

Application Performance Management

Frühling 2021

Garbage Collection Algorithmen

Zoltán Majó

Outline

Übersicht Speicherverwaltung

Gängige Garbage Collection Techniken

Danke an Prof. Dr. Thomas R. Gross (ETH Zürich) für die Folien

Speicherverwaltung

Viele moderne Programmiersprachen unterstützen dynamische Speicherallokation

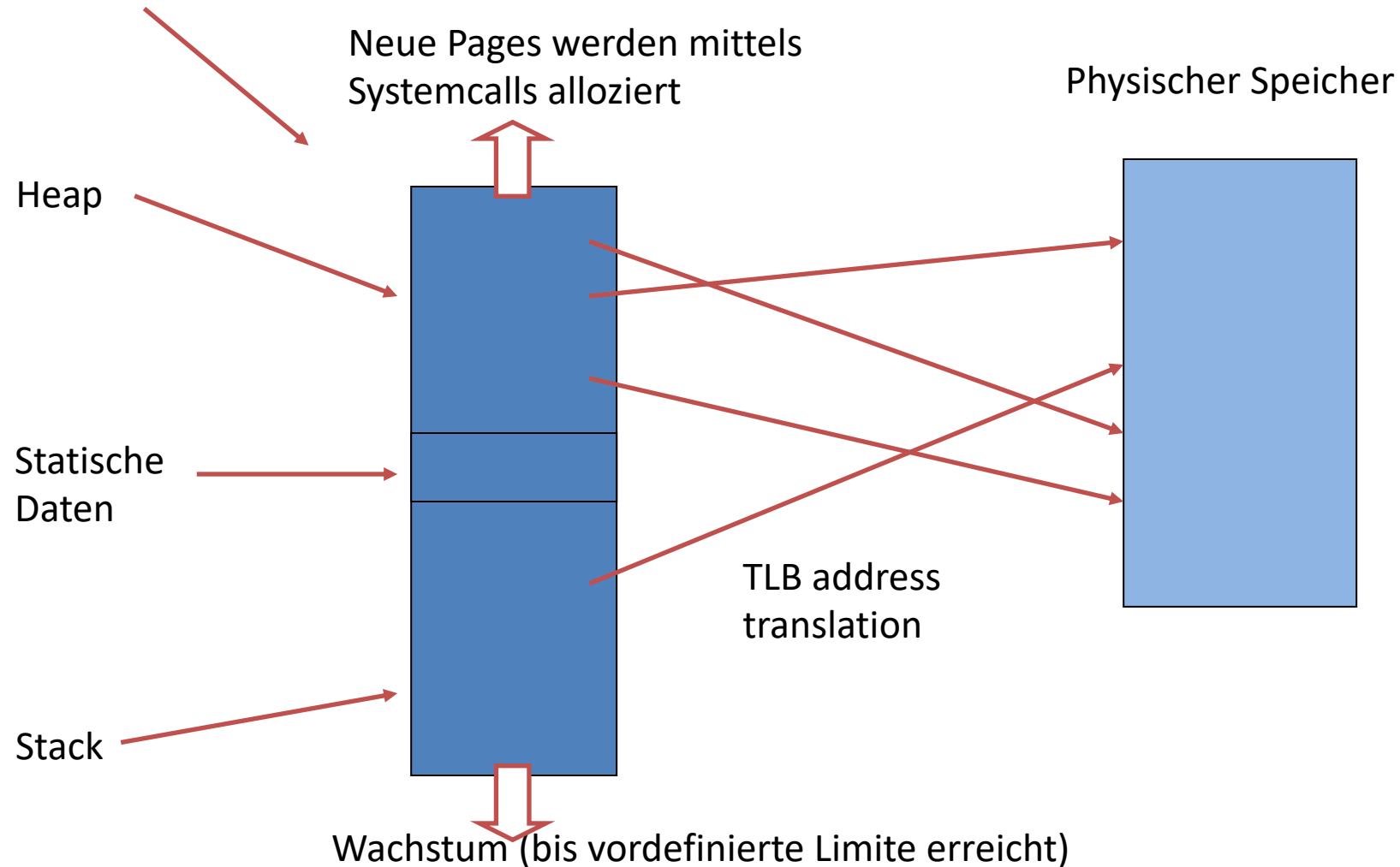
- Programme können z.B. Records, Arrays und Objekte zur Laufzeit allozieren

Die Programmiersprache muss mit der Zurückforderung und Recycling allozierter Speicher umgehen können

- Teil der Spezifikation der Programmiersprache
- Aufgabe dem Runtime System (z.B., der Java Virtuelle Maschine) überlassen

Speicherlayout

Virtueller Speicher
(per Prozess)



Speicherplatz

Virtueller Speicher ist unbegrenzt

- Zumindest konzeptuell

Physischer Speicher ist begrenzt

- Limite kann vom Betriebssystem gesetzt werden
- Limite kann beim Prozessstart gesetzt werden (wegen anderen Prozessen)
- Adressraum ist limitiert

Performanz ist wichtig

- Festplatten sind weniger performant als Hauptspeicher

Unbenutzte Daten müssen entfernt werden

→ «Garbage Collection» (GC)

GC

Was ist «Garbage» (dt. Müll)?

- Ein Objekt im Programm ist Müll, wenn das Objekt von keiner Berechnung wieder verwendet wird.

Ist es einfach festzustellen, welche Objekte Müll sind?

- Nein. Es ist unentscheidbar. Zum Beispiel:

```
v = new Object();  
if (long-and-tricky-computation) {  
    use v  
} else {  
    don't use v  
}
```

GC (Forts.)

