Lecture 24

Andreina Francisco
(based on slides by Tobias Wrigstad + …)

Garbage Collection

What is the Problem with this Code?

```
void priority_queue_destroy(priority_queue_t *priority_queue)
{
  link_t *cursor = priority_queue->first;
  while(cursor)
    {
      free(cursor->element);
      free(cursor);
      cursor = cursor->next;
    }
  free(priority_queue);
}
```

Use-After-Free

```
void priority_queue_destroy(priority_queue_t *priority_queue)
{
  link_t *cursor = priority_queue->first;
  while(cursor)
    {
      free(cursor->element);
      free(cursor);
      cursor = cursor->next;
    }
  free(priority_queue);
}
```

Use-After-Free

- Attempt to access memory after it has been freed

  corrupts validly used memory

  can cause a program to crash

  can potentially result in the execution of arbitrary code

  (see Use After Free Vulnerabilities)

What is the Problem with this Code?

```
void priority_queue_destroy(priority_queue_t *priority_queue)
{
  link_t *cursor = priority_queue->first;
  while(cursor)
    {
      link_t *temp = cursor->next;
      free(cursor->element);
      free(cursor);
      cursor = temp;
    }
  free(cursor);
  free(priority_queue);
}
```

Double Free

```
void priority_queue_destroy(priority_queue_t *priority_queue)
{
  link_t *cursor = priority_queue->first;
  while(cursor)
    {
      link_t *temp = cursor->next;
      free(cursor->element);
      free(cursor);
      cursor = temp;
    }
  free(cursor);
  free(priority_queue);
}
```

Double Free

- Frees memory after it has been free'd

  more or less the same problems as use-after-free

    corrupts validly used memory

    tends to cause a program to crash

    can be difficult to detect

    can result in the execution of arbitrary code

Automatic Memory Management

- Java handles memory automatically

  `new ClassName(…)` automatically allocates enough memory to the heap

  When the last reference to an object is removed, the object is junk

  When the memory becomes full, junk is automatically cleared to leave space for new items

- This means:

  No `malloc/calloc` (`new` always allocate on the heap, at least for what you know!)

  No `free`

Automatic Garbage Collection

- Objective: to give the programmer the illusion that memory is infinite

  Also, to avoid use-after-free and double-free vulnerabilities!

- Method: allocate memory, and identify the junk data and release it **automatically**
- Definition of garbage: data that cannot be accessed by the program (no references)
- More formally, the object O is garbage if there is no path in the memory graph from any root (the variable on the stack, global variables, etc.) to O
- Two basic ways to do automatic garbage collection:

  Reference counting

  Tracing

Reference Counting (Project 1)

- Basic idea: each item saves information on how many things point to it
- When this counter reaches 0: free it!
- Each time a reference is created / deleted, update the reference counter:

```
void *p = malloc(2048); // refcount 1
void *x = p; // refcount 2
p = NULL; // refcount 1
x = NULL; // refcount 0, free(x)
```

Reference Counting (Project 1)

- Advantages:

  One the simplest forms of memory management to implement

  Memory is reclaimed as soon as it can no longer be referenced

  Incremental - without long pauses for collection cycles:

