

# Garbage Collection

**Lecture Notes by Keith Schwarz**

[http://www.keithschwarz.com/cs143/WWW/sum2011/lectures/180\\_Garbage\\_Collection.pdf](http://www.keithschwarz.com/cs143/WWW/sum2011/lectures/180_Garbage_Collection.pdf)

# Runtime Memory Management

- Most constructs in a programming language need memory.
- Some need a fixed amount of memory
  - *(such as?)*
- Some require a variable amount of memory:
  - Local variables
  - Objects
  - Arrays
  - Strings

# Memory Management So Far

- Some memory is preallocated and persists throughout the program:
  - Global variables, virtual function tables, executable code, etc.
- Some memory is allocated on the runtime stack:
  - Local variables, parameters, temporaries.
- Some memory is allocated in the heap:
  - Arrays, objects.
- Memory management for the first two is trivial.
- How do we manage heap-allocated memory?

# Manual Memory Management

- **Option One:** Have the programmer handle allocation and deallocation of dynamic memory.
- Approach used in C, C++.
- Advantages:
  - Programmer can exercise precise control over memory usage.
- Disadvantages:
  - Programmer **has to** exercise precise control over memory usage.

# Strengths of Manual Management

- Comparatively easy to implement.
  - “Just” need a working memory manager.
- Allows programmers to make aggressive performance optimizations.
  - Programmer can choose allocation scheme that achieves best performance.

# Problems with Manual Management

- Easily leads to troublesome bugs:
  - **Memory leaks** where resources are never freed.
  - **Double frees** where a resource is freed twice (major security risk).
  - **Use-after-frees** where a deallocated resource is still used (major security risk).
- Programming languages with manual memory management are almost always not type-safe.

# Automatic Memory Management

- **Idea:** Have the runtime environment automatically reclaim memory.
- Objects that won't be used again are called **garbage**.
- Reclaiming garbage objects automatically is called **garbage collection**.
- Advantages:
  - Programmer doesn't have to reclaim unused resources.
- Disadvantages:
  - Programmer **can't** reclaim unused resources.

# Preliminaries



# What is Garbage?

- An object is called **garbage** at some point during execution if it will never be used again.
- What is garbage at the indicated points?

# What is Garbage?

- An object is called **garbage** at some point during execution if it will never be used again.
- What is garbage at the indicated points?

```
int main() {  
    Object x, y;  
    x = new Object();  
    y = new Object();  
    /* Point A */  
  
    x.doSomething();  
    y.doSomething();  
    /* Point B */  
  
    y = new Object();  
    /* Point C */  
}
```

# Approximating Garbage

- In general, it is **undecidable** whether an object is garbage.
  - Need to rely on a conservative approximation.
- An object is **reachable** if it can still be referenced by the program.
  - Goal for today: detect and reclaim unreachable objects.
- This does not prevent memory leaks!
  - Many reachable objects are never used again.
  - It is **very easy** to have memory leaks in garbage-collected languages.
- Interesting read: “Low-Overhead Memory Leak Detection Using Adaptive Statistical Profiling” by Chilimbi and Hauswirth.

# Assumptions for Today

- Assume that, at runtime, we can find all existing references in the program.
  - Cannot fabricate a reference to an existing object *ex nihilo*.
  - Cannot cast pointers to integers or vice-versa.
- Examples: Java, Python, JavaScript, PHP, etc.
- Non-examples: C, C++
- Advance knowledge of references allows for precise introspection at runtime.

# Types of Garbage Collectors

- Incremental vs stop-the-world:
  - An **incremental** collector is one that runs concurrently with the program.
  - A **stop-the-world** collector pauses program execution to look for garbage.
  - Which is (generally) more precise?
  - Which would you use in a nuclear reactor control system?
- Compacting vs non-compacting:
  - A **compacting** collector is one that moves objects around in memory.
  - A **non-compacting** collector is one that leaves all objects where they originated.
  - Which (generally) spends more time garbage collecting?
  - Which (generally) leads to faster program execution?

# Reference Counting

# Reference Counting

- A simple framework for garbage collection.
  - Though it has several serious weaknesses!
- Idea: Store in each object a **reference count (refcount)** tracking how many references exist to the object.
- Creating a reference to an object increments its refcount.
- Removing a reference to an object decrements its refcount.
- When an object has zero refcount, it is unreachable and can be reclaimed.
  - This might decrease other objects' counts and trigger more reclamations.

# Reference Counting in Action



# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```

# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```

# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}
```

head

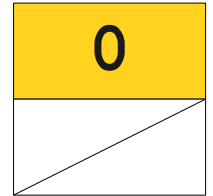


```
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```

# Reference Counting in Action

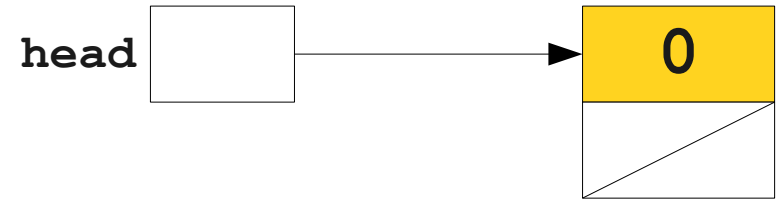
```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```

head



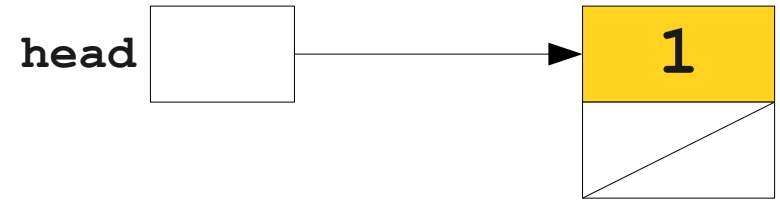
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



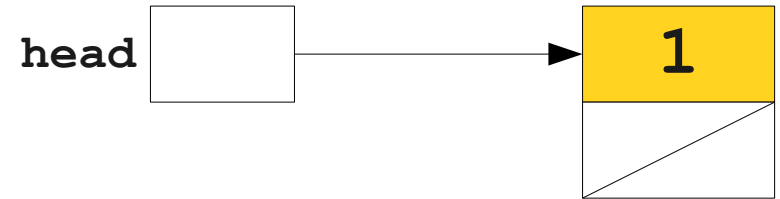
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



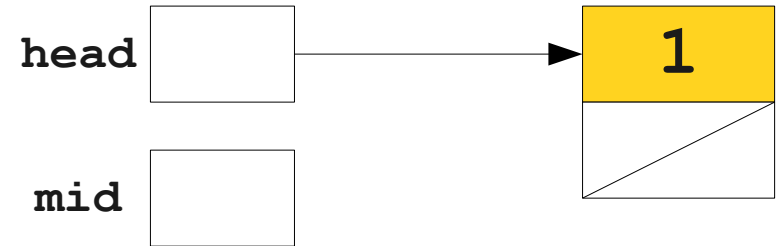
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



# Reference Counting in Action

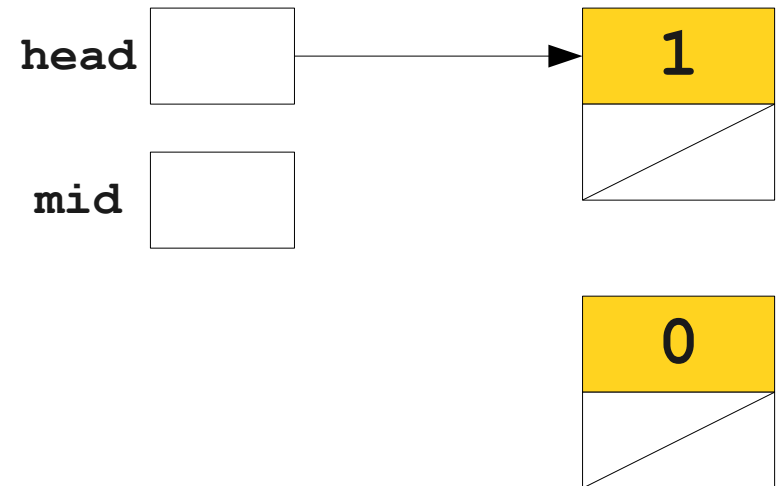
```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```





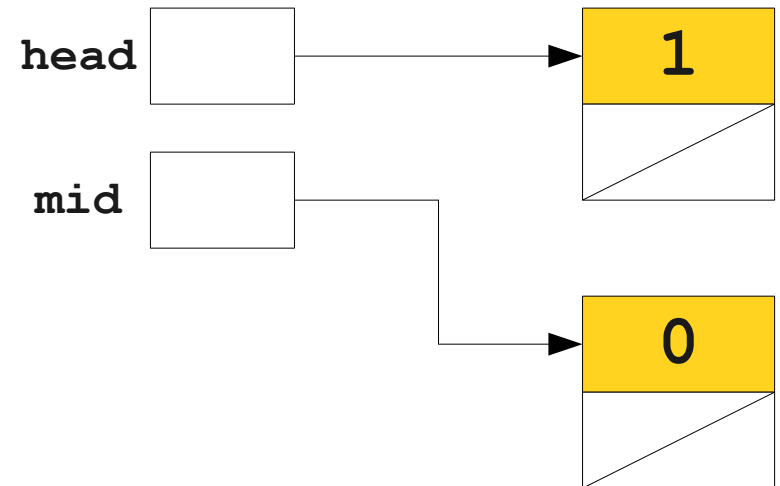
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



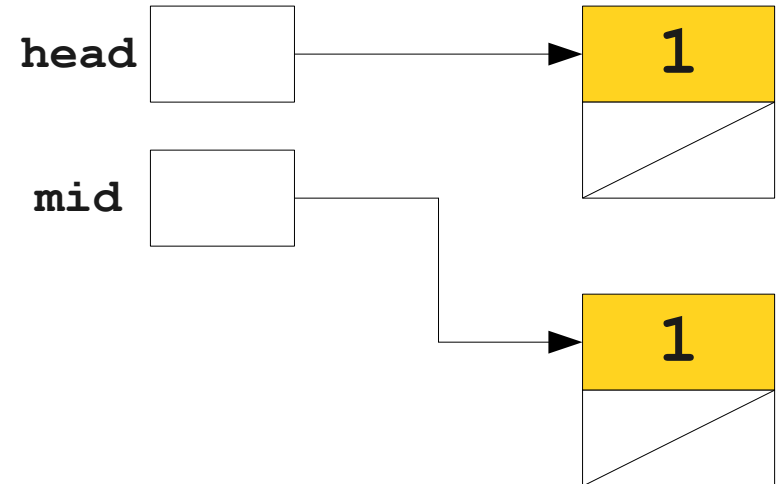
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



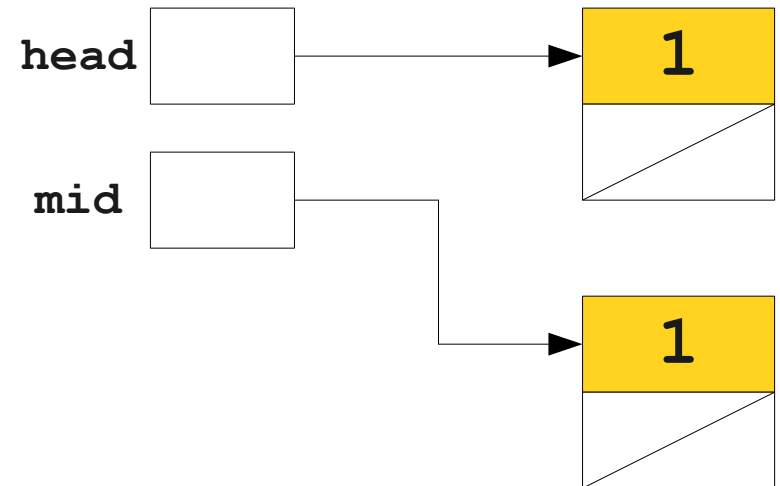
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



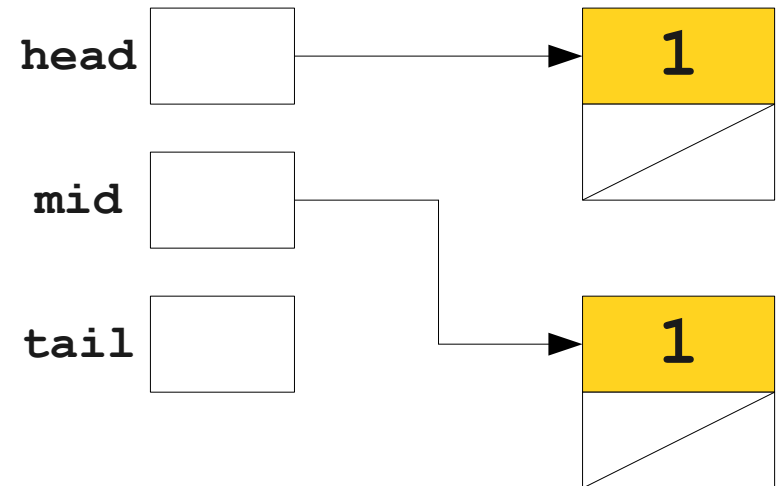
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



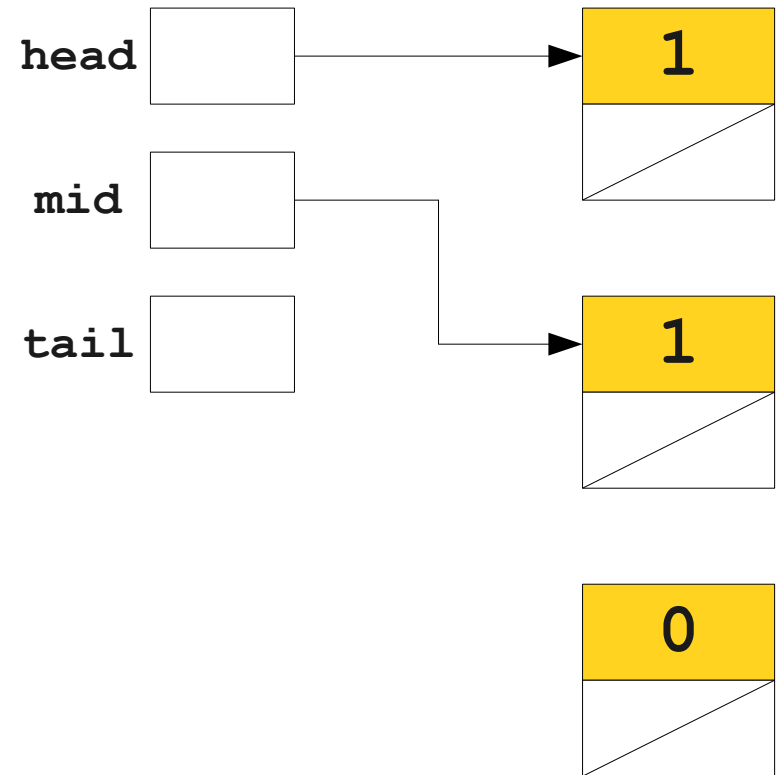
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



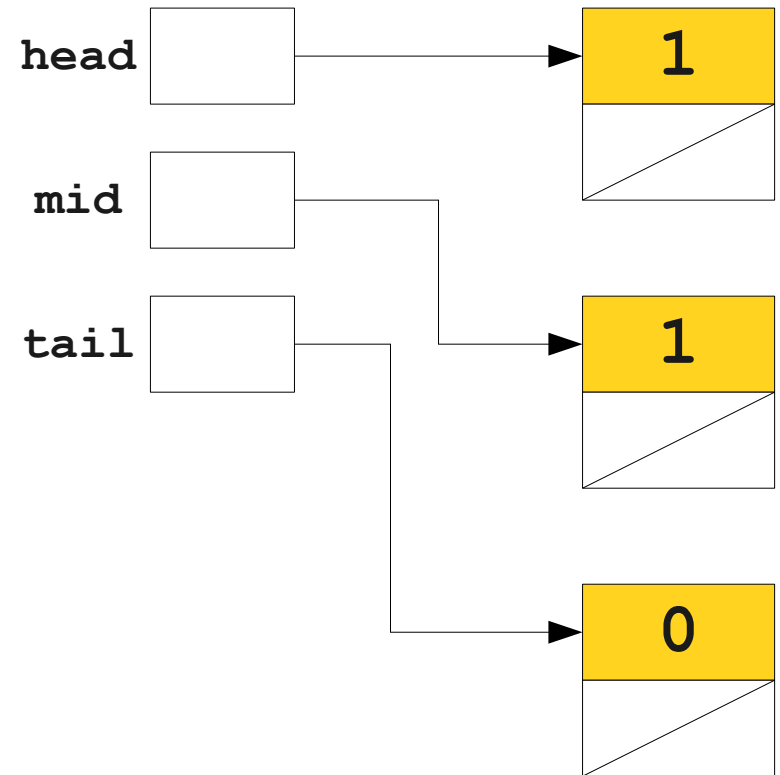
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



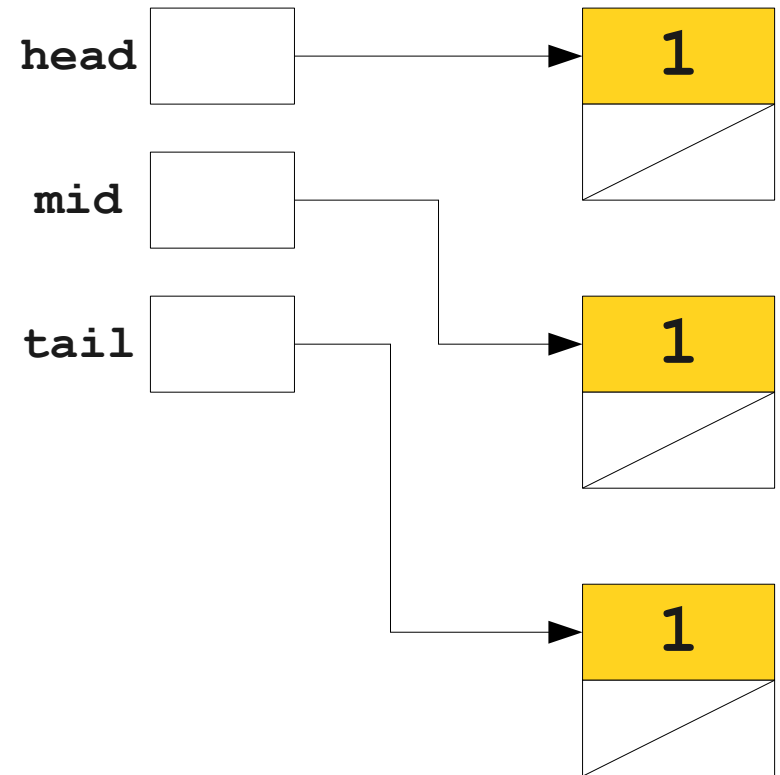
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



# Reference Counting in Action

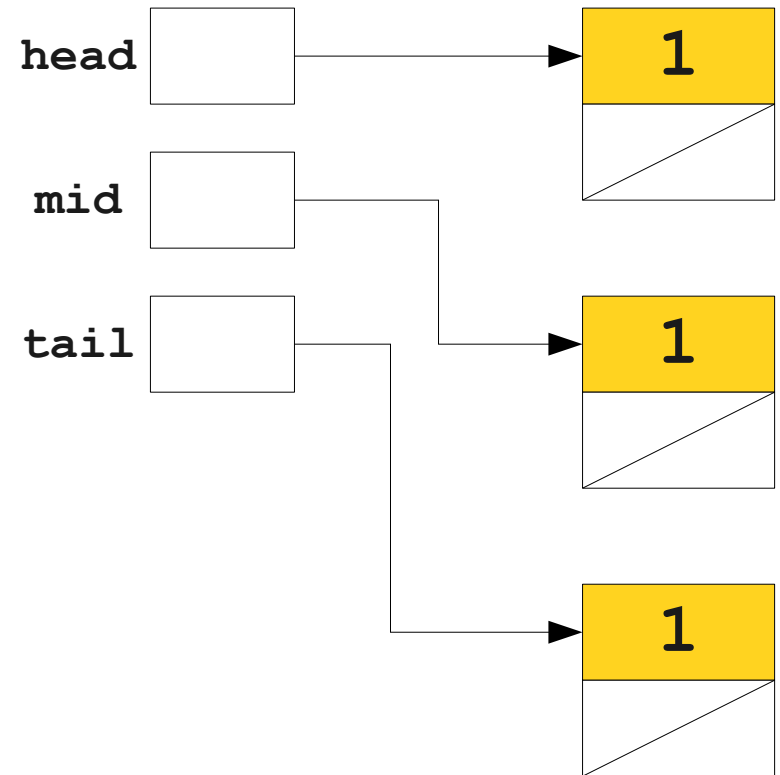
```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```





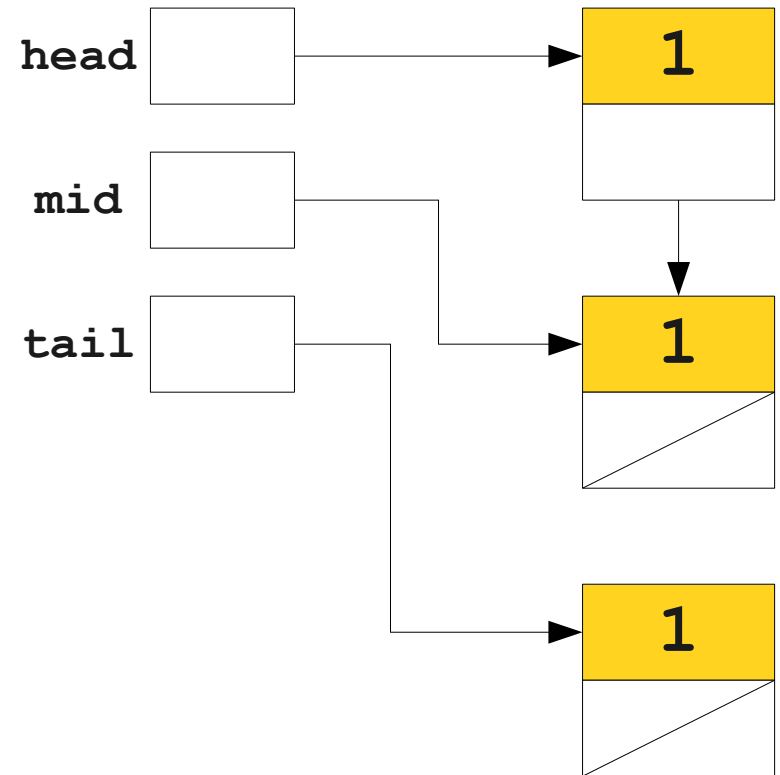
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



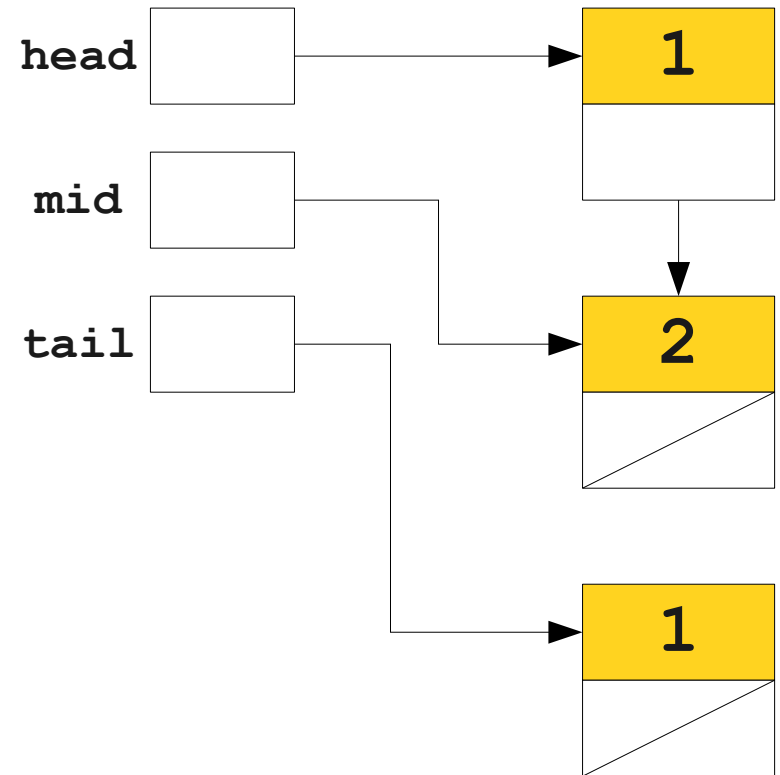
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



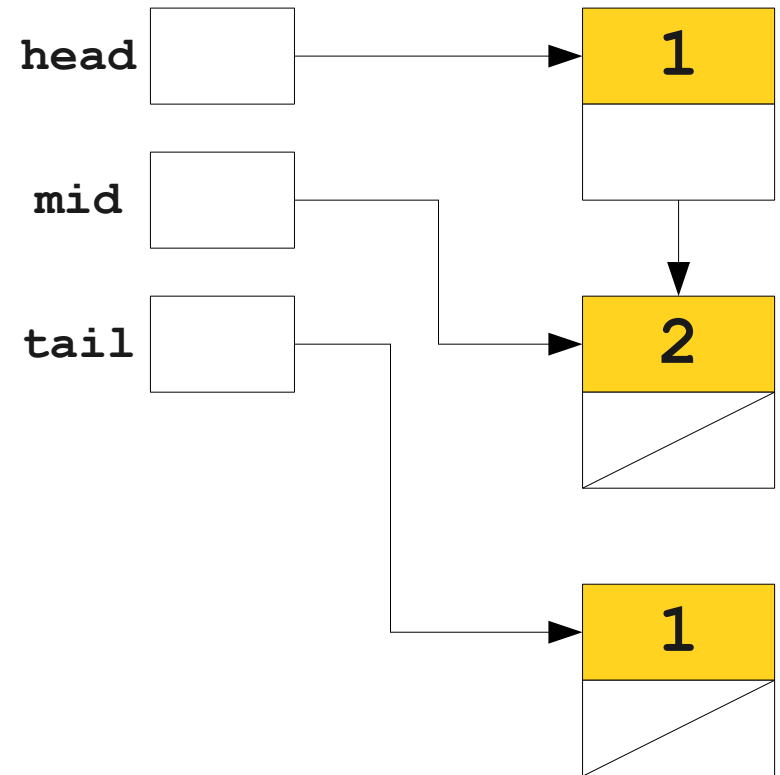
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



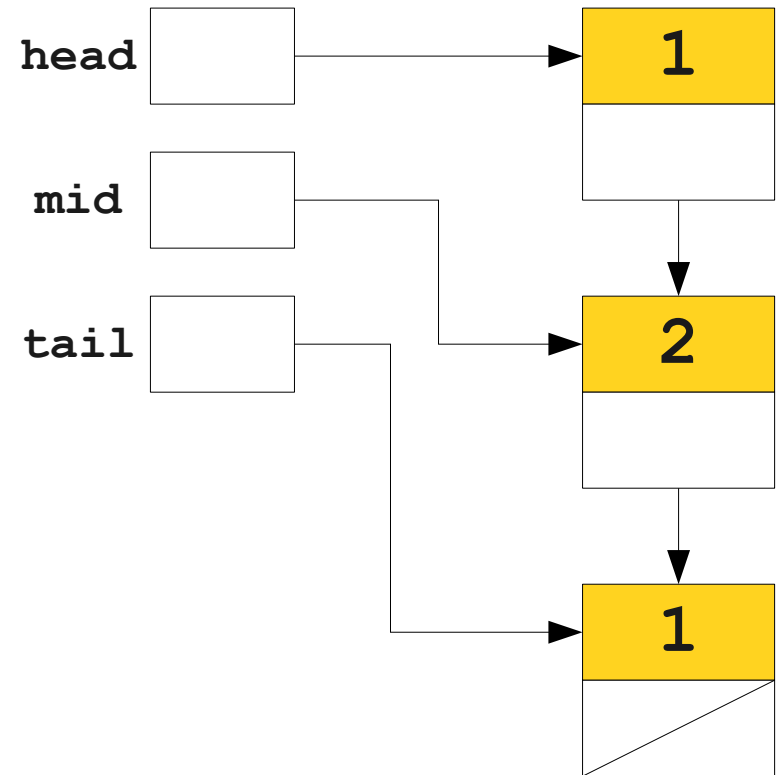
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



# Reference Counting in Action

```
class LinkedList {
    LinkedList next;
}

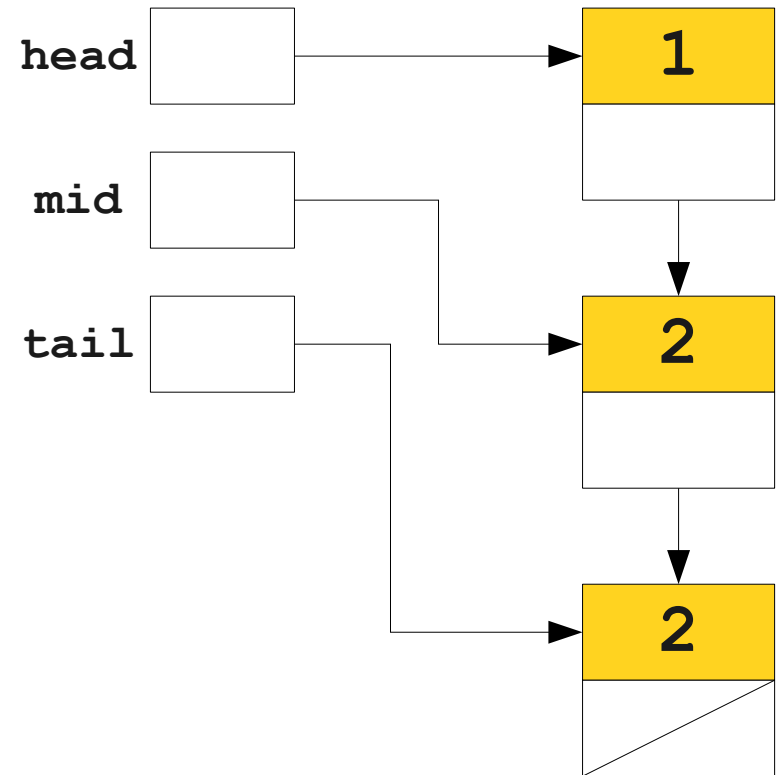
int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;

    mid = tail = null;

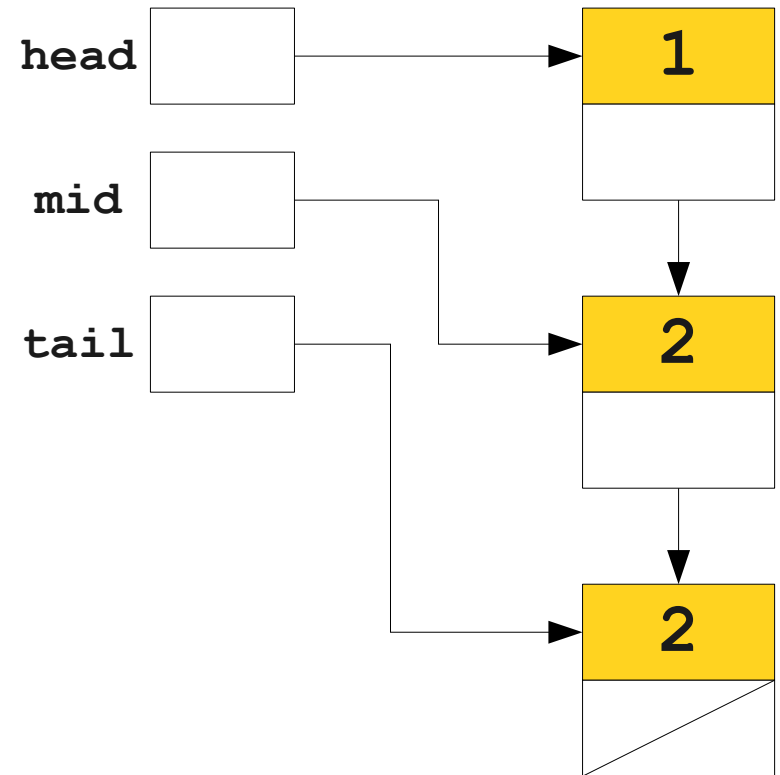
    head.next.next = null;

    head = null;
}
```



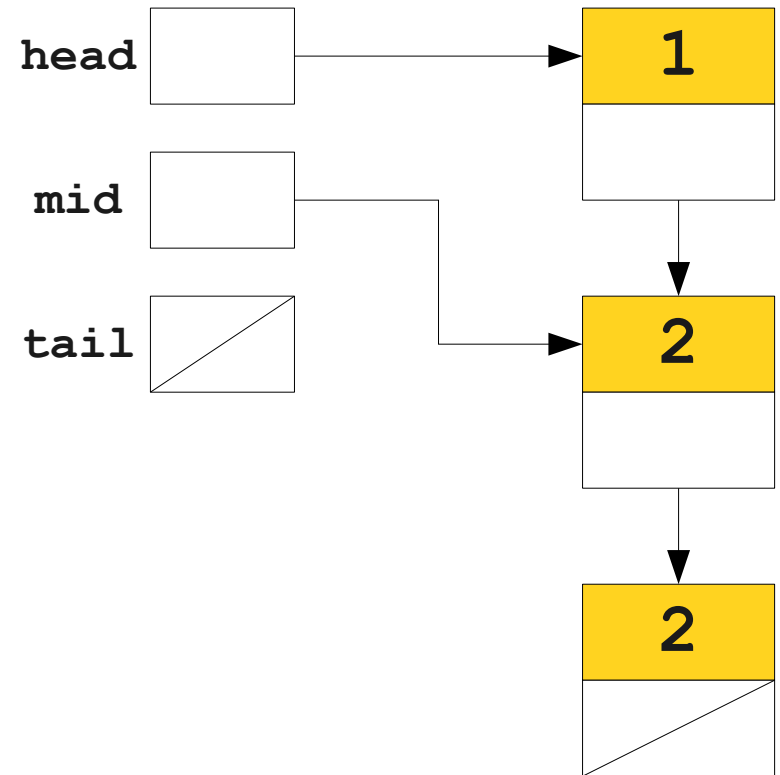
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



# Reference Counting in Action

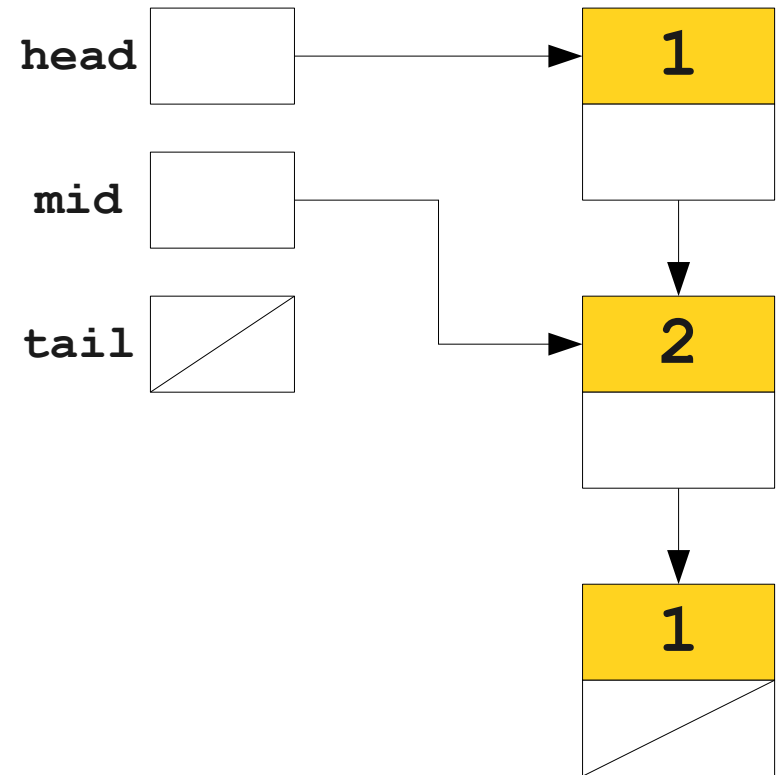
```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```





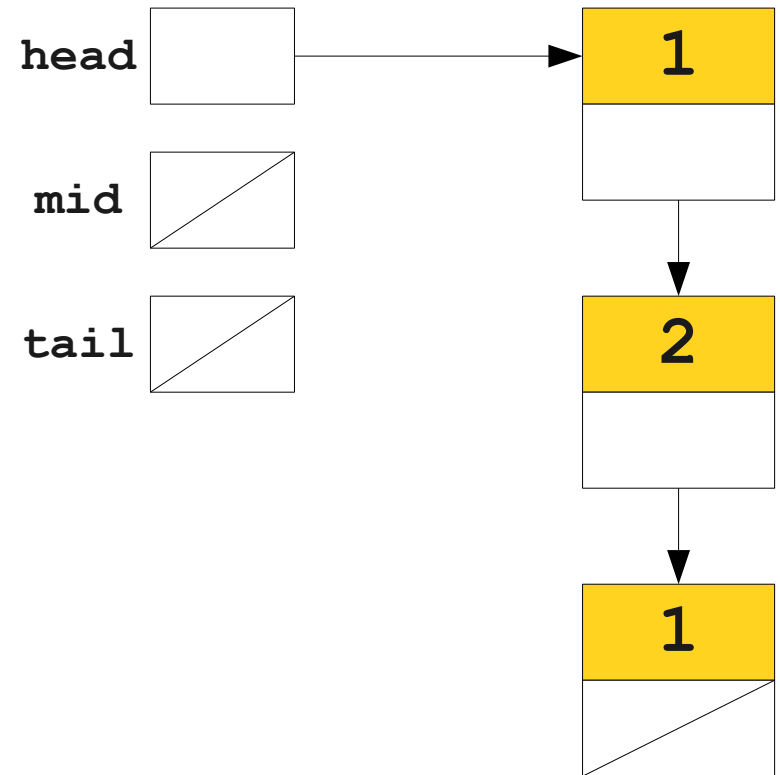
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



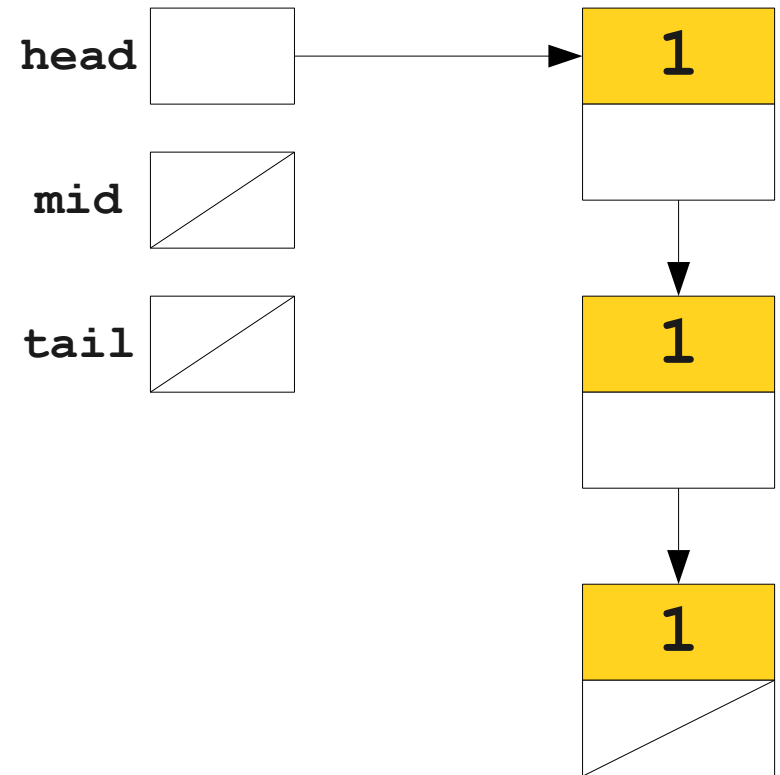
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



# Reference Counting in Action

```
class LinkedList {
    LinkedList next;
}

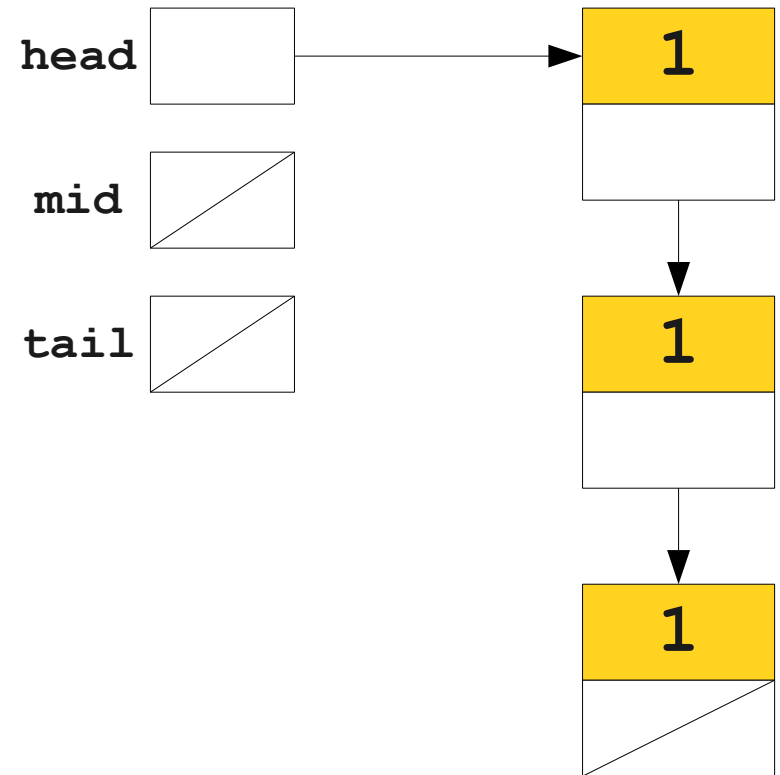
int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;

    mid = tail = null;

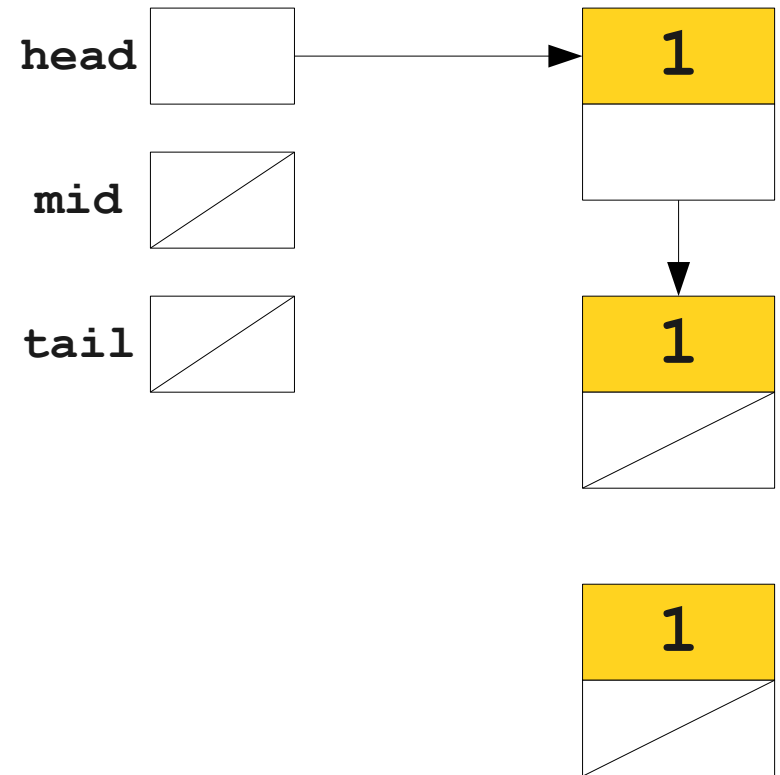
    head.next.next = null;

    head = null;
}
```



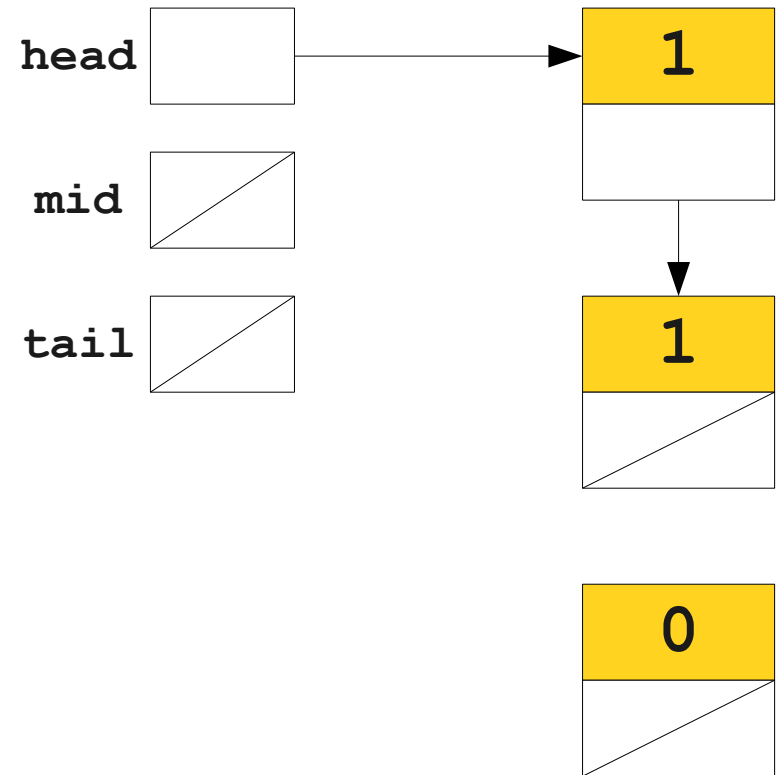
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



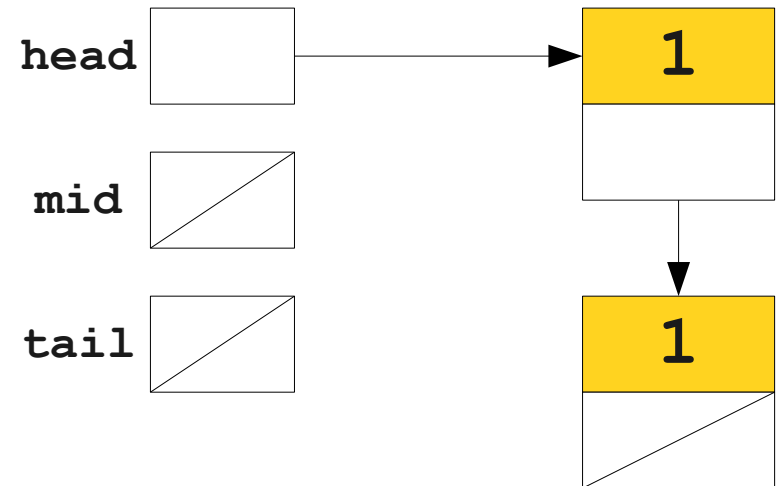
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



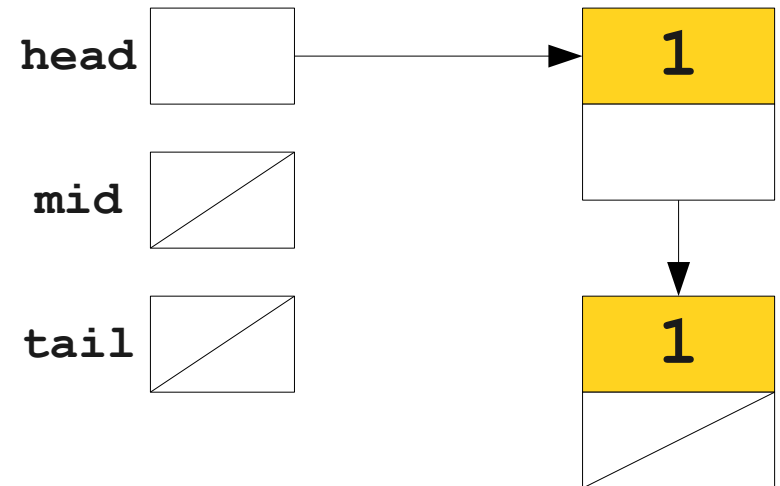
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



# Reference Counting in Action

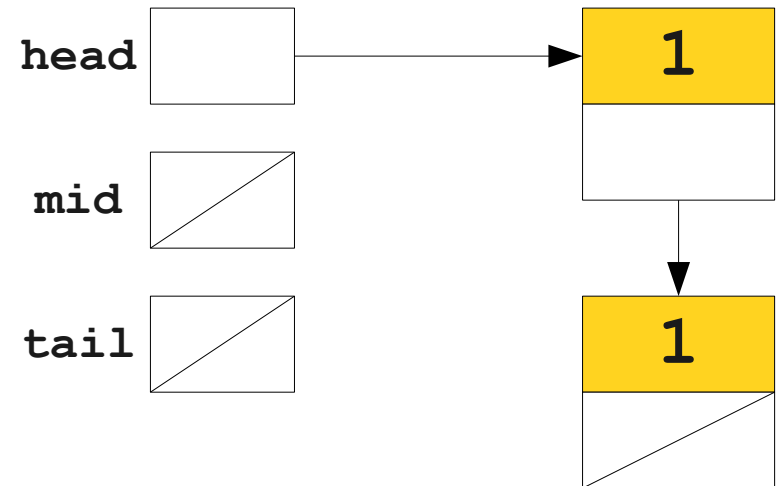
```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```





# Reference Counting in Action

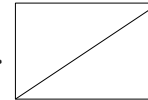
```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```



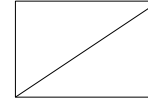
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```

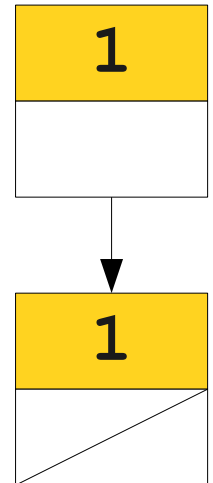
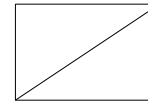
**head**



**mid**



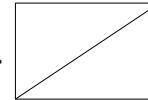
**tail**



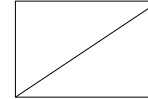
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```

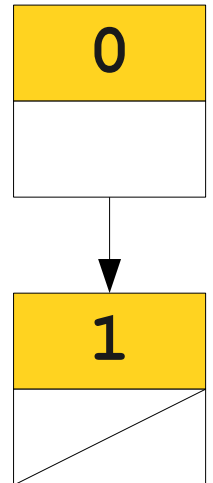
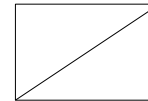
**head**



**mid**



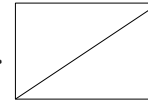
**tail**



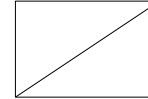
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```

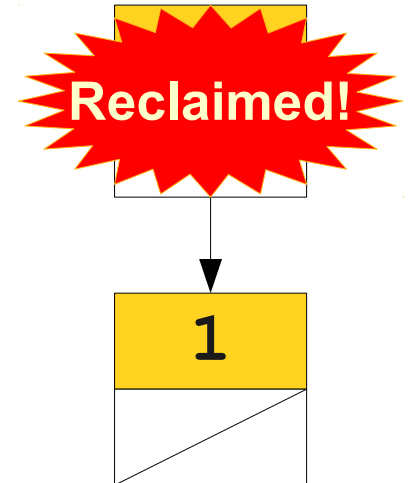
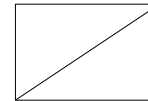
head



mid



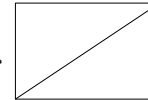
tail



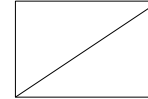
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```

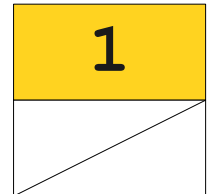
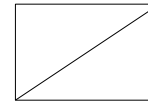
**head**



**mid**



**tail**



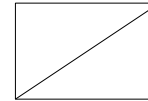
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```

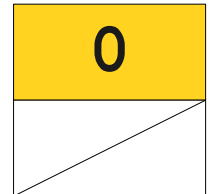
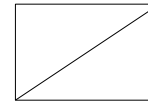
**head**



**mid**



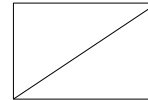
**tail**



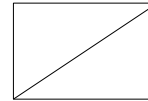
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```

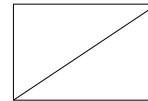
**head**



**mid**



**tail**

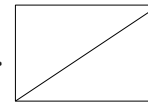


**Reclaimed!**

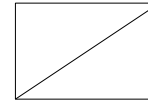
# Reference Counting in Action

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
  
    mid = tail = null;  
  
    head.next.next = null;  
  
    head = null;  
}
```

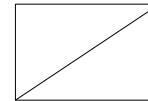
**head**



**mid**



**tail**





# Reference Counting Details

- When creating an object, set its refcount to 0.
- When creating a reference to an object, increment its refcount.
- When removing a reference from an object:
  - Decrement its refcount.
  - If its refcount is zero:
    - Remove all outgoing references from that object.
    - Reclaim the memory for that object.

# One Major Problem

# One Major Problem

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;
    tail.next = head;

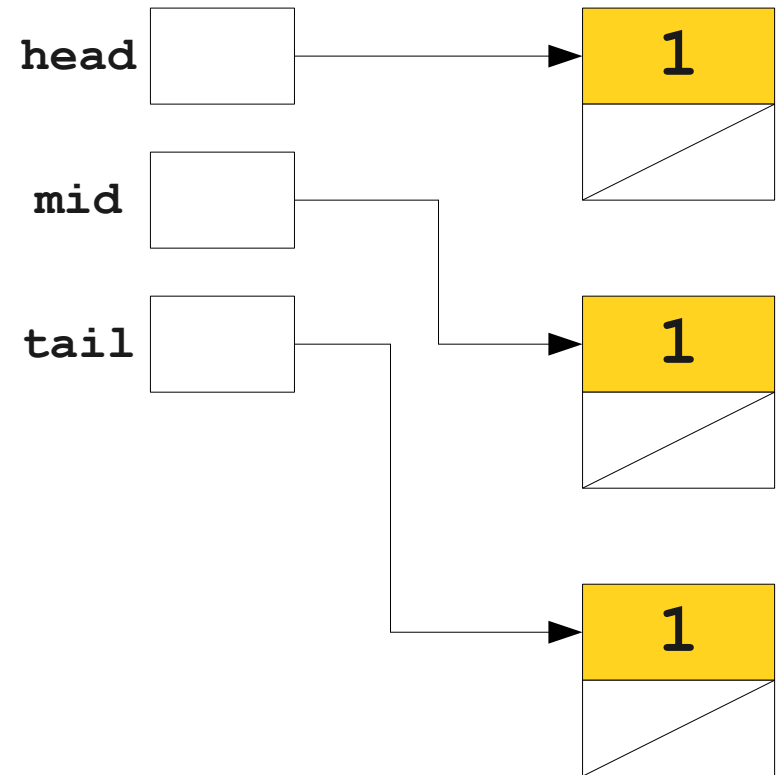
    head = null;
    mid = null;
    tail = null;
}
```

# One Major Problem

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
    tail.next = head;  
  
    head = null;  
    mid = null;  
    tail = null;  
}
```

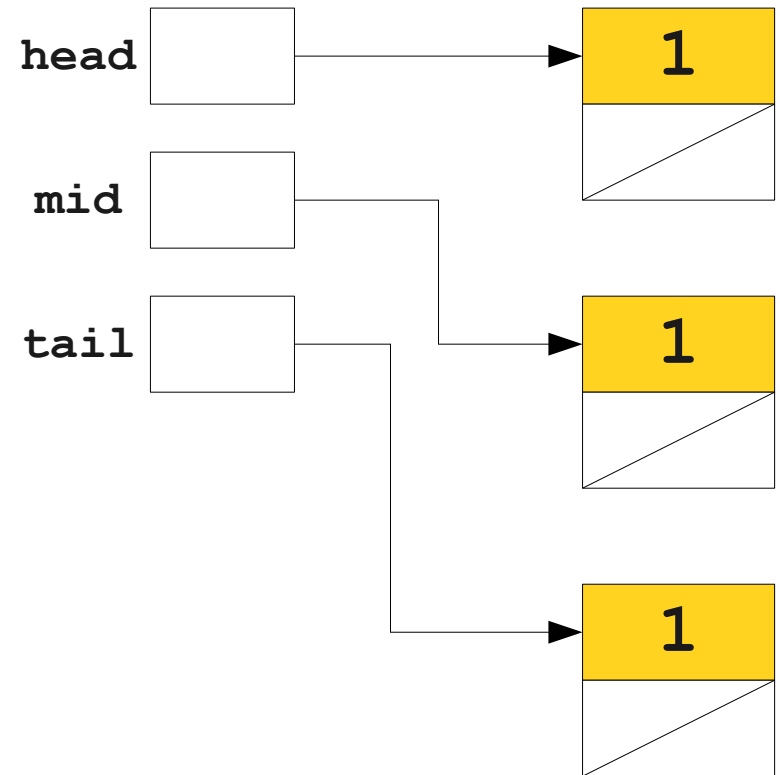
# One Major Problem

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
    tail.next = head;  
  
    head = null;  
    mid = null;  
    tail = null;  
}
```



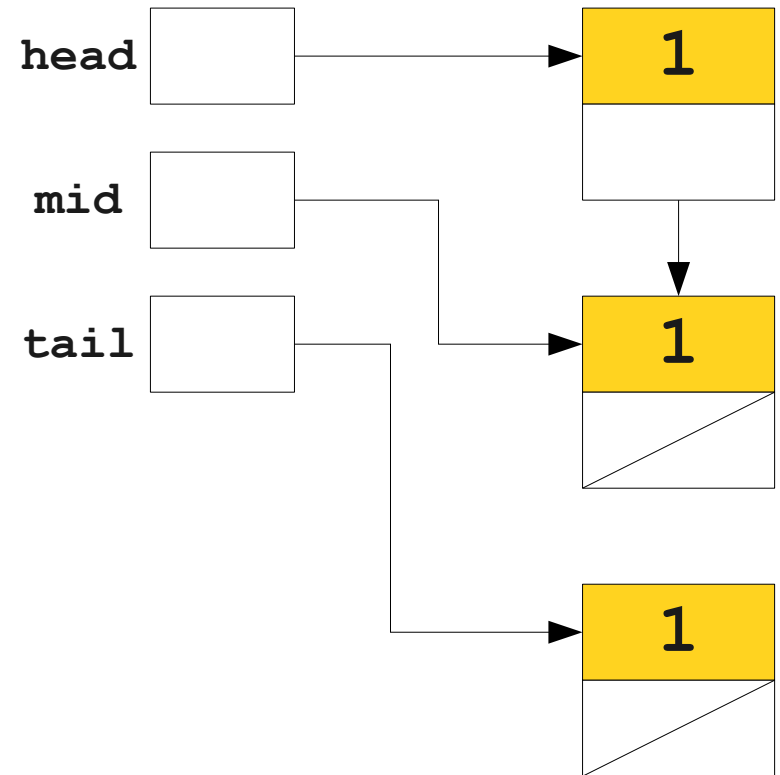
# One Major Problem

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
    tail.next = head;  
  
    head = null;  
    mid = null;  
    tail = null;  
}
```



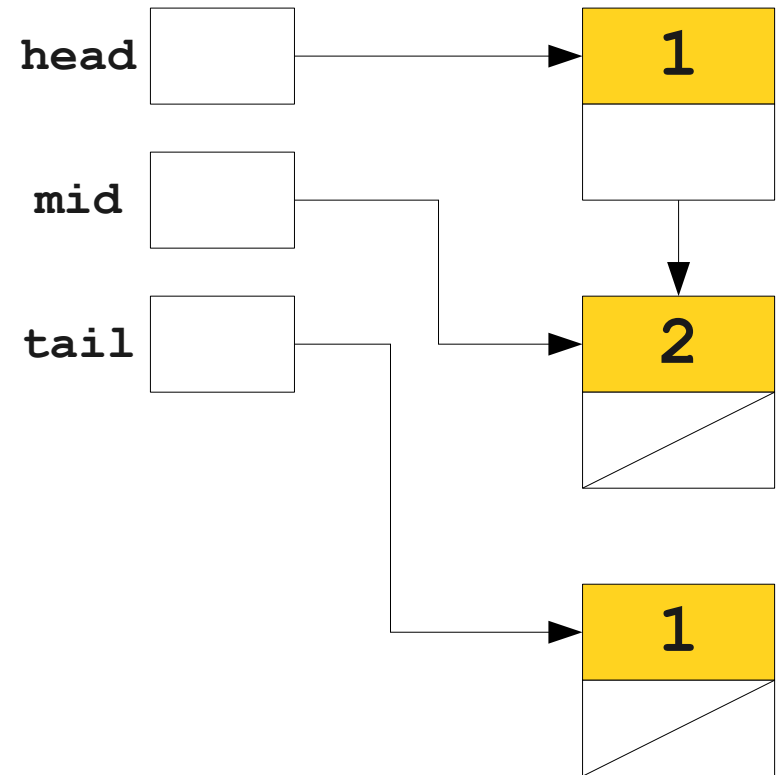
# One Major Problem

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
    tail.next = head;  
  
    head = null;  
    mid = null;  
    tail = null;  
}
```



# One Major Problem

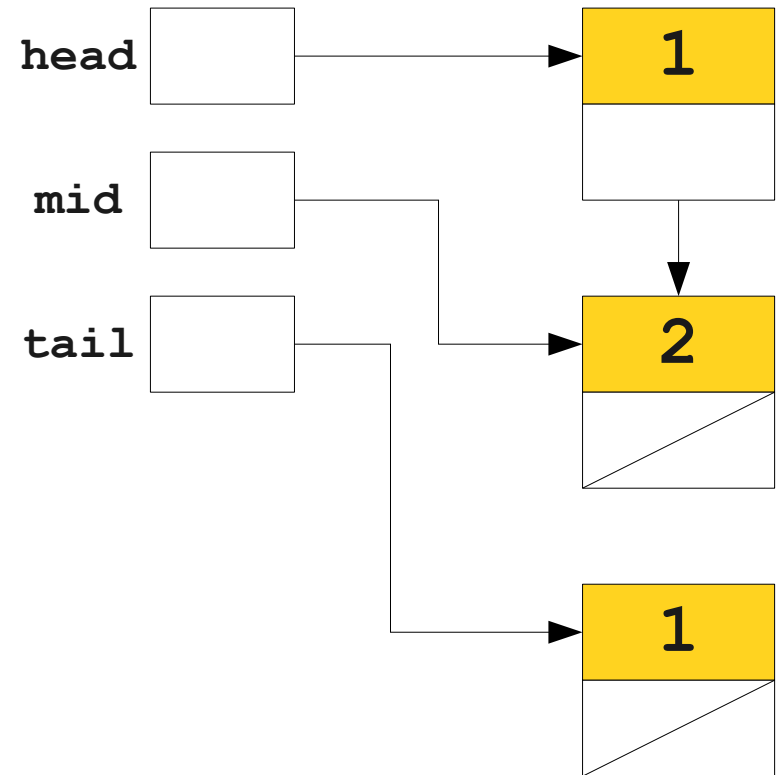
```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
    tail.next = head;  
  
    head = null;  
    mid = null;  
    tail = null;  
}
```





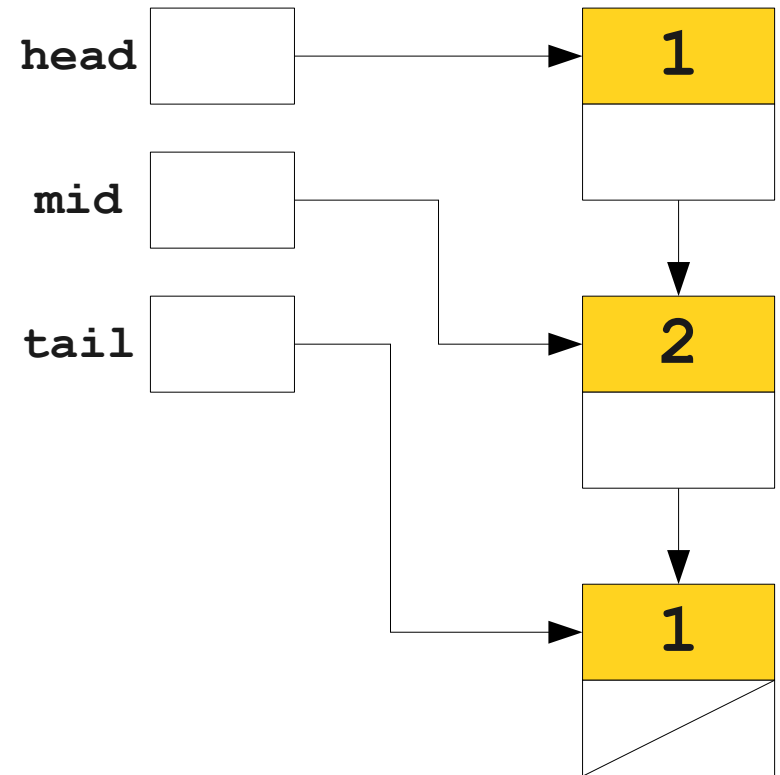
# One Major Problem

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
    tail.next = head;  
  
    head = null;  
    mid = null;  
    tail = null;  
}
```



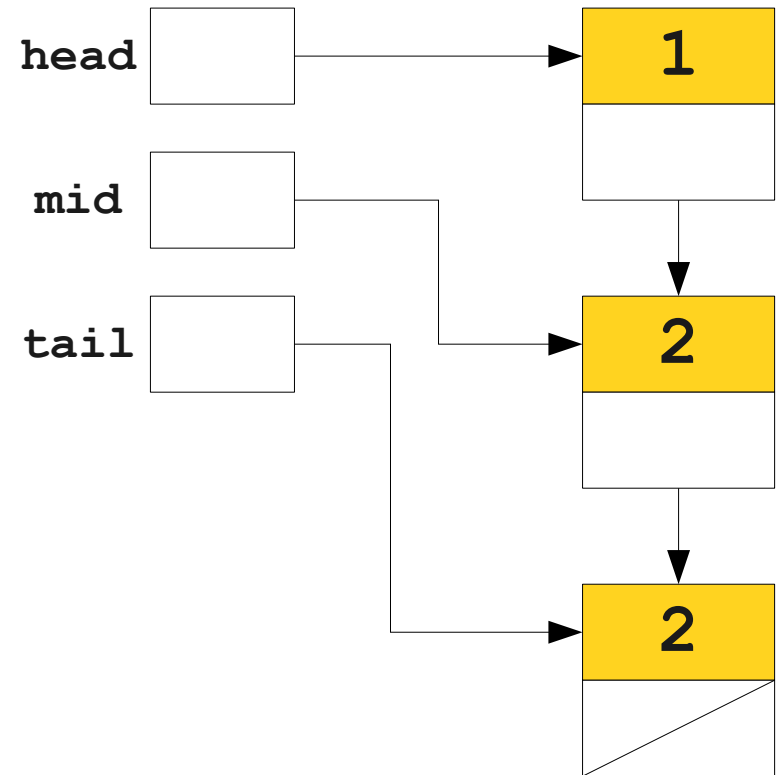
# One Major Problem

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
    tail.next = head;  
  
    head = null;  
    mid = null;  
    tail = null;  
}
```