Da es schwierig ist festzustellen, welche Objekte Müll sind, unterstützen Programmiersprachen unterschiedliche Ansätze

Ansatz 1: Der Programmierer muss sich darum kümmern

- Explizite Allokation/Deallokation

Ansatz 2: Das Laufzeitsystem muss sich darum kümmern

- Automatisch
- Viele Algorithmen

Ansatz 1: Explizite Speicherverwaltung

Verwaltung des Speichers mittels einer Bibliothek

Programmierer entscheidet wann und wo Speicher alloziert/dealloziert wird

```
void* malloc(long n)
void free(void *addr)
```

Wenn nötig, die Bibliothek beantragt mehr Pages vom Betriebssystem

- Mittels Systemcalls

Vorteile/Nachteile der expliziten Speicherverwaltung

Vorteile:

- Programmierer sind schlau
- Programmierer entscheidet, wann der Mehraufwand der Allokierung akzeptabel ist

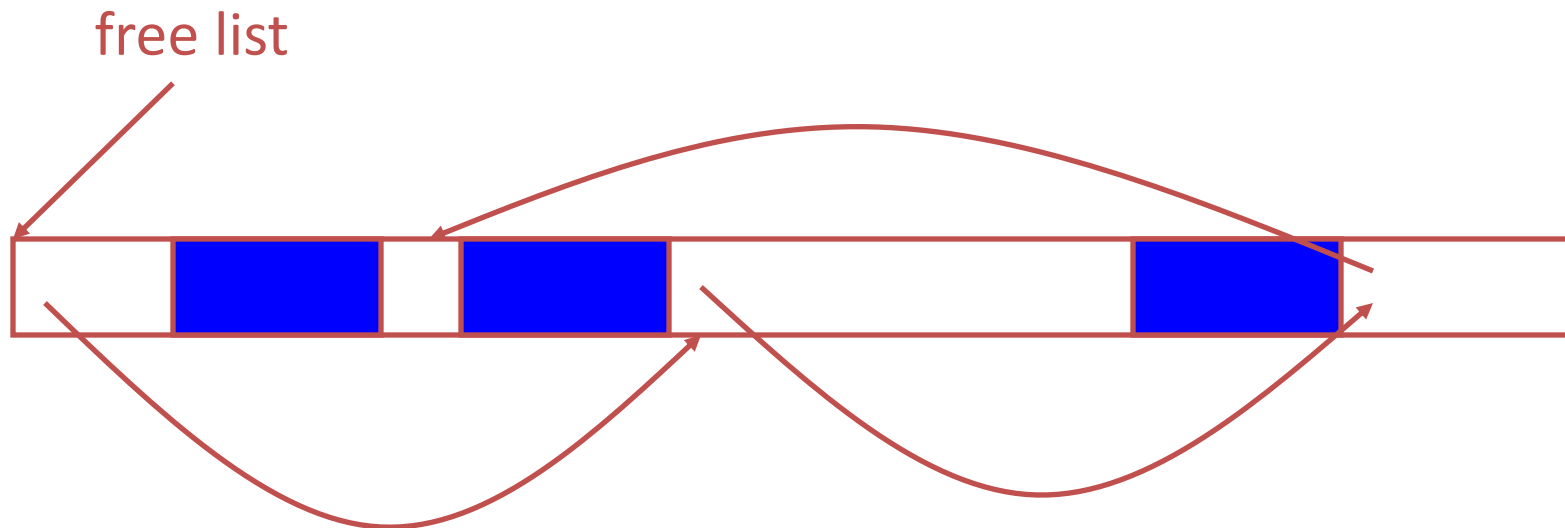
Nachteile:

- Auch schlaue Programmierer machen Fehler
- Programmierer möchten sich nicht unbedingt mit solchen Details beschäftigen
- Automatische Speicherverwaltung kann günstig sein

Explizite Speicherverwaltung: Details

Wie funktioniert malloc/free?

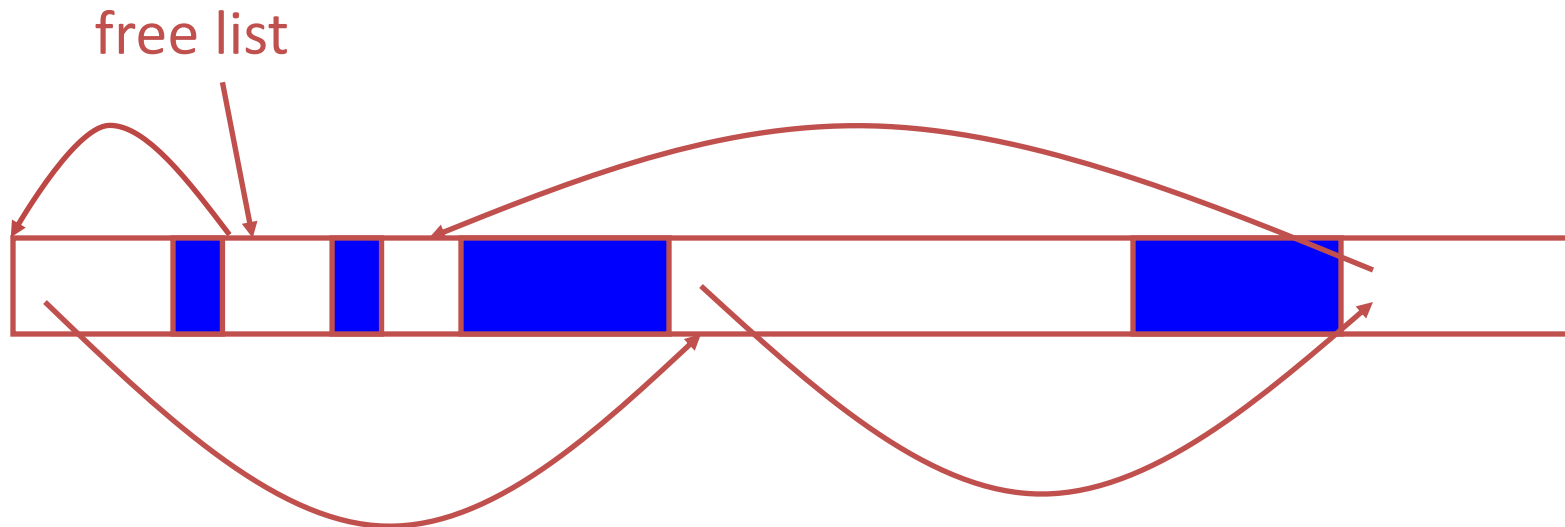
- Nicht (mehr) verwendete Speicherblöcke befinden sich in der “free list”
- `malloc`: sucht in der free list nach einem genug grossen Speicherblock
- `free`: plaziert Speicherblock zum Anfang der free list