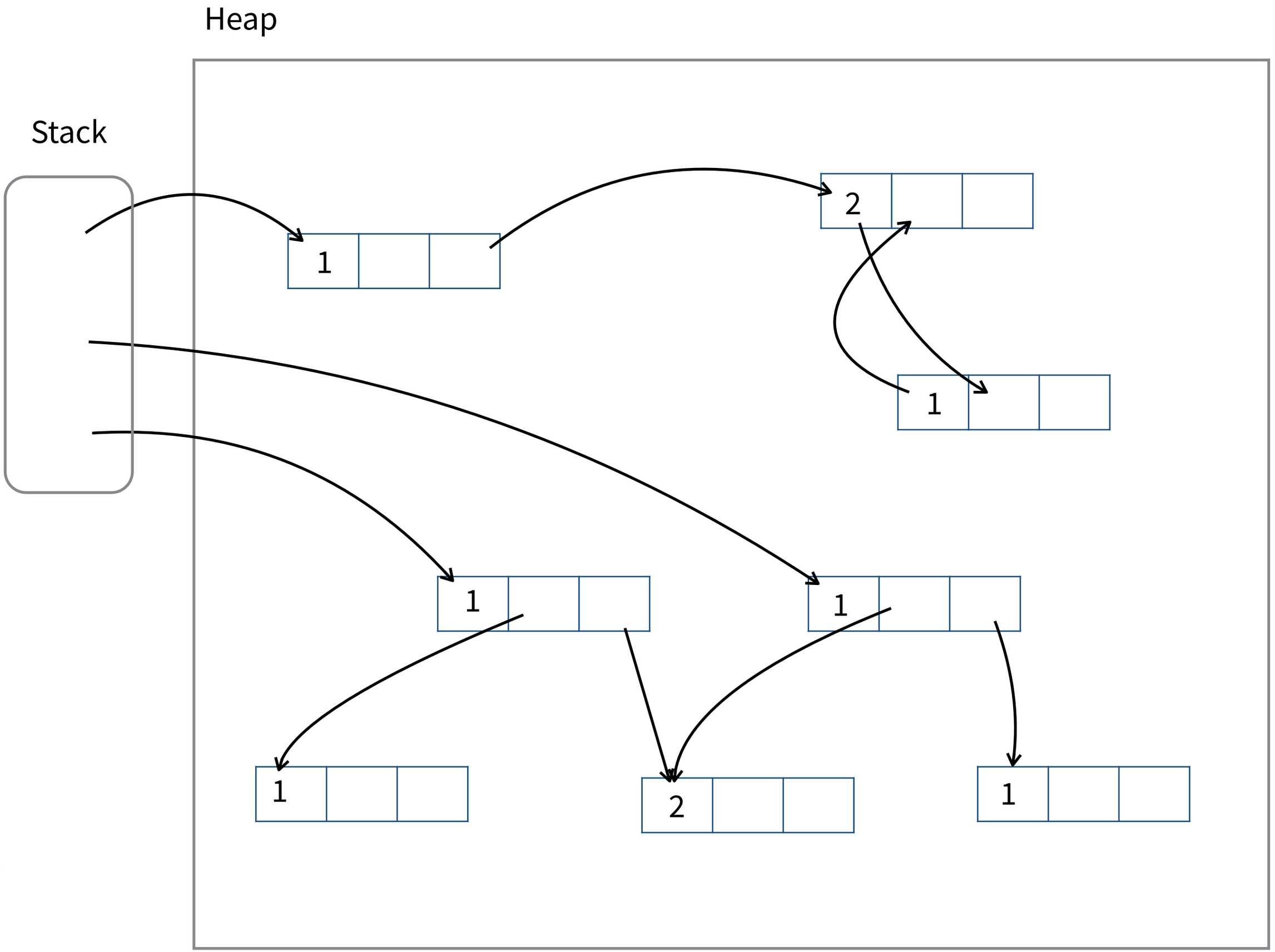
    Important in real-time applications or systems with limited memory!