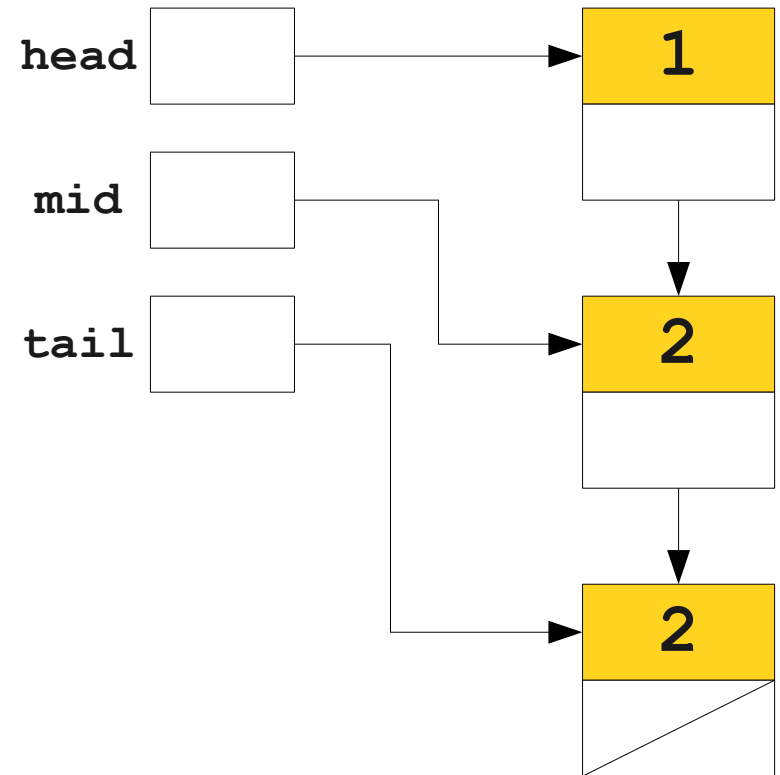
# One Major Problem

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
    tail.next = head;  
  
    head = null;  
    mid = null;  
    tail = null;  
}
```



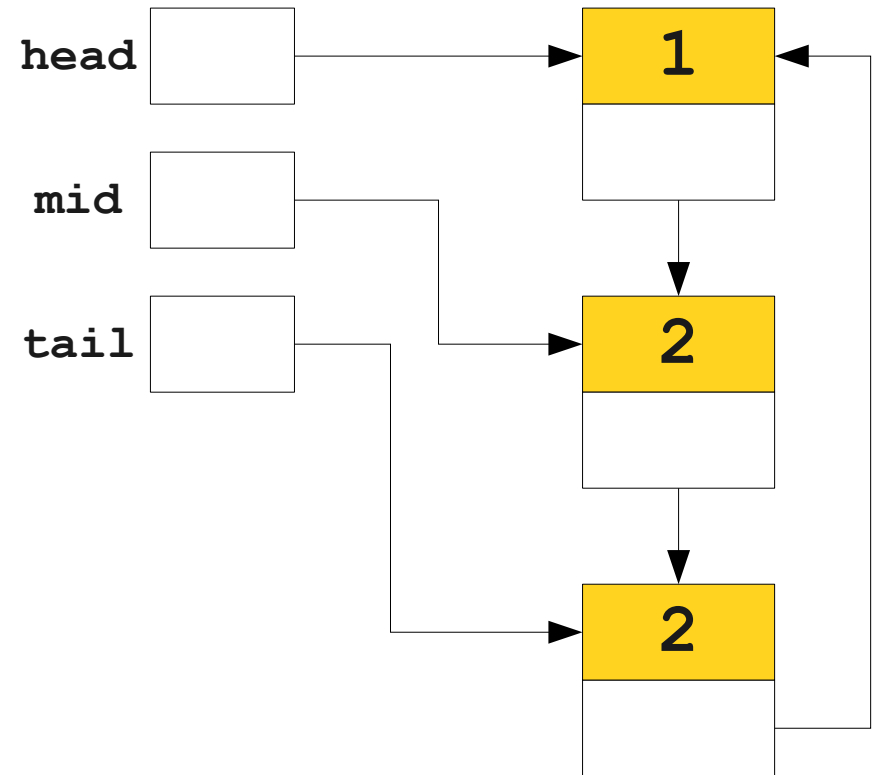
# One Major Problem

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
    tail.next = head;  
  
    head = null;  
    mid = null;  
    tail = null;  
}
```



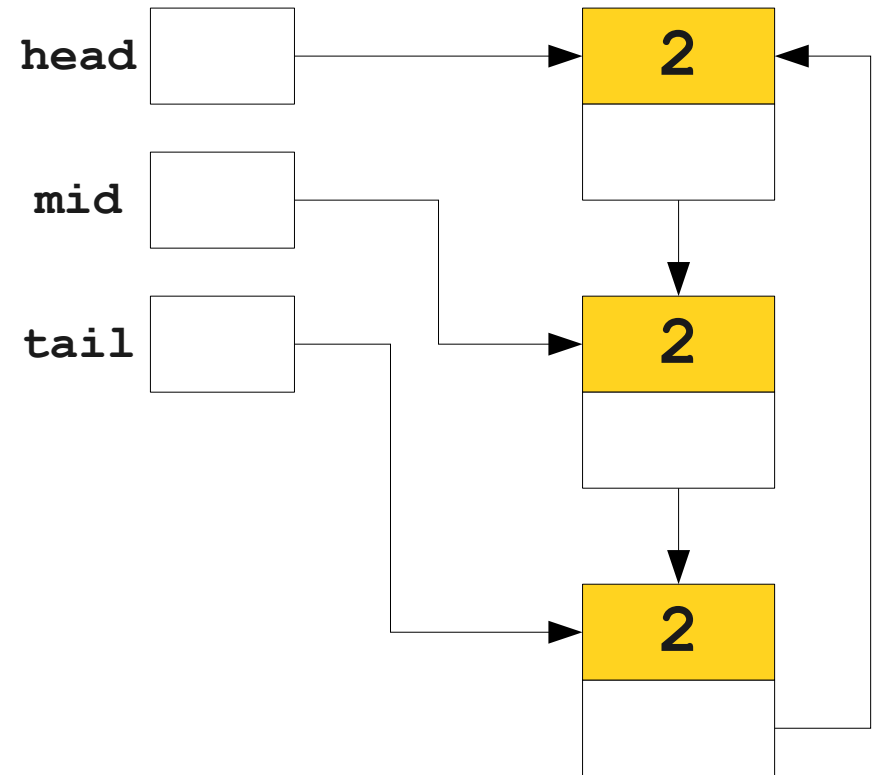
# One Major Problem

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
    tail.next = head;  
  
    head = null;  
    mid = null;  
    tail = null;  
}
```



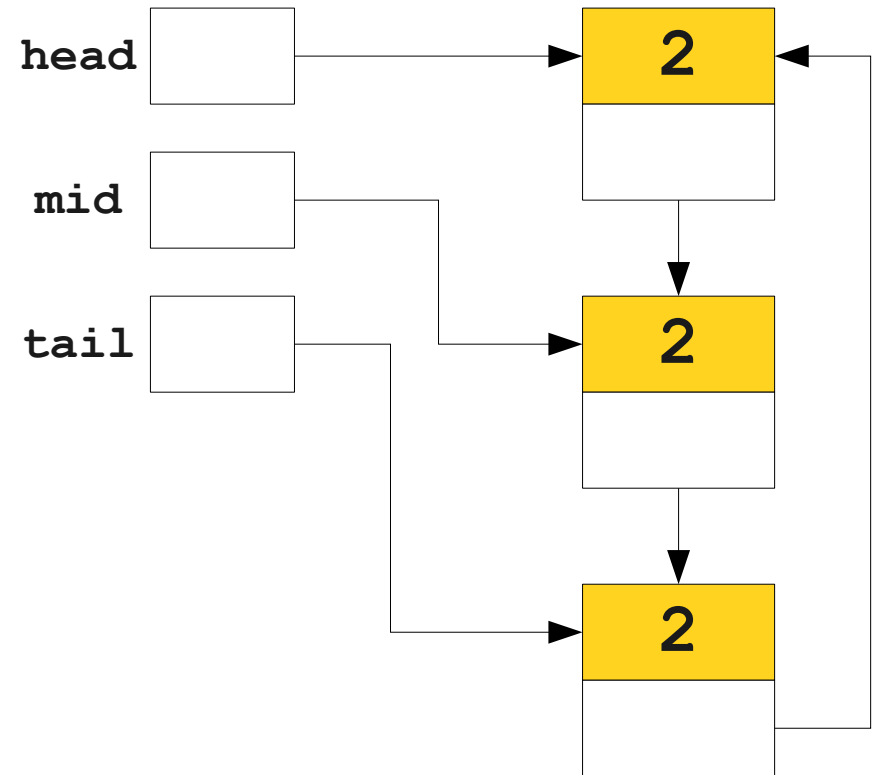
# One Major Problem

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
    tail.next = head;  
  
    head = null;  
    mid = null;  
    tail = null;  
}
```



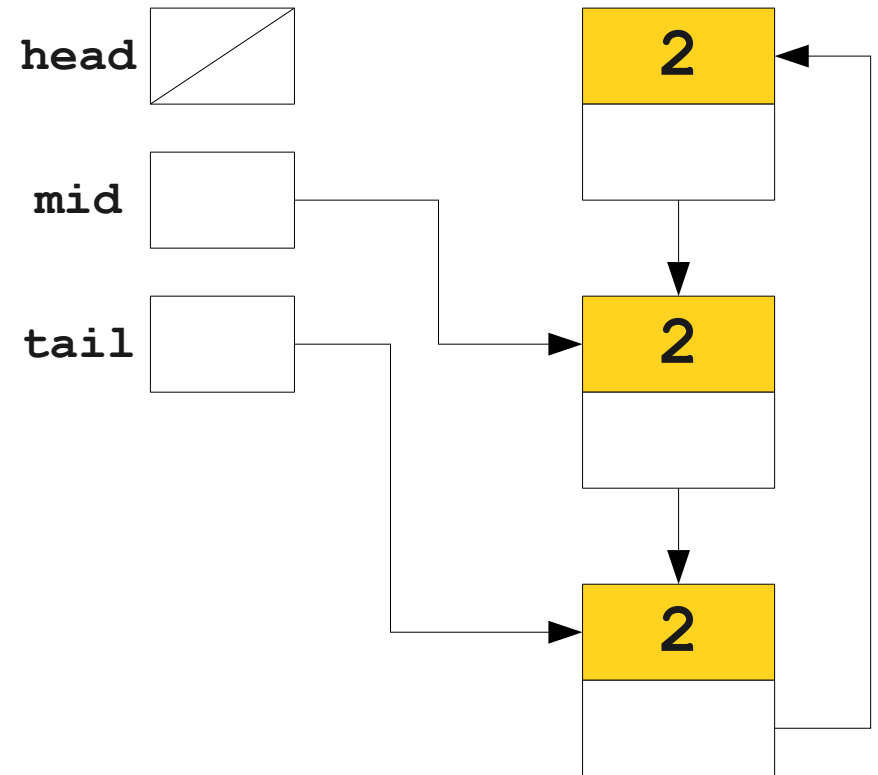
# One Major Problem

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
    tail.next = head;  
  
    head = null;  
    mid = null;  
    tail = null;  
}
```



# One Major Problem

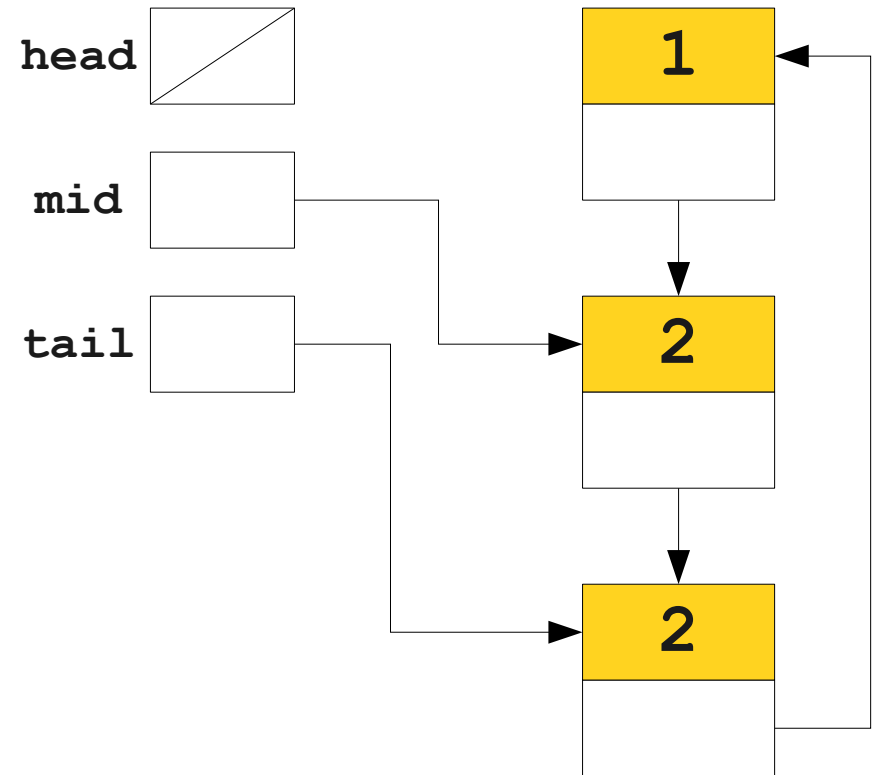
```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
    tail.next = head;  
  
    head = null;  
    mid = null;  
    tail = null;  
}
```





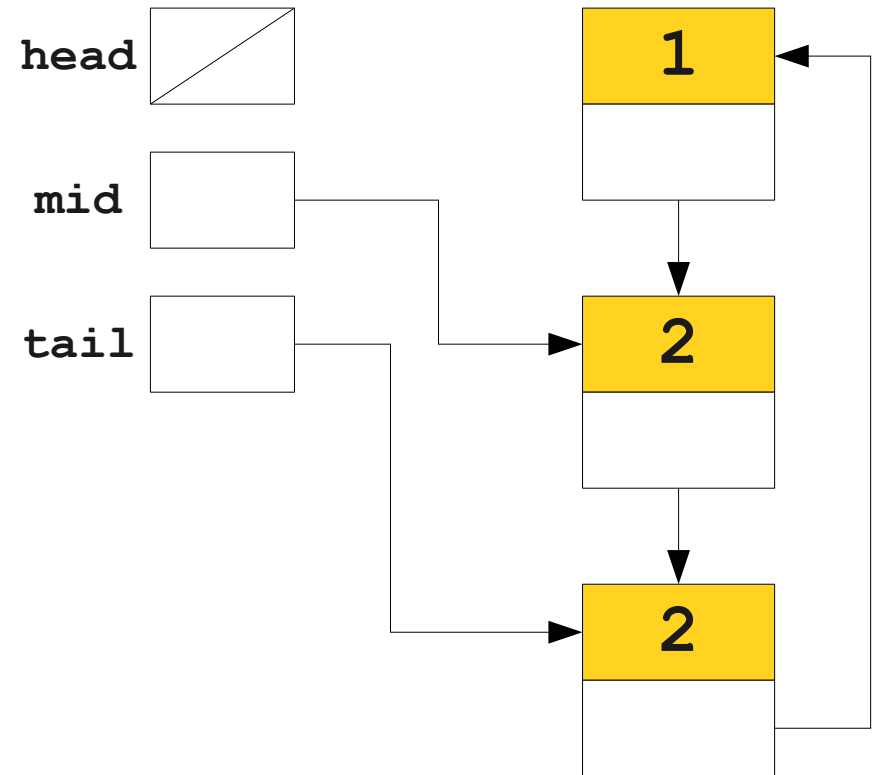
# One Major Problem

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
    tail.next = head;  
  
    head = null;  
    mid = null;  
    tail = null;  
}
```



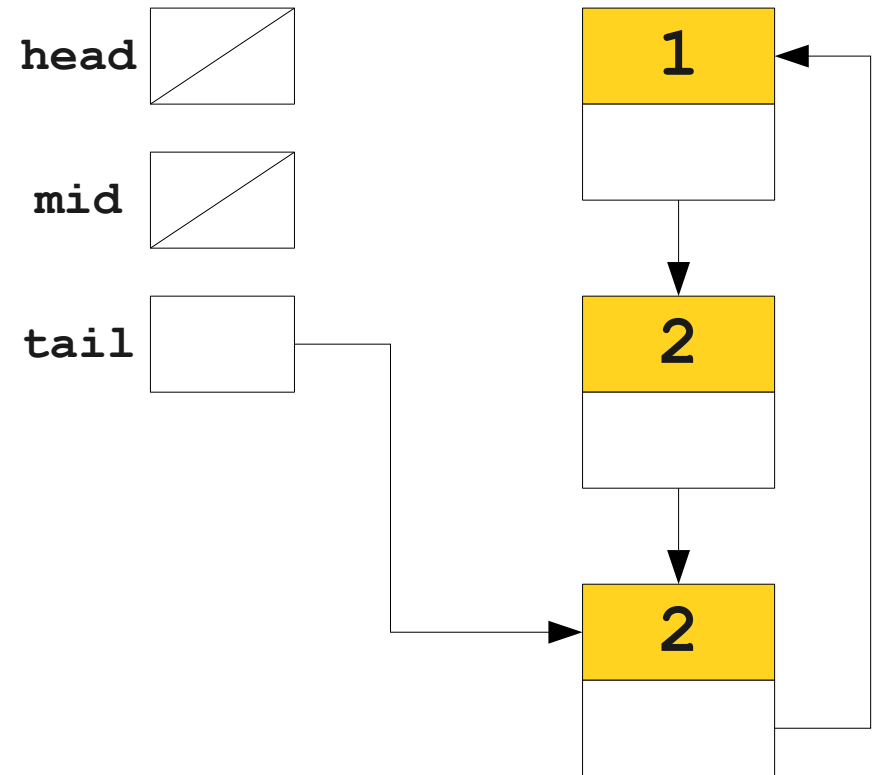
# One Major Problem

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
    tail.next = head;  
  
    head = null;  
    mid = null;  
    tail = null;  
}
```



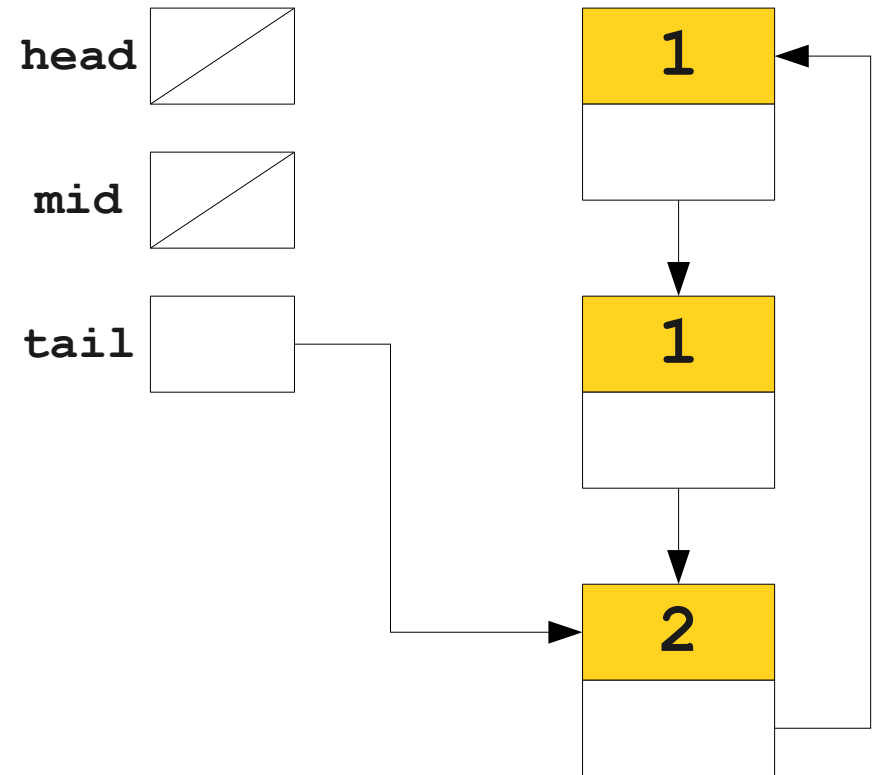
# One Major Problem

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
    tail.next = head;  
  
    head = null;  
    mid = null;  
    tail = null;  
}
```



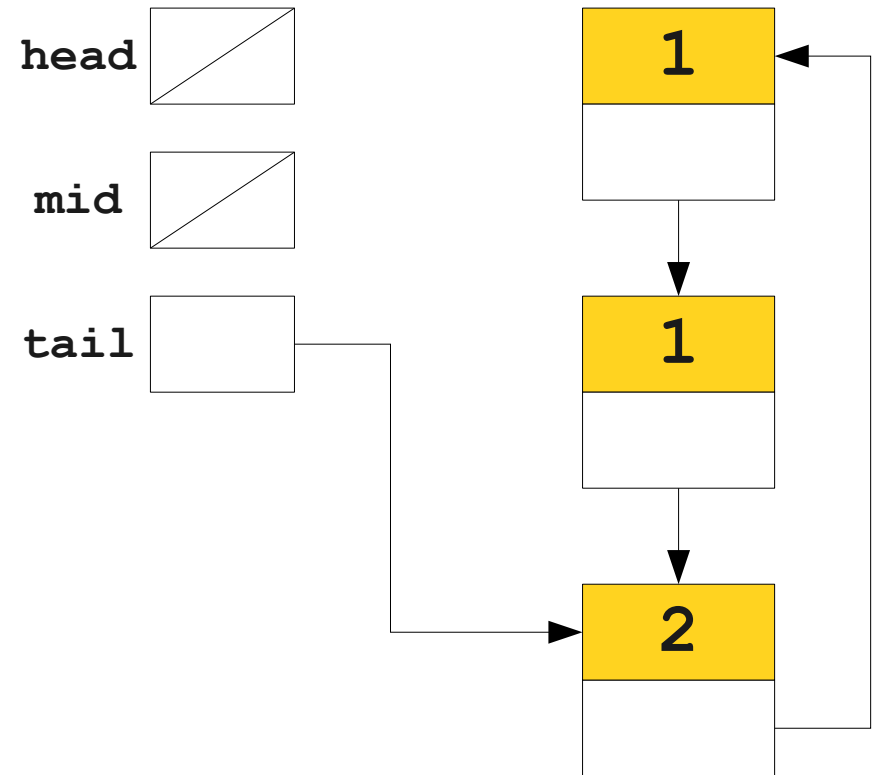
# One Major Problem

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
    tail.next = head;  
  
    head = null;  
    mid = null;  
    tail = null;  
}
```



# One Major Problem

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
    tail.next = head;  
  
    head = null;  
    mid = null;  
    tail = null;  
}
```



# One Major Problem

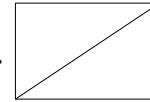
```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

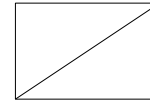
    head.next = mid;
    mid.next = tail;
    tail.next = head;

    head = null;
    mid = null;
    tail = null;
}
```

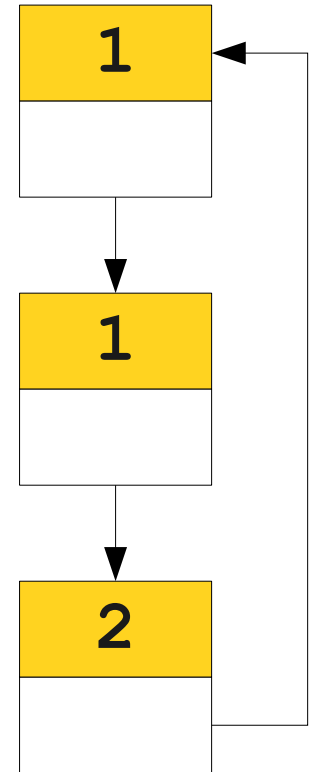
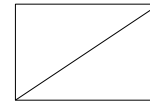
**head**



**mid**



**tail**



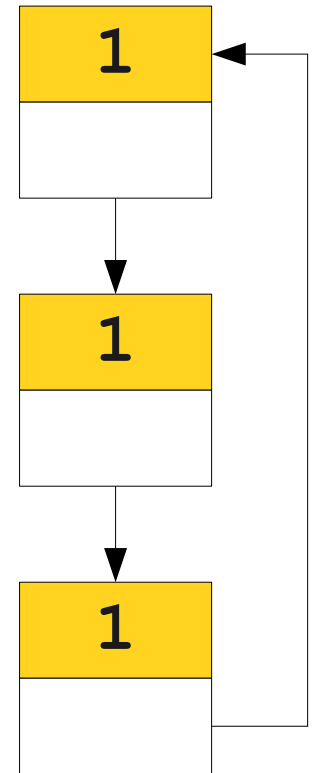
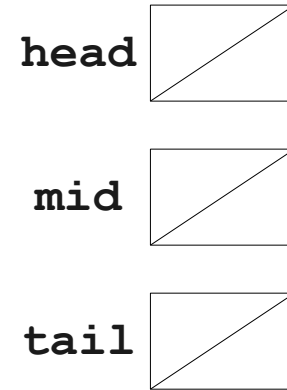
# One Major Problem

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;
    tail.next = head;

    head = null;
    mid = null;
    tail = null;
}
```



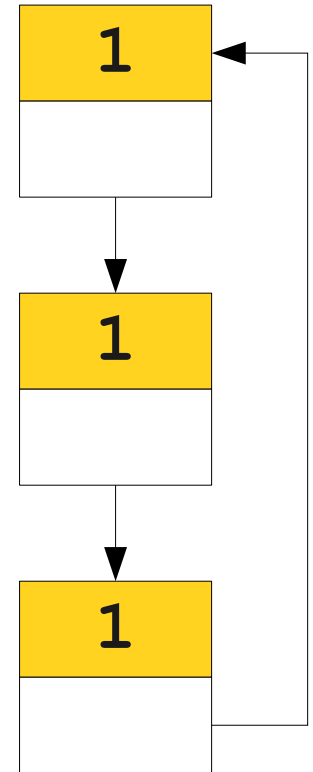
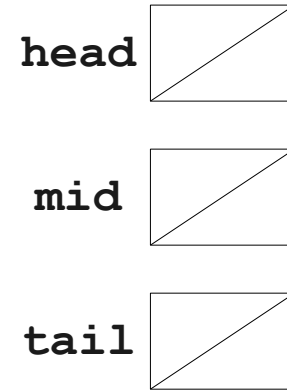
# One Major Problem

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;
    tail.next = head;

    head = null;
    mid = null;
    tail = null;
}
```

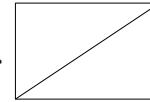




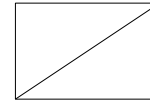
# One Major Problem

```
class LinkedList {  
    LinkedList next;  
}  
  
int main() {  
    LinkedList head = new LinkedList;  
    LinkedList mid = new LinkedList;  
    LinkedList tail = new LinkedList;  
  
    head.next = mid;  
    mid.next = tail;  
    tail.next = head;  
  
    head = null;  
    mid = null;  
    tail = null;  
}
```

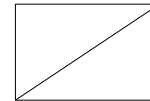
**head**



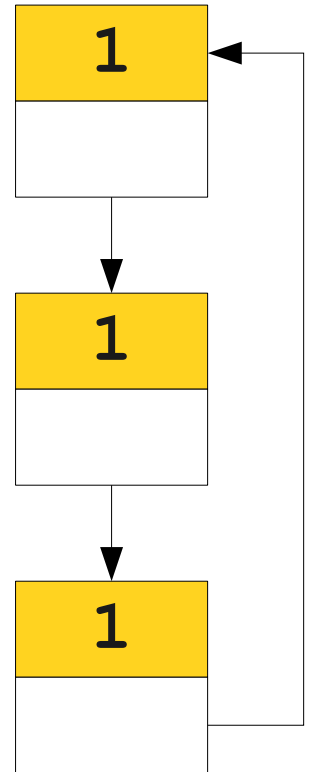
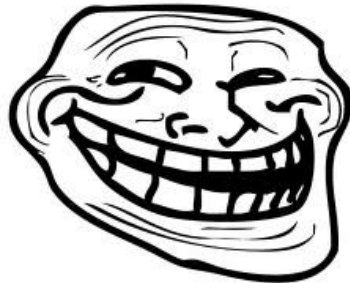
**mid**



**tail**



Problem?



# Reference Cycles

- A **reference cycle** is a set of objects that cyclically refer to one another.
- Because all the objects are referenced, all have nonzero refcounts and are never reclaimed.
- Issue: Refcount tracks number of references, not number of *reachable* references.
- Major problems in languages/systems that use reference counting:
  - e.g. Perl, Firefox 2.

# Analysis of Reference Counting

- Advantages:
  - Simple to implement.
  - Can be implemented as a library on top of explicit memory management (see C++ `shared_ptr`).
- Disadvantages:
  - Fails to reclaim all unreachable objects.
  - Can be slow if a large collection is initiated.
  - Noticeably slows down assignments.

# Mark-and-Sweep

# Reachability Revisited

- Recall that the goal of our garbage collector is to reclaim all unreachable objects.
- Reference counting tries to find unreachable objects by finding objects with no incoming references.
- Imprecise because we forget **which** references those are.

# Mark-and-Sweep: The Intuition

- **Intuition:** Given knowledge of what's immediately accessible, find everything reachable in the program.
- The **root set** is the set of memory locations in the program that are known to be reachable.
  - such as?
- Any objects reachable from the root set are reachable.
- Any objects not reachable from the root set are not reachable.
- Do a **graph search** starting at the root set!

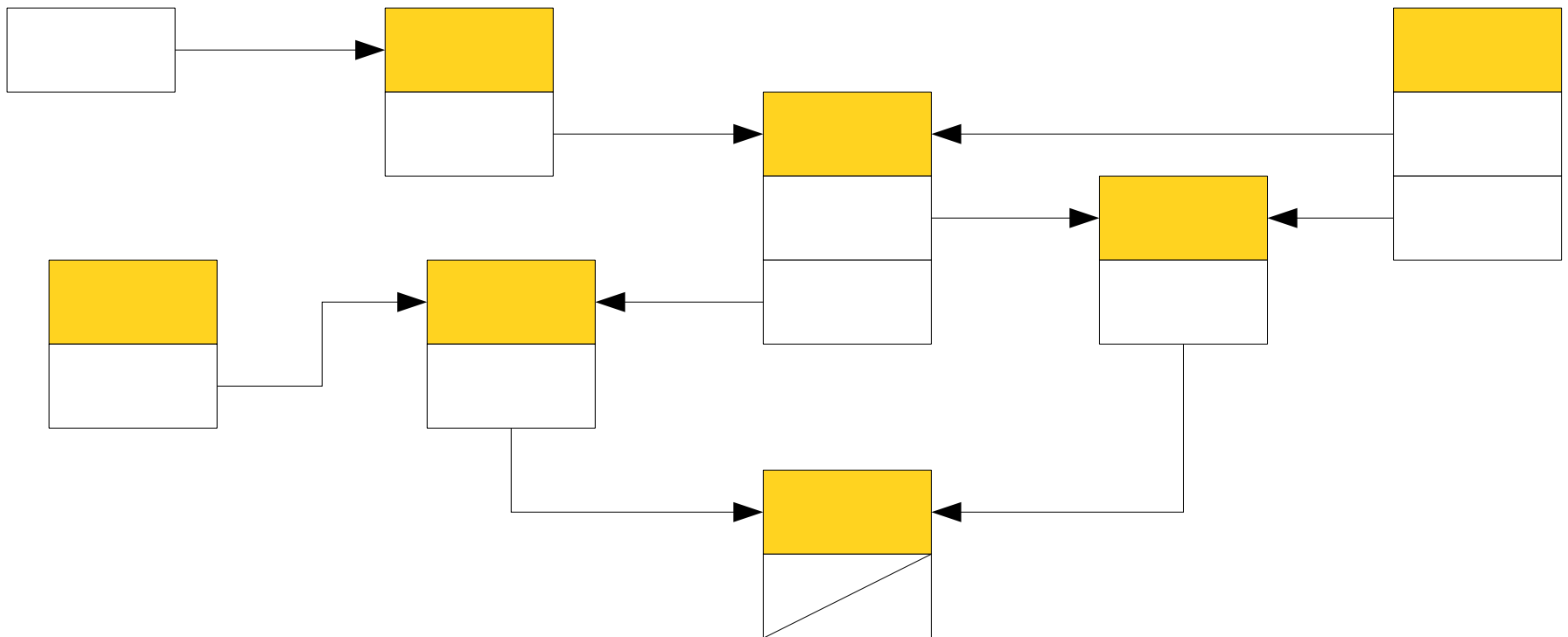
# Mark-and-Sweep: The Algorithm

- Mark-and-sweep runs in two phases.
- **Marking phase:** Find reachable objects.
  - Add the root set to a worklist.
  - While the worklist isn't empty:
    - Remove an object from the worklist.
    - If it is not **marked**, mark it and add to the worklist all objects reachable from that object.
- **Sweeping phase:** Reclaim free memory.
  - For each allocated object:
    - If that object isn't **marked**, reclaim its memory.
    - If the object is marked, **unmark** it (so on the next mark-and-sweep iteration we have to mark it again).

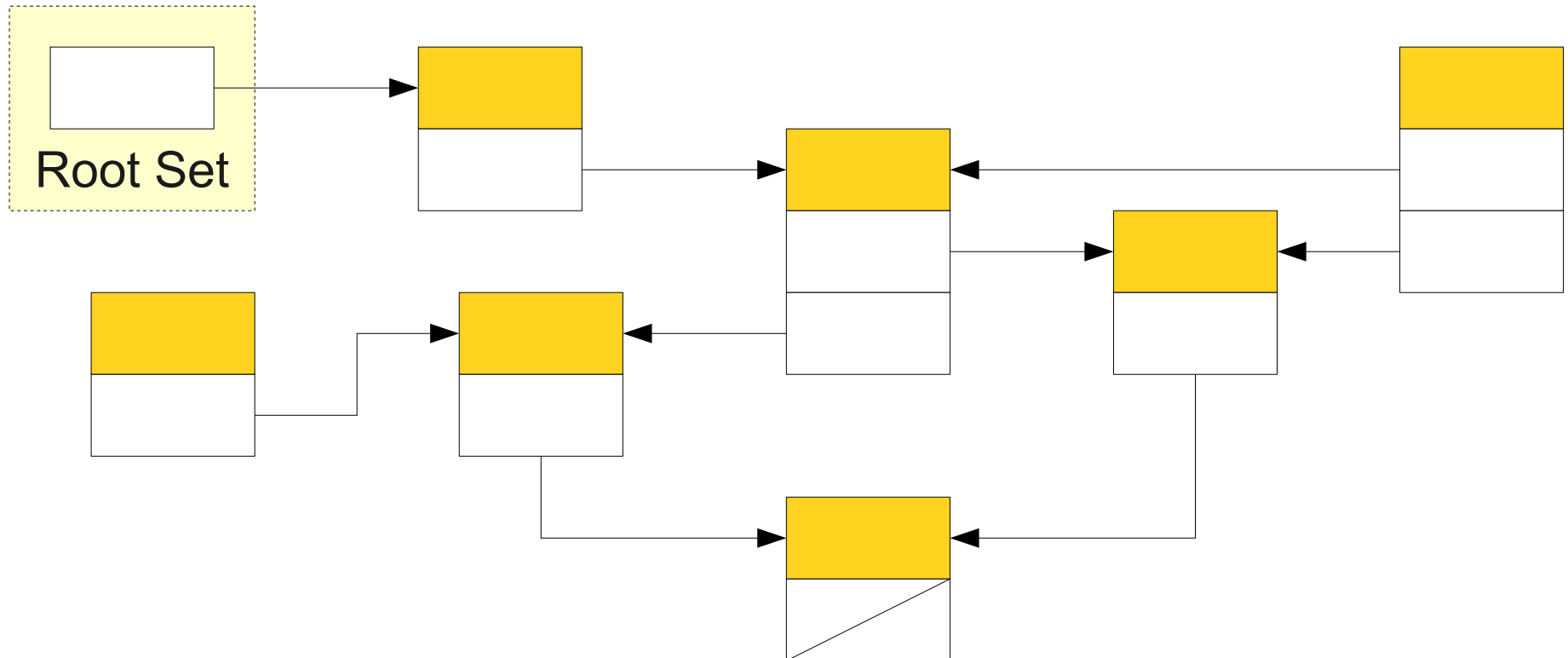
# Mark-and-Sweep In Action



# Mark-and-Sweep In Action

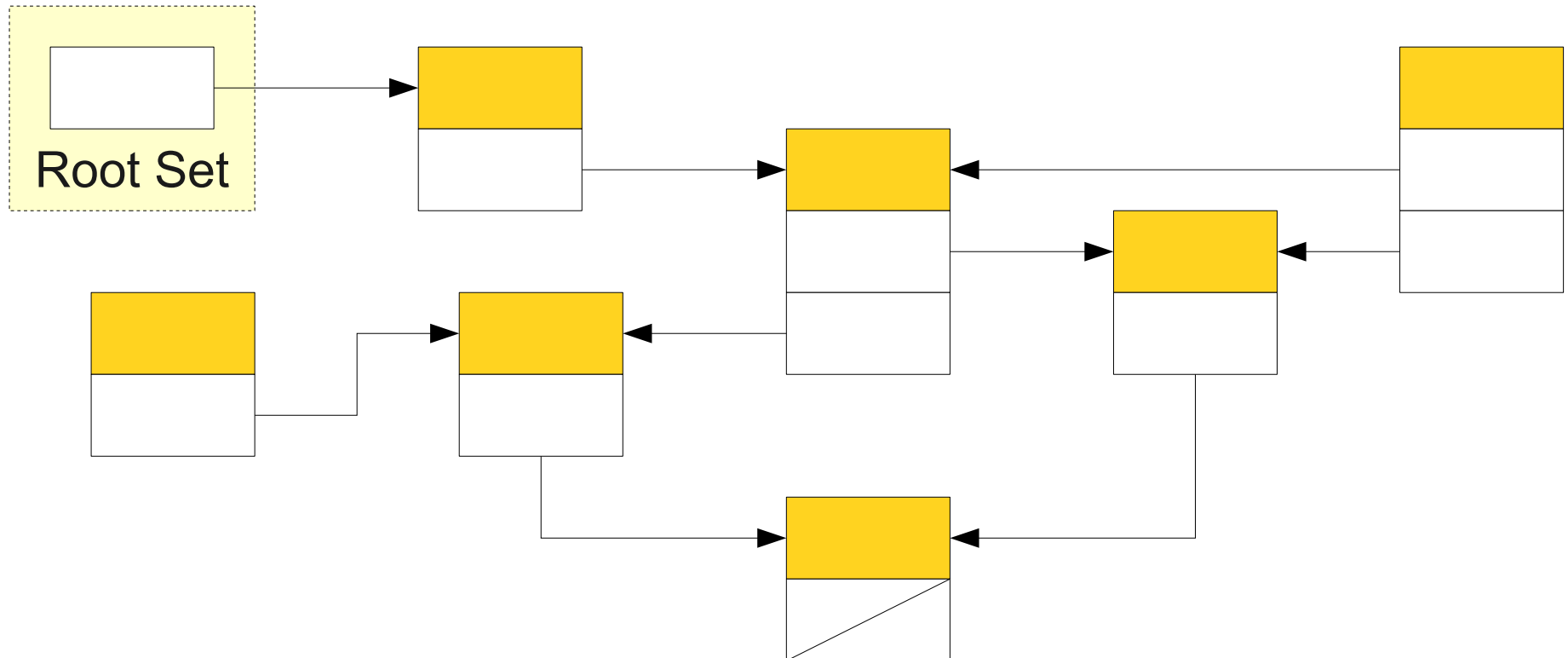


# Mark-and-Sweep In Action

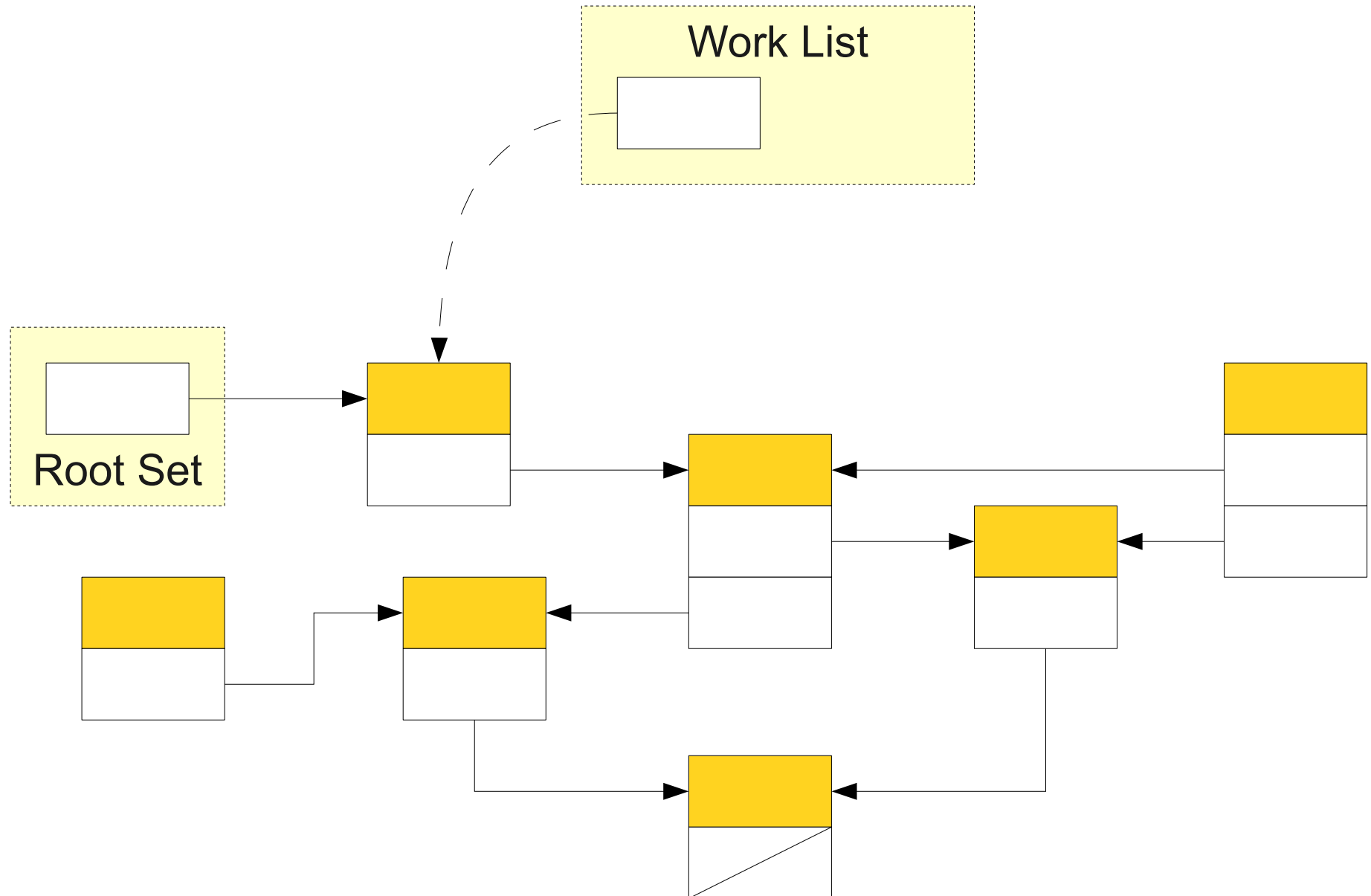


# Mark-and-Sweep In Action

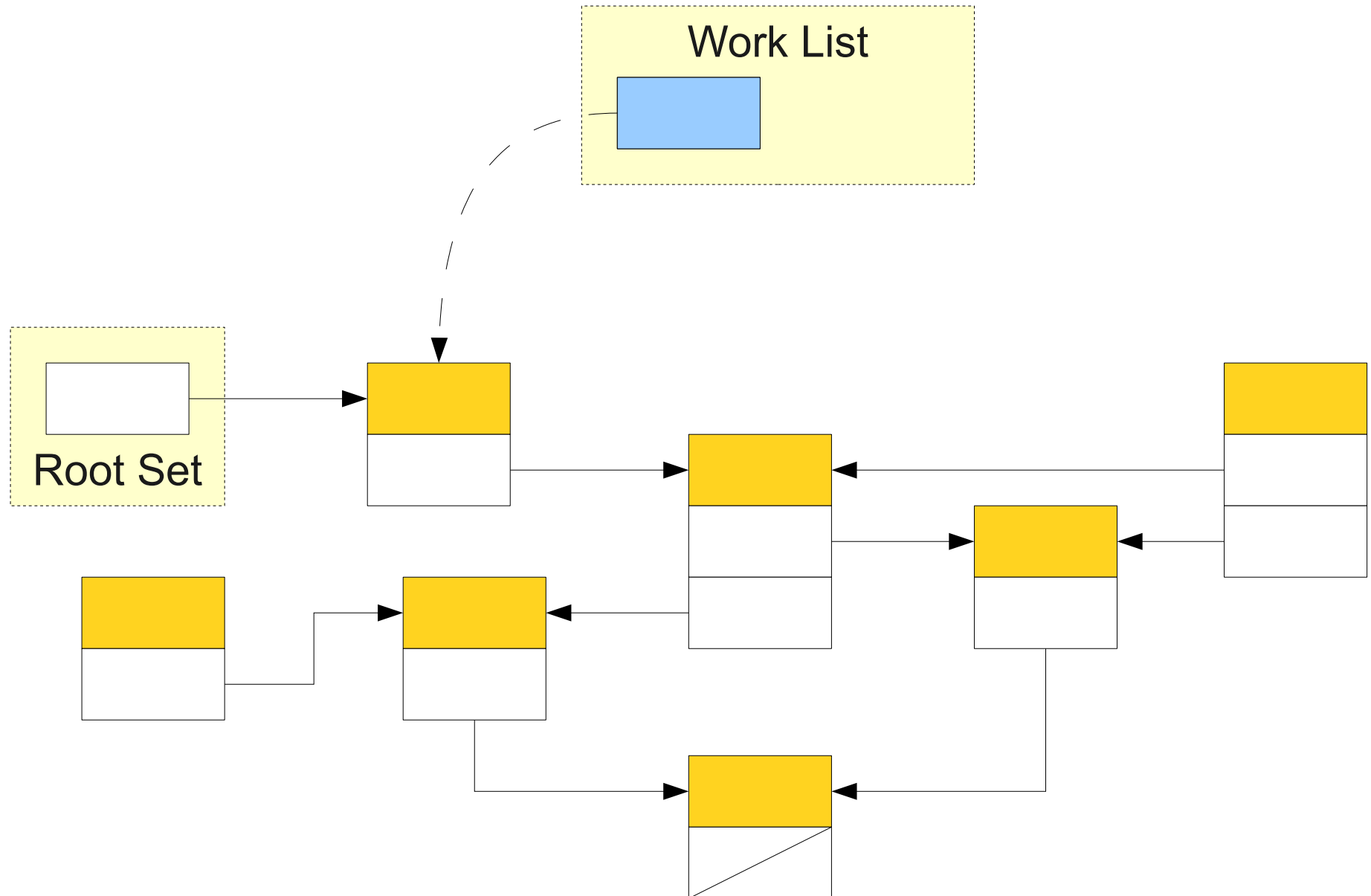
# Work List



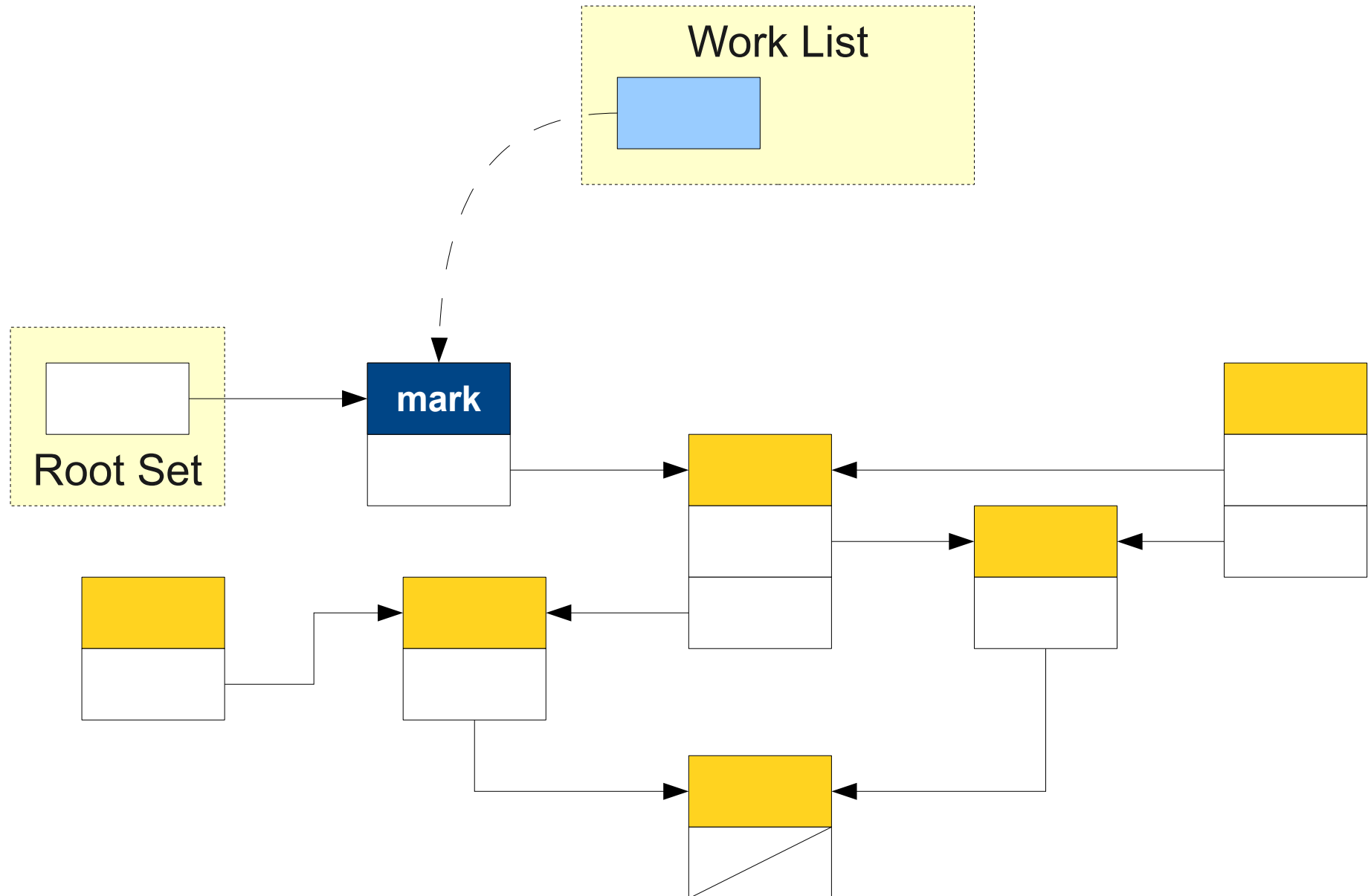
# Mark-and-Sweep In Action



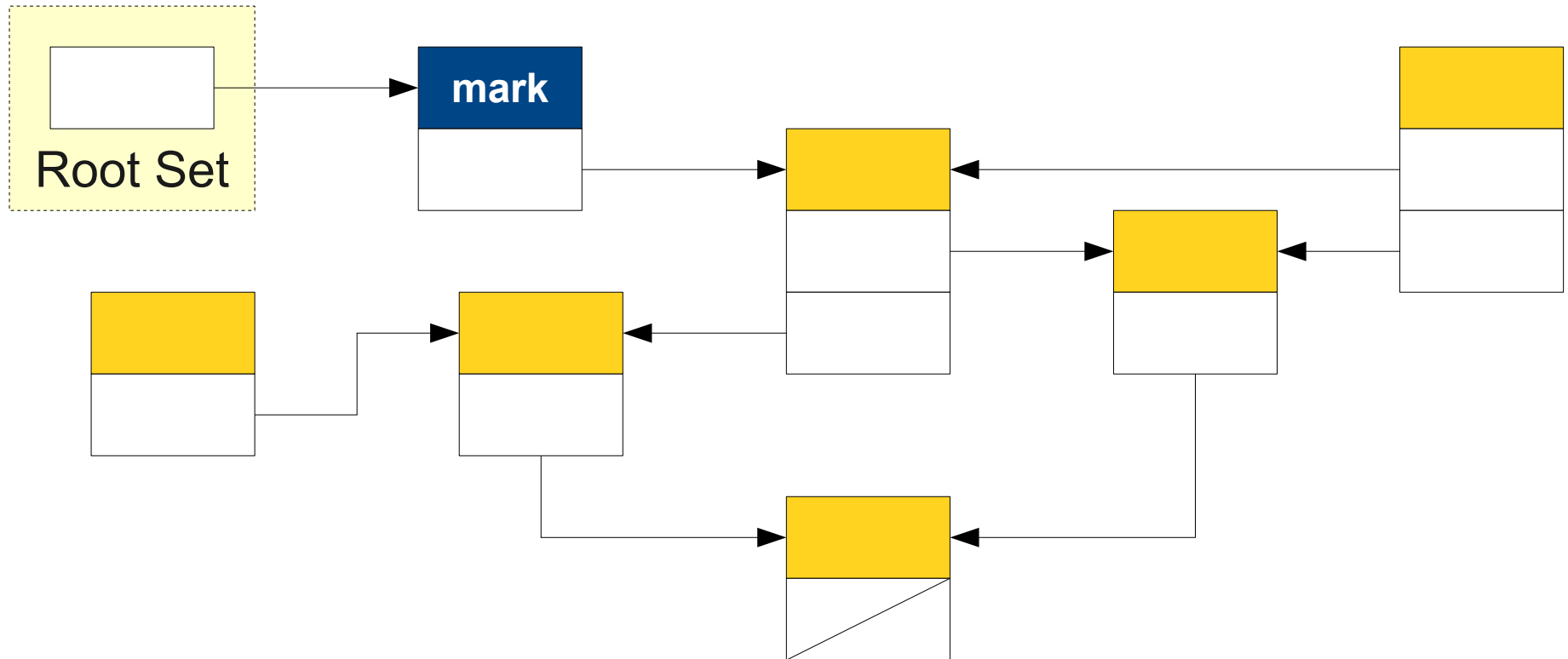
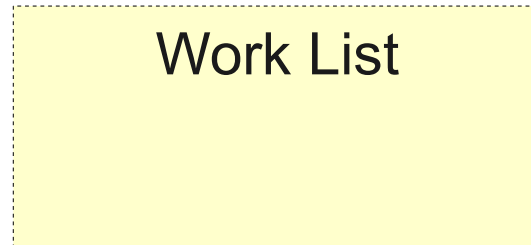
# Mark-and-Sweep In Action



