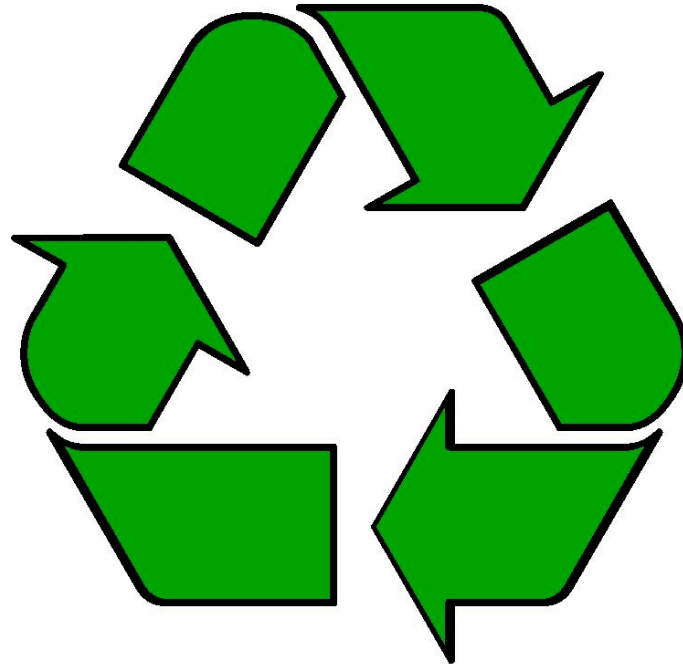


Garbage Collection



Memory Management So Far

- Some items are preallocated and persist throughout the program:
 - What are they?
- Some are allocated on the runtime stack:
 - What are they?
- Some are allocated in the heap:
 - What are they?
- How do we manage heap-allocated memory?

Manual Memory Management

- **Option One:** Have programmer handle heap allocation and deallocation
- What languages do this?

Advantages?

Disadvantages?

What is a memory leak?

Automatic Memory Management

Idea:

Runtime environment automatically reclaims memory.

What is the **garbage**?

Advantages?

Disadvantages?

Can you Identify Garbage?

- 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, **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.

Garbage Collector - Detect and reclaim unreachable objects.

GC Assumptions

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

GC Types

Incremental vs stop-the-world:

Incremental- run concurrently with program

Stop-the-world - pause 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:

compacting - moves objects around in memory.

non-compacting - leaves all objects where they originated.

Which (generally) spends more time garbage collecting?

Which (generally) leads to faster program execution?

Major Approaches to GC

1. Reference Counting
2. Mark and Sweep
3. Stop and Copy
4. Generational

Reference Counting Technique

- * Store in each object a **reference count (refcount)** tracking how many references exist to the object.
- * Create a reference to an object: increments refcount.
- * Remove a reference to an object: decrements refcount.
- * When object $\text{refcount}==0$, unreachable & reclaimed.

Might decrease other objects' refcounts and trigger more reclamations.

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 

 0



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;

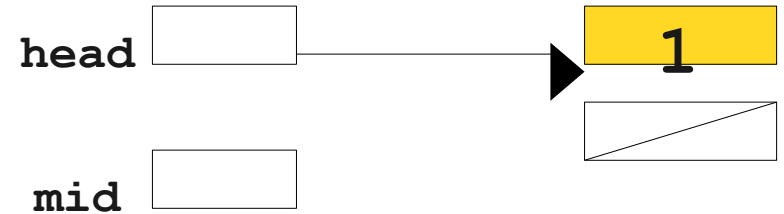
int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

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

    mid = tail = null;

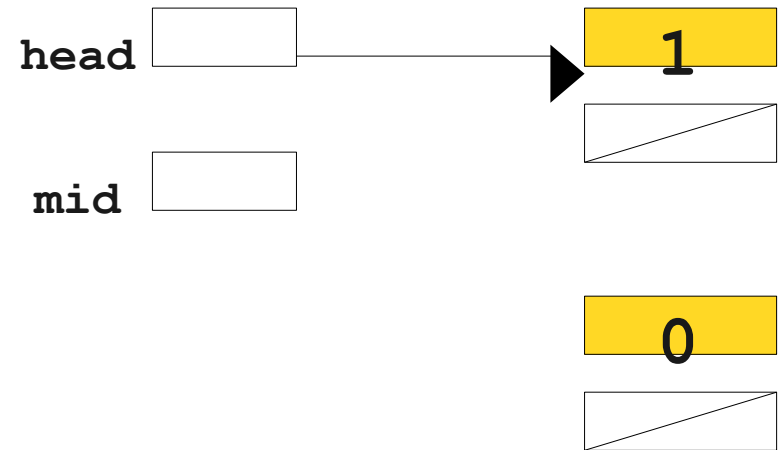
    head.next.next = null;

    head = null;
}
```



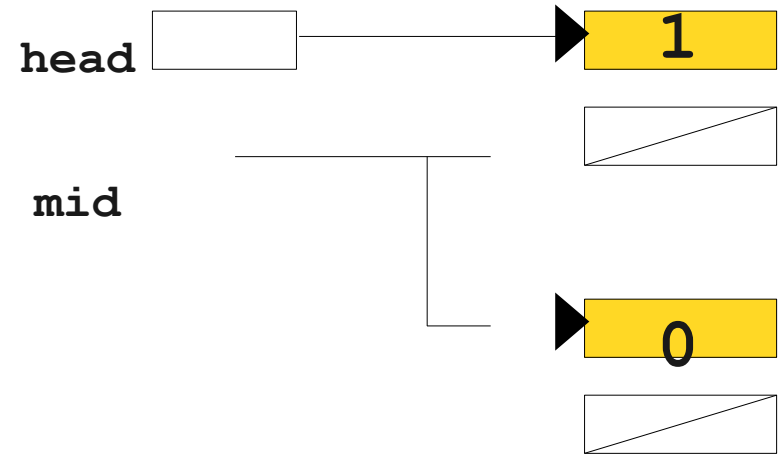
Reference Counting in Action

```
class LinkedList {  
} LinkedList next;  
  
int main()      head = new LinkedList;  
{ LinkedLi  
st  
  
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()    head = new LinkedList;  
{ LinkedList  
  st  
  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

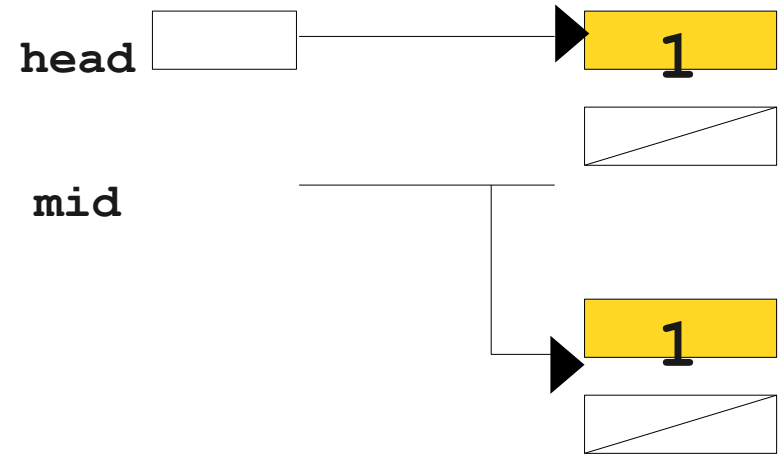
```
class LinkedList {
} LinkedList next;

int main()      head = new LinkedList;
{ LinkedList
  st
  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

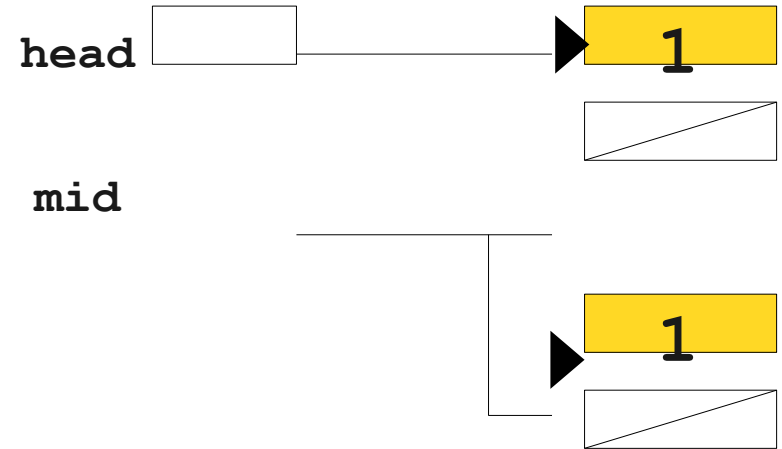
```
class LinkedList {
} LinkedList next;

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

  mid = tail = null;

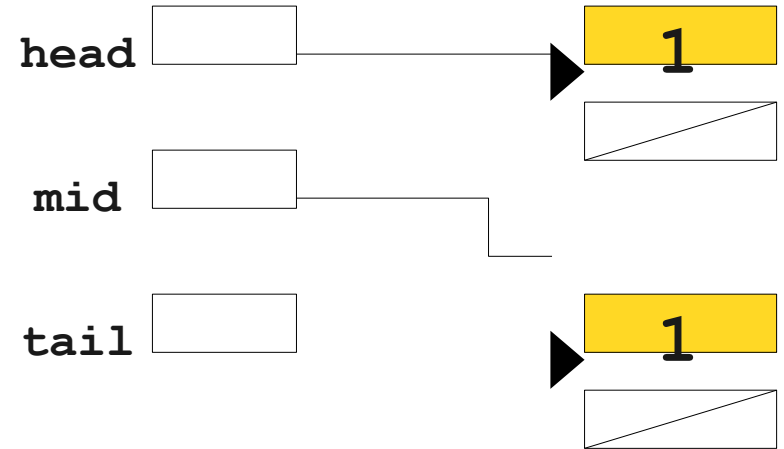
  head.next.next = null;

  head = null;
}
```



Reference Counting in Action

```
class LinkedList {  
}    LinkedList next;  
  
int main()    head = new LinkedList;  
{ LinkedList  
  st  
  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

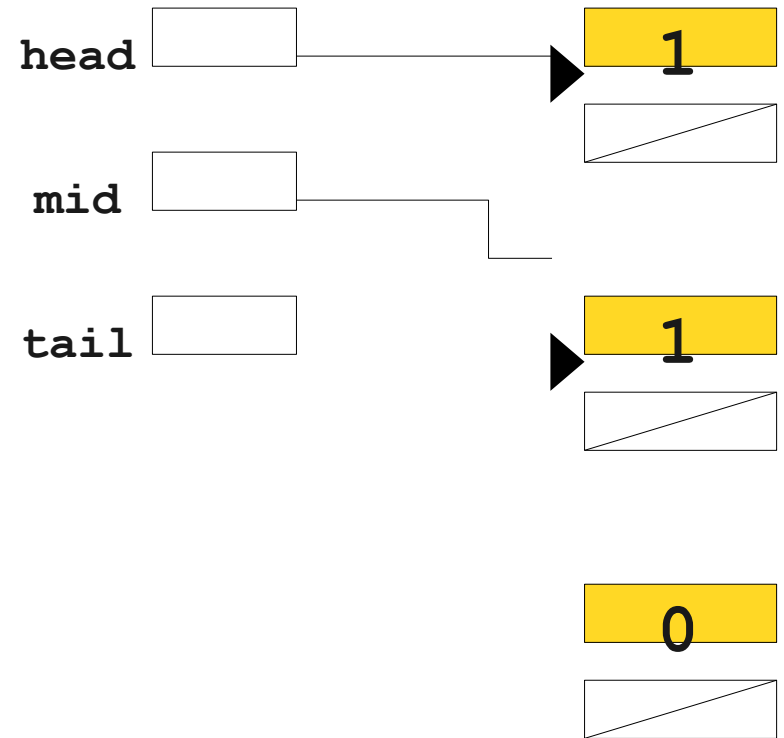
```
class LinkedList {
} LinkedList next;

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

  mid = tail = null;

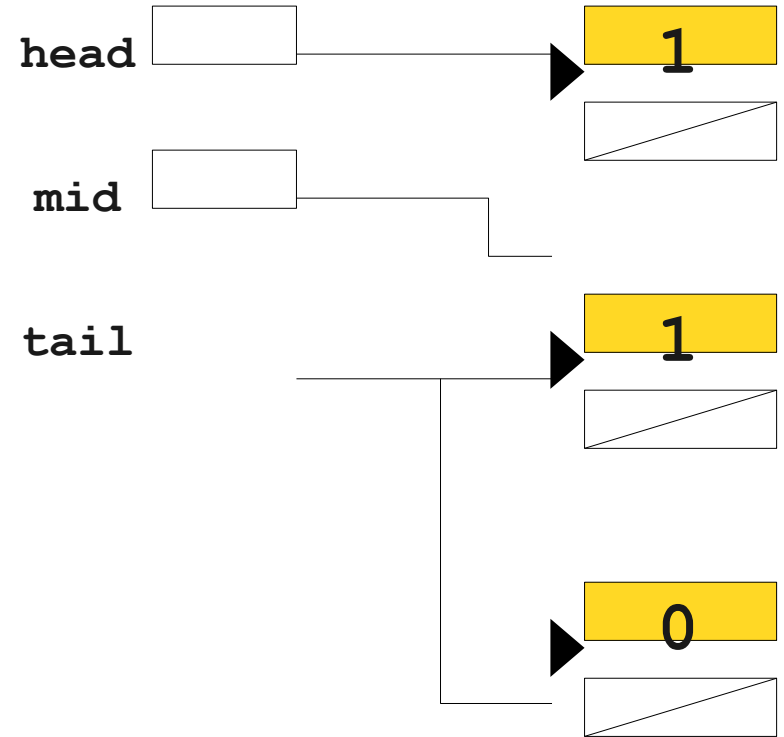
  head.next.next = null;

  head = null;
}
```



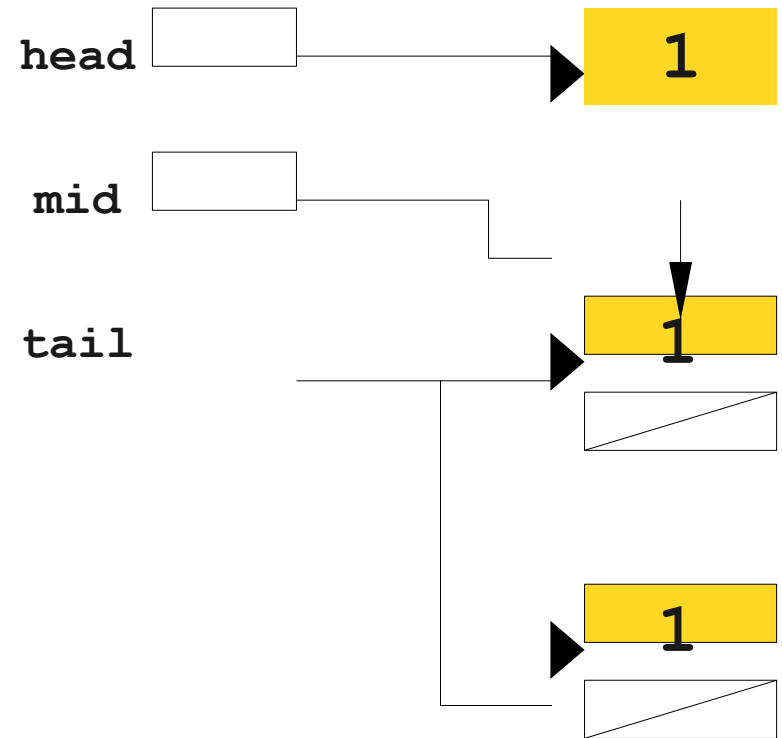
Reference Counting in Action

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



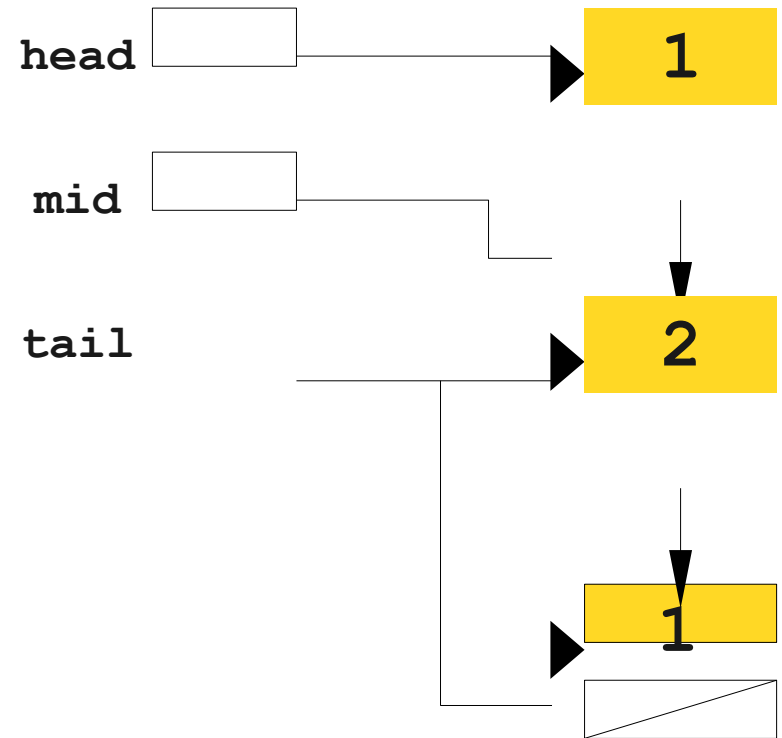
Reference Counting in Action

```
class LinkedList {  
} LinkedList next;  
  
int main()      head = new LinkedList;  
{ LinkedList  
  st  
  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()      head = new LinkedList;  
{ LinkedLi  
st  
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

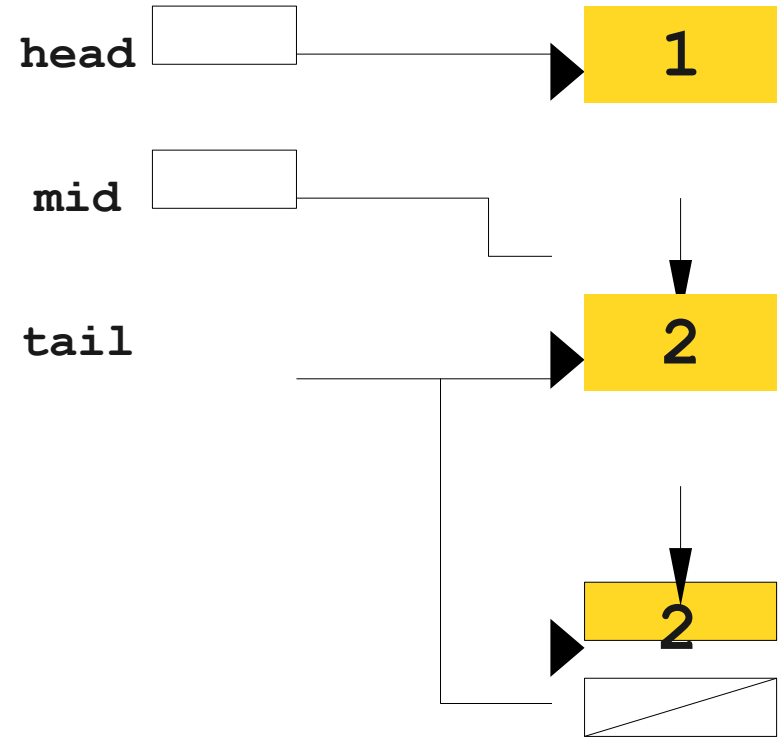
```
class LinkedList {
} LinkedList next;

int main()      head = new LinkedList;
{ LinkedList
  st
  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

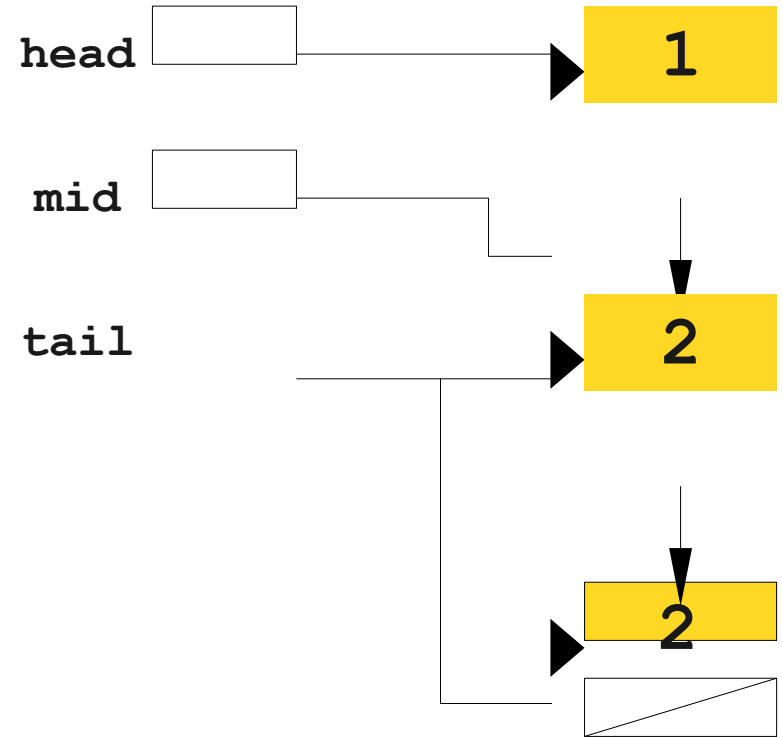
```
class LinkedList {
} LinkedList next;

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

  mid = tail = null;

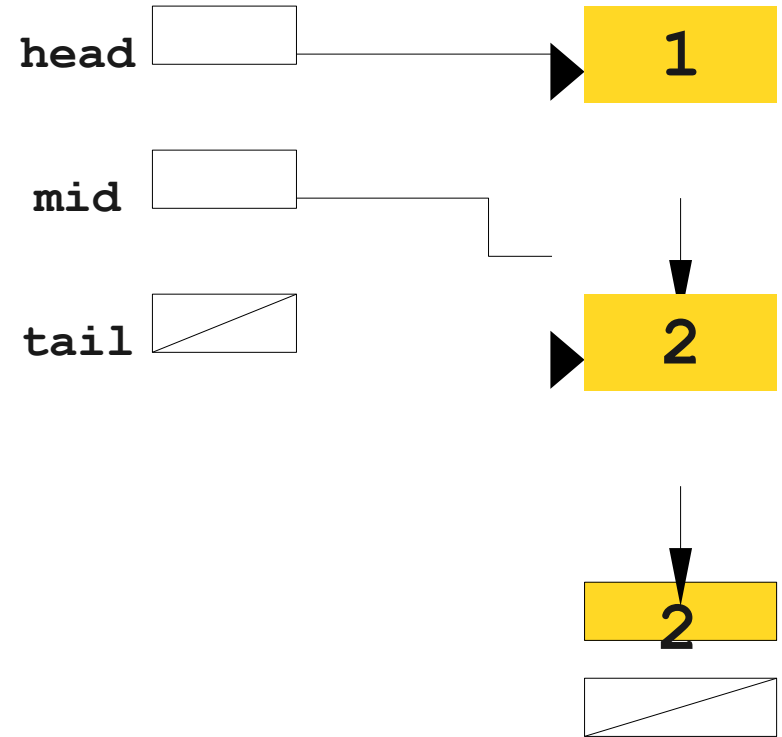
  head.next.next = null;

  head = null;
}
```



Reference Counting in Action

```
class LinkedList {  
} LinkedList next;  
  
int main()      head = new LinkedList;  
{ LinkedList  
  st  
  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

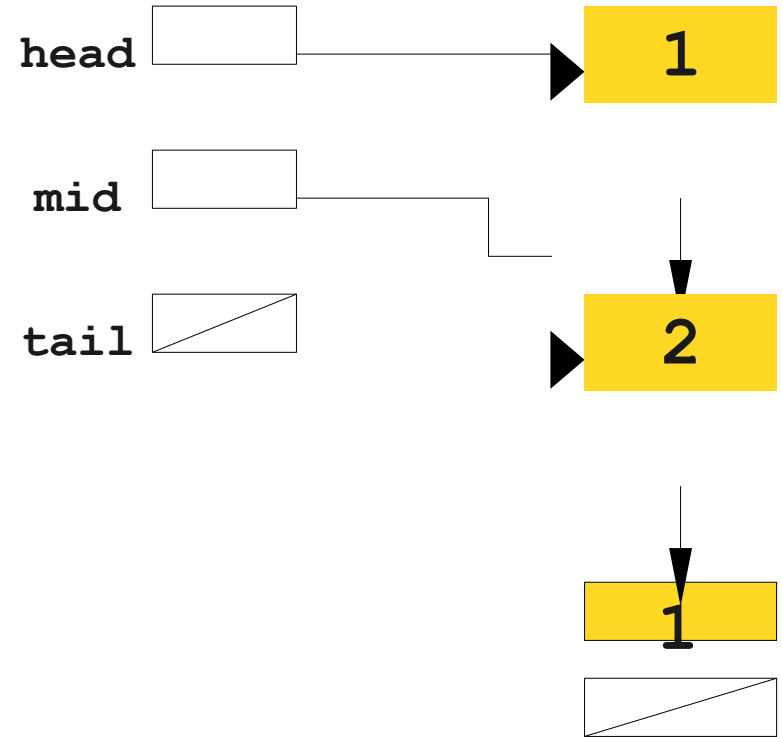
```
class LinkedList {
} LinkedList next;

int main()      head = new LinkedList;
{ LinkedList
  st
  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