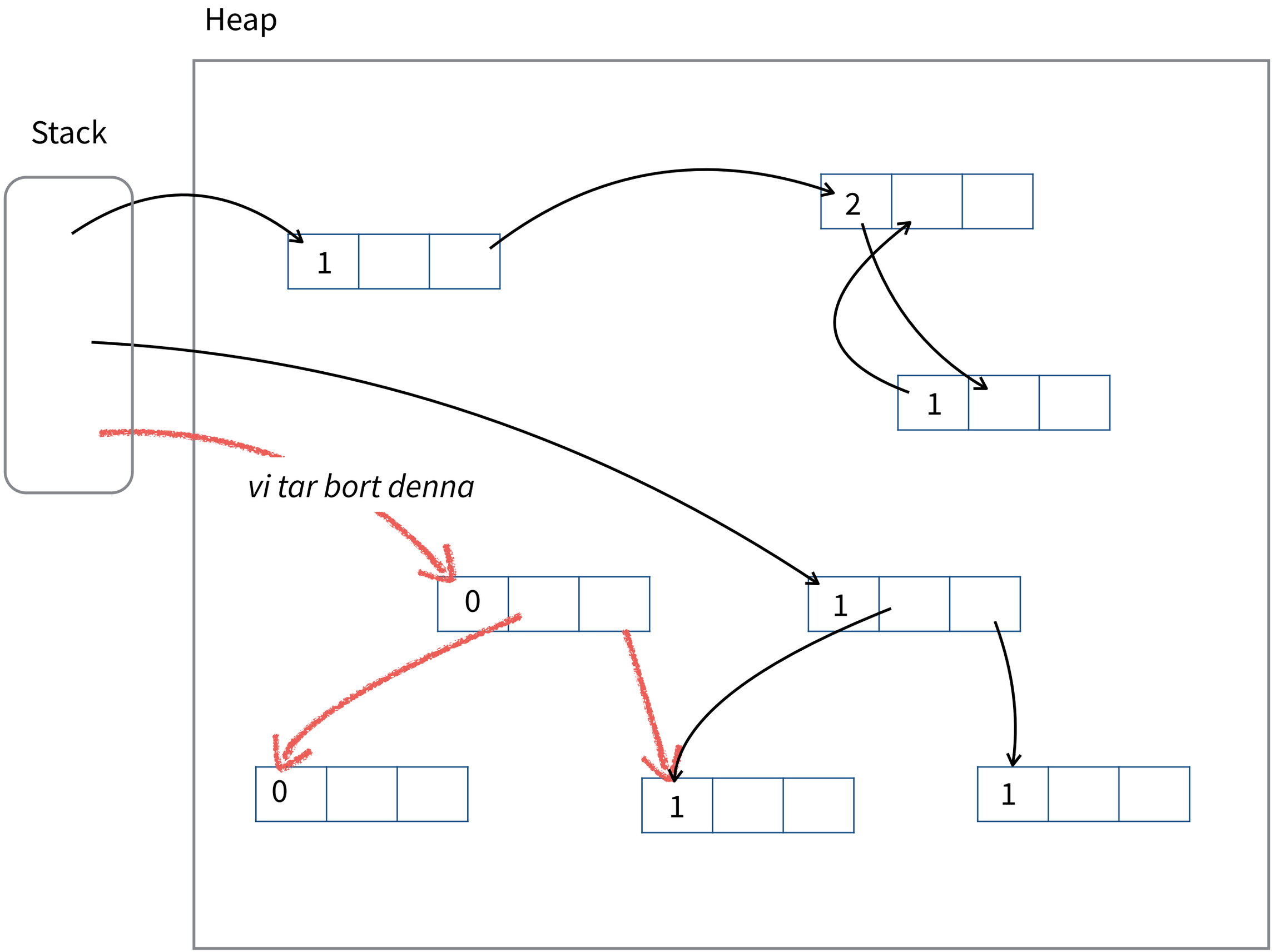
    What happens when a lot of memory needs to be free'd?

