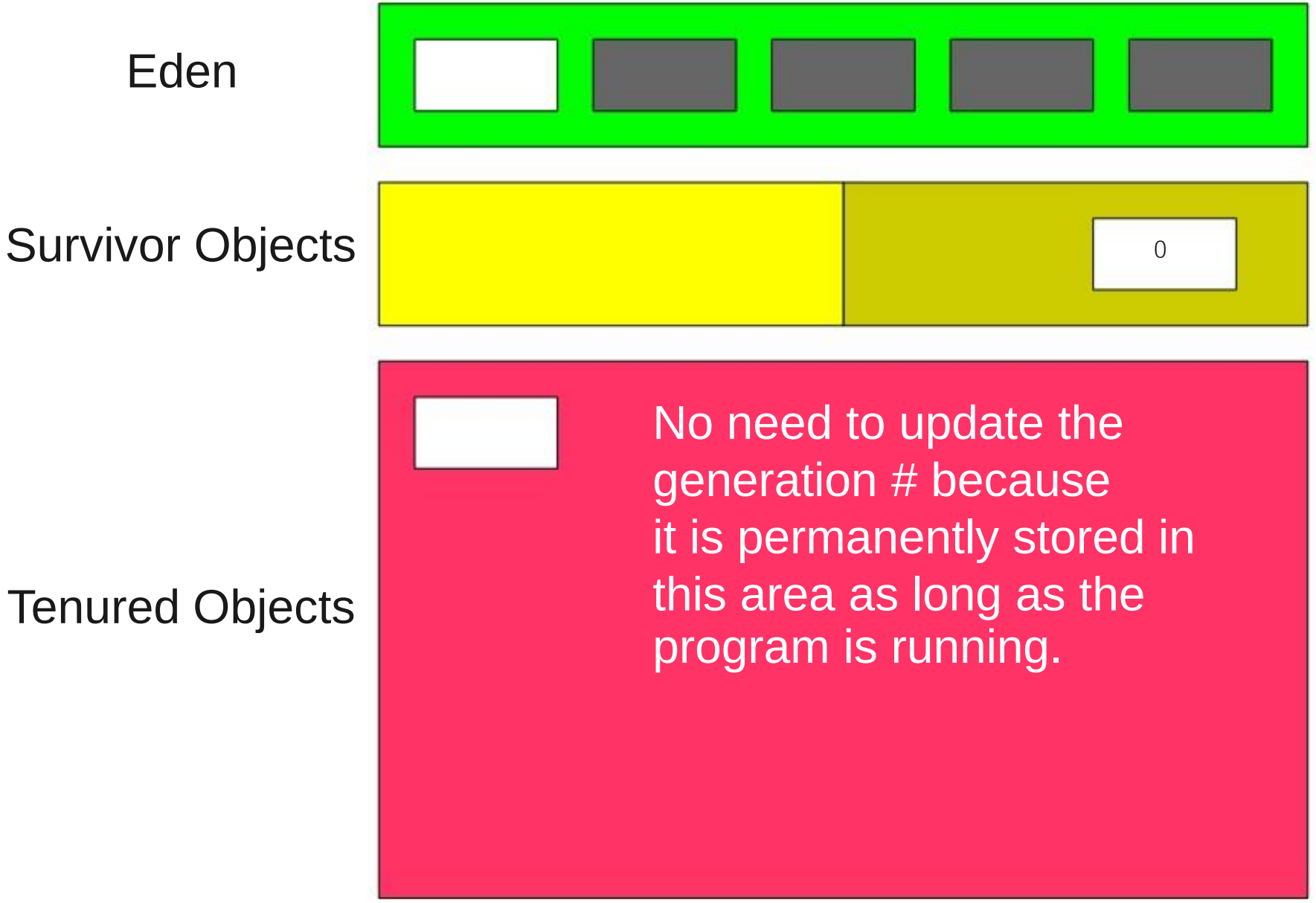


Garbage Collection in Java

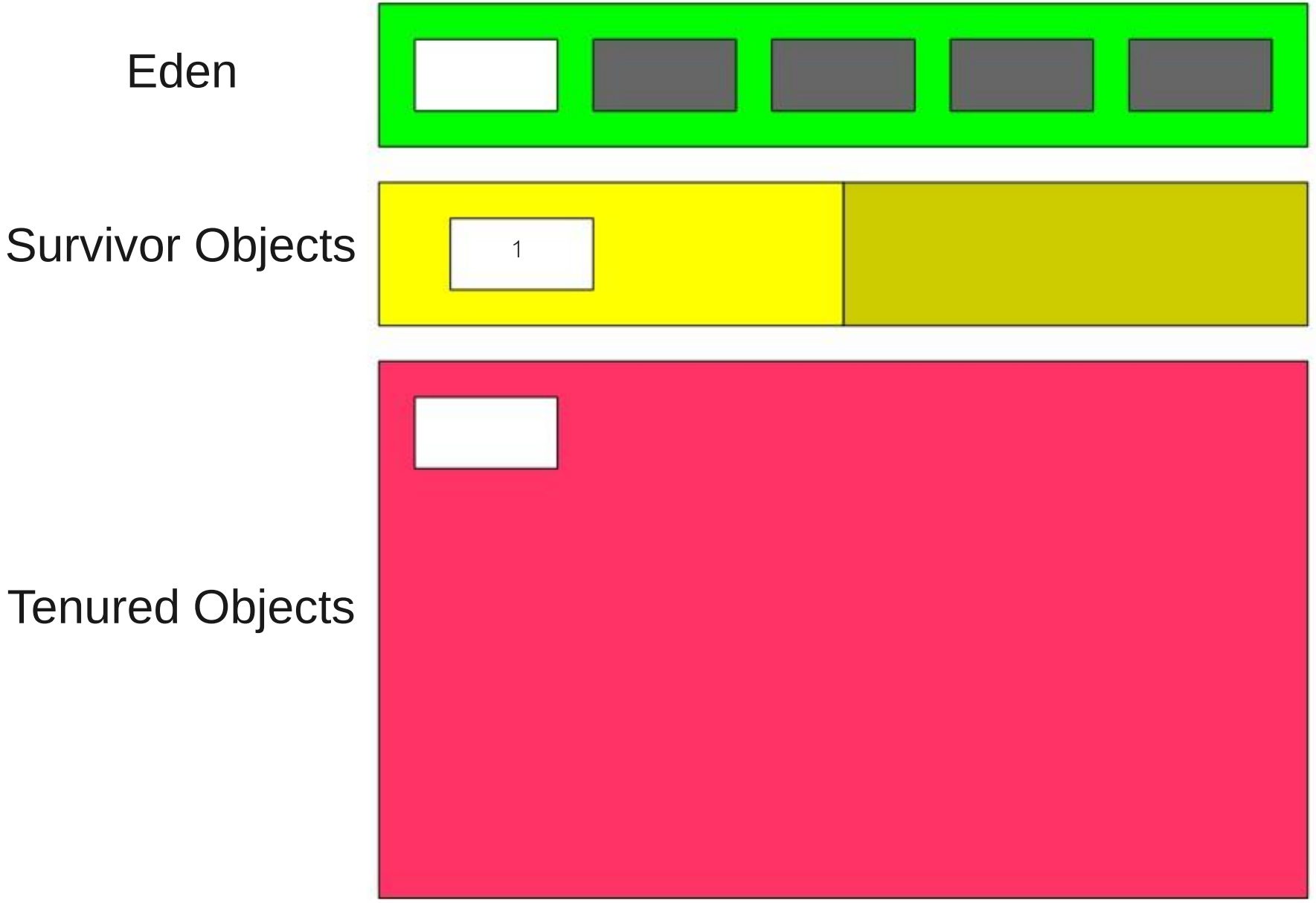
Garbage Collection in Java



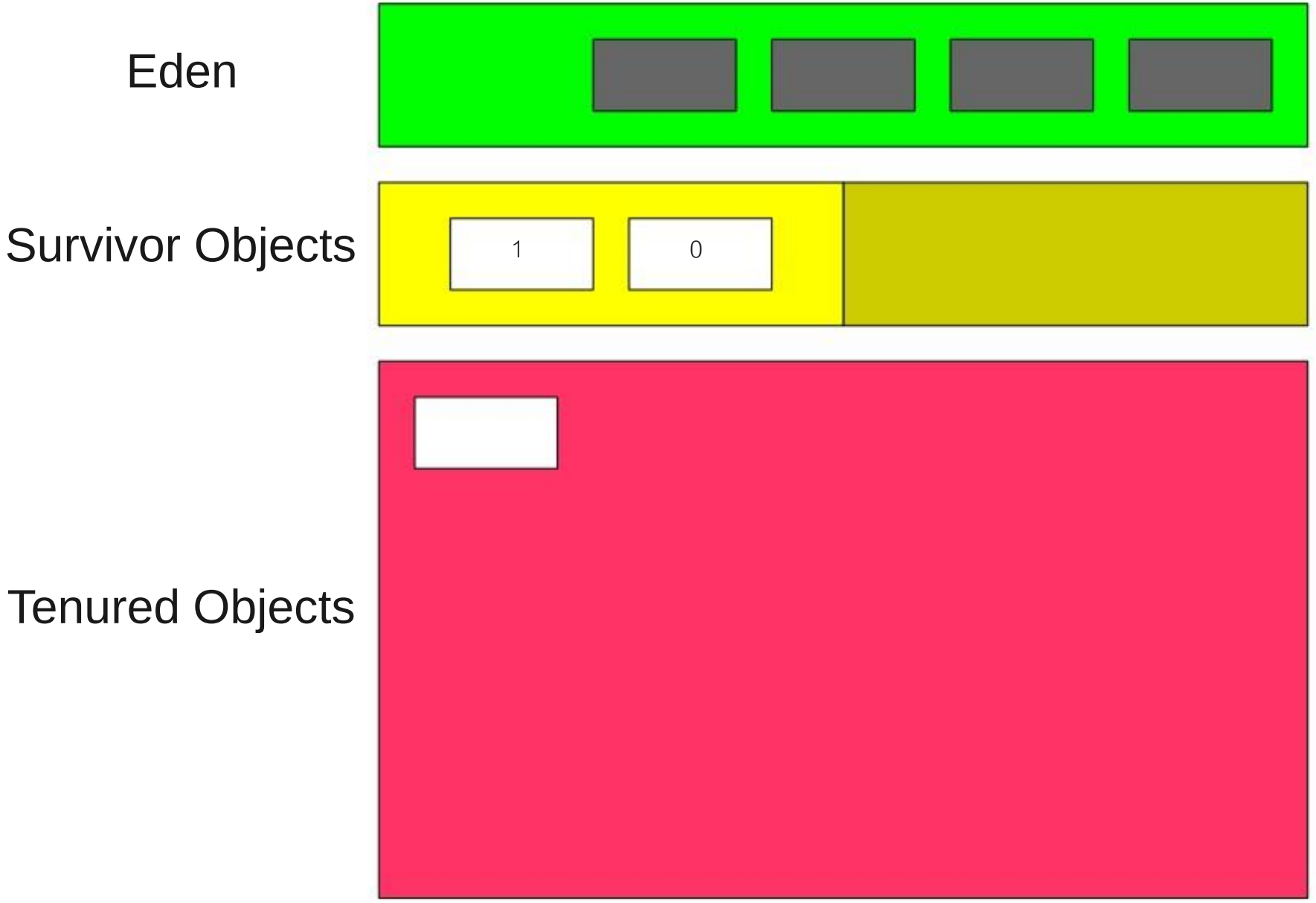
Garbage Collection in Java



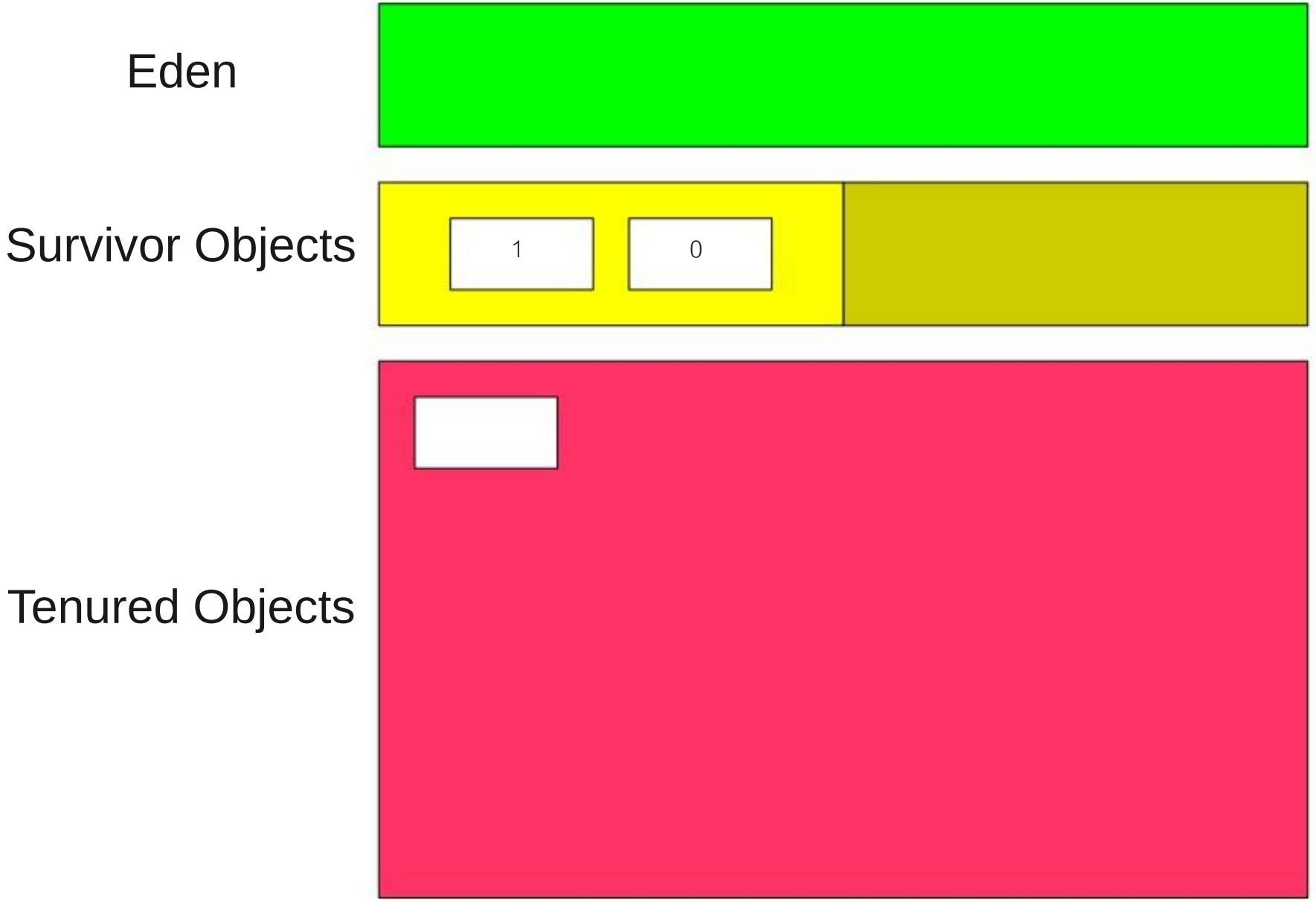
Garbage Collection in Java



Garbage Collection in Java



Garbage Collection in Java

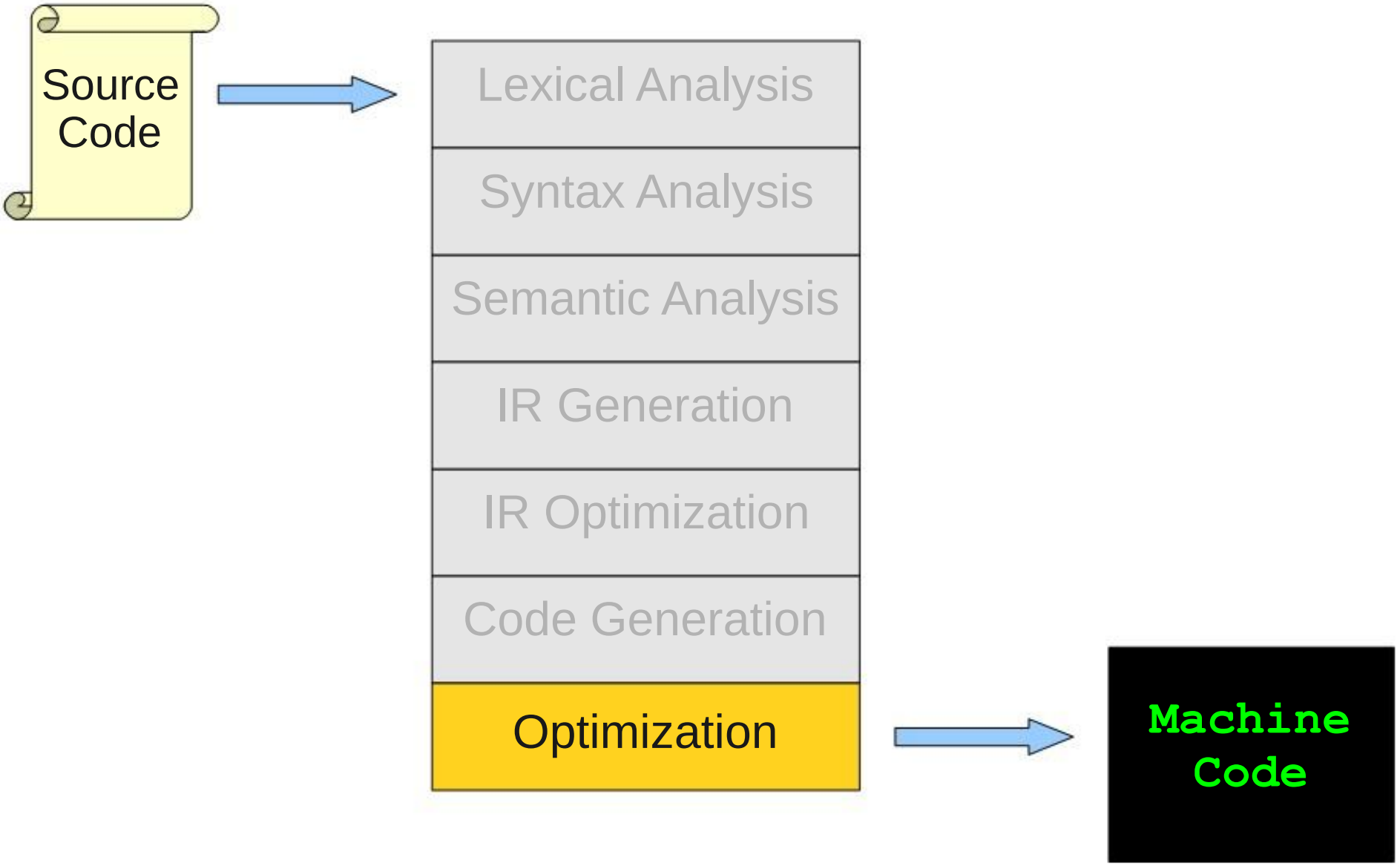


Generational Garbage Collection

- Partition memory into several “generations.”
- Objects are always allocated in the first generation.
- When the first generation fills up, garbage collect it.
  - Runs quickly; collects only a small region of memory.
- Move objects that survive in the first generation long enough into the next generation.
- When no space can be found, run a full (slower) garbage collection on all of memory.