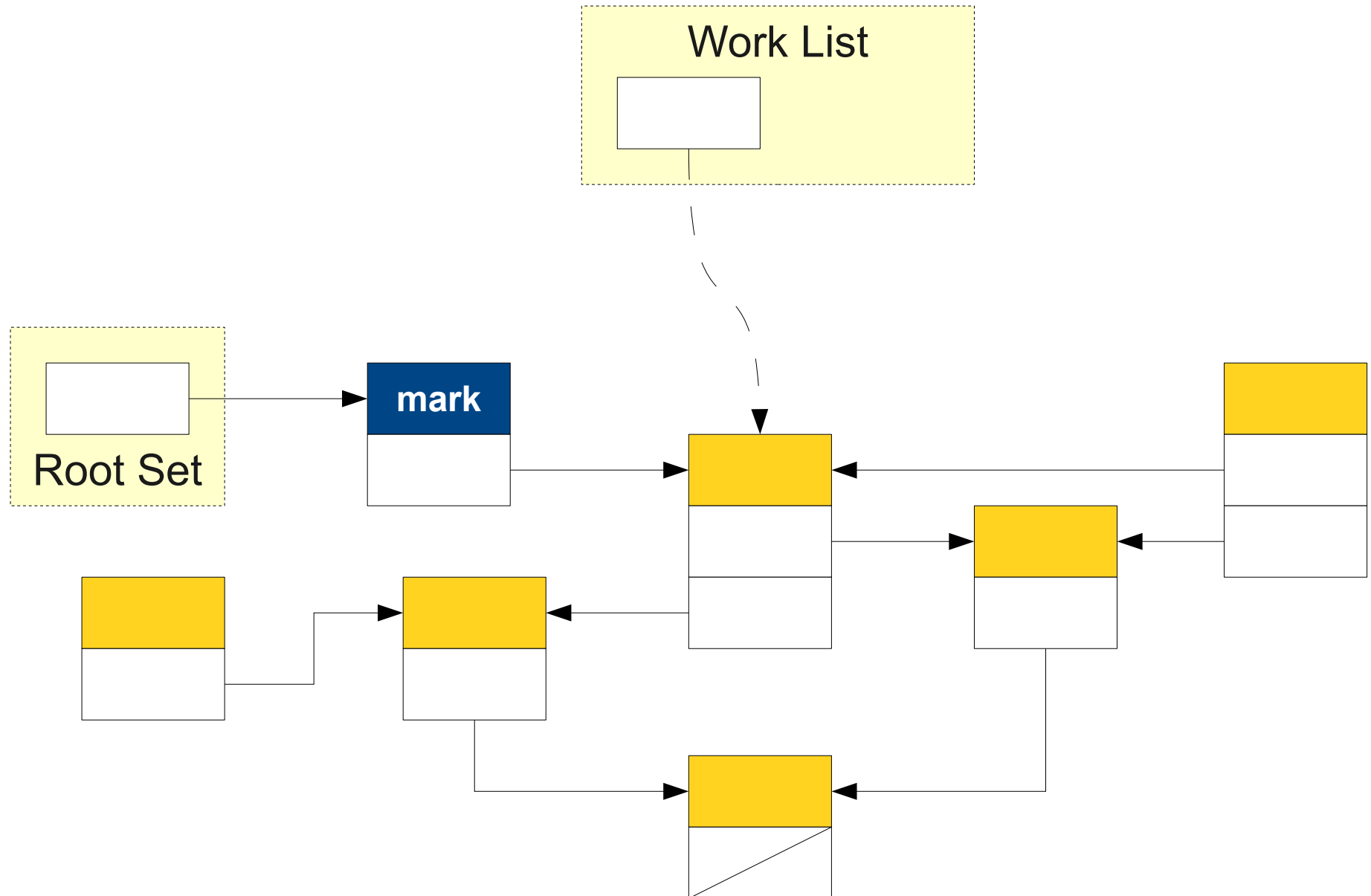
# Mark-and-Sweep In Action



# Mark-and-Sweep In Action

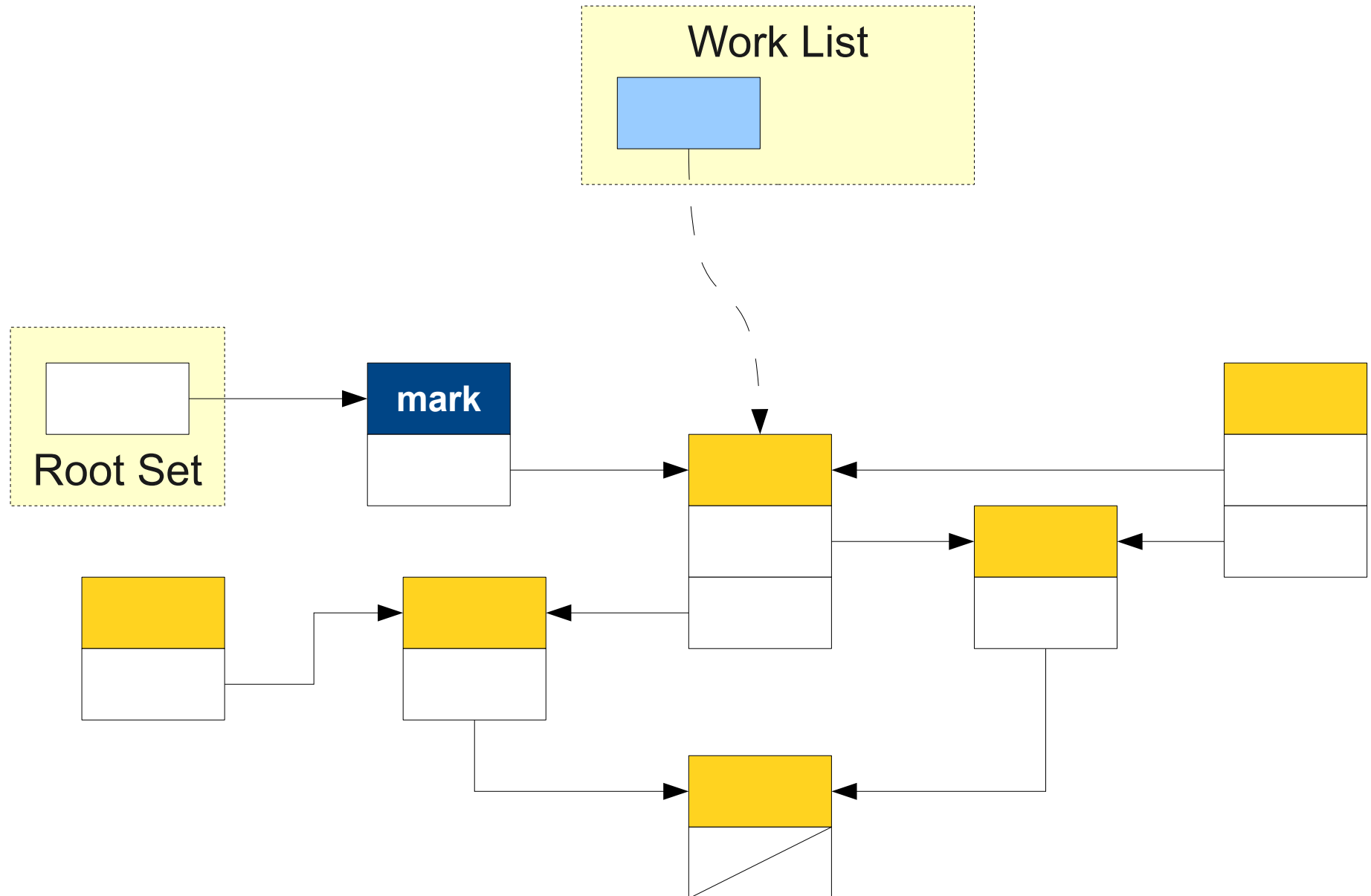


# Mark-and-Sweep In Action

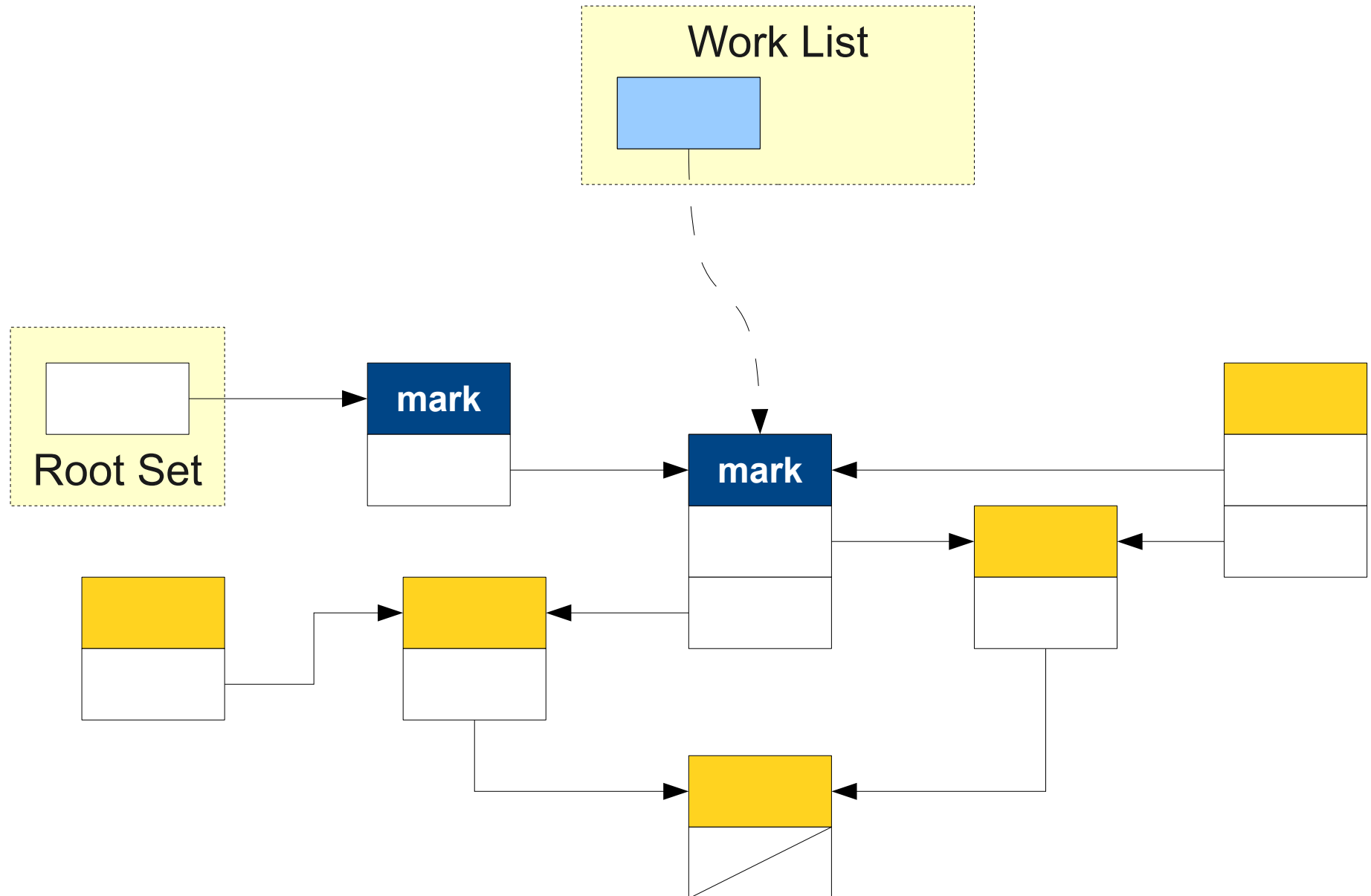




# Mark-and-Sweep In Action

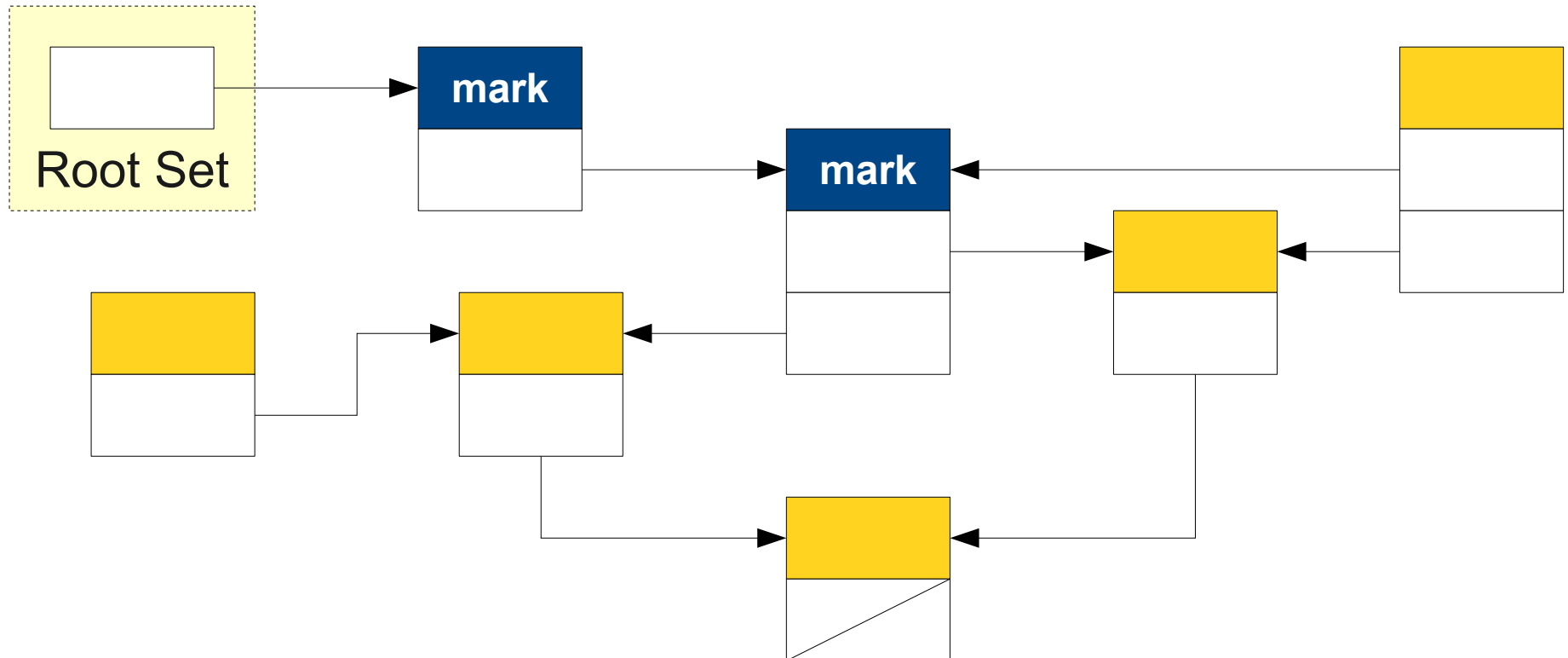


# Mark-and-Sweep In Action

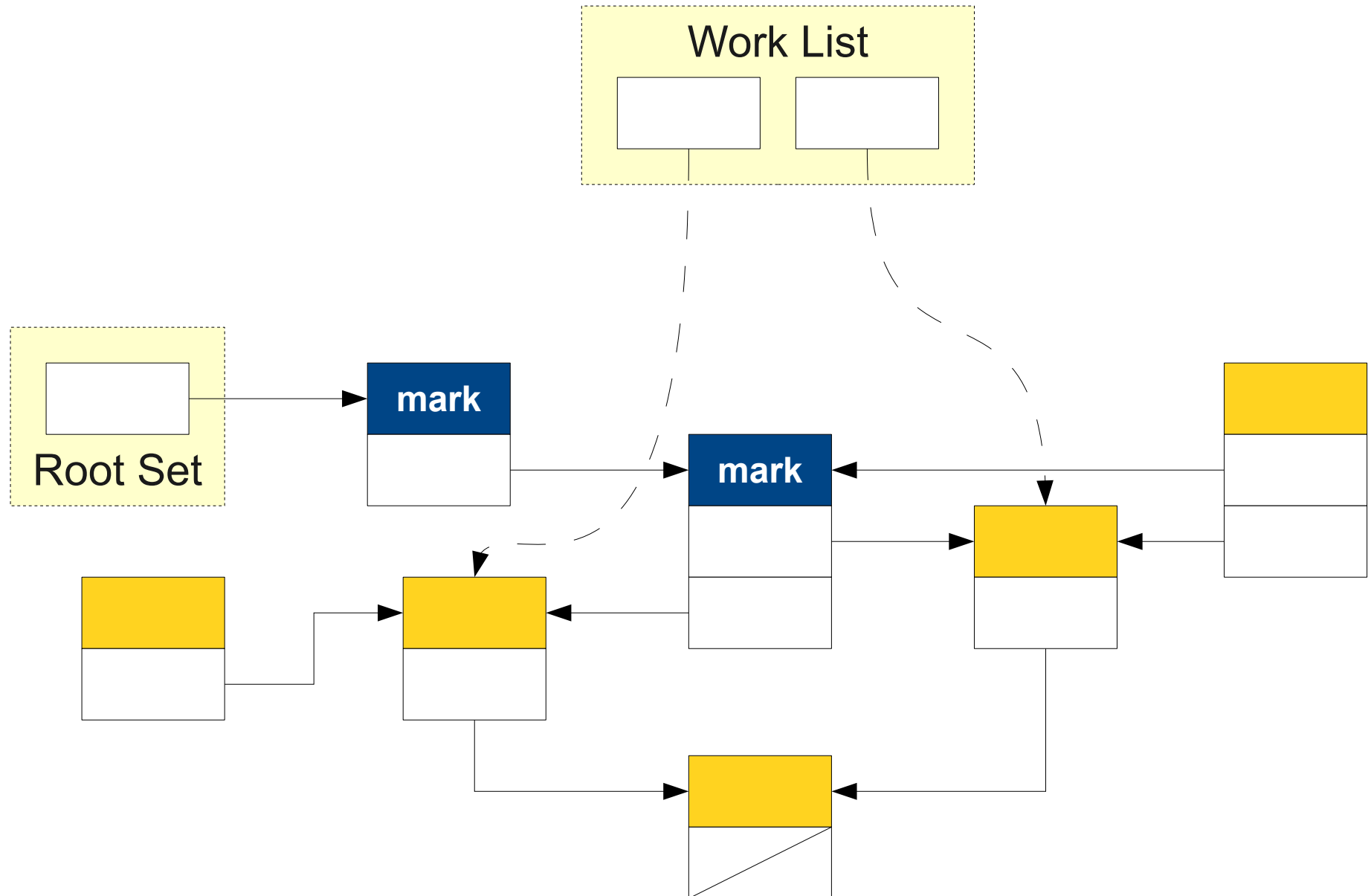


# Mark-and-Sweep In Action

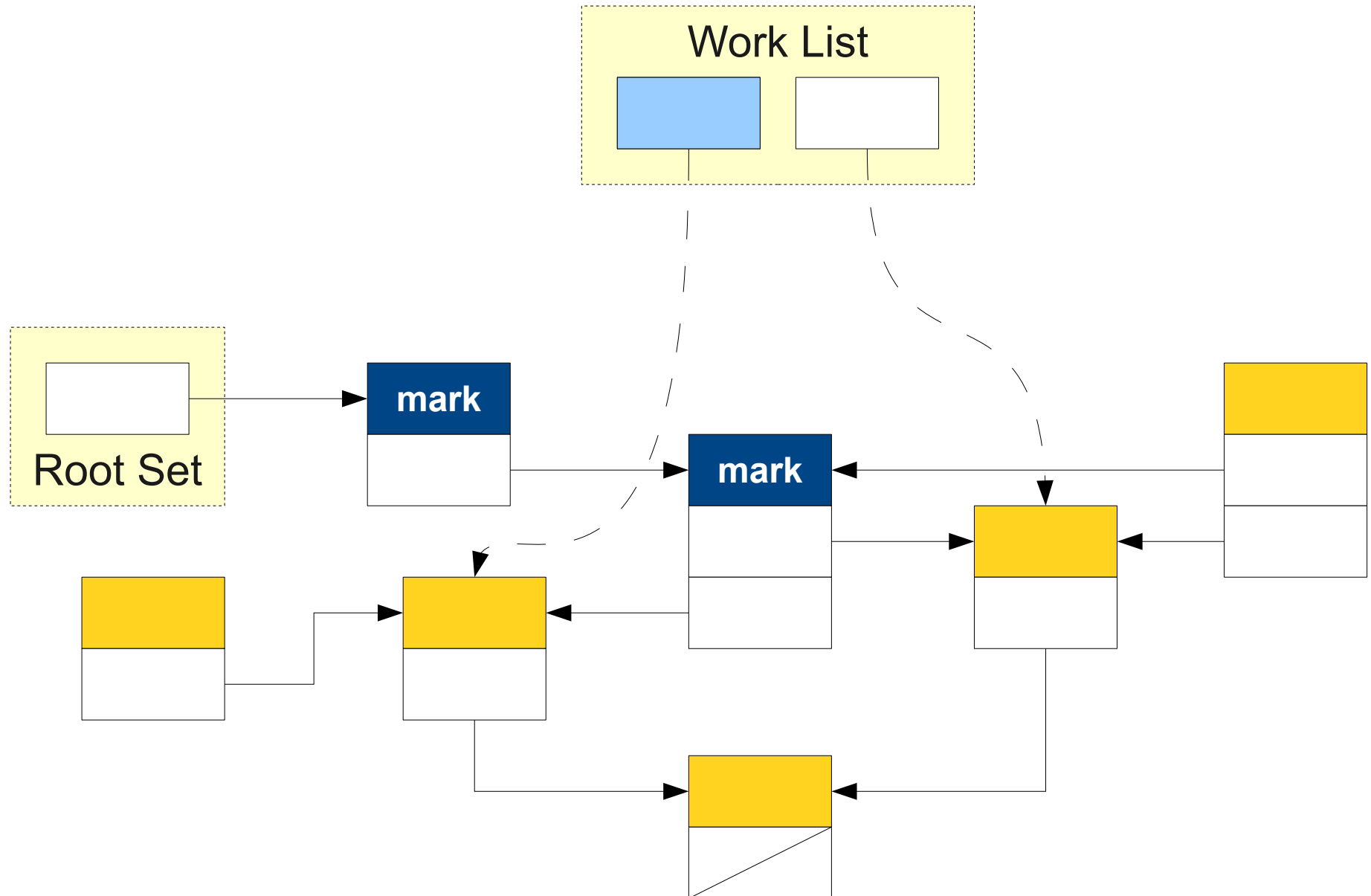
# Work List



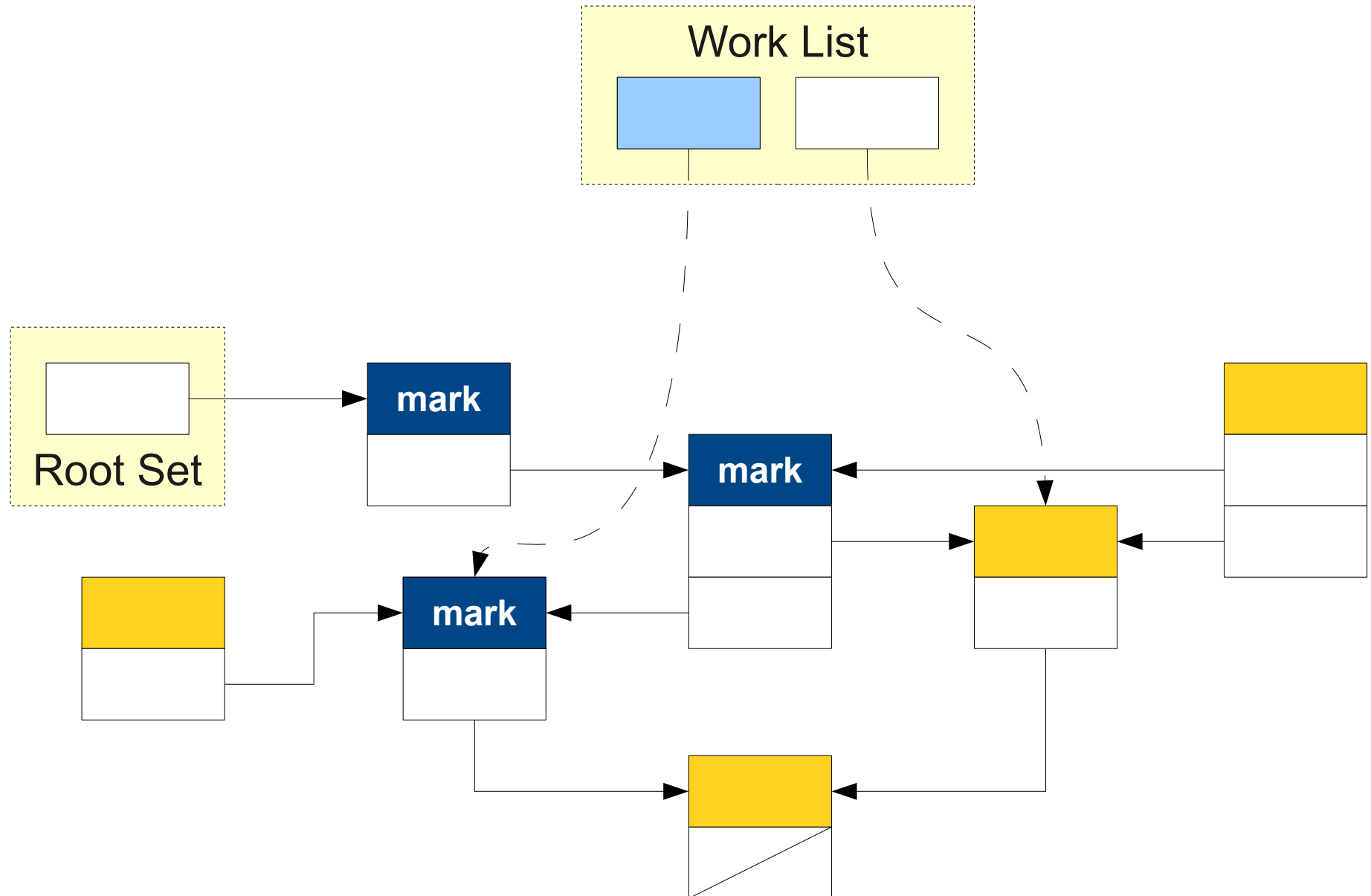
# Mark-and-Sweep In Action



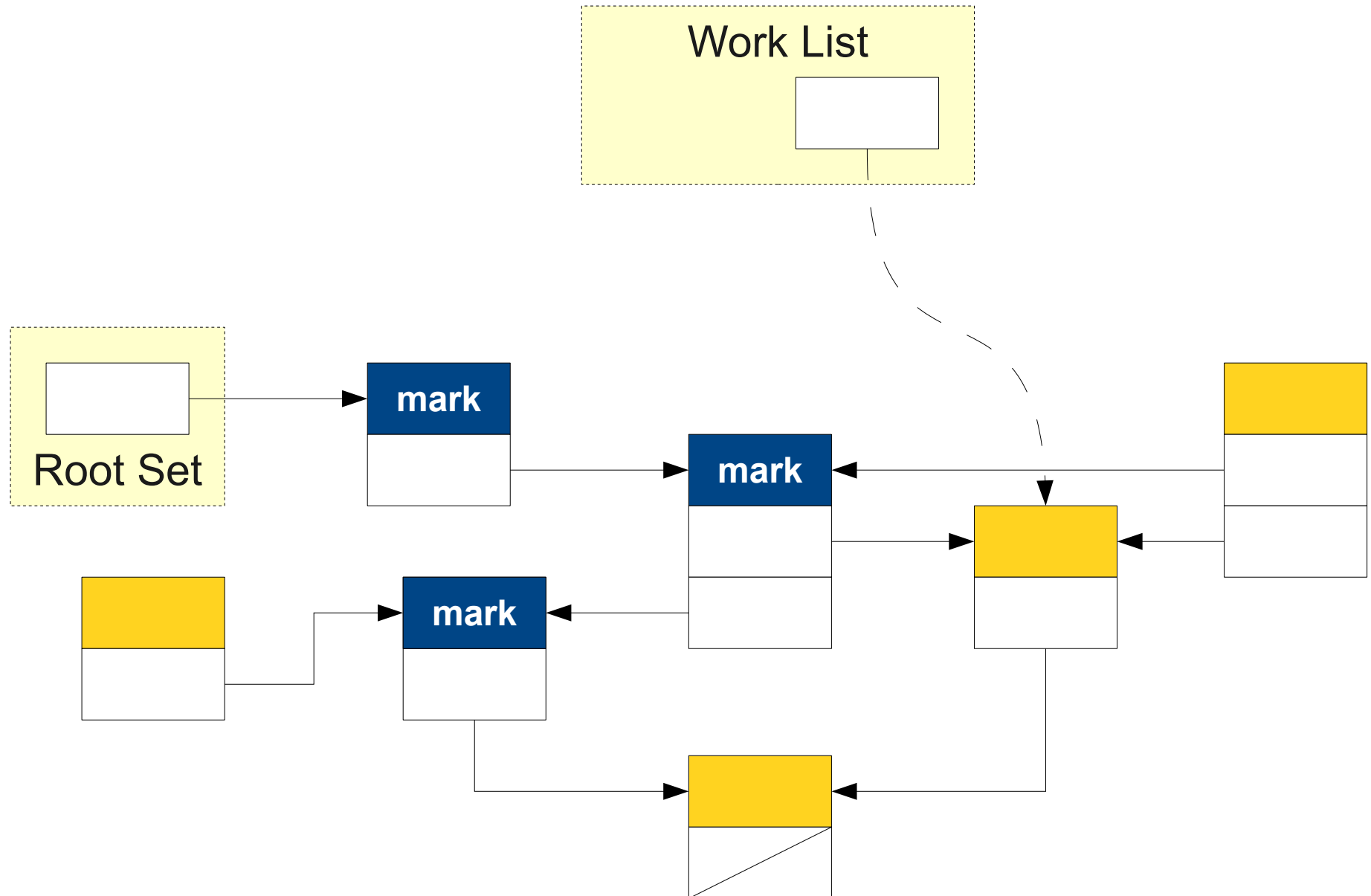
# Mark-and-Sweep In Action



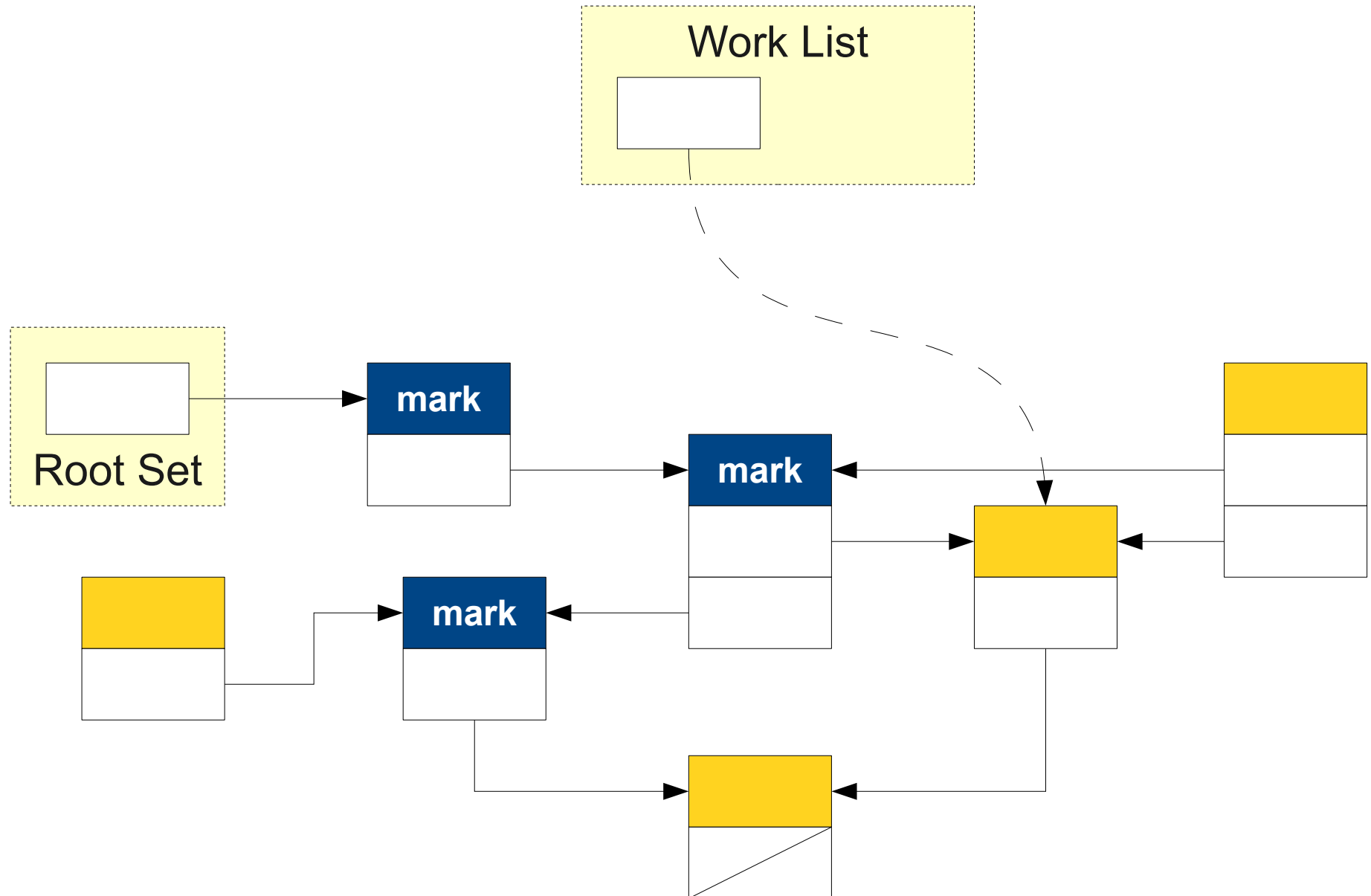
# Mark-and-Sweep In Action



# Mark-and-Sweep In Action

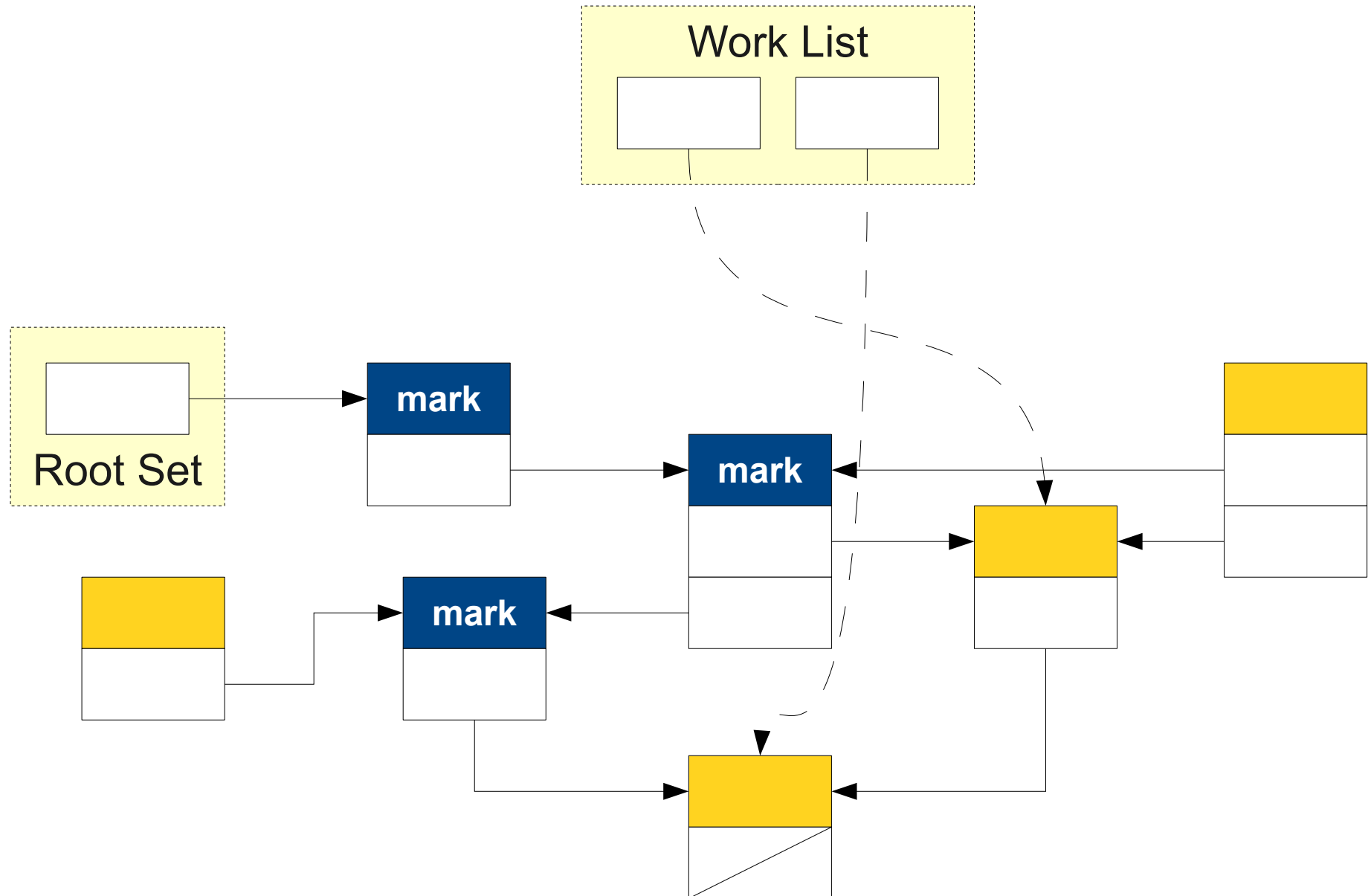


# Mark-and-Sweep In Action

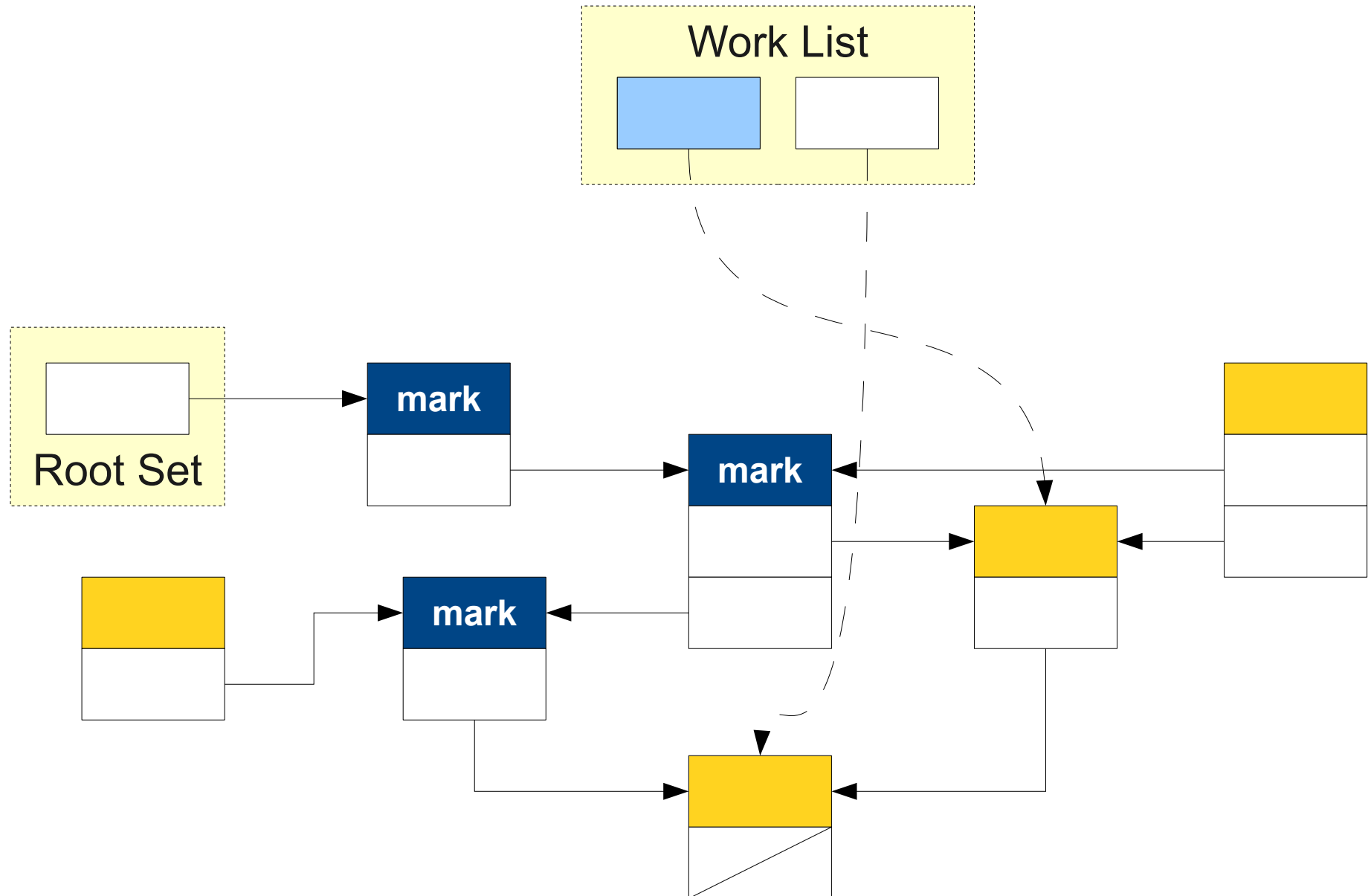




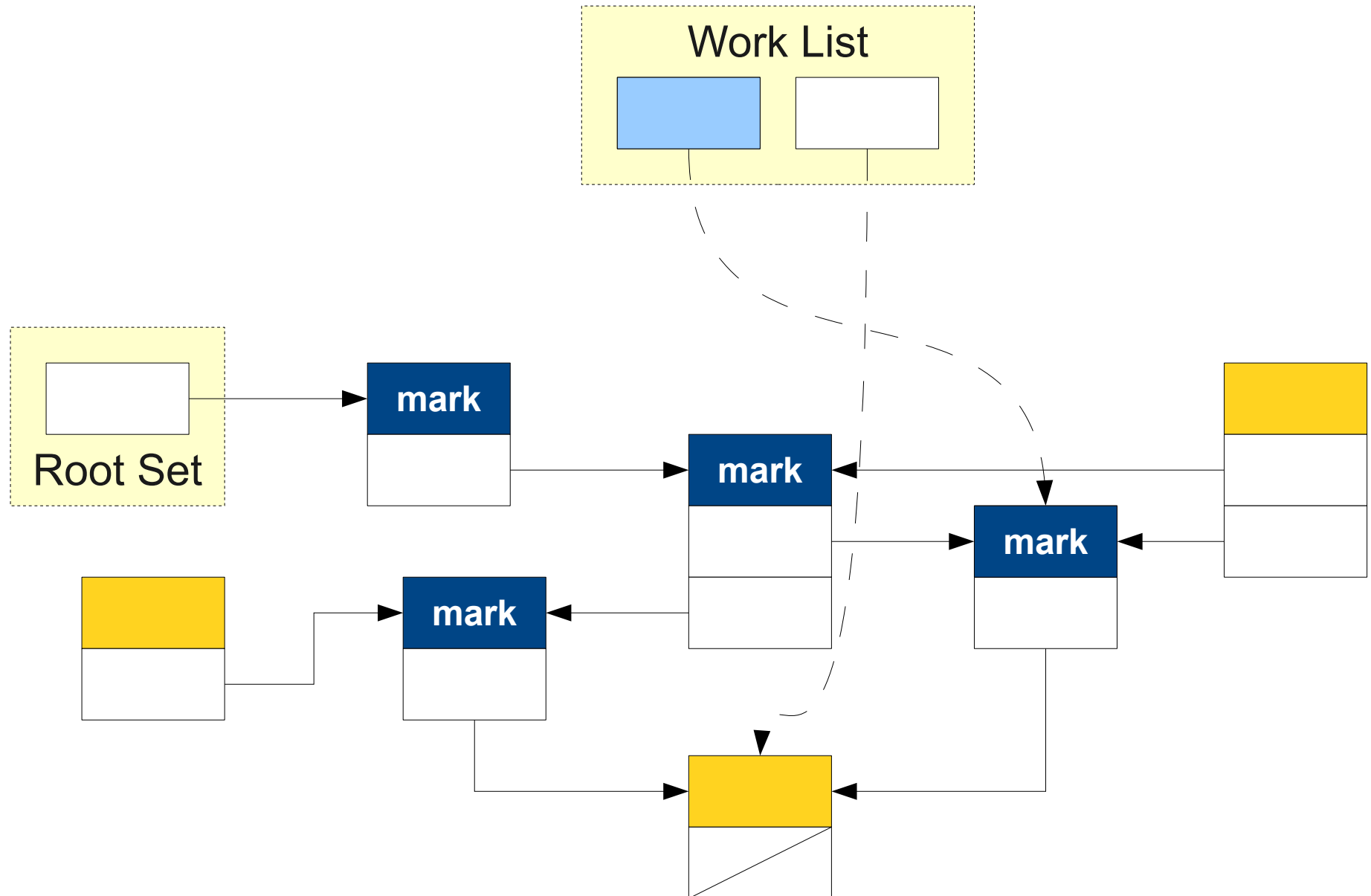
# Mark-and-Sweep In Action



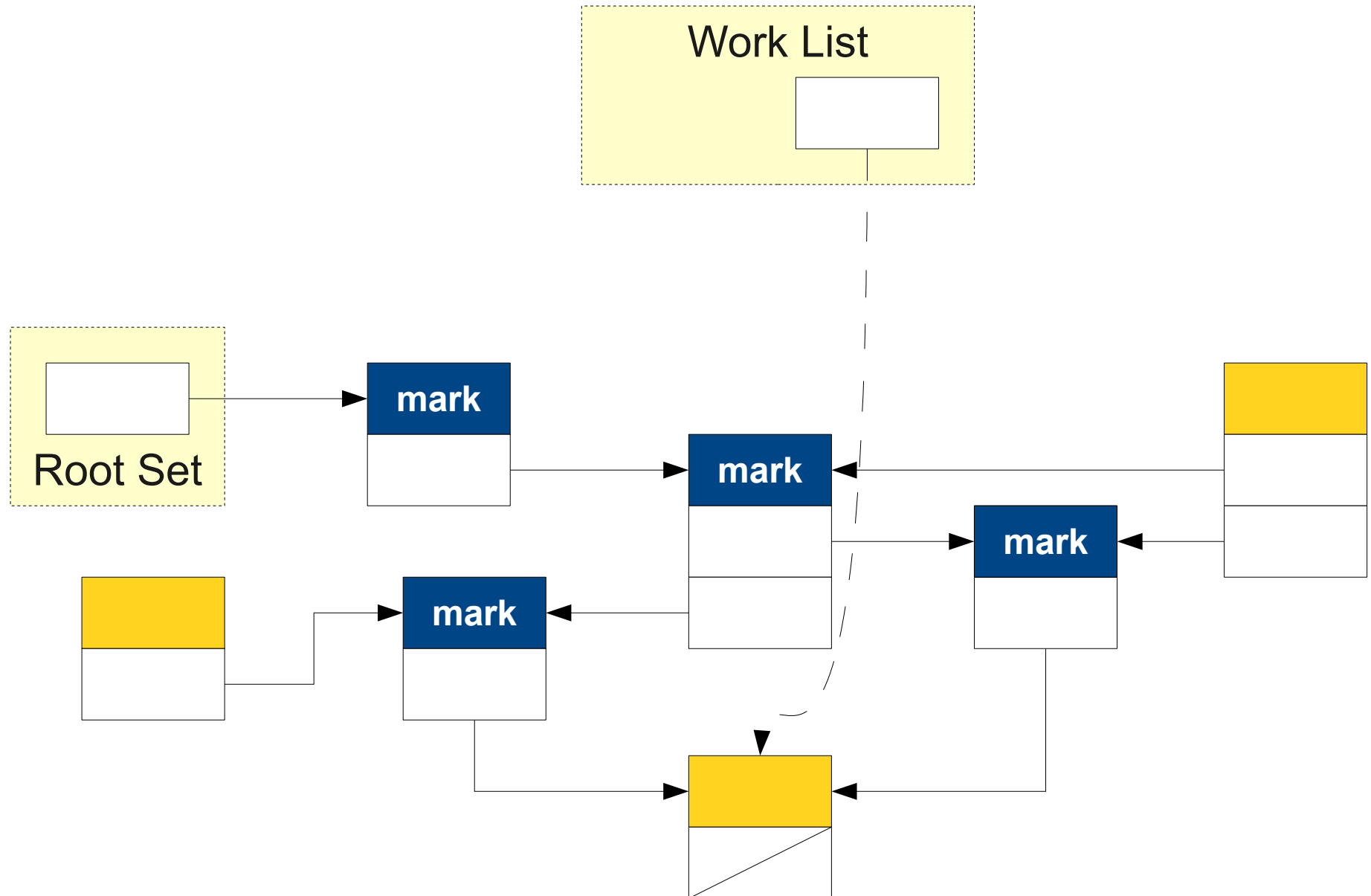
# Mark-and-Sweep In Action



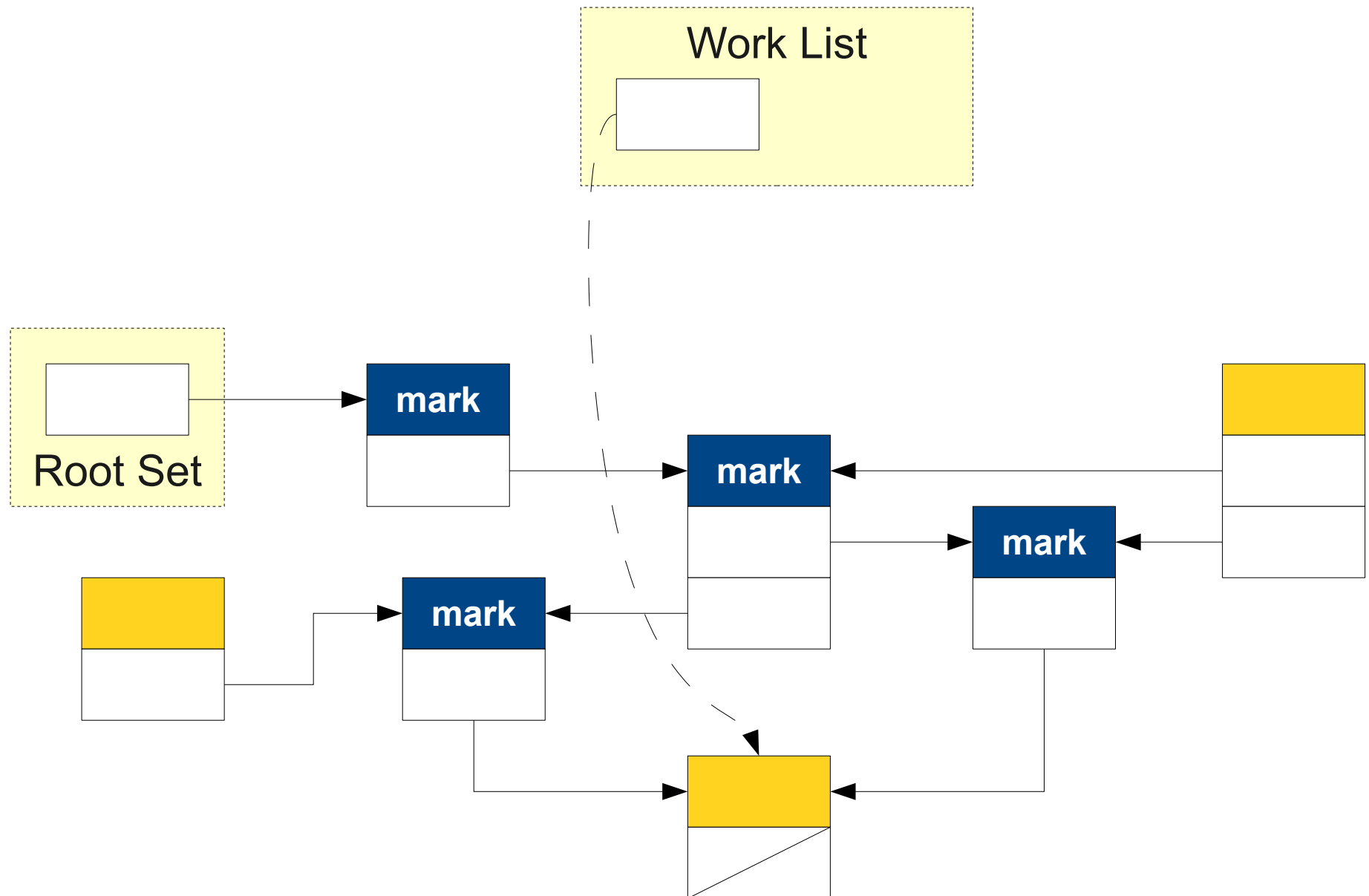
# Mark-and-Sweep In Action



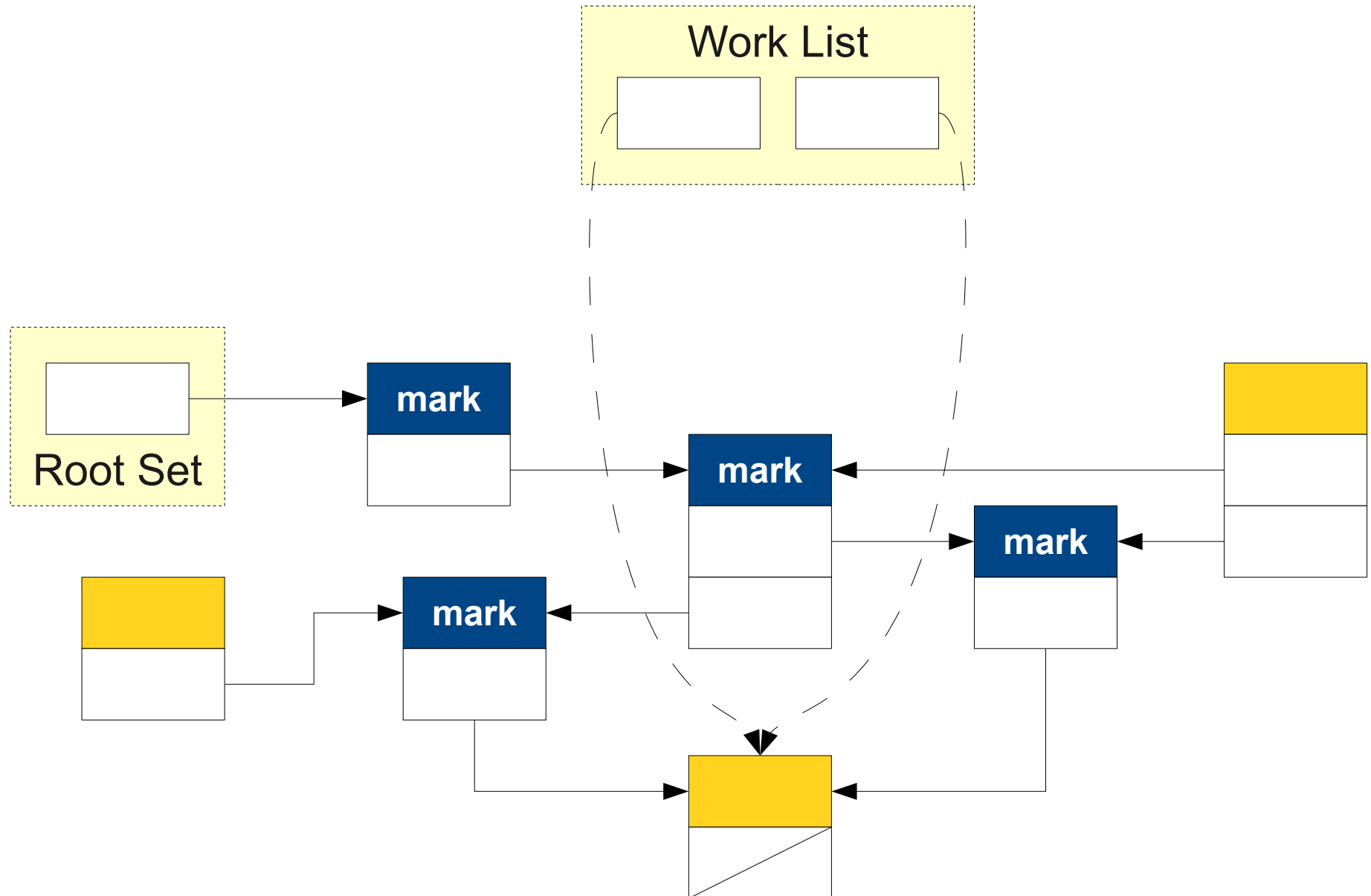
# Mark-and-Sweep In Action



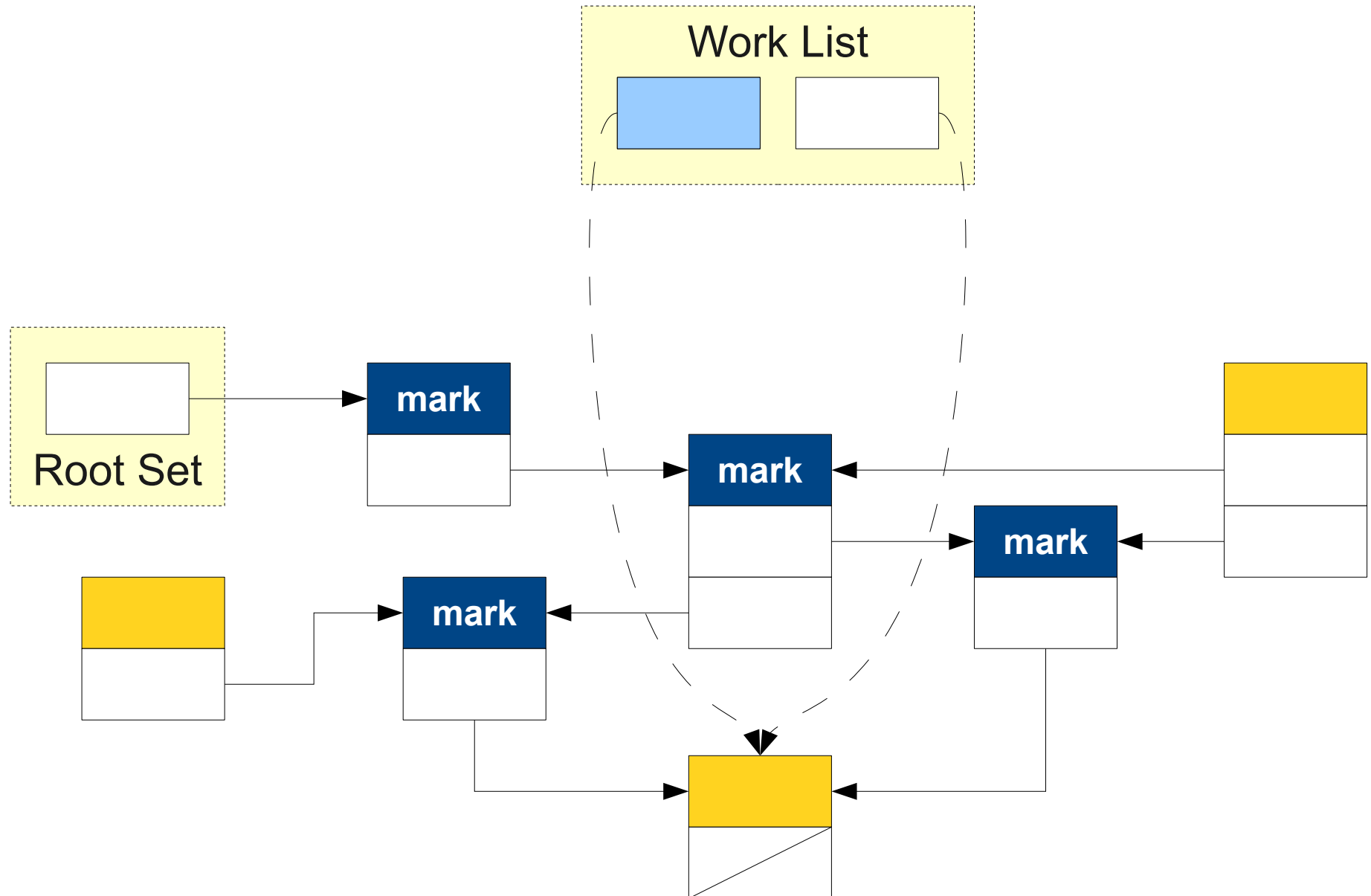
# Mark-and-Sweep In Action



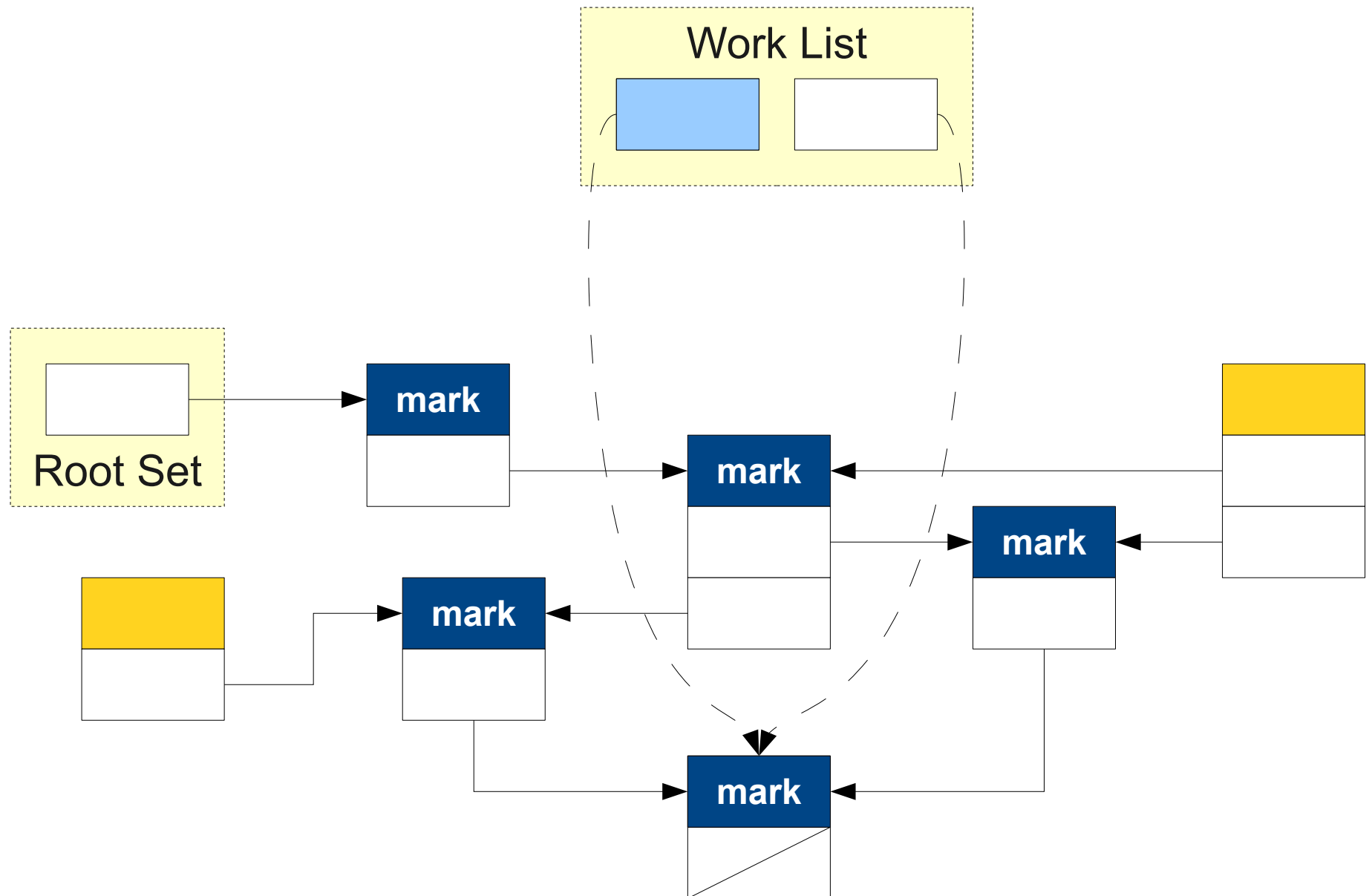
# Mark-and-Sweep In Action



# Mark-and-Sweep In Action

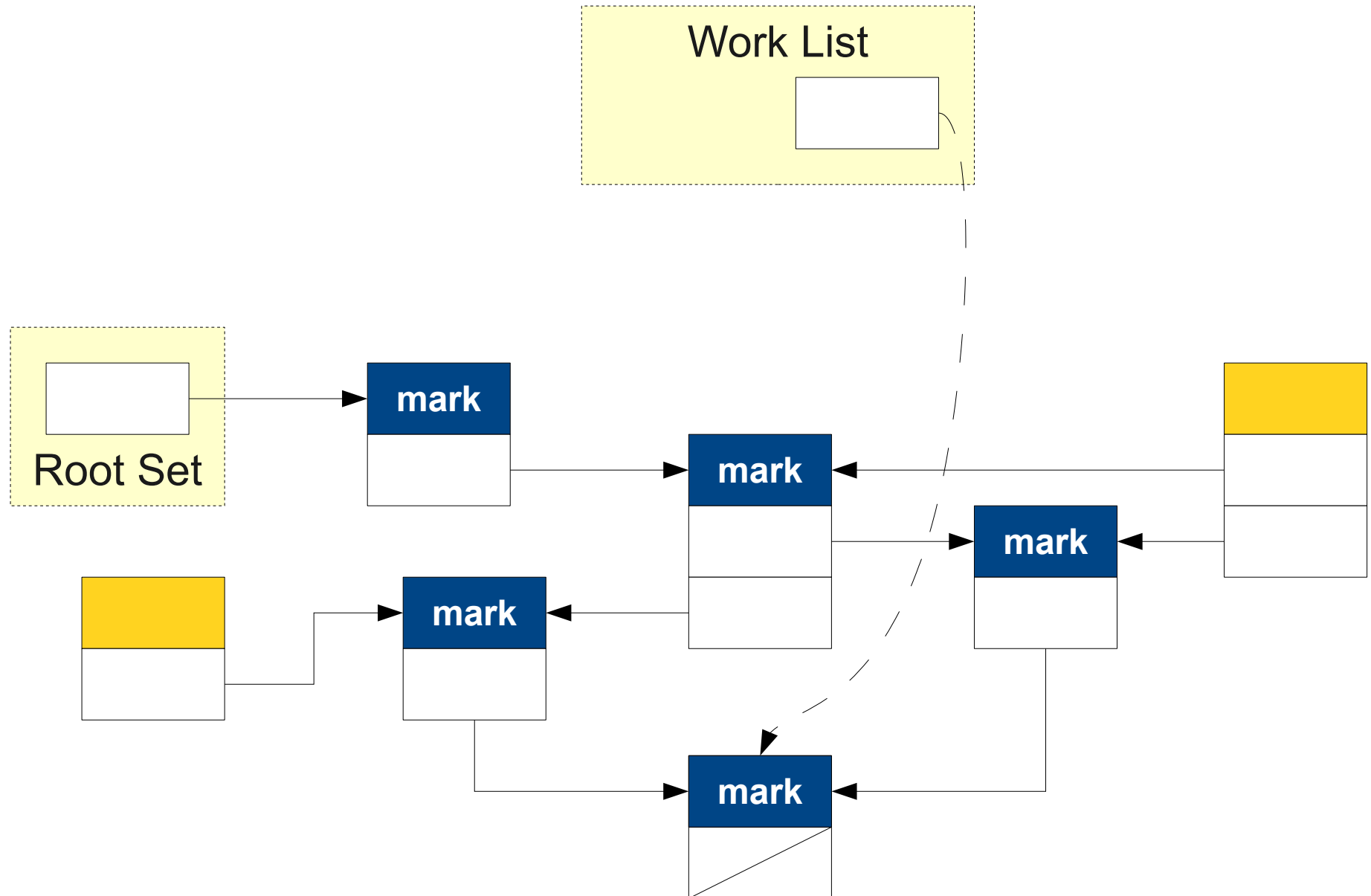


# Mark-and-Sweep In Action

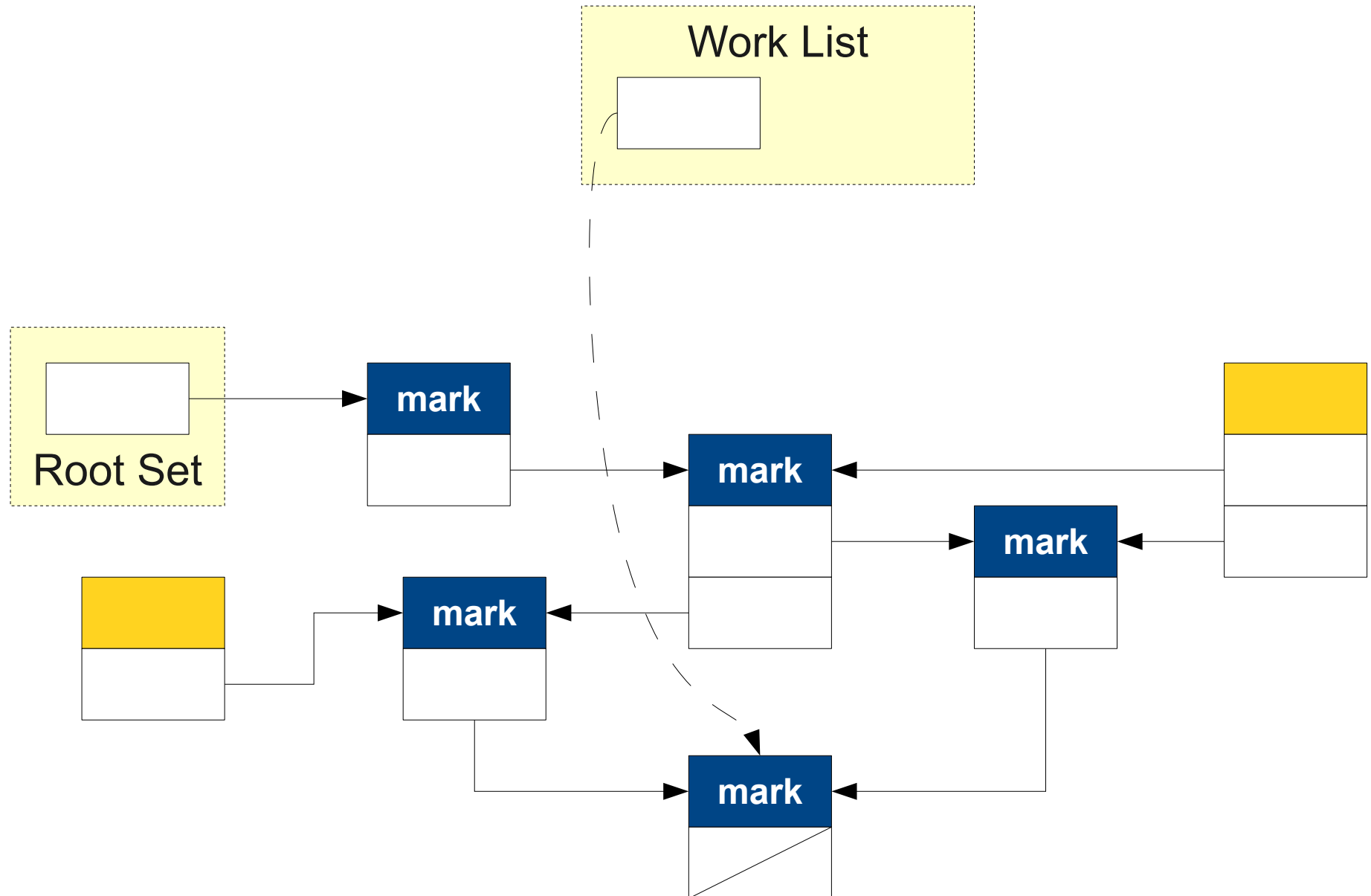




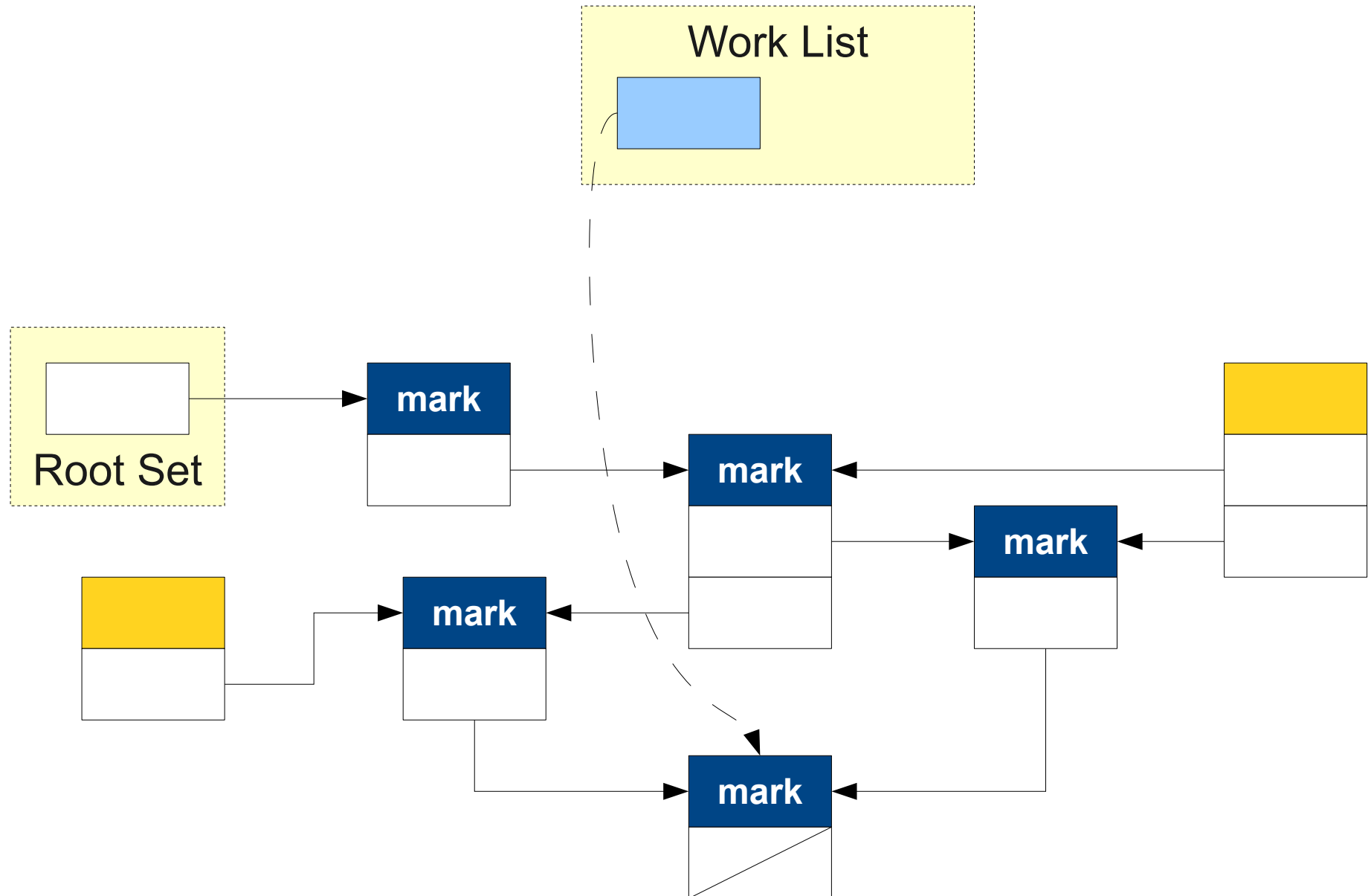
# Mark-and-Sweep In Action



# Mark-and-Sweep In Action

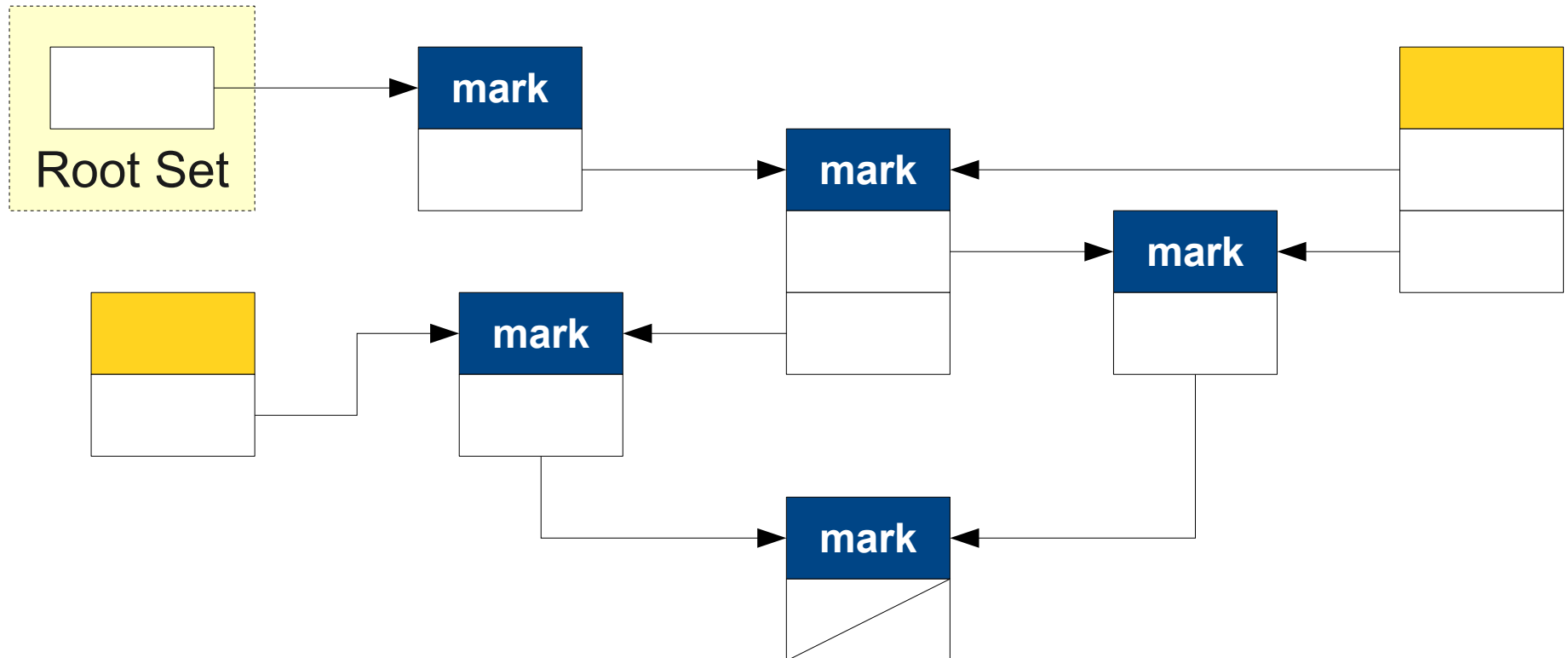


# Mark-and-Sweep In Action



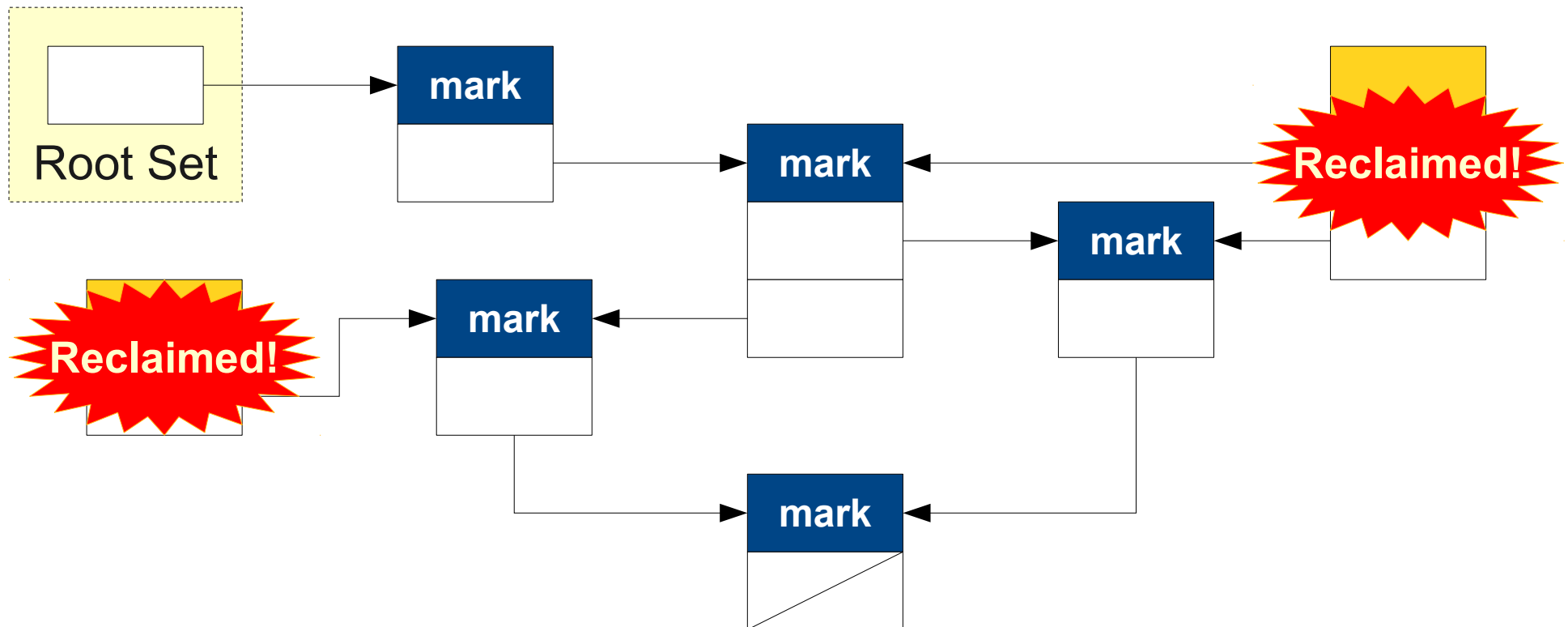
# Mark-and-Sweep In Action

## Work List



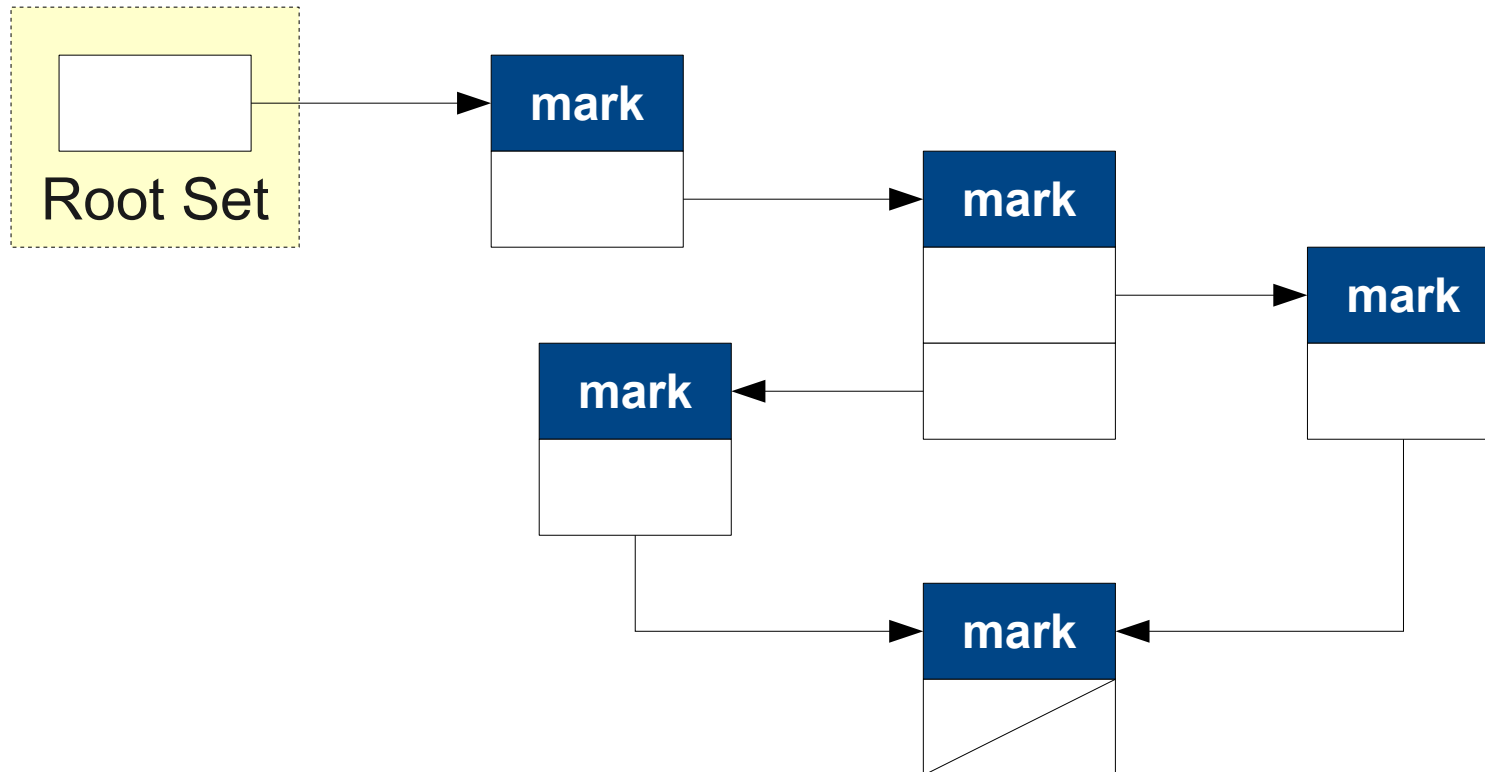
# Mark-and-Sweep In Action

Work List

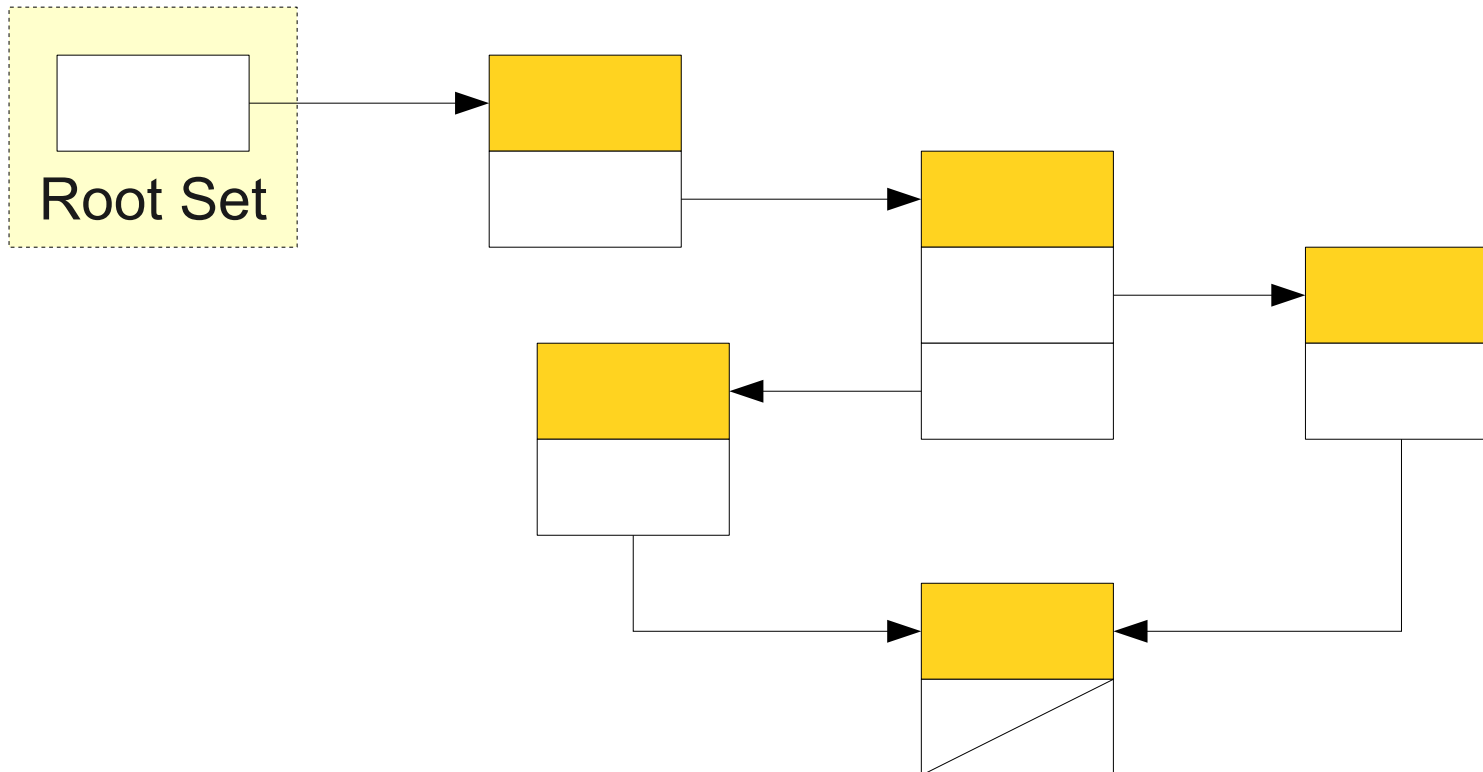
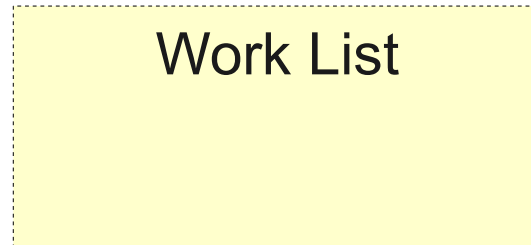


# Mark-and-Sweep In Action

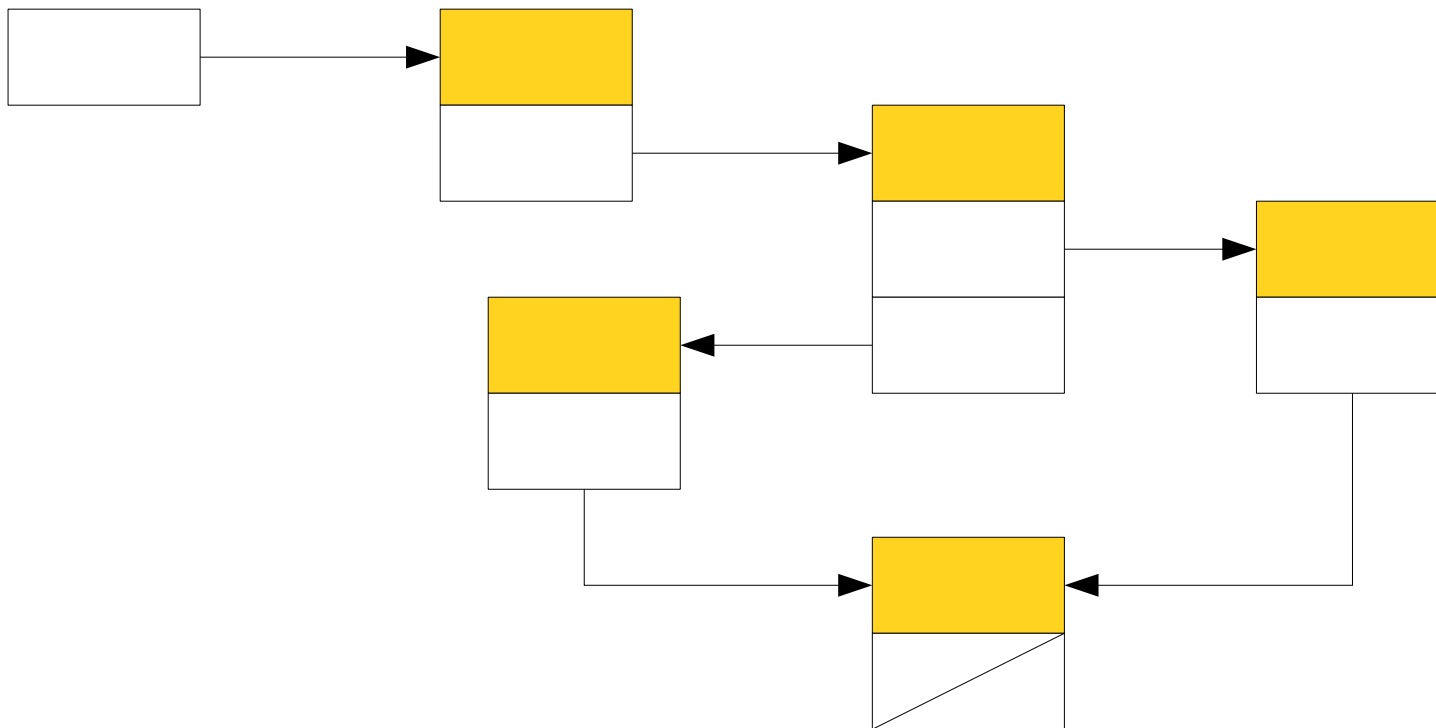
## Work List



# Mark-and-Sweep In Action



# Mark-and-Sweep In Action





# Implementing Mark-and-Sweep

- The mark-and-sweep algorithm, as described, has two serious problems.
- **Runtime proportional to number of allocated objects.**
  - 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.
  - Can't preallocate this space!

# The Key Idea

- During a mark-and-sweep collection, every allocated block must be in exactly one of four states:
  - **Marked**: This object is known to be reachable.
  - **Enqueued**: This object is in the worklist.
  - **Unknown**: This object has not yet been seen.
  - **Deallocated**: This object has already been freed.
- Augment every allocated block with two bits to encode which of these four states the object is in.
- Maintain doubly-linked lists of all the objects in each of these states.

# Baker's Algorithm

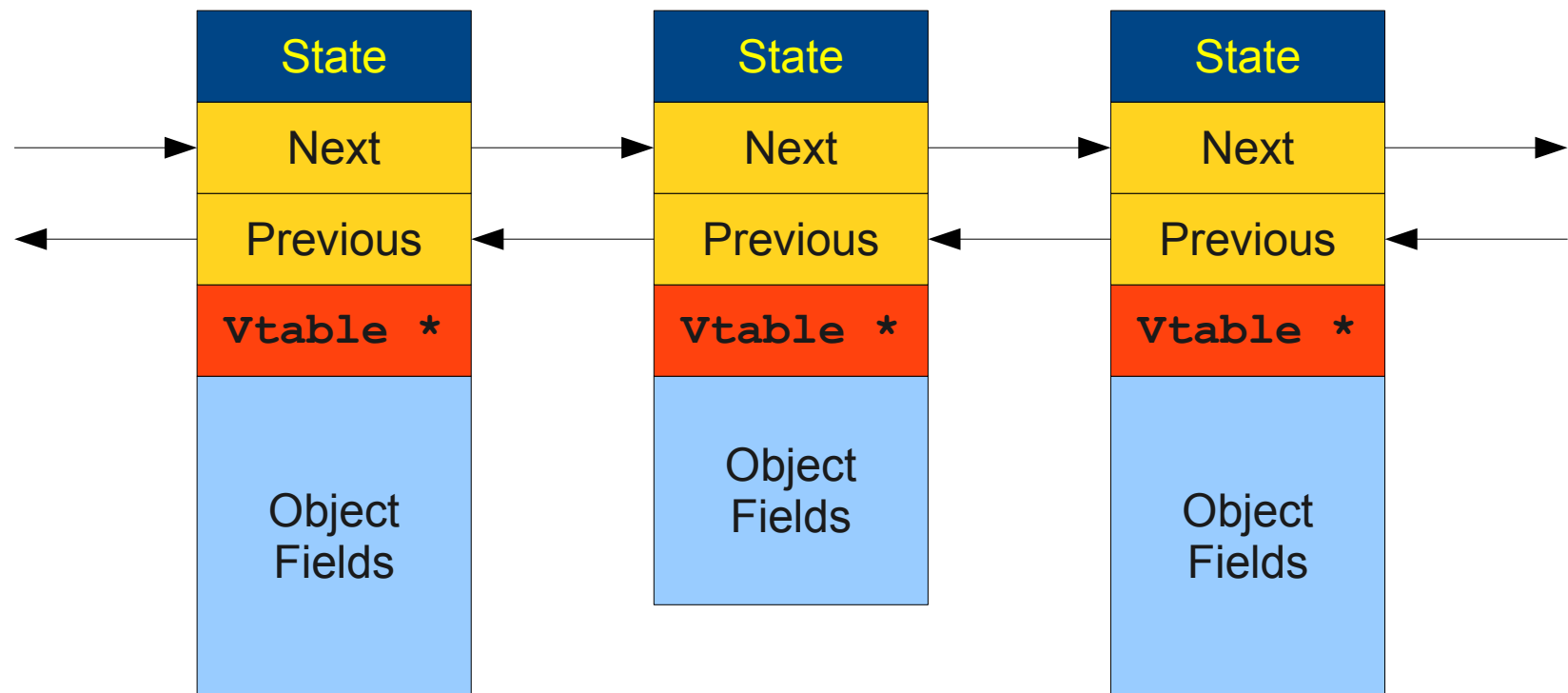
- Move all of the root set to the **enqueued** list.
- While the **enqueued** list is not empty:
  - Move the first object from the **enqueued** list to the **marked** list.
  - For each **unknown** object referenced, add it to the **enqueued** list.
- At this point, everything reachable is in **marked** and everything unreachable is in **unknown**.
- Concatenate the **unknown** and **deallocated** lists
  - Deallocates all garbage in  $O(1)$ .
- Move everything from the **marked** list to the **unknown** list.
  - Can be done in  $O(1)$ .
  - Indicates objects again must be proven reachable on next scan.