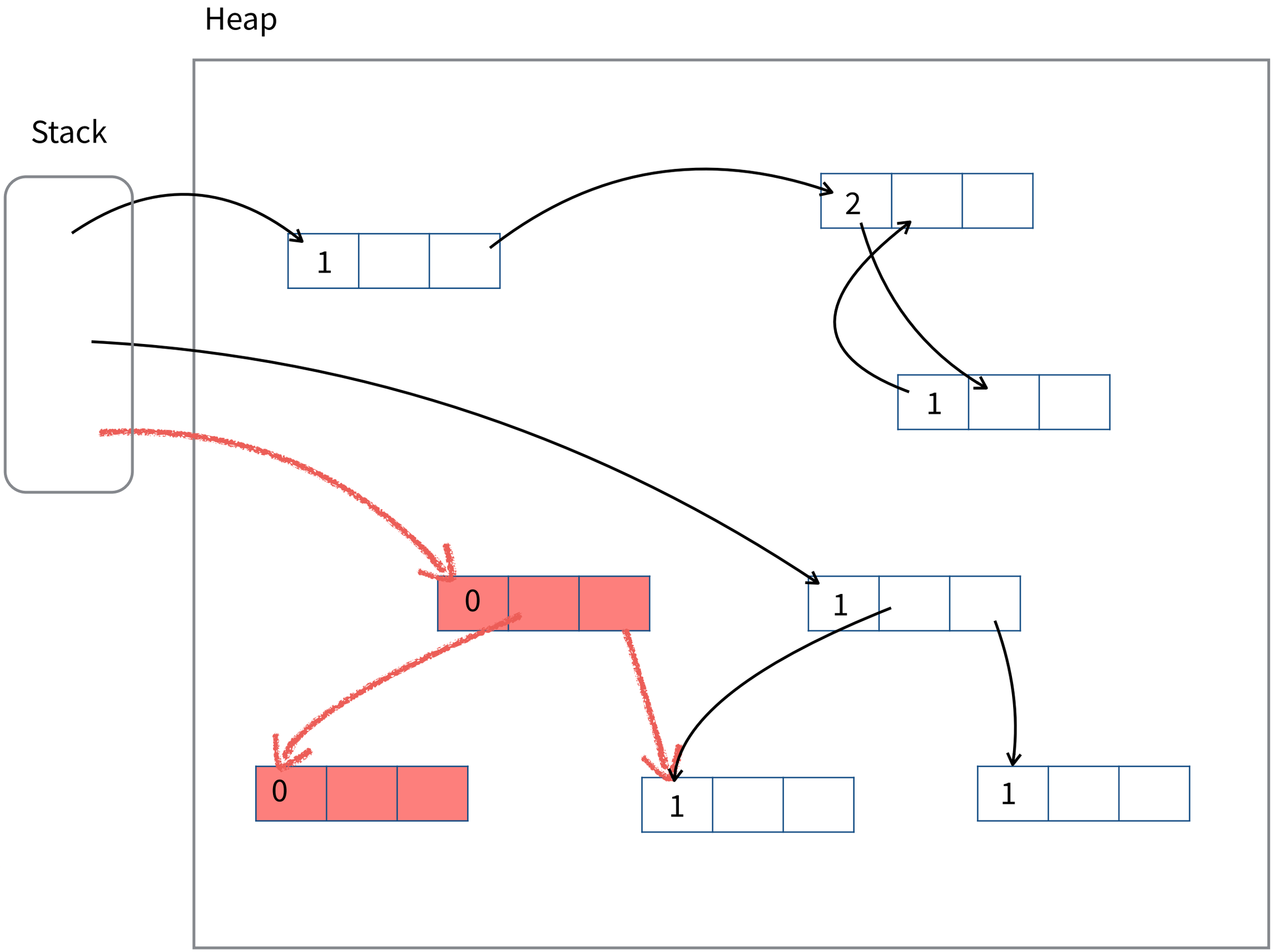
- Disadvantages:

  Counter updates are a source of inefficiency

  A naïve implementation can't deal with cyclic structures (see next pages)