Code Optimization

Where We Are

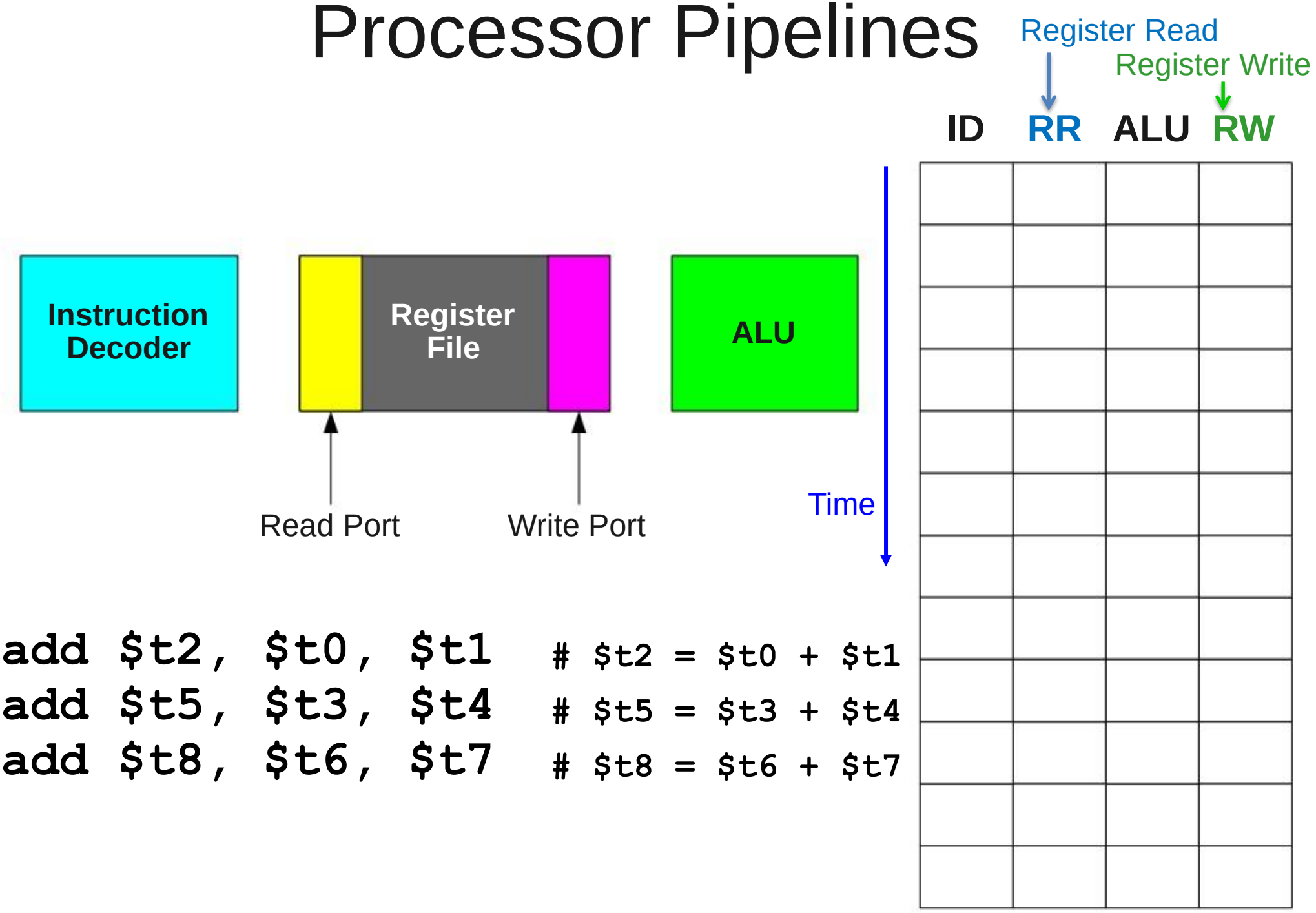

Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Final Code Optimization

- **Goal**: Optimize generated code by exploiting machine-dependent properties not visible at the IR level.
- Critical step in most compilers, but often very messy:
  - Techniques developed for one machine may be completely useless on another.
  - Techniques developed for one language may be completely useless with another.

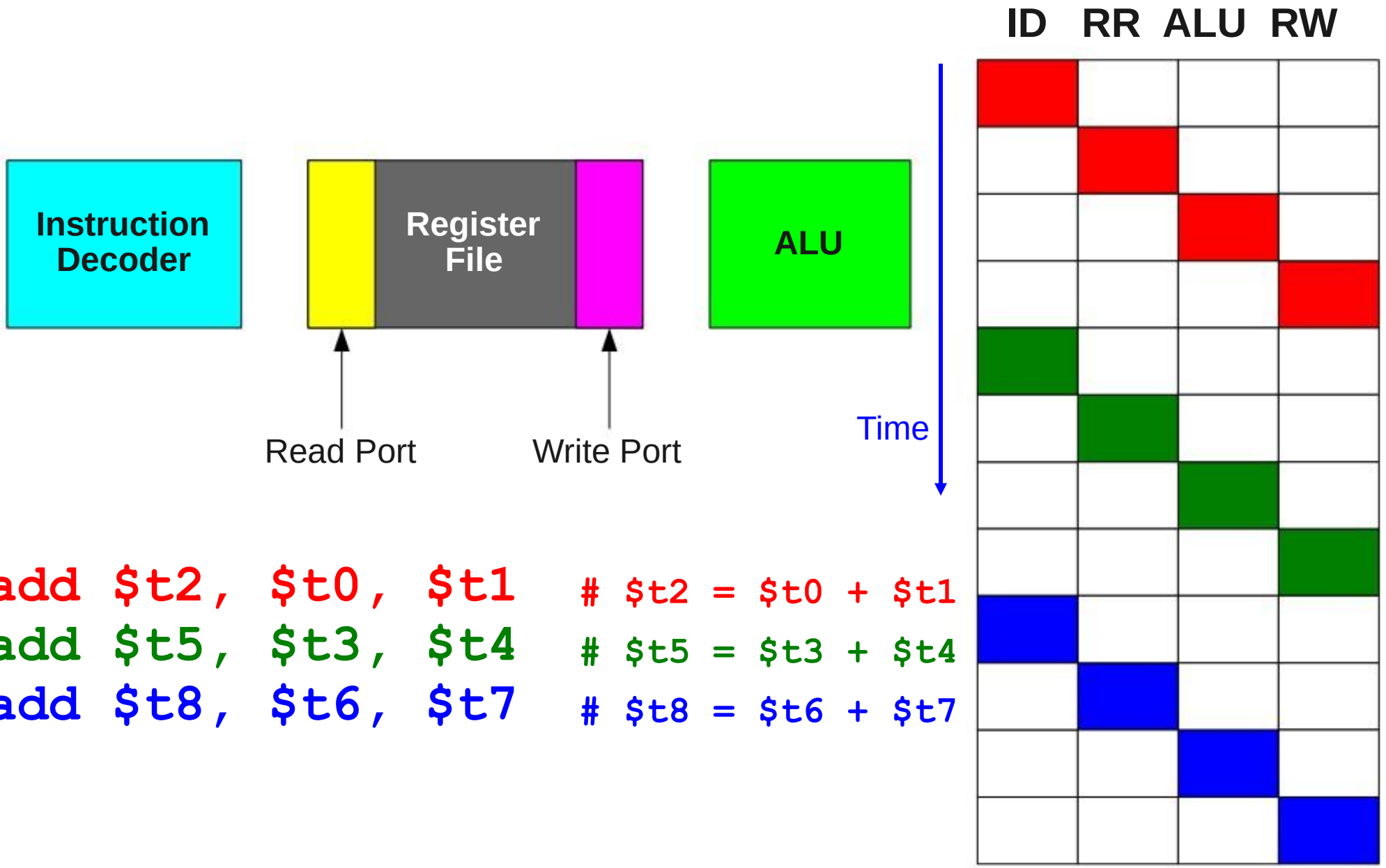
Optimizations for Pipelining

Processor Pipelines




```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t5, $t3, $t4   # $t5 = $t3 + $t4
add $t8, $t6, $t7   # $t8 = $t6 + $t7
```