Explizite Speicherverwaltung: Details

Wie funktioniert malloc/free?

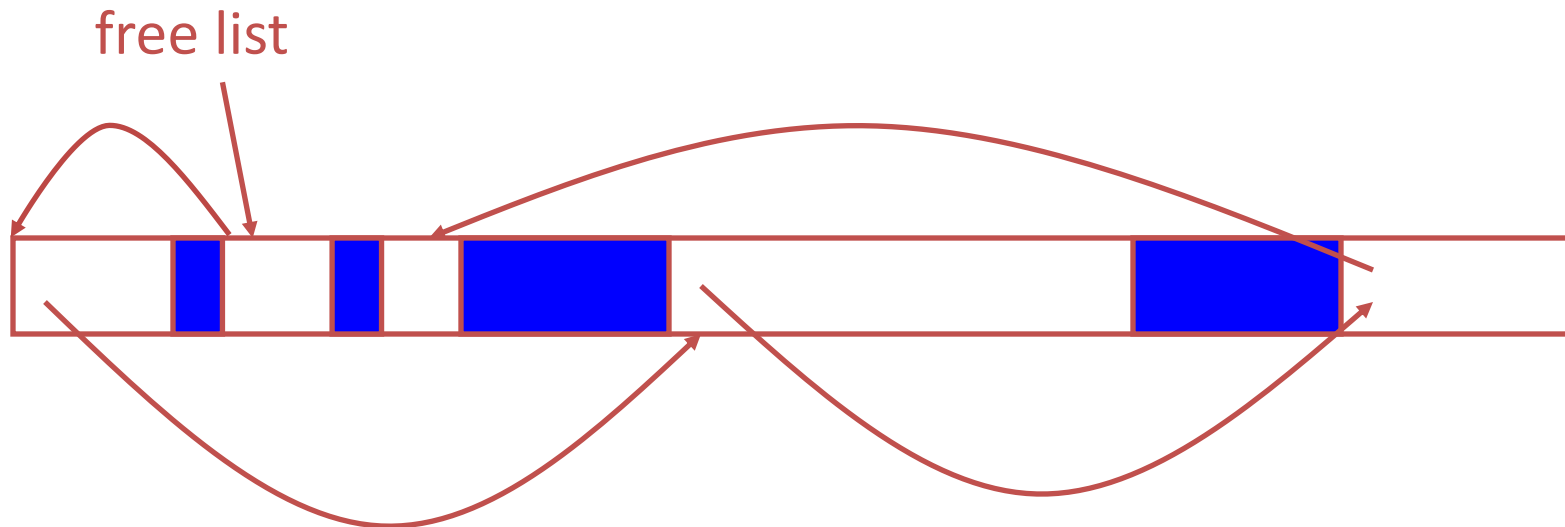
- Nicht (mehr) verwendete Speicherblöcke befinden sich in der “free list”
- `malloc`: sucht in der free list nach einem genug grossen Speicherblock
- `free`: platziert Speicherblock zum Anfang der free list



Explizite Speicherverwaltung: Details

Nachteile

- `malloc` is nicht für umsonst: Aufwand der Suche nach einem Block, der gross genug ist, kann signifikant sein
- Der Heap wird fragmentiert, während das Programm ausgeführt wird



Mögliche Lösungen

Mehrere free lists

- Eine free list für jede gegebene Blockgrösse
- Malloc und free sind beide $O(1)$
- Mögliches Problem: Liste mit Blöcken der Grösse 4 ist verbraucht, auch wenn Blöcke der Grösse 2 und 6 verfügbar ist

Blöcke sind Zweierpotenzen

- Blöcke werden aufgeteilt um richtige Grösse zu erreichen
- Bei Freigabe werden angrenzende Blöcke zusammengeschmolzen

Fragmentierung in jedem Fall vorhanden

- Verschwendeter Speicherplatz
- «No magic bullet»: Speicherverwaltung kostet immer was

Automatische Speicherverwaltung – wieso?

Programmieren mit expliziter Speicherverwaltung viel schwieriger ist als mit automatischer Speicherverwaltung

- Konstante Sorge wegen «Dangling Pointers»
 - Instabilität, Maintenance
- Es ist unmöglich ein sicheres System zu entwickeln
 - System gibt keine Garantien
- Programmieren mit Sprachen, die automatische Speicherverwaltung unterstützen, ist einfacher
- Unterliegendes Laufzeitsystem kann den Speicher immer noch explizit verwalten

Ansatz 2: Automatische Speicherverwaltung

Zentrale Frage:

Wie wird entschieden, welche Objekte Müll sind?

- (Ein Objekt im Programm ist Müll, wenn keine Berechnung im Programm dieses Objekt wieder verwendet.)