Heap
Stack
1
2
1
1
1
1
2
1


Heap
Stack
1
2
1
vi tar bort denna
0
1
0
1
1

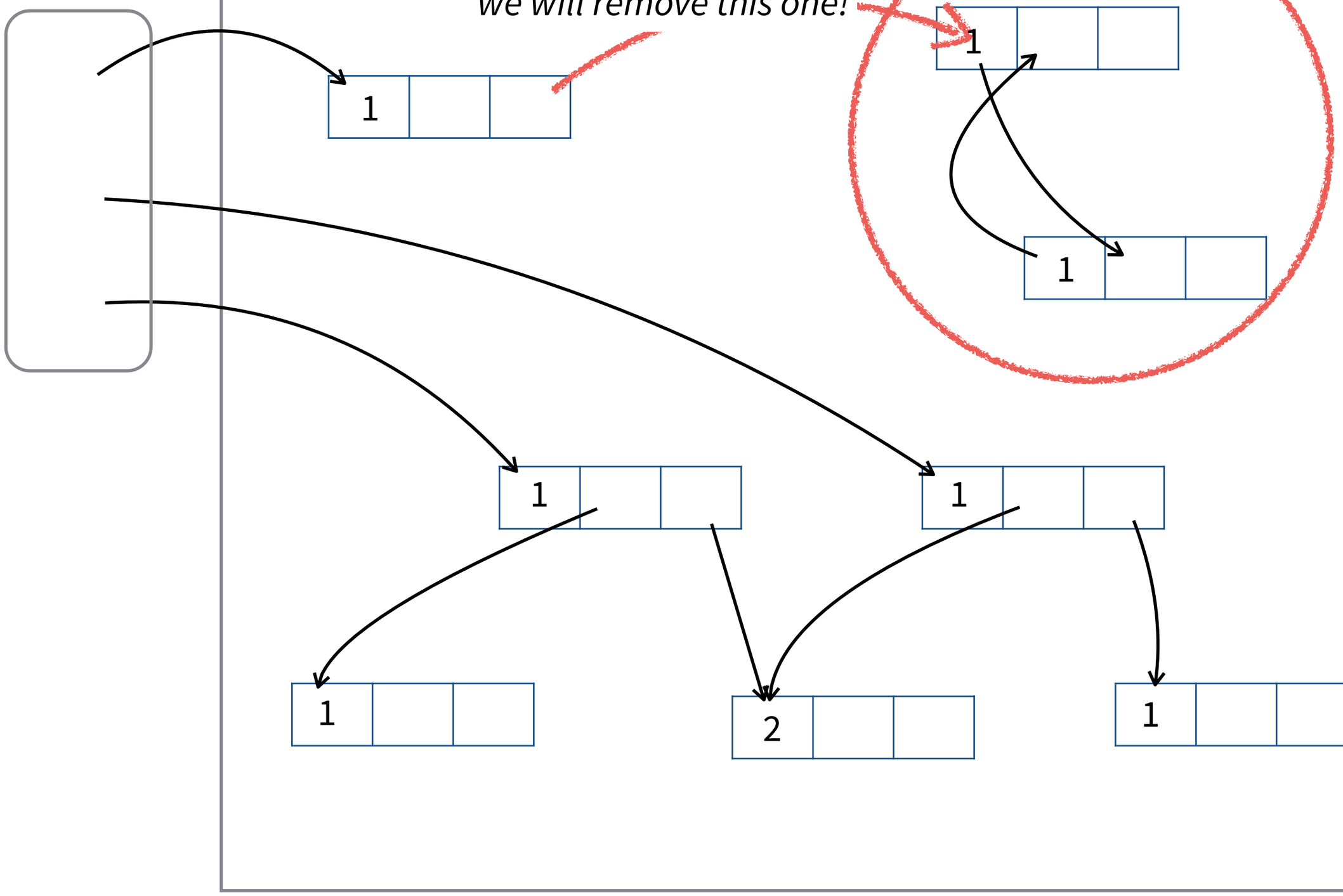

Heap
Stack
1
2
1
0
1
0
1
1

Heap

Leakage!

Stack

we will remove this one!



Reference Counting (Project 1)

- Things to consider:

  Memory management overhead (time)

  When should free actually happen?

  Counters (memory overhead vs overflow)

Tracing GC: Mark-Sweep

- Basic idea - when memory runs out, do the following:

    1. Mark all items as unreachable

    2. Follow the roots and select all items that can be reached

    3. Iterate over all items and free all of those that are not marked in 2.

Tracing GC: Mark-Sweep

- Advantages:

  Can handle cyclic references

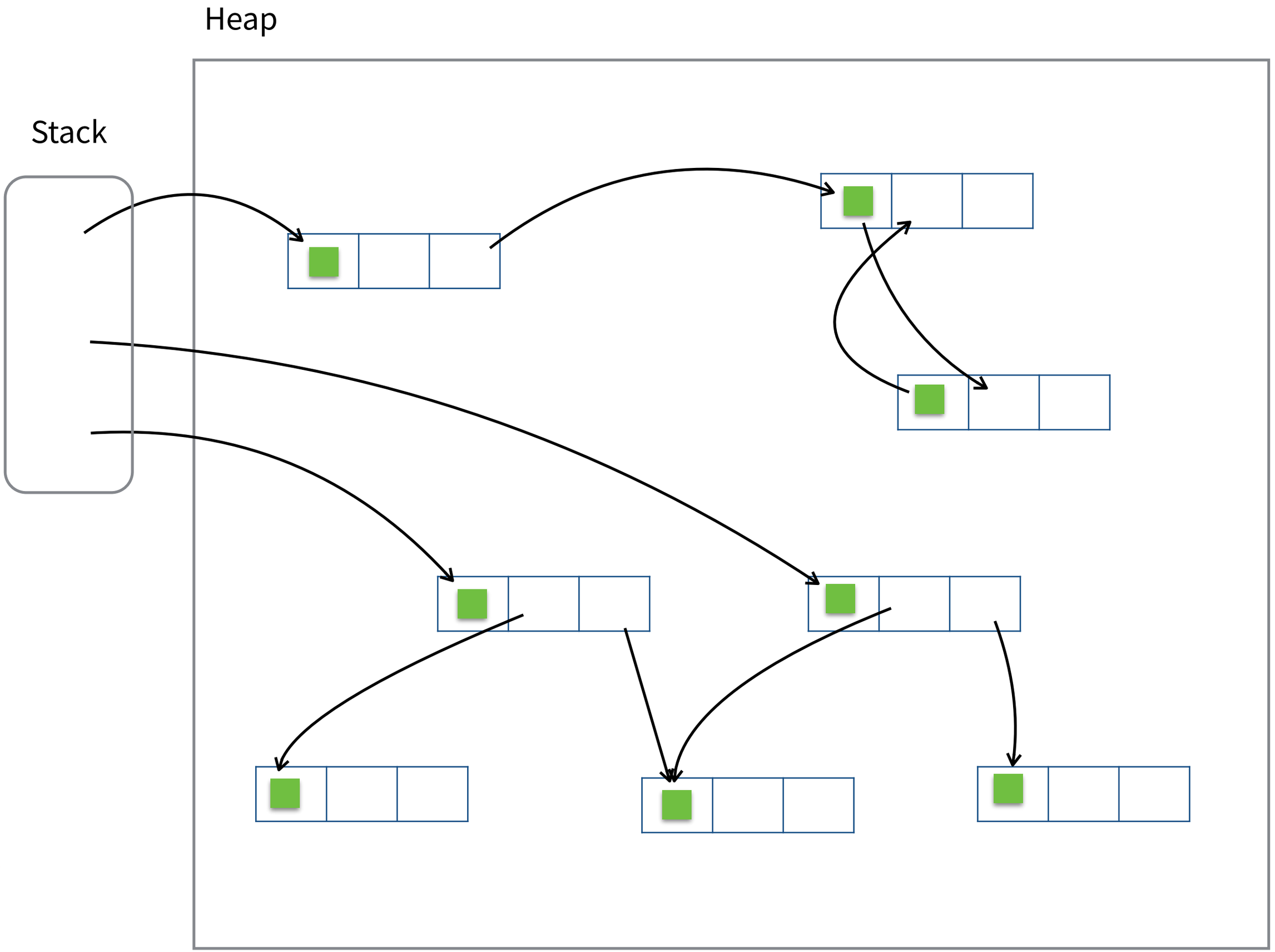
- Disadvantages:

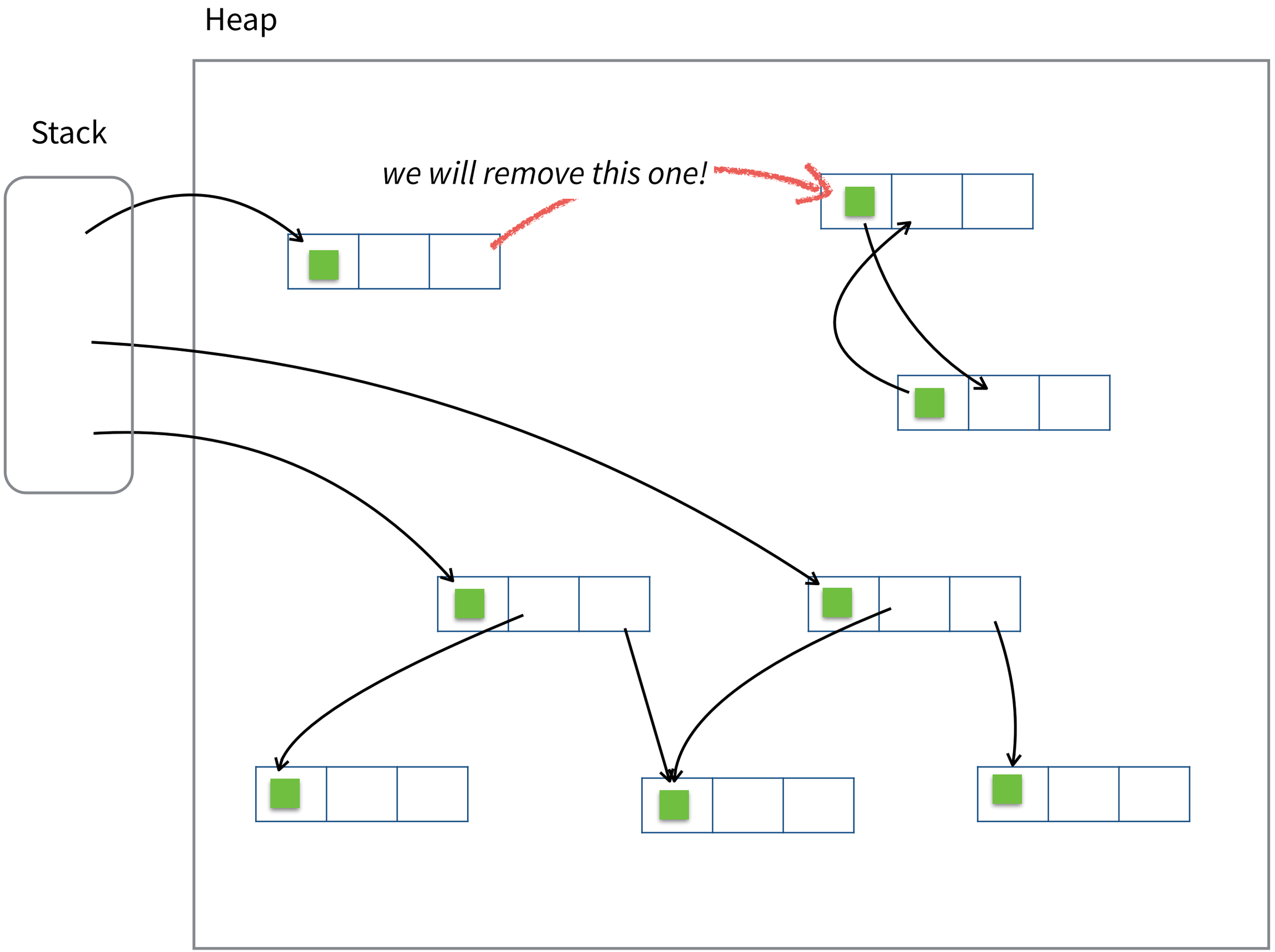
  Stop-the-world! Normal program execution is suspended while the garbage collection algorithm runs

  Memory overhead

  Fragmentation

  When will it run? Will it ever run?


Heap
Stack


Heap
Stack
we will remove this one!