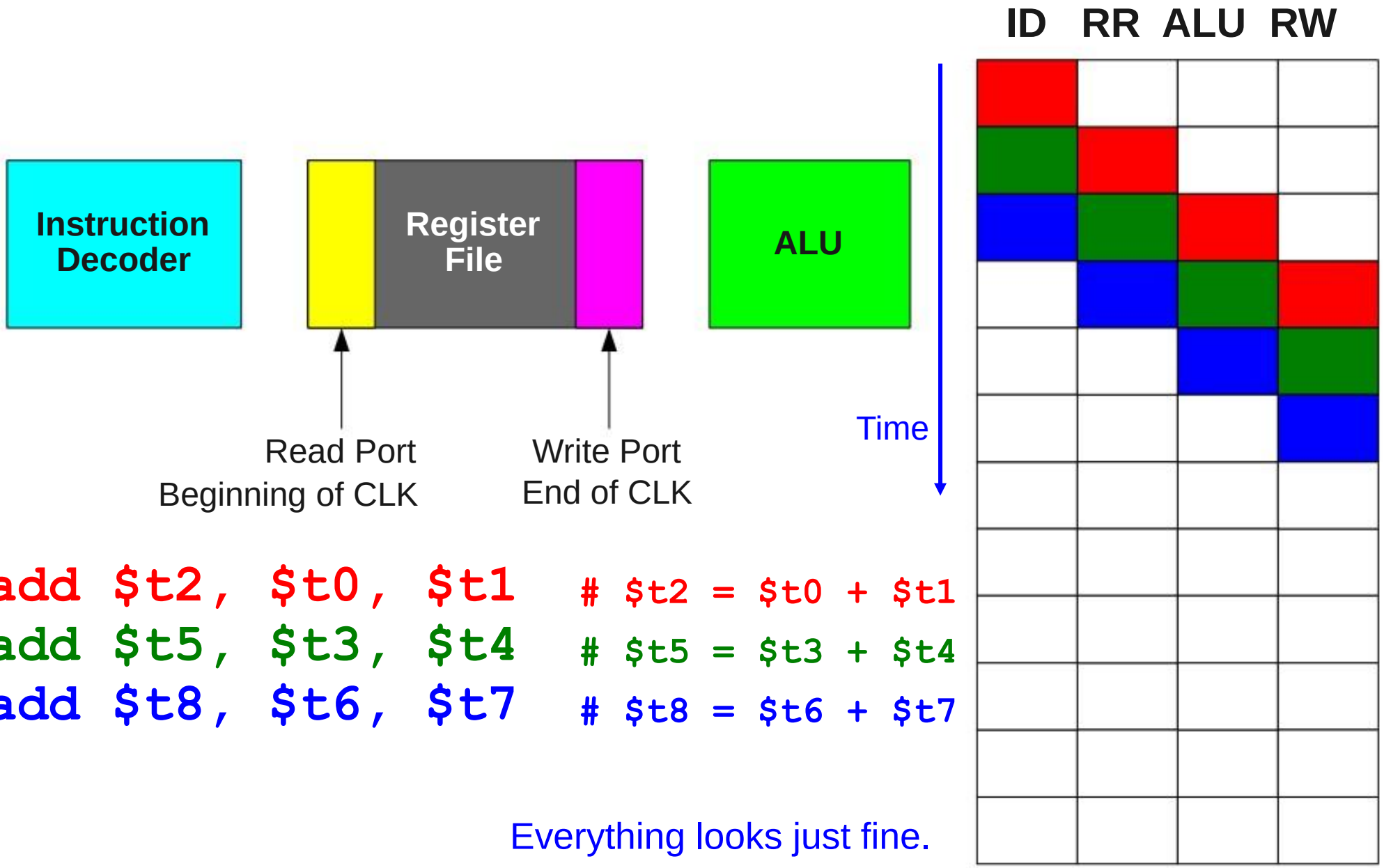
Processor Pipelines




```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t5, $t3, $t4   # $t5 = $t3 + $t4
add $t8, $t6, $t7   # $t8 = $t6 + $t7
```

No pipeline

Processor Pipelines

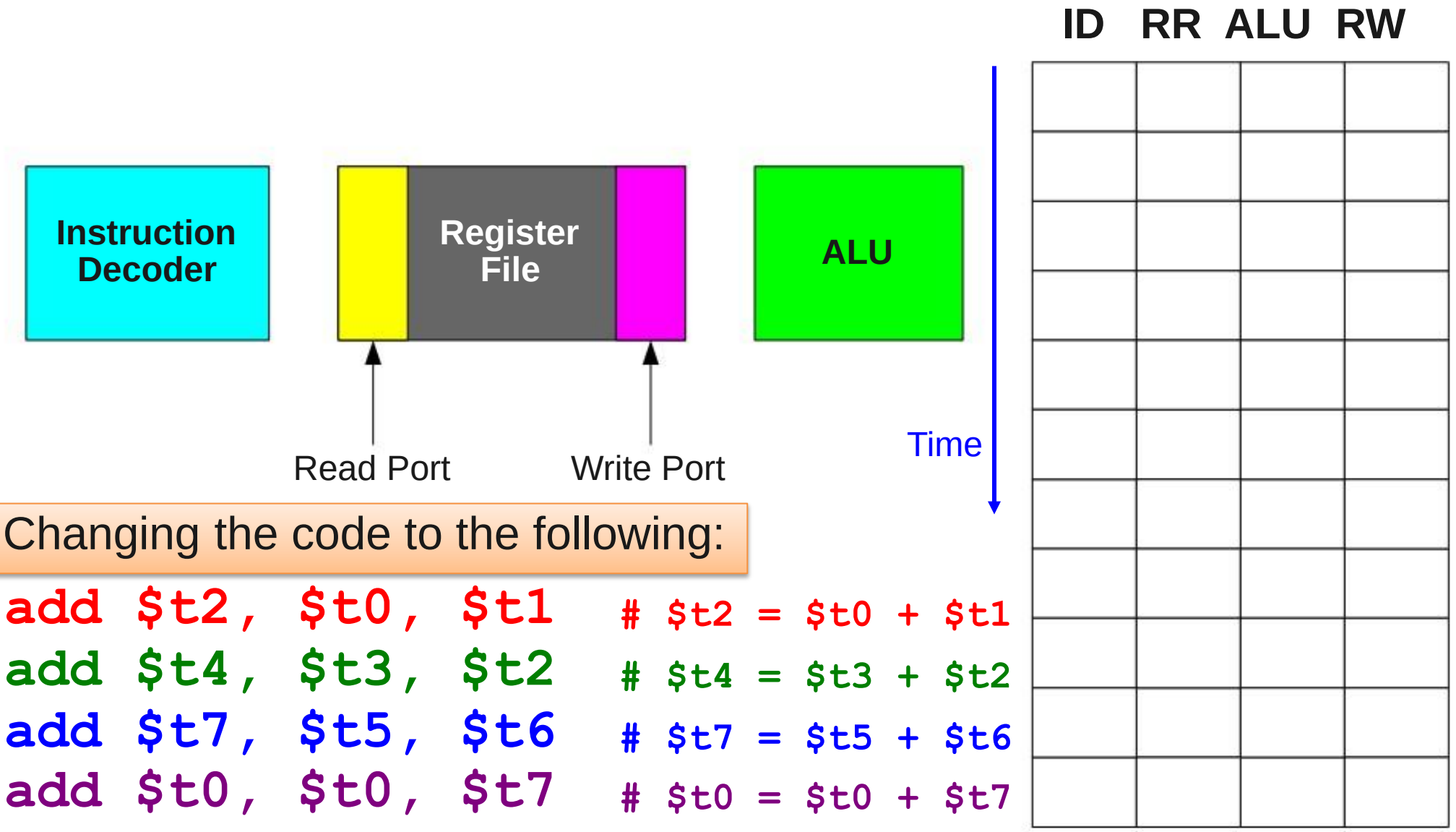



```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t5, $t3, $t4   # $t5 = $t3 + $t4
add $t8, $t6, $t7   # $t8 = $t6 + $t7
```

Everything looks just fine.

Pipeline

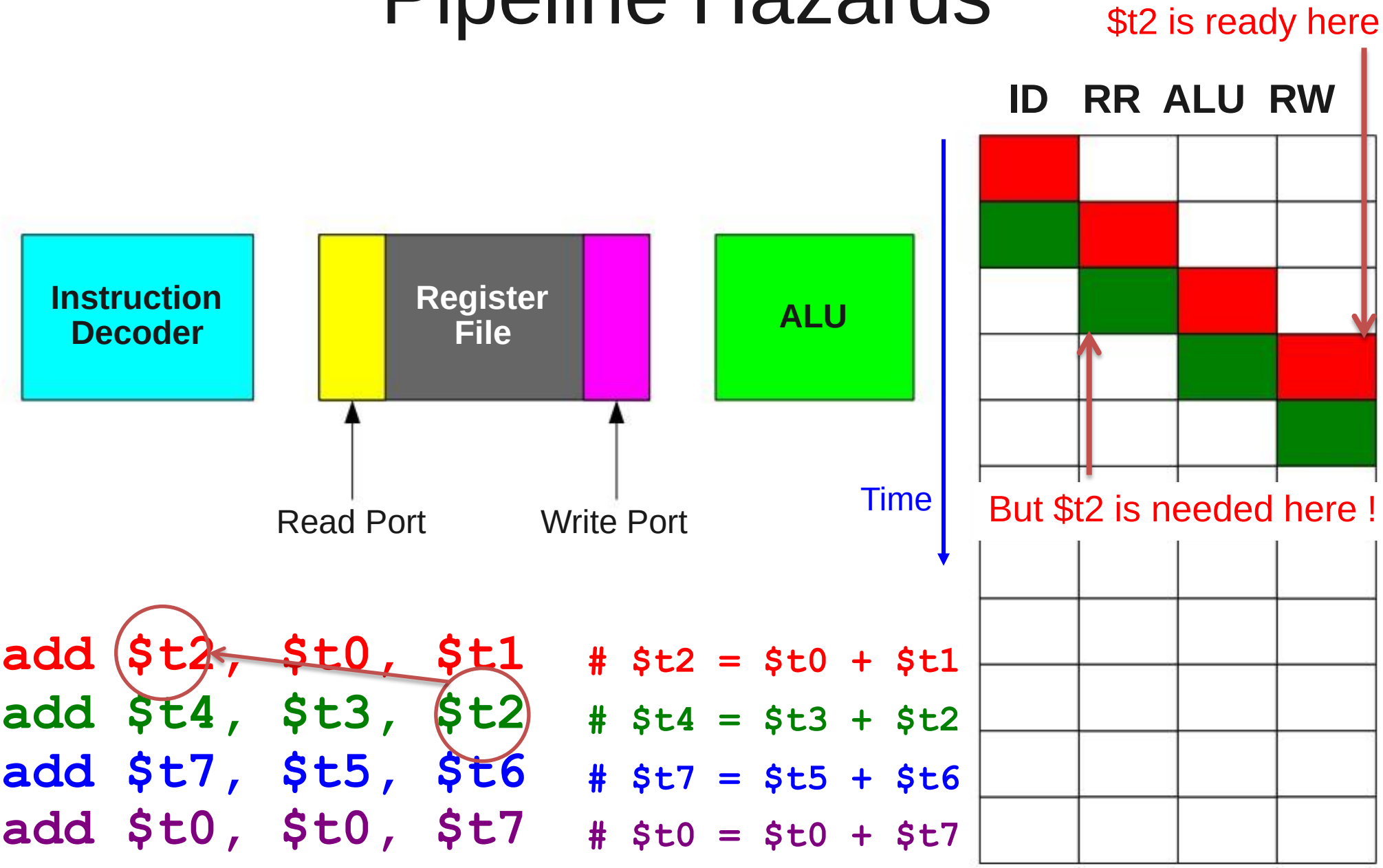
Pipeline Hazards



Changing the code to the following:

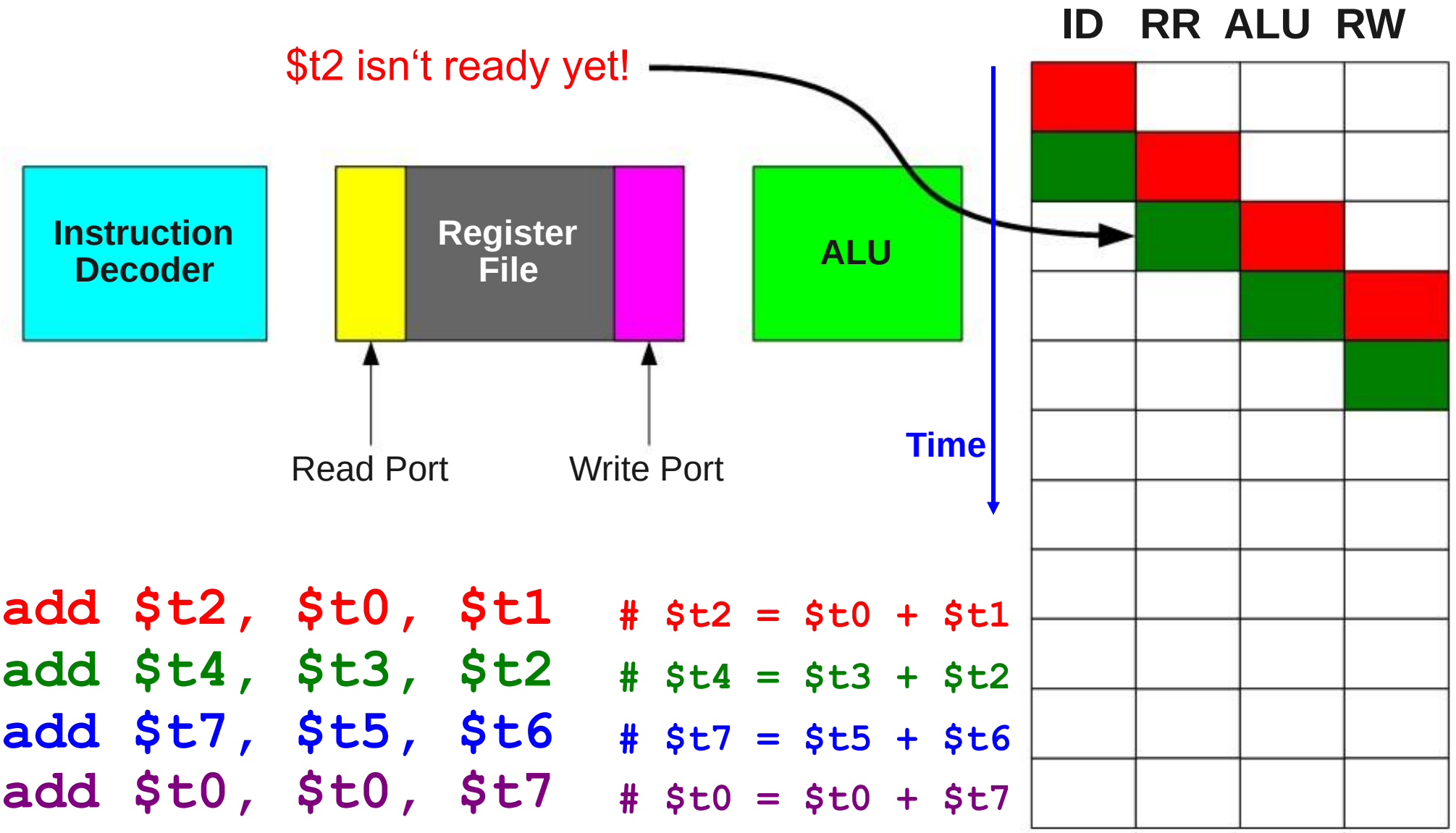
```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t4, $t3, $t2   # $t4 = $t3 + $t2
add $t7, $t5, $t6   # $t7 = $t5 + $t6
add $t0, $t0, $t7   # $t0 = $t0 + $t7
```

Pipeline Hazards




```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t4, $t3, $t2    # $t4 = $t3 + $t2
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

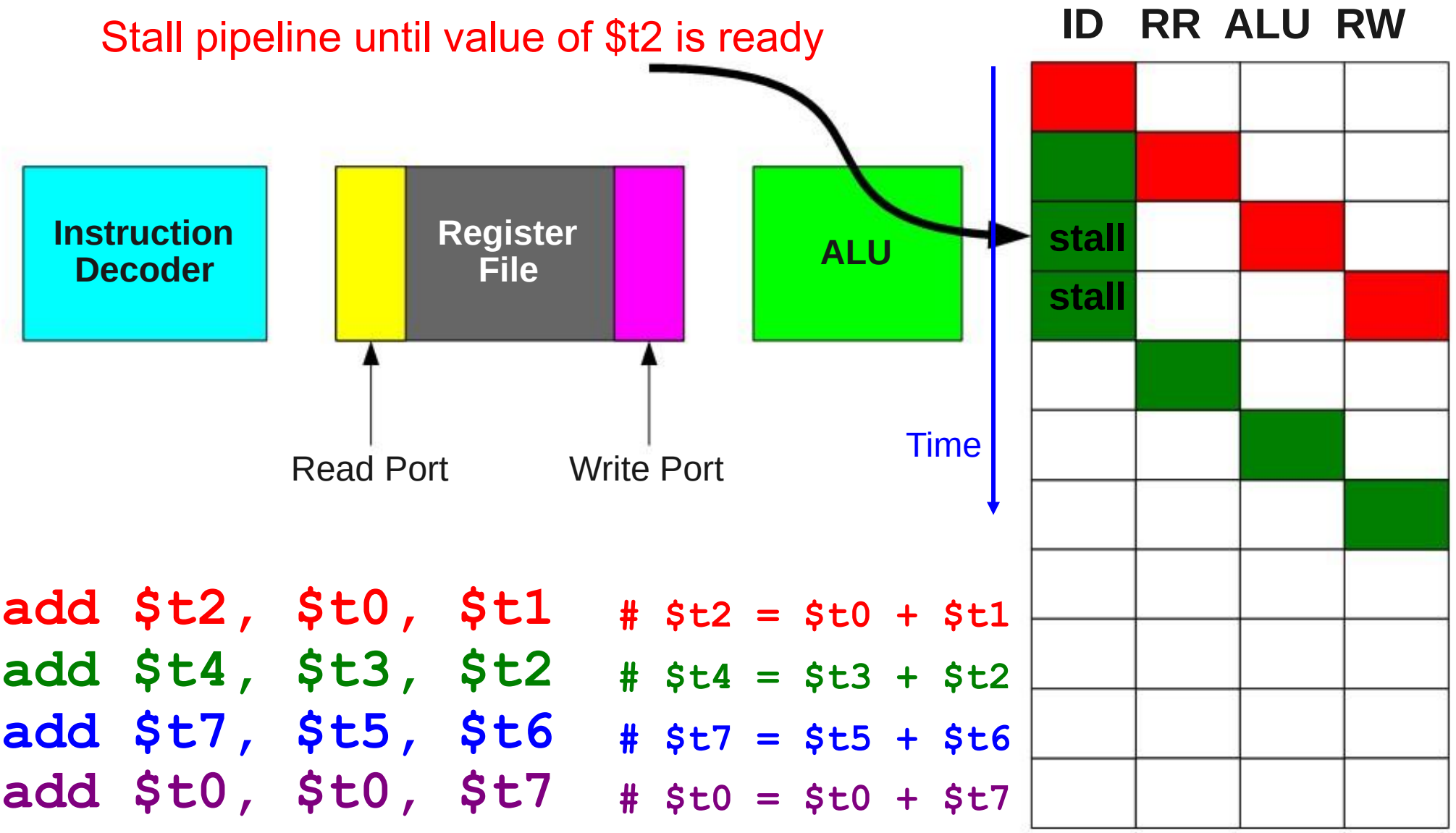
Pipeline Hazards