# One Last Detail

- But wait – if we're already out of memory, how do we build these linked lists?

# One Last Detail

- But wait – if we're already out of memory, how do we build these linked lists?
- **Idea:** Since every object can only be in one linked list, embed the next and previous pointers into each allocated block.



# Analysis of Mark-and-Sweep

- Advantages:
  - Precisely finds exactly the reachable objects.
  - Using Baker's algorithm, runs in time proportional to the number of reachable objects.
- Disadvantages:
  - Stop-the-world approach may introduce huge pause times.
  - Linked list / state information in each allocated block uses lots of memory per object.

# Stop-and-Copy

# Improving Performance

- There are many ways to improve a program's performance, some of which can be improved by a good garbage collector:
- **Increasing locality.**
  - Memory caches are often designed to hold adjacent memory locations.
  - Placing objects consecutively in memory can improve performance by reducing cache misses.
- **Increasing allocation speed.**
  - Many languages (Java, Haskell, Python, etc.) allocate objects frequently.
  - Speeding up object allocation can speed up program execution.



# Increasing Locality

- **Idea:** When doing garbage collection, move all objects in memory so that they are adjacent to one another.
  - This is called **compaction**.
- Ideally, move objects that reference one another into adjacent memory locations.
- Garbage collector must update all pointers in all objects to refer to the new object locations.
- Could you do this in Java?
- Could you do this in C++?

# Increasing Allocation Speed

- Typically implementations of `malloc` and `free` use **free lists**, linked lists of free memory blocks.
- Allocating an object requires following these pointers until a suitable object is found.
  - Usually fast, but at least 10 assembly instructions.
- Contrast with stack allocation – just one assembly instruction!
- Can we somehow get the performance speed of the stack for dynamic allocation?

# The Stop-and-Copy Collector

# The Stop-and-Copy Collector



# The Stop-and-Copy Collector

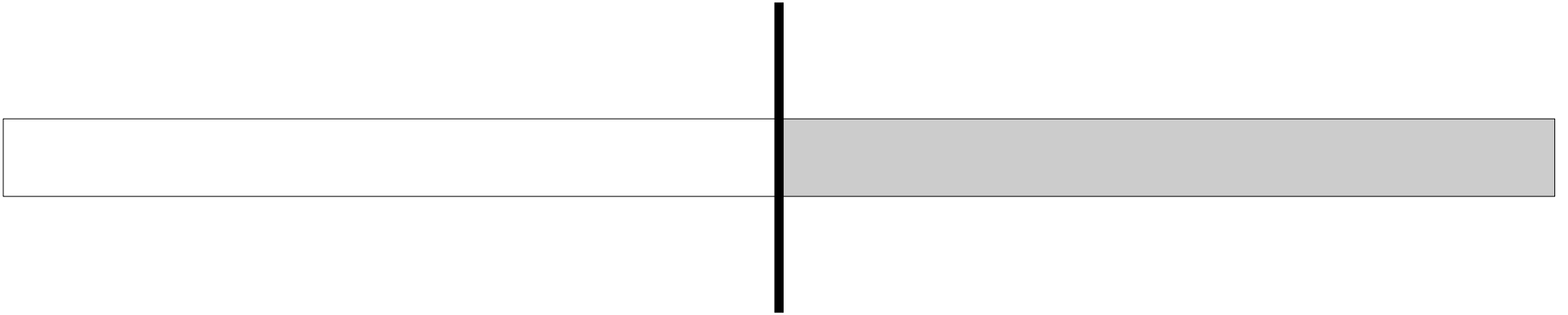


All of memory

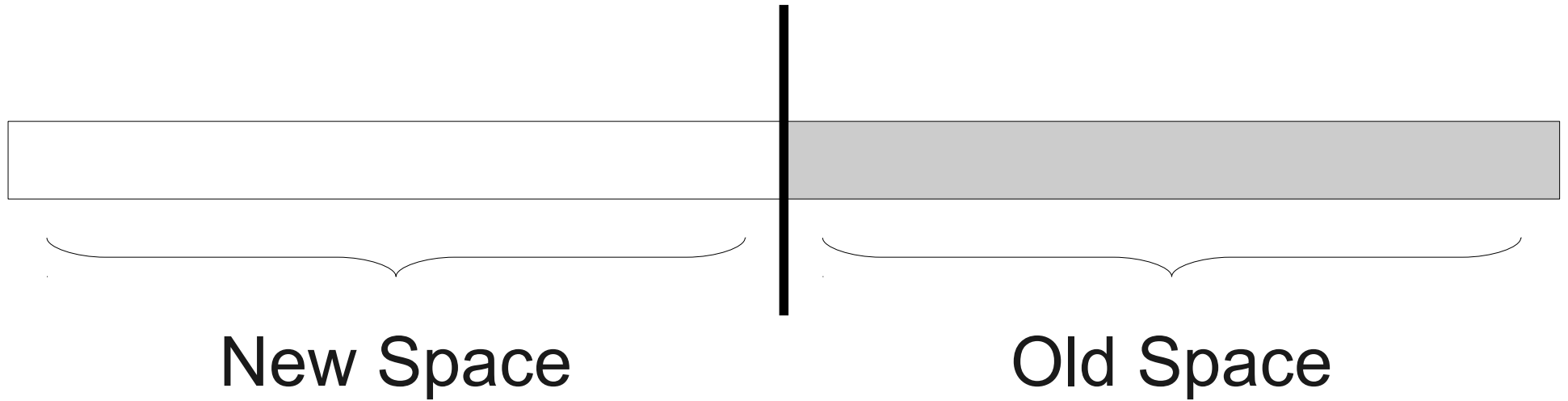
# The Stop-and-Copy Collector



# The Stop-and-Copy Collector

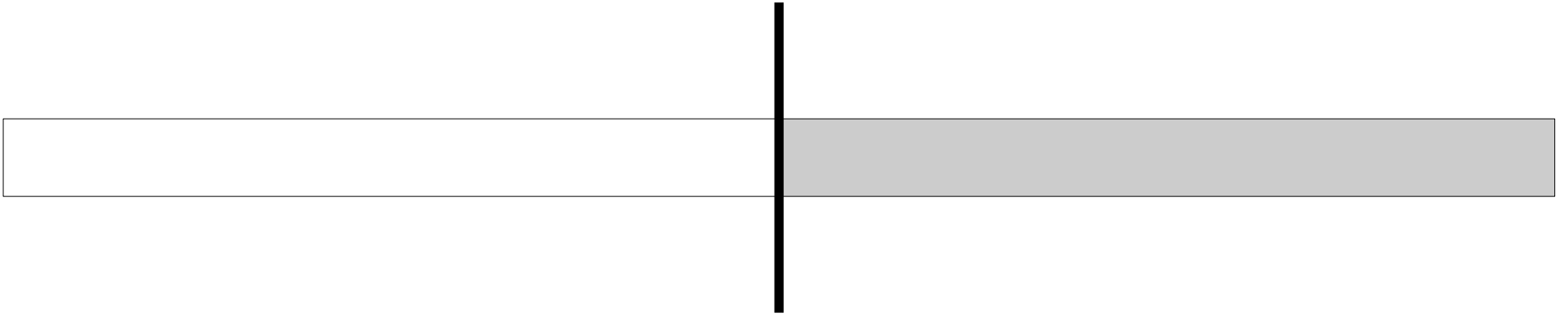


# The Stop-and-Copy Collector

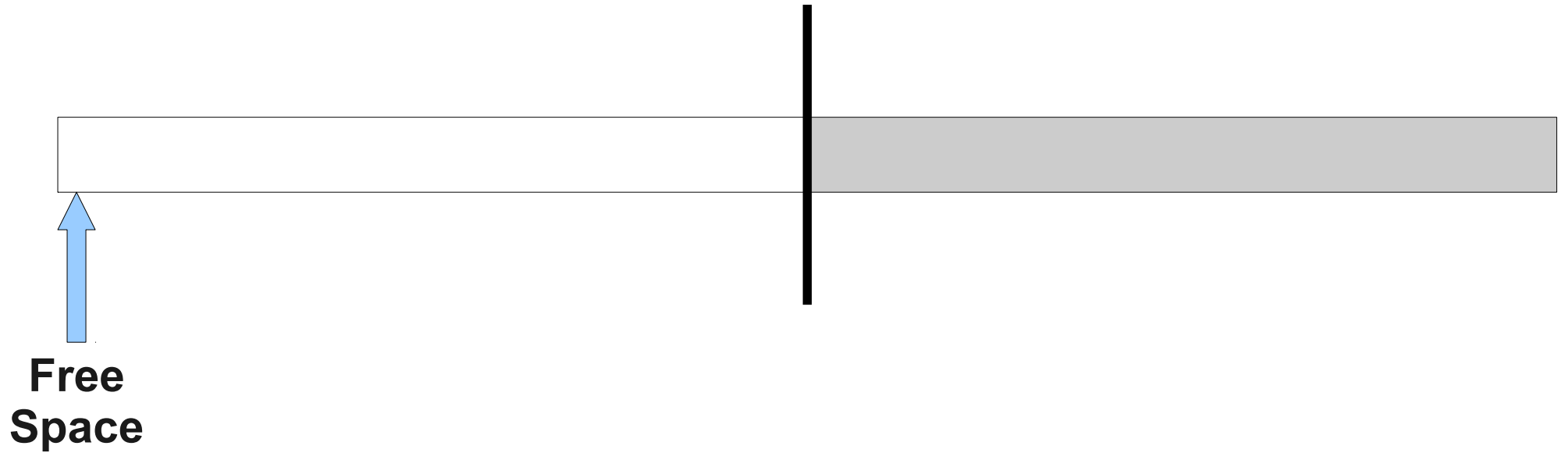




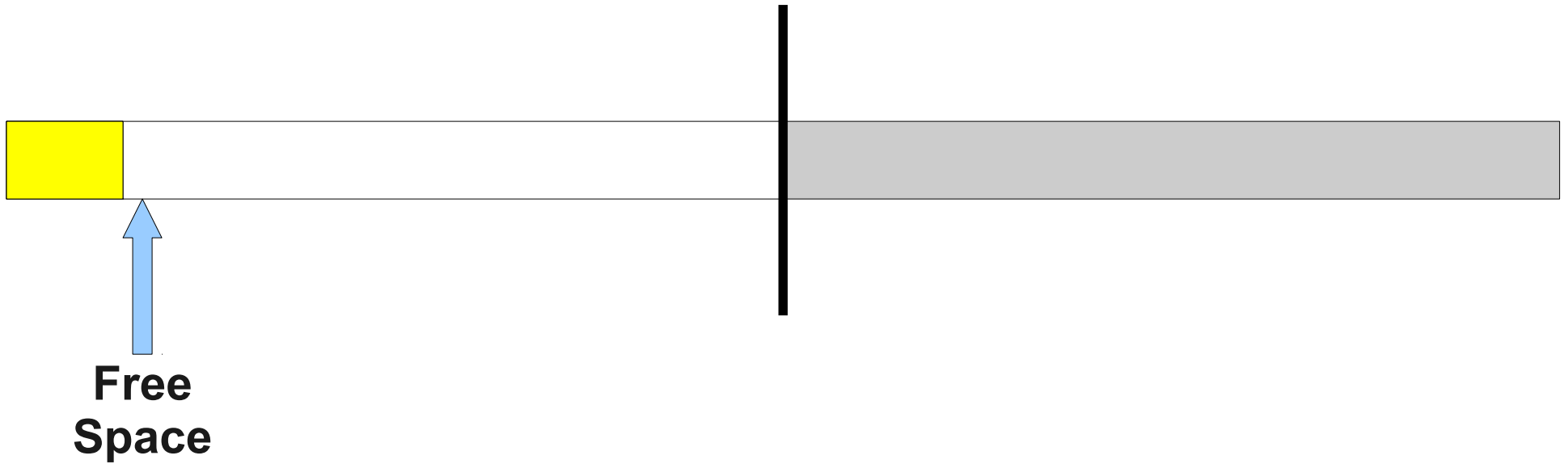
# The Stop-and-Copy Collector



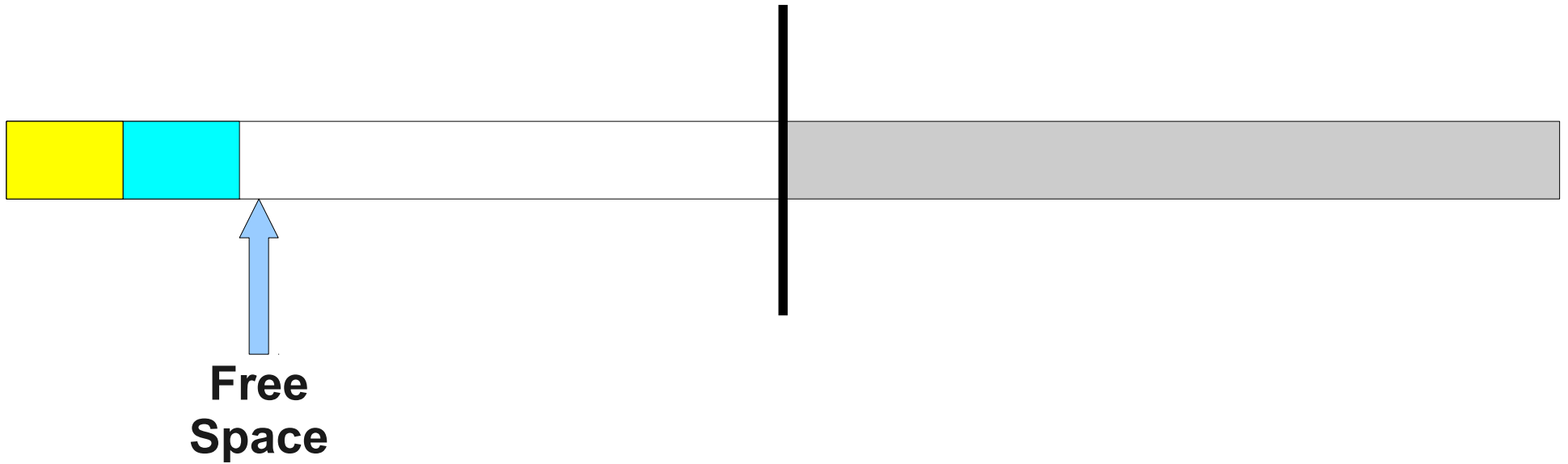
# The Stop-and-Copy Collector



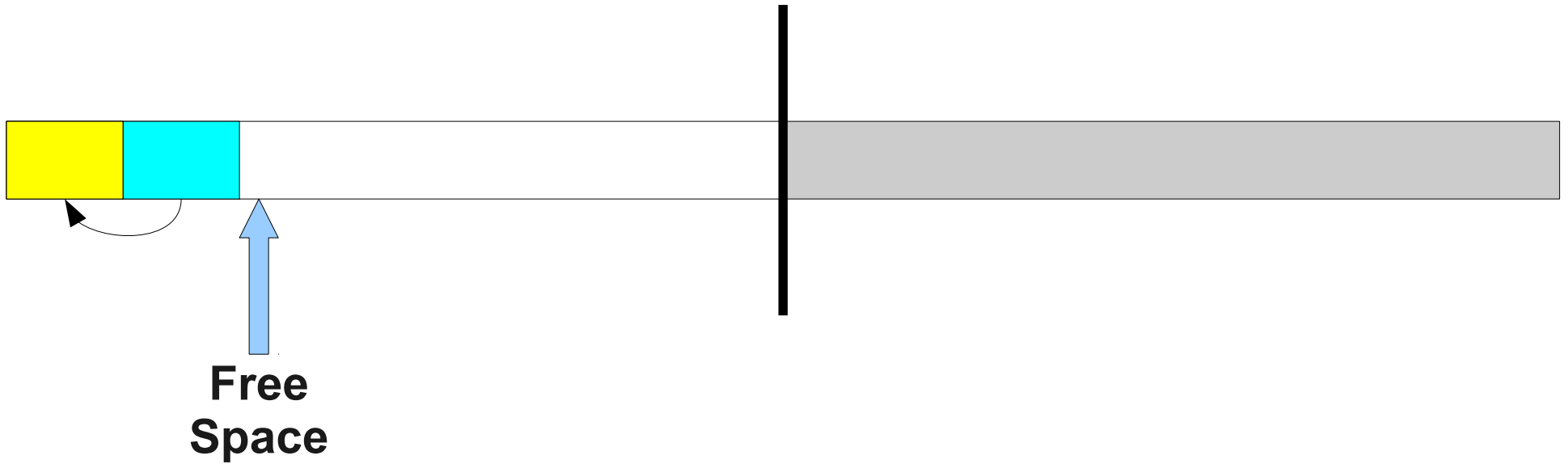
# The Stop-and-Copy Collector



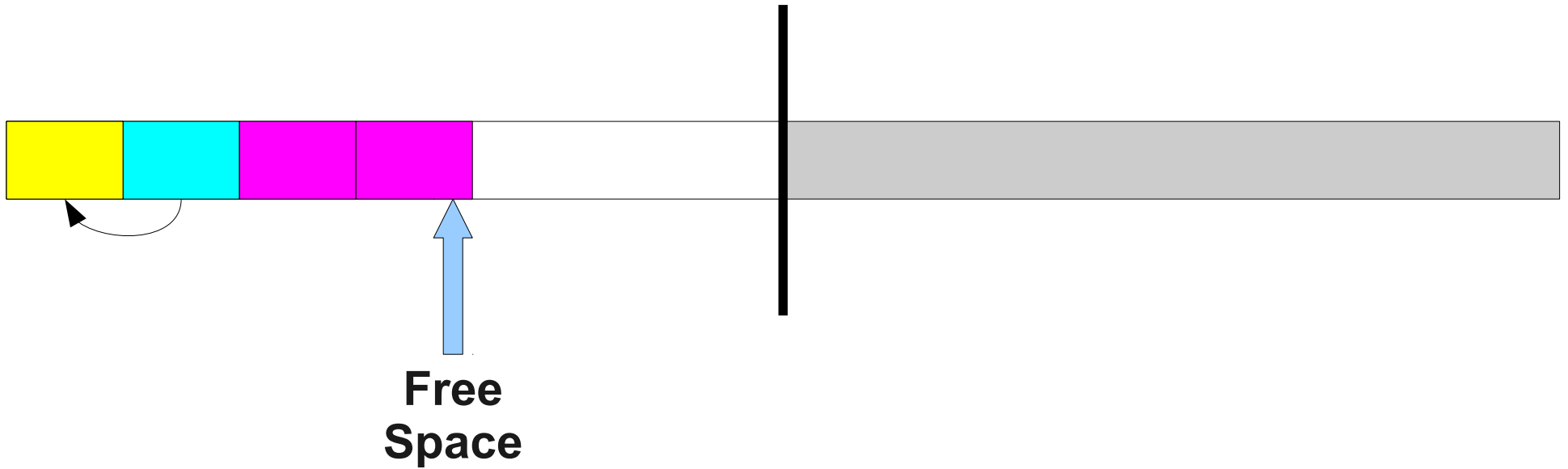
# The Stop-and-Copy Collector



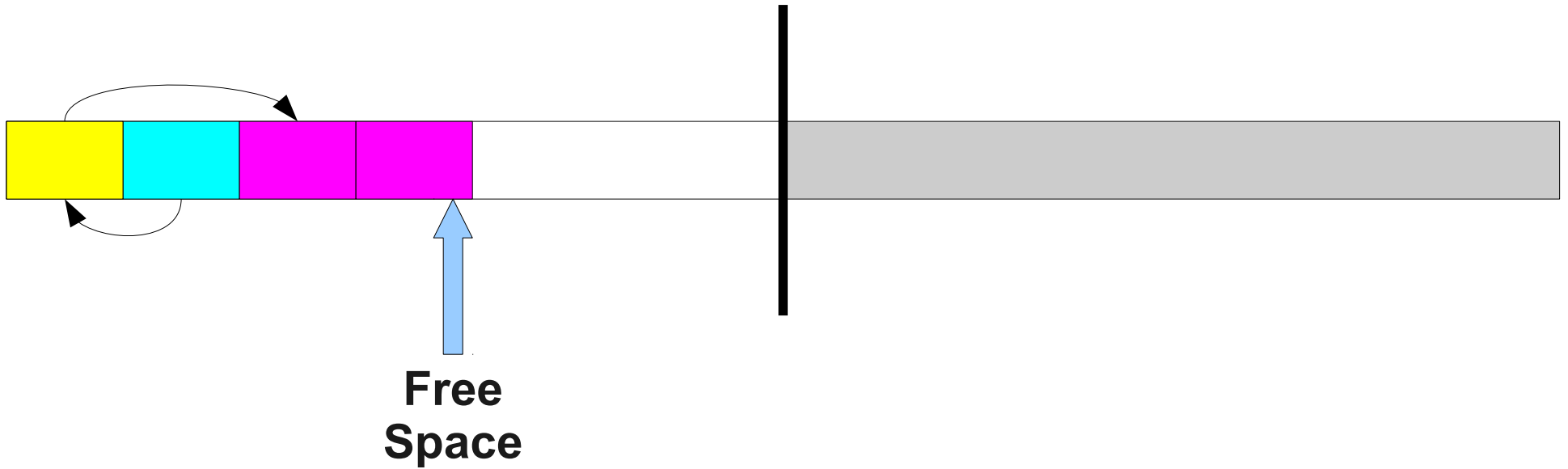
# The Stop-and-Copy Collector



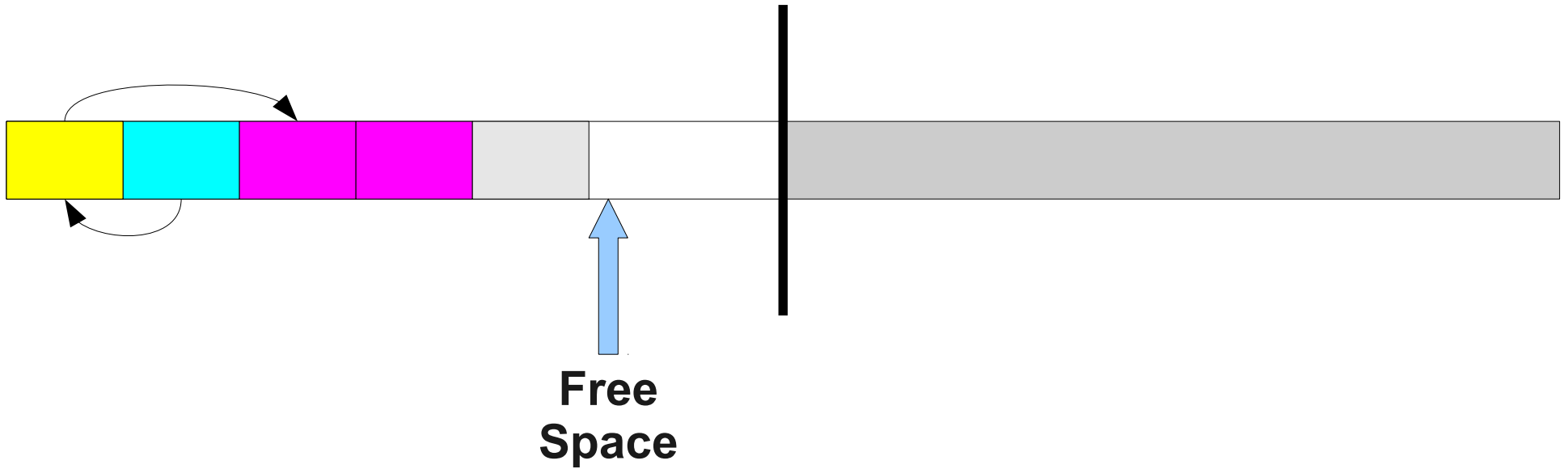
# The Stop-and-Copy Collector



# The Stop-and-Copy Collector

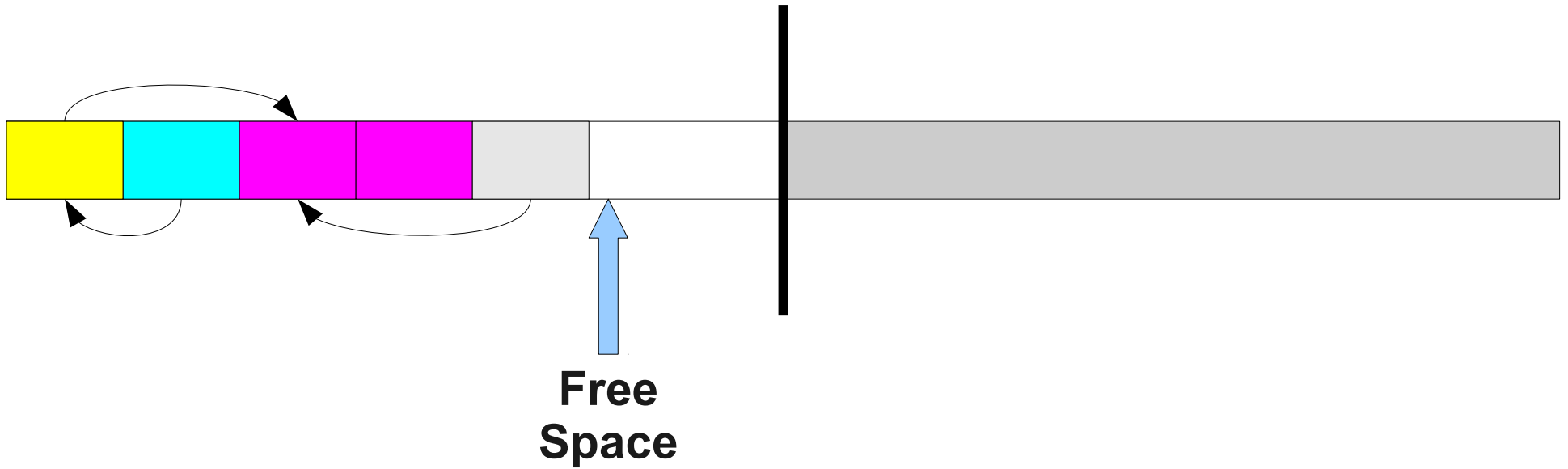


# The Stop-and-Copy Collector

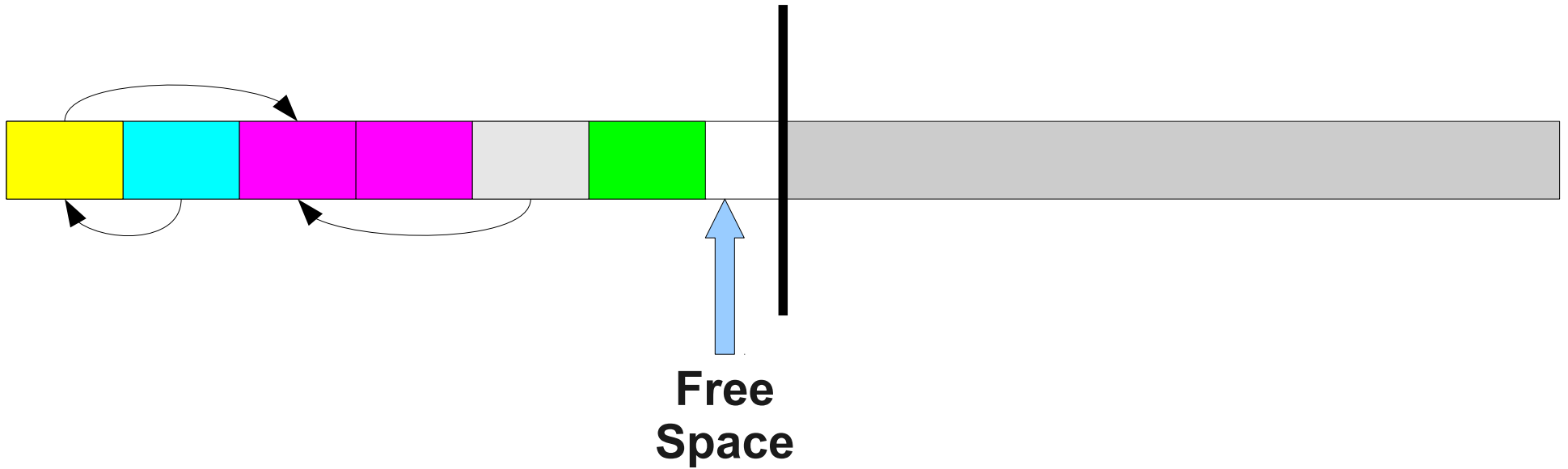




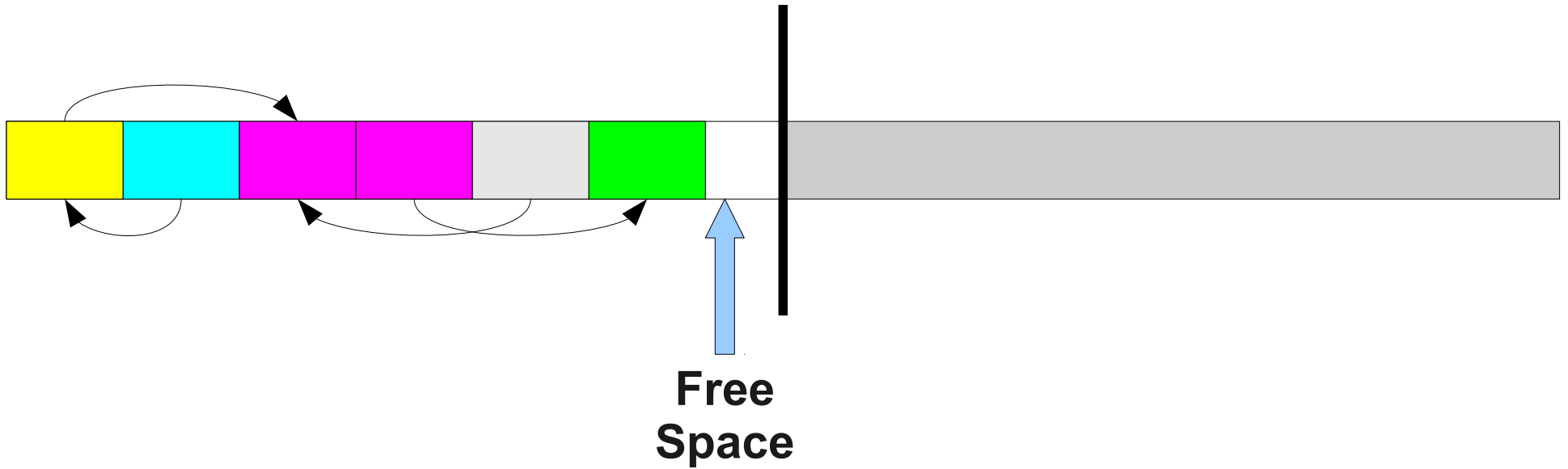
# The Stop-and-Copy Collector



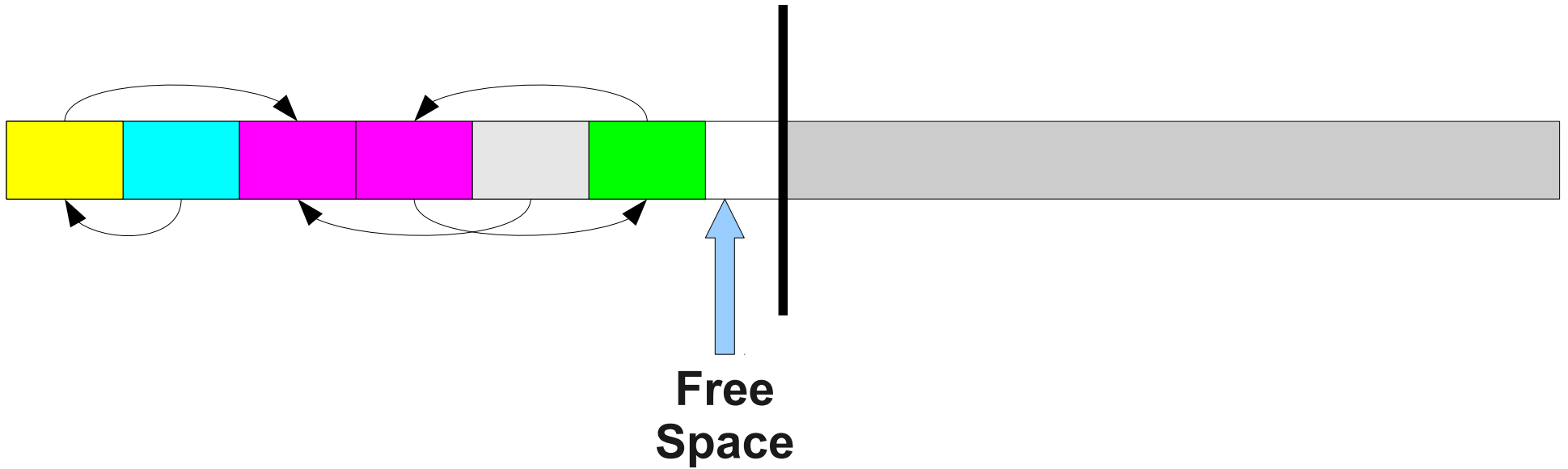
# The Stop-and-Copy Collector



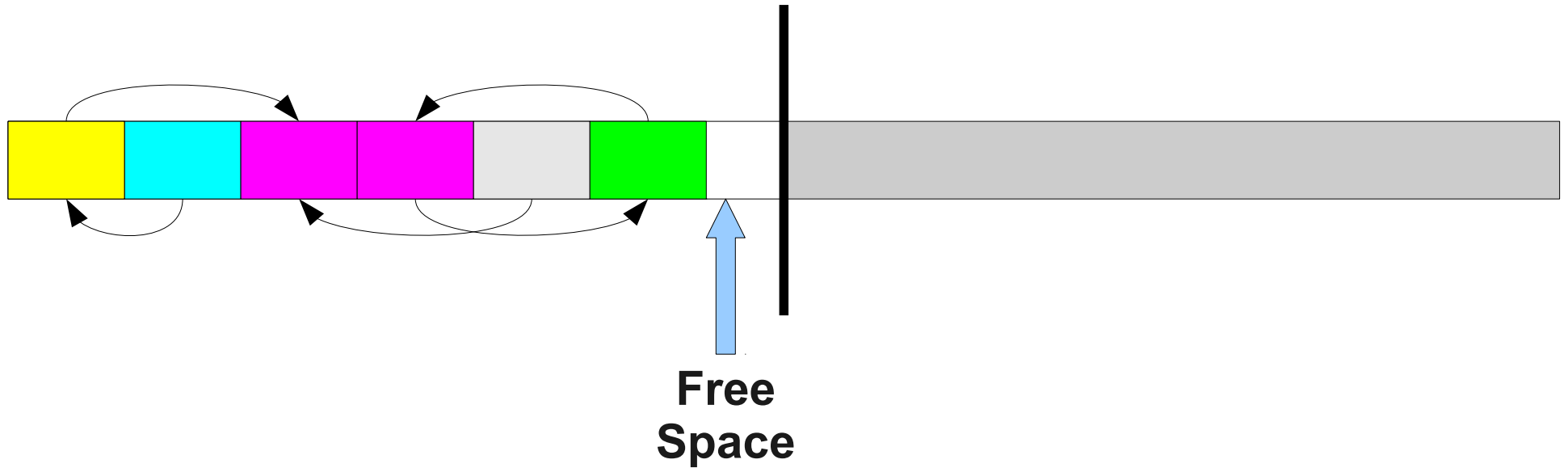
# The Stop-and-Copy Collector



# The Stop-and-Copy Collector

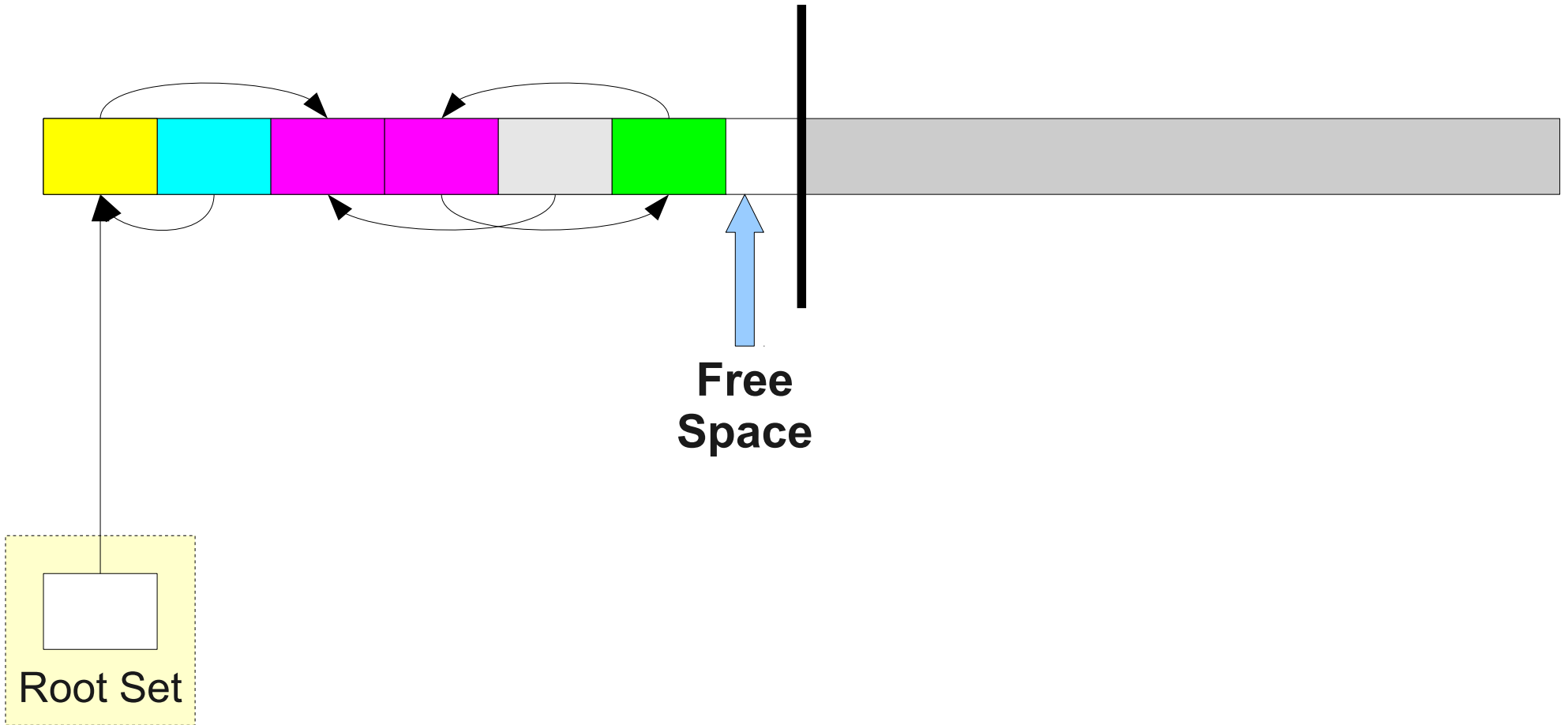


# The Stop-and-Copy Collector

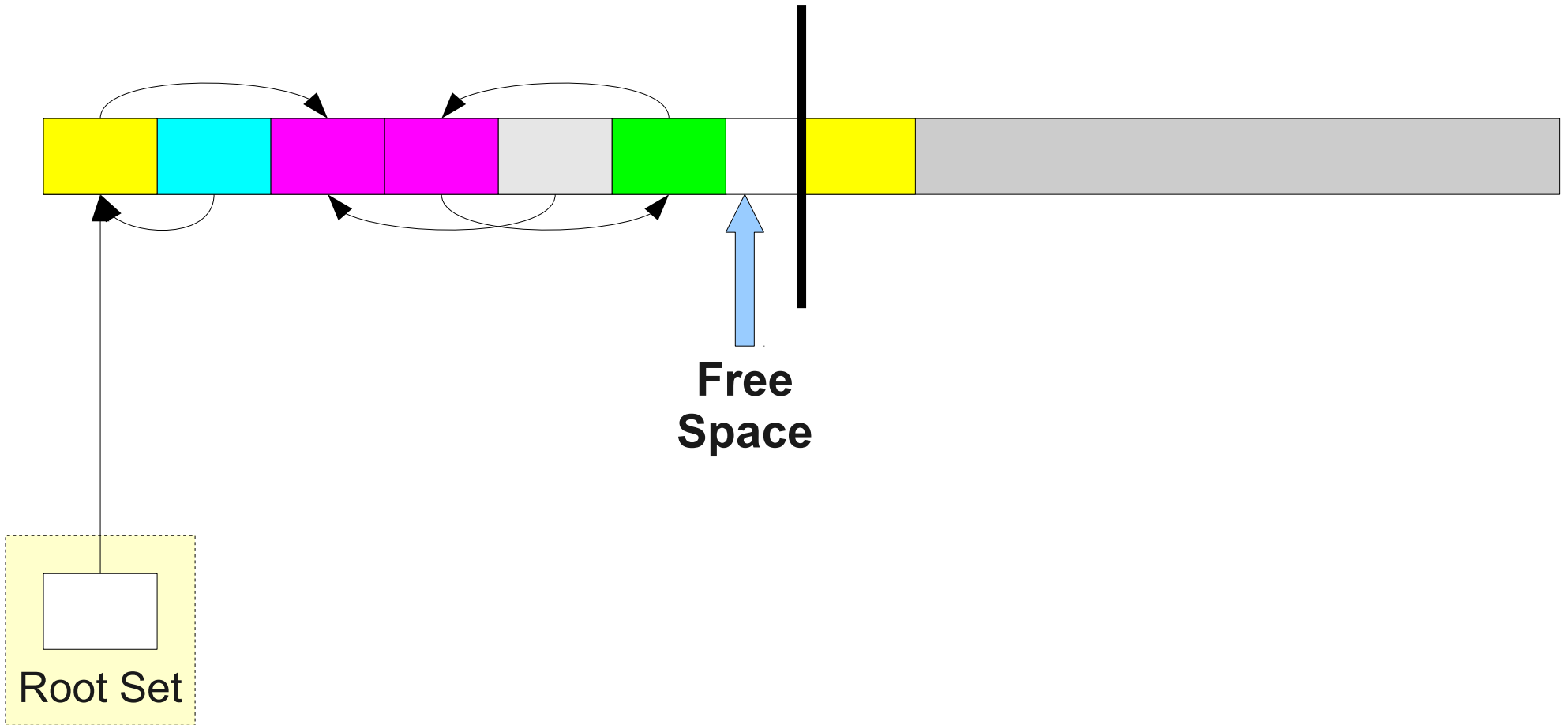


out of space!