Übliche Lösung: Ein Objekt ist Müll, wenn es von den "Roots" aus nicht mehr erreichbar ist

- Roots = Register, Stack, globale statische Daten
 - Falls es vom Root aus zu einem Objekt keinen Pfad gibt, das Objekt kann nicht mehr im Programm verwendet werden und kann daher zurückgefordert werden.
- Zurückhaltende Approximation
 - Engl. «conservative approximation»

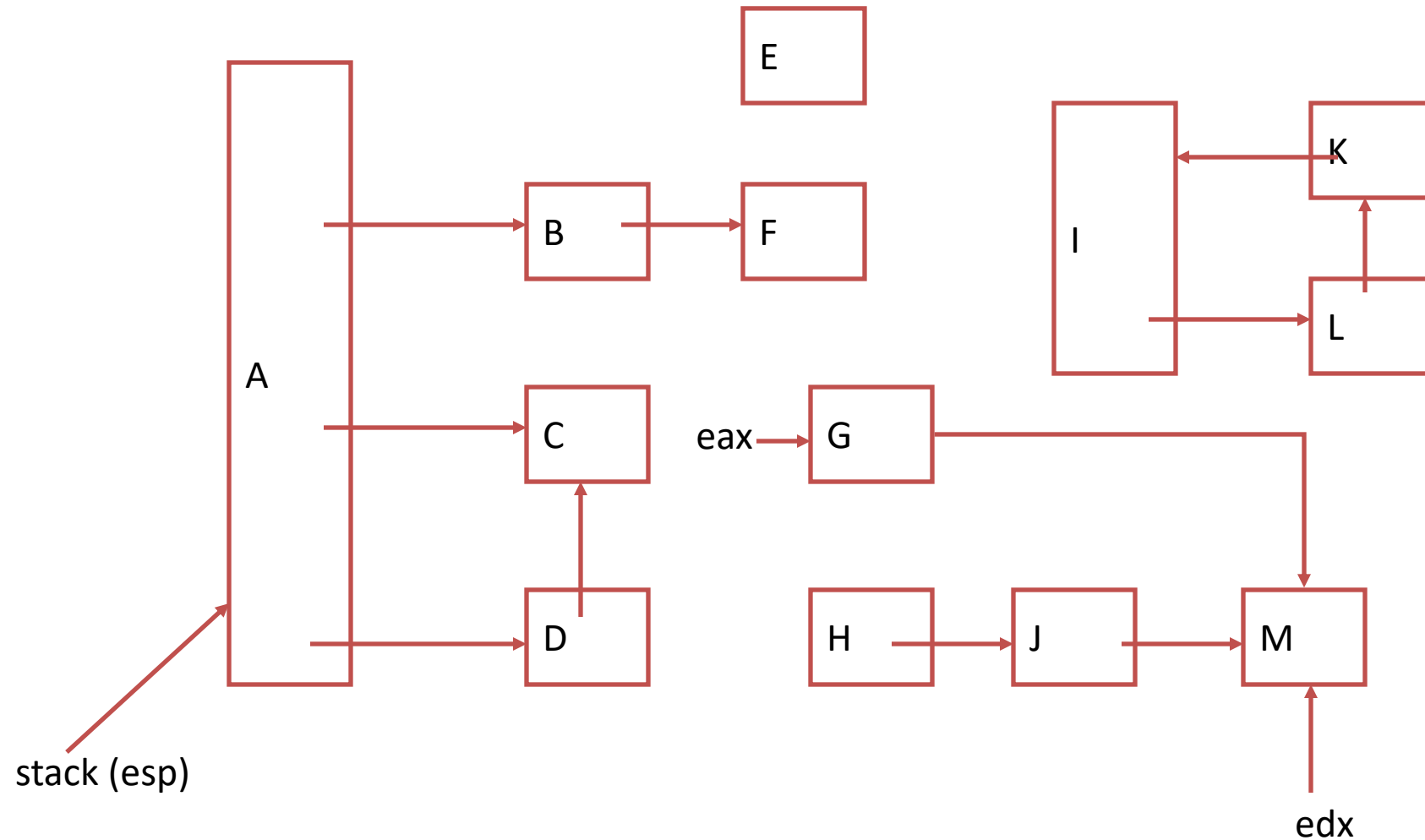
Ansatz 2: Automatische Speicherverwaltung (Forts.)