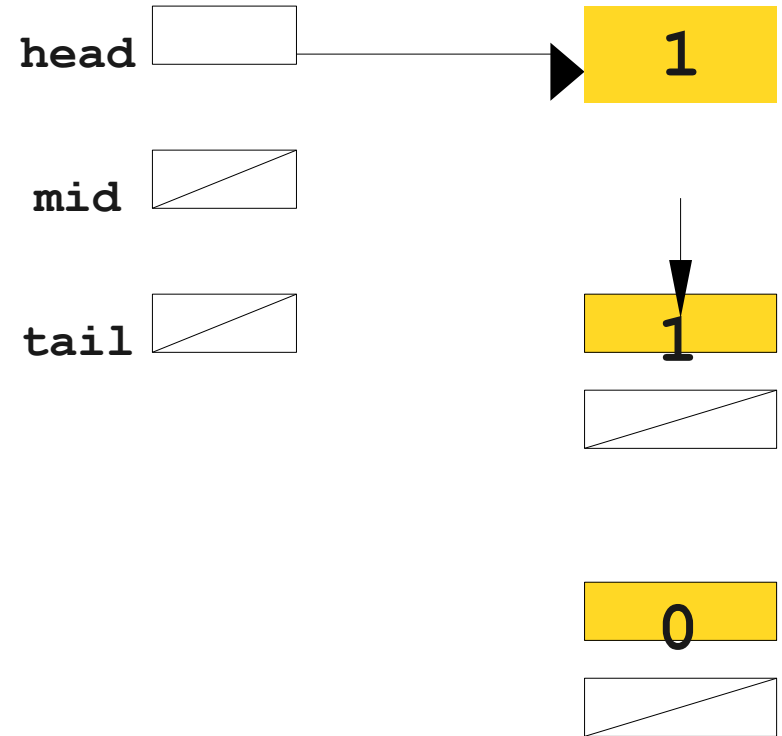
```
class LinkedList {
} LinkedList next;

int main()      head = new LinkedList;
{ LinkedList
  st
  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

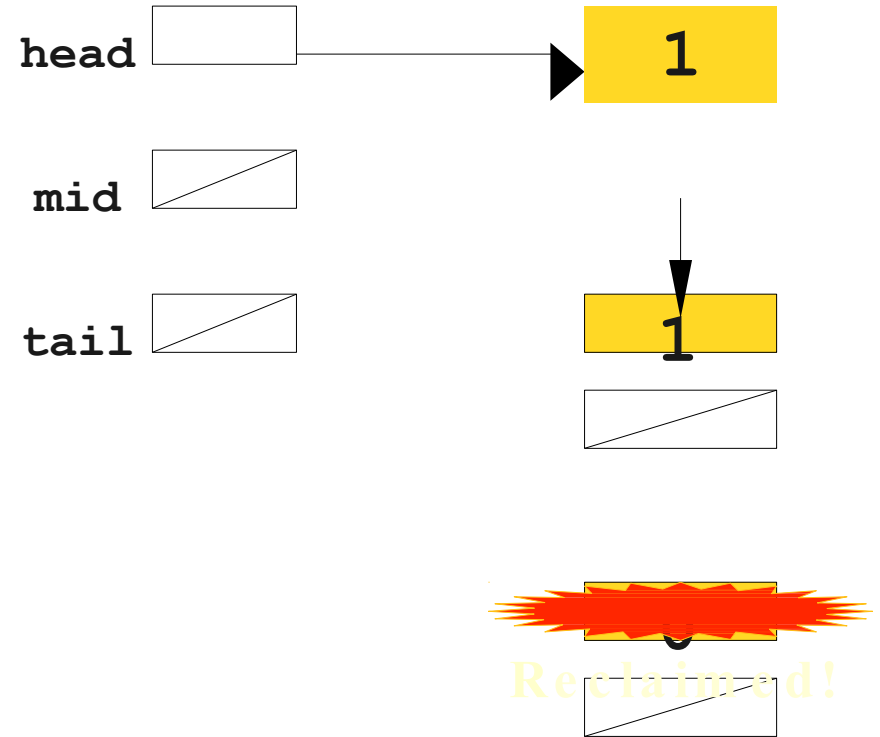
```
class LinkedList {
} LinkedList next;

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

  mid = tail = null;

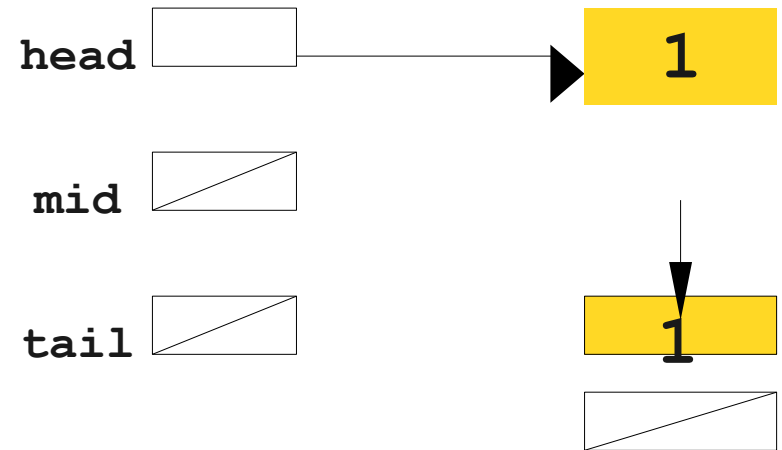
  head.next.next = null;

  head = null;
}
```



Reference Counting in Action

```
class LinkedList {  
} LinkedList next;  
  
int main()      head = new LinkedList;  
{ LinkedLi  
st  
  
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

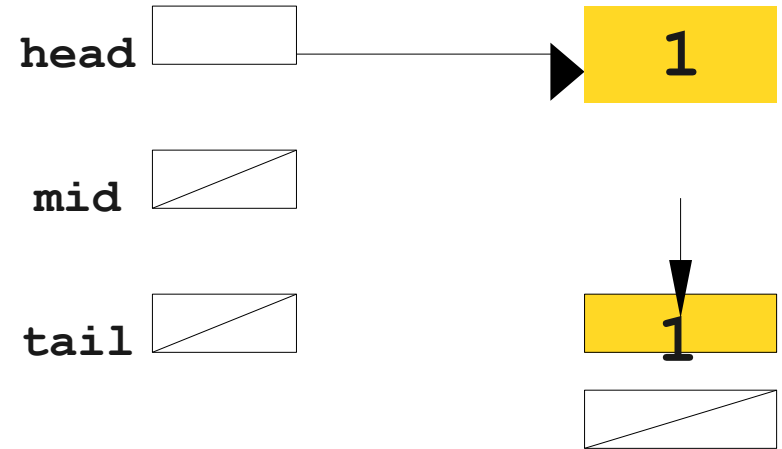
```
class LinkedList {
} LinkedList next;

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

  mid = tail = null;

  head.next.next = null;

  head = null;
}
```

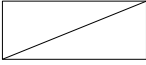


Reference Counting in Action

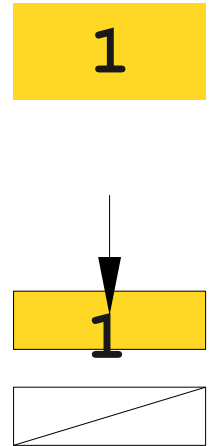
```
class LinkedList {  
} LinkedList next;
```

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

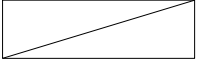
head 

mid 

tail 

0

↓
1



Reference Counting in Action

```
class LinkedList
{ LinkedList
  next;
}

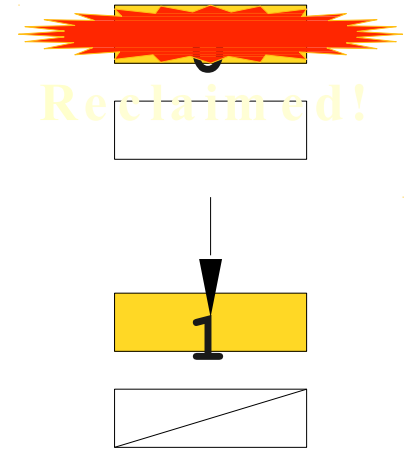
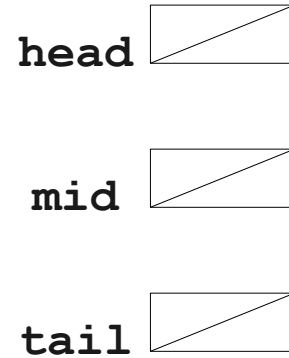
int main() {
  LinkedList head = new LinkedList;
  LinkedList mid = new LinkedList;
  LinkedList tail = new LinkedList;

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

  mid = tail = null;

  head.next.next = null;

  head = null;
}
```

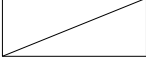


Reference Counting in Action

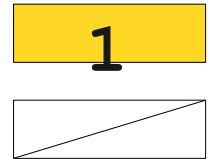
```
class LinkedList {  
}    LinkedList next;
```

```
int main()    head = new LinkedList;  
{ LinkedLi  
st  
  
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 

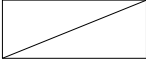


Reference Counting in Action

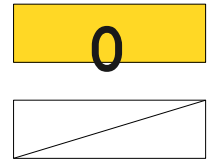
```
class LinkedList {  
} LinkedList next;
```

```
int main()      head = new LinkedList;  
{ LinkedLi  
st  
  
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 



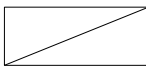
Reference Counting in Action

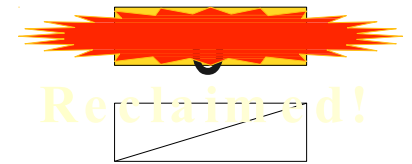
```
class LinkedList {  
} LinkedList next;
```

```
int main()      head = new LinkedList;  
{ LinkedLi  
st  
  
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;

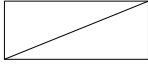
    head.next = mid;
    mid.next = tail;

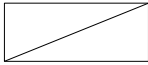
    mid = tail = null;

    head.next.next = null;

    head = null;
}
```

head 

mid 

tail 

Reference Counting: 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;
}
```

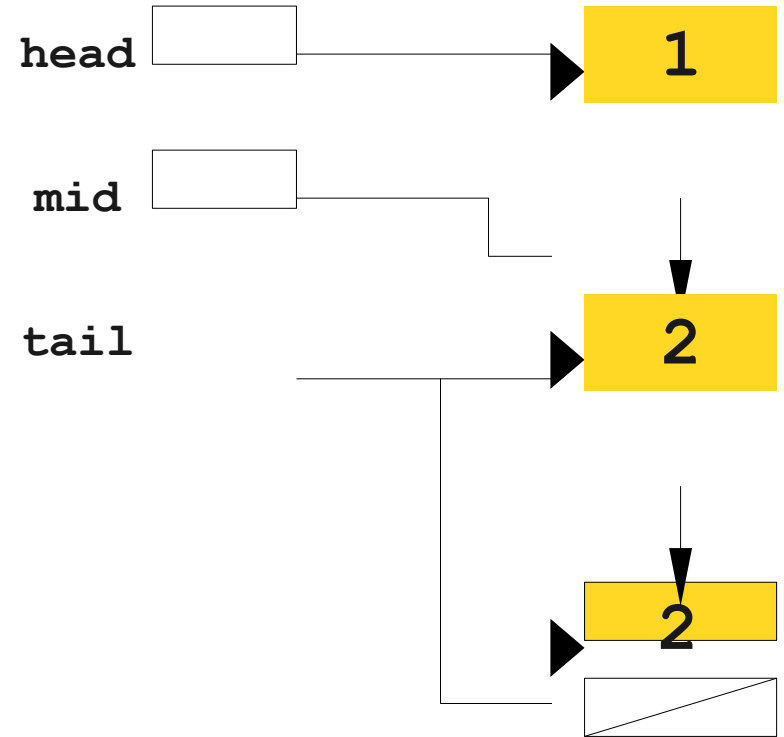
One Major Problem

```
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;
}
```



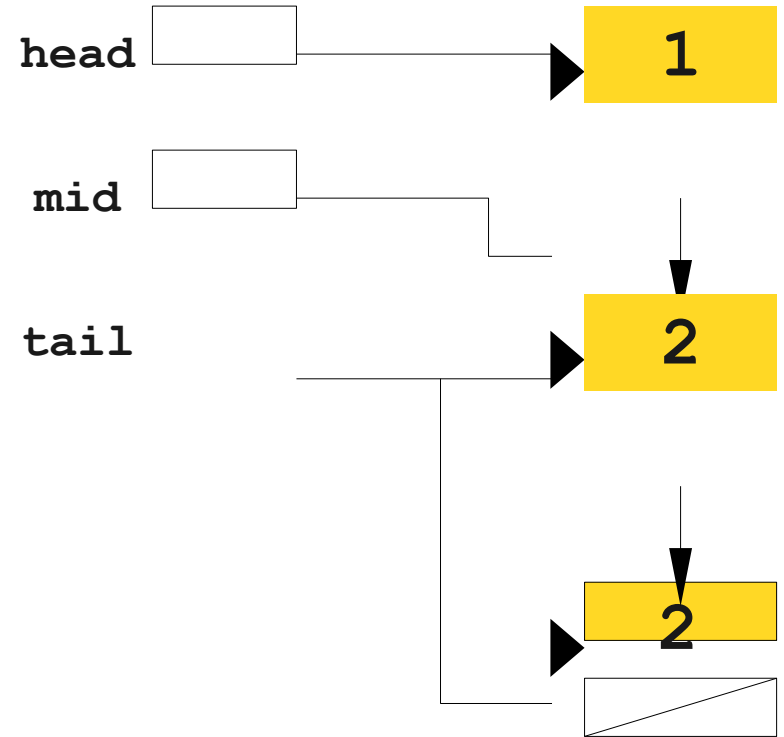
One Major Problem

```
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;
}
```



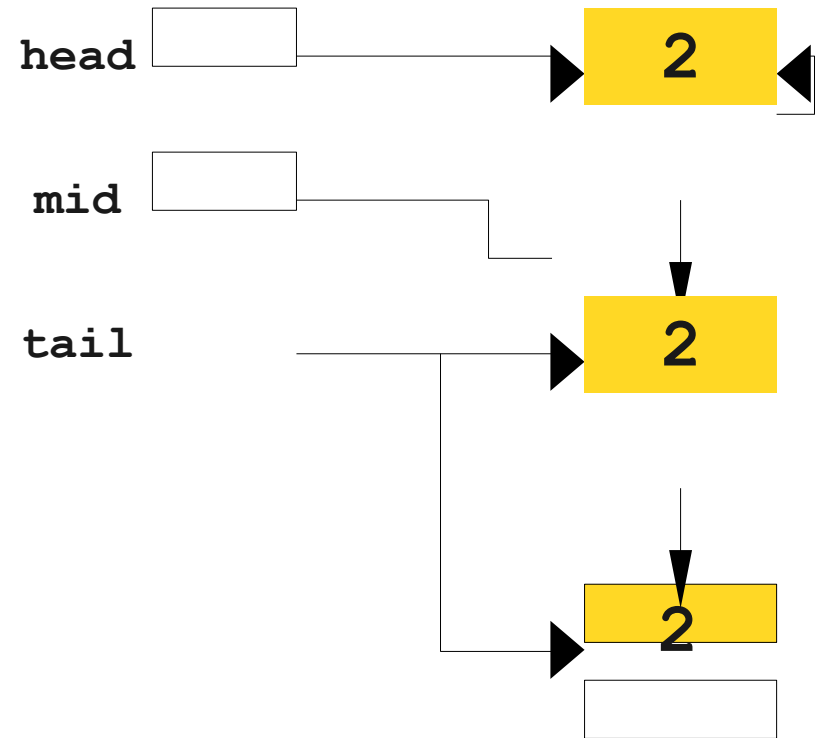
One Major Problem

```
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;
}
```



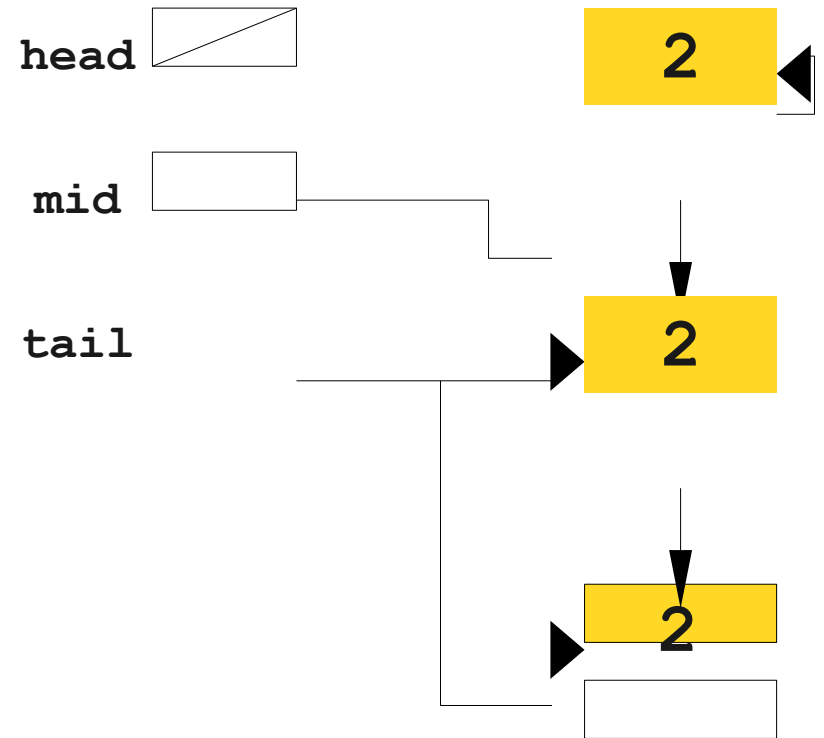
One Major Problem

```
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;
}
```



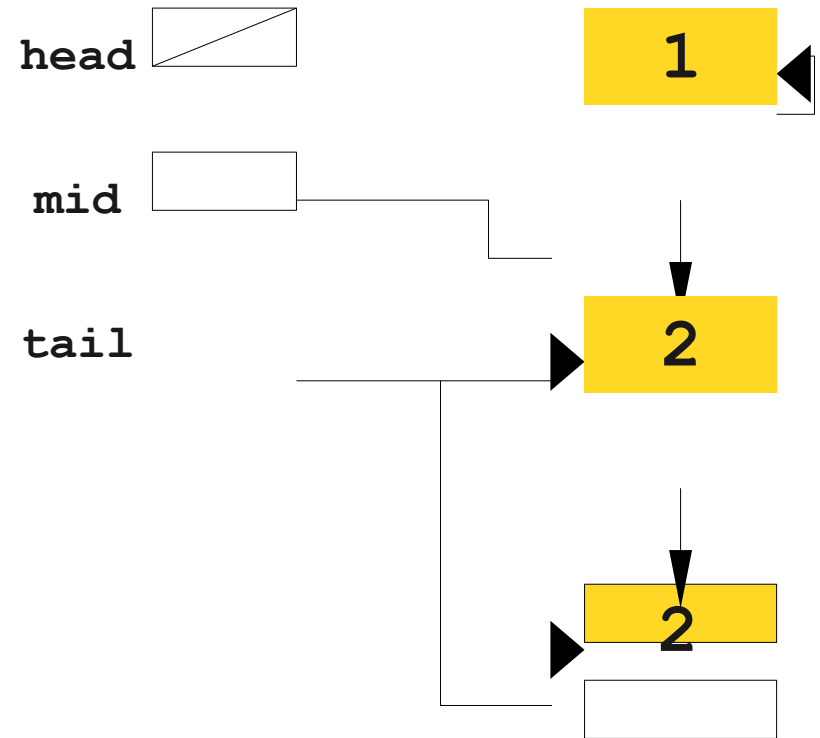
One Major Problem

```
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;
}
```



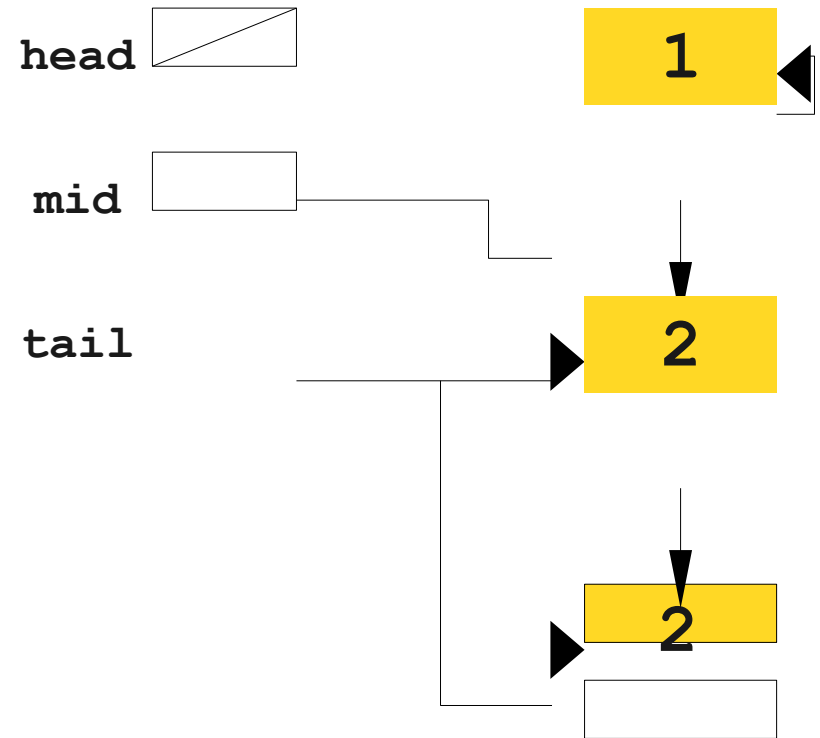
One Major Problem

```
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;
}
```



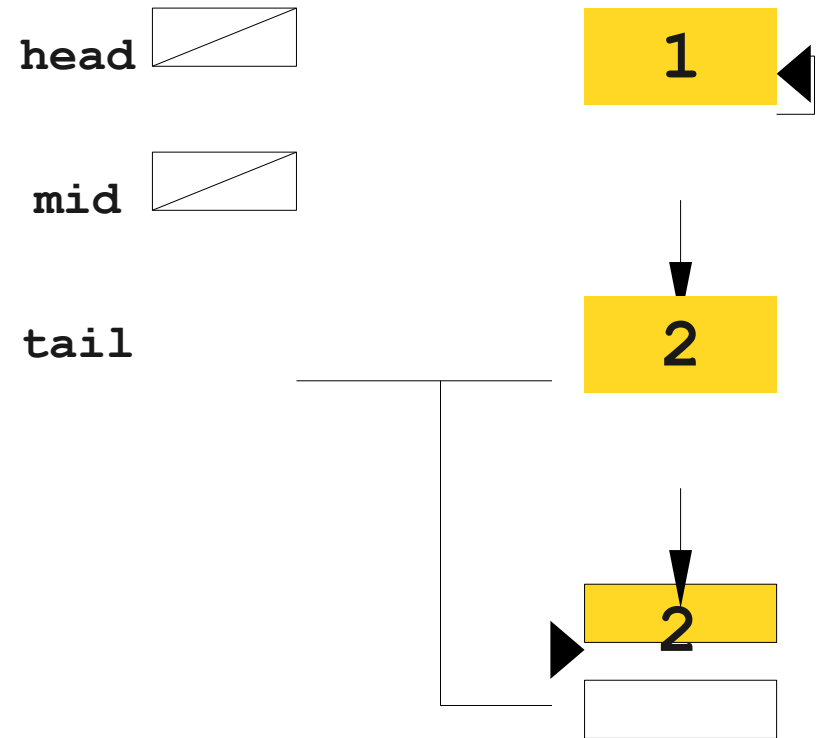
One Major Problem