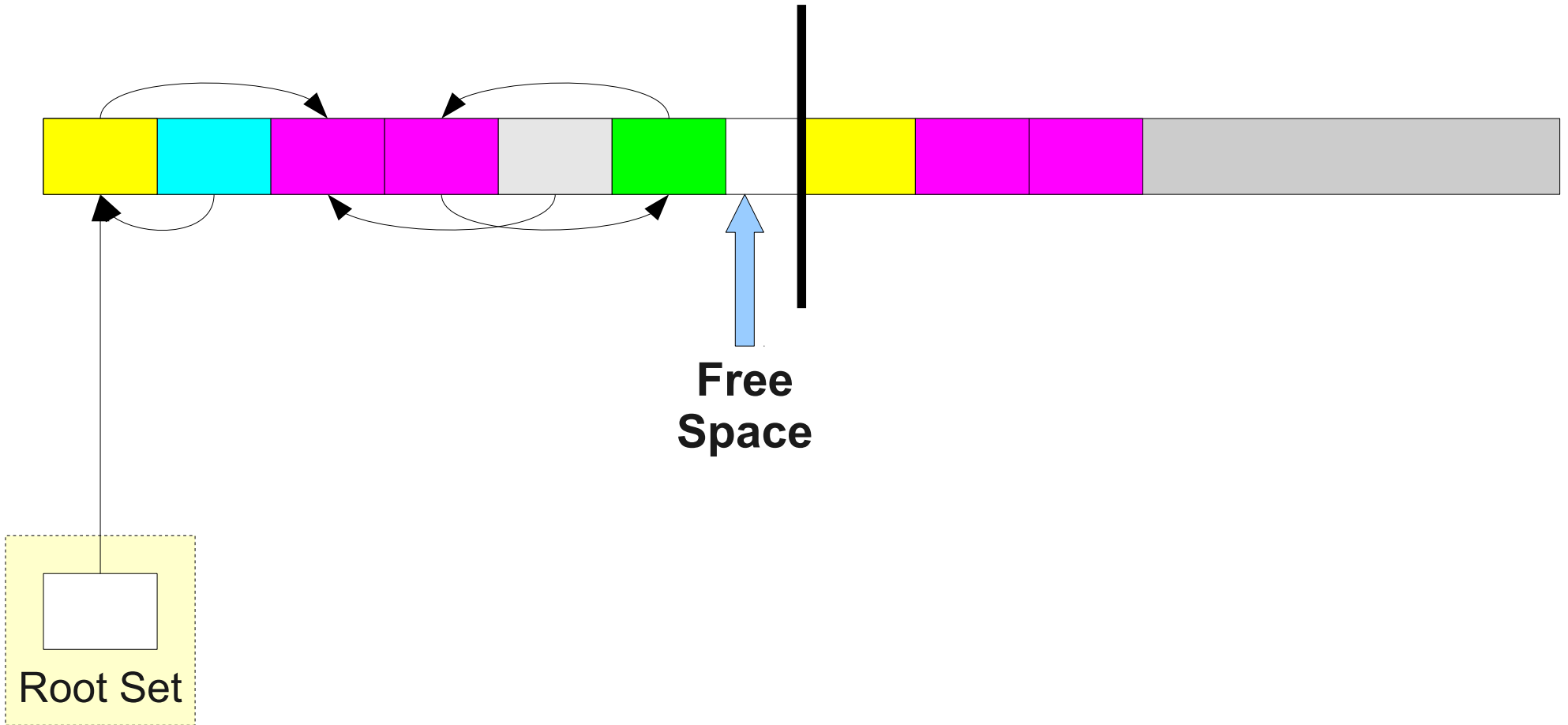
# The Stop-and-Copy Collector



# The Stop-and-Copy Collector

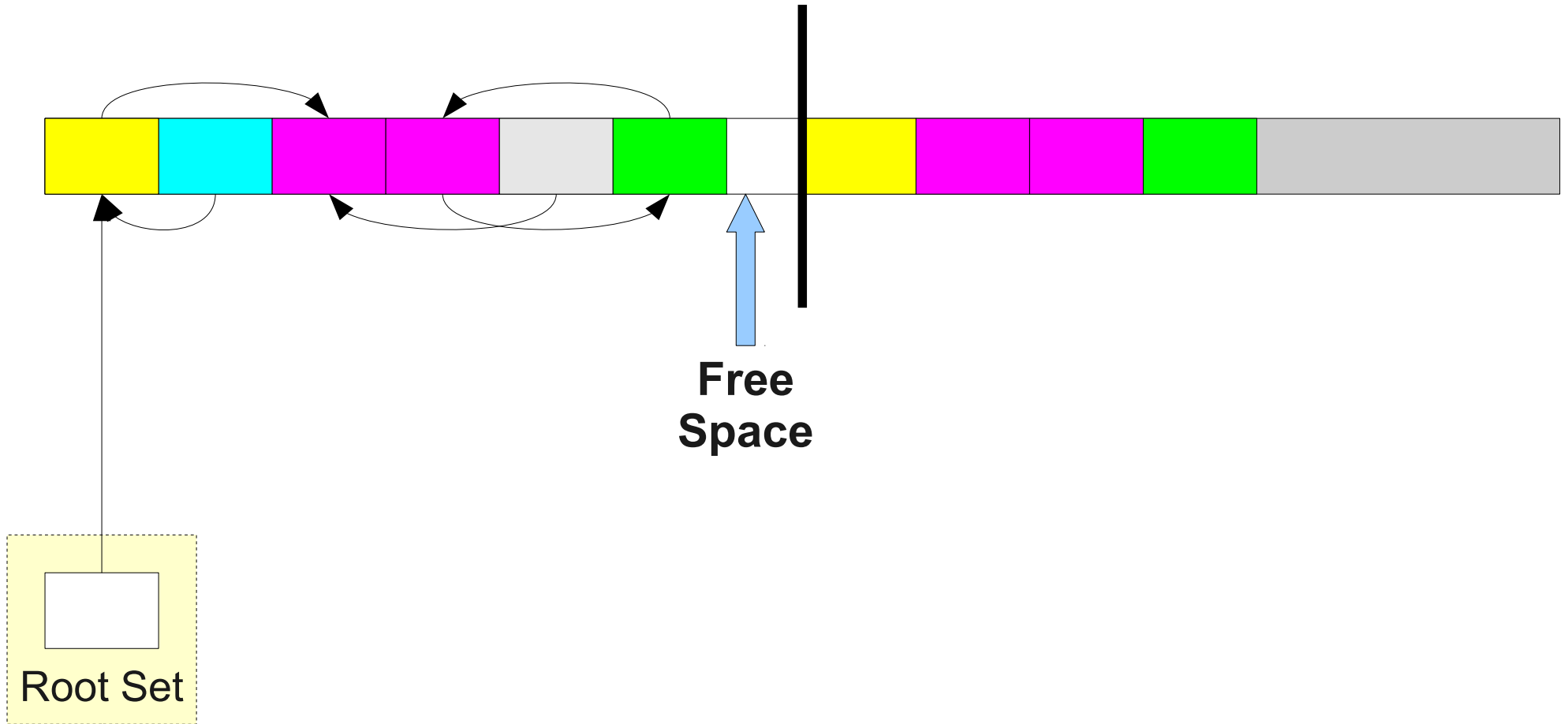


# The Stop-and-Copy Collector

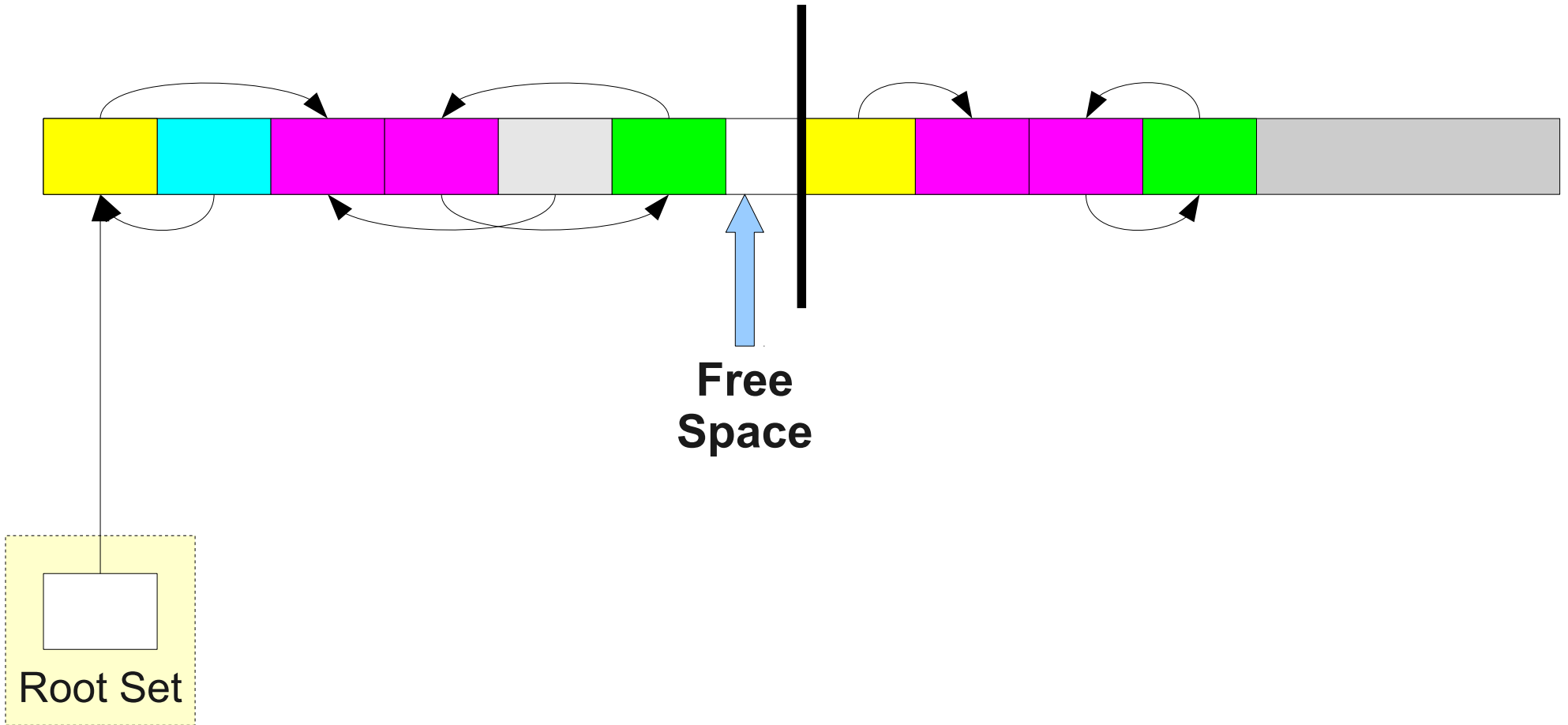




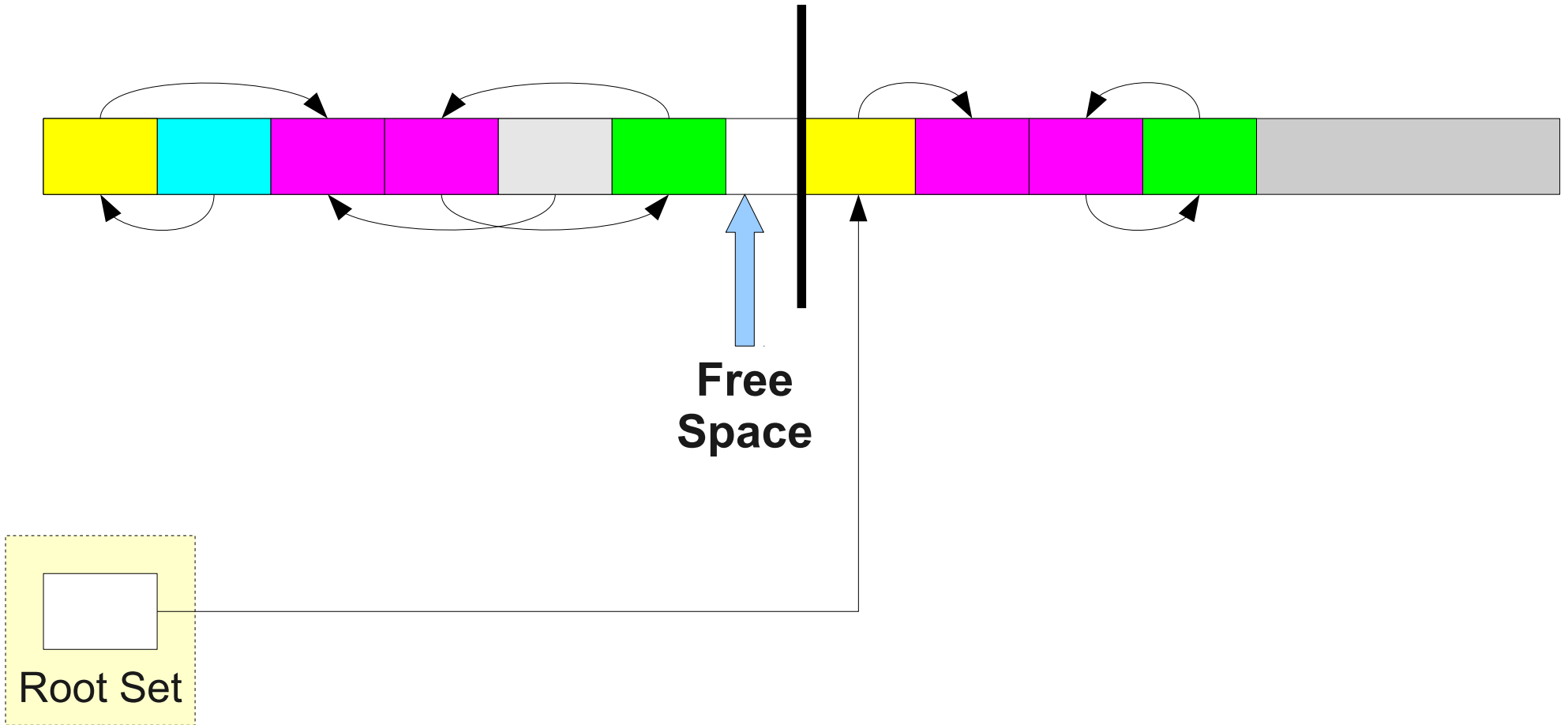
# The Stop-and-Copy Collector



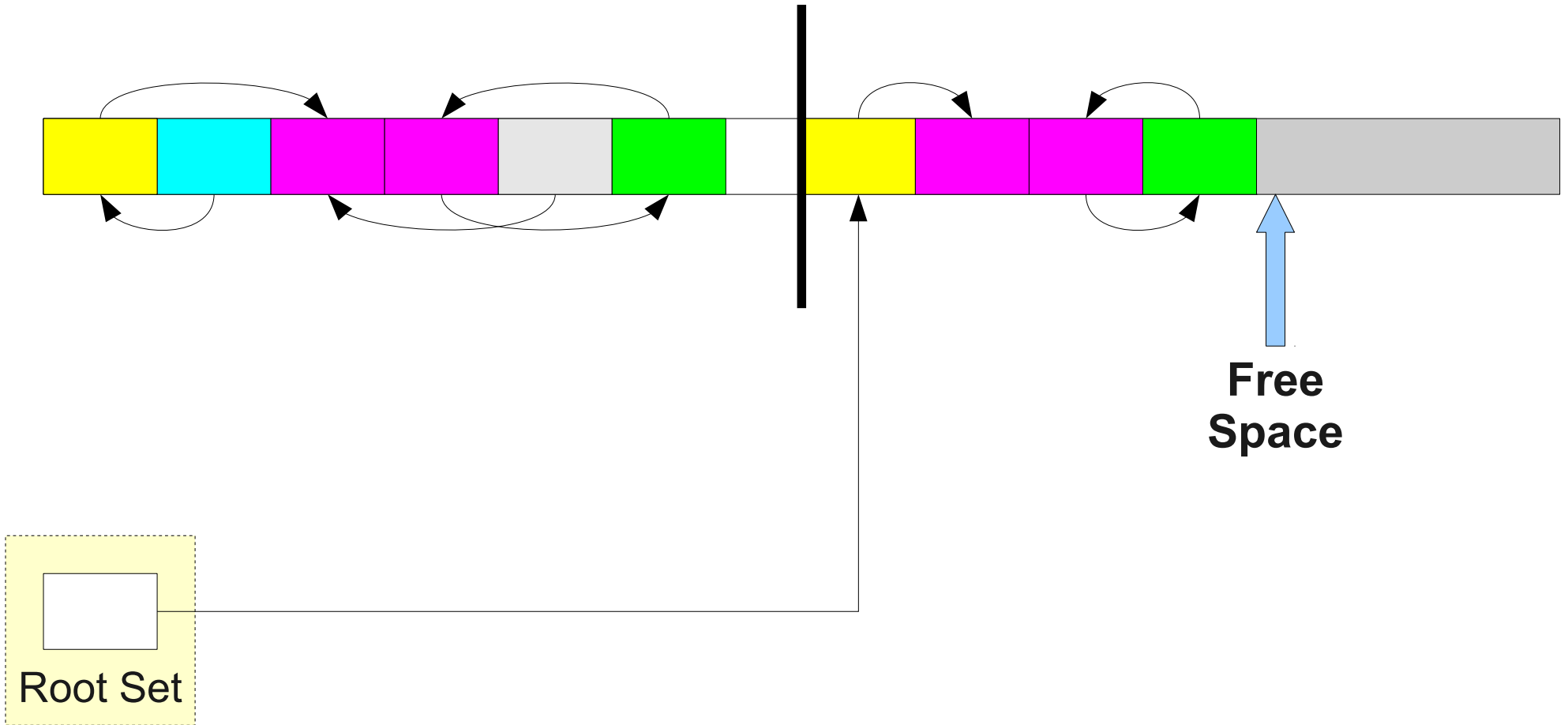
# The Stop-and-Copy Collector



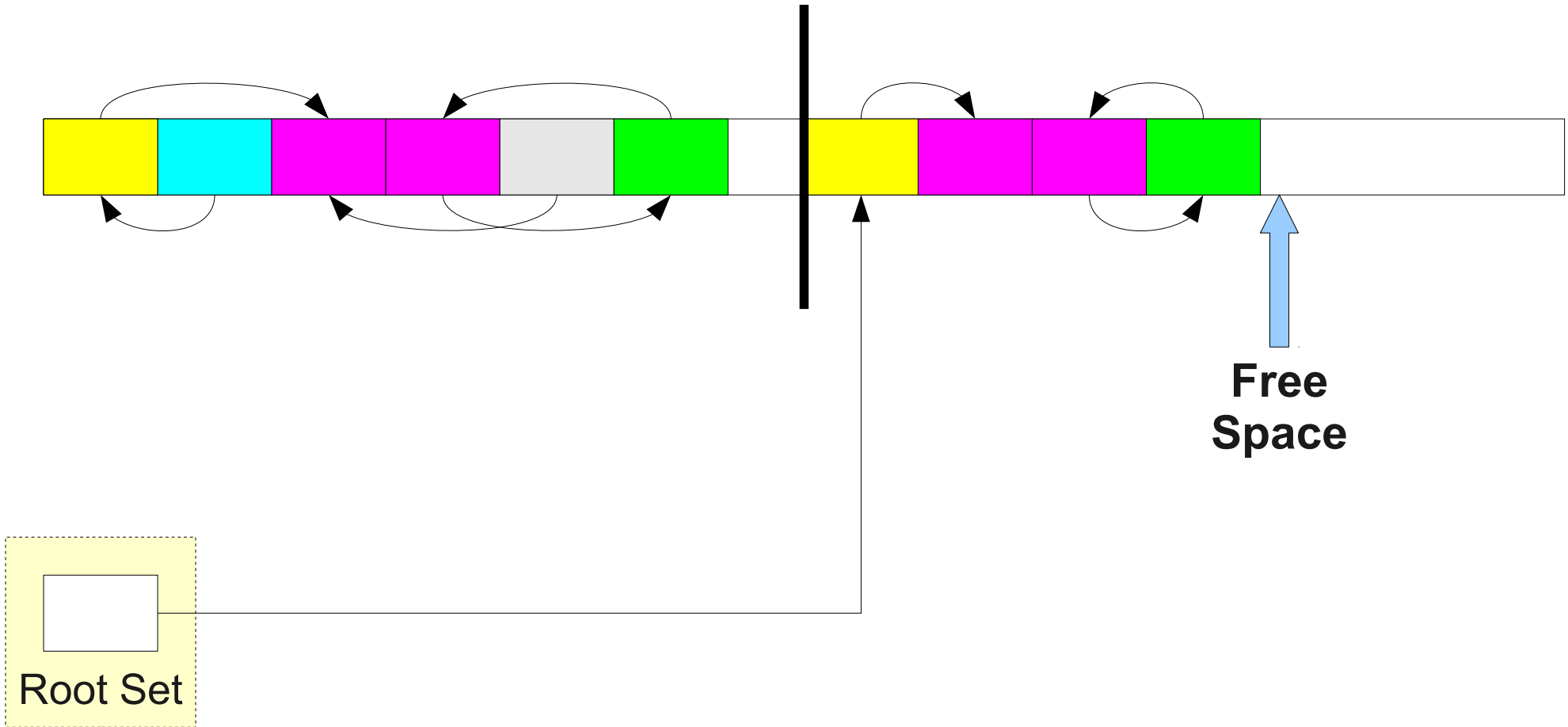
# The Stop-and-Copy Collector



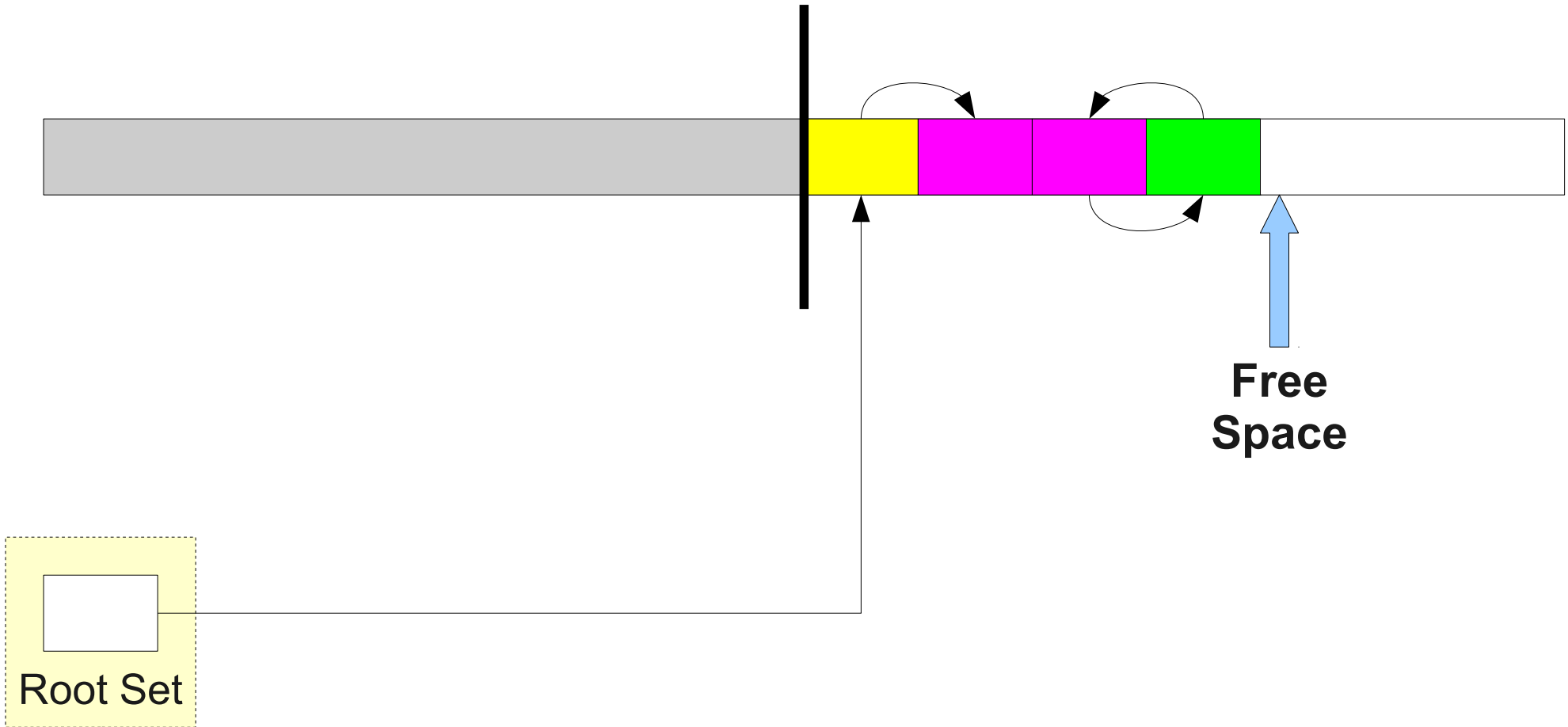
# The Stop-and-Copy Collector



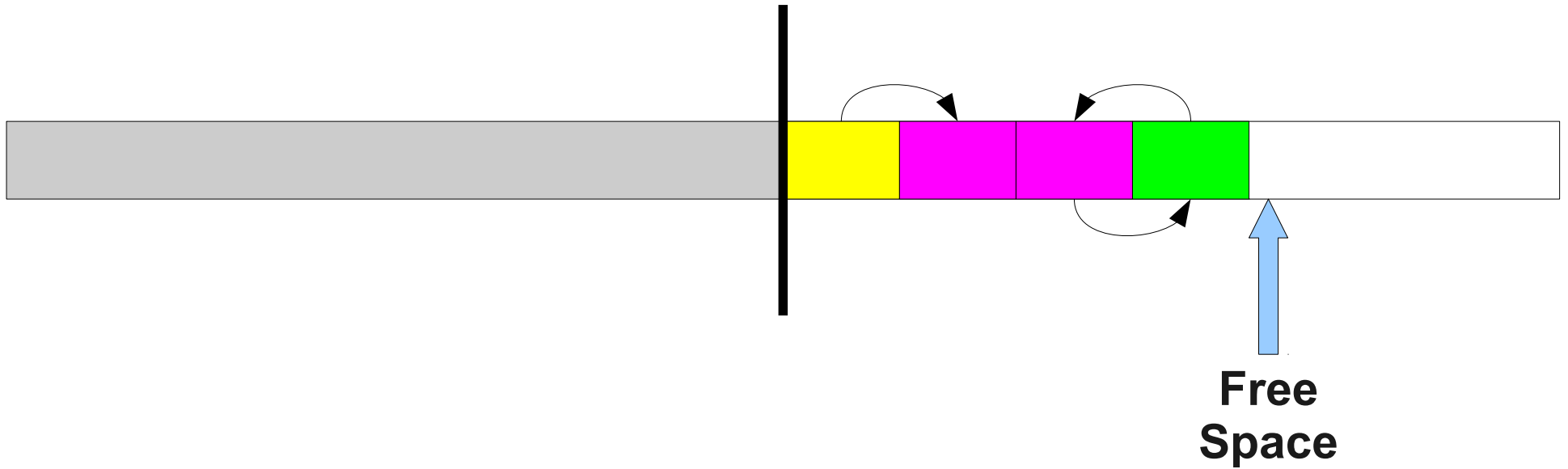
# The Stop-and-Copy Collector



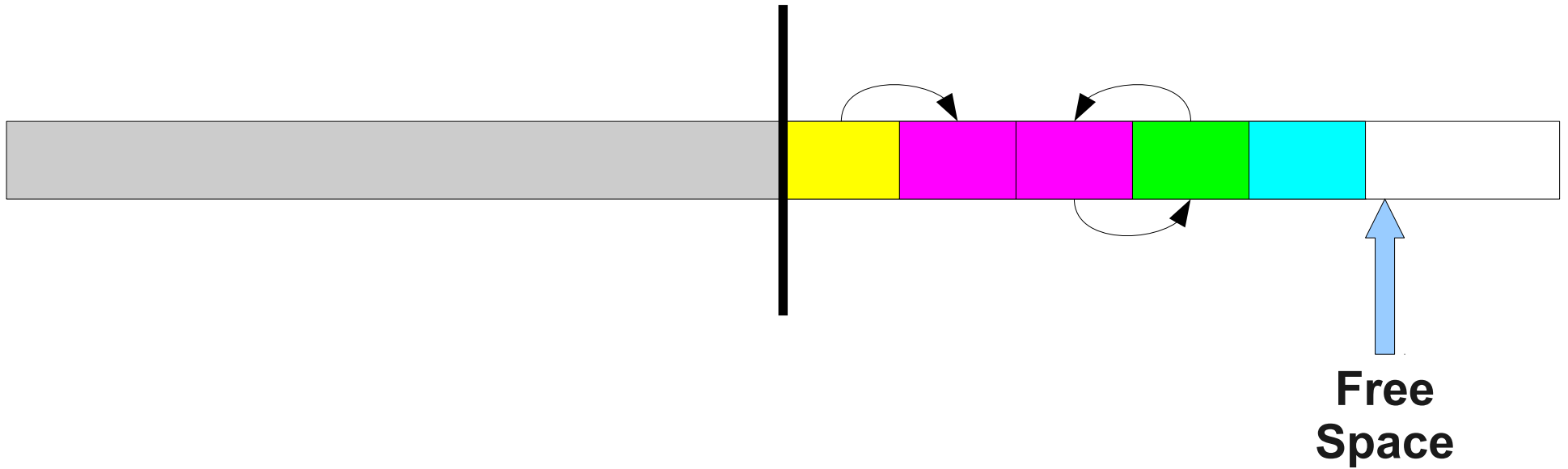
# The Stop-and-Copy Collector



# The Stop-and-Copy Collector

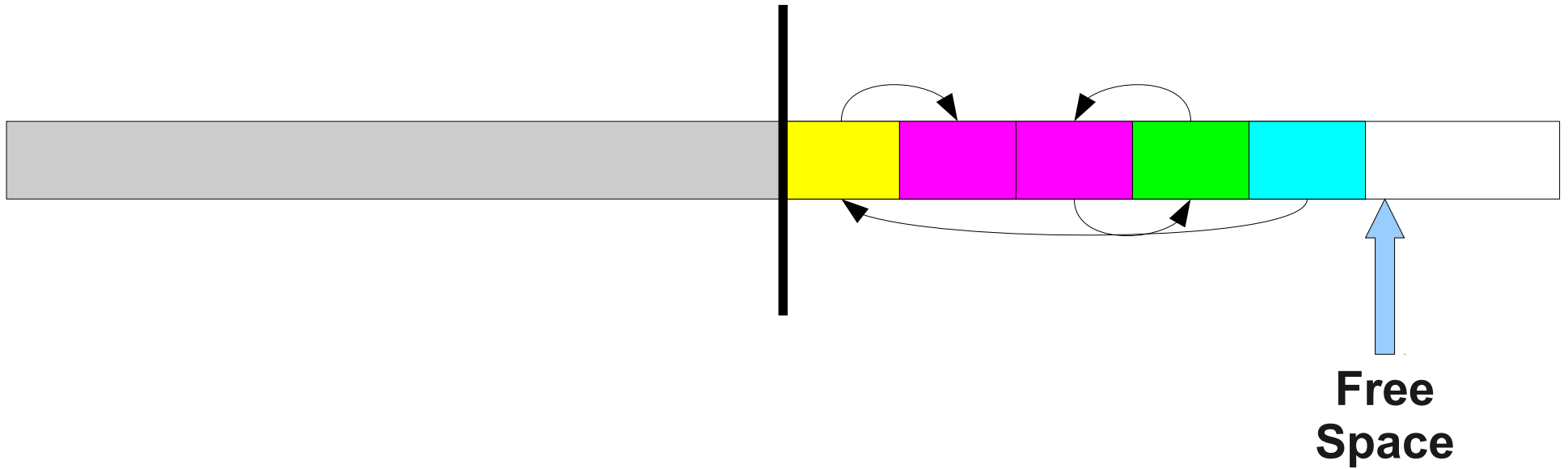


# The Stop-and-Copy Collector

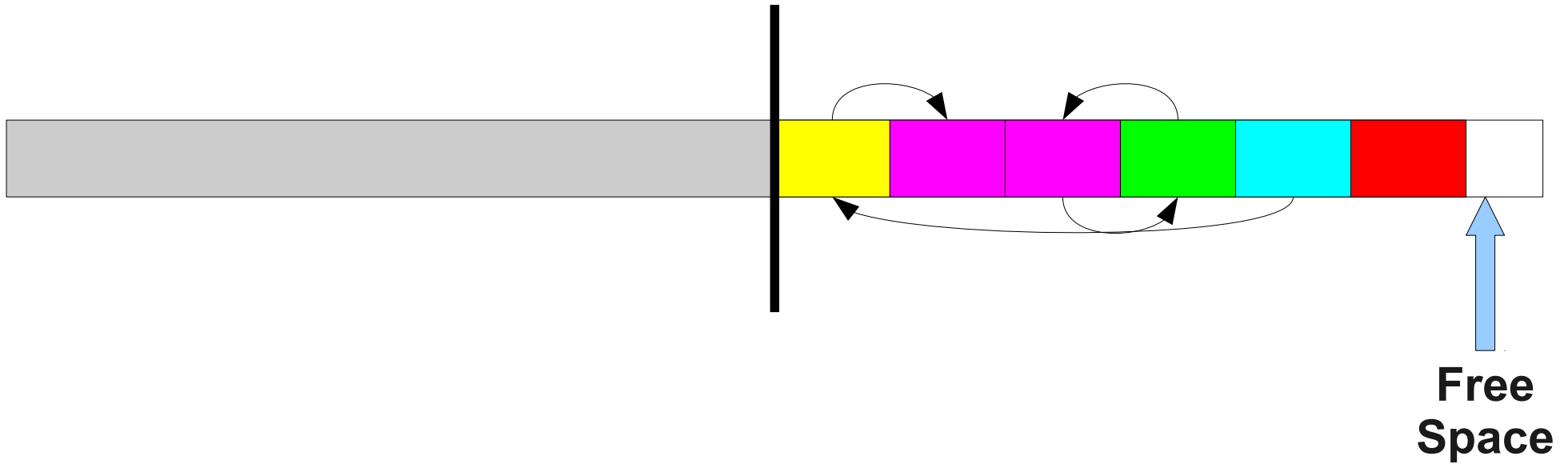




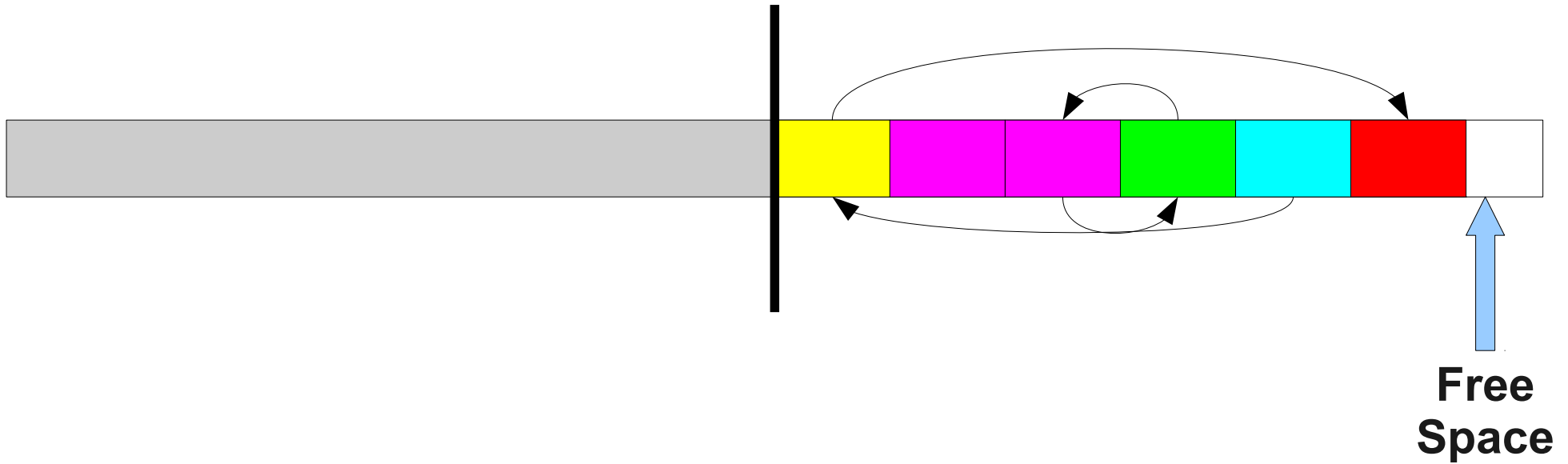
# The Stop-and-Copy Collector



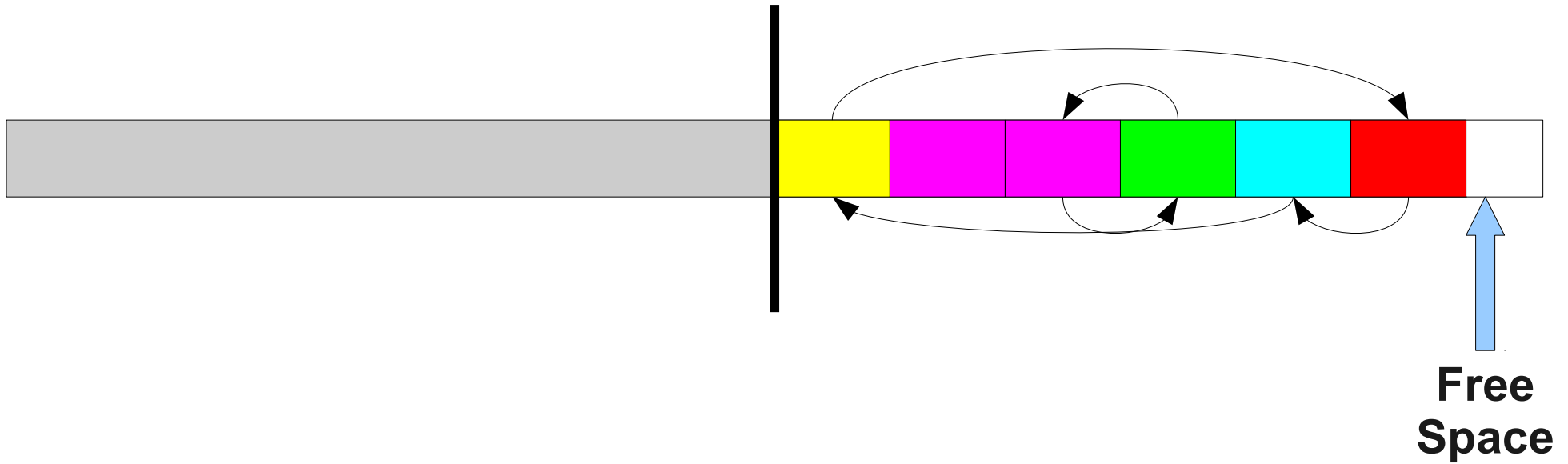
# The Stop-and-Copy Collector



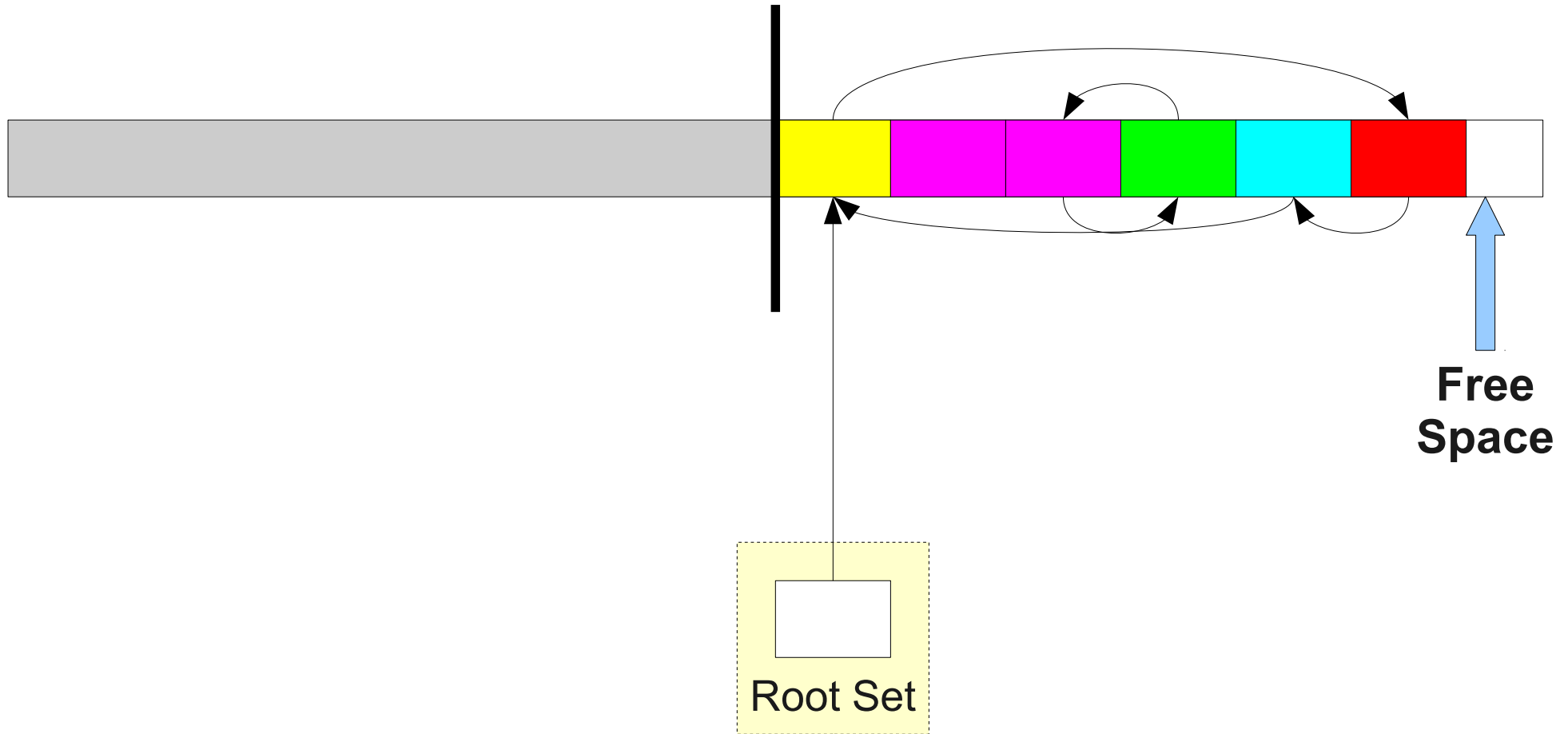
# The Stop-and-Copy Collector



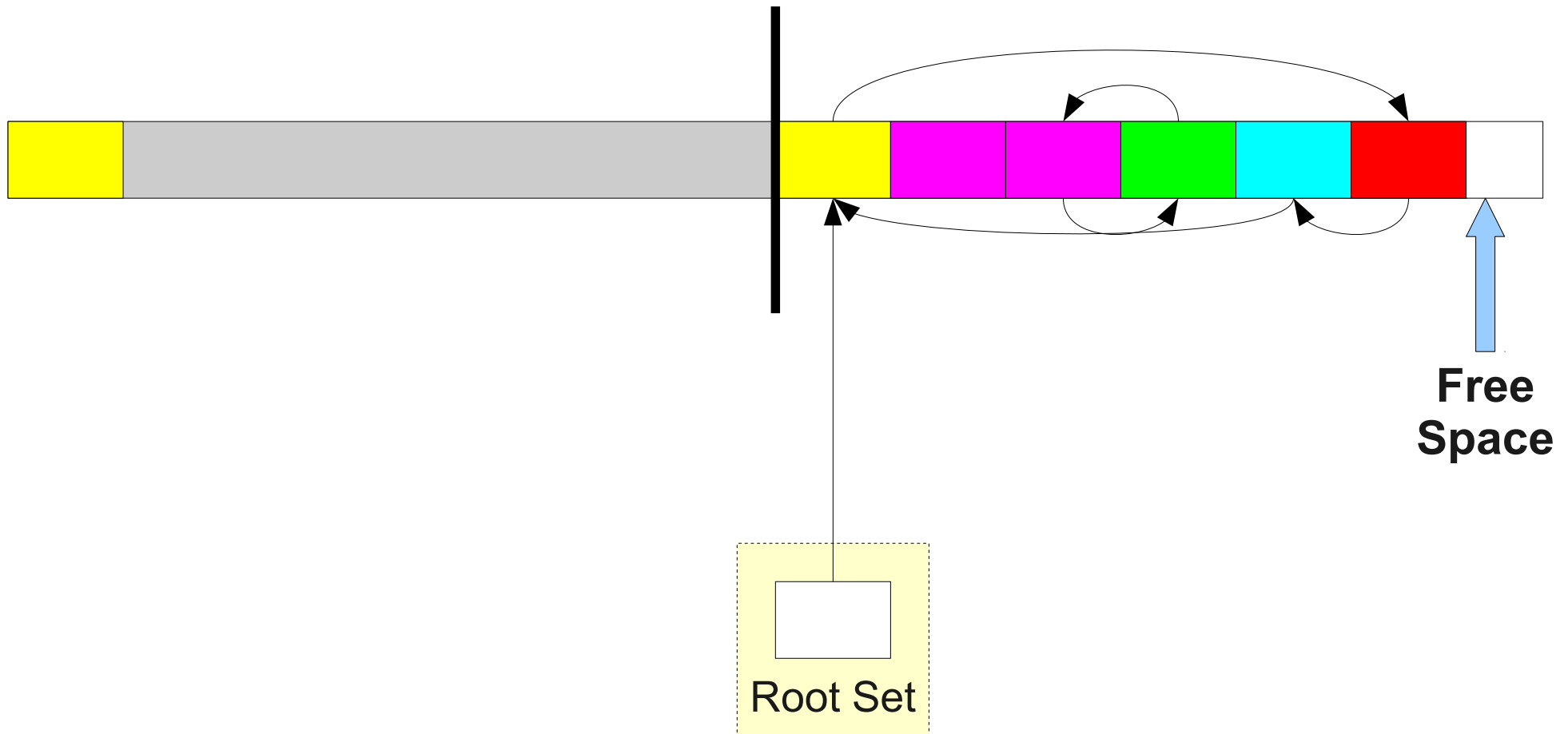
# The Stop-and-Copy Collector



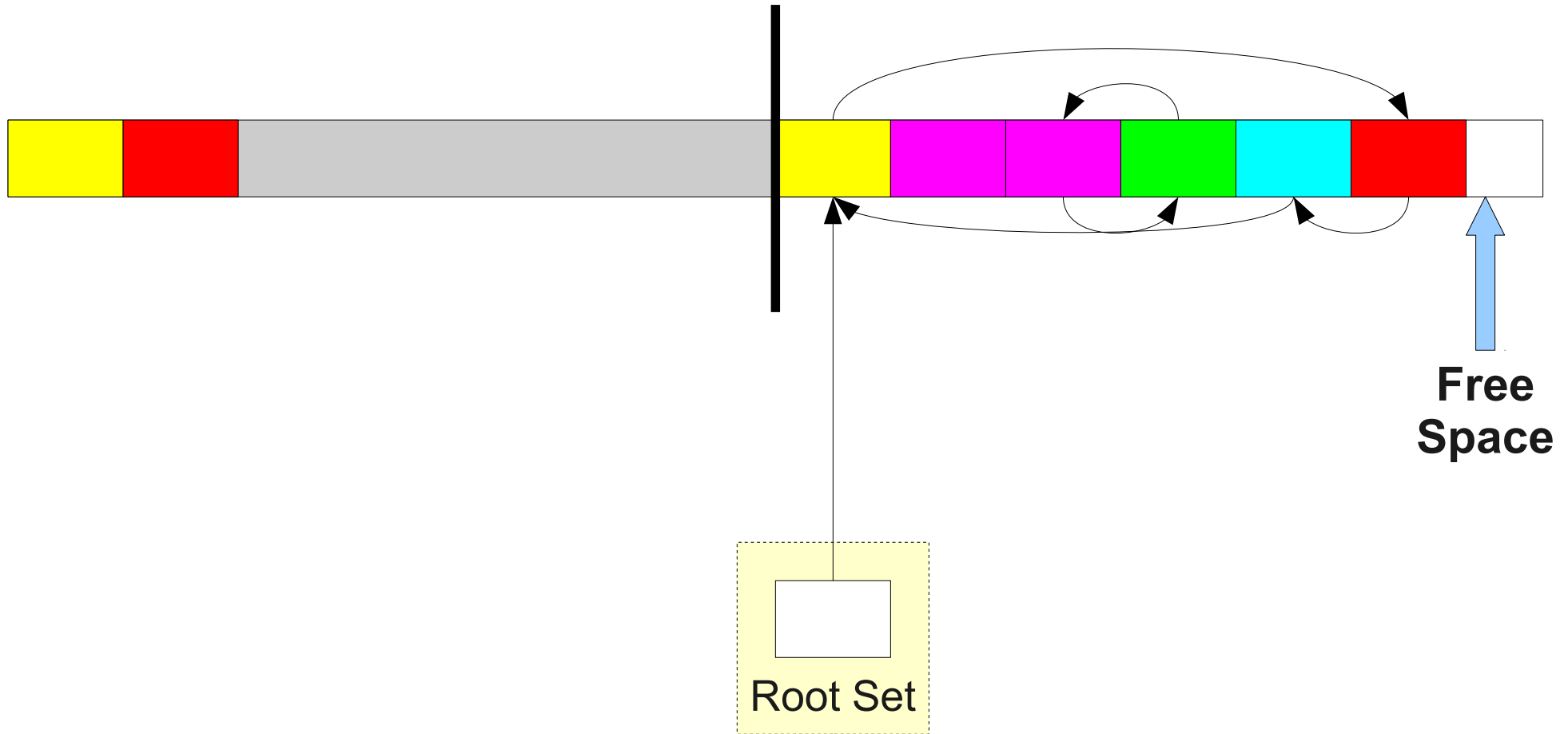
# The Stop-and-Copy Collector



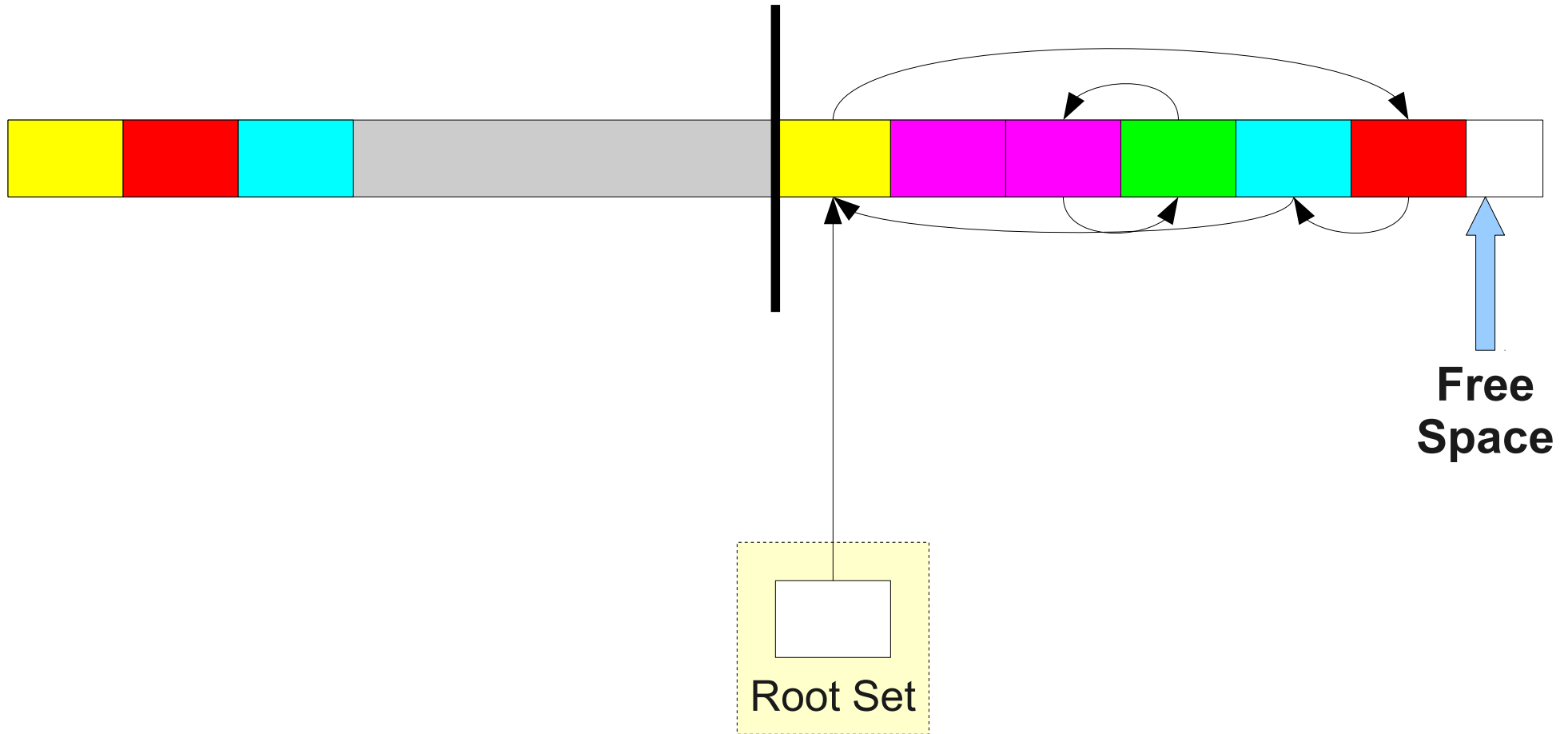
# The Stop-and-Copy Collector



# The Stop-and-Copy Collector

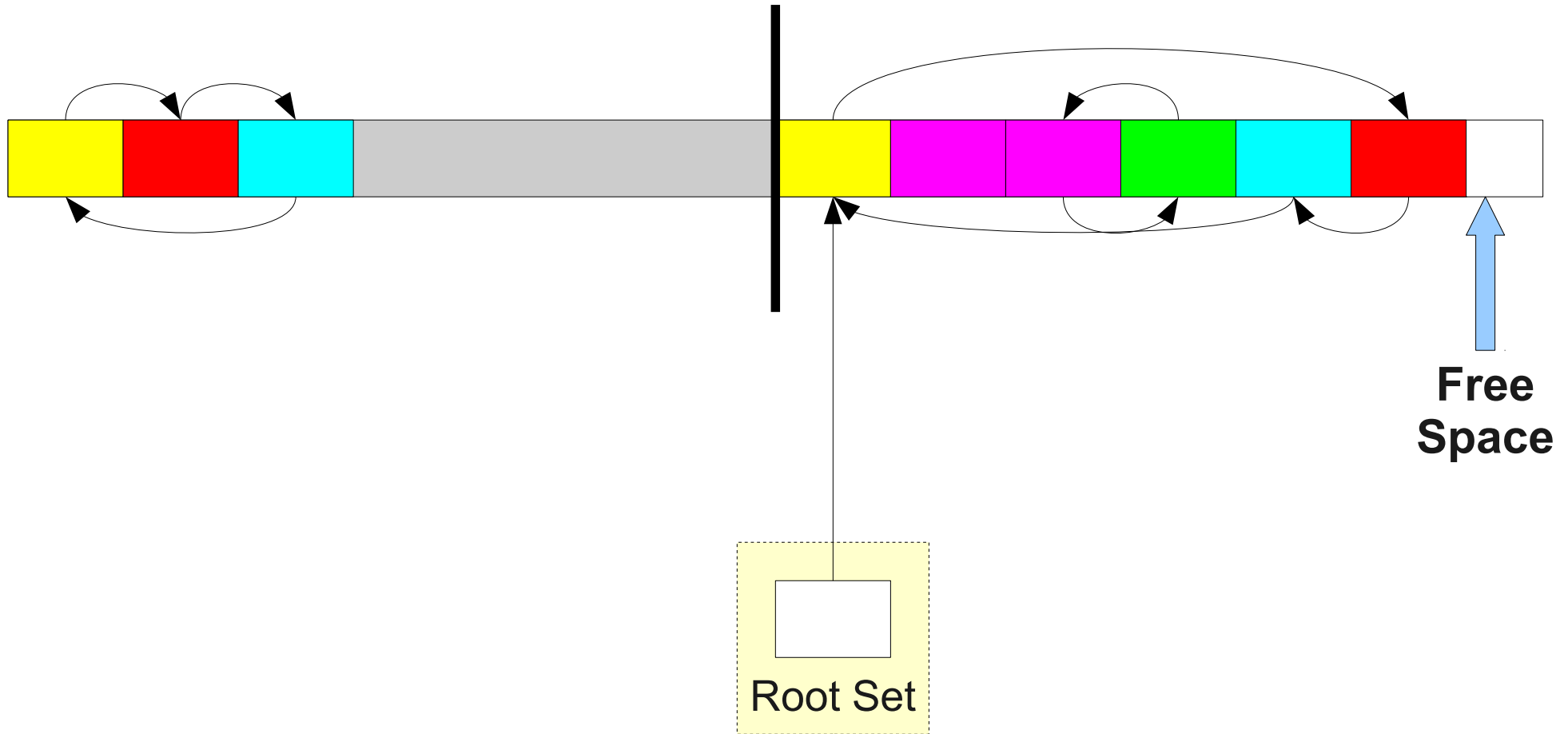


# The Stop-and-Copy Collector

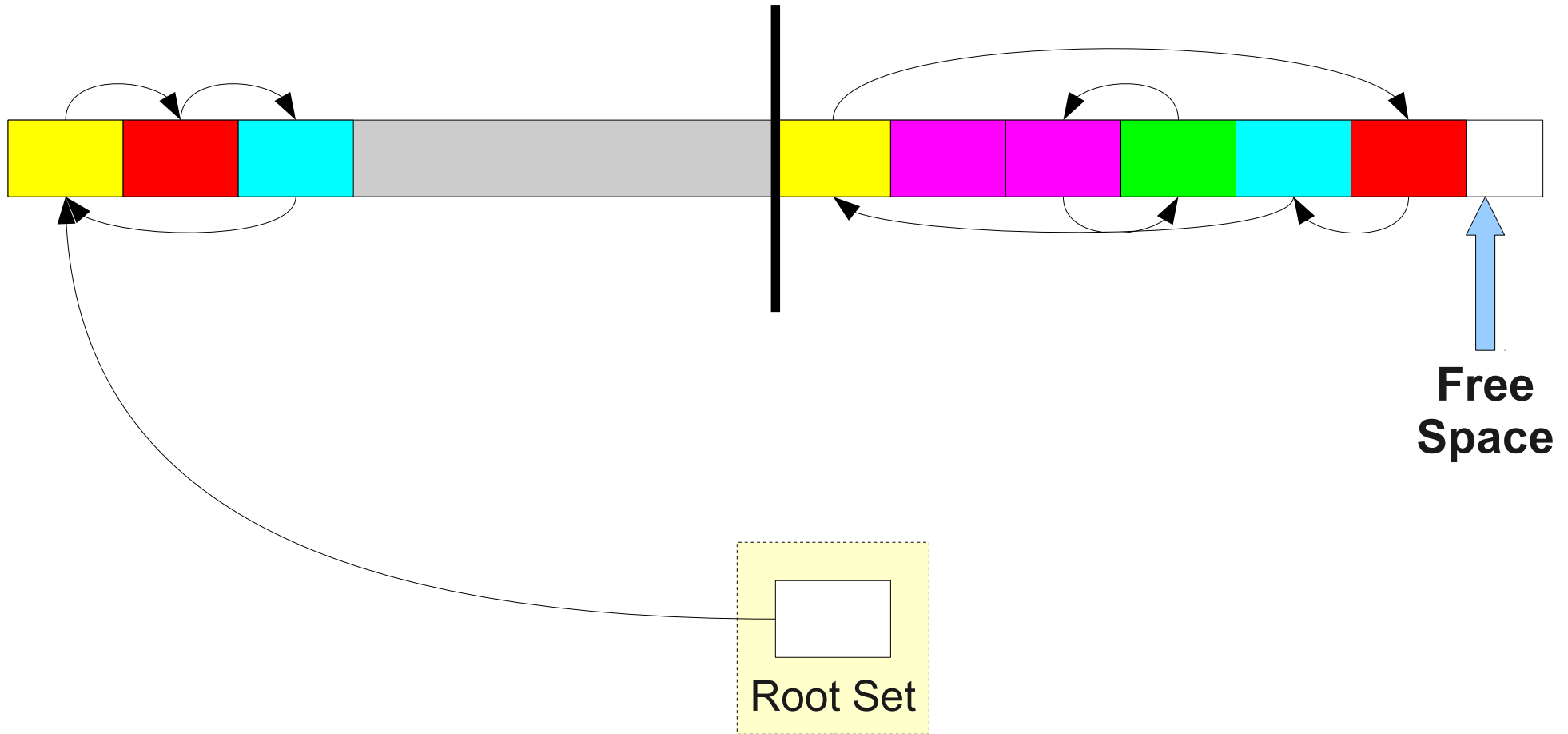




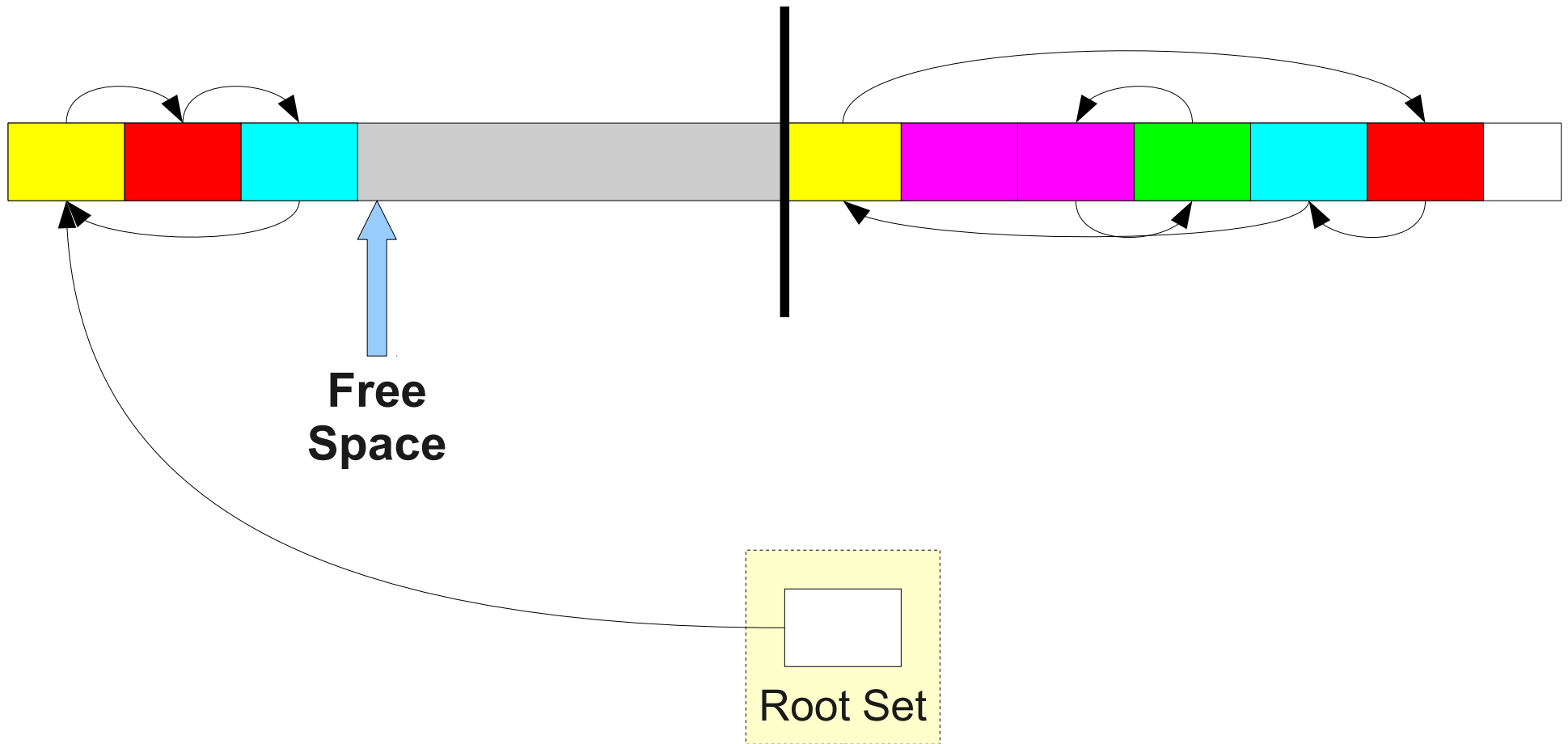
# The Stop-and-Copy Collector



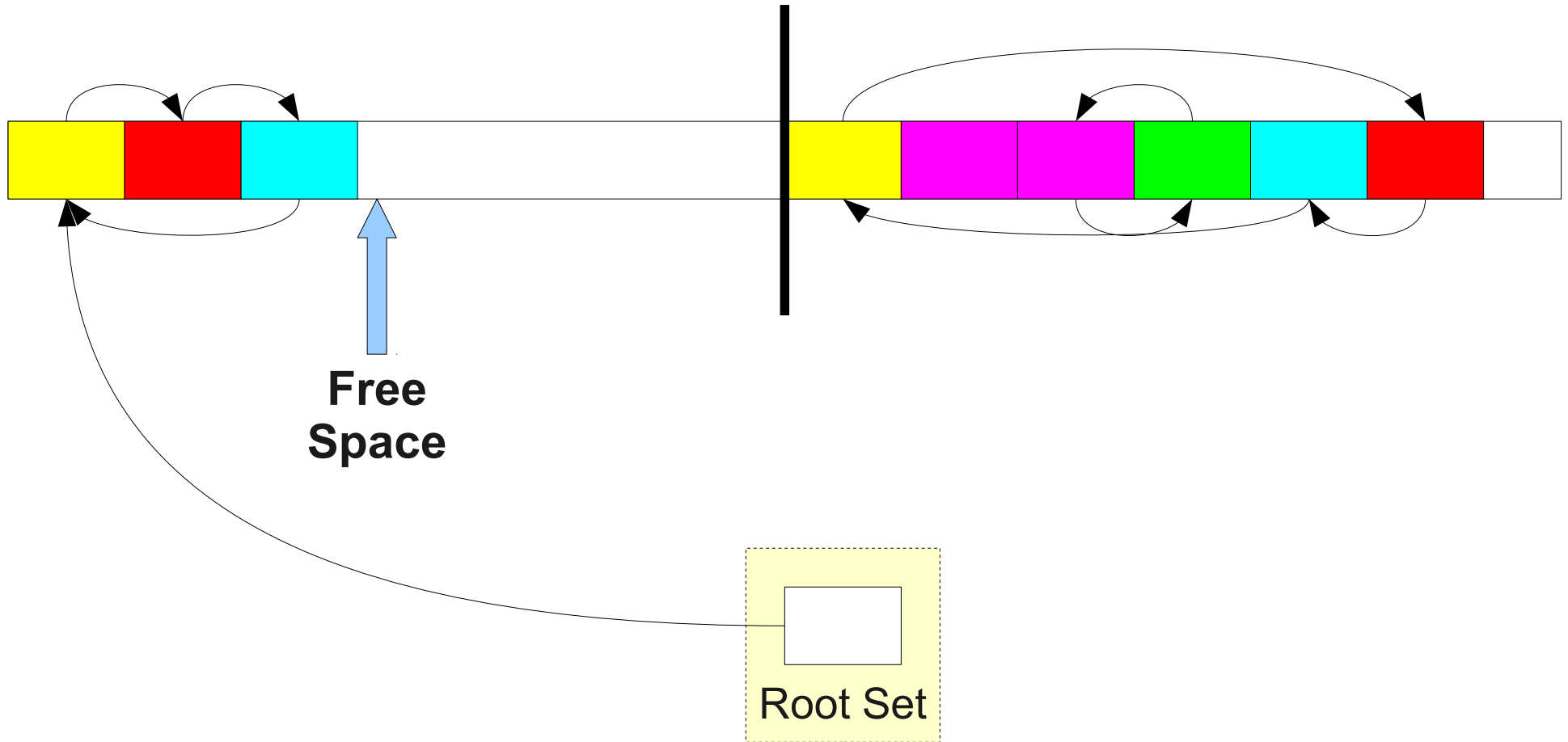
# The Stop-and-Copy Collector



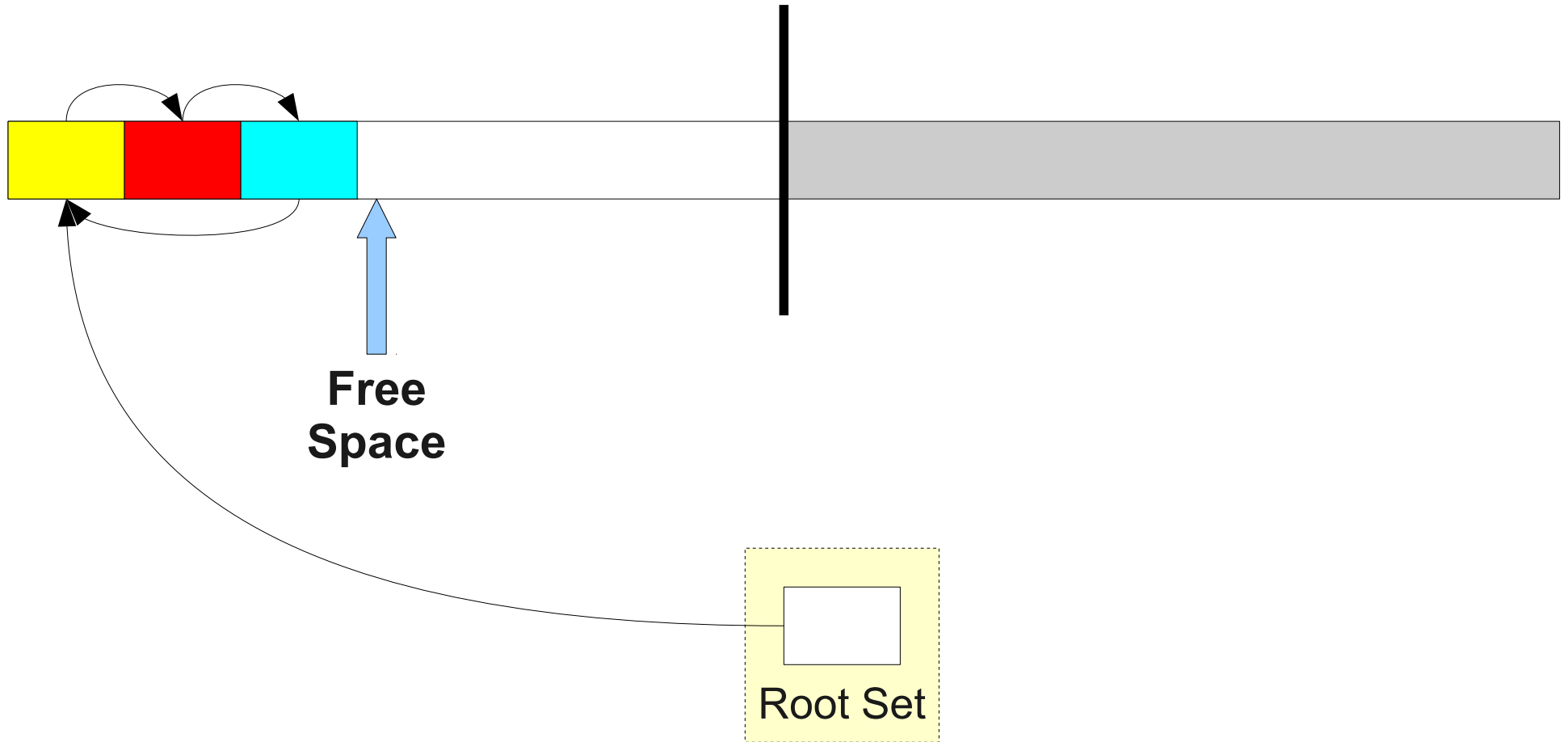
# The Stop-and-Copy Collector



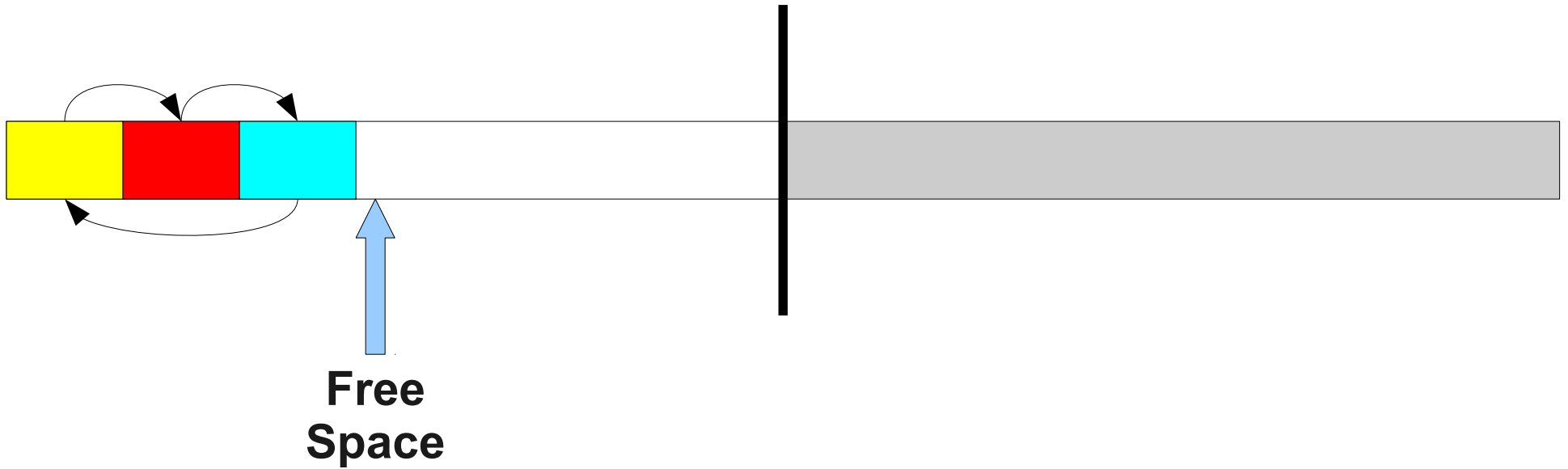
# The Stop-and-Copy Collector



# The Stop-and-Copy Collector



# The Stop-and-Copy Collector



# Stop-and-Copy in Detail

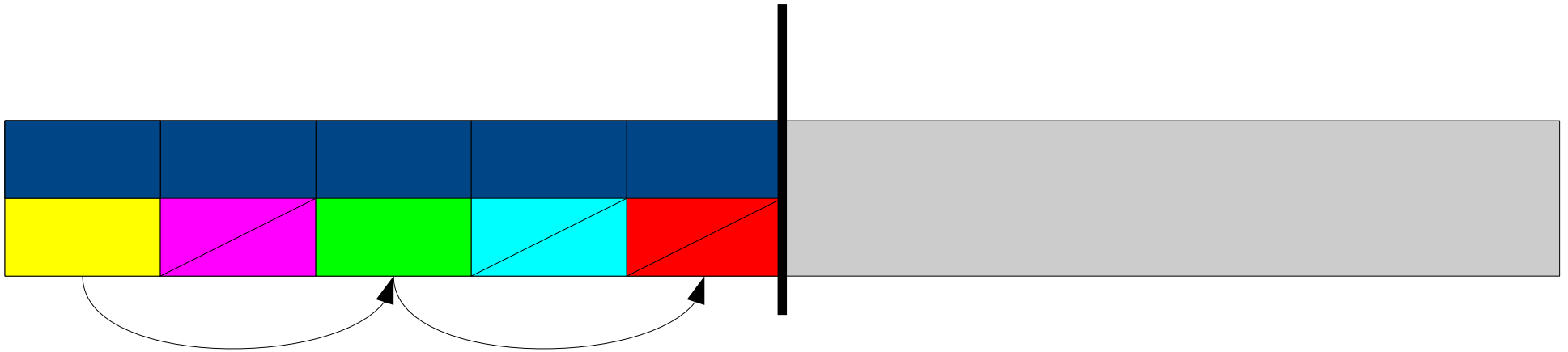
- Partition memory into two regions: the **old space** and the **new space**.
- Keep track of the next free address in the **new** space.
- To allocate **n** bytes of memory:
  - If **n** bytes space exist at the free space pointer, use those bytes and advance the pointer.
  - Otherwise, do a **copy** 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.