```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t4, $t3, $t2   # $t4 = $t3 + $t2
add $t7, $t5, $t6   # $t7 = $t5 + $t6
add $t0, $t0, $t7   # $t0 = $t0 + $t7
```

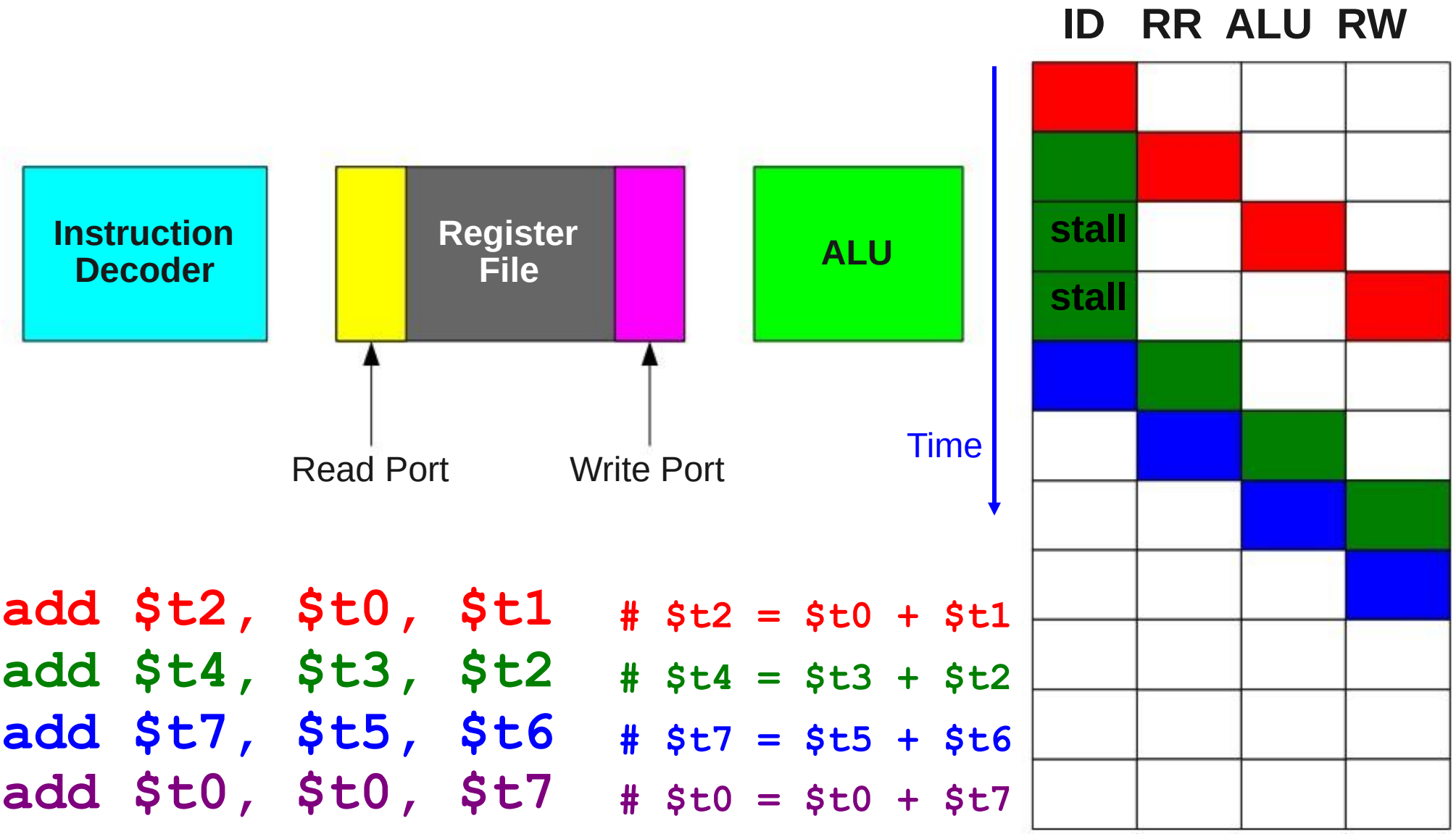
Pipeline Hazards

Stall pipeline until value of $t2 is ready



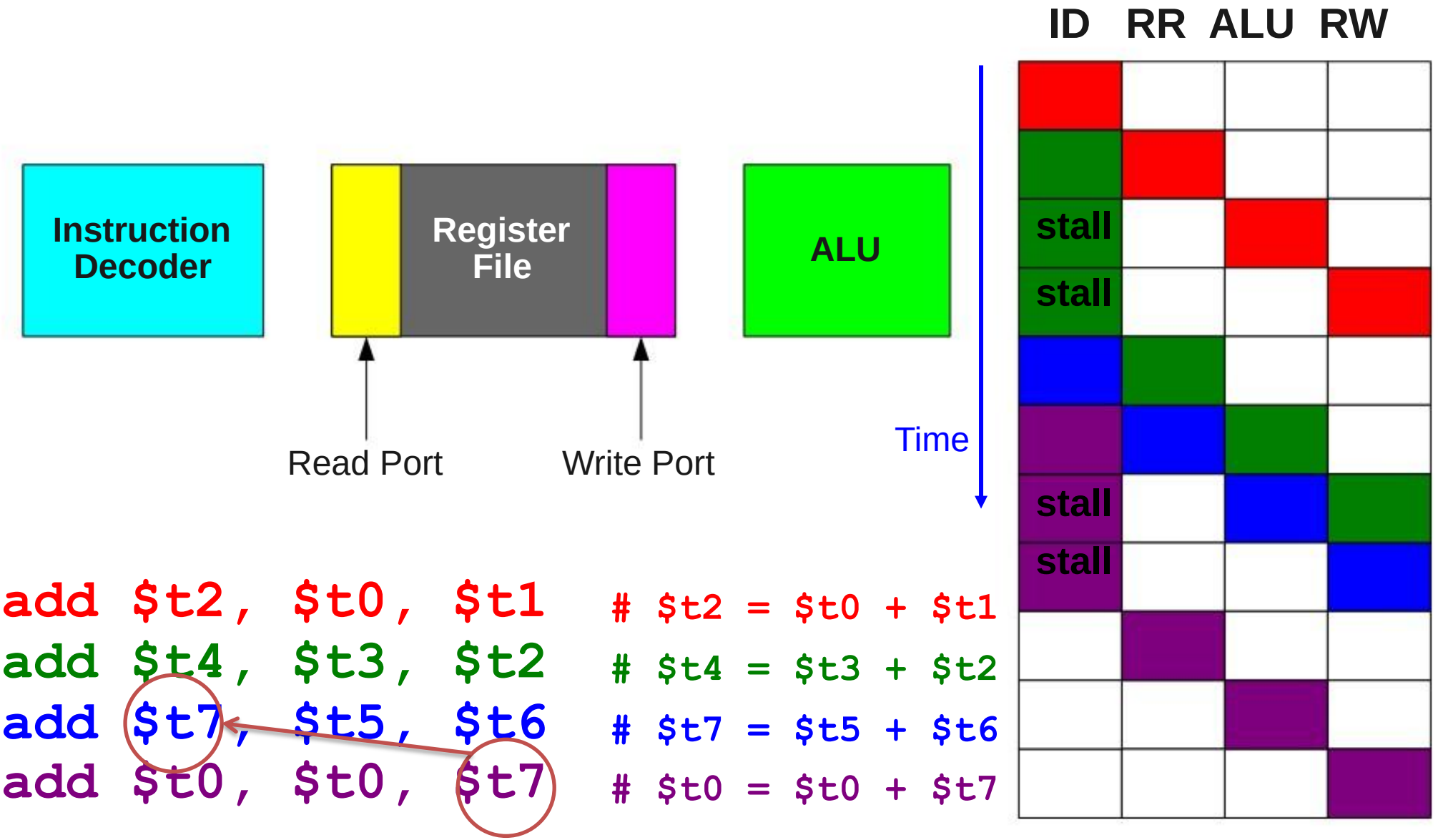