```
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;
}
```



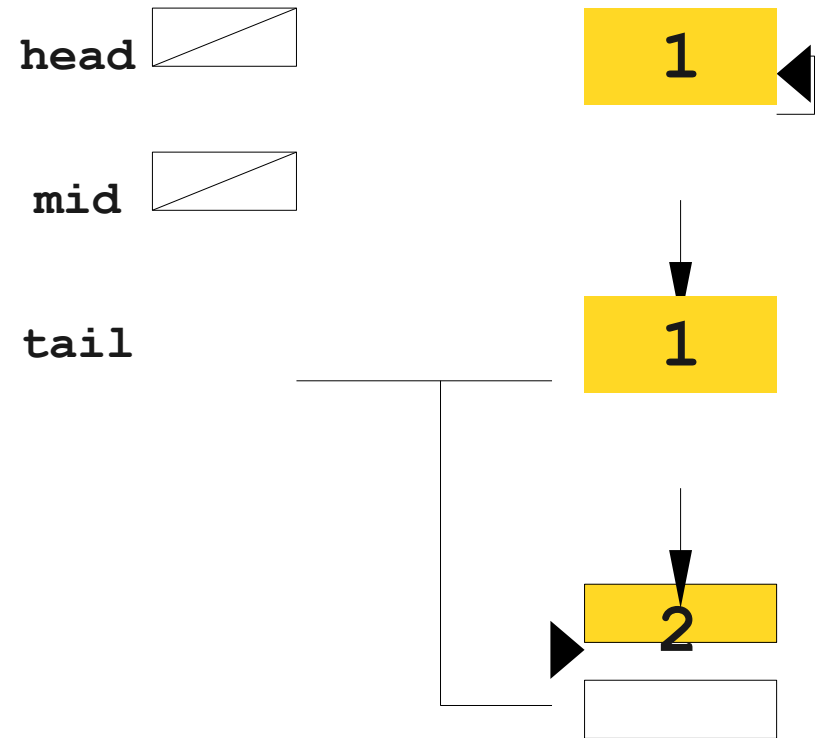
One Major Problem

```
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;
}
```



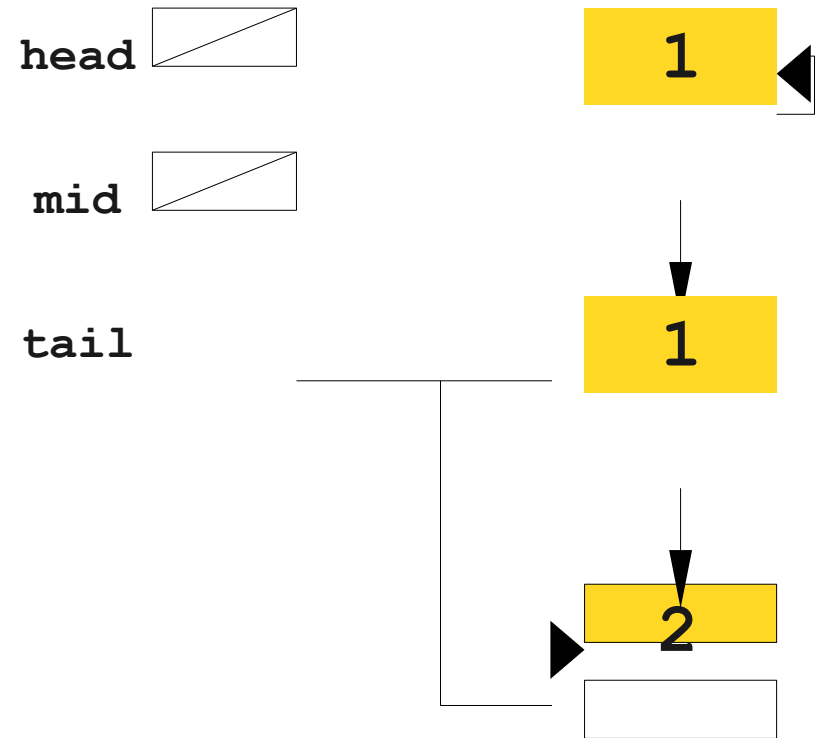
One Major Problem

```
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;
}
```



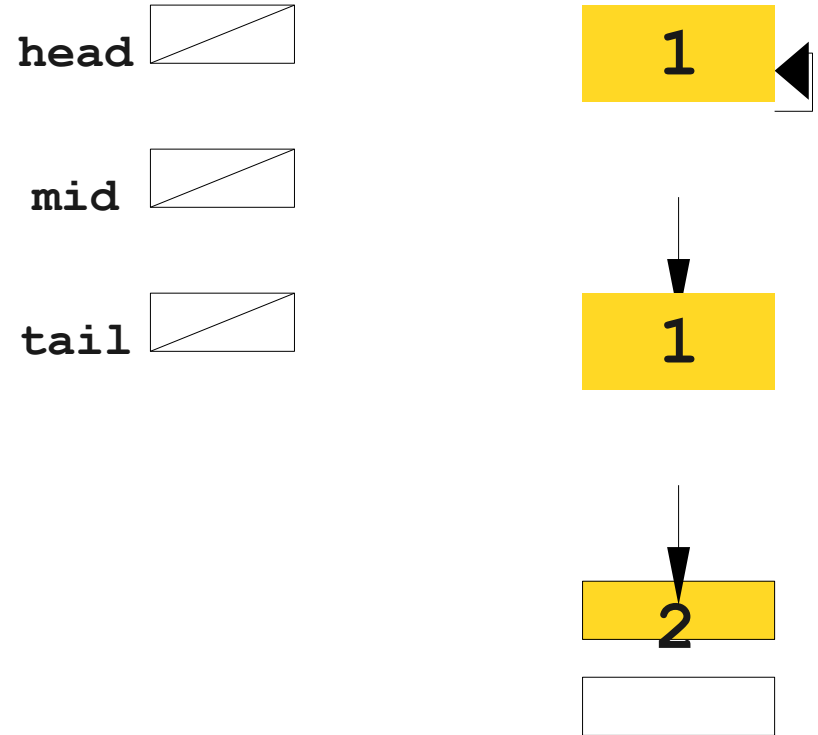
One Major Problem

```
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;
}
```



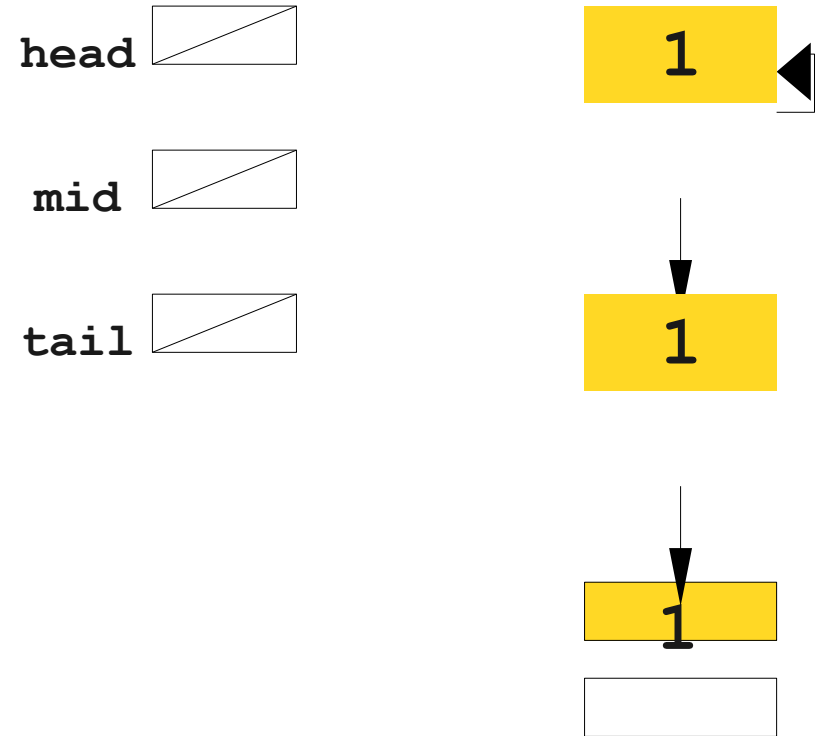
One Major Problem

```
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;
}
```



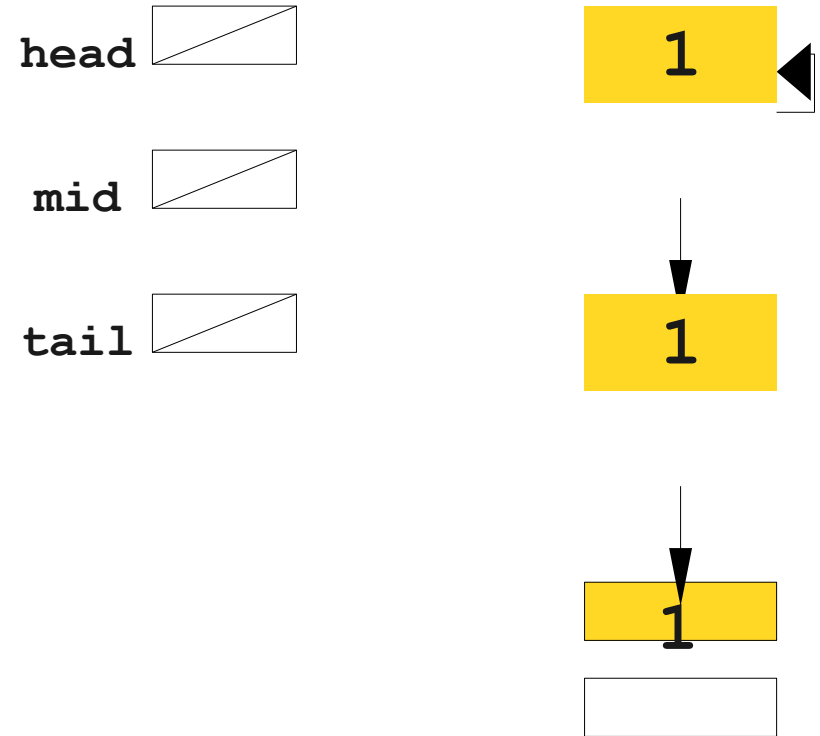
One Major Problem