# Implementing Stop and Copy

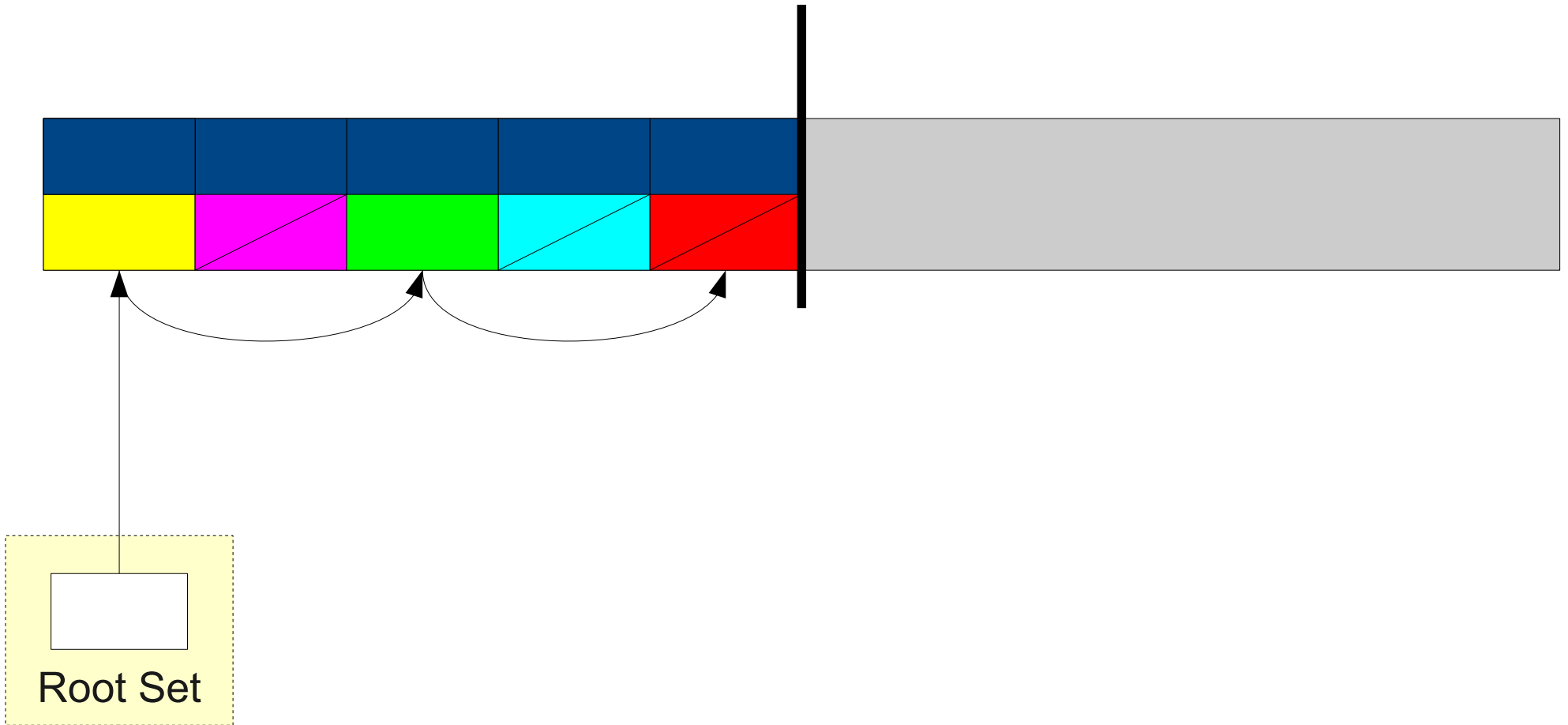
- The only tricky part about stop-and-copy is adjusting the pointers in the copied objects correctly.
- **Idea:** Have each object contain a extra space for a **forwarding pointer**.
- To clone an object:
  - First, do a complete bitwise copy of the object.
    - All pointers still point to their original locations.
  - Next, set the **forwarding pointer** of the original object to point to the new object.
- Finally, after cloning each object, for each pointer:
  - Follow the pointer to the object it references.
  - Replace the pointer with the pointee's forwarding pointer.



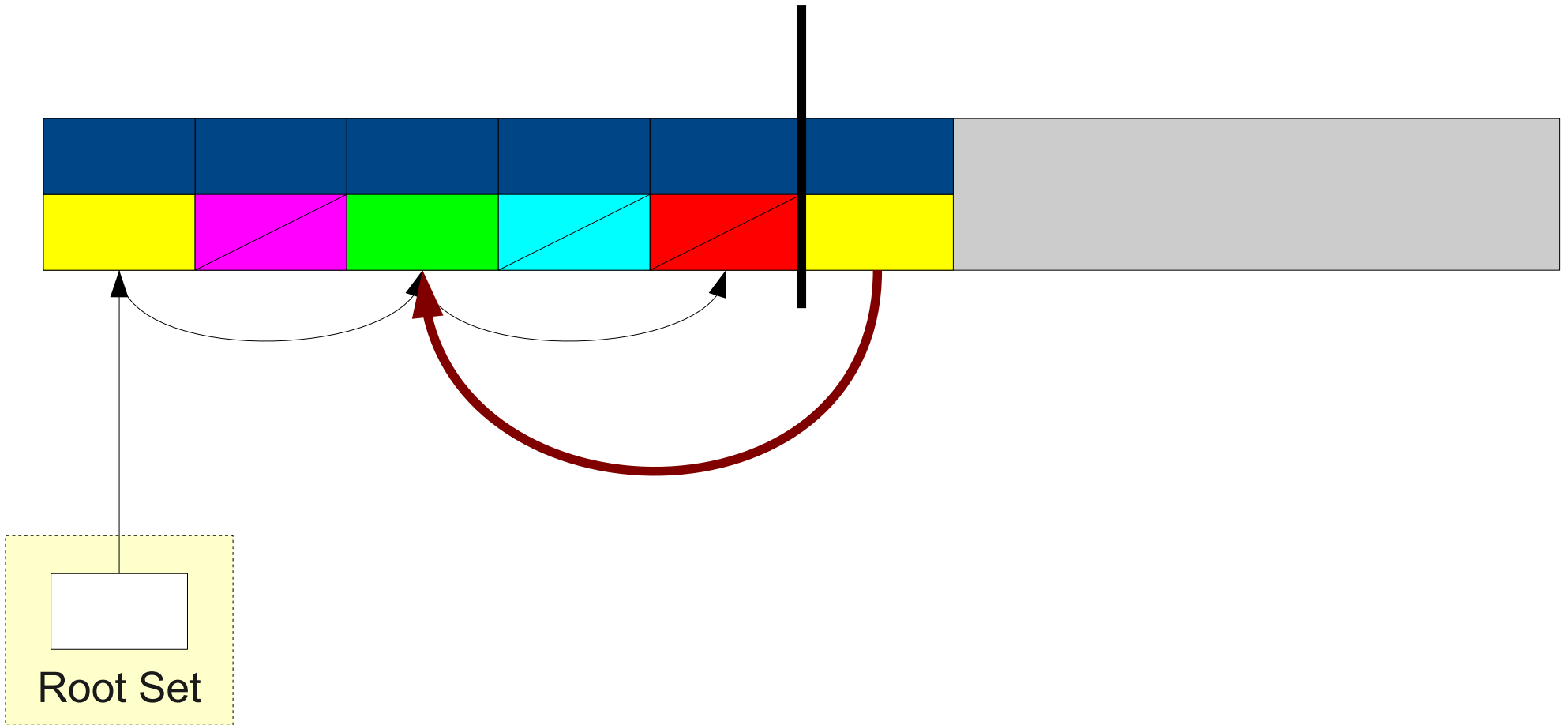
# Forwarding Pointers



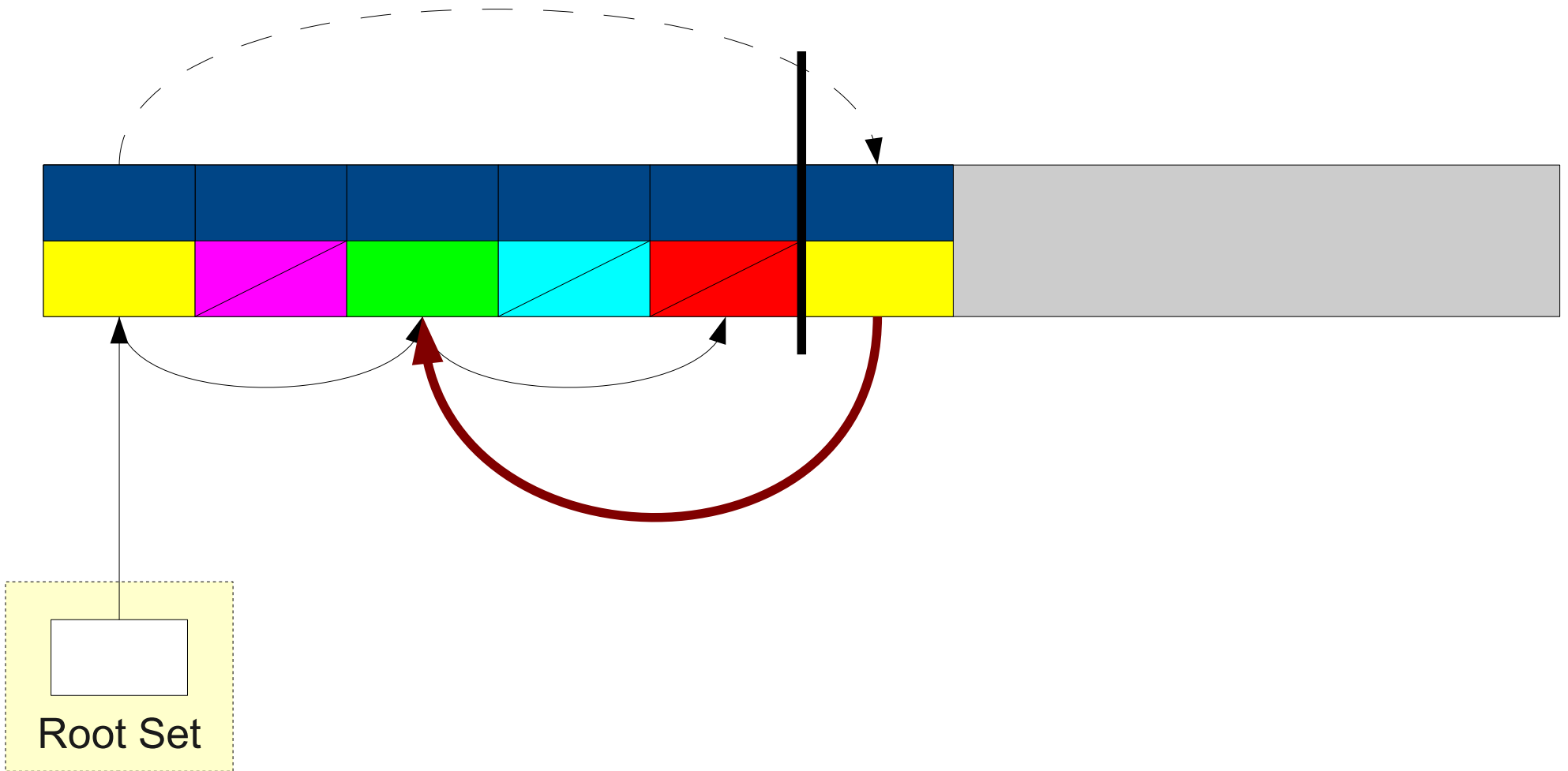
# Forwarding Pointers



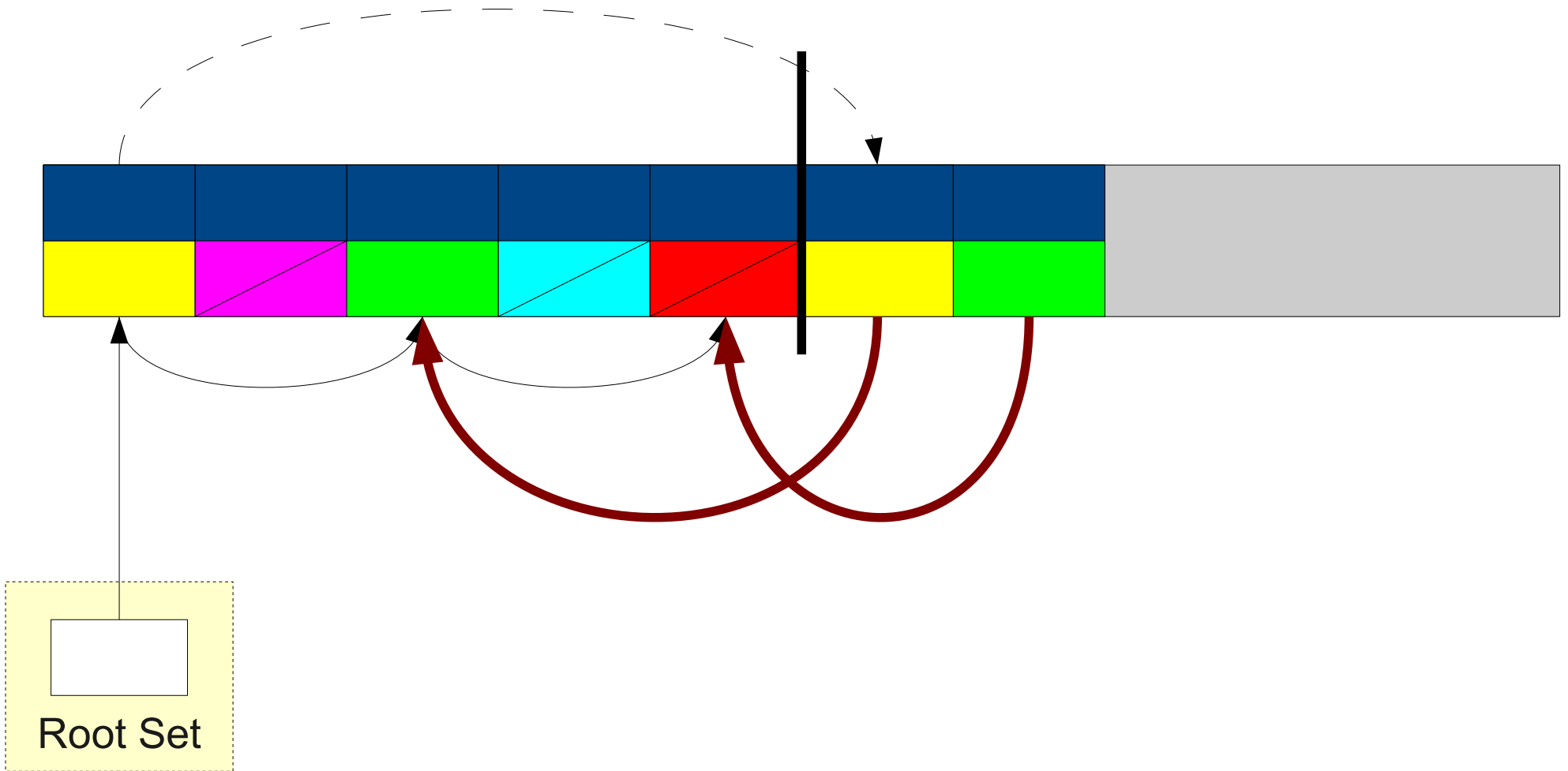
# Forwarding Pointers



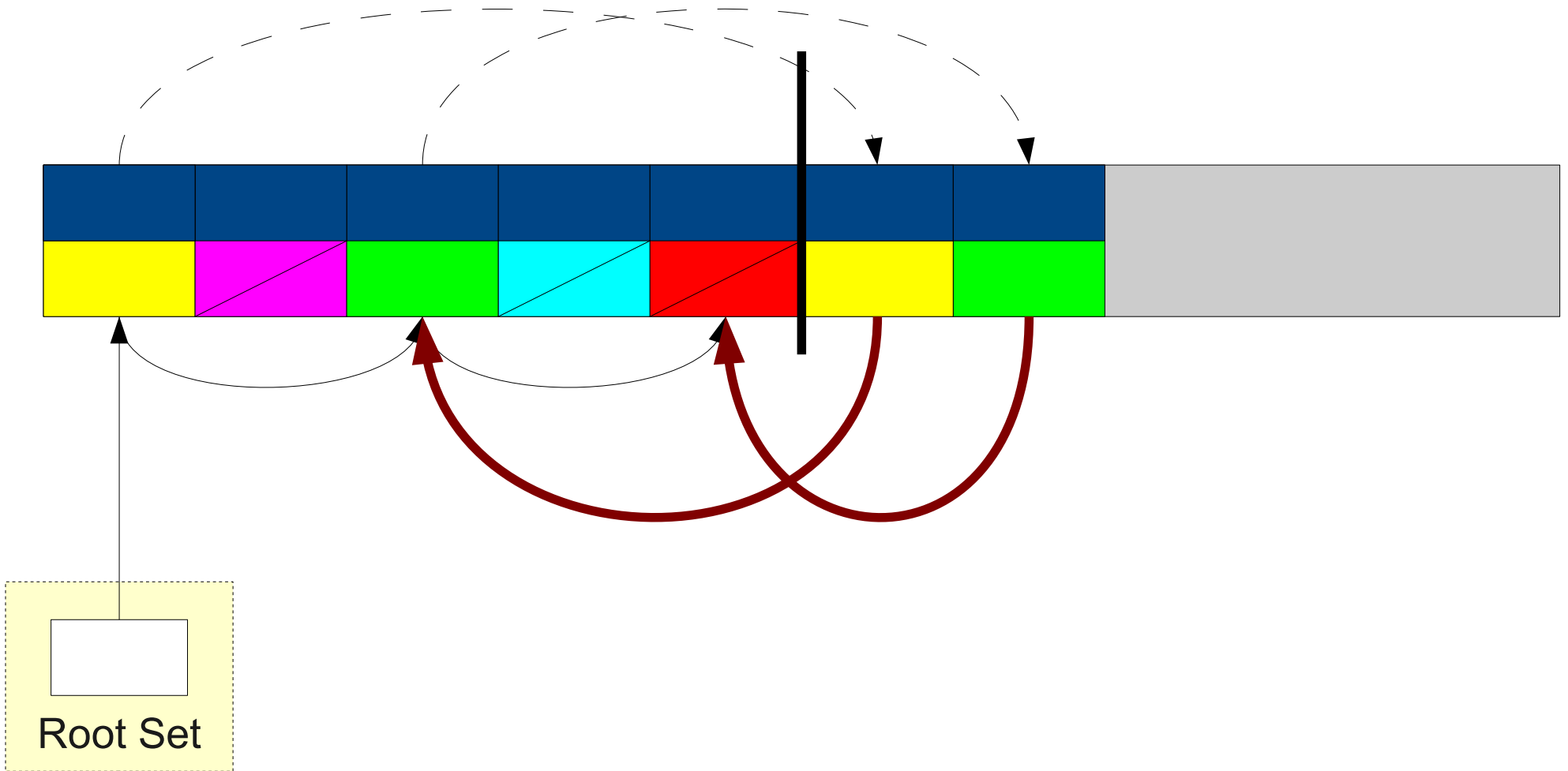
# Forwarding Pointers



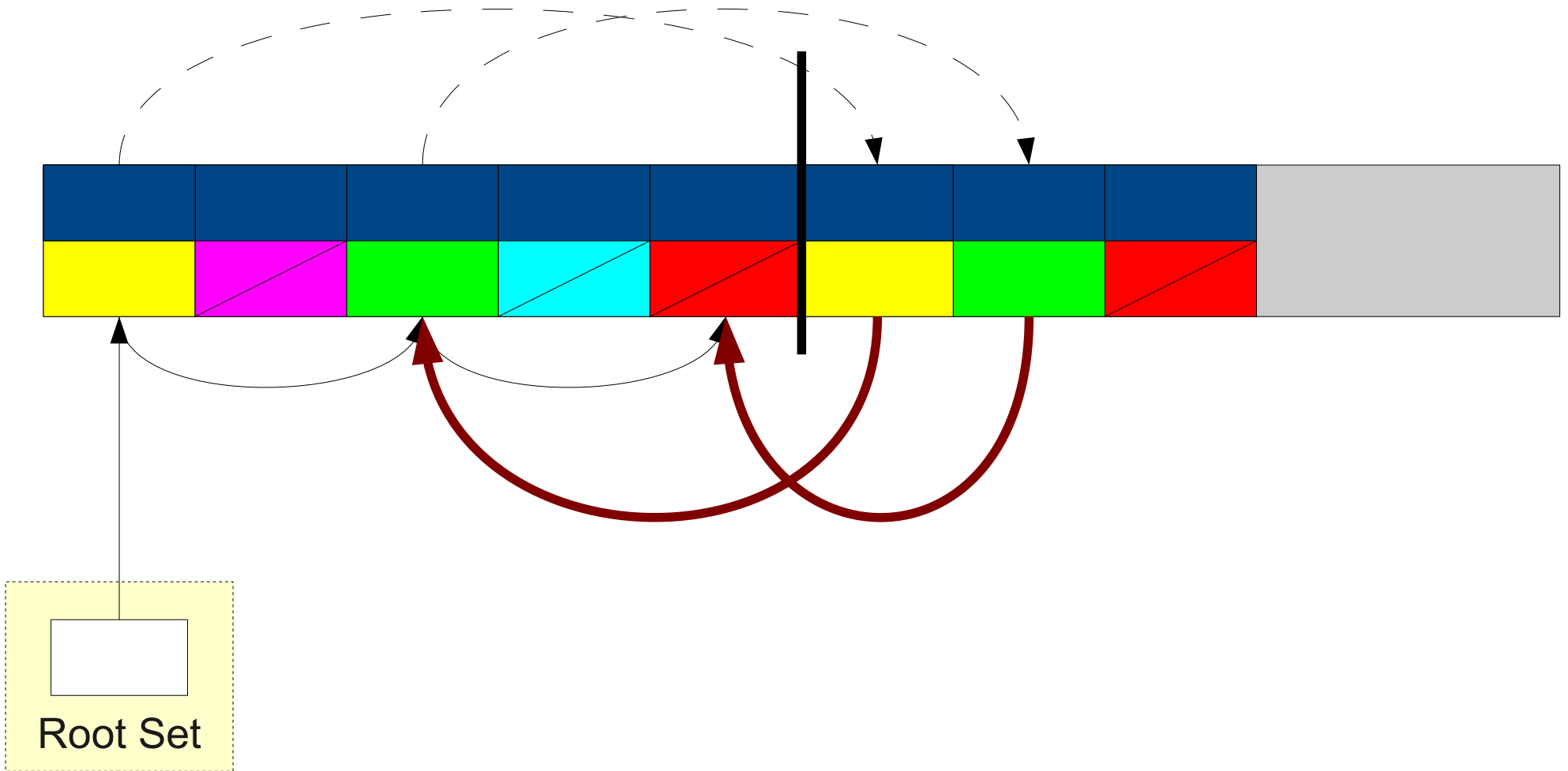
# Forwarding Pointers



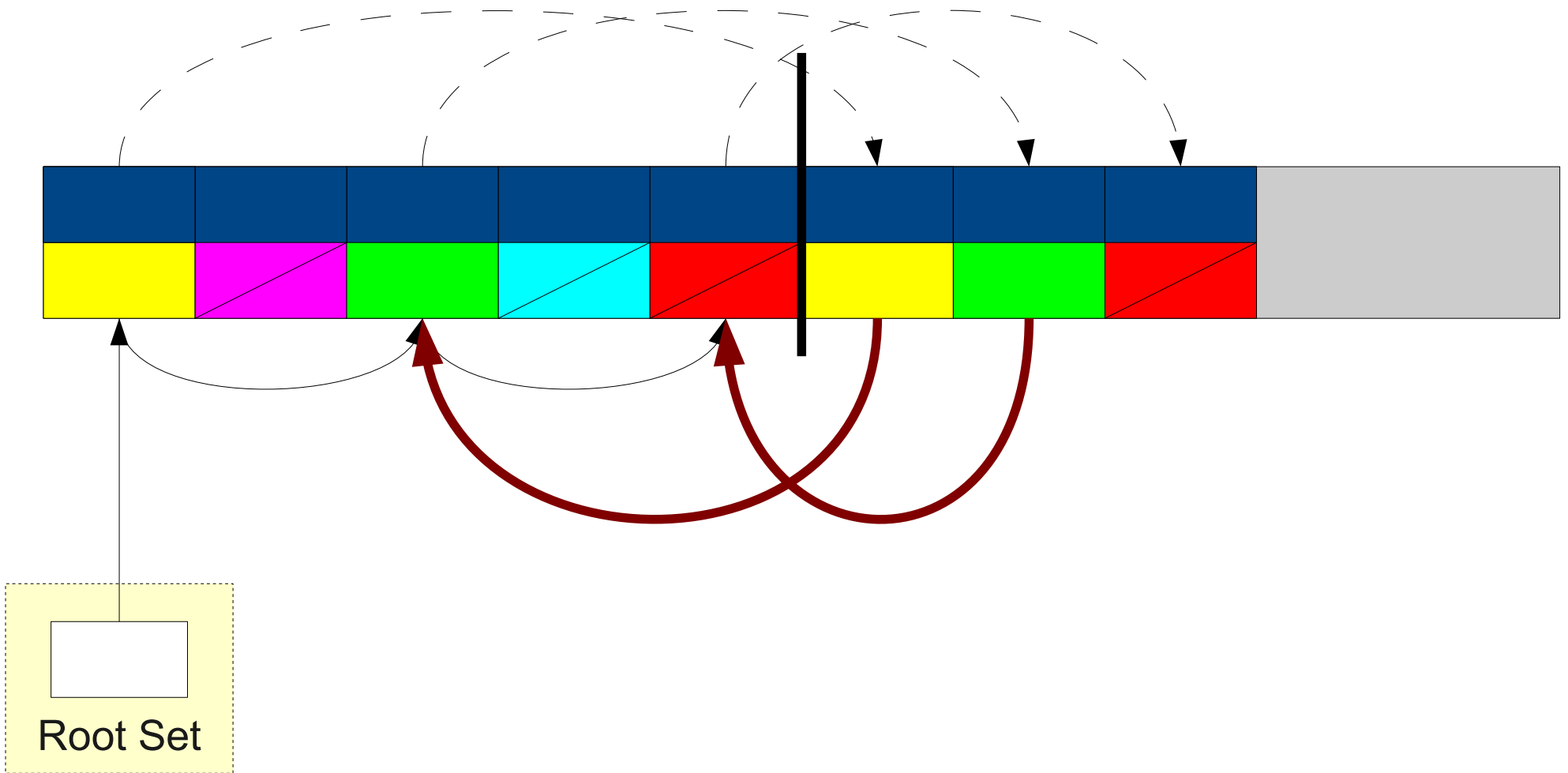
# Forwarding Pointers



# Forwarding Pointers

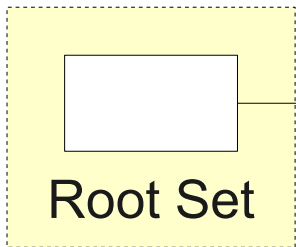
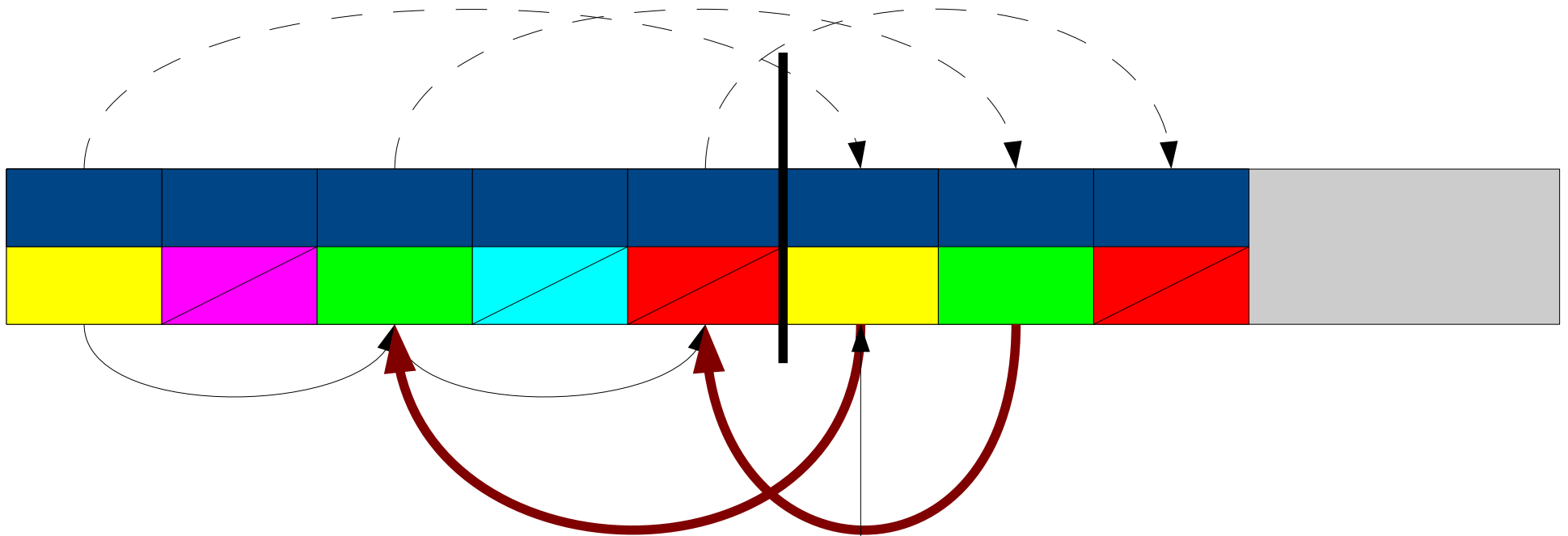


# Forwarding Pointers

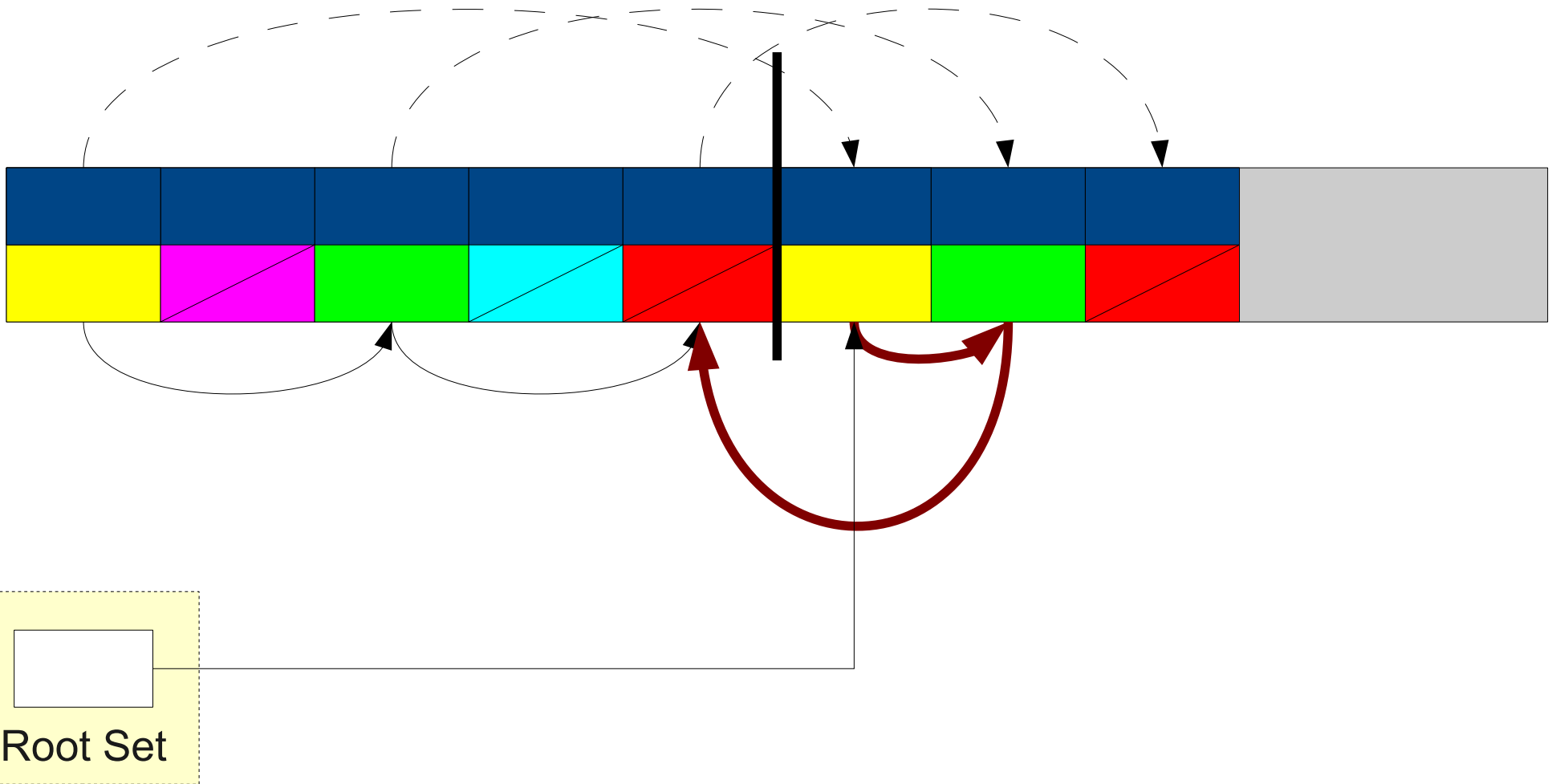




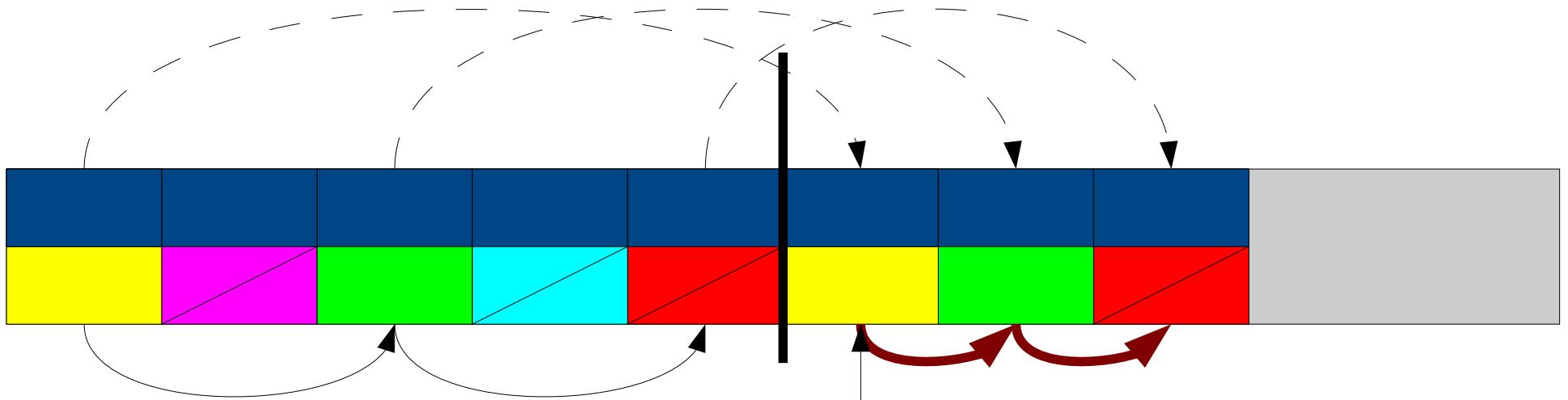
# Forwarding Pointers



# Forwarding Pointers

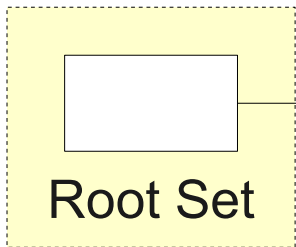
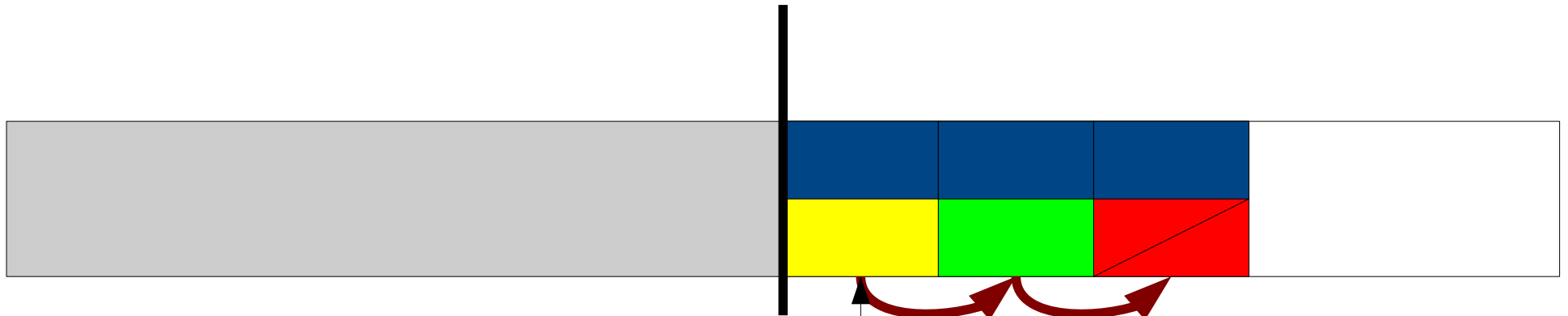


# Forwarding Pointers



Root Set

# Forwarding Pointers



# Analysis of Stop-and-Copy

- Advantages:
  - Implementation simplicity (compared to mark-and-sweep).
  - Fast memory allocation; using OS-level tricks, can allocate in a single assembly instruction.
  - Excellent locality; depth-first ordering of copied objects places similar objects near each other.
- Disadvantages:
  - Requires half of memory to be free at all times.
  - Collection time proportional to number of bytes used by objects.

# Hybrid Approaches

# The Best of All Worlds

- The best garbage collectors in use today are based on a combination of smaller garbage collectors.
- Each garbage collector is targeted to reclaim specific types of garbage.
- Usually has some final “fallback” garbage collector to handle everything else.

# Objects Die Young

- The Motto of Garbage Collection: **Objects Die Young.**
- Most objects have extremely short lifetimes.
  - Objects allocated locally in a function.
  - Temporary objects used to construct larger objects.
- Optimize garbage collection to reclaim young objects rapidly while spending less time on older objects.



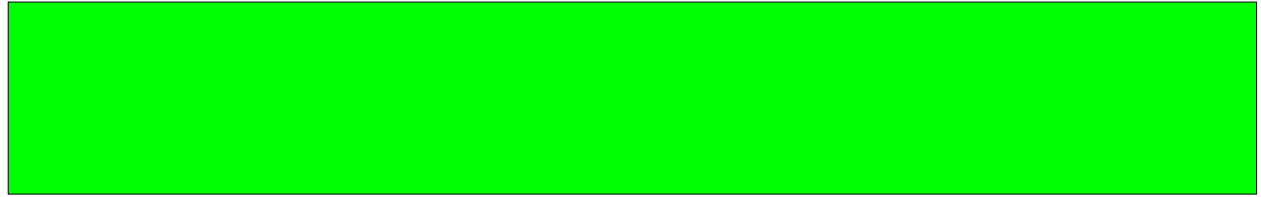
# Generational Garbage Collection

- 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.