```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t4, $t3, $t2   # $t4 = $t3 + $t2
add $t7, $t5, $t6   # $t7 = $t5 + $t6
add $t0, $t0, $t7   # $t0 = $t0 + $t7
```

Pipeline Hazards



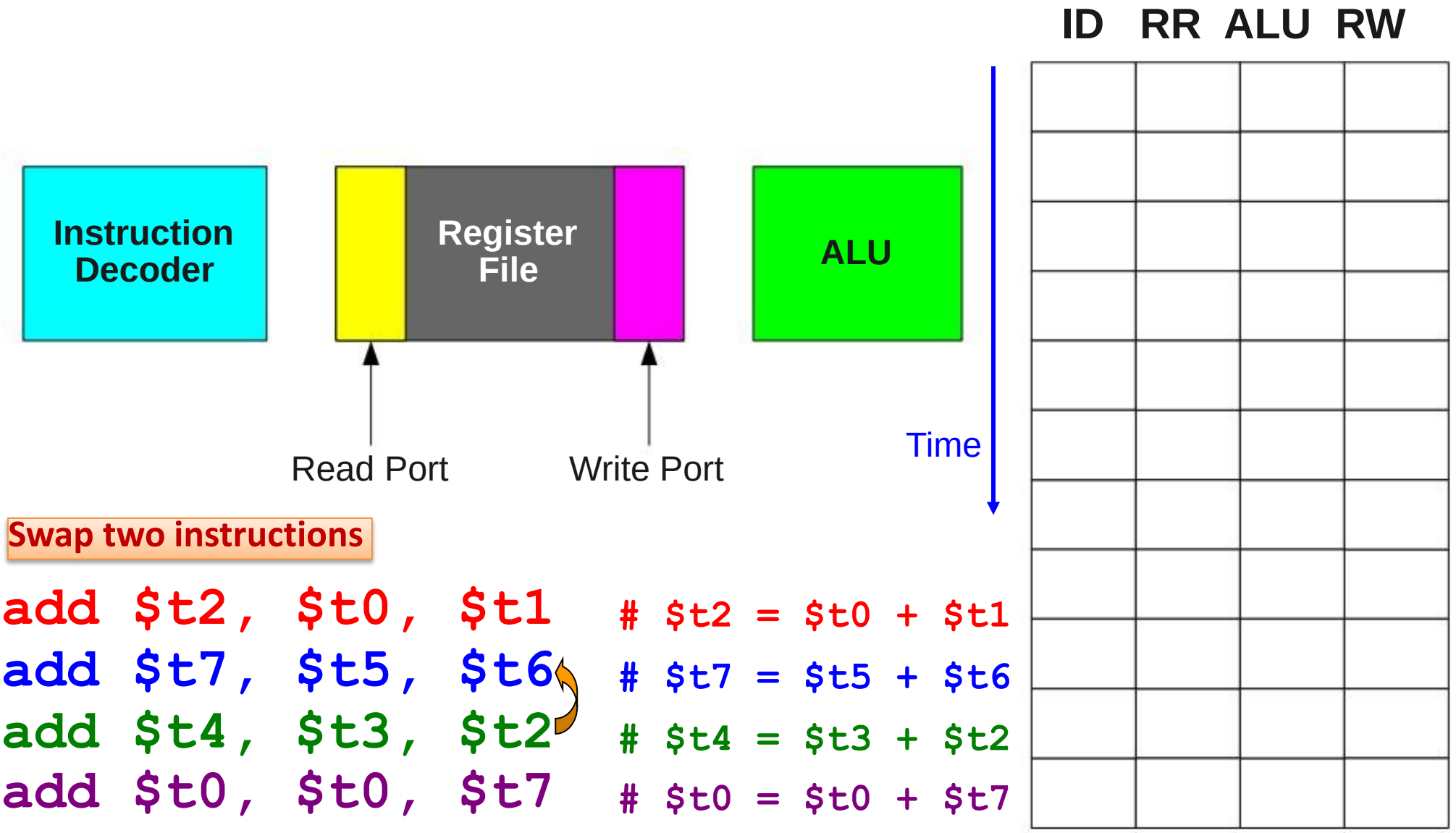

```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t4, $t3, $t2   # $t4 = $t3 + $t2
add $t7, $t5, $t6   # $t7 = $t5 + $t6
add $t0, $t0, $t7   # $t0 = $t0 + $t7
```

Pipeline Hazards




```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t4, $t3, $t2   # $t4 = $t3 + $t2
add $t7, $t5, $t6   # $t7 = $t5 + $t6
add $t0, $t0, $t7   # $t0 = $t0 + $t7
```

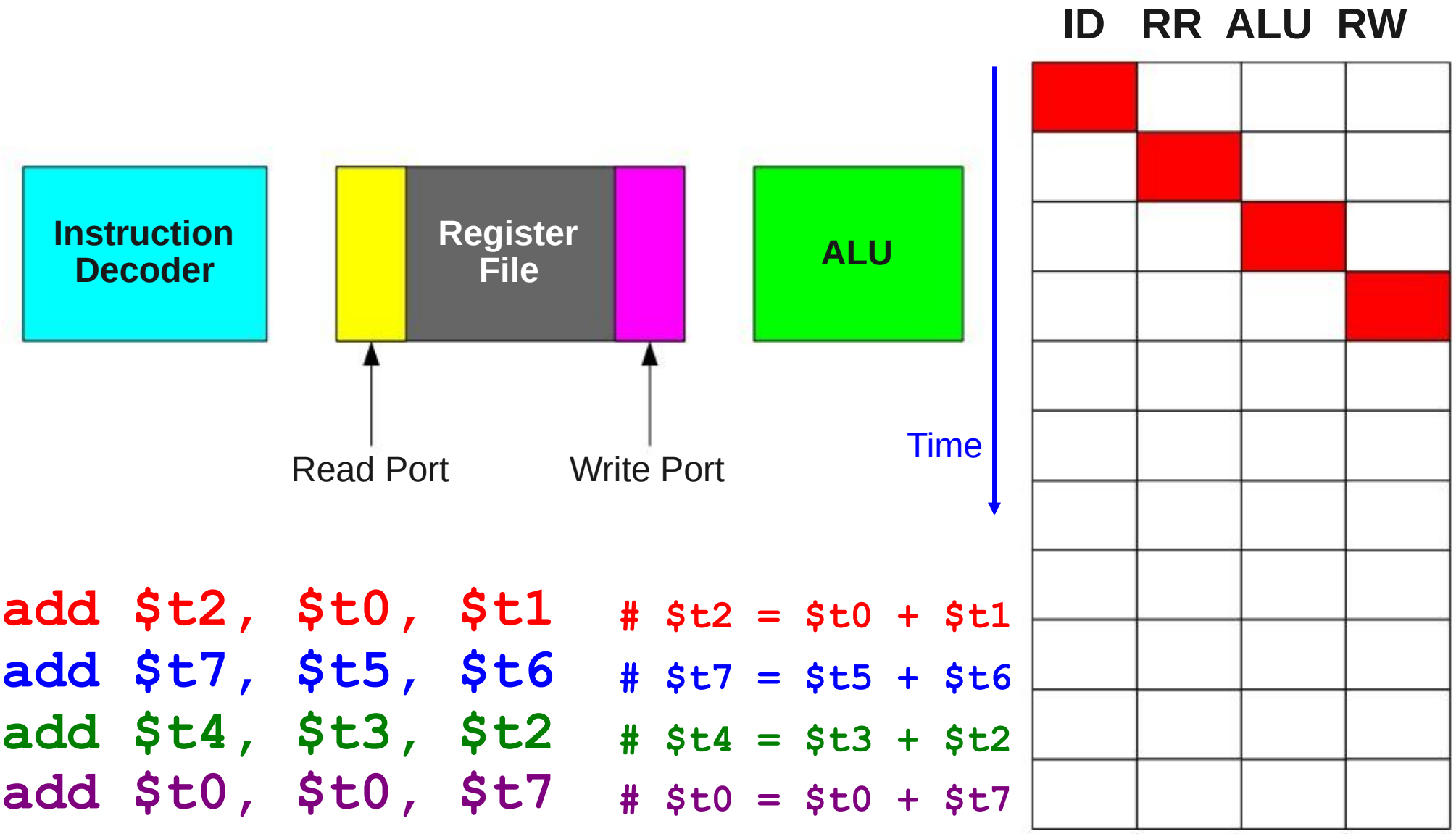
Pipeline Hazards



Swap two instructions

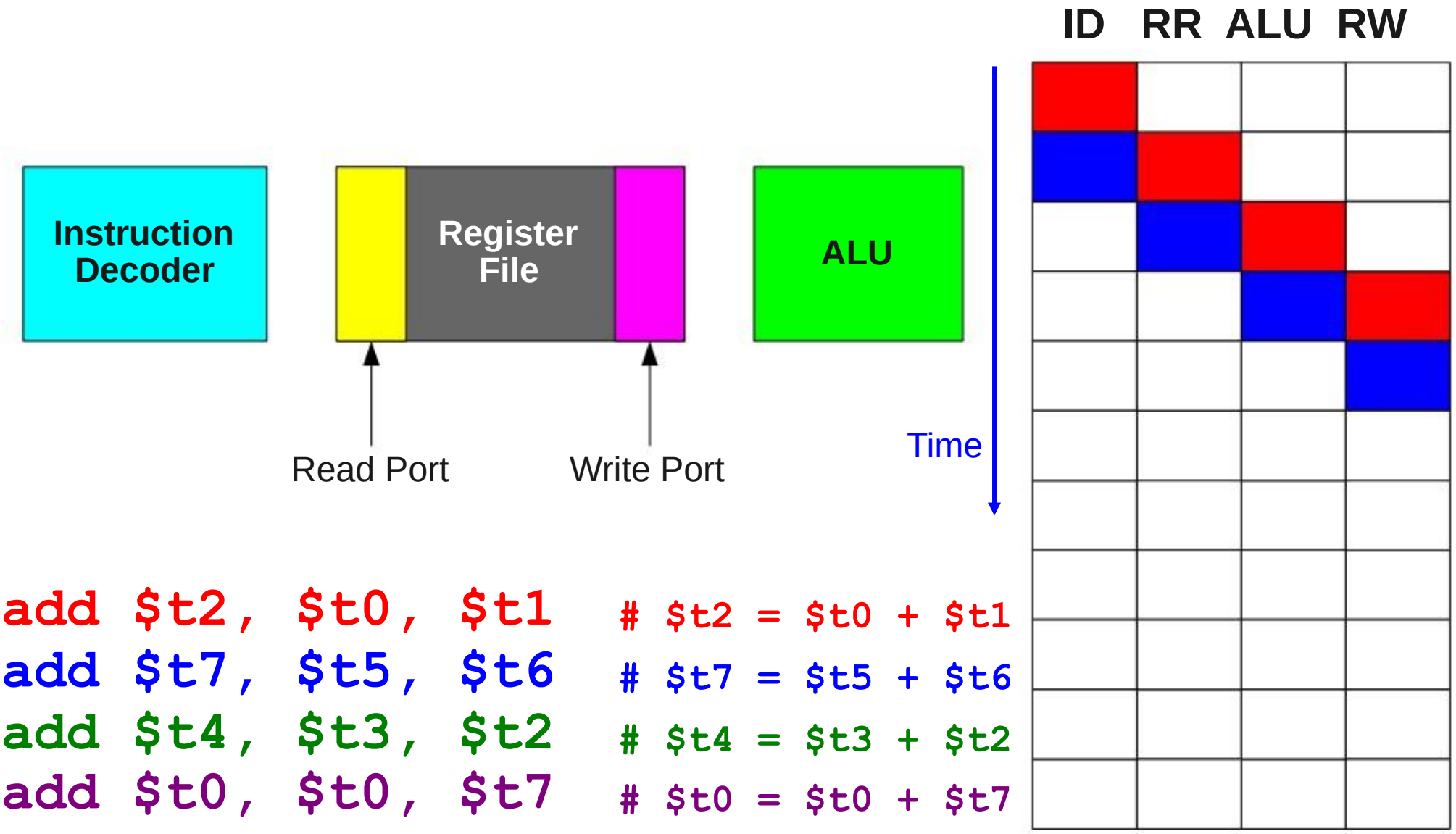
```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t4, $t3, $t2    # $t4 = $t3 + $t2
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