```
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;
}
```



Does not Reclaim 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.

- **Disadvantages**

- Fails to reclaim all unreachable objects.
- Can be slow if a large collection is initiated.
- Noticeably slows down assignments.

Mark-and-Sweep

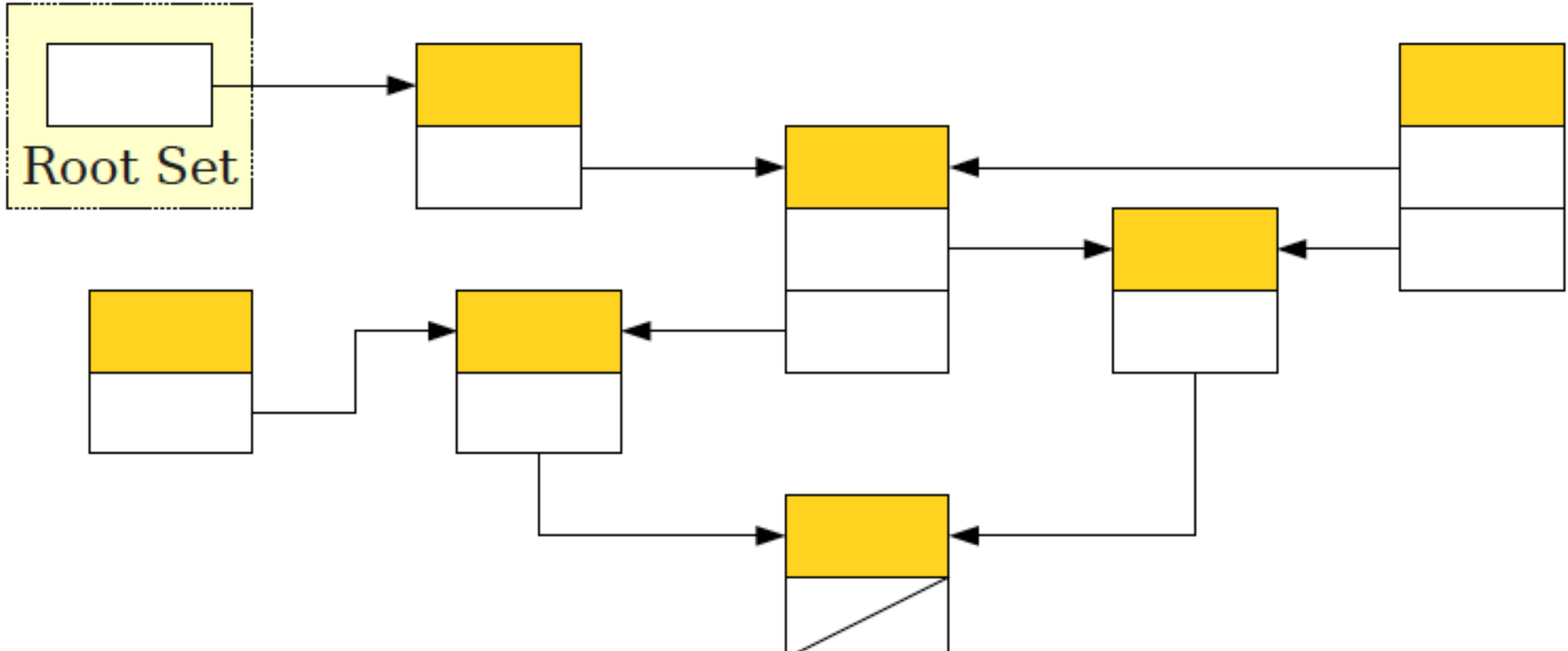
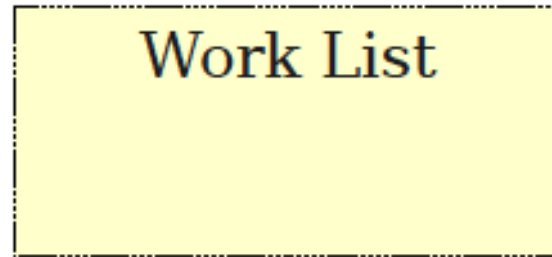
Mark-and-Sweep Technique

- Given knowledge of what's immediately accessible, find everything reachable in the program.
 - **Root Set** - set of memory locations in the program that are known immediately to be reachable.
 - Any objects reachable from the root set are reachable.
 - Any objects not reachable from the root set are not reachable.
- > Graph search starting at the root set

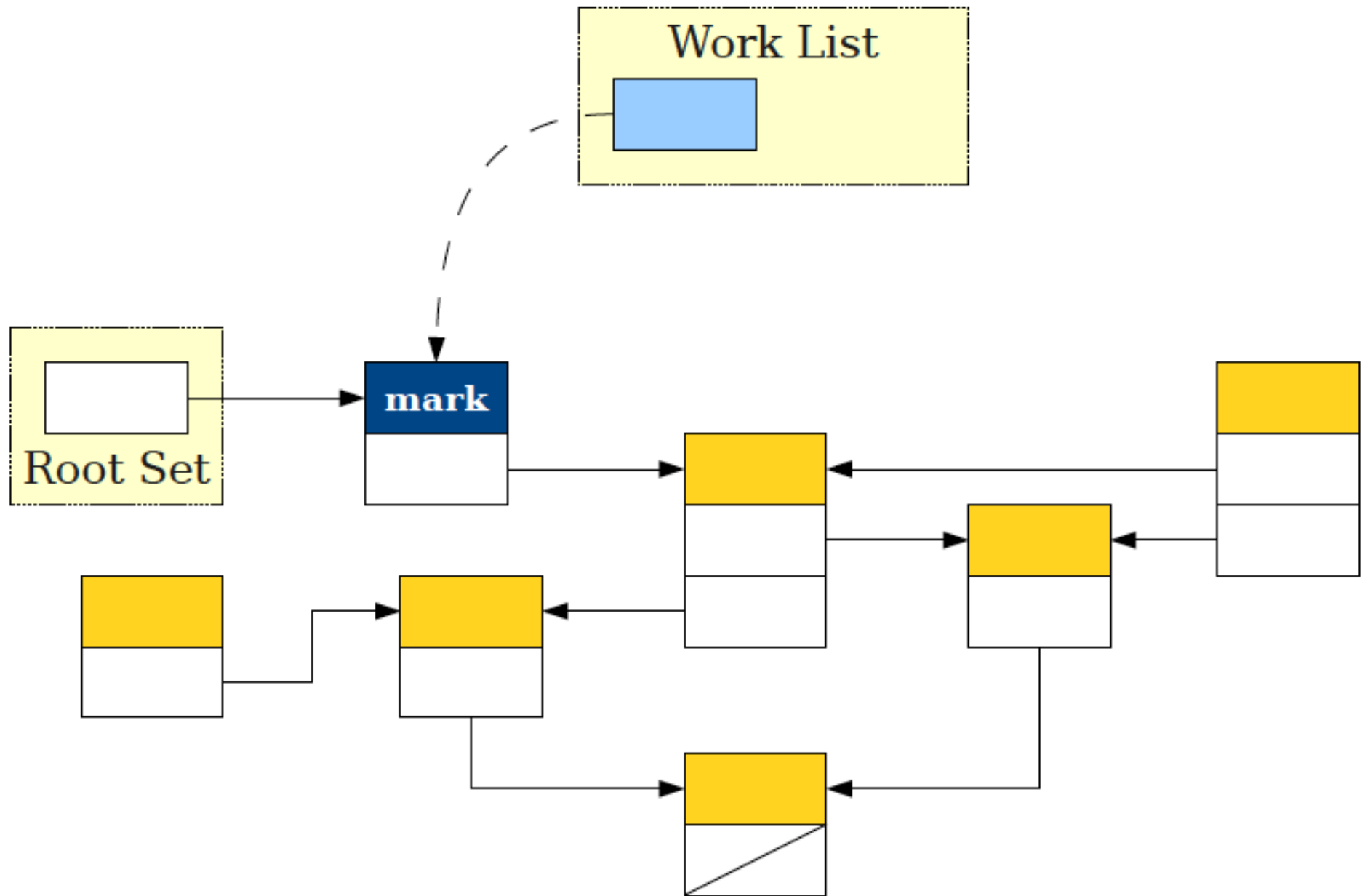
Mark-and-Sweep: The Algorithm

- **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 not marked, mark 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).

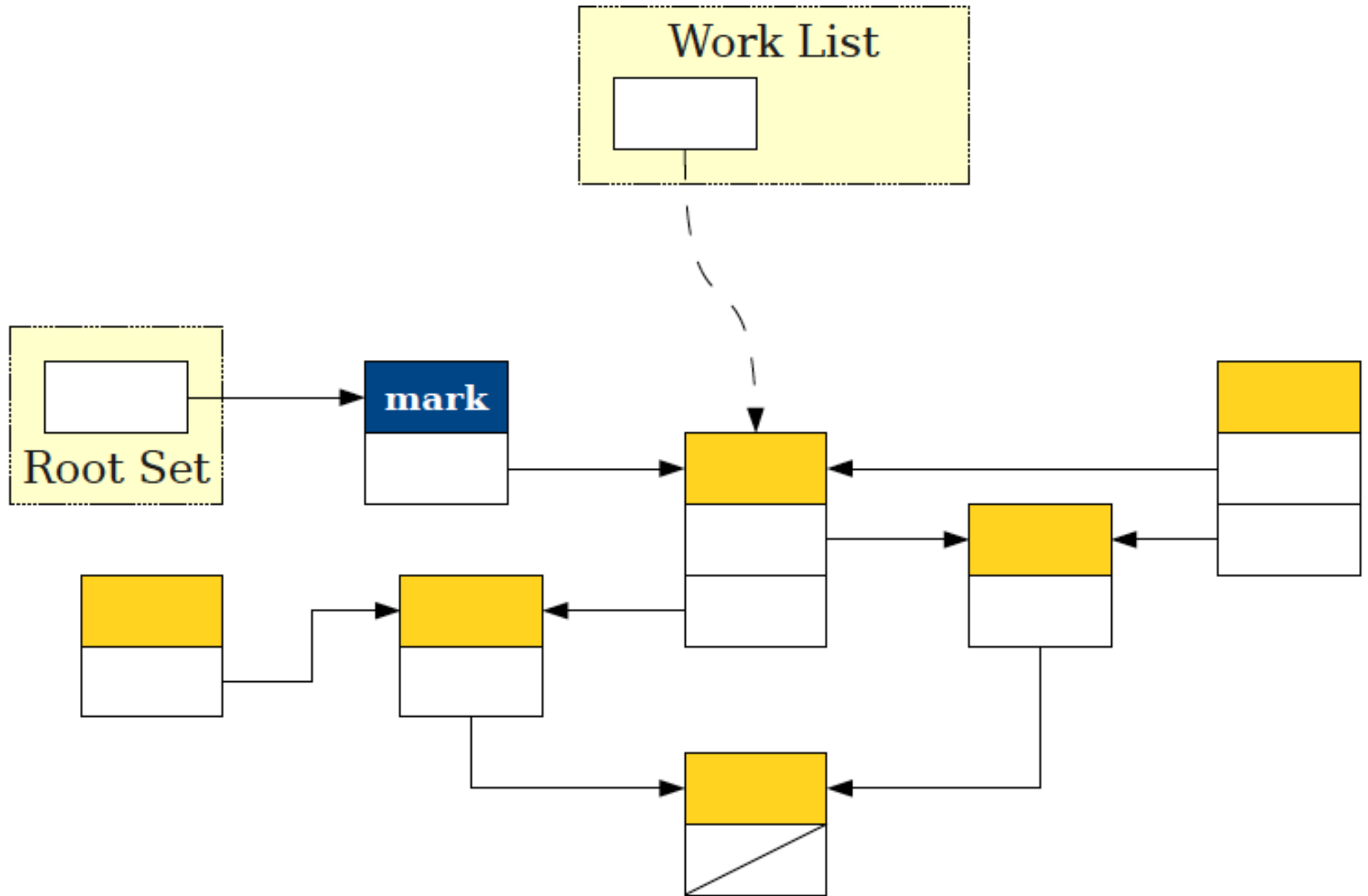
Marking Phase



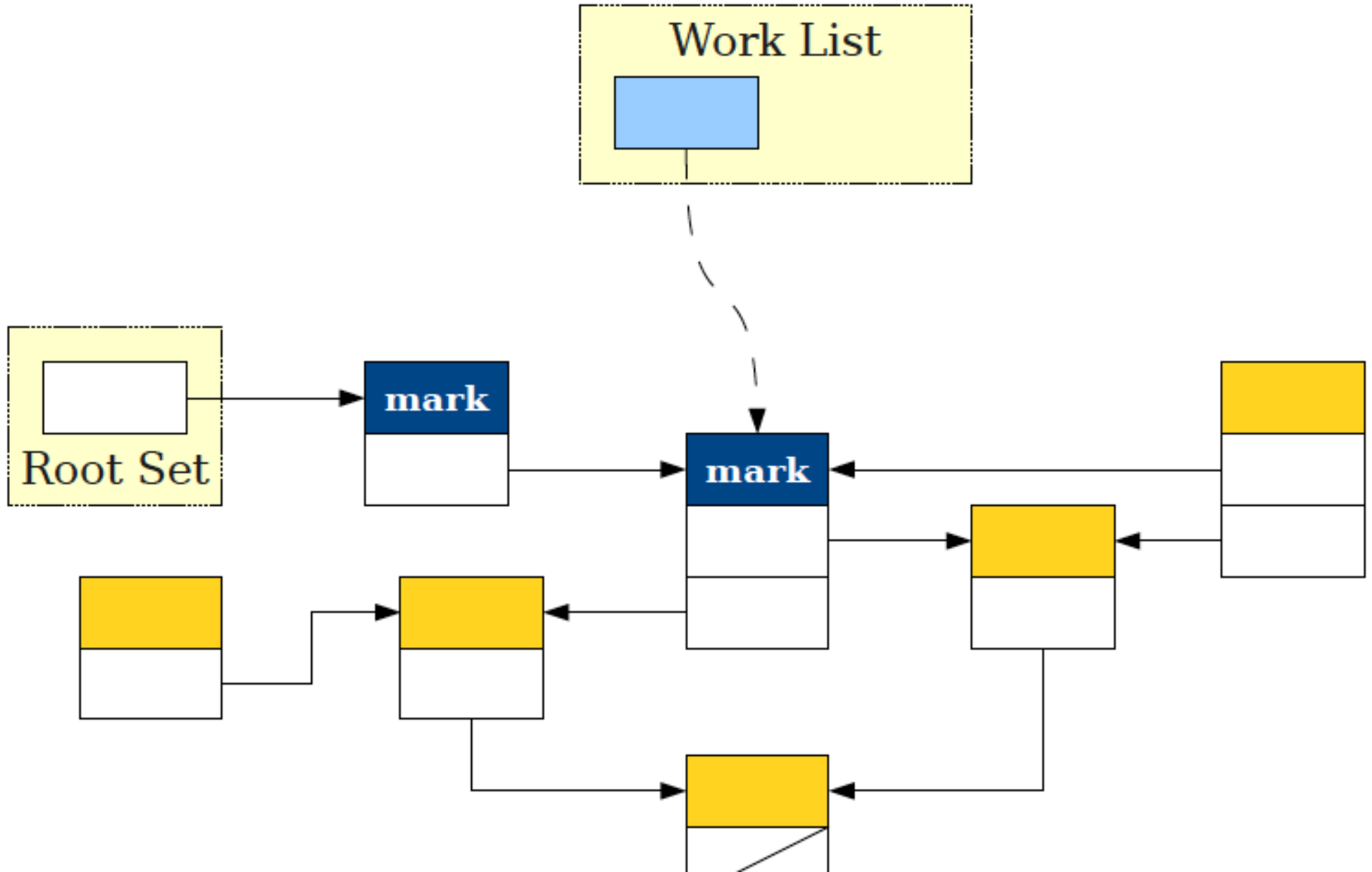
Marking Phase



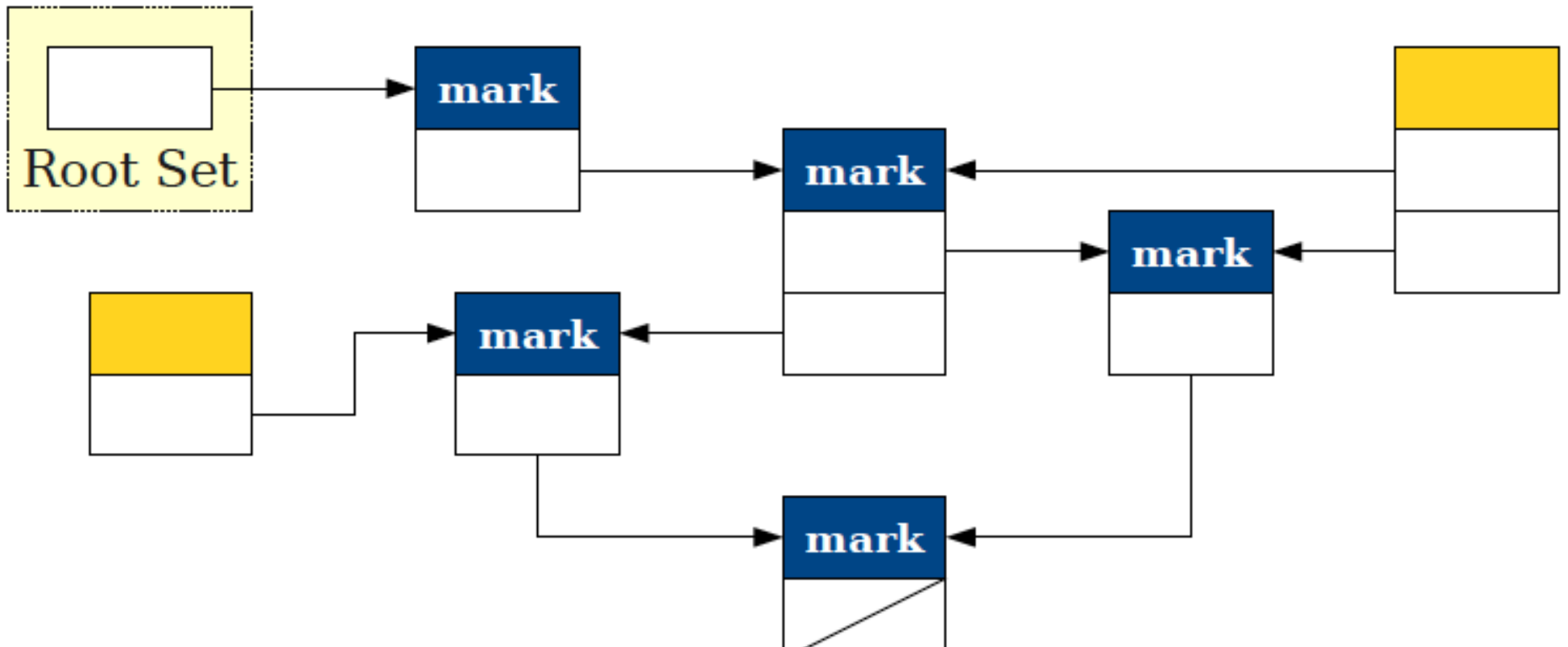
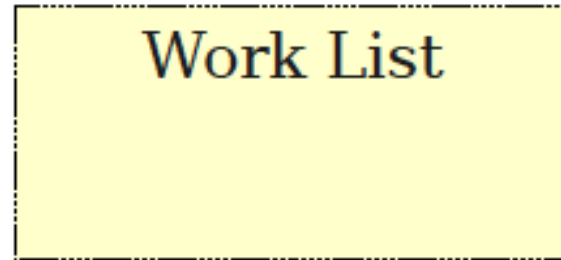
Marking Phase



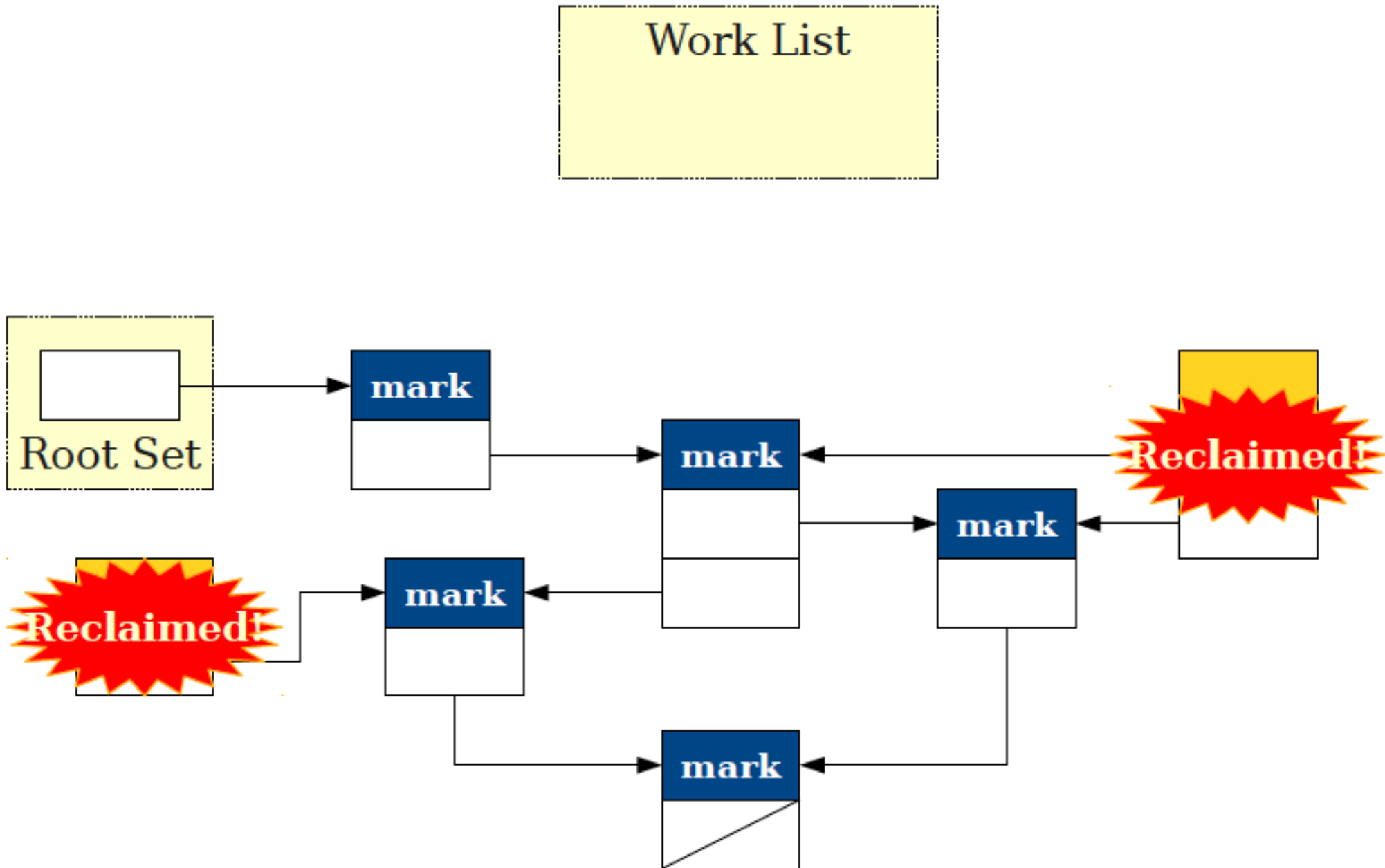
Marking Phase



At End of Marking Phase



Sweep Phase: Sweep Entire Heap



Mark-and-Sweep Problems

Sweep phase visits all objects to free them or clear marks.

Amount of space for worklist could potentially be as large as all of memory.

Can't preallocate this space.

Implementation Details

- 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.

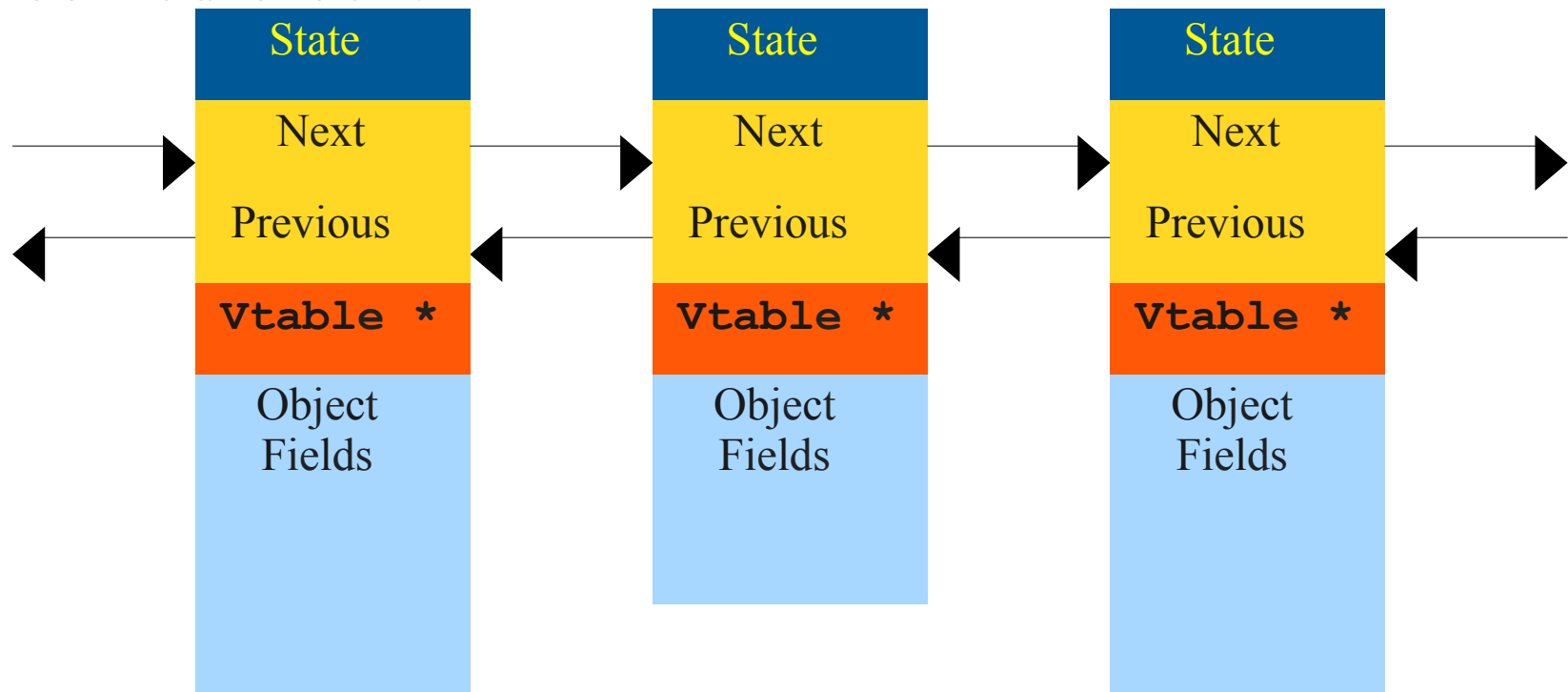
Maintain doubly-linked lists of all the objects in each of these states.

More Efficient: Baker's Algorithm

- Move entire Root set to the **enqueued** list.
- While **enqueued** list is not empty:
 - Move first object from **enqueued** list to **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 **unknown** and **deallocated** lists
 - Deallocates all garbage in $O(1)$.
- Move everything from **marked** list to **unknown** list.
 - Can be done in $O(1)$.
 - Indicates objects again must be proven reachable on next scan.

One Last Detail

- 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 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 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

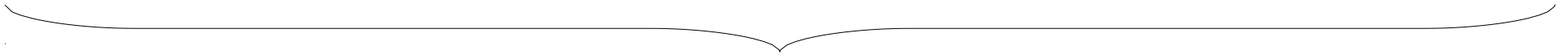
- **Idea:** During garbage collection, move all objects in memory so they are adjacent to one another.
 - -> **compaction**.
- Ideally, move objects that reference one another into adjacent memory locations.
- Garbage collector must update all pointers in all objects to refer to the new object locations.

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 – 20 assembly instructions.
 - Contrast with stack allocation – just one assembly instruction!

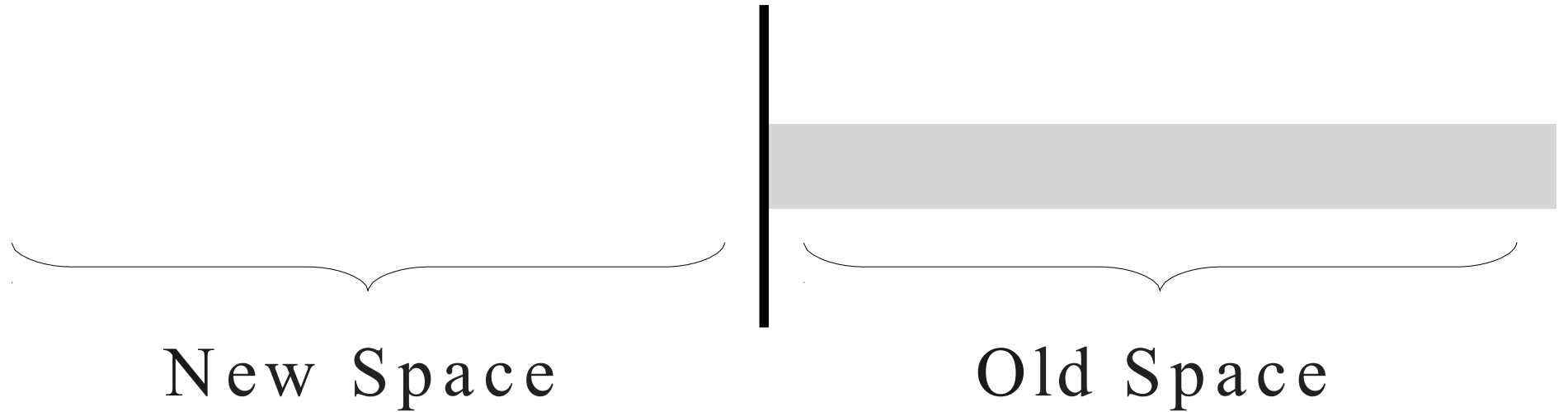
Can we somehow get the performance speed of the stack for dynamic allocation?

The Stop-and-Copy Collector



All of memory

The Stop-and-Copy Collector



The Stop-and-Copy Collector

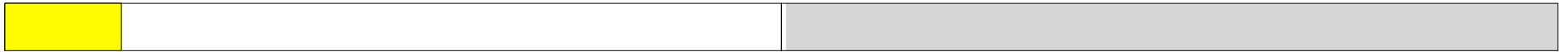
I



**Free
Space**

The Stop-and-Copy Collector

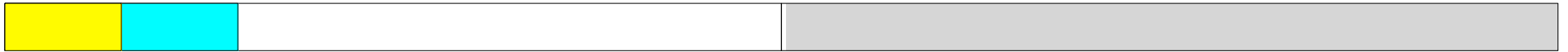
I



**Free
Space**

The Stop-and-Copy Collector

I



**Free
Space**

The Stop-and-Copy Collector

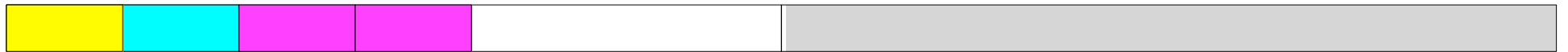
I



**Free
Space**

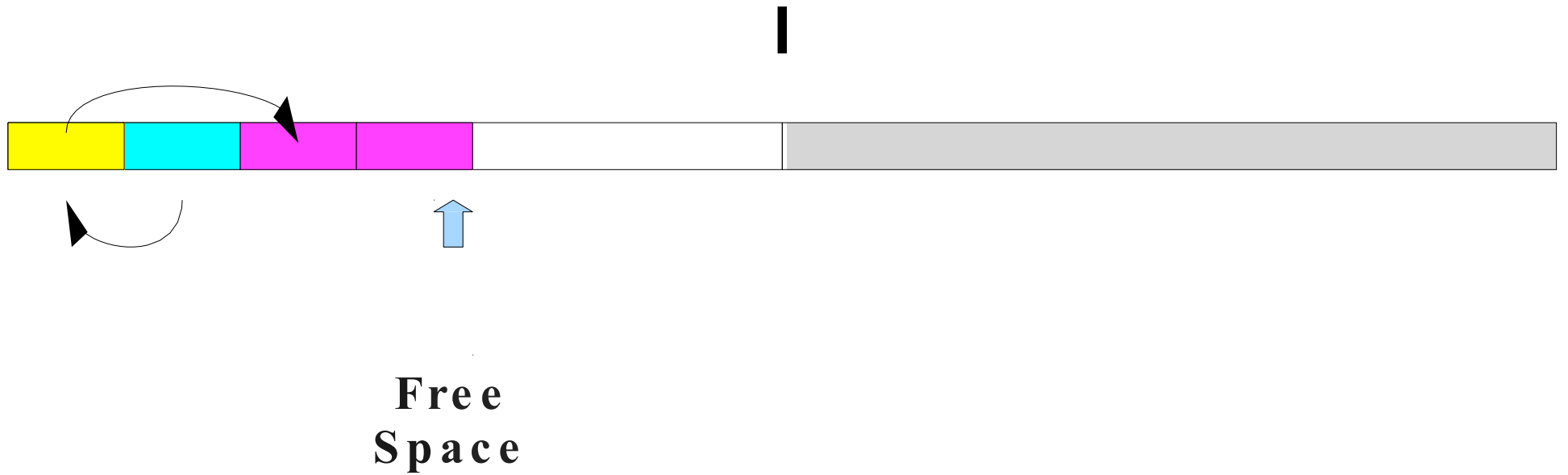
The Stop-and-Copy Collector

I



**Free
Space**

The Stop-and-Copy Collector



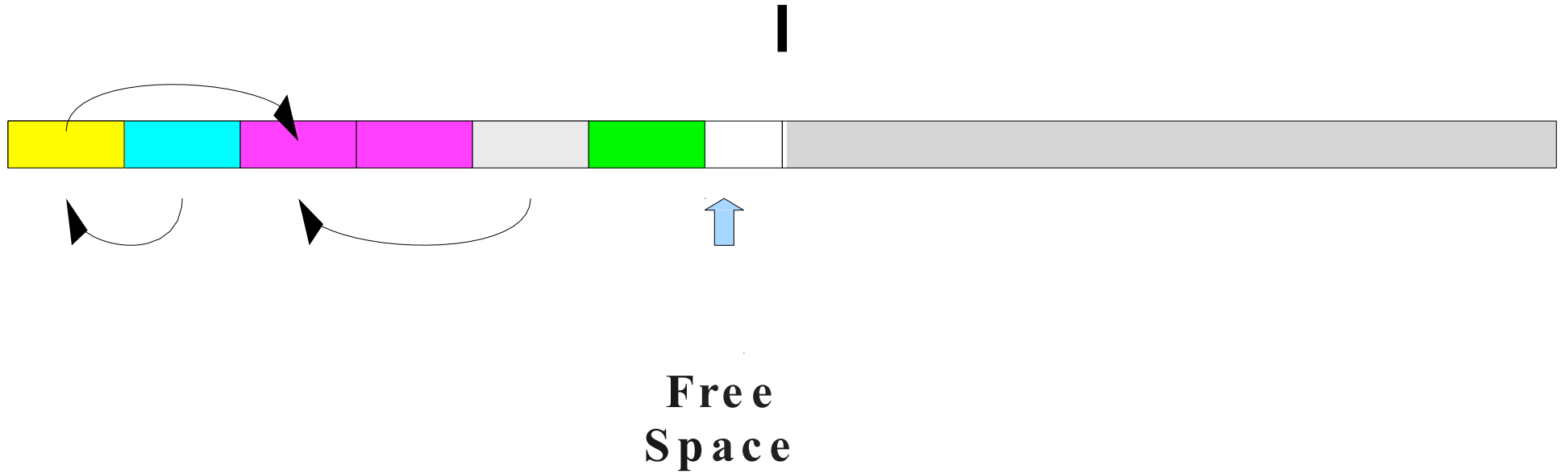
The Stop-and-Copy Collector



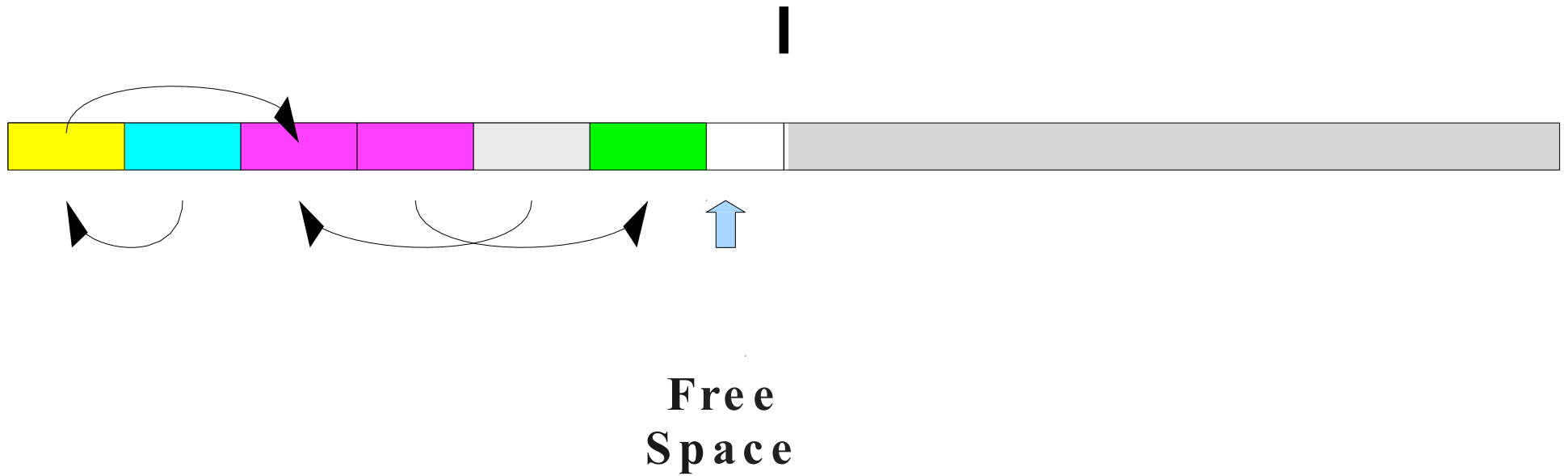
The Stop-and-Copy Collector



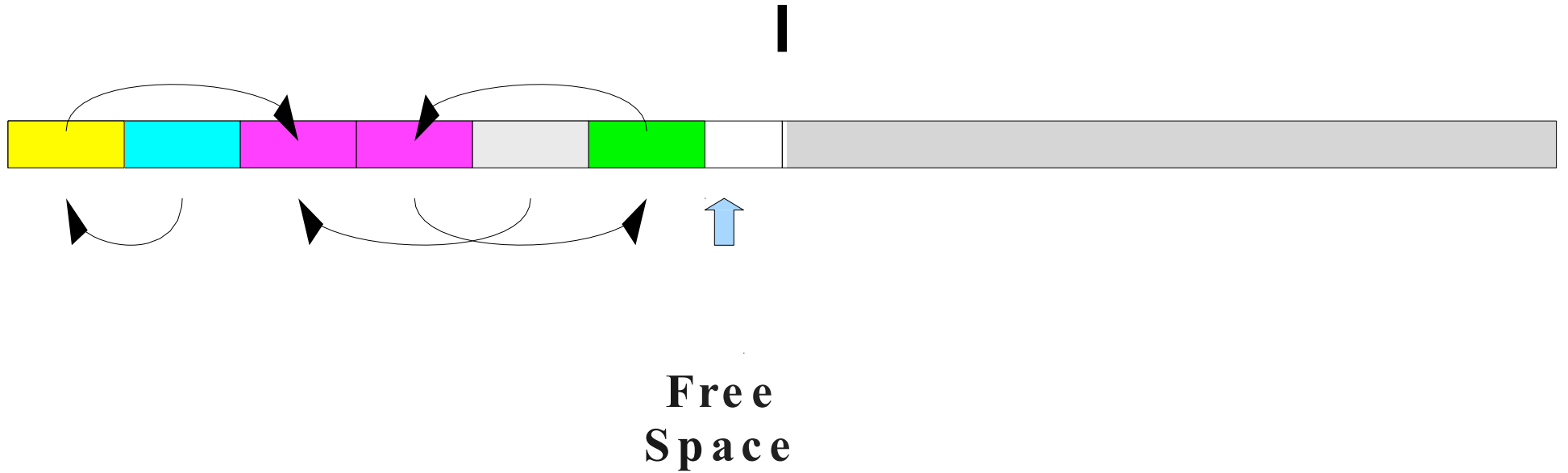
The Stop-and-Copy Collector



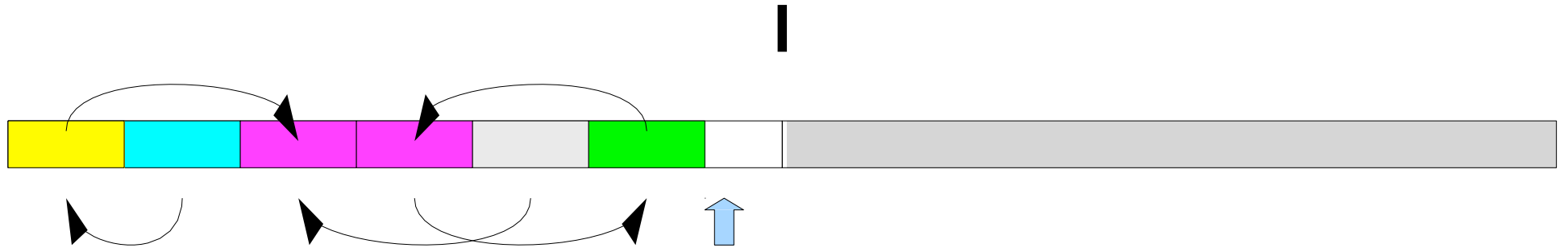