# Garbage Collection in Java

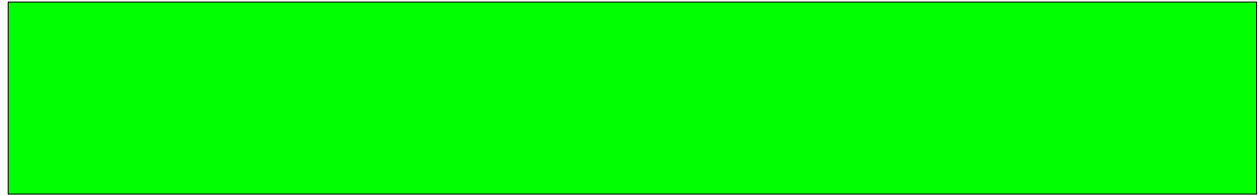
# Garbage Collection in Java

Eden

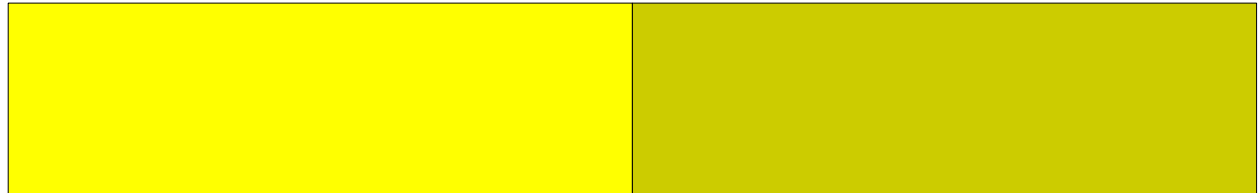


# Garbage Collection in Java

Eden

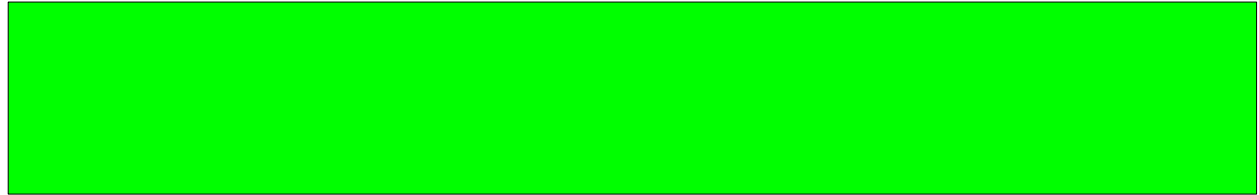


Survivor Objects

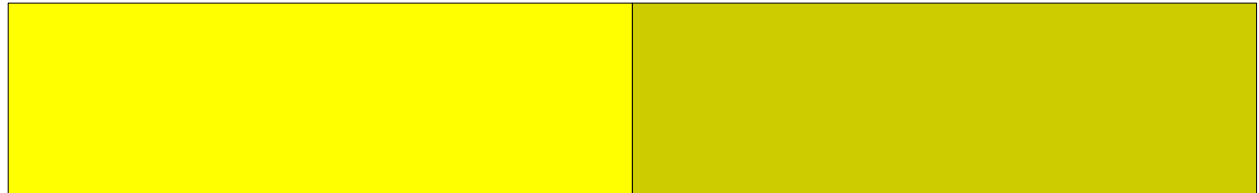


# Garbage Collection in Java

Eden



Survivor Objects

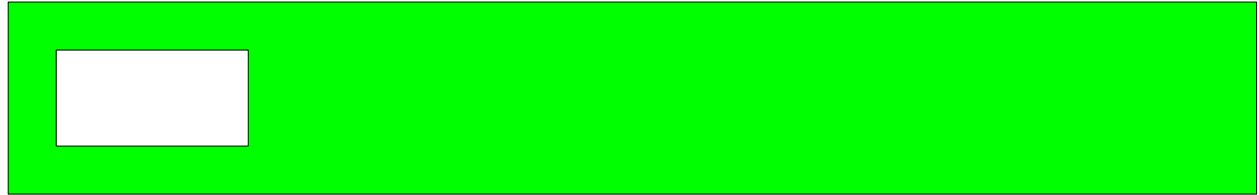


Tenured Objects

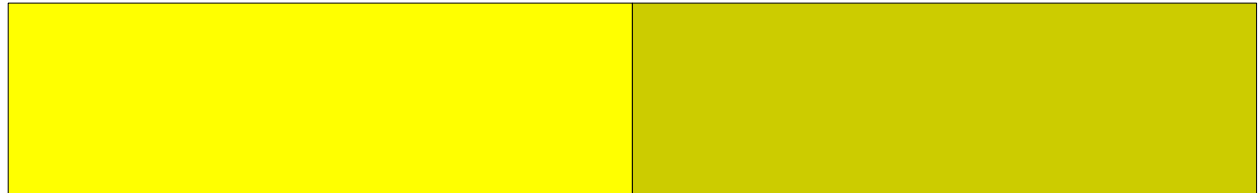


# Garbage Collection in Java

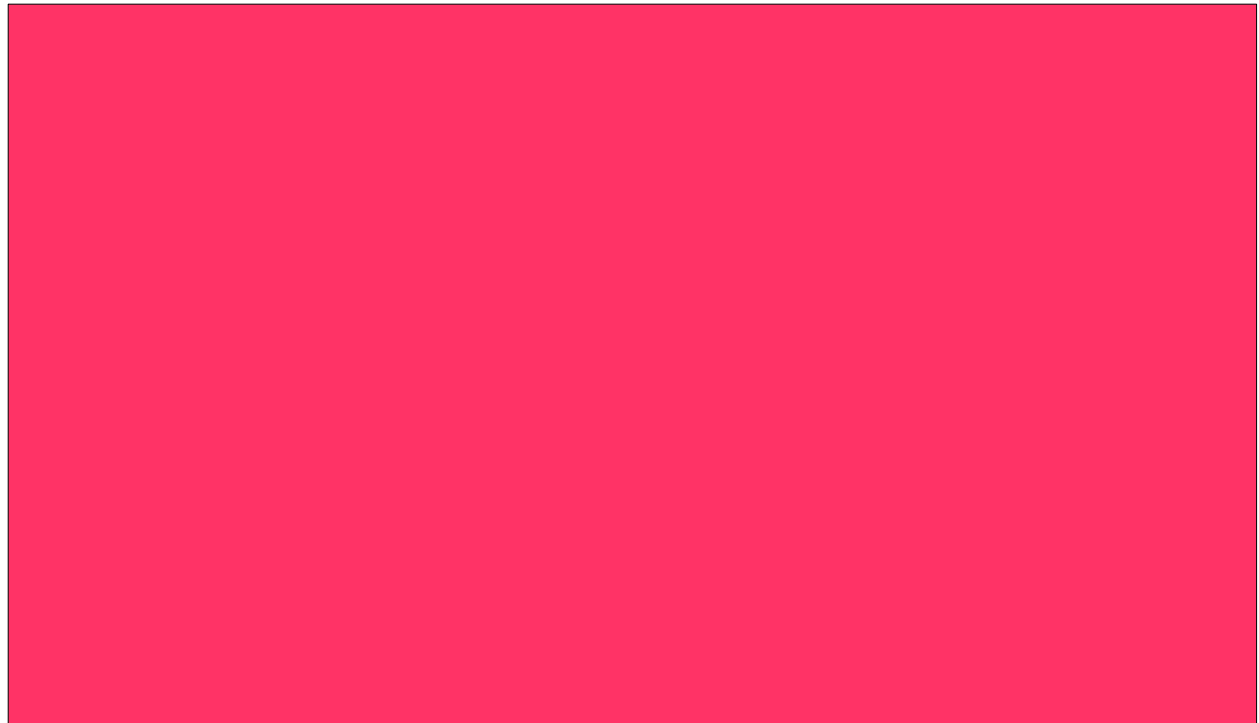
Eden



Survivor Objects

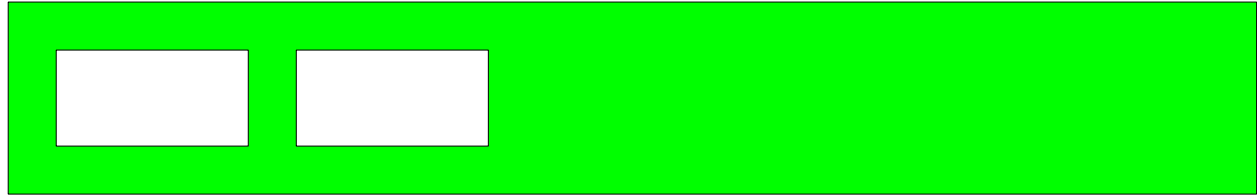


Tenured Objects

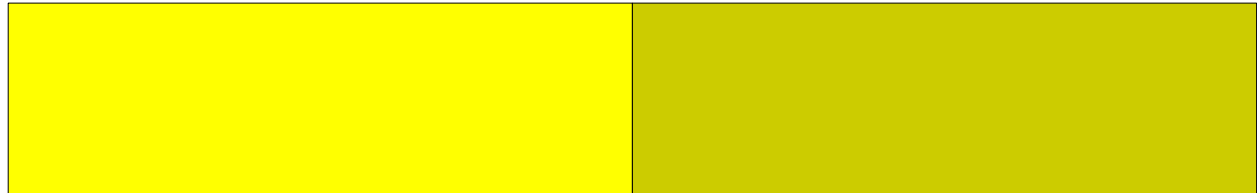


# Garbage Collection in Java

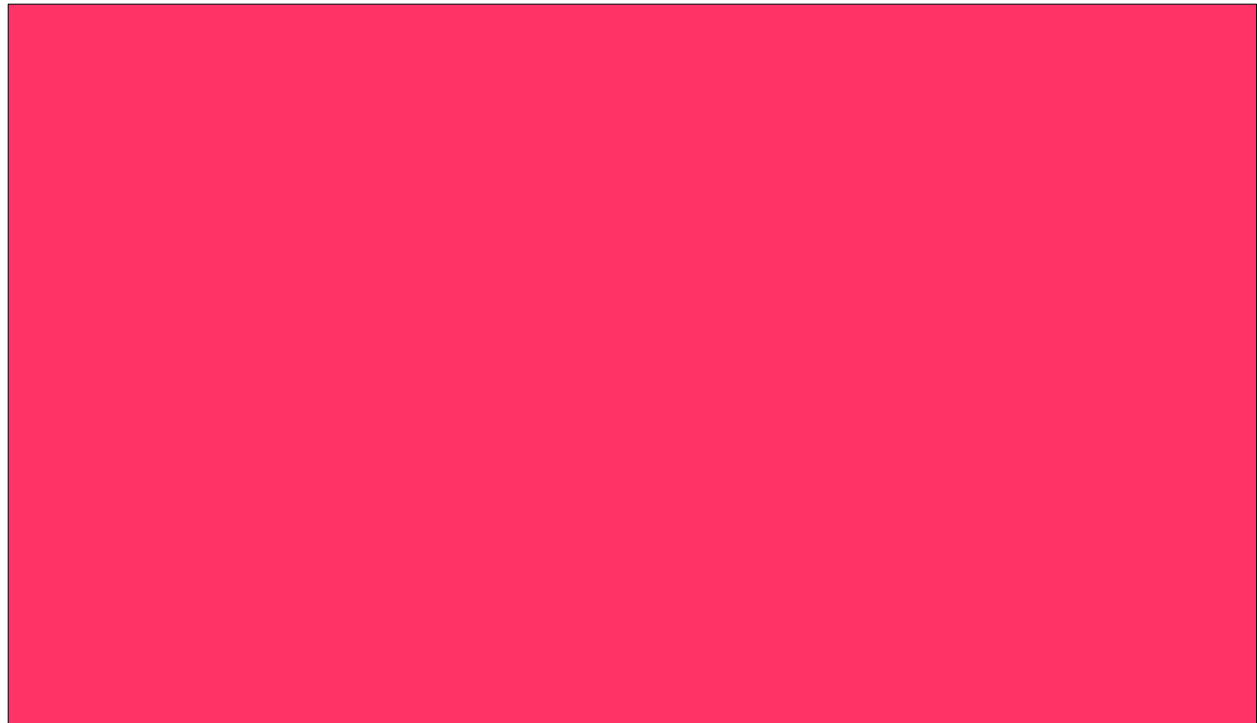
Eden



Survivor Objects

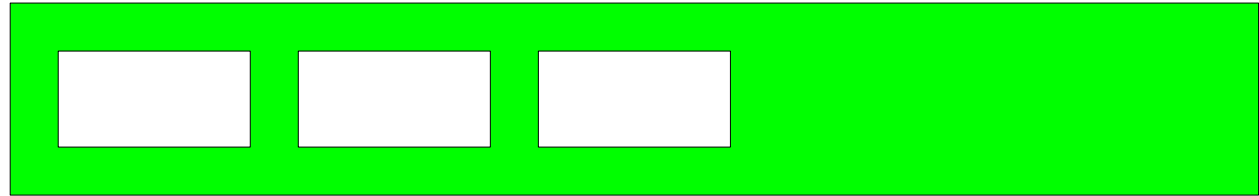


Tenured Objects

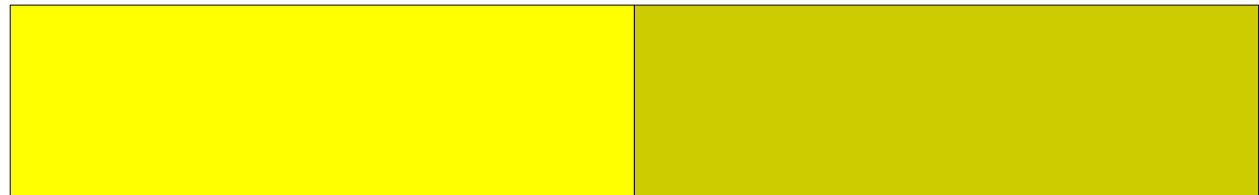


# Garbage Collection in Java

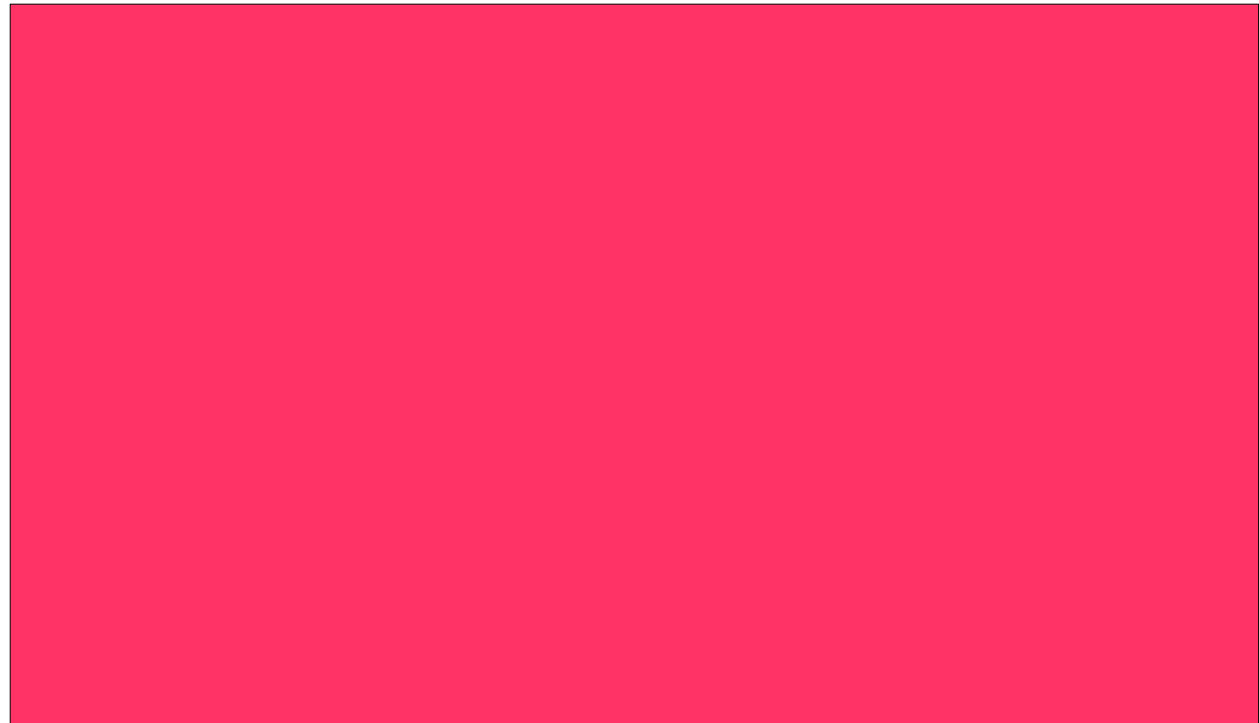
Eden



Survivor Objects



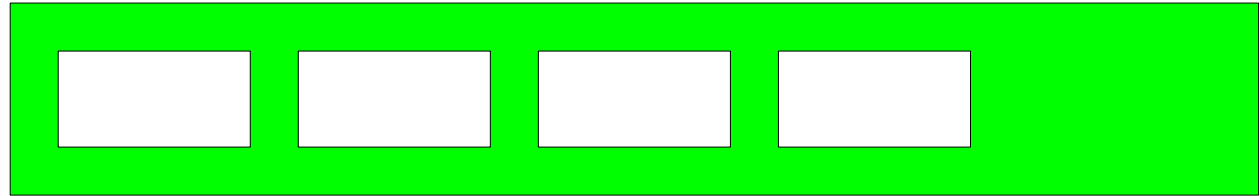
Tenured Objects



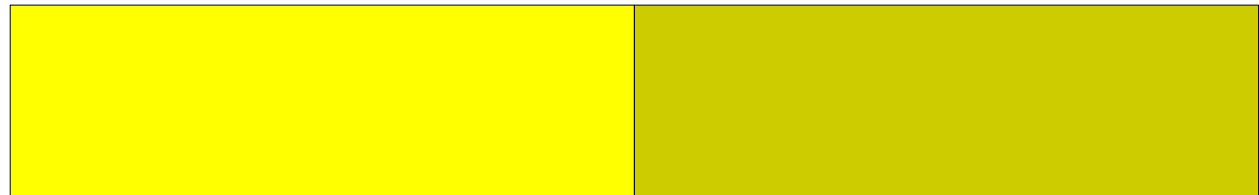


# Garbage Collection in Java

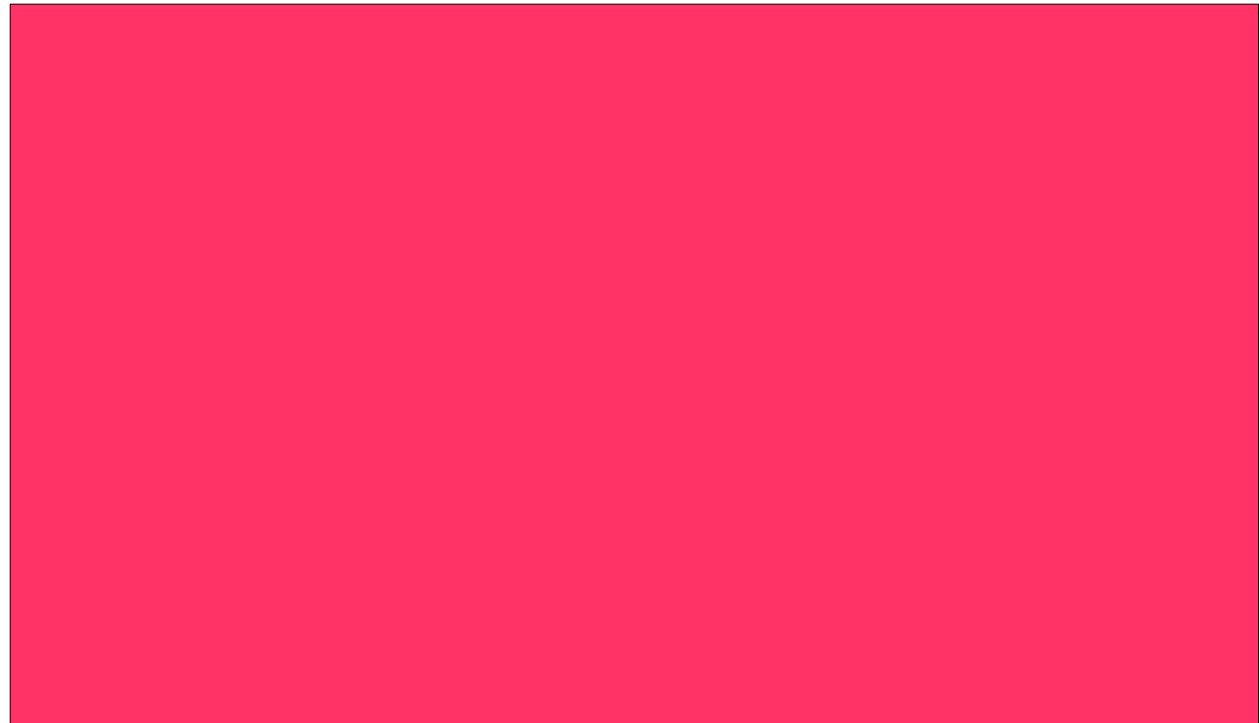
Eden



Survivor Objects

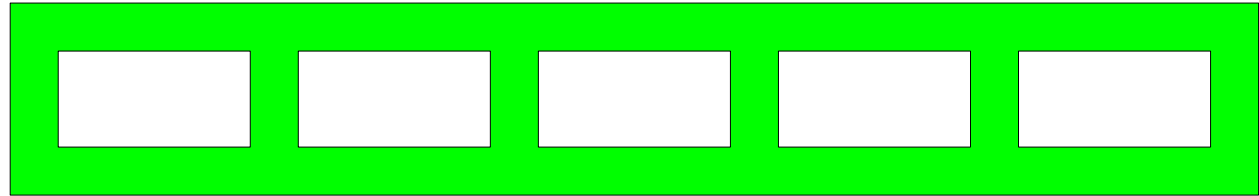


Tenured Objects

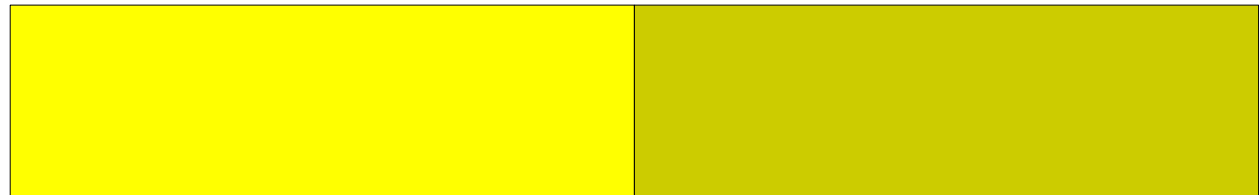


# Garbage Collection in Java

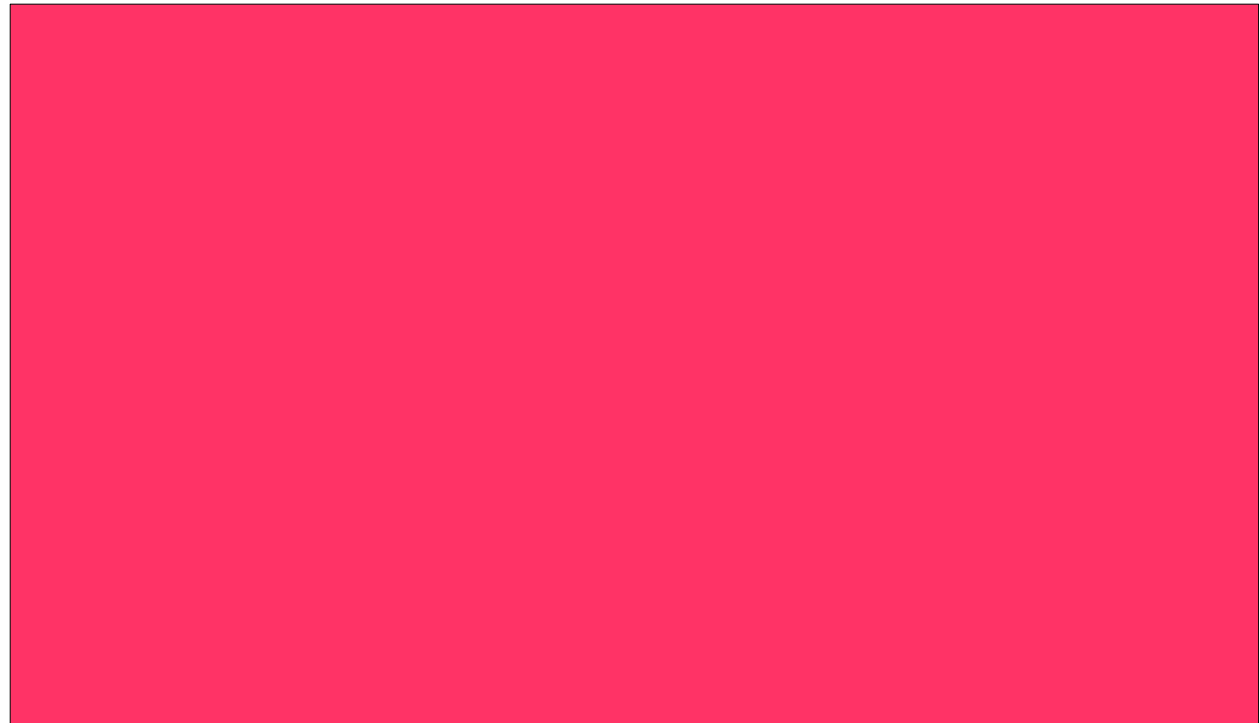
Eden



Survivor Objects

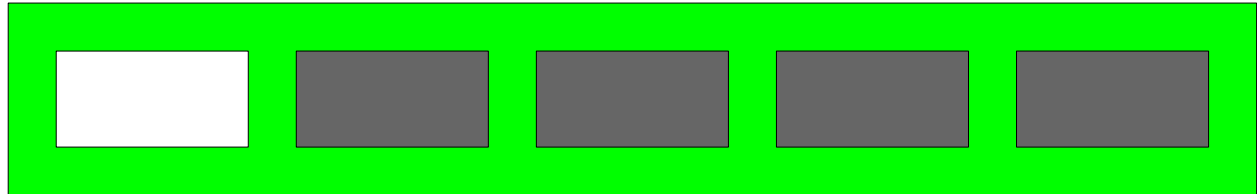


Tenured Objects

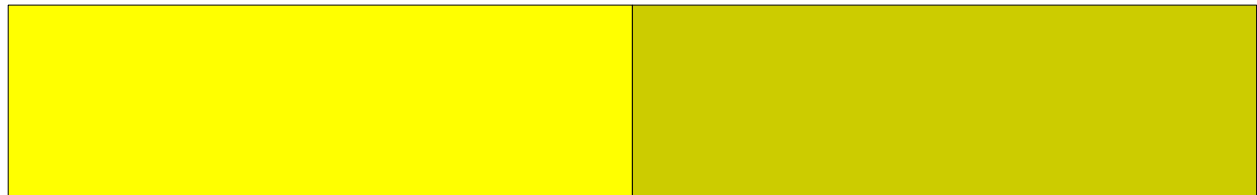


# Garbage Collection in Java

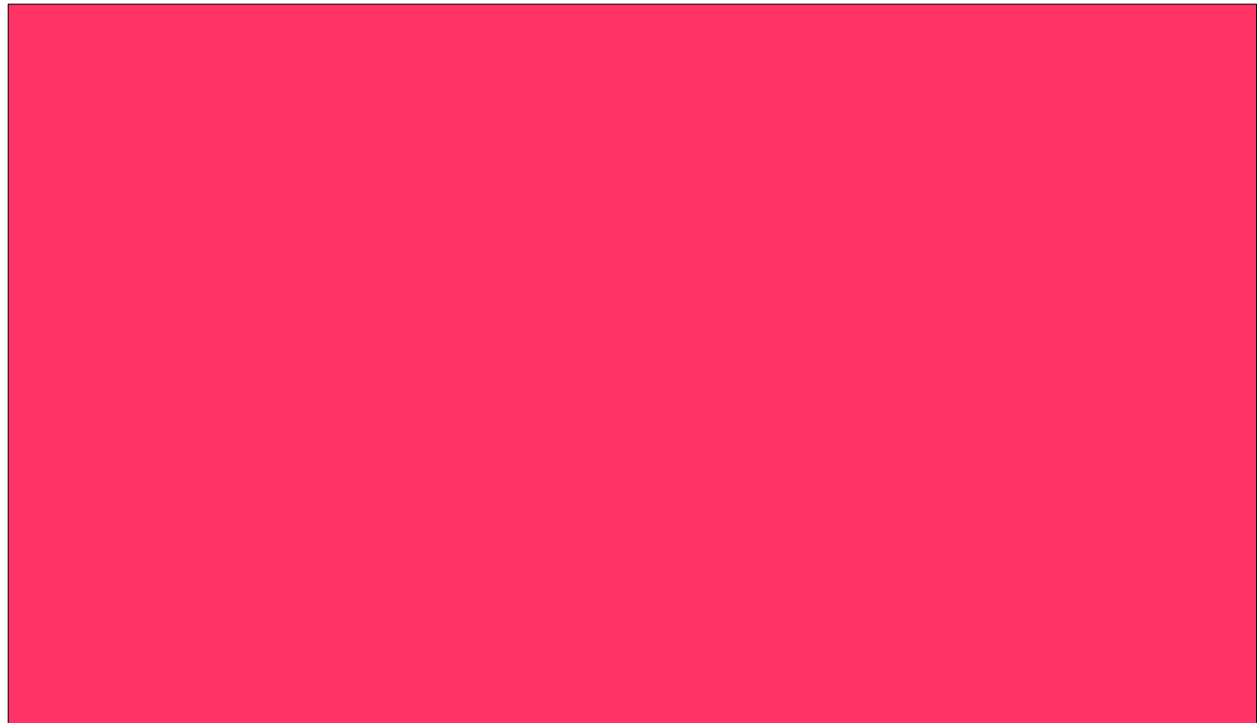
Eden



Survivor Objects

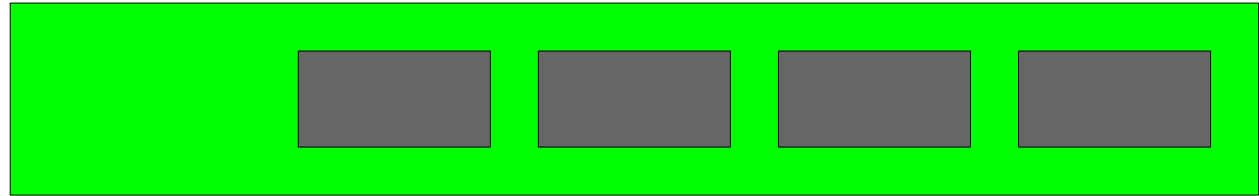


Tenured Objects

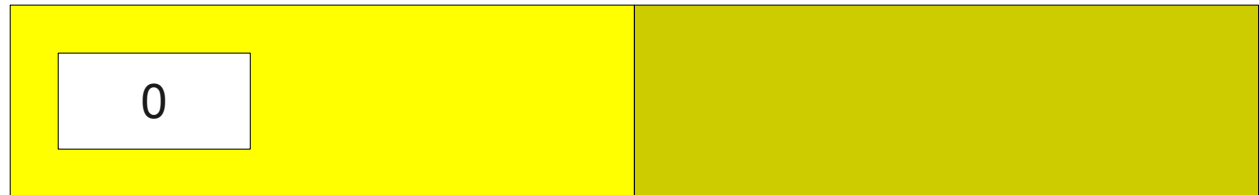


# Garbage Collection in Java

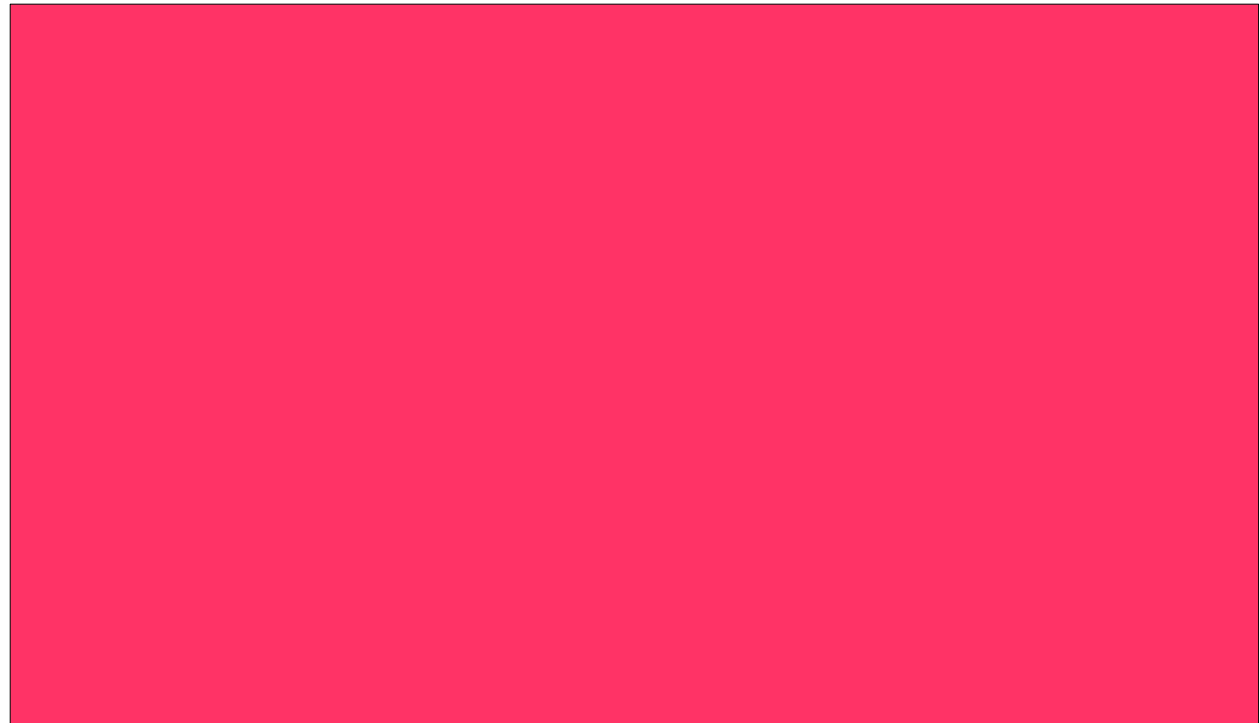
Eden



Survivor Objects

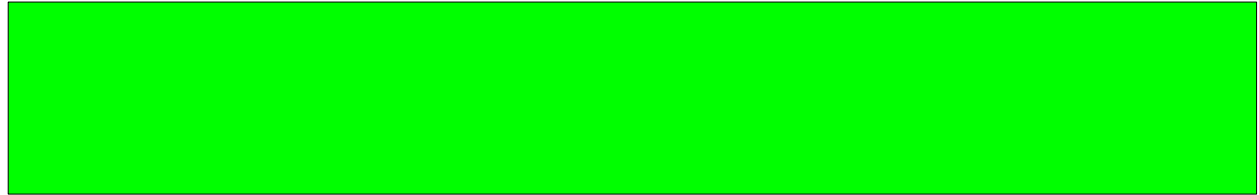


Tenured Objects

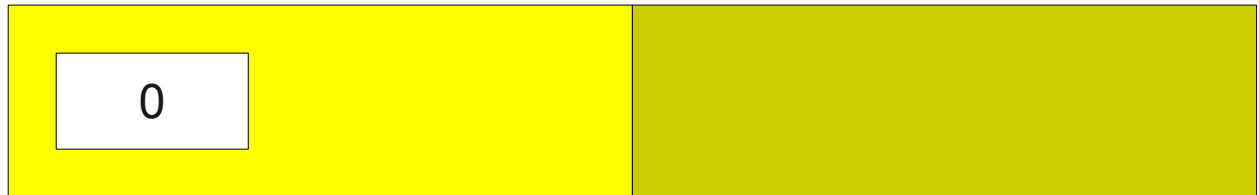


# Garbage Collection in Java

Eden



Survivor Objects

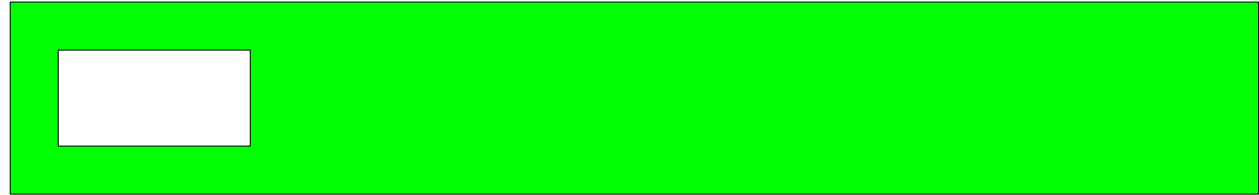


Tenured Objects

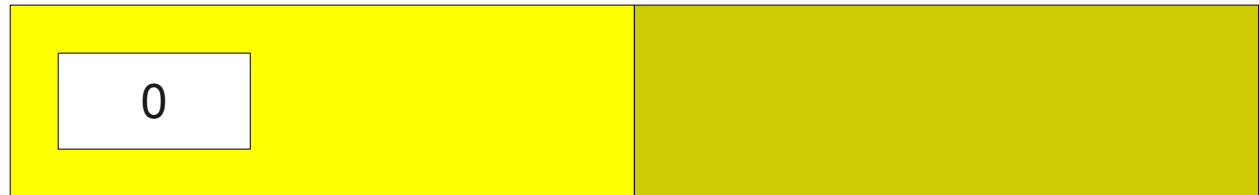


# Garbage Collection in Java

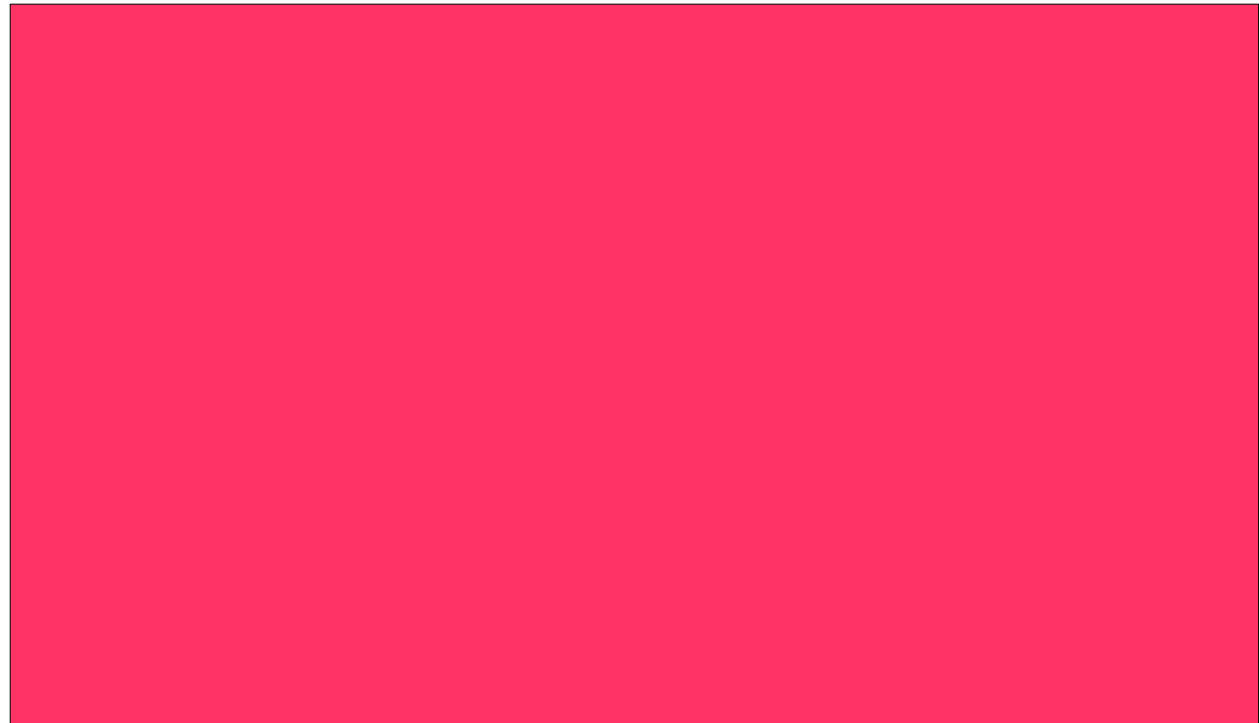
Eden



Survivor Objects

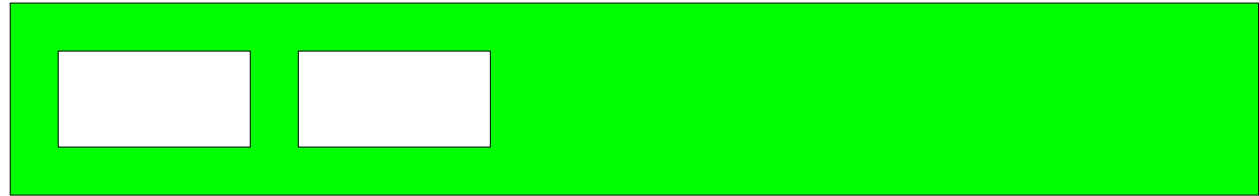


Tenured Objects

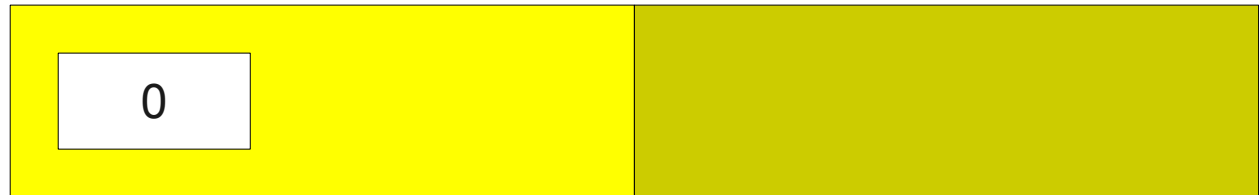


# Garbage Collection in Java

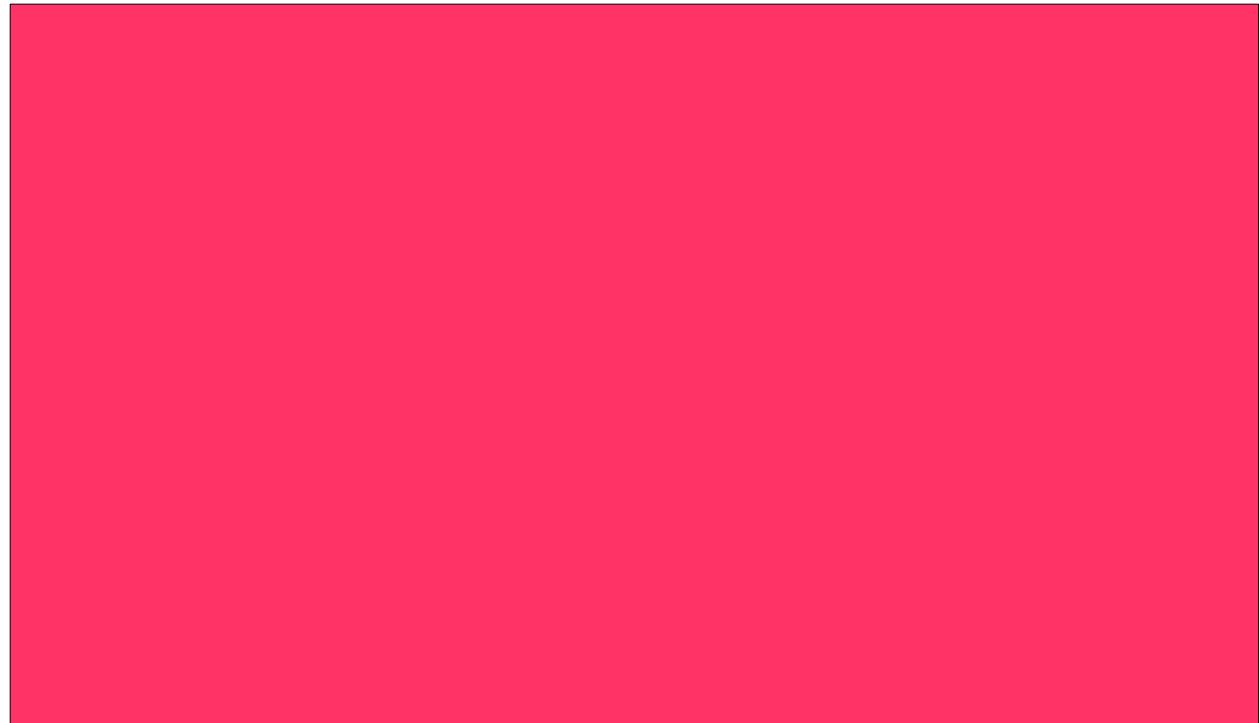
Eden



Survivor Objects

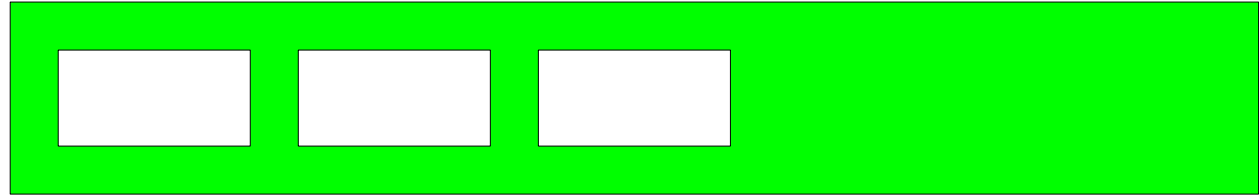


Tenured Objects

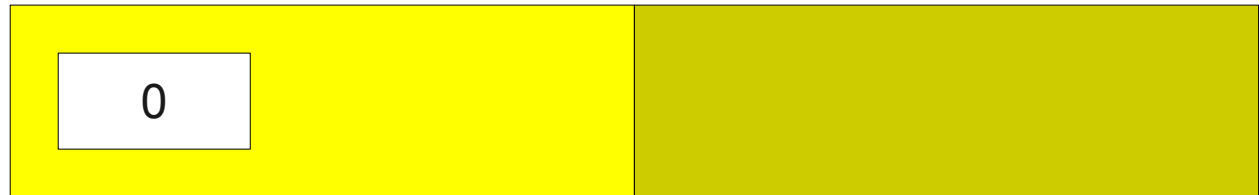


# Garbage Collection in Java

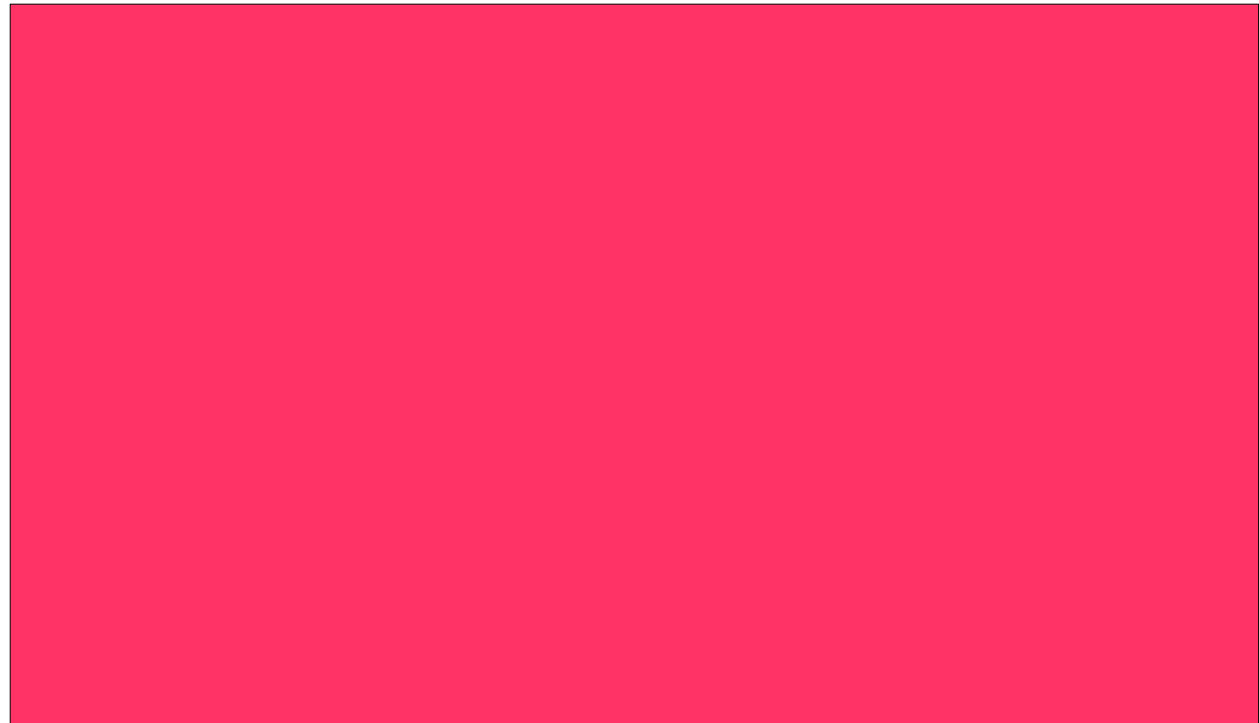
Eden



Survivor Objects



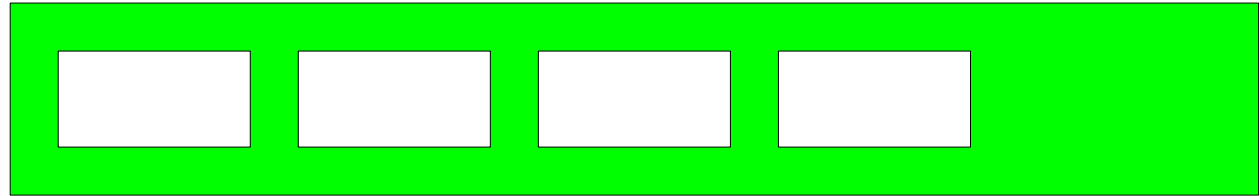
Tenured Objects



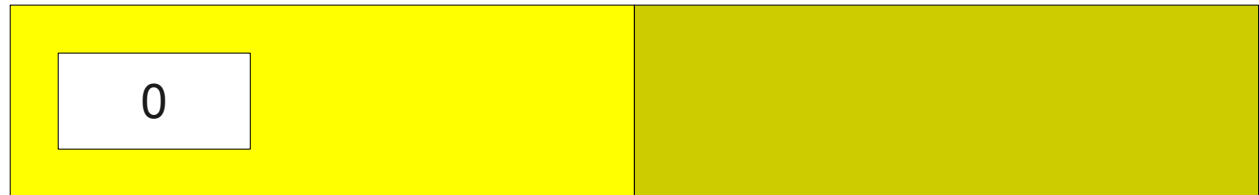


# Garbage Collection in Java

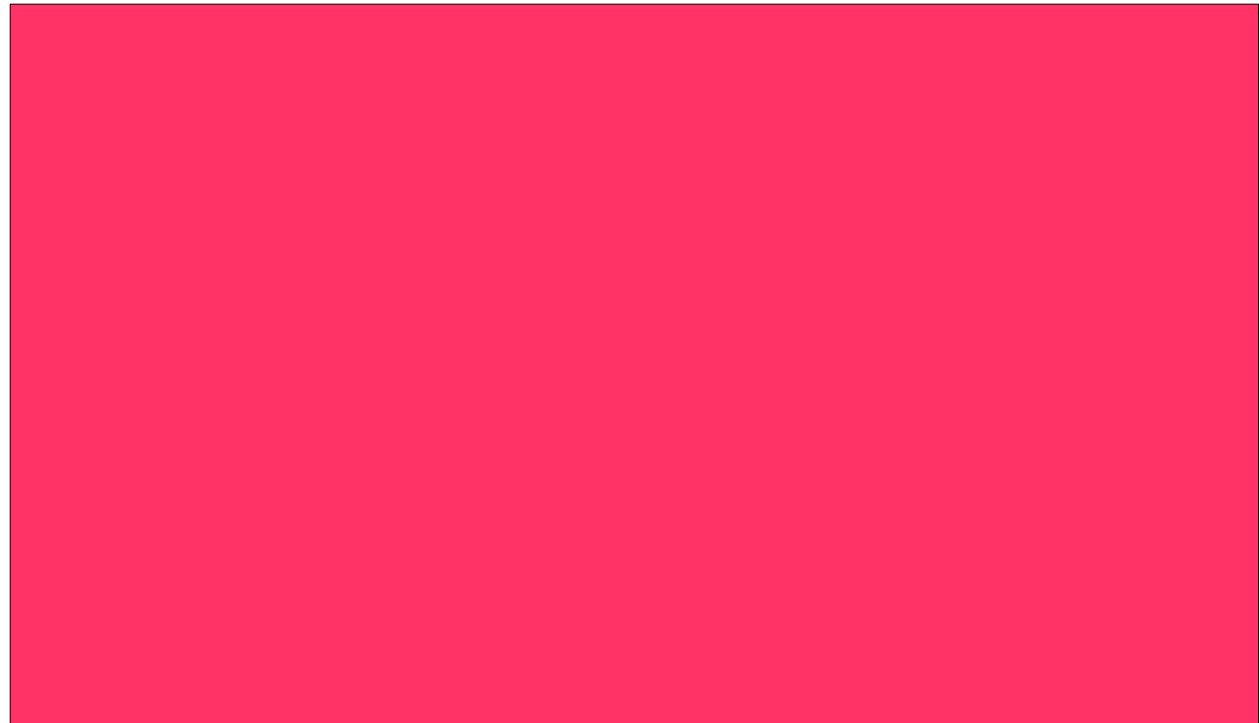
Eden



Survivor Objects

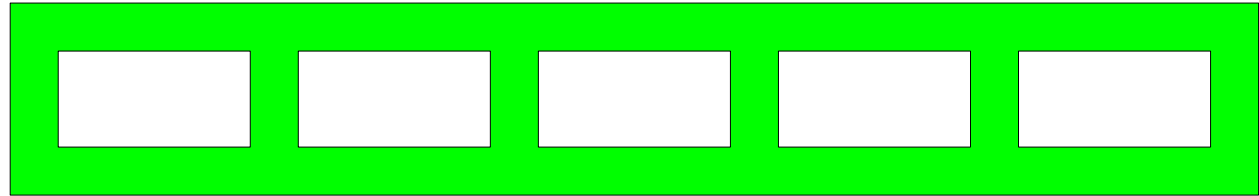


Tenured Objects

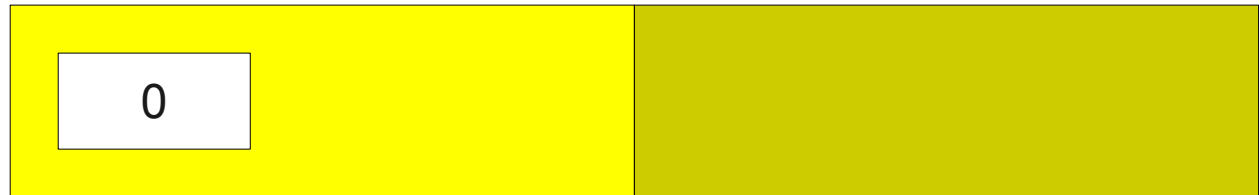


# Garbage Collection in Java

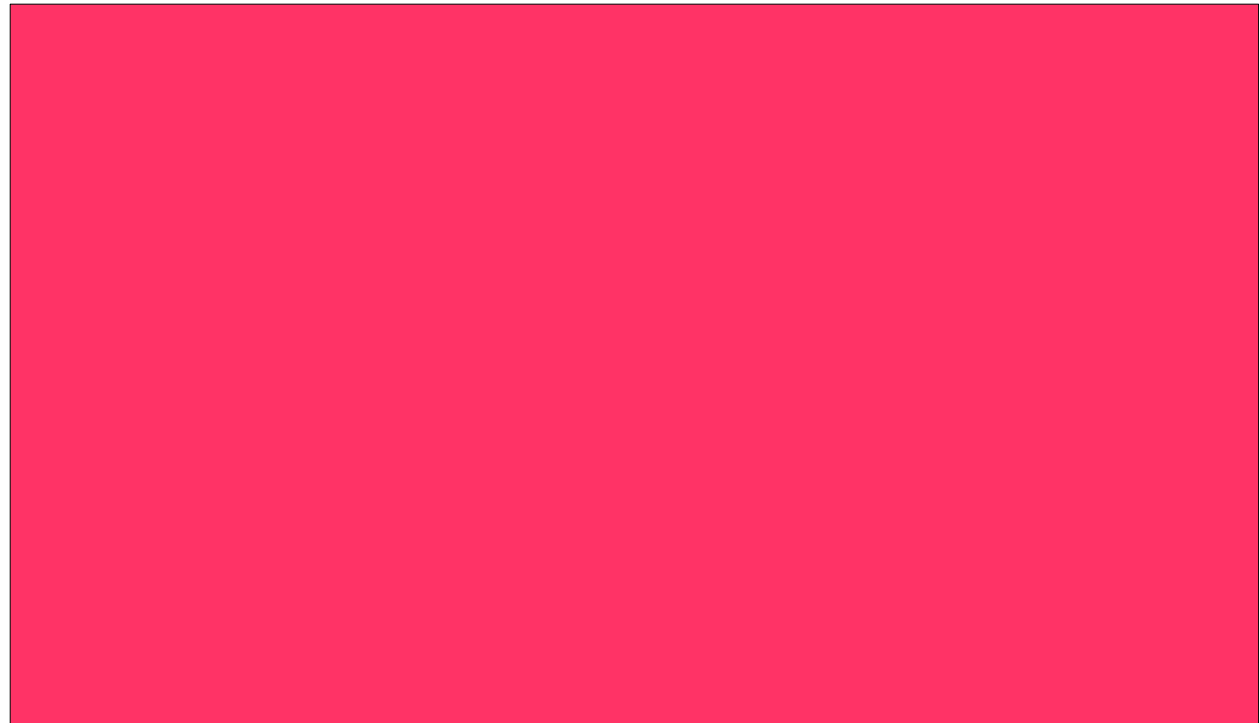
Eden



Survivor Objects

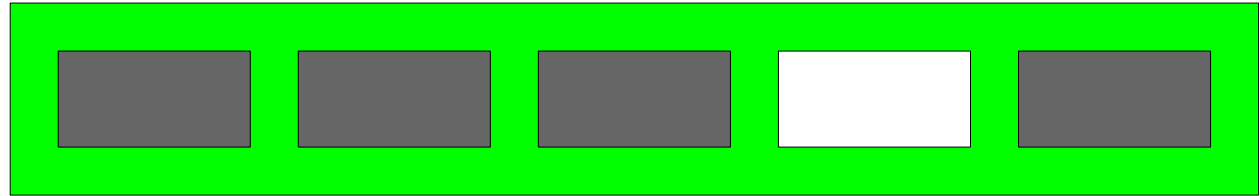


Tenured Objects

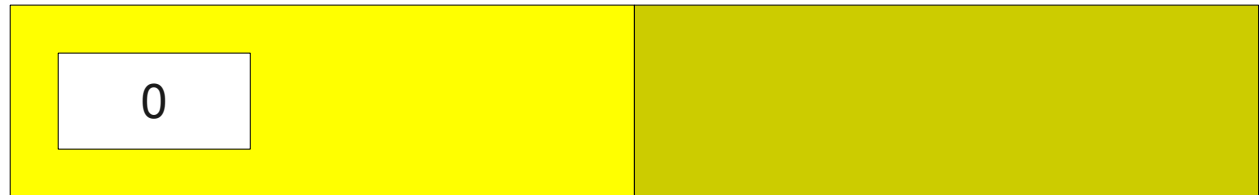


# Garbage Collection in Java

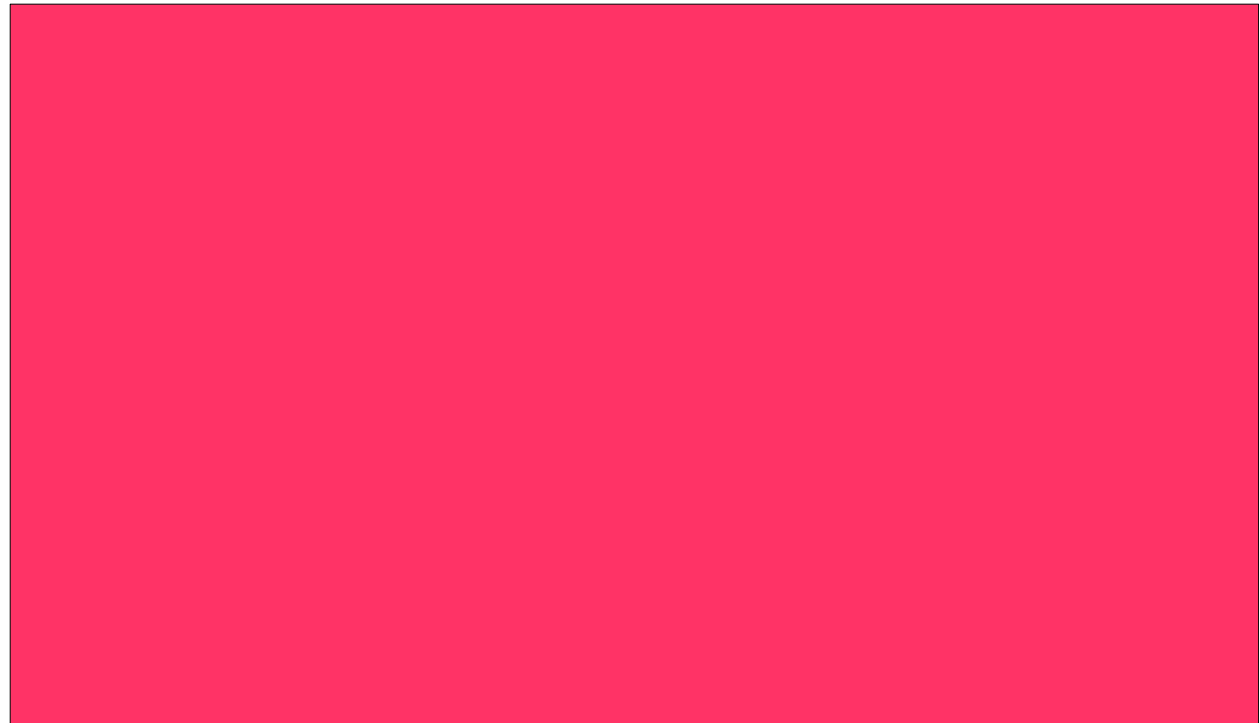
Eden



Survivor Objects

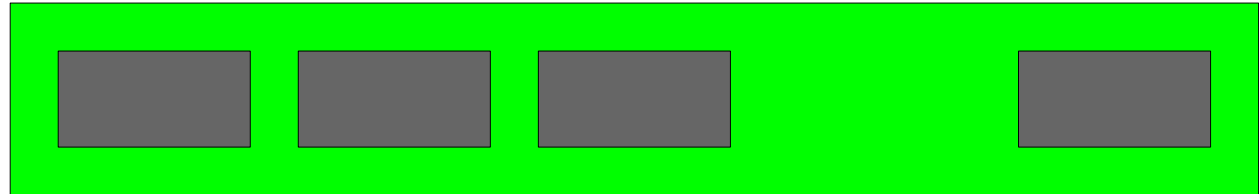


Tenured Objects

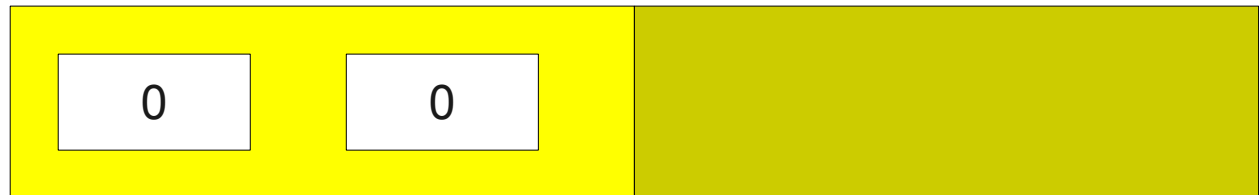


# Garbage Collection in Java

Eden



Survivor Objects

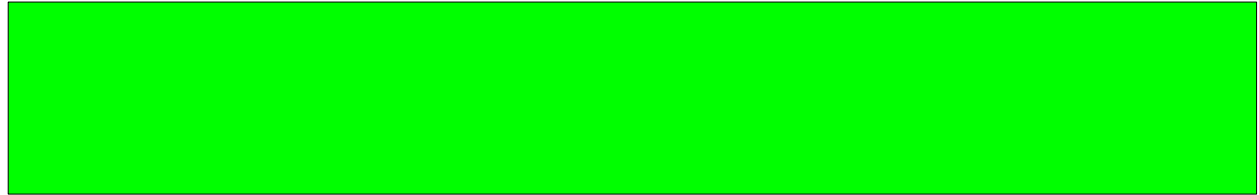


Tenured Objects

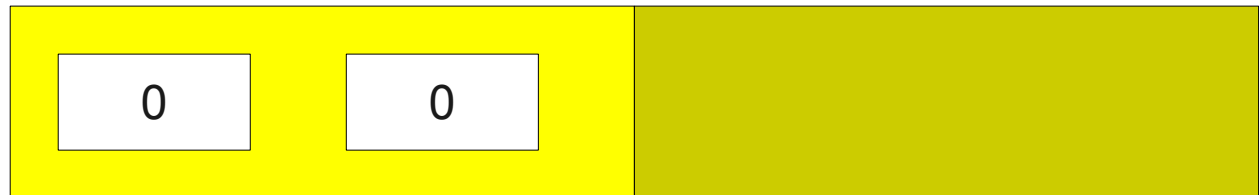


# Garbage Collection in Java

Eden



Survivor Objects

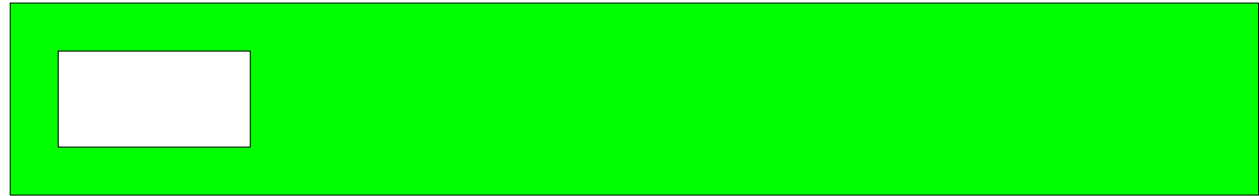


Tenured Objects

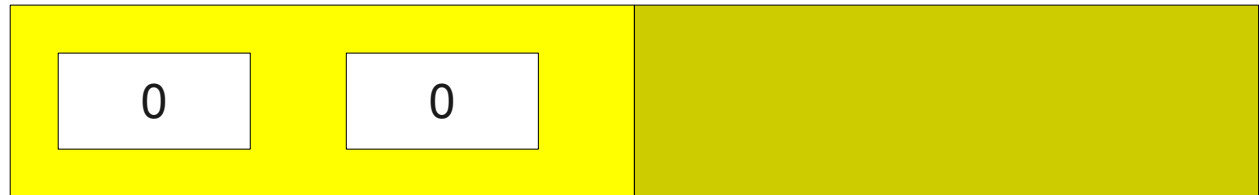


# Garbage Collection in Java

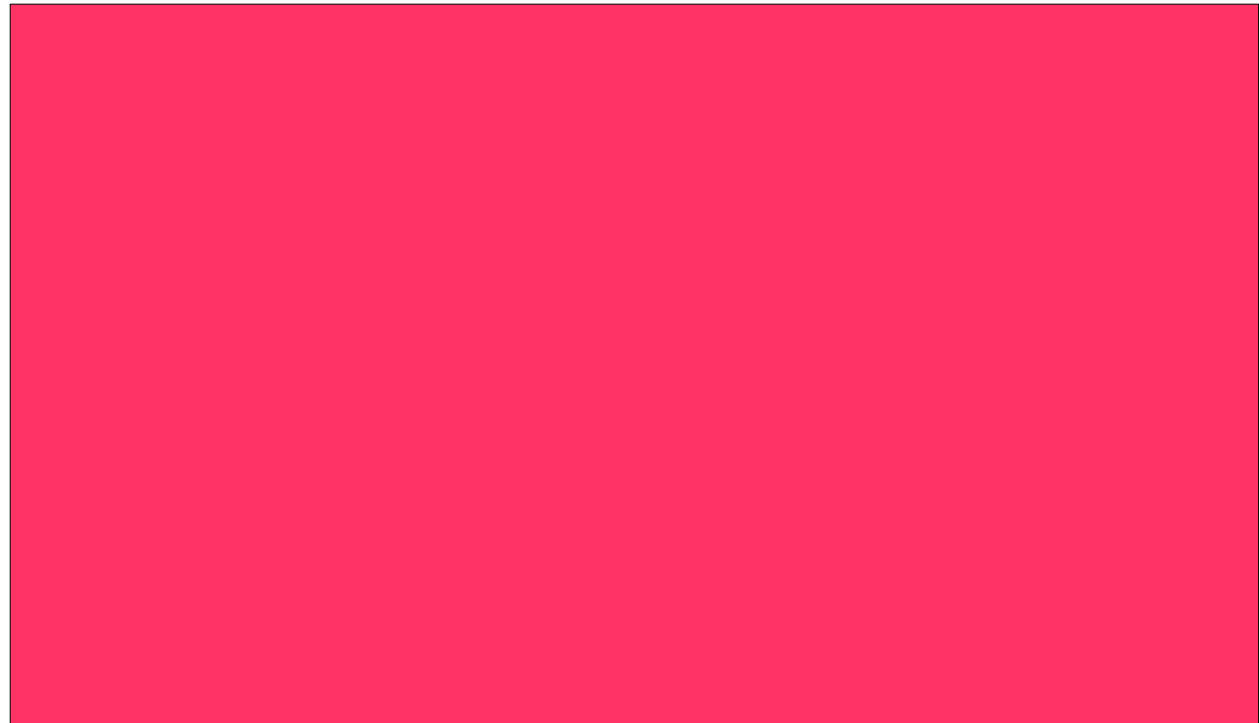
Eden



Survivor Objects

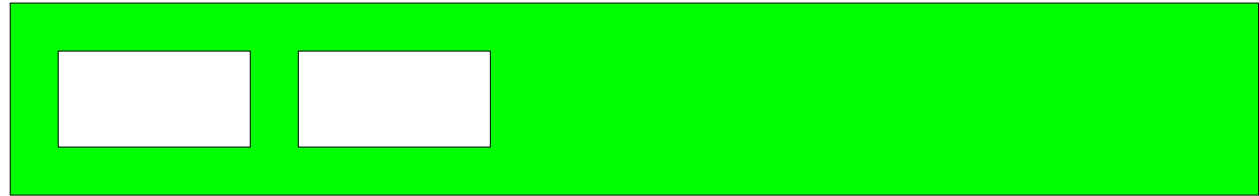


Tenured Objects

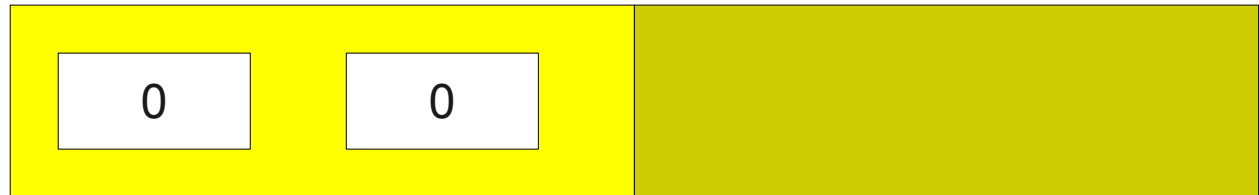


# Garbage Collection in Java

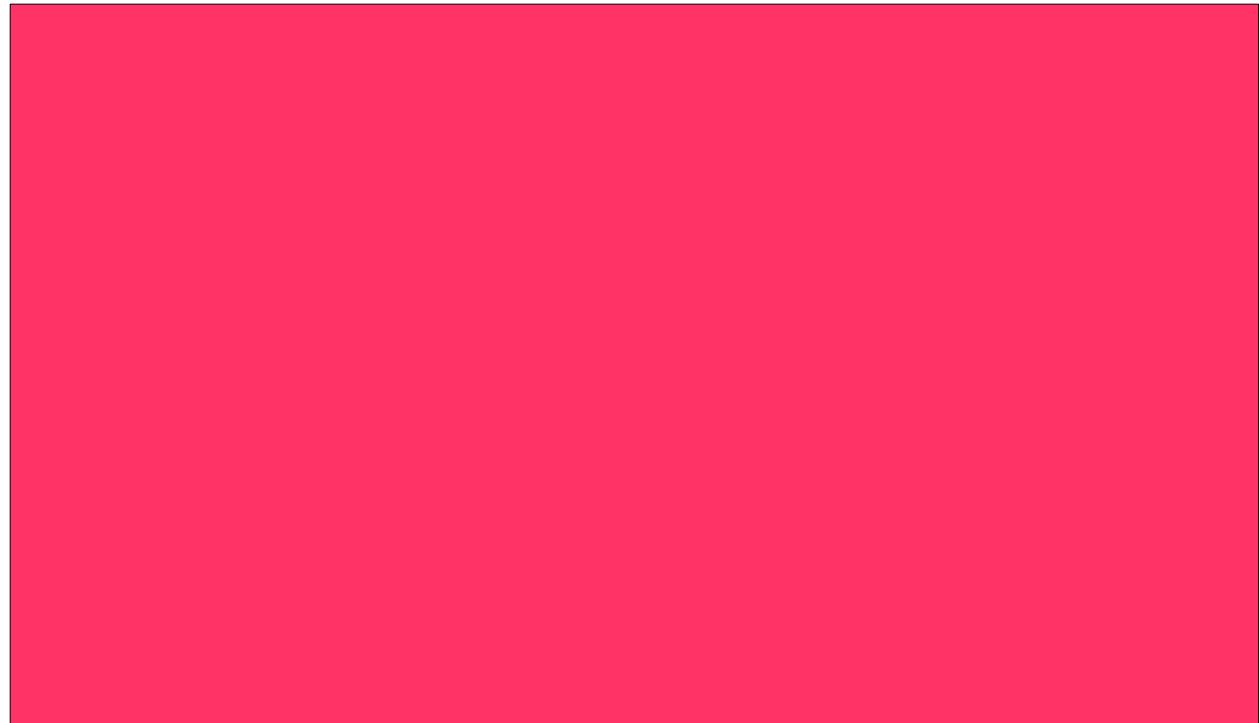
Eden



Survivor Objects

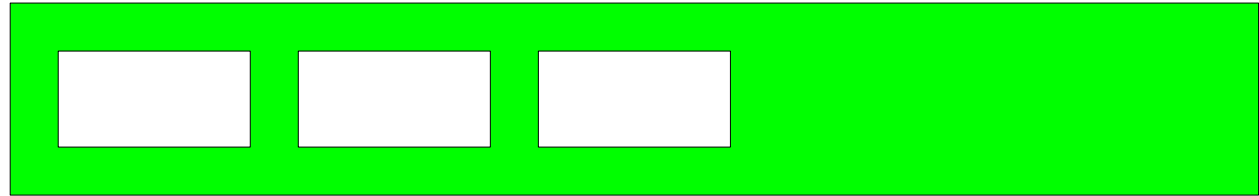


Tenured Objects

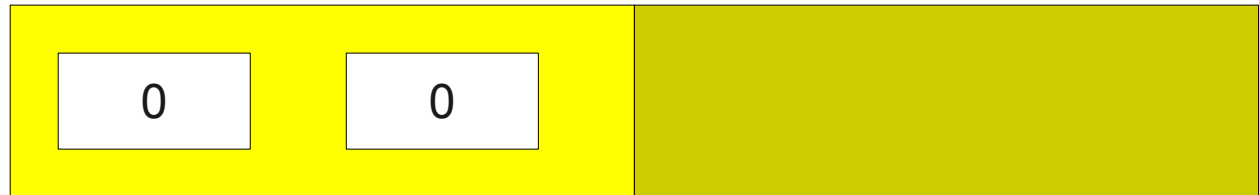


# Garbage Collection in Java

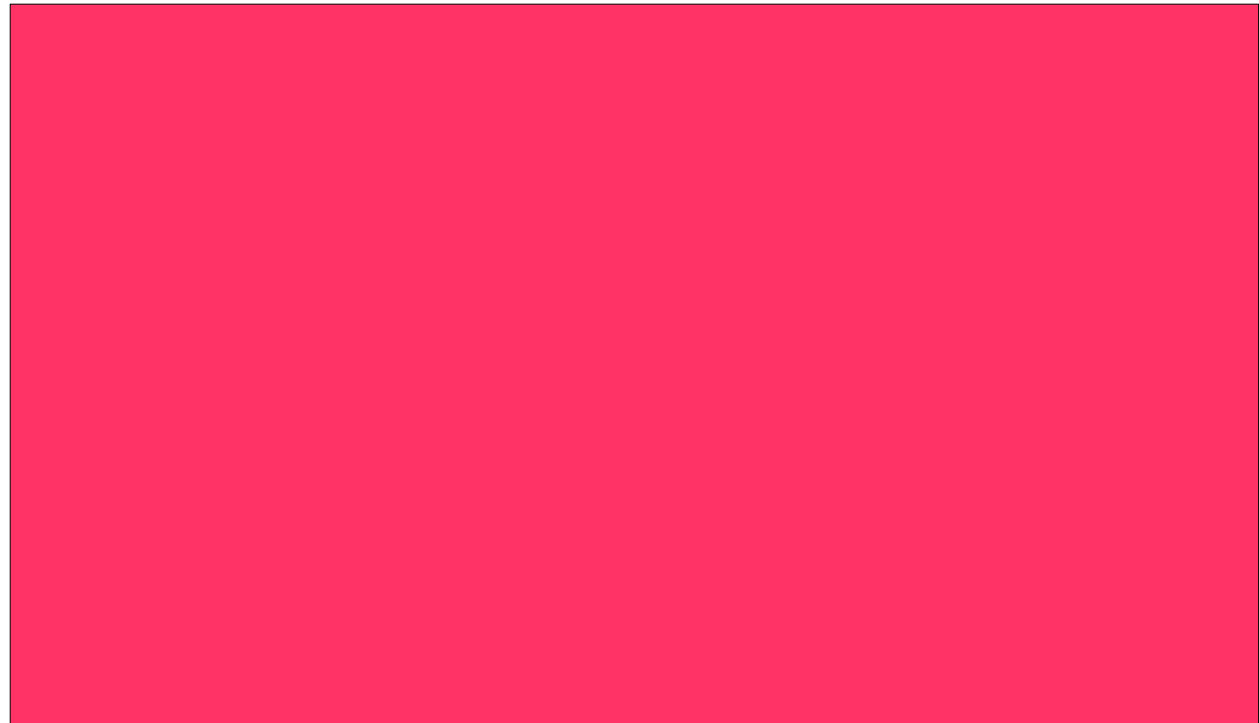
Eden



Survivor Objects



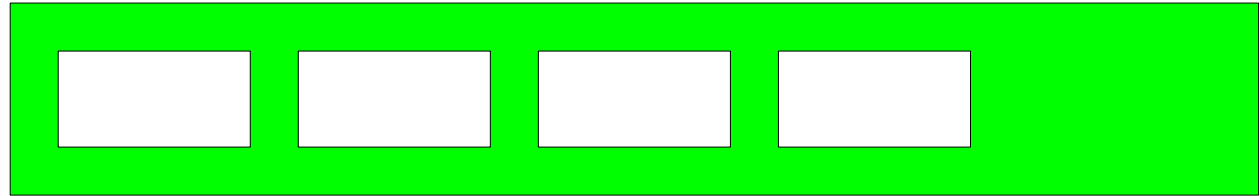
Tenured Objects



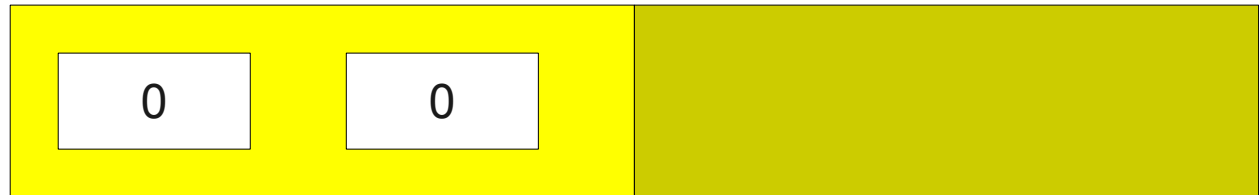


# Garbage Collection in Java

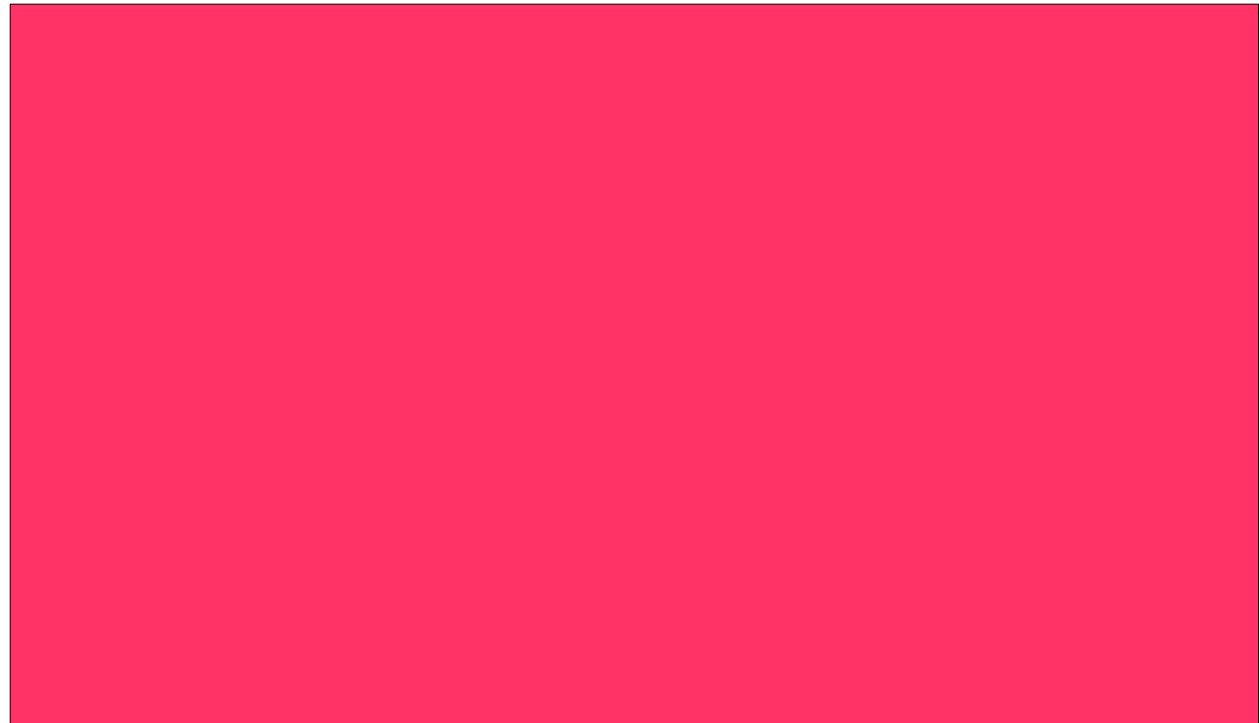
Eden