Es gibt verschiedene Ansätze um automatische Speicherverwaltung zu realisieren

Die meisten Differenzen sind bezüglich

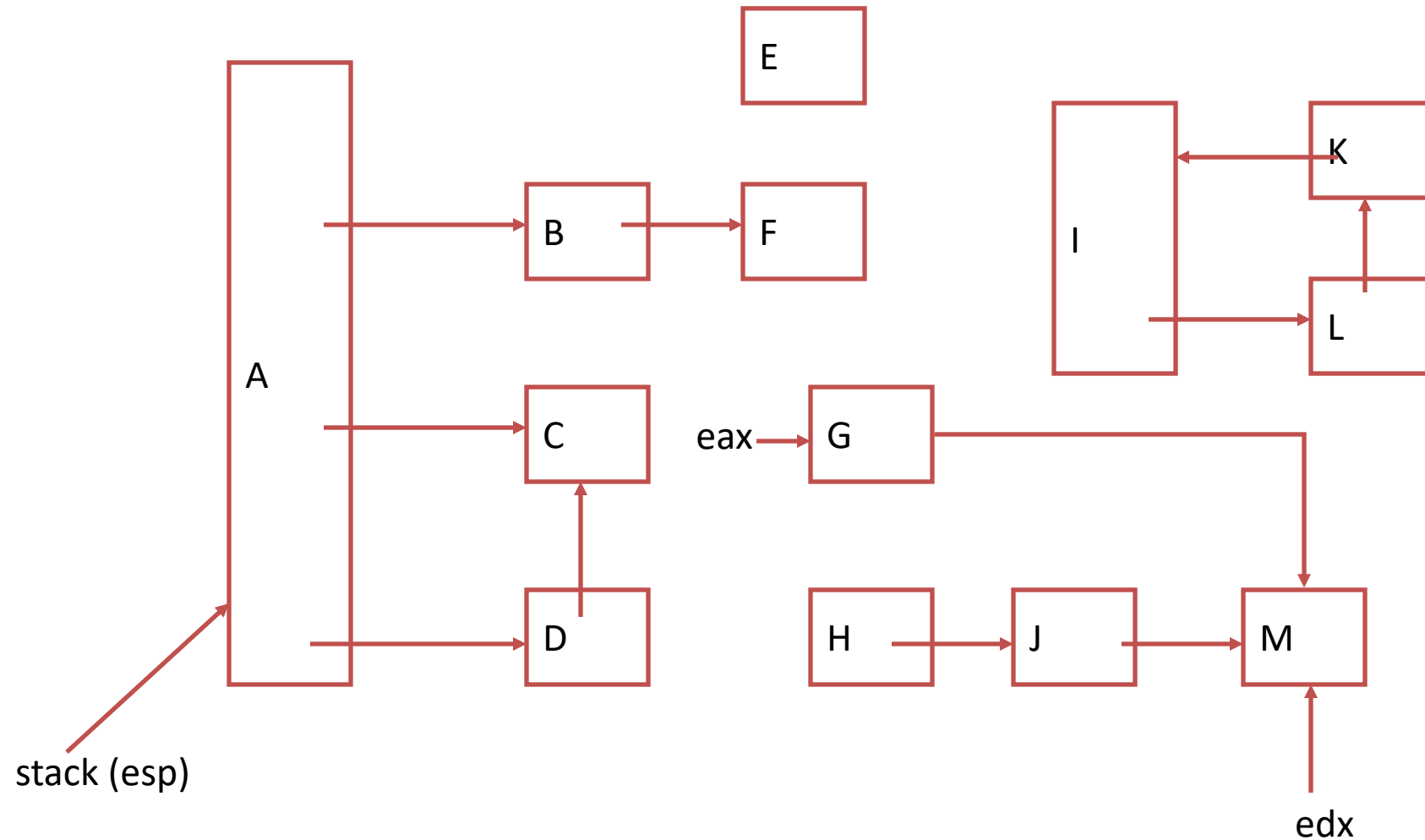
- Wie wird entschieden, welche Objekte nicht erreichbar sind
- Wie werden (nicht) erreichbare Objekte behandelt

Objektgraph (eines Programmes)



Welche Objekte sind erreichbar (von den Roots aus)?

Objektgraph (eines Programmes)



Wie können wir feststellen, welche Objekte erreichbar sind?