The Stop-and-Copy Collector



The Stop-and-Copy Collector



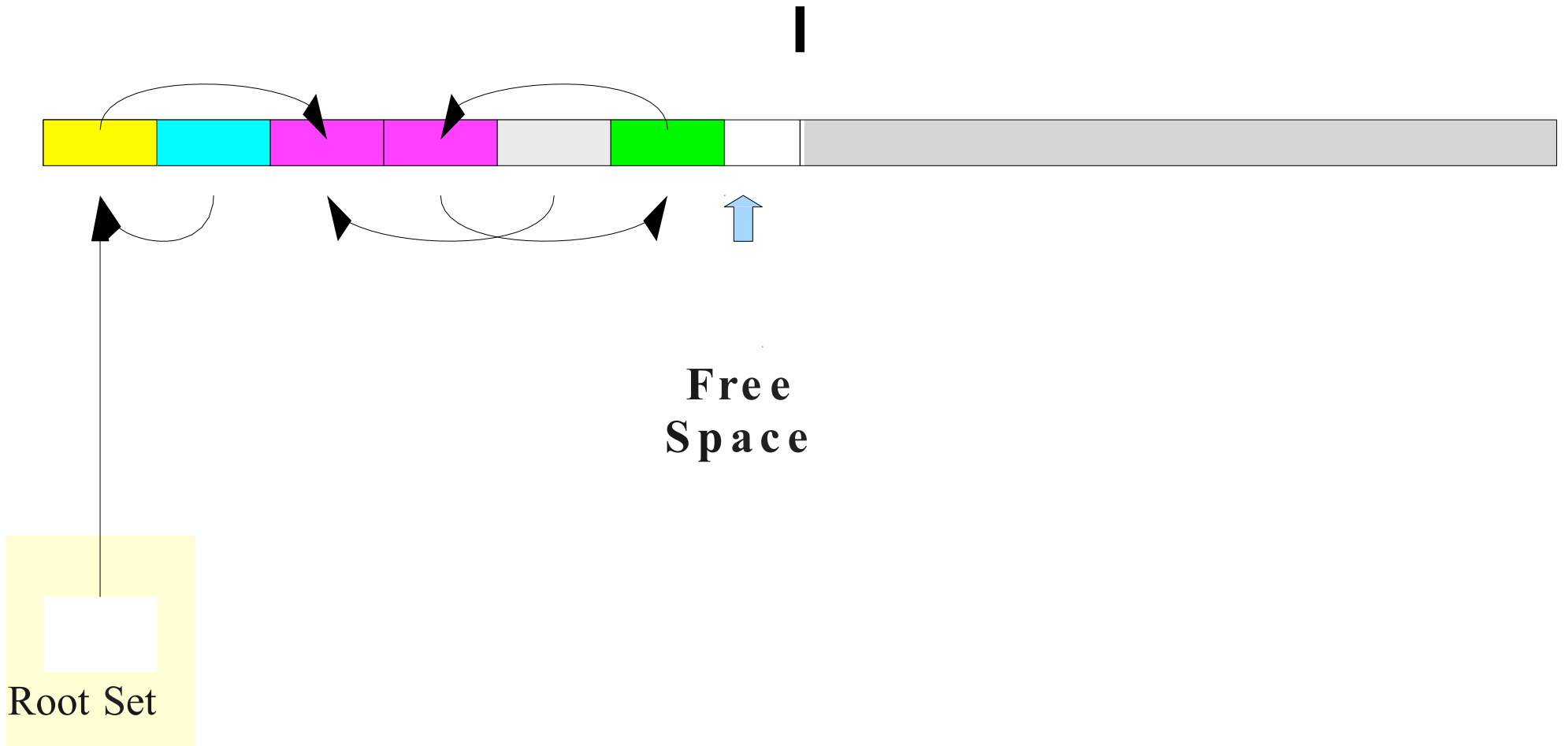
The Stop-and-Copy Collector



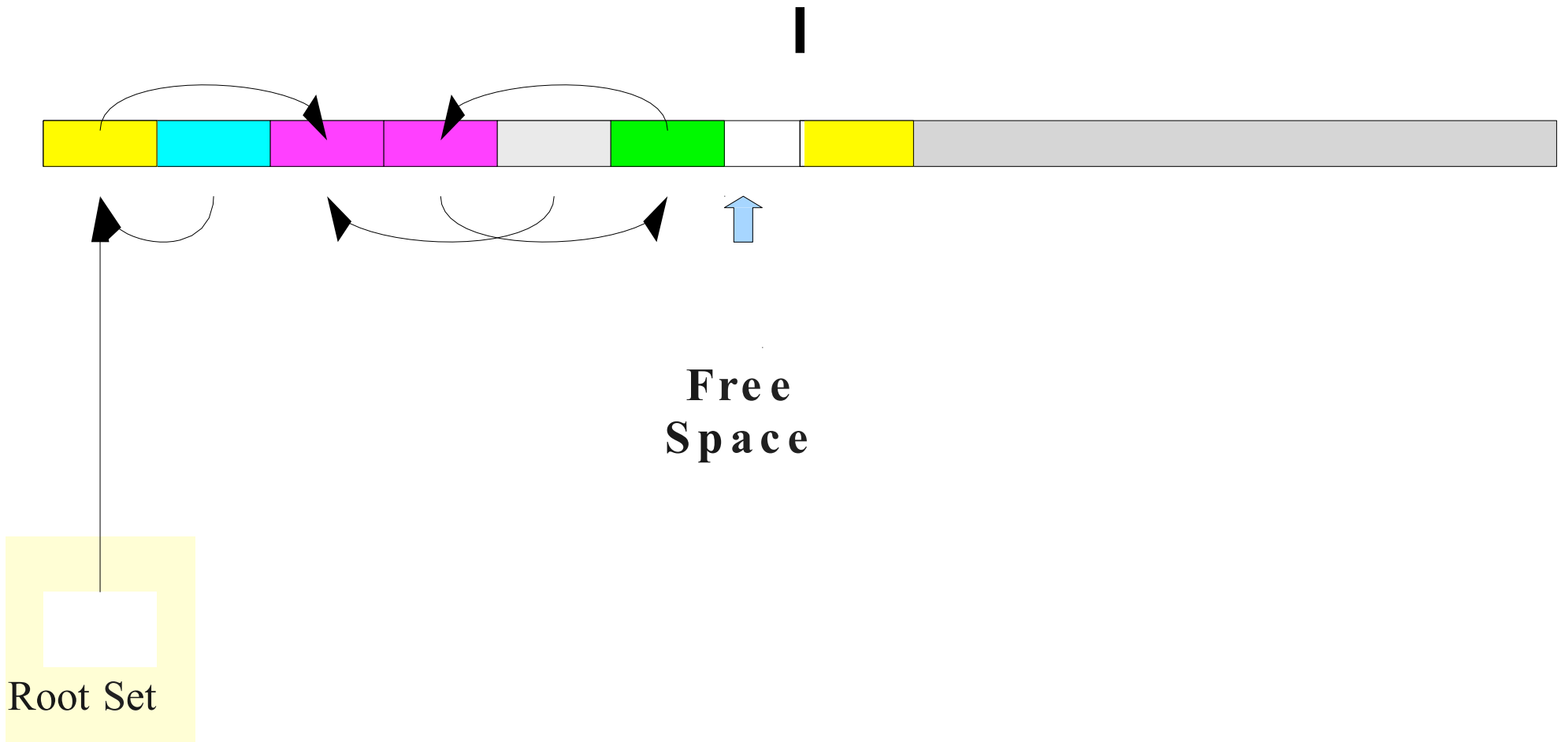
Free
Space

Out of space!

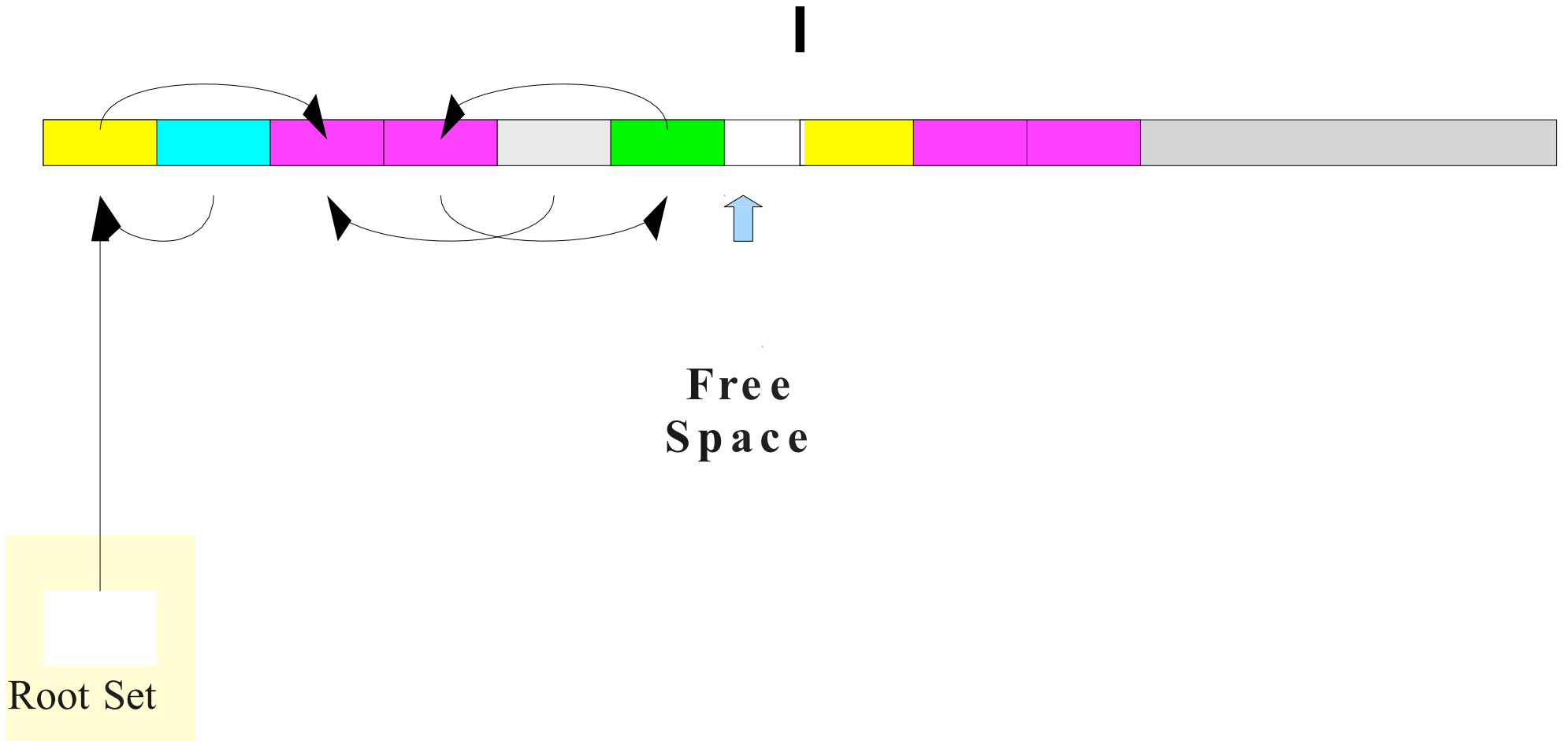
The Stop-and-Copy Collector



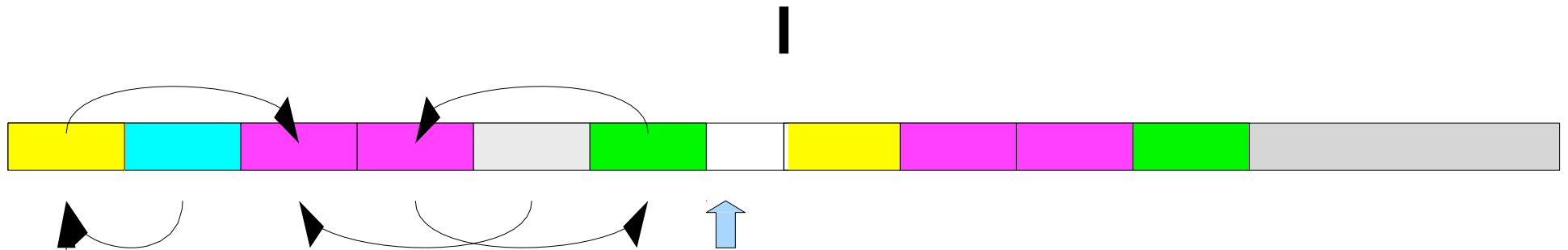
The Stop-and-Copy Collector



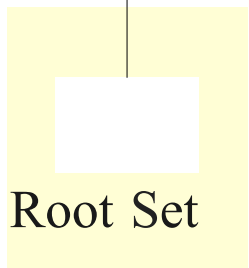
The Stop-and-Copy Collector



The Stop-and-Copy Collector

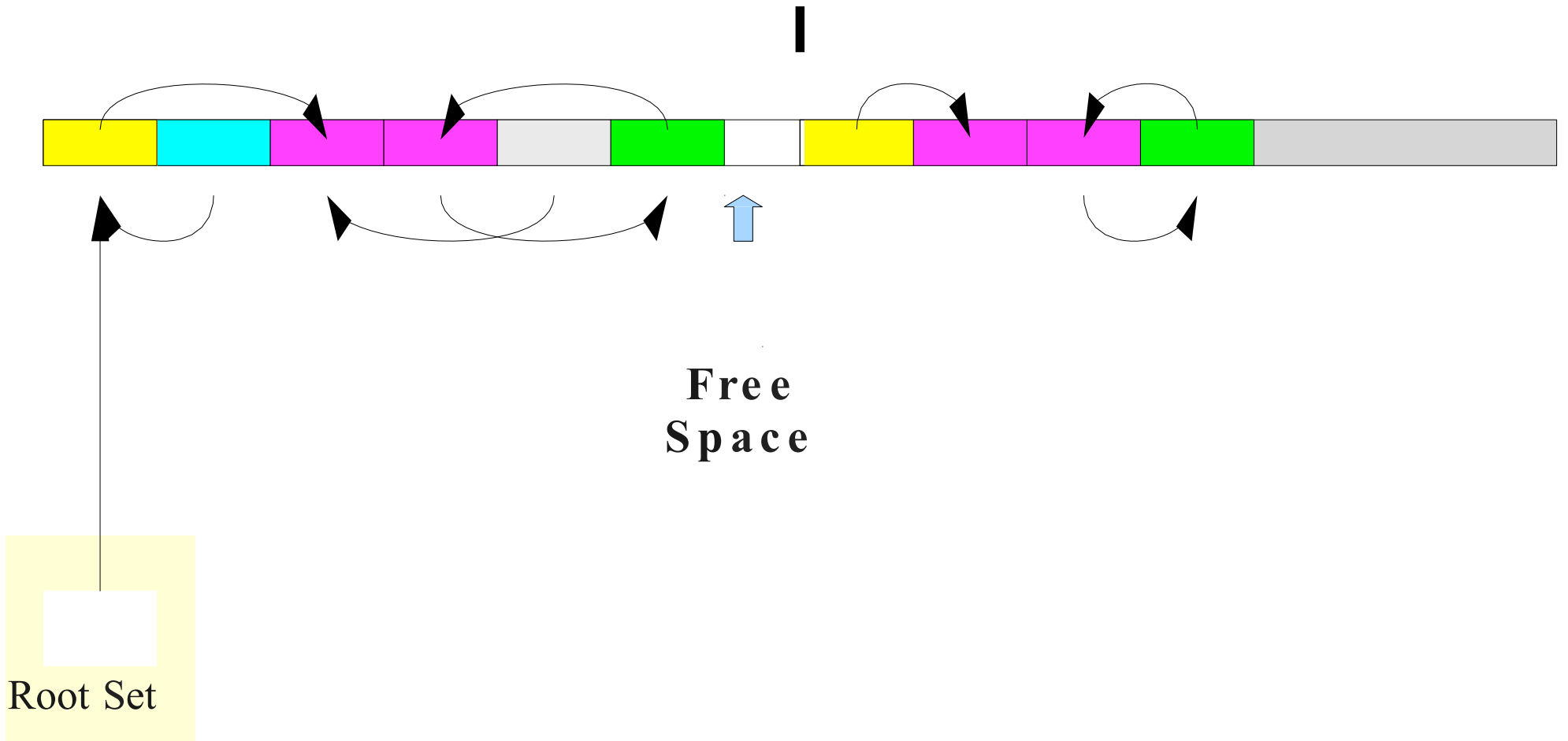


**Free
Space**

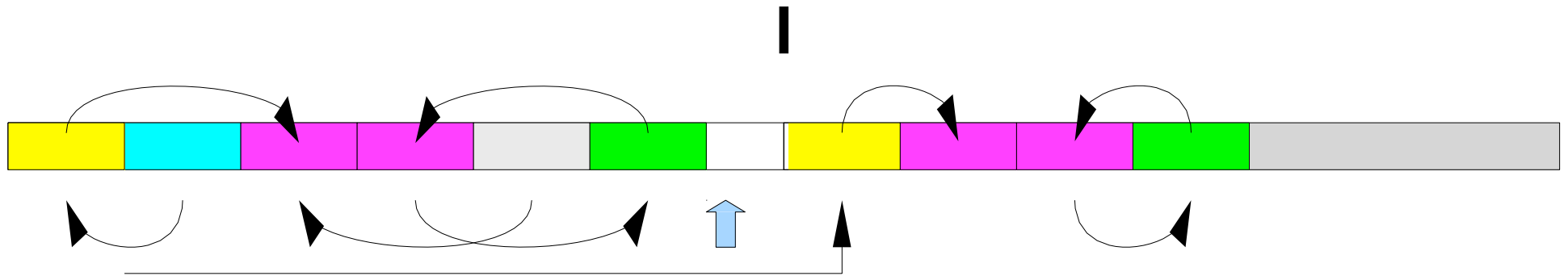


Root Set

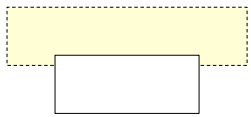
The Stop-and-Copy Collector



The Stop-and-Copy Collector

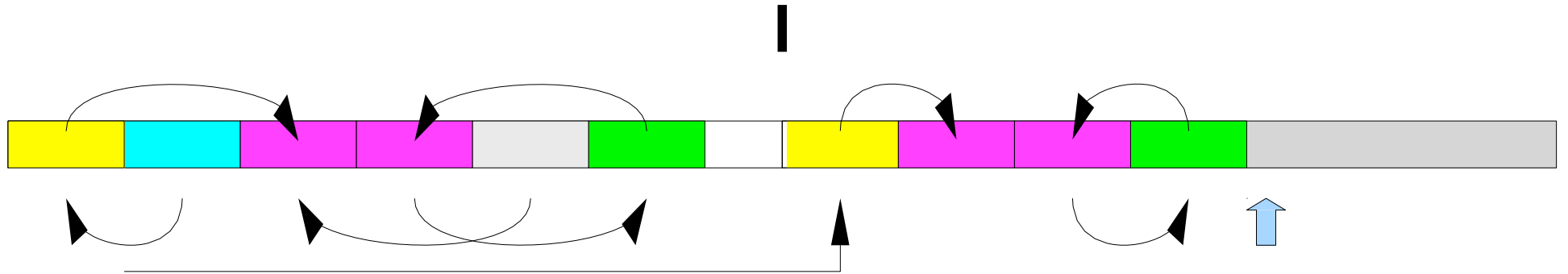


**Free
Space**

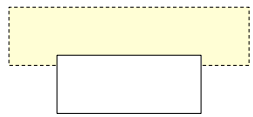


Root Set

The Stop-and-Copy Collector

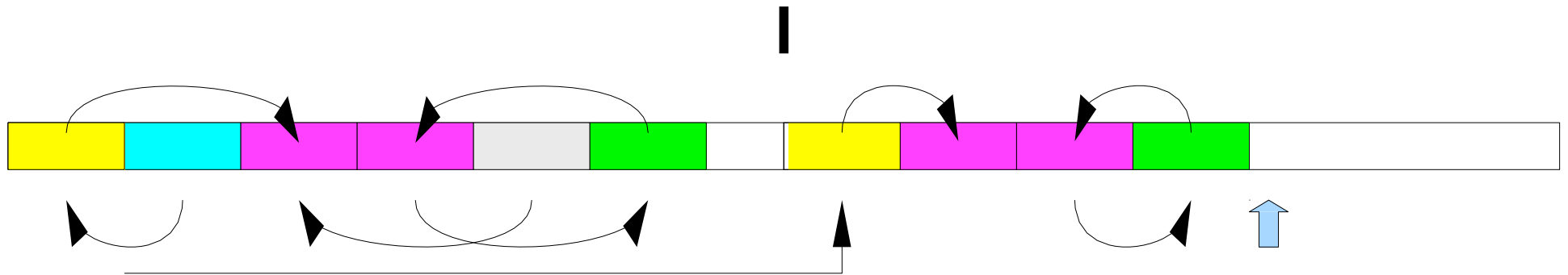


**Free
Space**

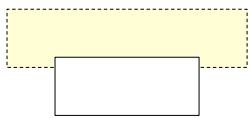


Root Set

The Stop-and-Copy Collector

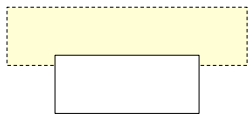
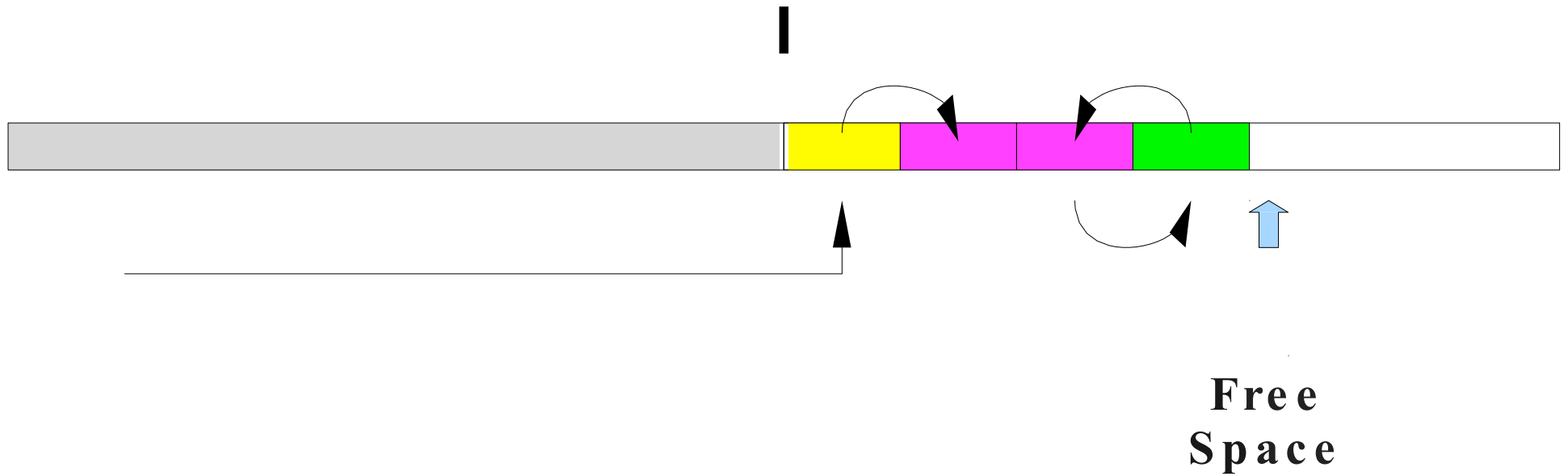


**Free
Space**



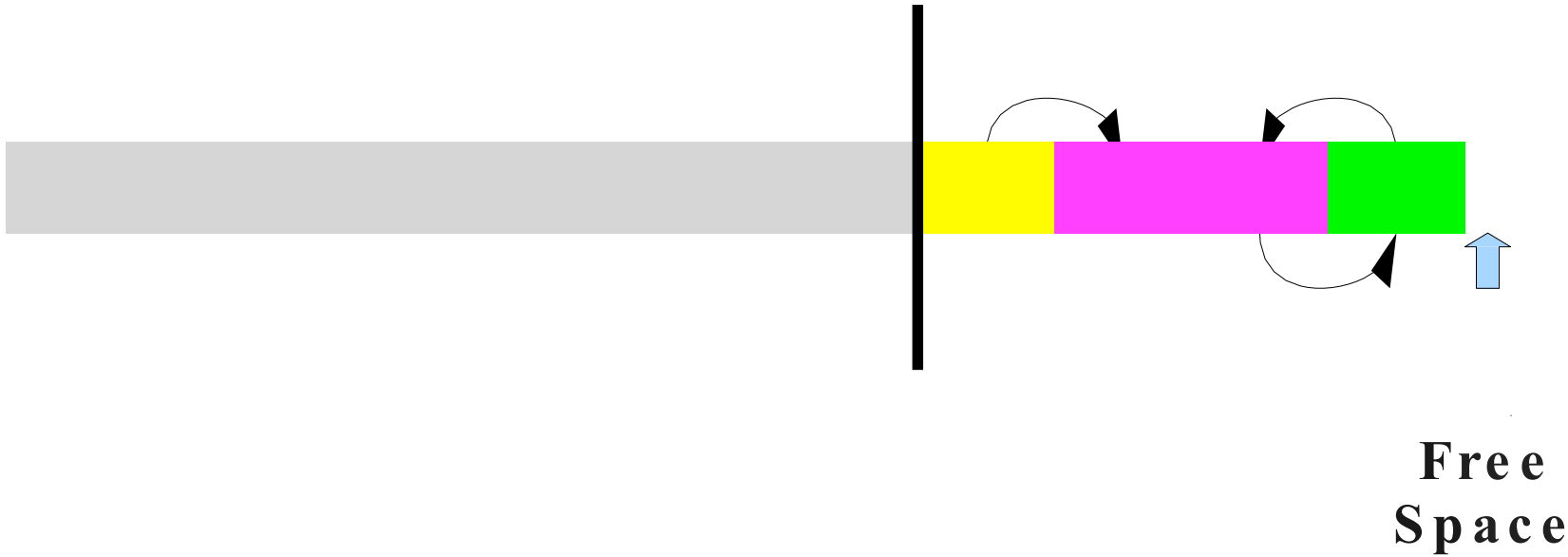
Root Set

The Stop-and-Copy Collector

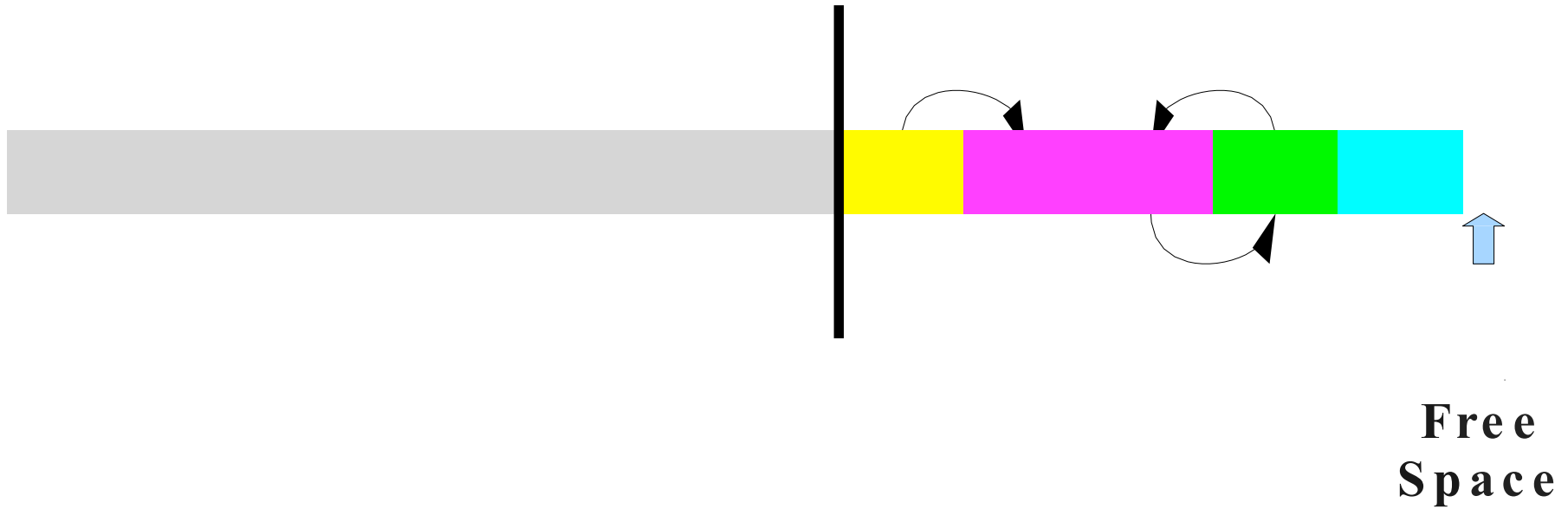


Root Set

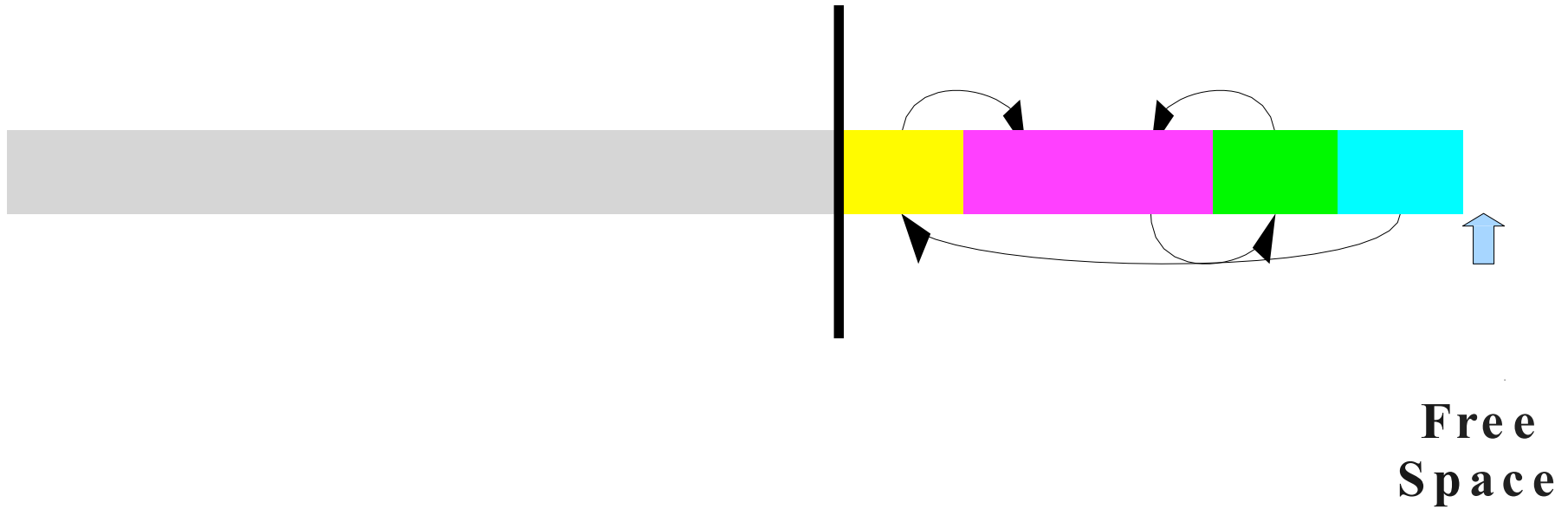
The Stop-and-Copy Collector



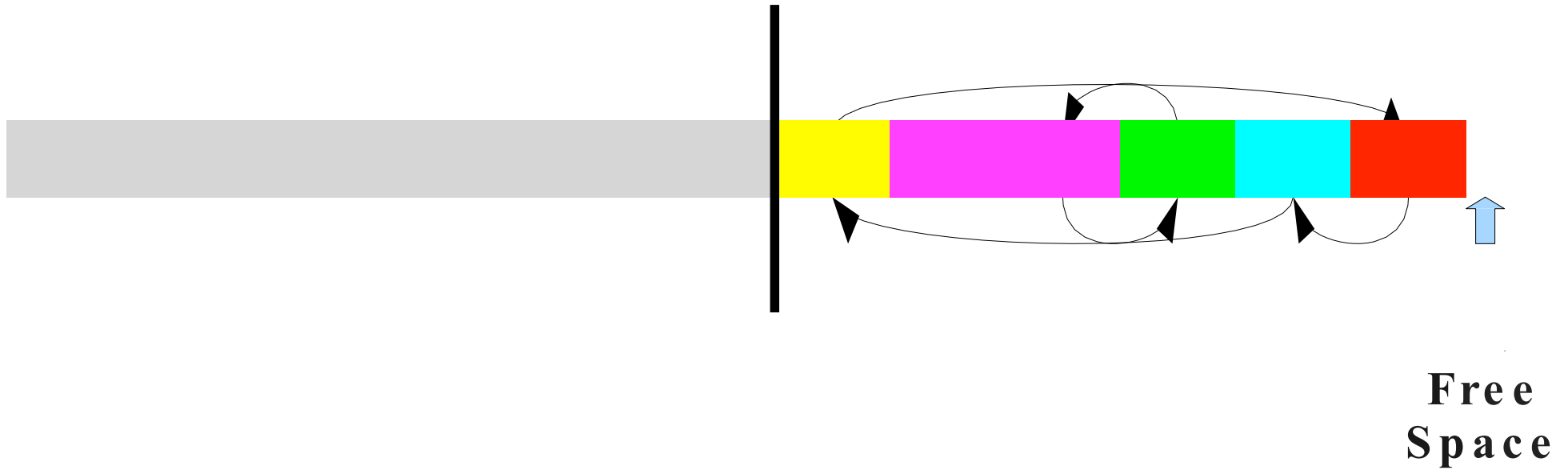
The Stop-and-Copy Collector



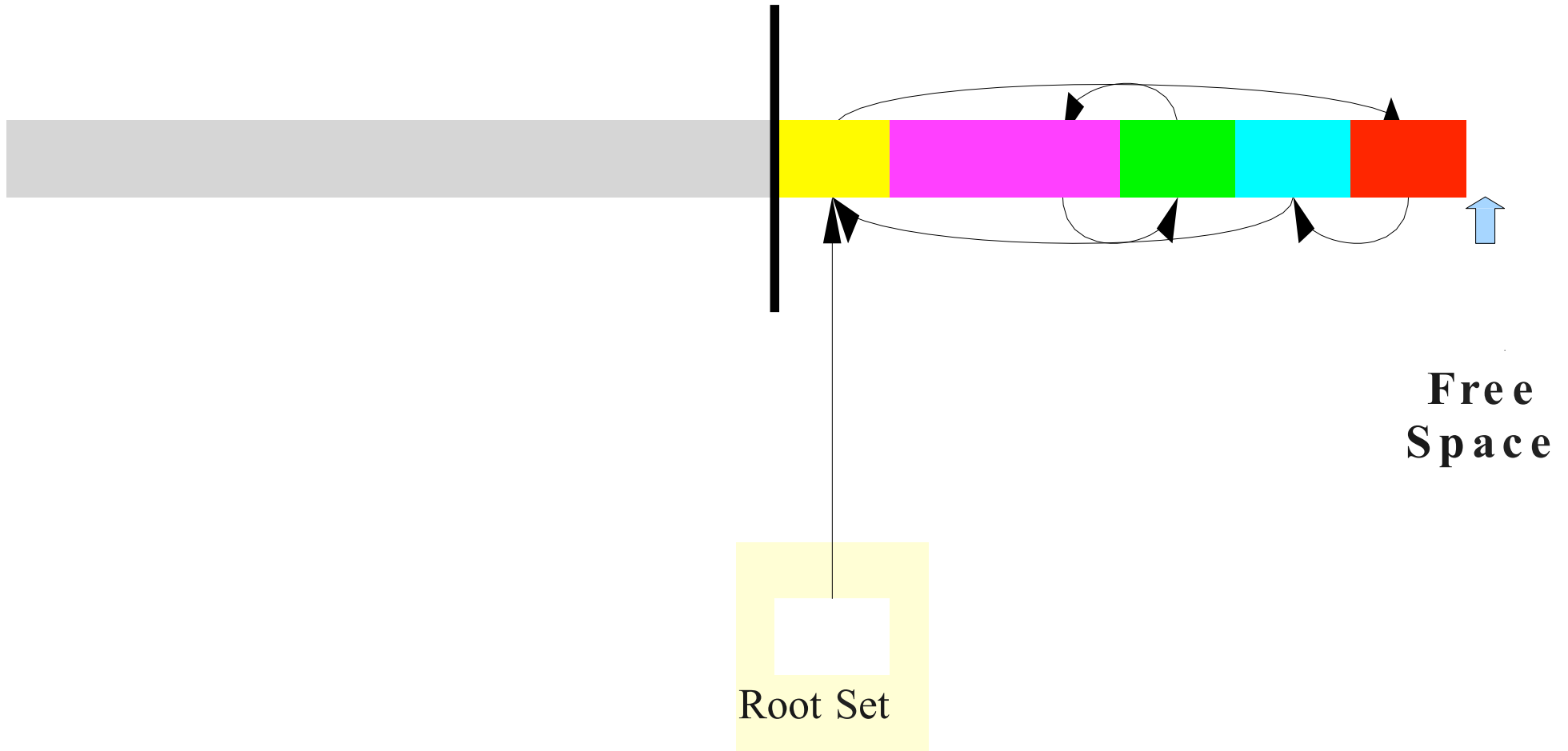
The Stop-and-Copy Collector



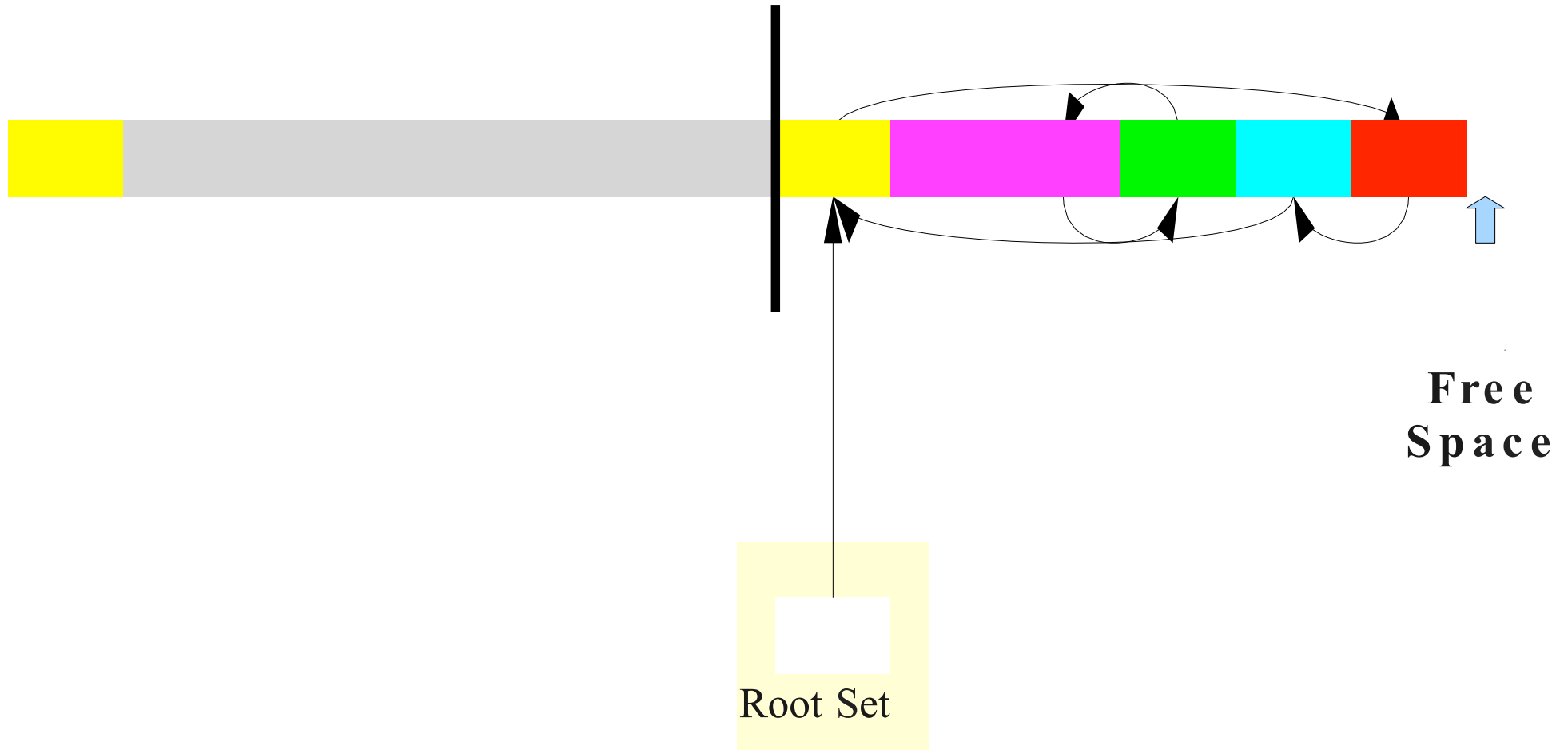
The Stop-and-Copy Collector



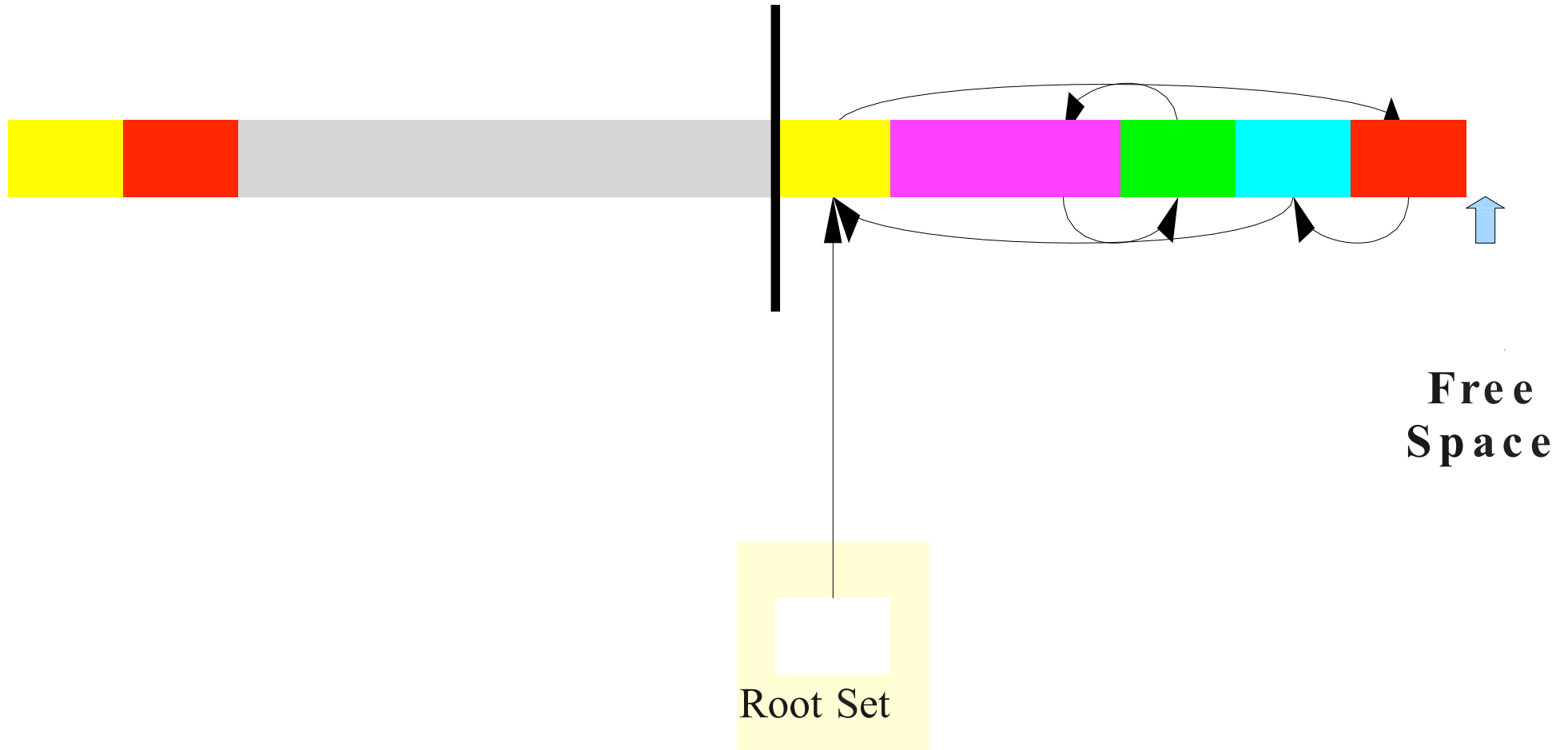
The Stop-and-Copy Collector



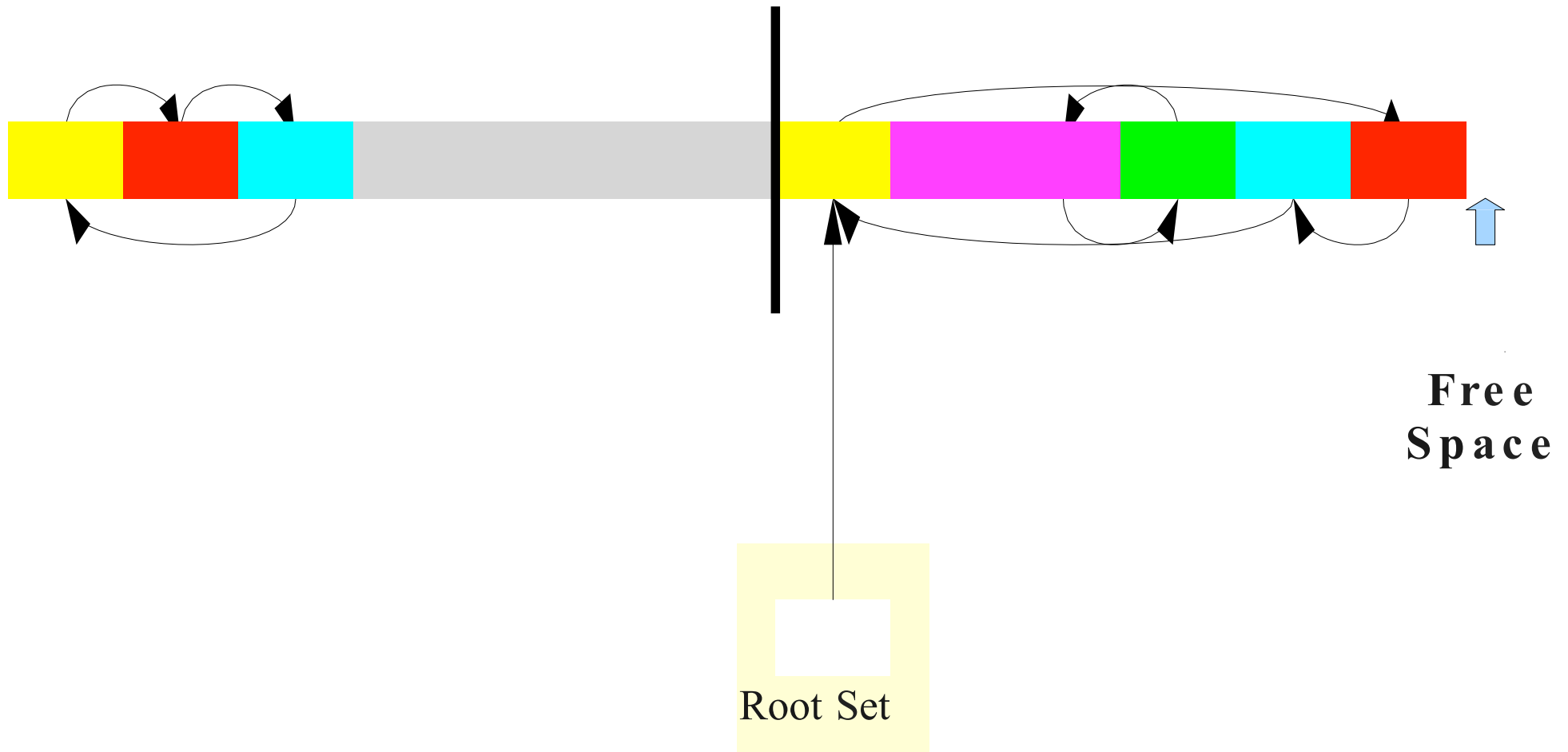
The Stop-and-Copy Collector



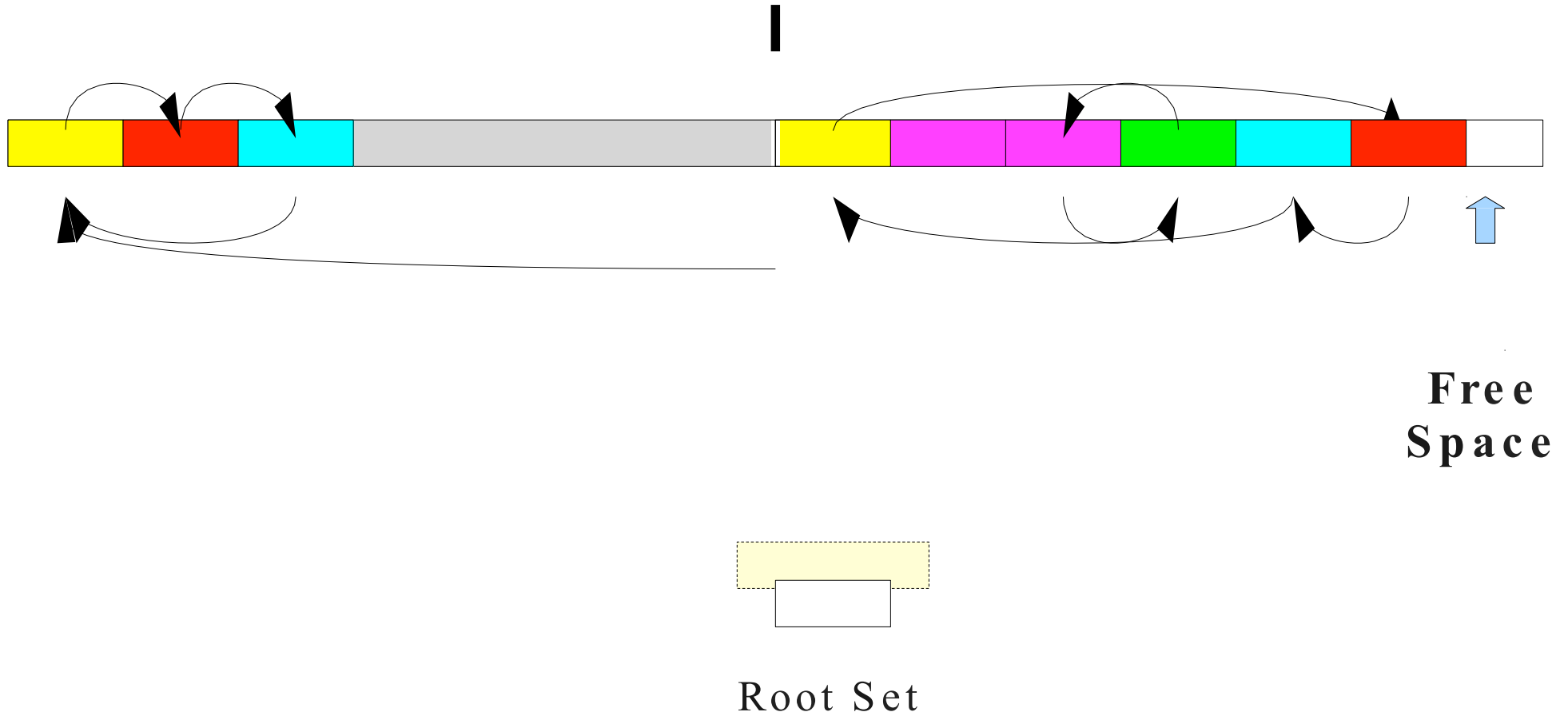
The Stop-and-Copy Collector



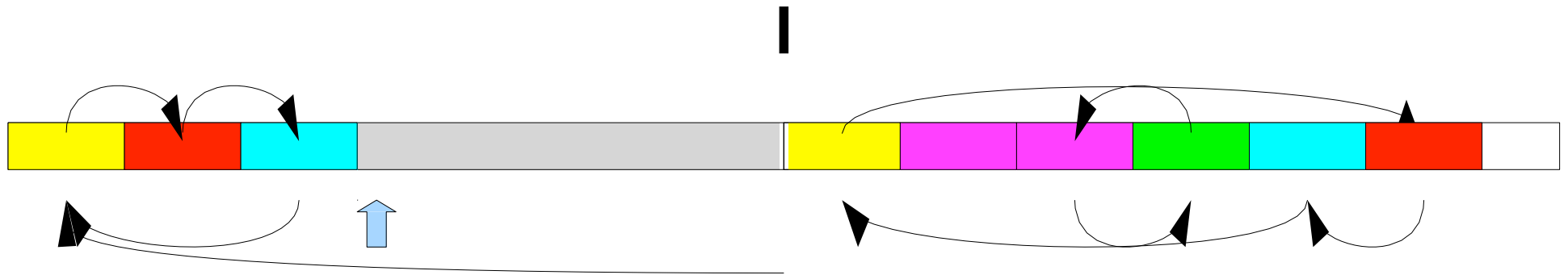
The Stop-and-Copy Collector



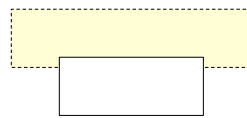
The Stop-and-Copy Collector



The Stop-and-Copy Collector