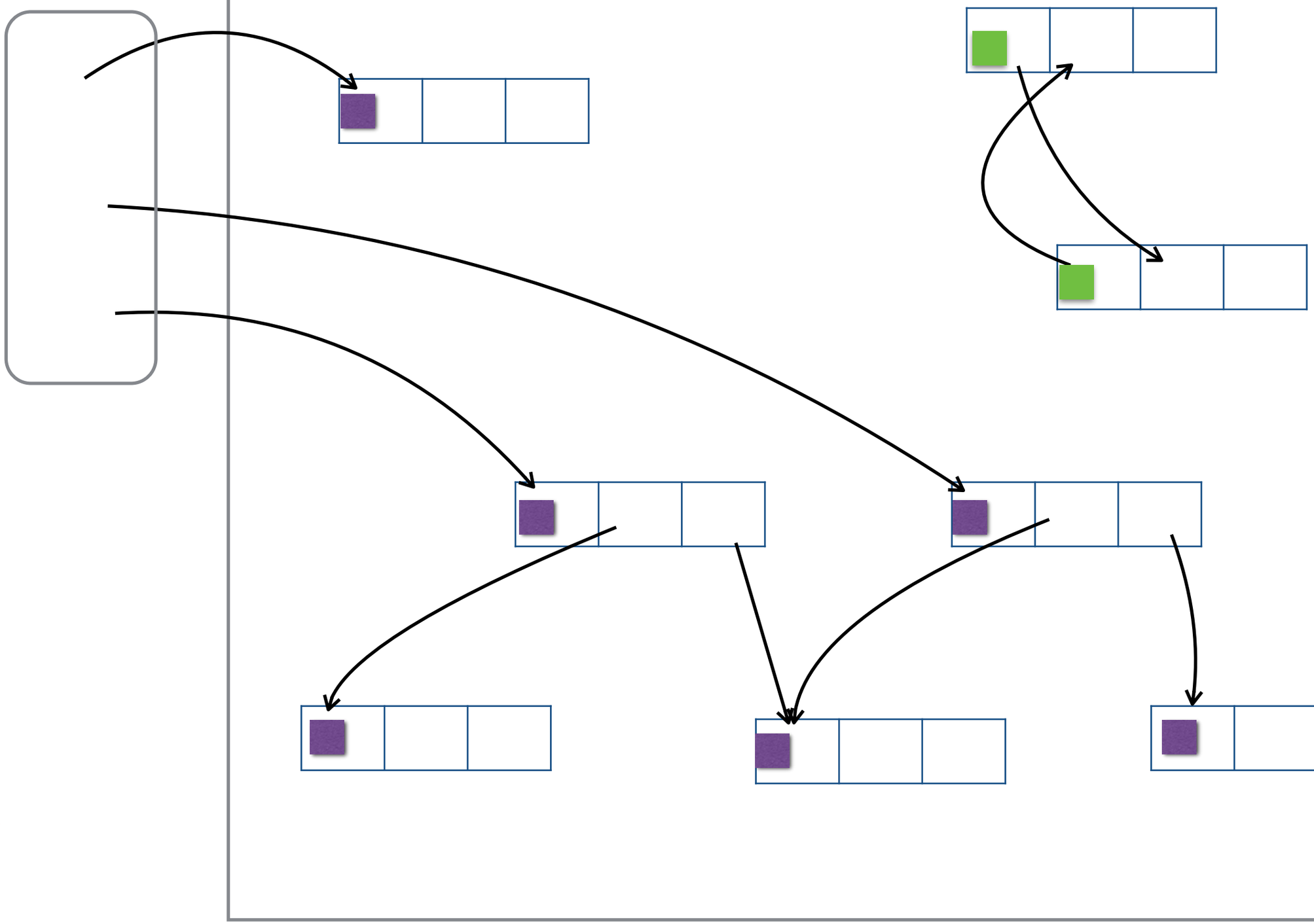
Mark reached memory with a different color

- If something is still green after this phase, it is rubbish and should be removed

Heap

Stack

These two can now be safely removed!



Tracing GC: Mark-Sweep

- Things to consider:

  Fragmentation (think about compacting garbage collectors)

  Stop-the-world

    can happen at arbitrary times and take arbitrarily long

    what about incremental and concurrent GC?

  Keep things marked as non-reachable

Food for Thought…

- Will GC happen??

  Some objects are tied to non-memory resources, whose release is an externally visible program behaviour:

  such as closing a network connection

  releasing a device or closing a file

  …

  But reference counting can handle this!!

Food for Thought…

- What about time overhead? What do we want?

  Tracing garbage collection:

  Allocating a new object can sometimes return quickly and at other times trigger a lengthy garbage collection cycle.

  Reference counting:

  allocation of objects is usually fast

  BUT decrementing a reference is nondeterministic, since a reference may reach zero, triggering recursion to decrement the reference counts of other objects which that object holds.