Übersicht

Reference Counting

Mark & Sweep GC

Mark & Copy GC

Mark & Compact GC

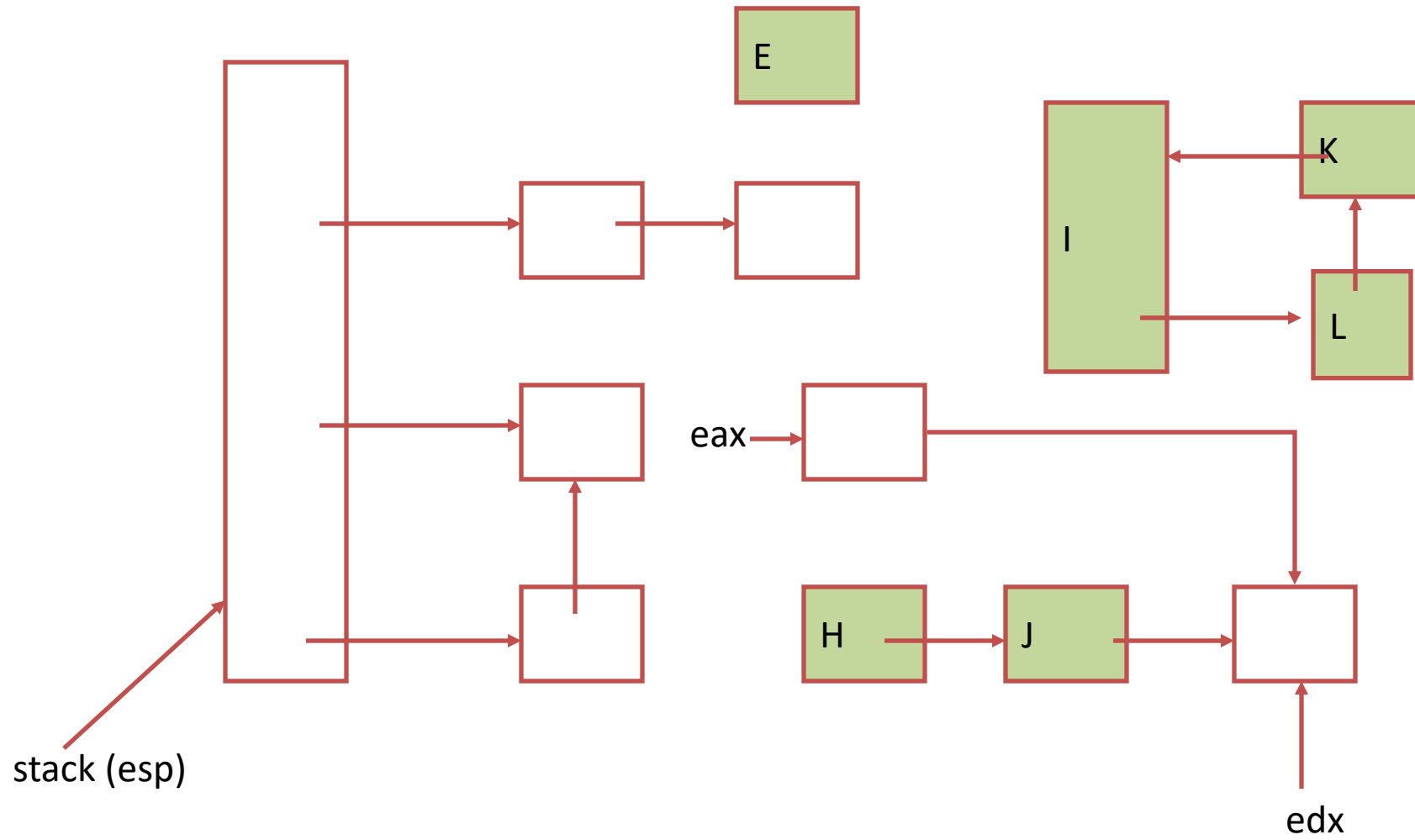
Generational GC

Reference Counting

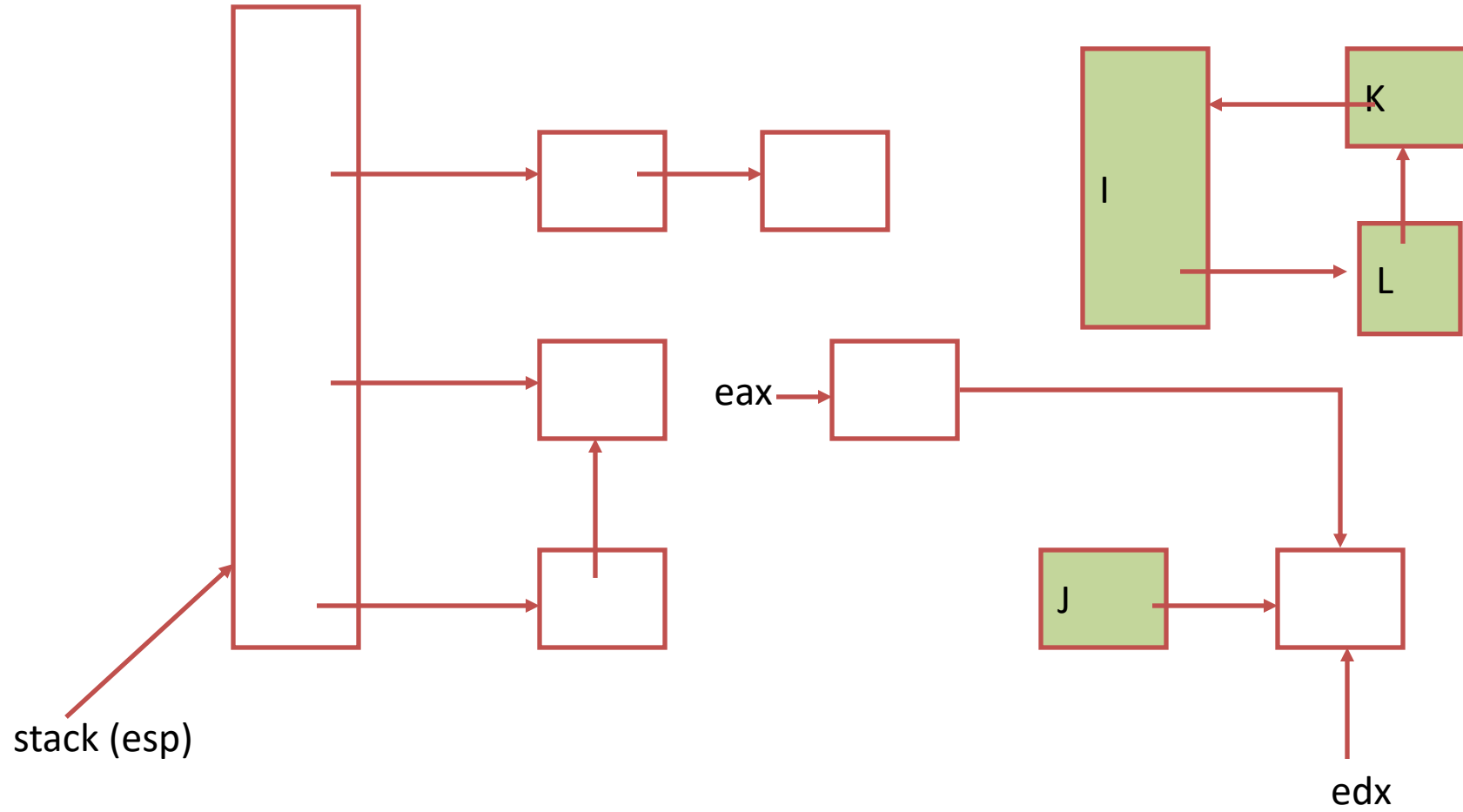
Einfache Idee: Für jedes Objekt führt das System Buch über die Anzahl Referenzen zum Objekt

- Falls die Anzahl Referenzen zu einem Objekt 0 erreicht, das Objekt ist nicht erreichbar (und daher Müll)