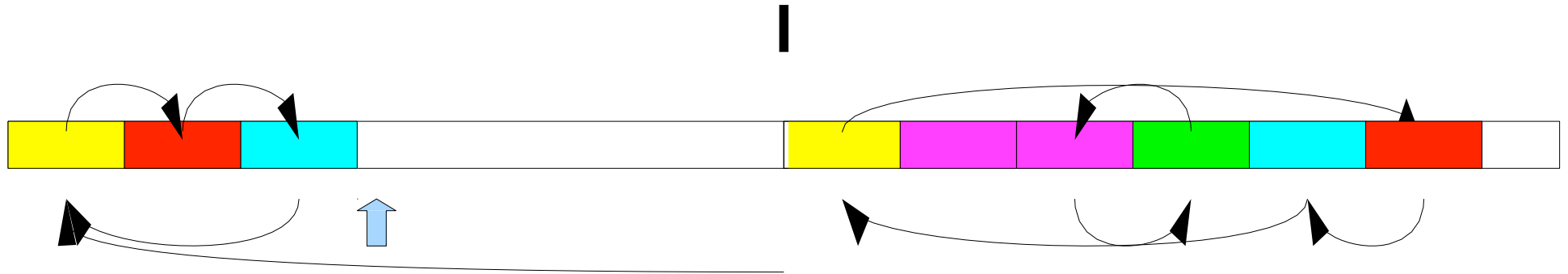


**Free
Space**

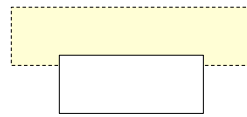


Root Set

The Stop-and-Copy Collector

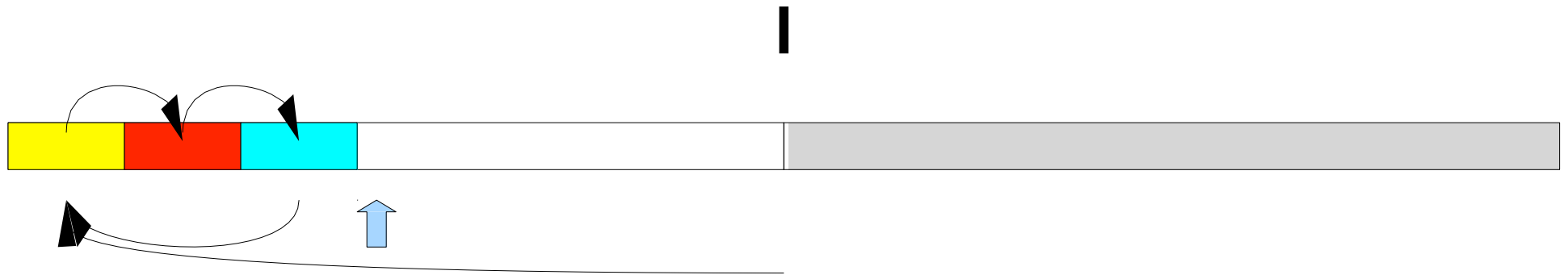


**Free
Space**

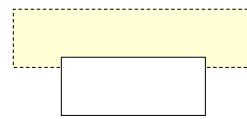


Root Set

The Stop-and-Copy Collector

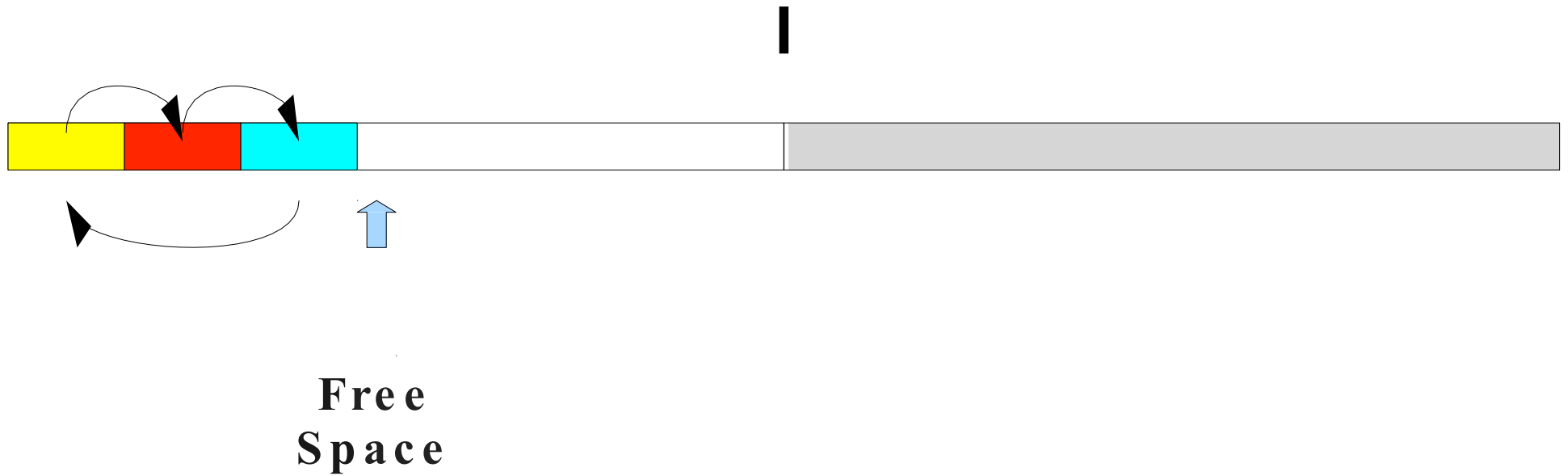


**Free
Space**



Root Set

The Stop-and-Copy Collector



Stop-and-Copy in Detail

Partition memory into two regions: **old space** and **new space**.

Keep track of the next free address in the new space.

To allocate **n** bytes of memory:

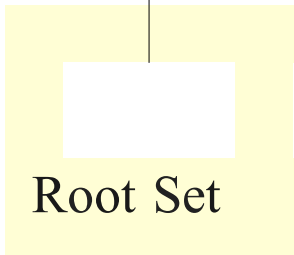
- If **n** bytes space exist at the free space pointer, use those bytes and advance the pointer.
- Otherwise, do a **copy** step.
- To execute a **copy** step:
 - For each object in the root set:
 - Copy that object to the start of the old space.
 - Recursively copy all objects reachable from that object.
 - 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

- Only tricky part about stop-and-copy is adjusting 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.

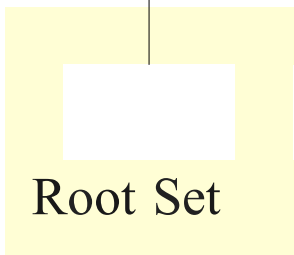
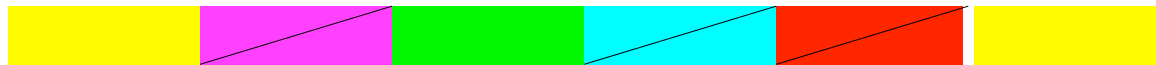
Forwarding Pointers

I

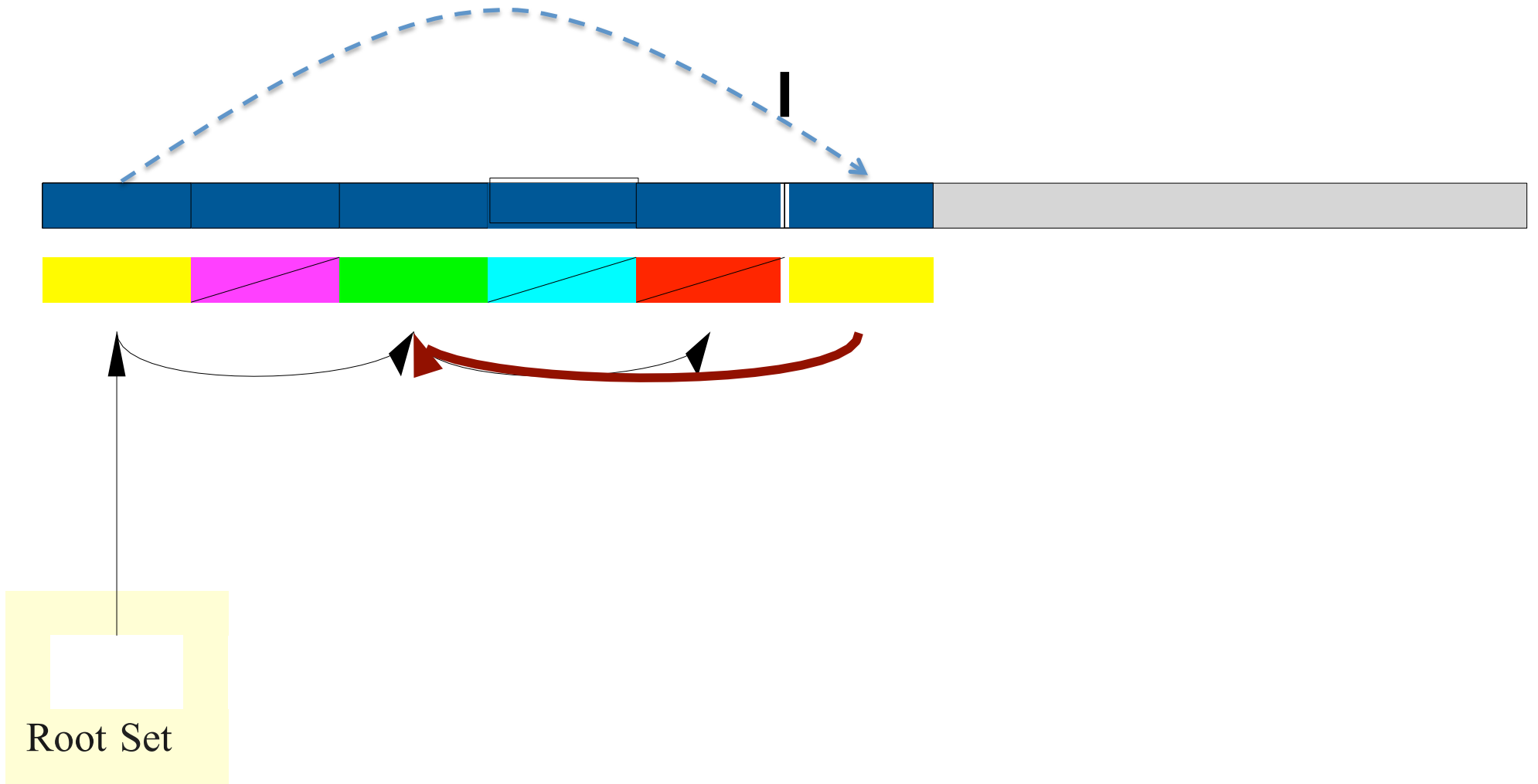


Forwarding Pointers

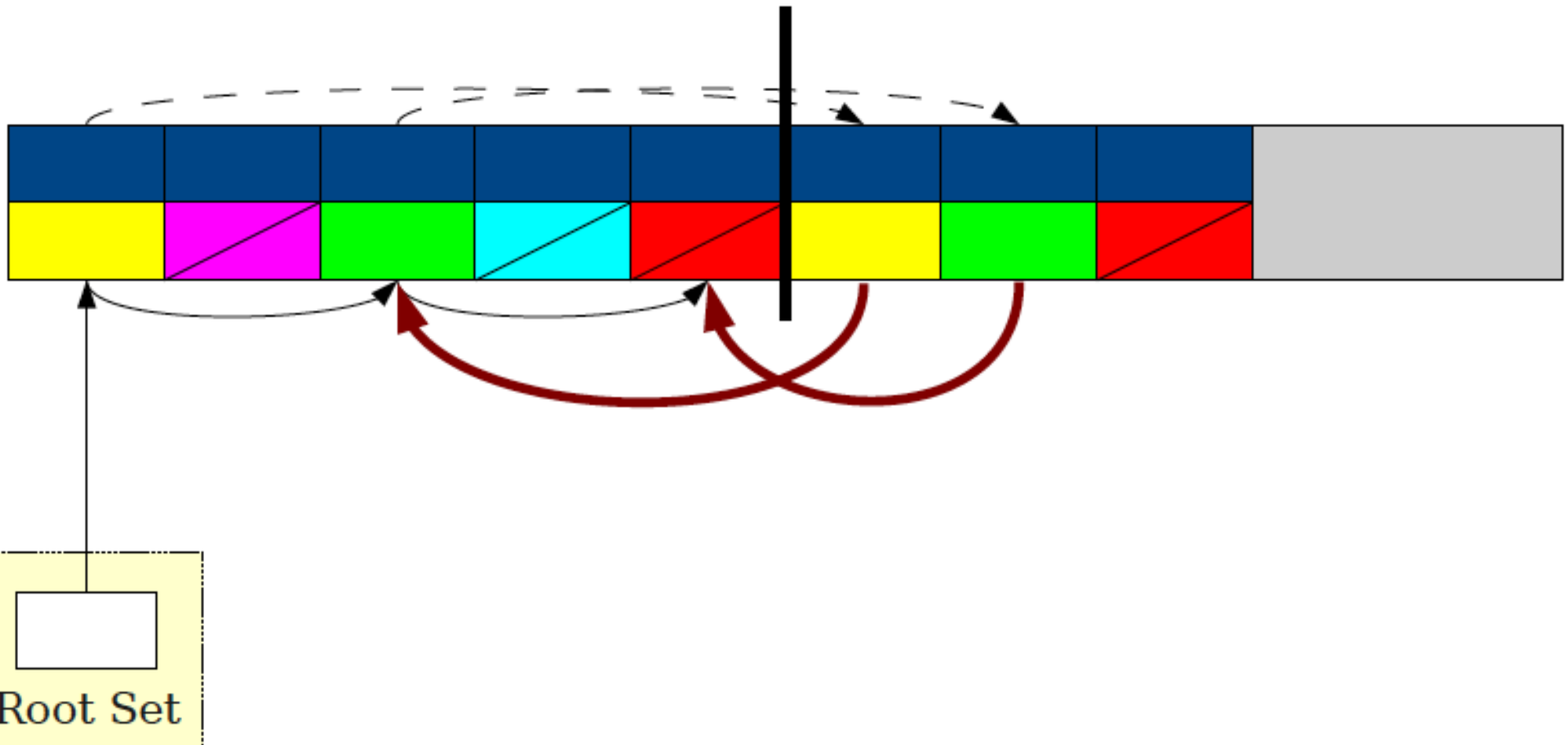
I



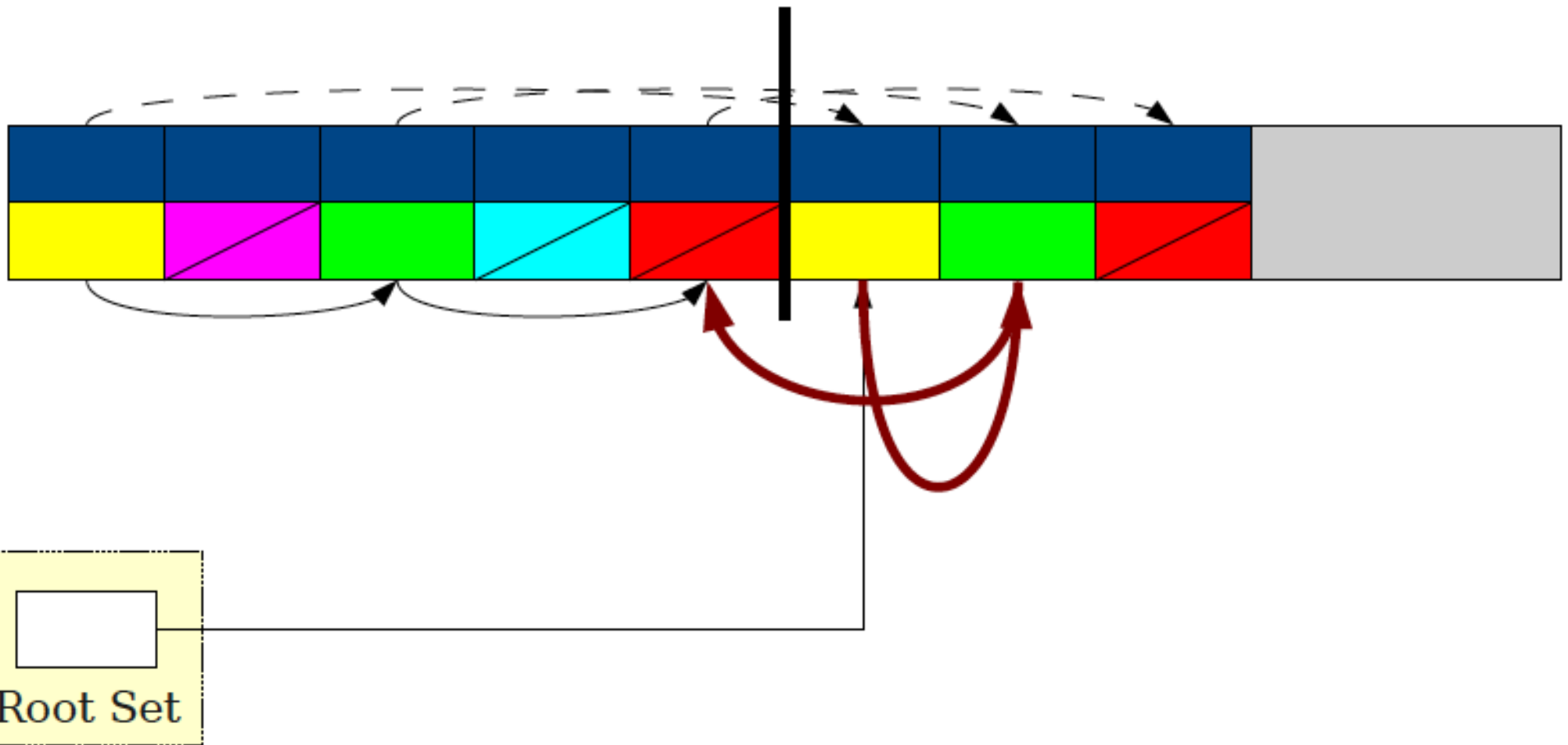
Forwarding Pointers



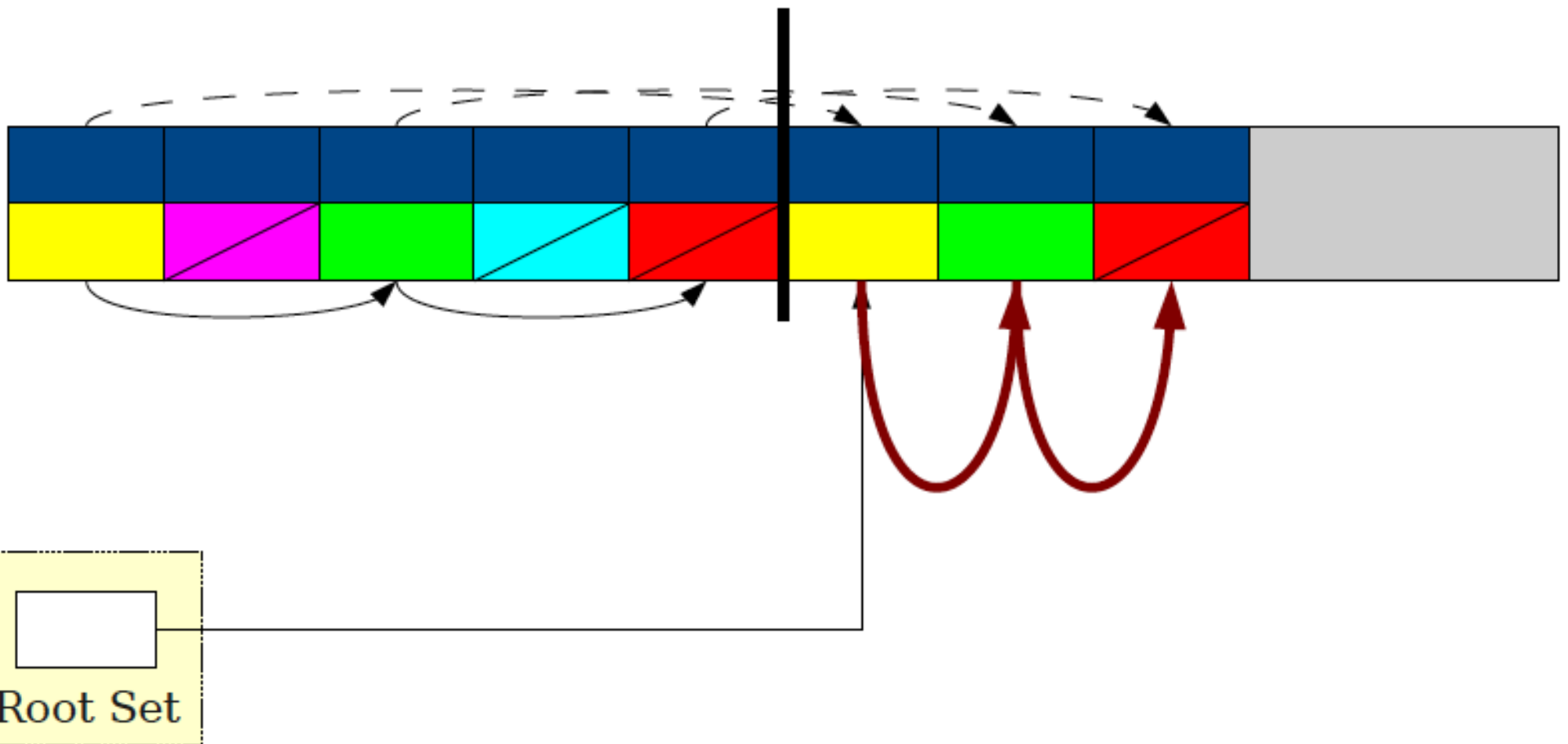
Forwarding Pointers



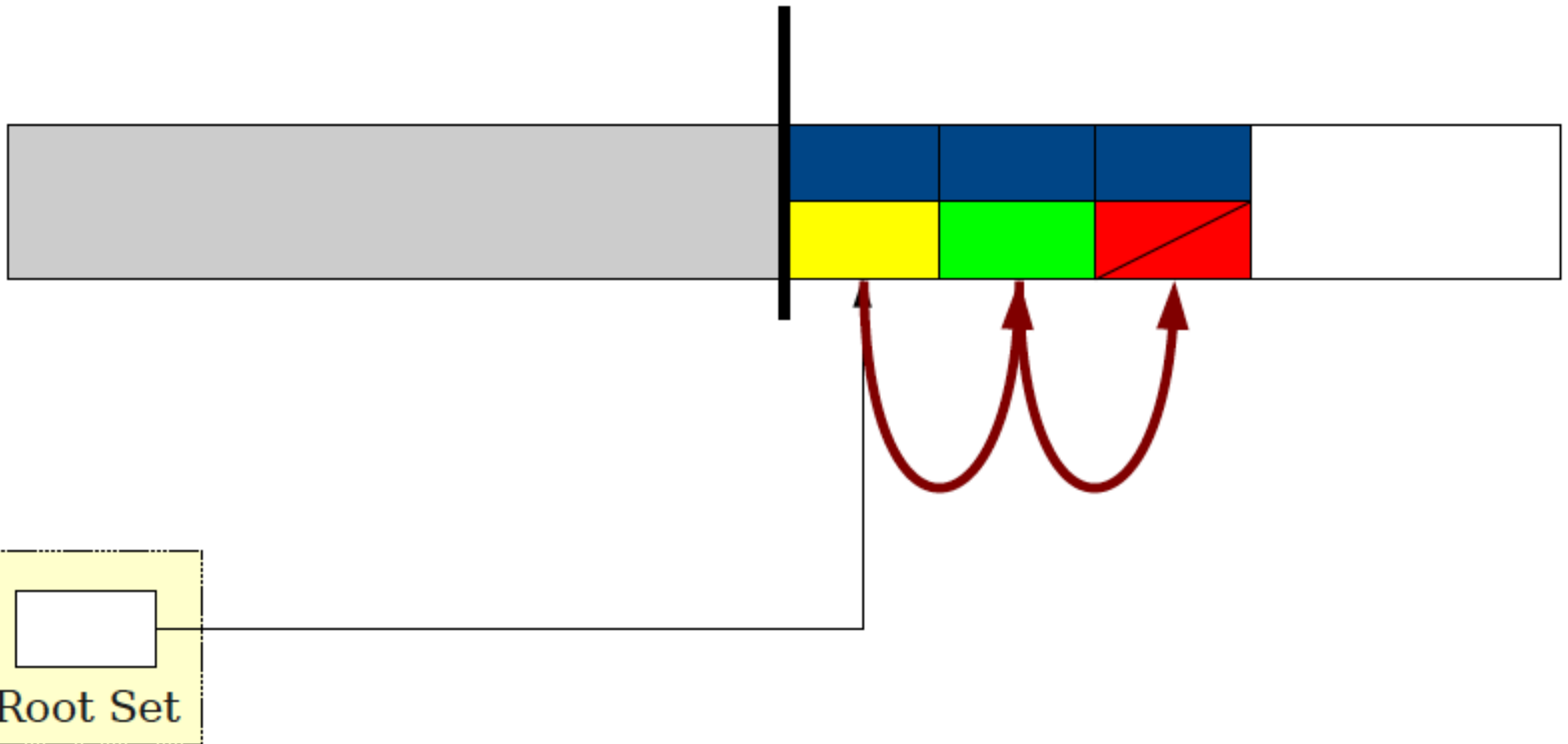
Forwarding Pointers



Forwarding Pointers



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 GC: *Objects Die Young*.
- Most objects have very short lifetimes.

Objects allocated locally in a function.

Temporary objects used to construct larger objects.

- Optimize GC to reclaim young objects rapidly while spending less time on older objects.

Generational GC

Partition memory into several “generations.”