Pipeline Hazards



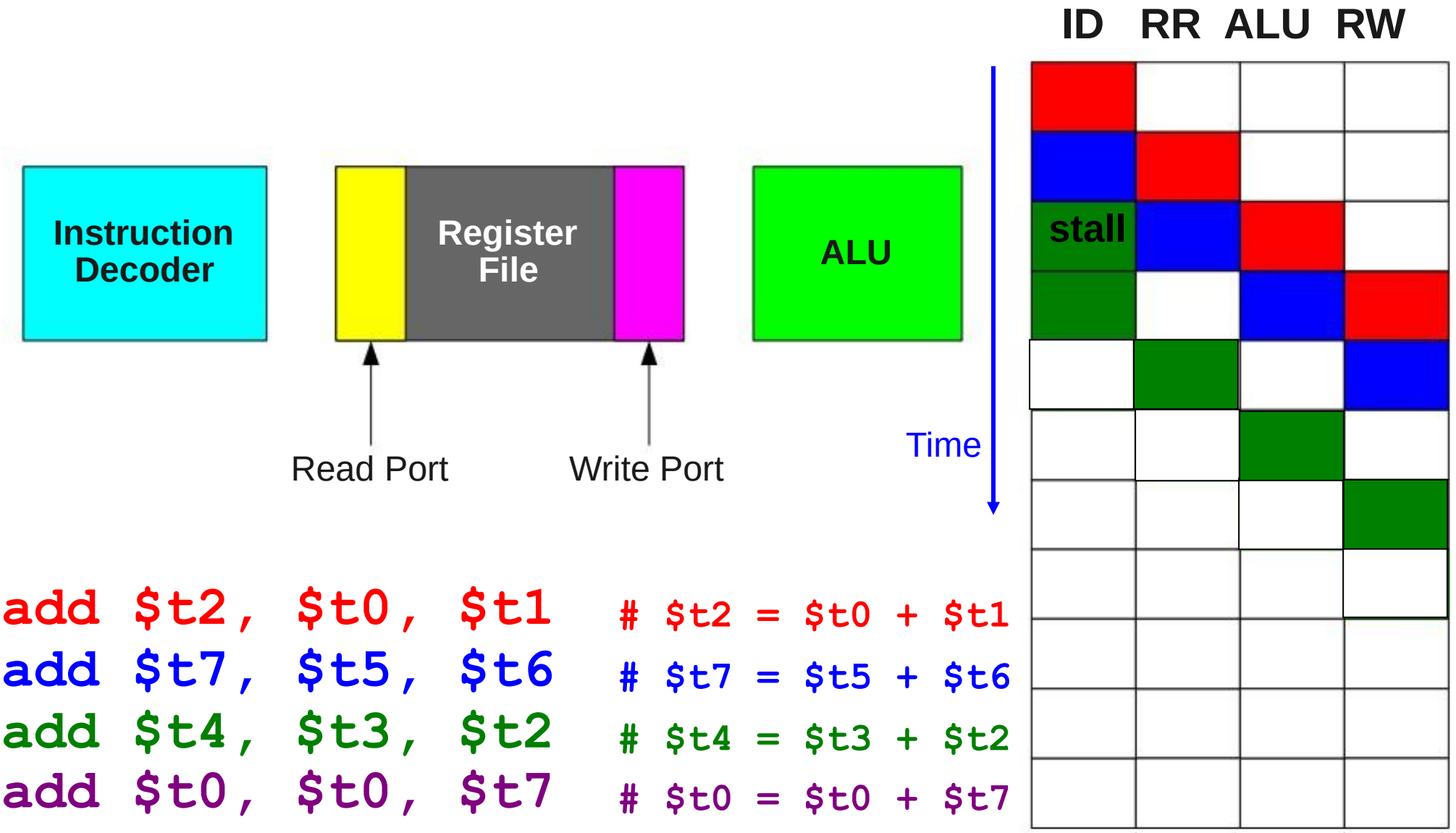

```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t7, $t5, $t6   # $t7 = $t5 + $t6
add $t4, $t3, $t2   # $t4 = $t3 + $t2
add $t0, $t0, $t7   # $t0 = $t0 + $t7
```

Pipeline Hazards



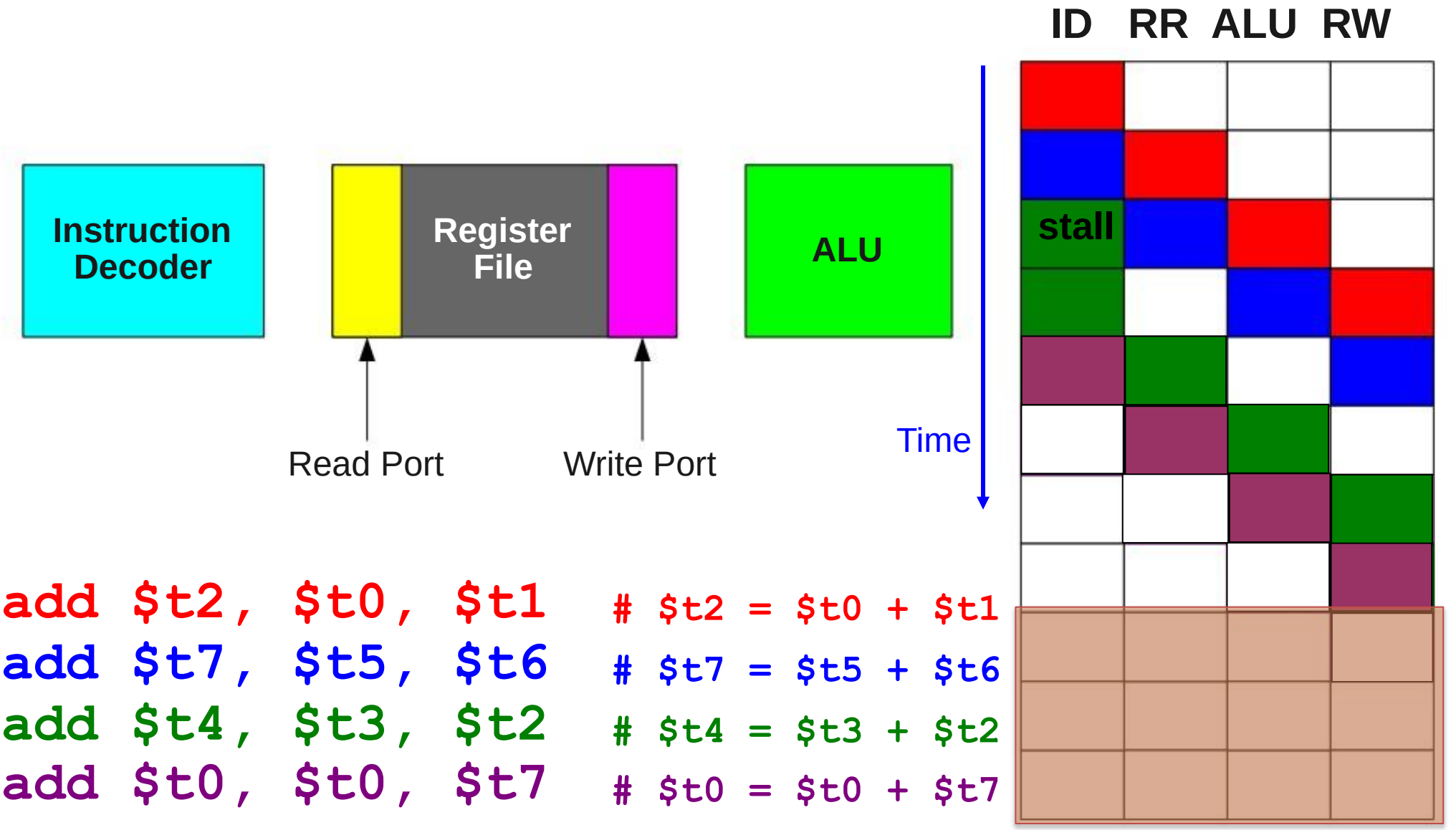

```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t7, $t5, $t6   # $t7 = $t5 + $t6
add $t4, $t3, $t2   # $t4 = $t3 + $t2
add $t0, $t0, $t7   # $t0 = $t0 + $t7
```

Pipeline Hazards




```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t7, $t5, $t6   # $t7 = $t5 + $t6
add $t4, $t3, $t2   # $t4 = $t3 + $t2
add $t0, $t0, $t7   # $t0 = $t0 + $t7
```

Pipeline Hazards




```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t7, $t5, $t6   # $t7 = $t5 + $t6
add $t4, $t3, $t2   # $t4 = $t3 + $t2
add $t0, $t0, $t7   # $t0 = $t0 + $t7
```

Three clock cycles faster

Next class: Other optimization techniques