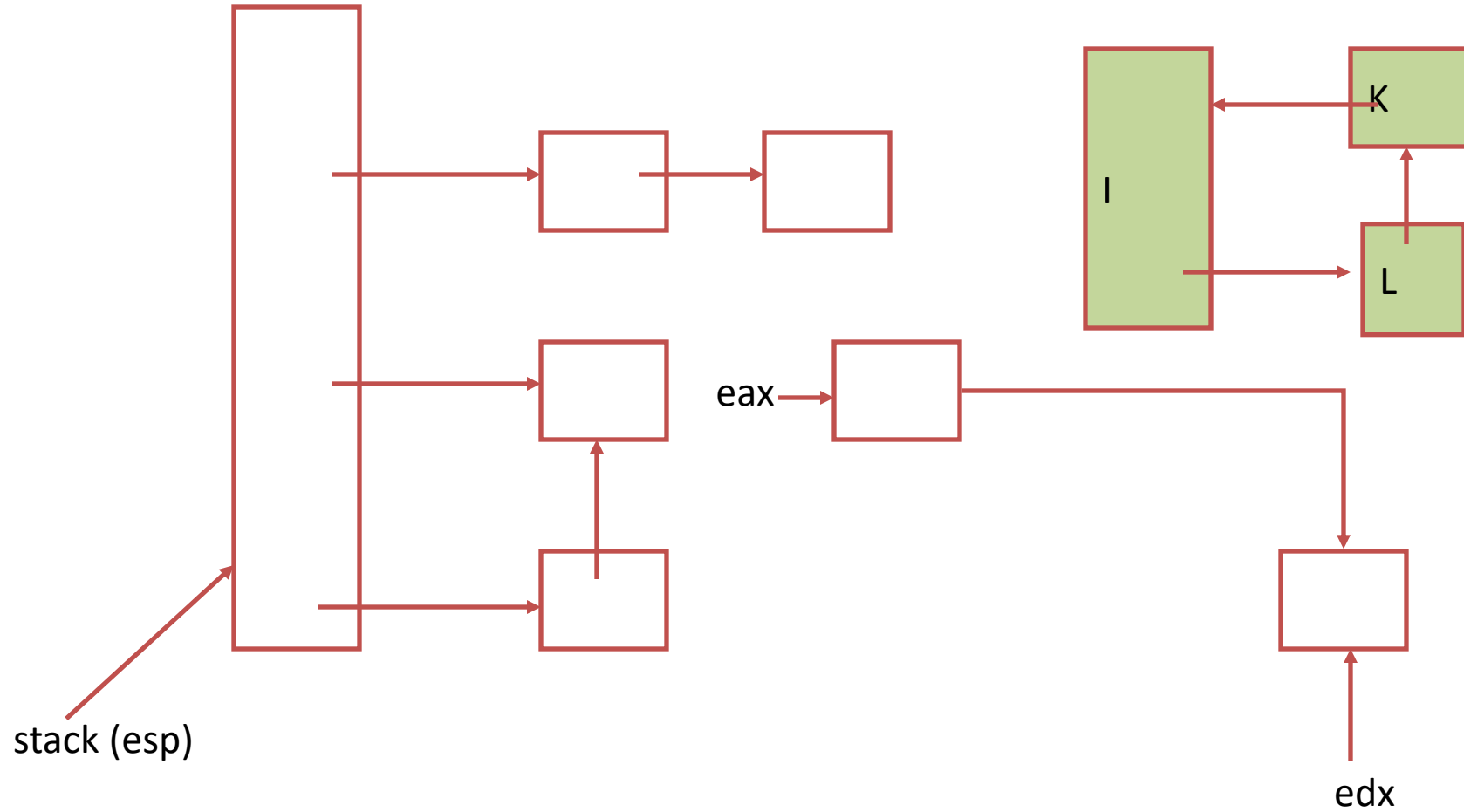
Objektgraph



Objektgraph



Objektgraph



Reference Counting kann Zyklen nicht detektieren

Reference Counting im Hintergrund

Einfache Zuweisung im Programm $x.f = p$

Was muss im Hintergrund passieren?

```
z = x.f
c = z.count
c = c - 1
z.count = c
If c = 0
    call putOnFreeList(z)
x.f = p
c = p.count
c = c + 1
p.count = c
```