Survivor Objects

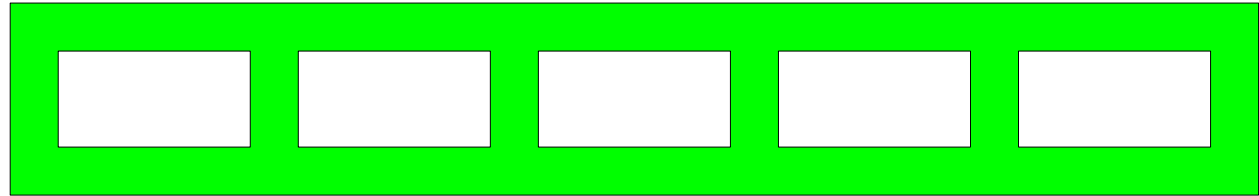


Tenured Objects

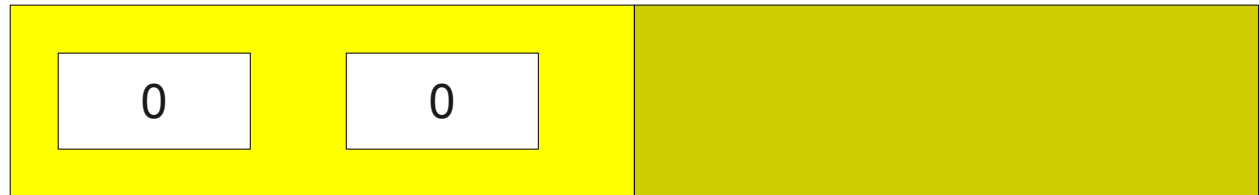


# Garbage Collection in Java

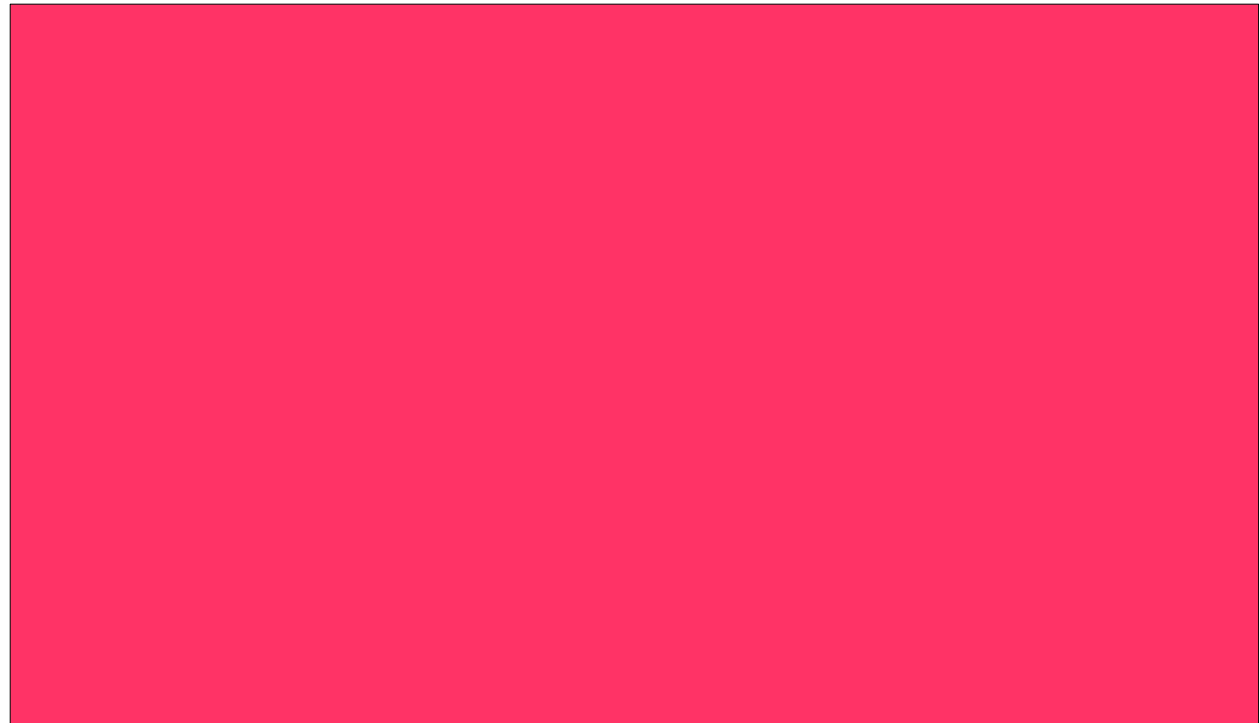
Eden



Survivor Objects

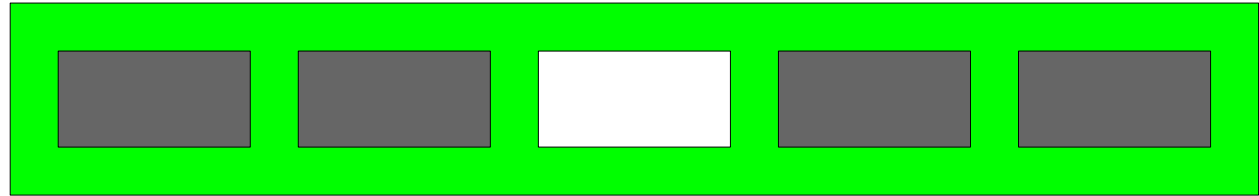


Tenured Objects

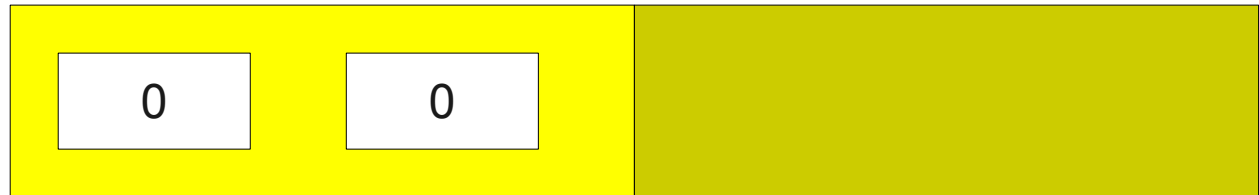


# Garbage Collection in Java

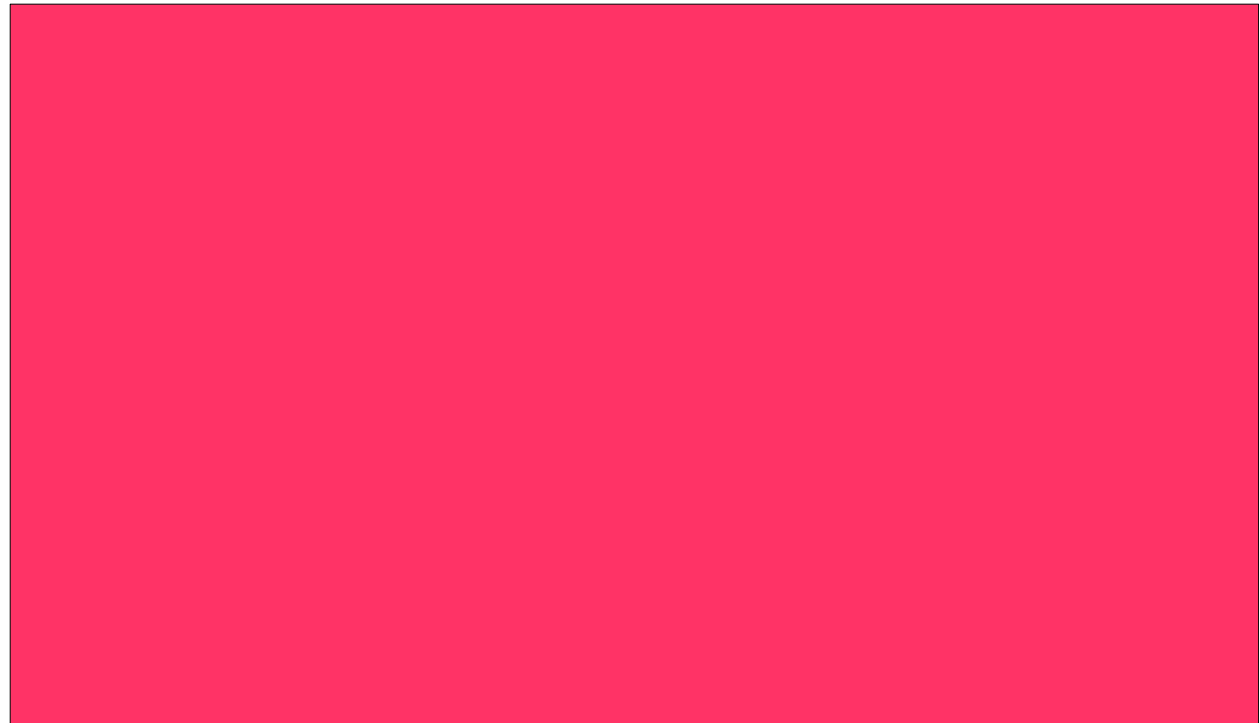
Eden



Survivor Objects

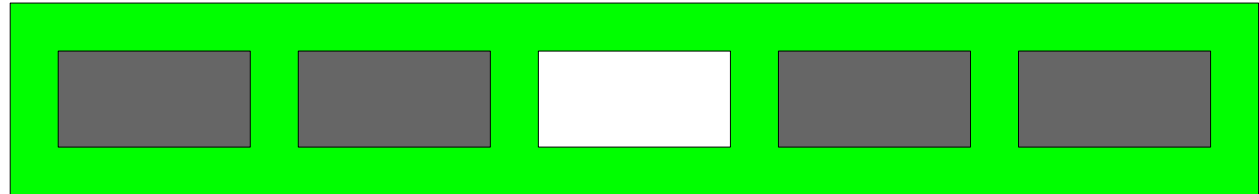


Tenured Objects

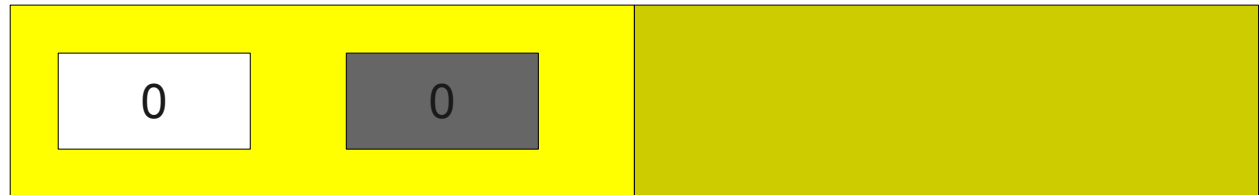


# Garbage Collection in Java

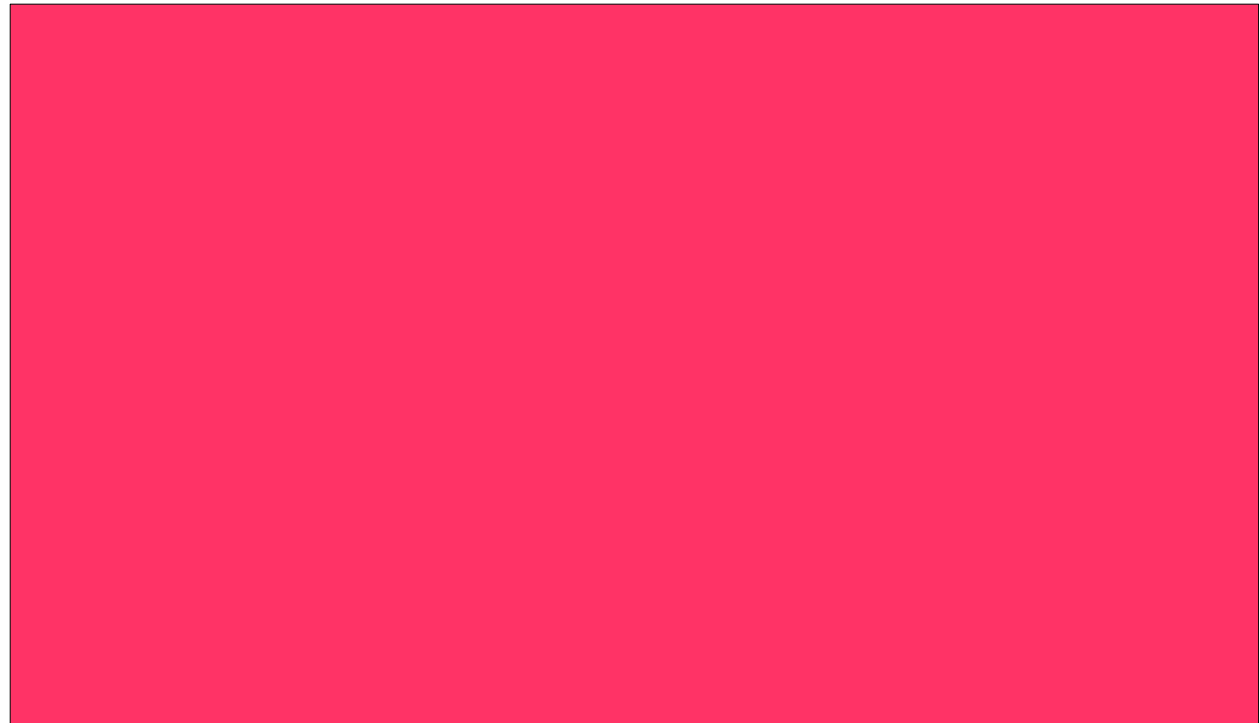
Eden



Survivor Objects

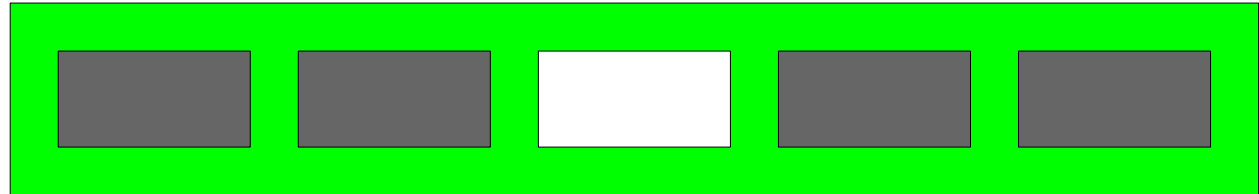


Tenured Objects

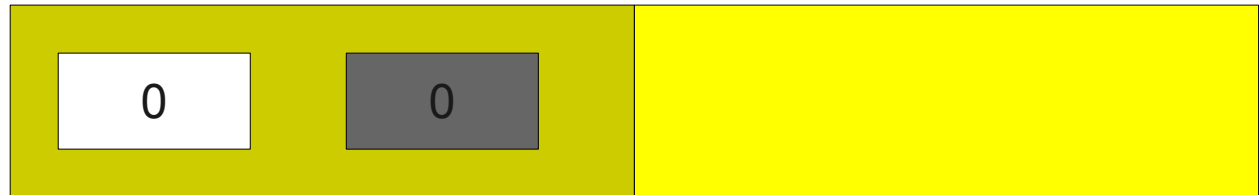


# Garbage Collection in Java

Eden



Survivor Objects

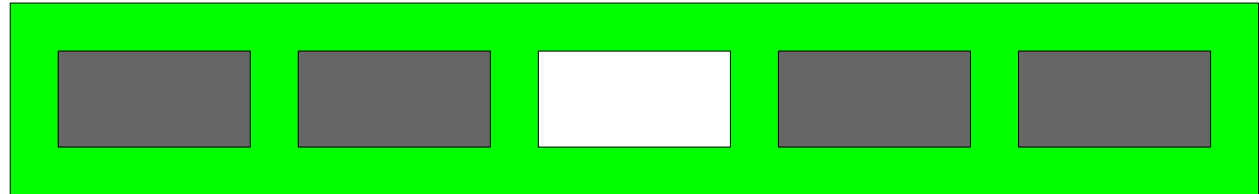


Tenured Objects

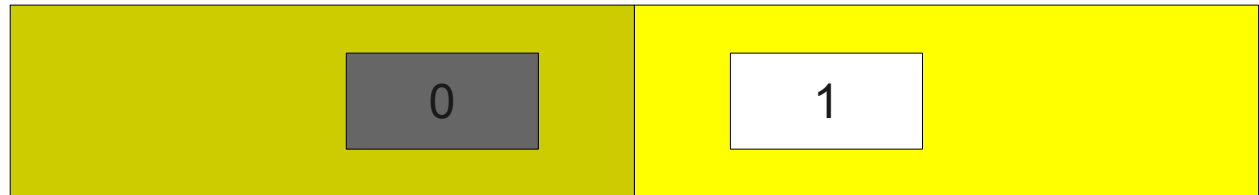


# Garbage Collection in Java

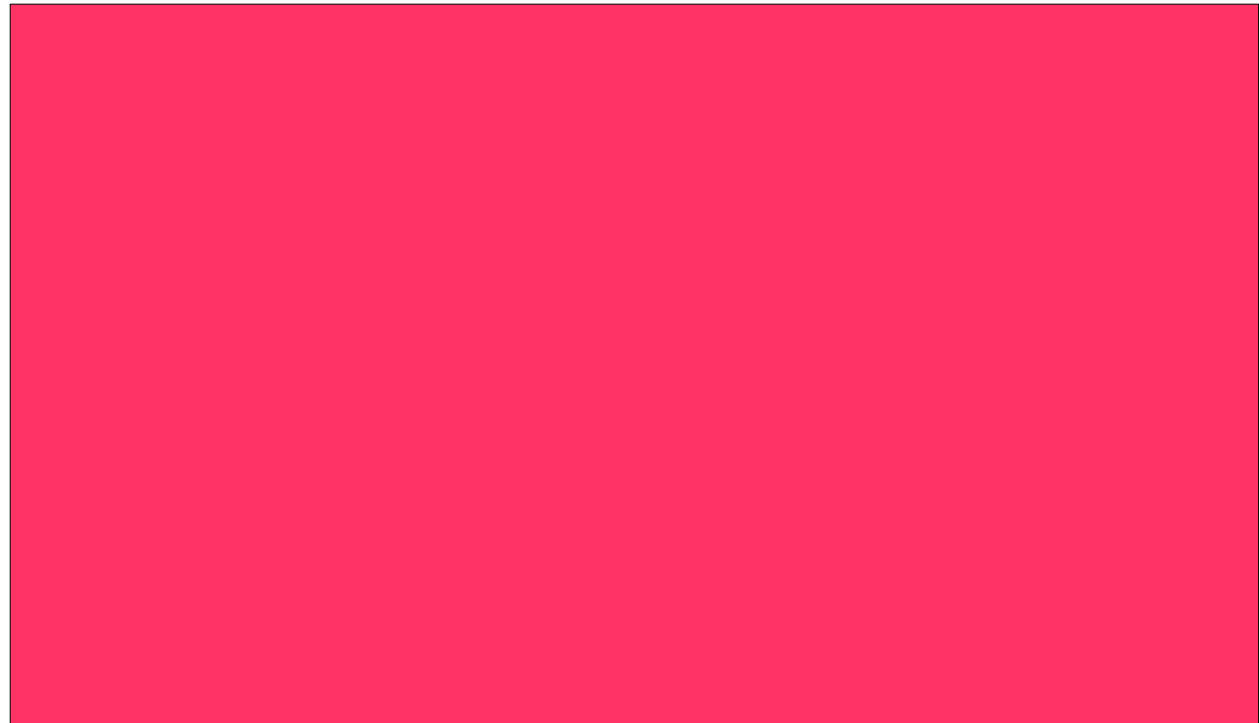
Eden



Survivor Objects

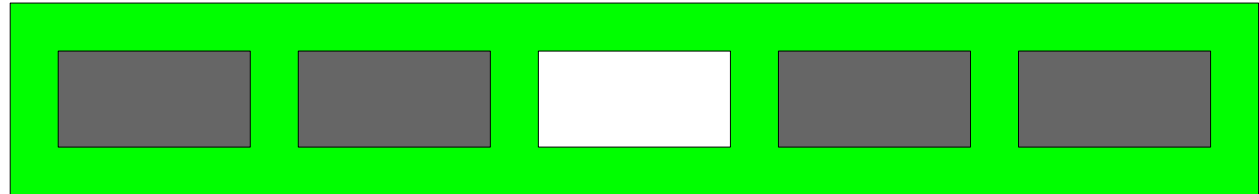


Tenured Objects

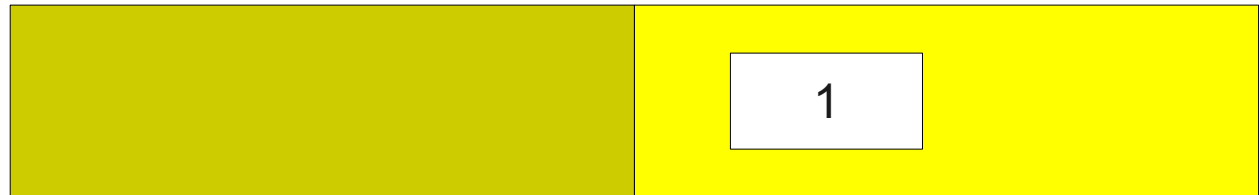


# Garbage Collection in Java

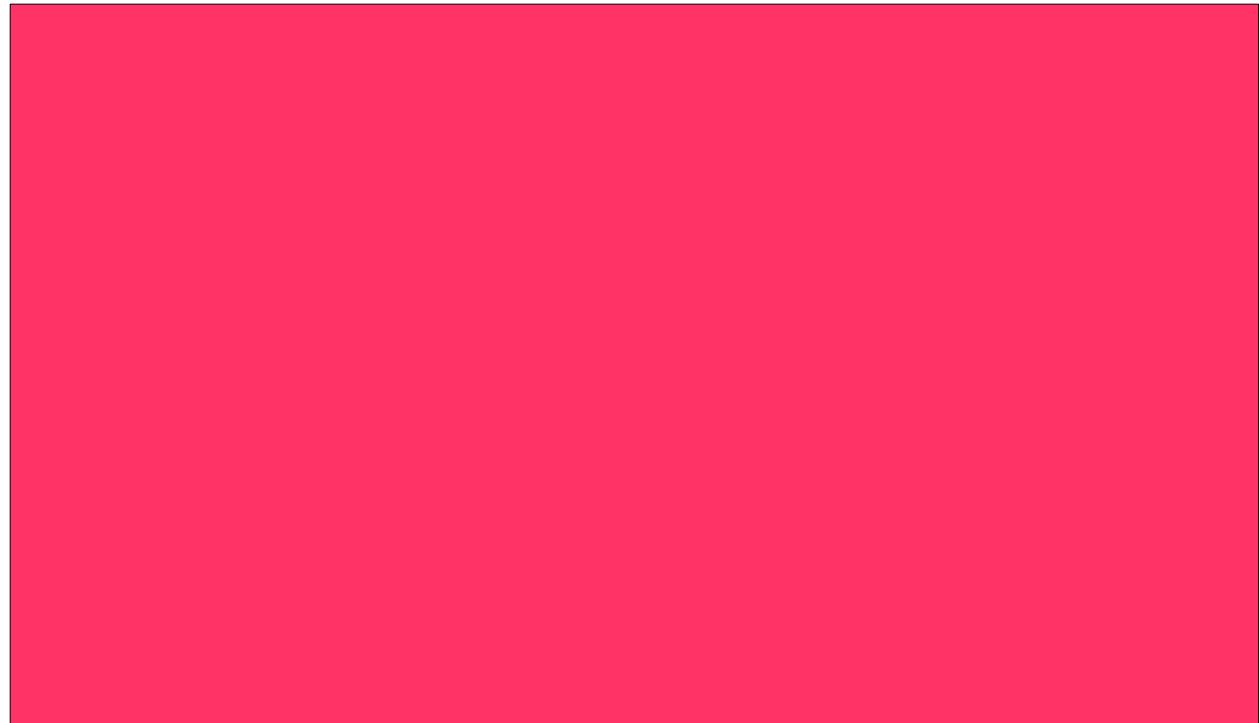
Eden



Survivor Objects



Tenured Objects

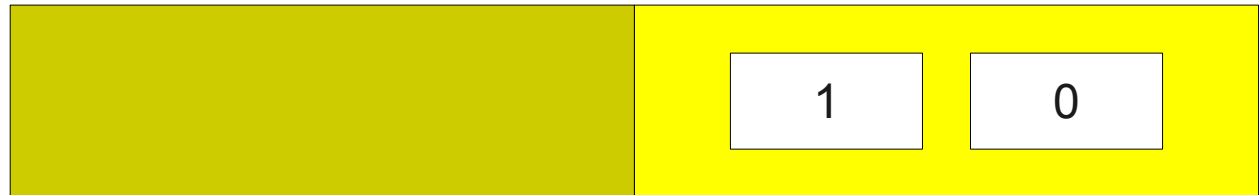


# Garbage Collection in Java

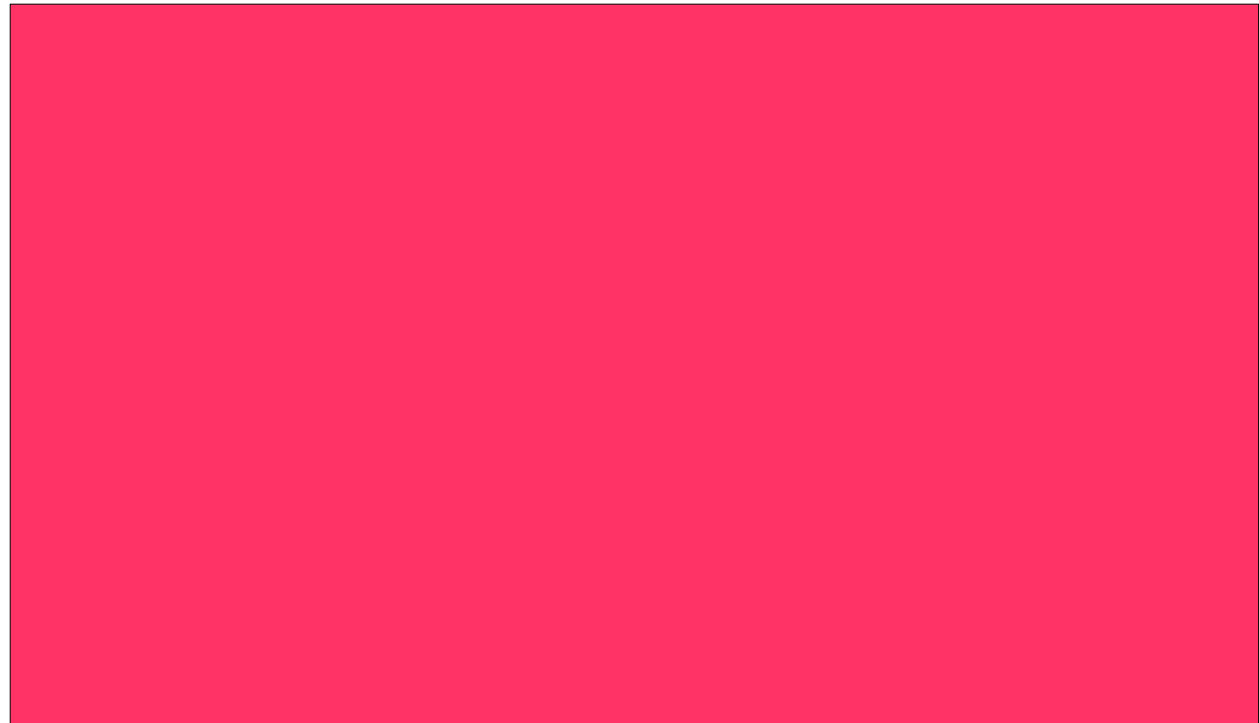
Eden



Survivor Objects



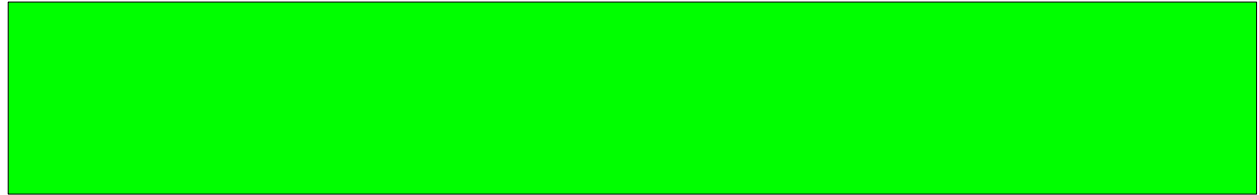
Tenured Objects



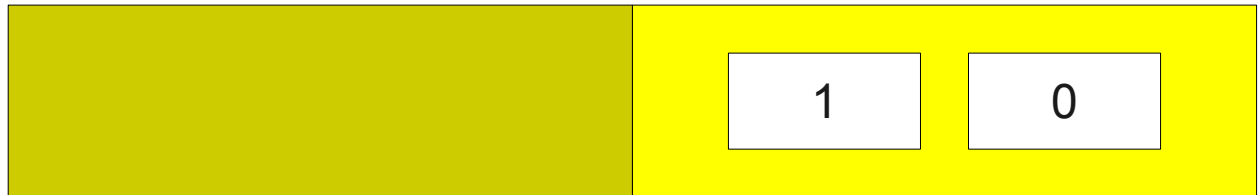


# Garbage Collection in Java

Eden



Survivor Objects

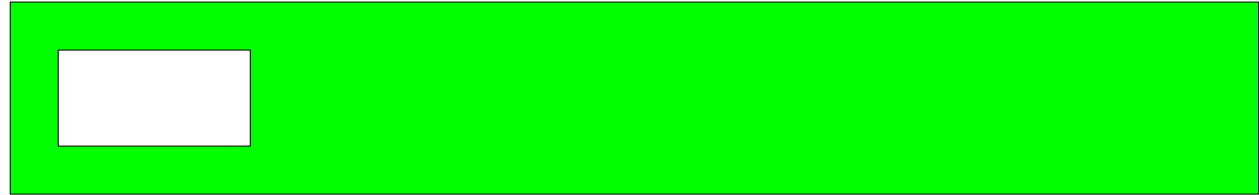


Tenured Objects

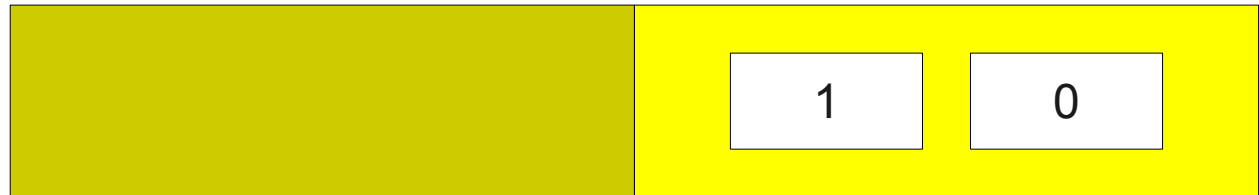


# Garbage Collection in Java

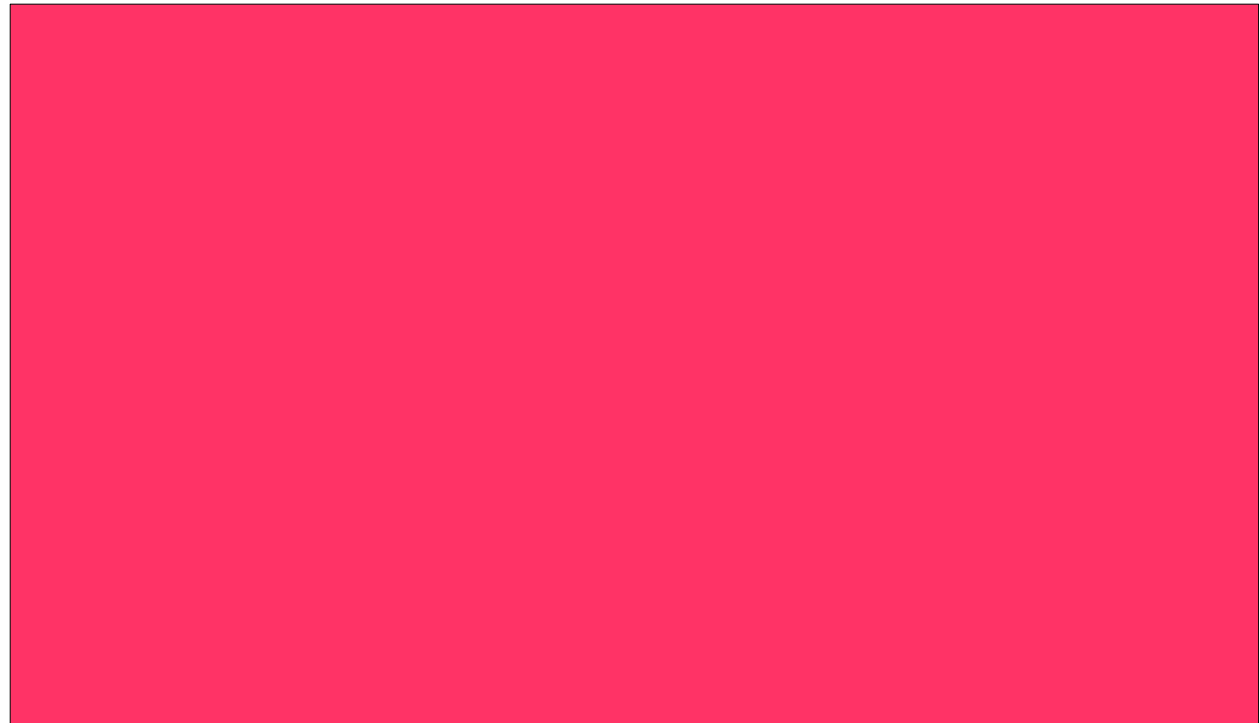
Eden



Survivor Objects

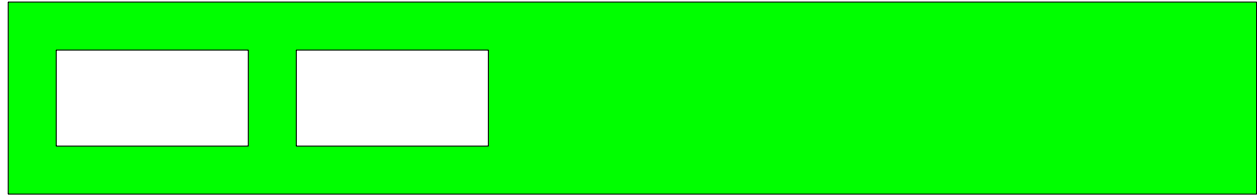


Tenured Objects

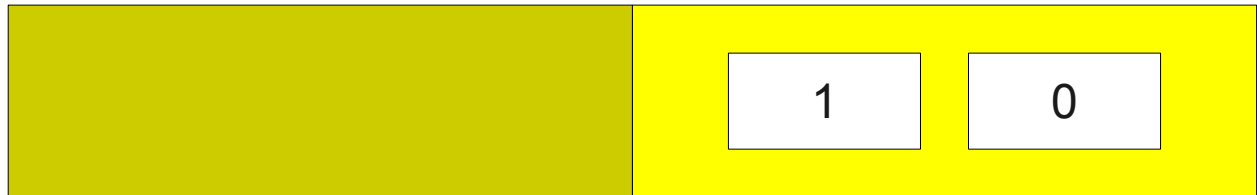


# Garbage Collection in Java

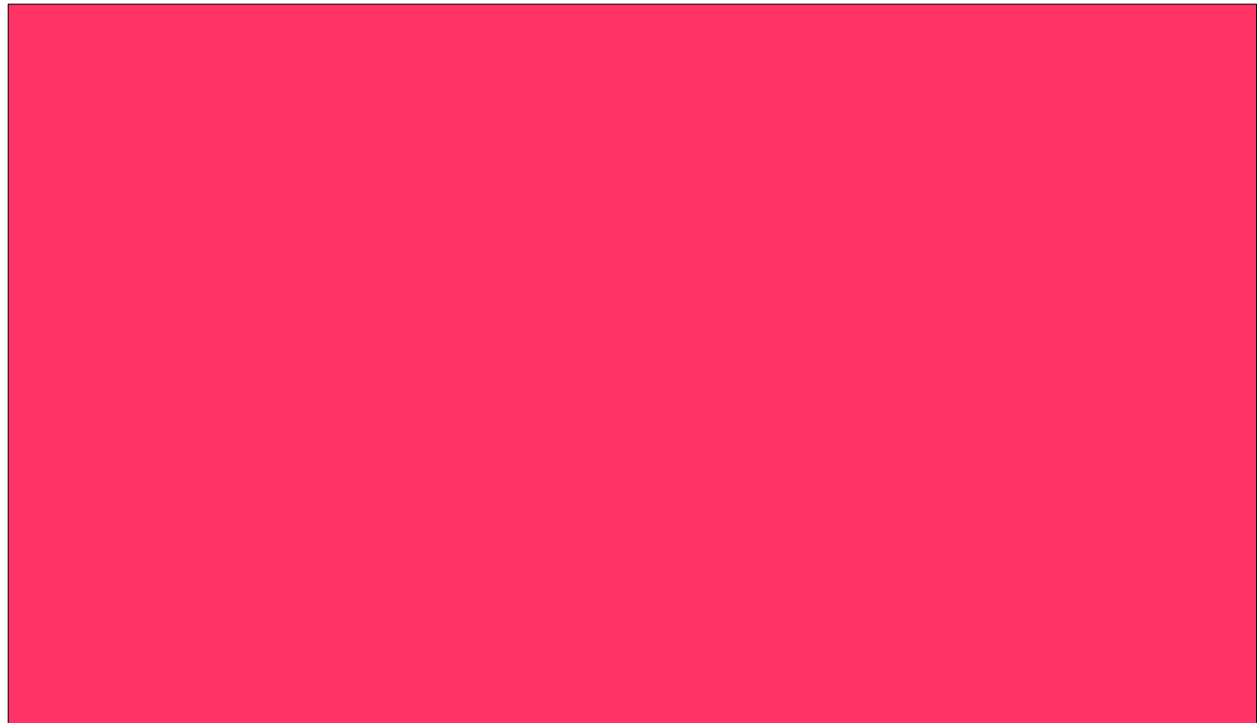
Eden



Survivor Objects

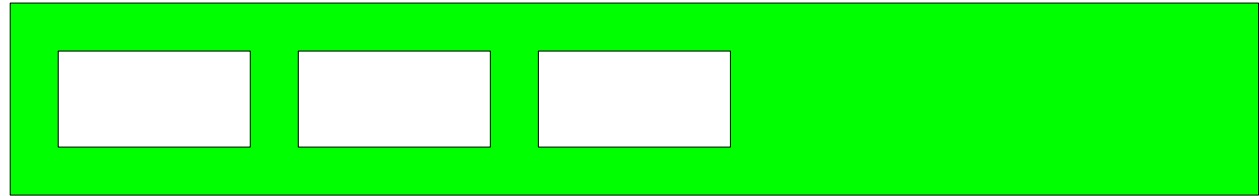


Tenured Objects

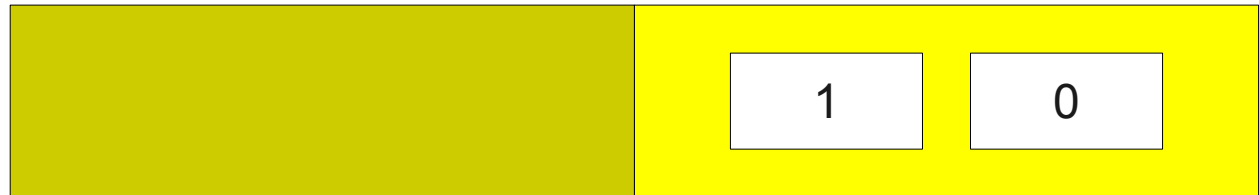


# Garbage Collection in Java

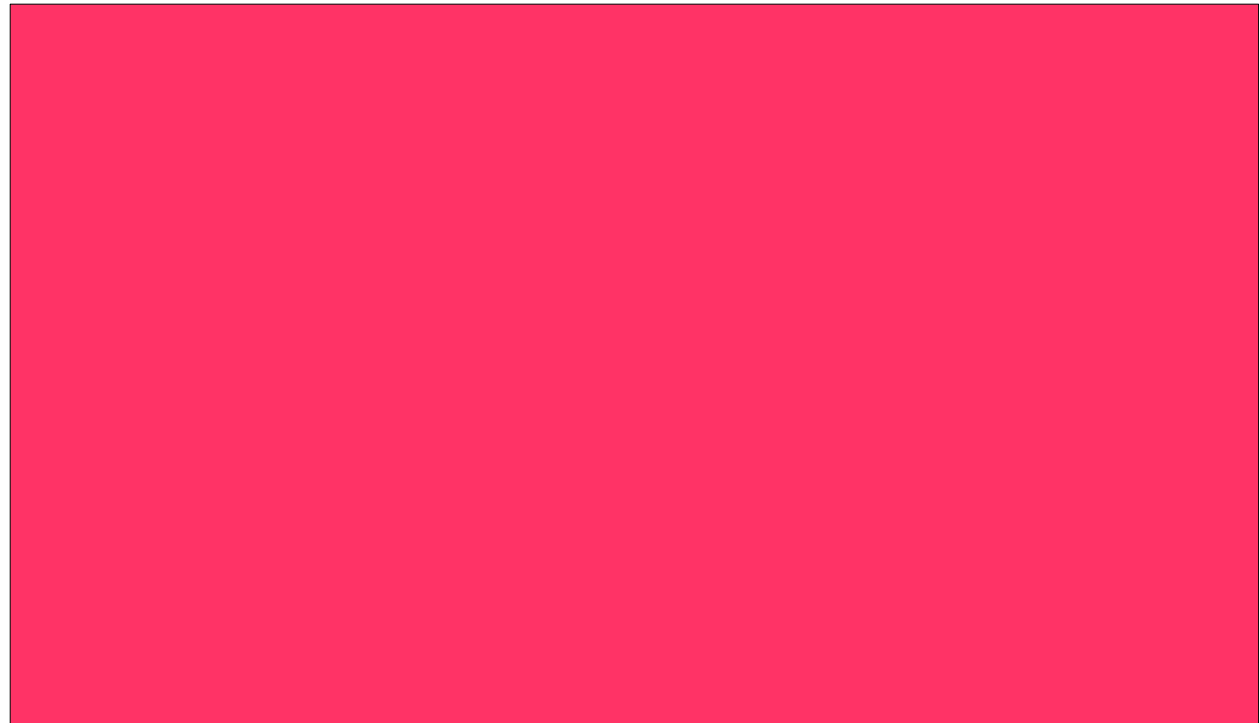
Eden



Survivor Objects

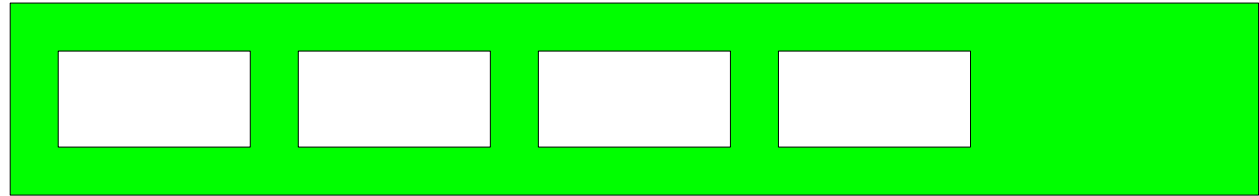


Tenured Objects

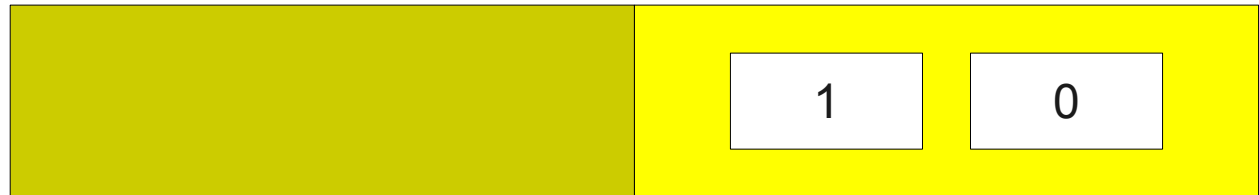


# Garbage Collection in Java

Eden



Survivor Objects

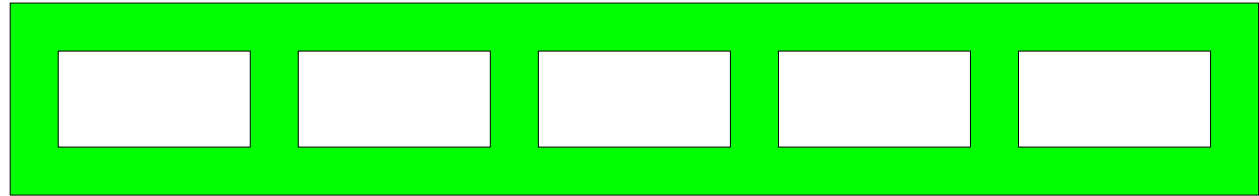


Tenured Objects

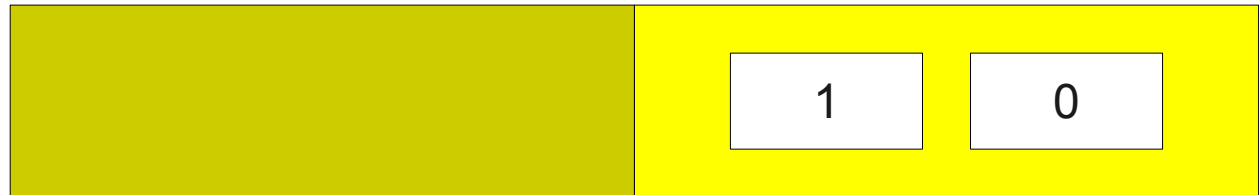


# Garbage Collection in Java

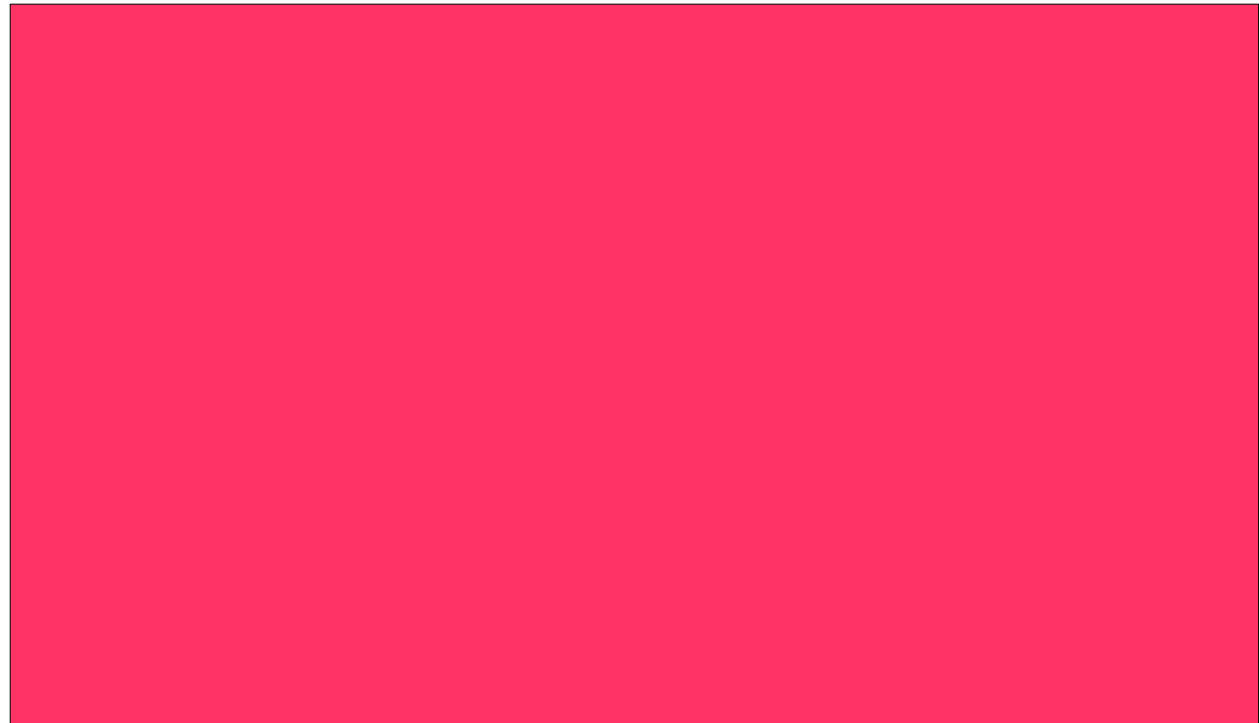
Eden



Survivor Objects

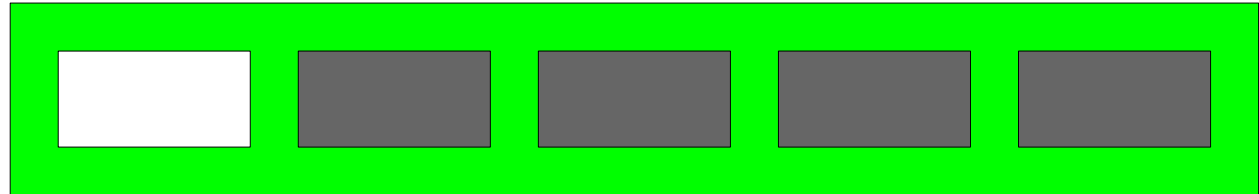


Tenured Objects

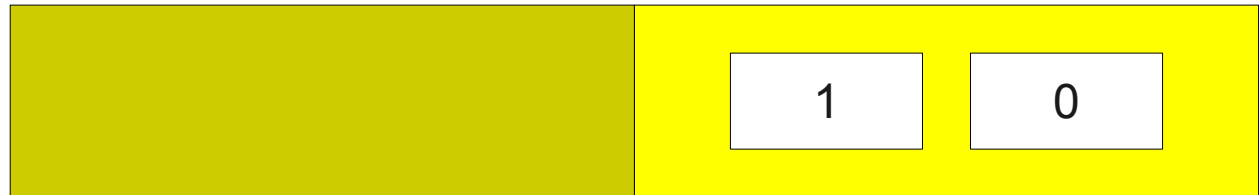


# Garbage Collection in Java

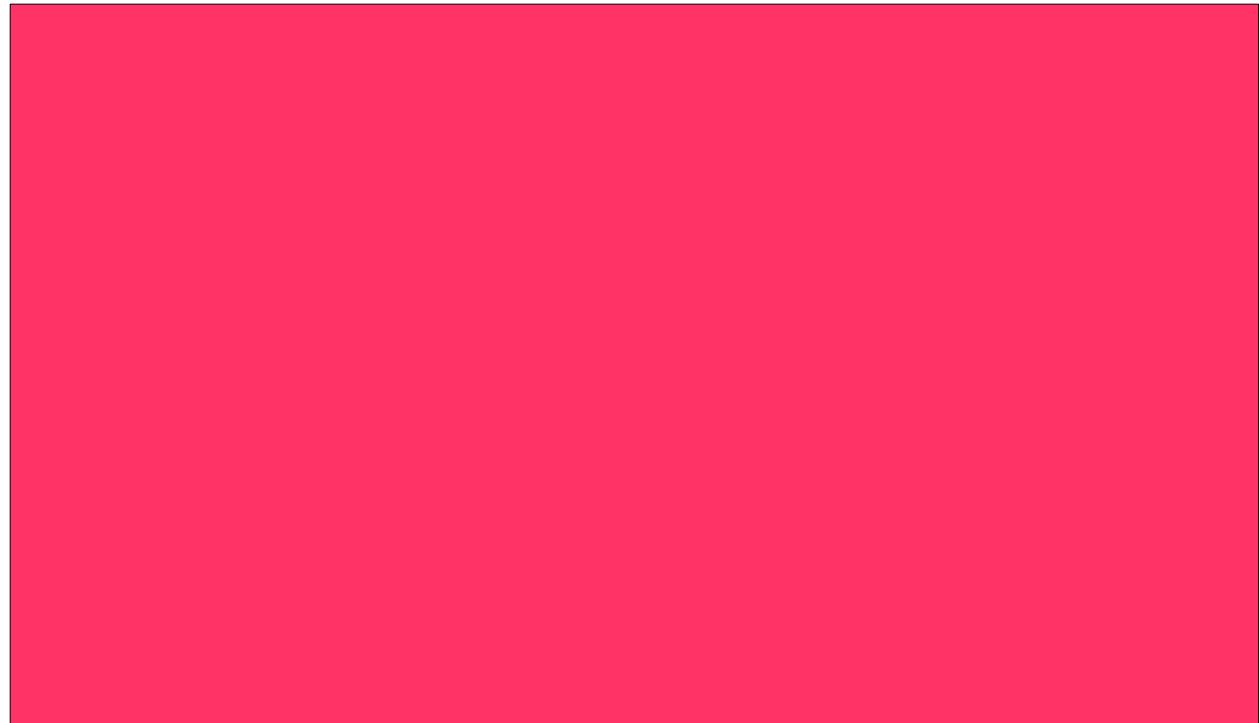
Eden



Survivor Objects

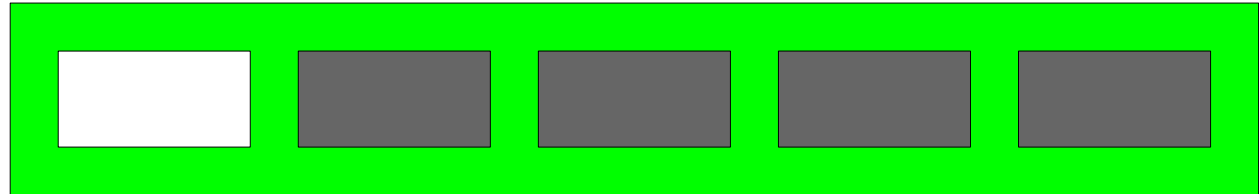


Tenured Objects

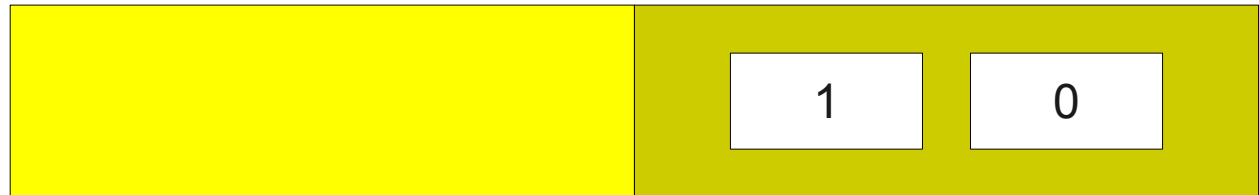


# Garbage Collection in Java

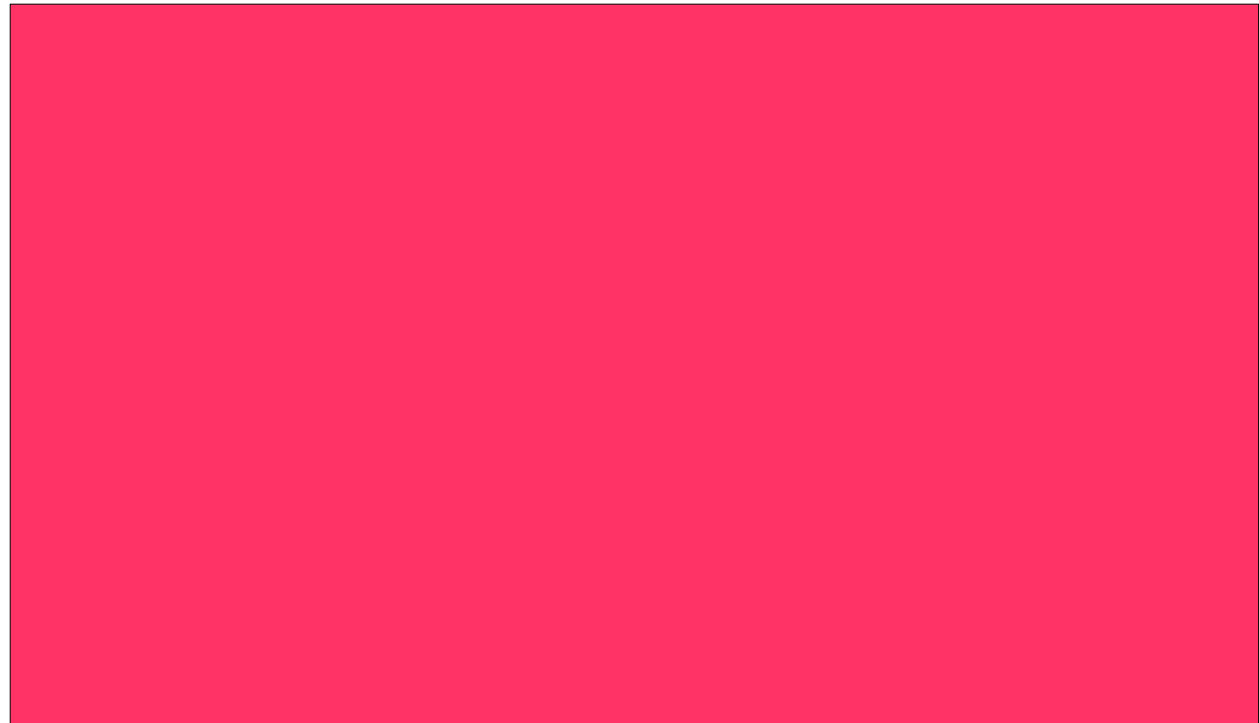
Eden



Survivor Objects



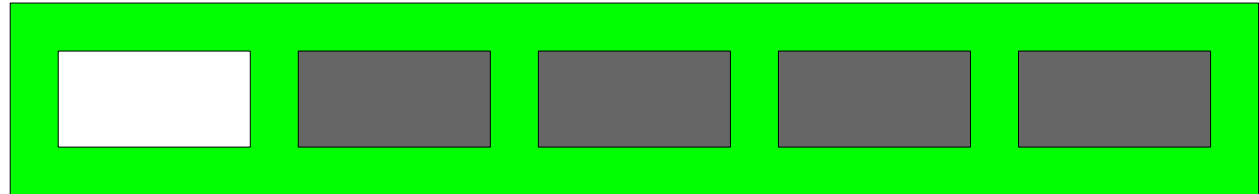
Tenured Objects



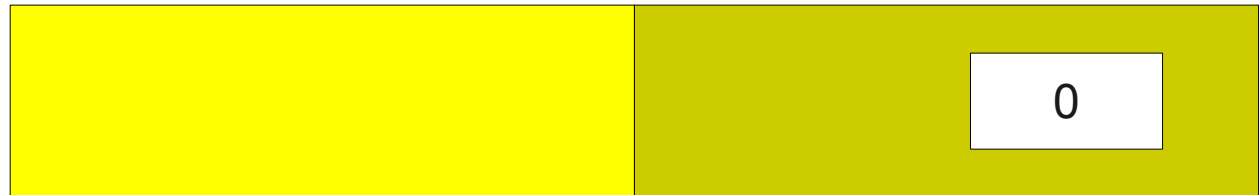


# Garbage Collection in Java

Eden



Survivor Objects

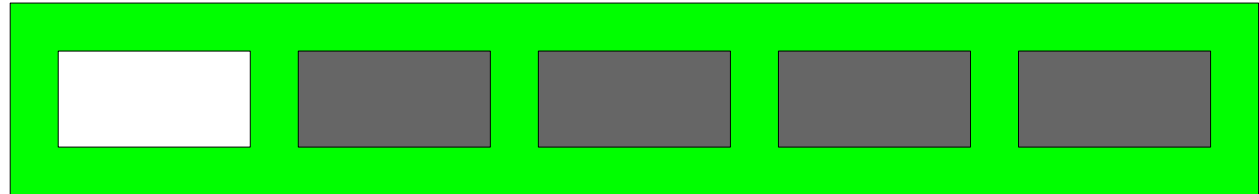


Tenured Objects

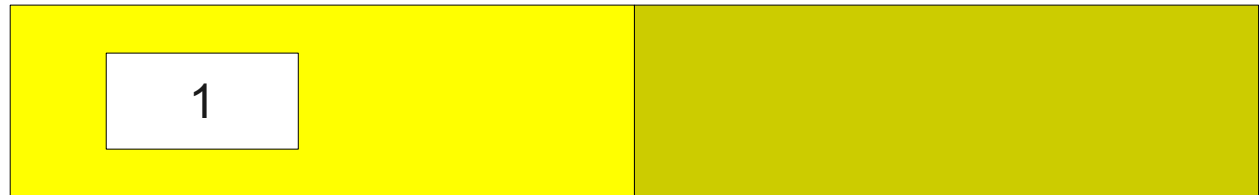


# Garbage Collection in Java

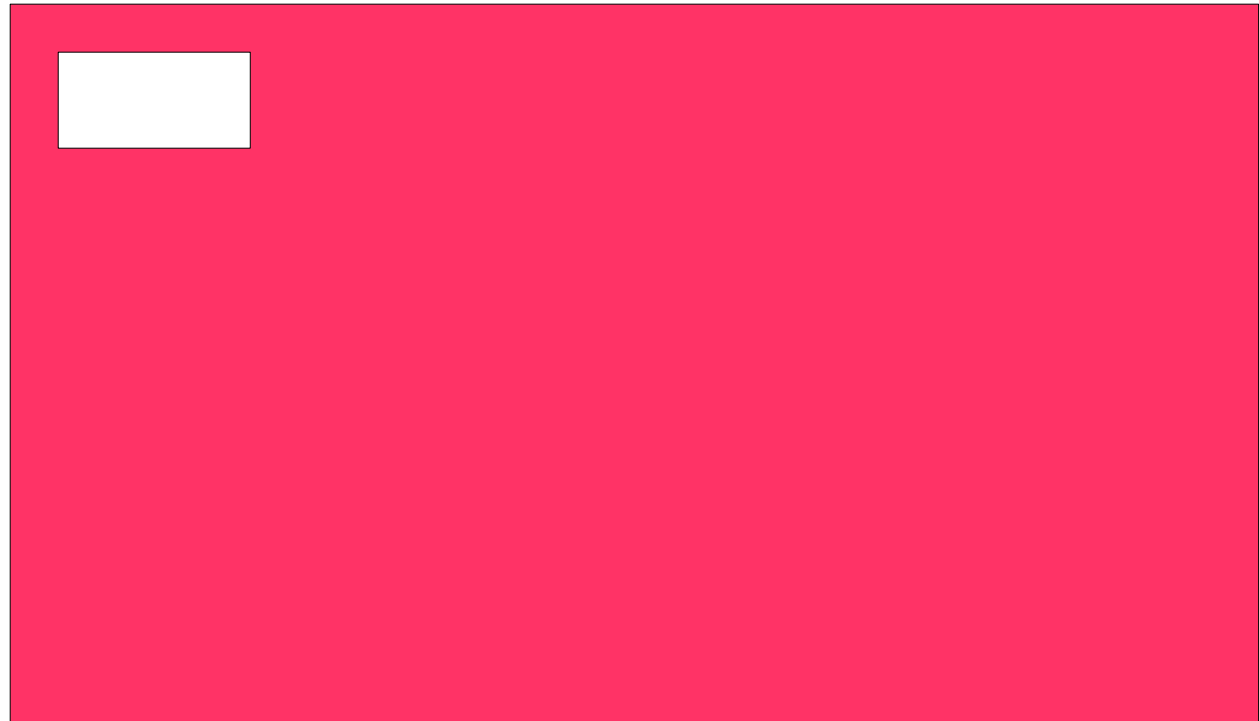
Eden



Survivor Objects

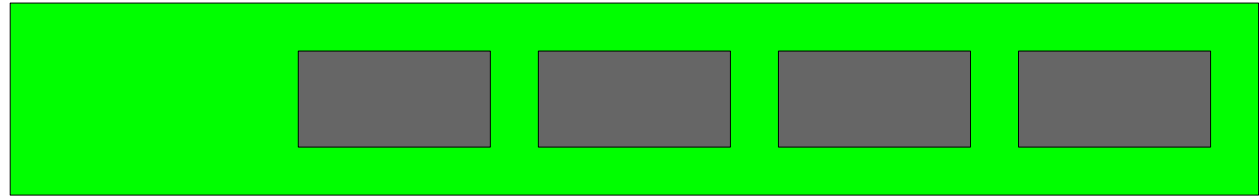


Tenured Objects

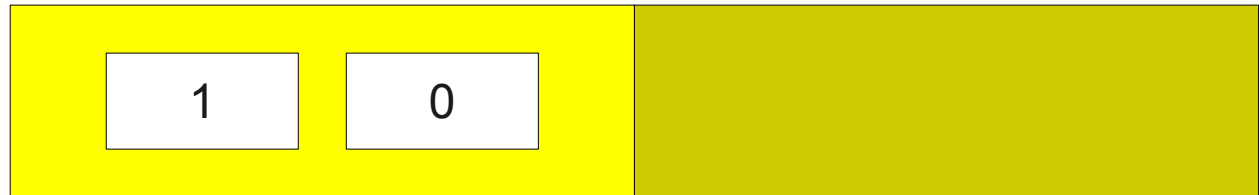


# Garbage Collection in Java

Eden



Survivor Objects

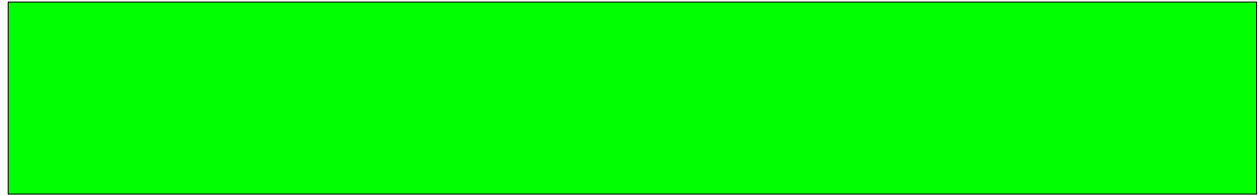


Tenured Objects

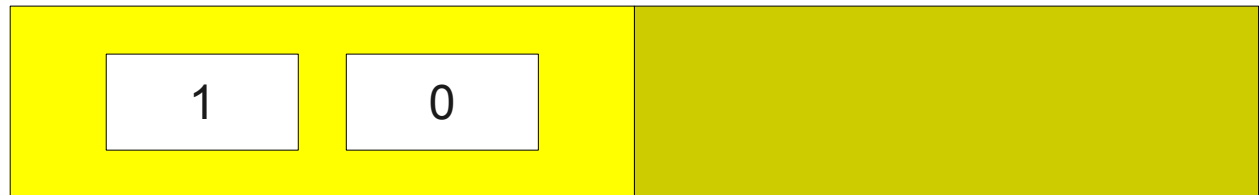


# Garbage Collection in Java

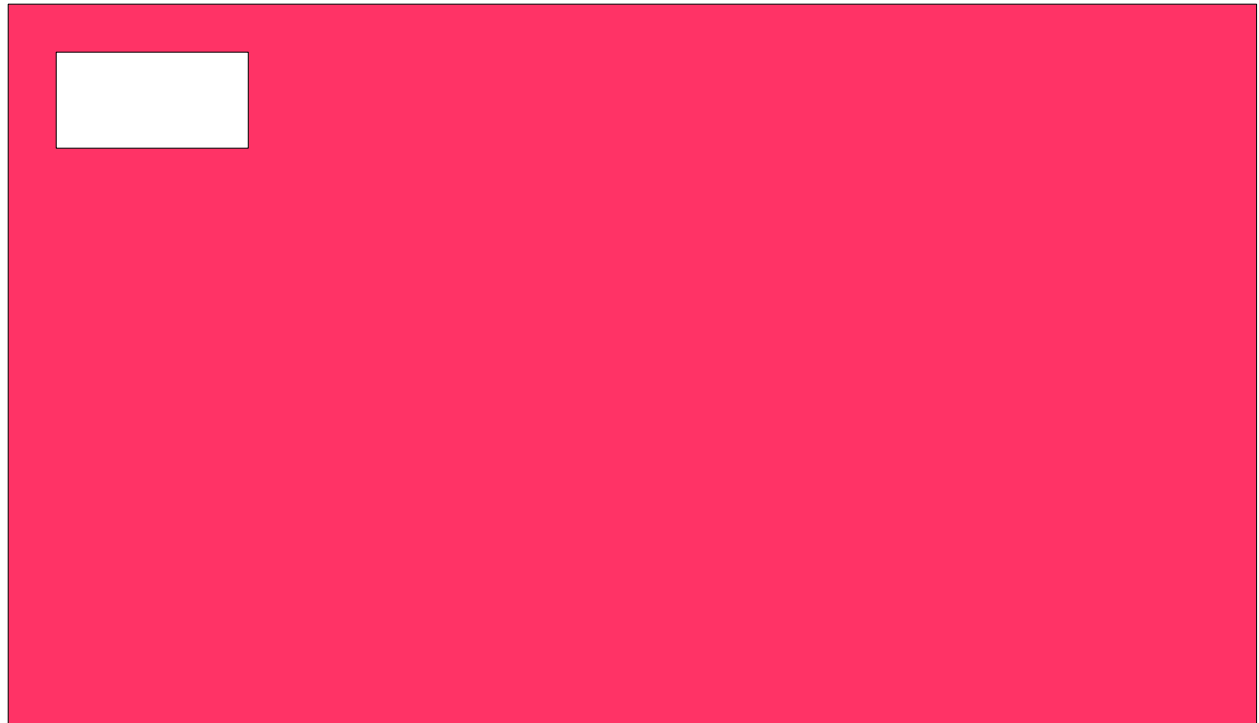
Eden



Survivor Objects



Tenured Objects



# HotSpot Garbage Collection

- New objects are allocated using a modified stop-and-copy collector in the **Eden** space.
- When Eden runs out of space, the stop-and-copy collector moves its elements to the **survivor space**.
- Objects that survive long enough in the survivor space become **tenured** and are moved to the **tenured space**.
- When memory fills up, a full garbage collection (perhaps mark-and-sweep) is used to garbage-collect the tenured objects.

# Next Time

- **Final Code Optimization**
  - Instruction scheduling.
  - Locality optimizations.
- **Where to Go From Here**
- **Final Thoughts**