Objects always allocated in the first generation.

When the first generation fills up, garbage collect it.

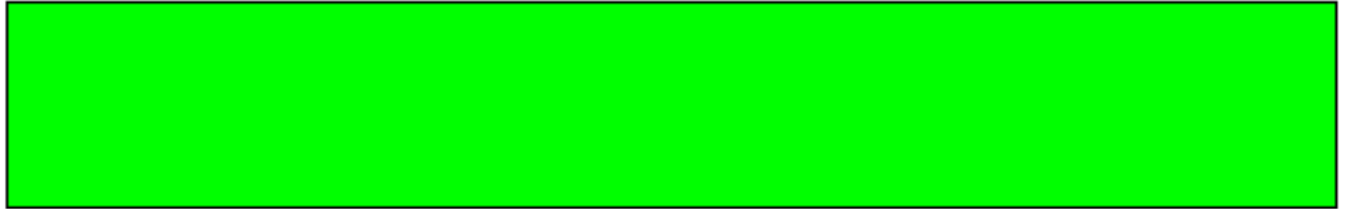
- Runs quickly; collects only a small region of memory.
- Move objects that survive in the first generation long enough into the next generation.
- When no space can be found, run a full (slower) garbage collection on all of memory.

Generational GC Technique

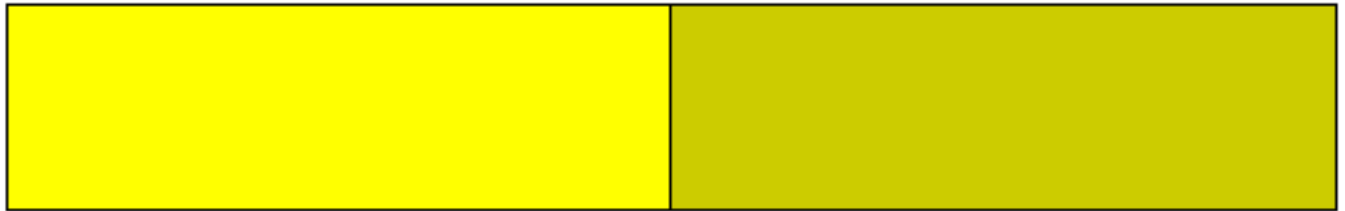
- New objects are allocated using a modified stop-and-copy collector in the **Eden** space.
- When Eden runs out of space, the stop-and-copy collector moves its elements to the **survivor space**.
- Objects that survive long enough in the survivor space become **tenured** and are moved to the **tenured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

Generational GC

Eden



Survivor Objects

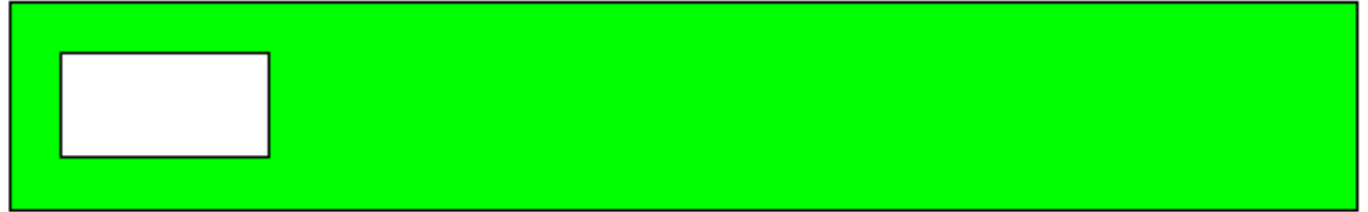


Tenured Objects



Allocating in Eden

Eden



Survivor Objects

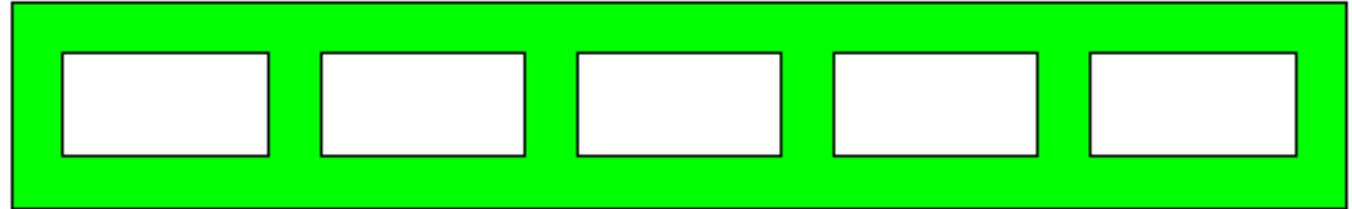


Tenured Objects



Allocating in Eden

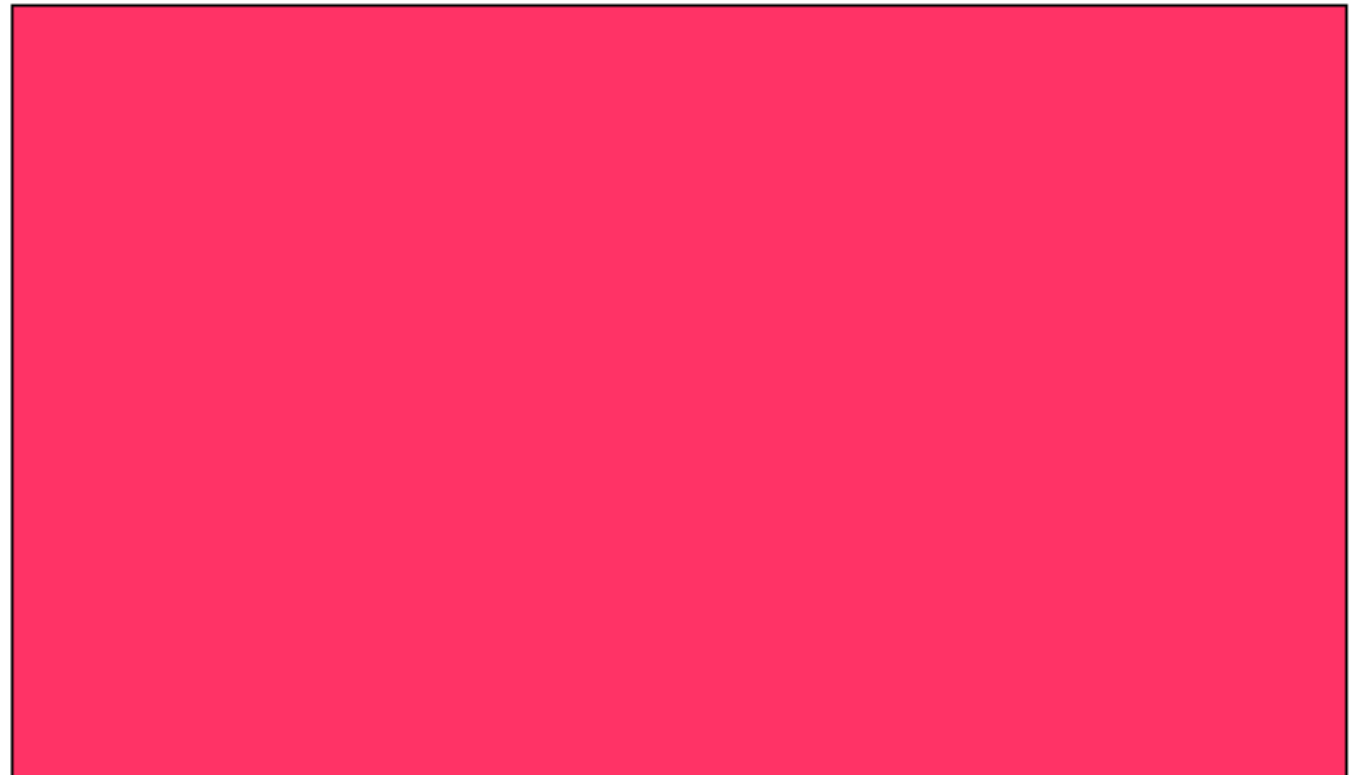
Eden



Survivor Objects

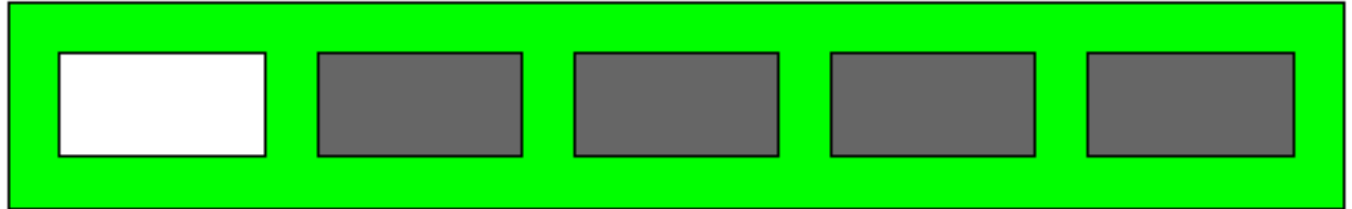


Tenured Objects



Some are short lived, others not

Eden



Survivor Objects

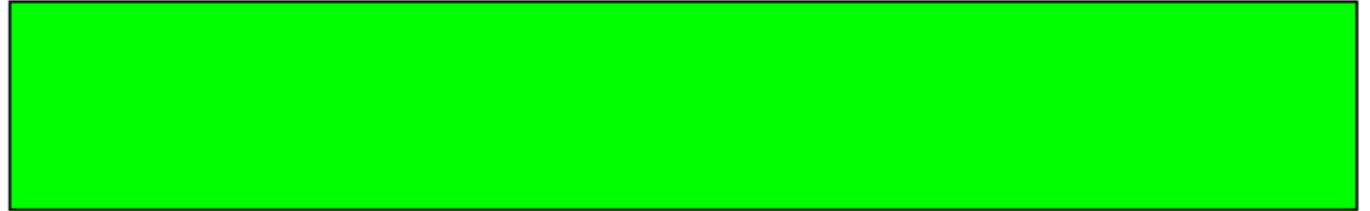


Tenured Objects



Stop & Copy moves survivors

Eden

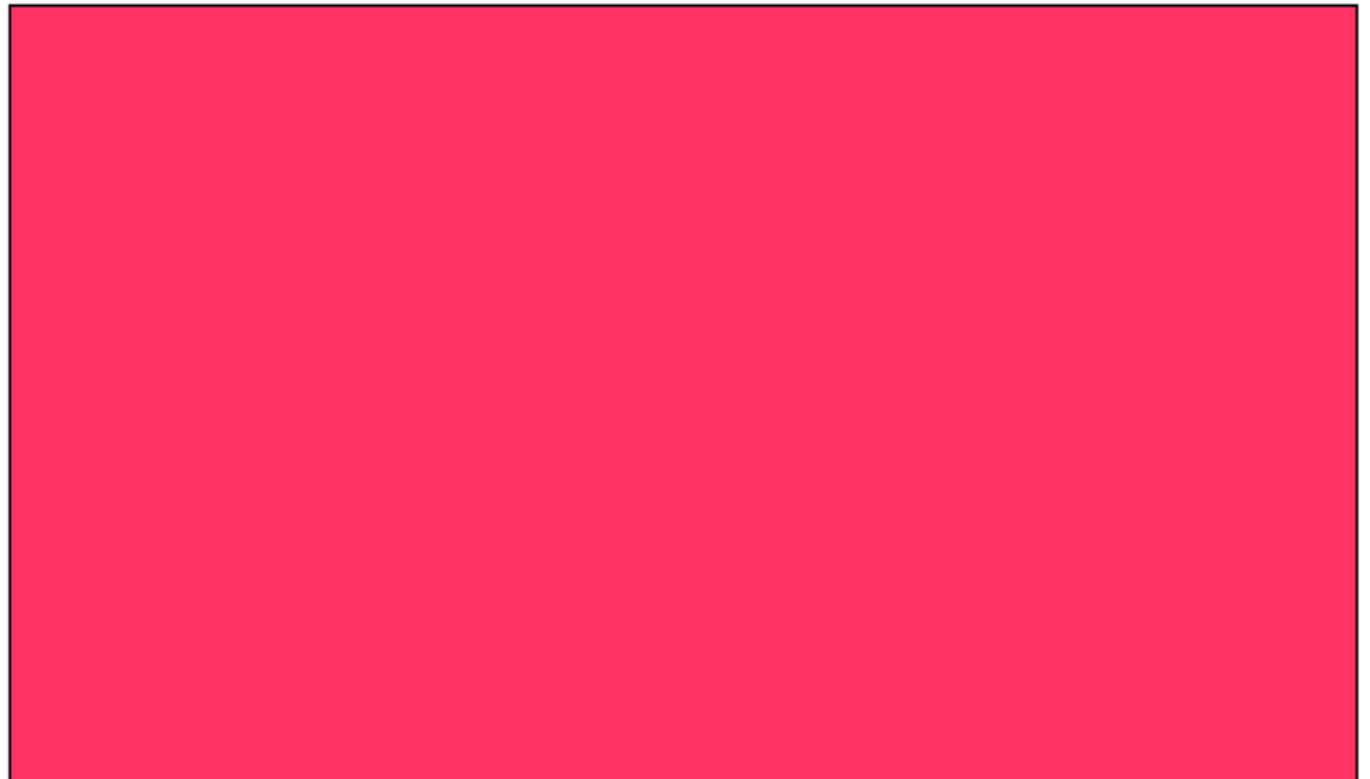


Survivor Objects

0

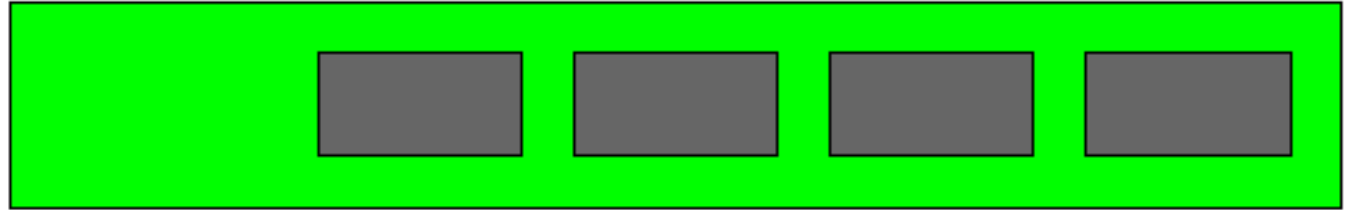


Tenured Objects

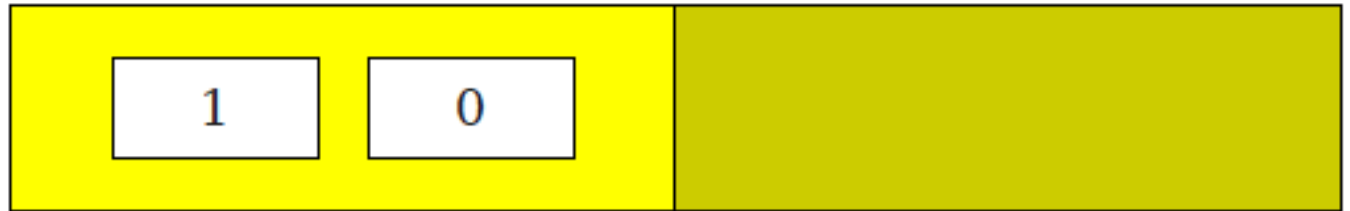


Some survive enough to be tenured

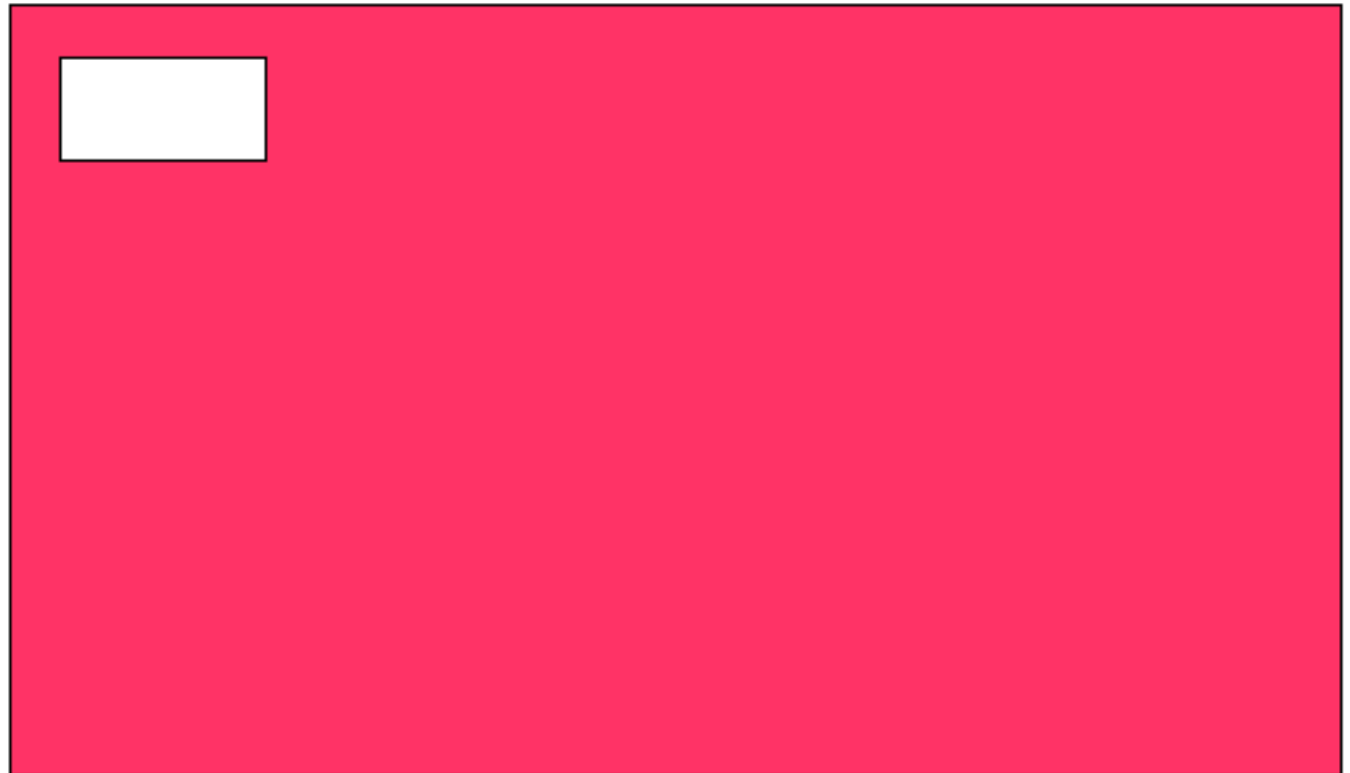
Eden



Survivor Objects



Tenured Objects



Checkpoint on GC

1. Why GC?
2. What are the main types of GC ?
3. For each main GC:
 1. How does it work?
 2. What is the main intuition behind it?
 3. Advantages?
 4. Disadvantages?

Reference Counting

Mark and Sweep

Stop and Copy

Generational