Ouch

Übersicht

Reference Counting

Mark & Sweep GC

Mark & Copy GC

Mark & Compact GC

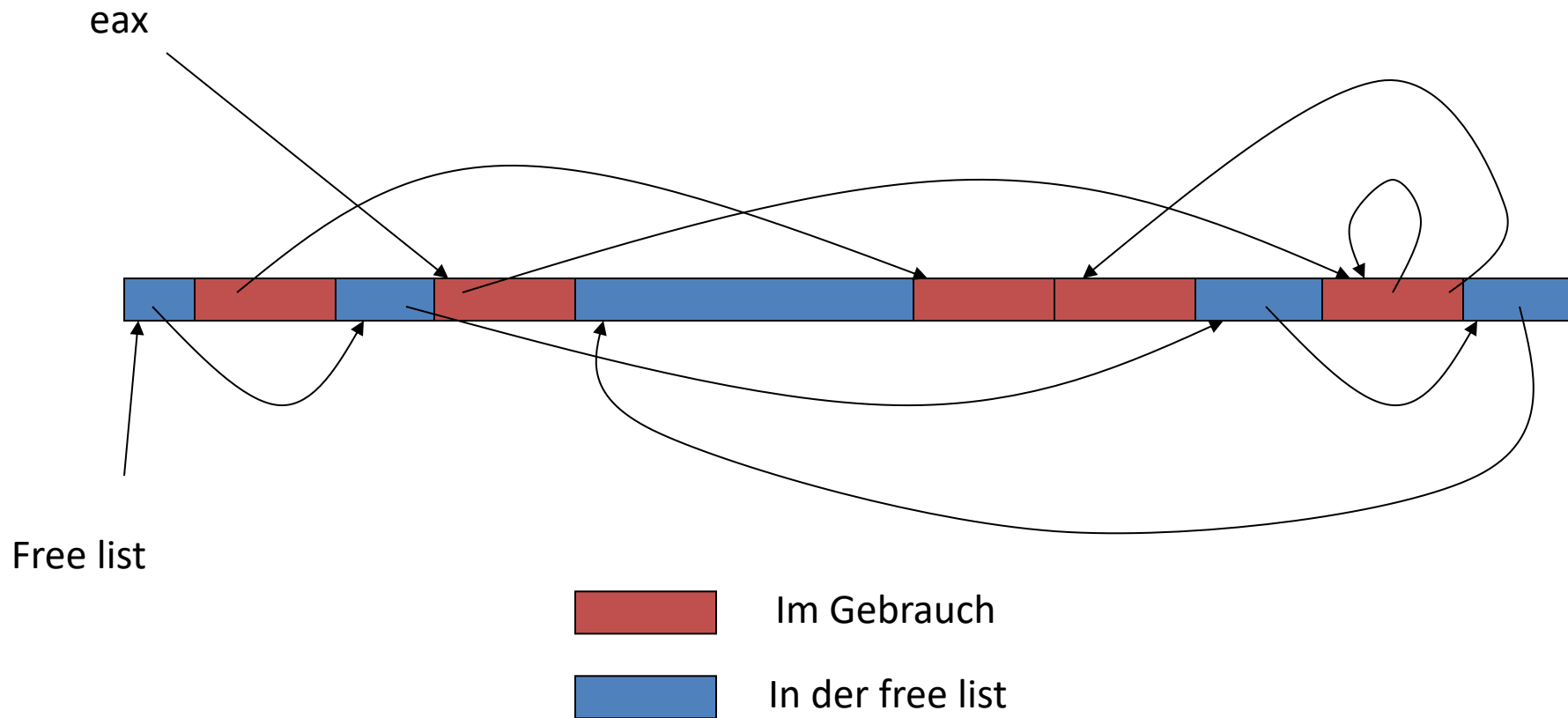
Generational GC

Mark & Sweep GC

Algorithmus besteht aus zwei Phasen

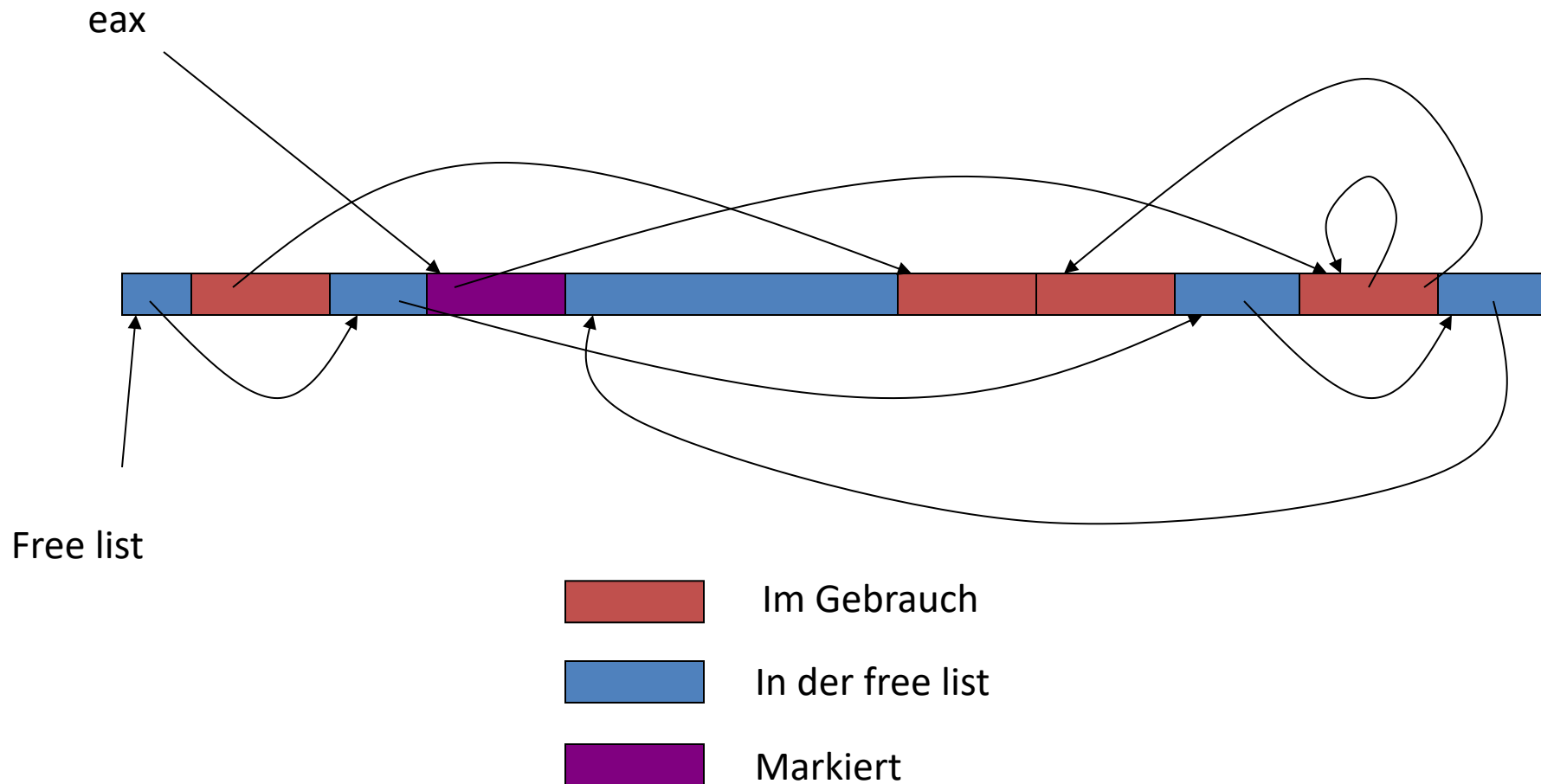
- **Mark:** Objektgraph wird "depth-first" durchquert und vom Root aus erreichbare Objekte werden markiert
- **Sweep:** Der ganze Heap wird durchquert, nicht markierte Objekte werden der free list zugefügt, die Markierung aller Objekte wird gelöscht

Mark & Sweep: Zeitlupe



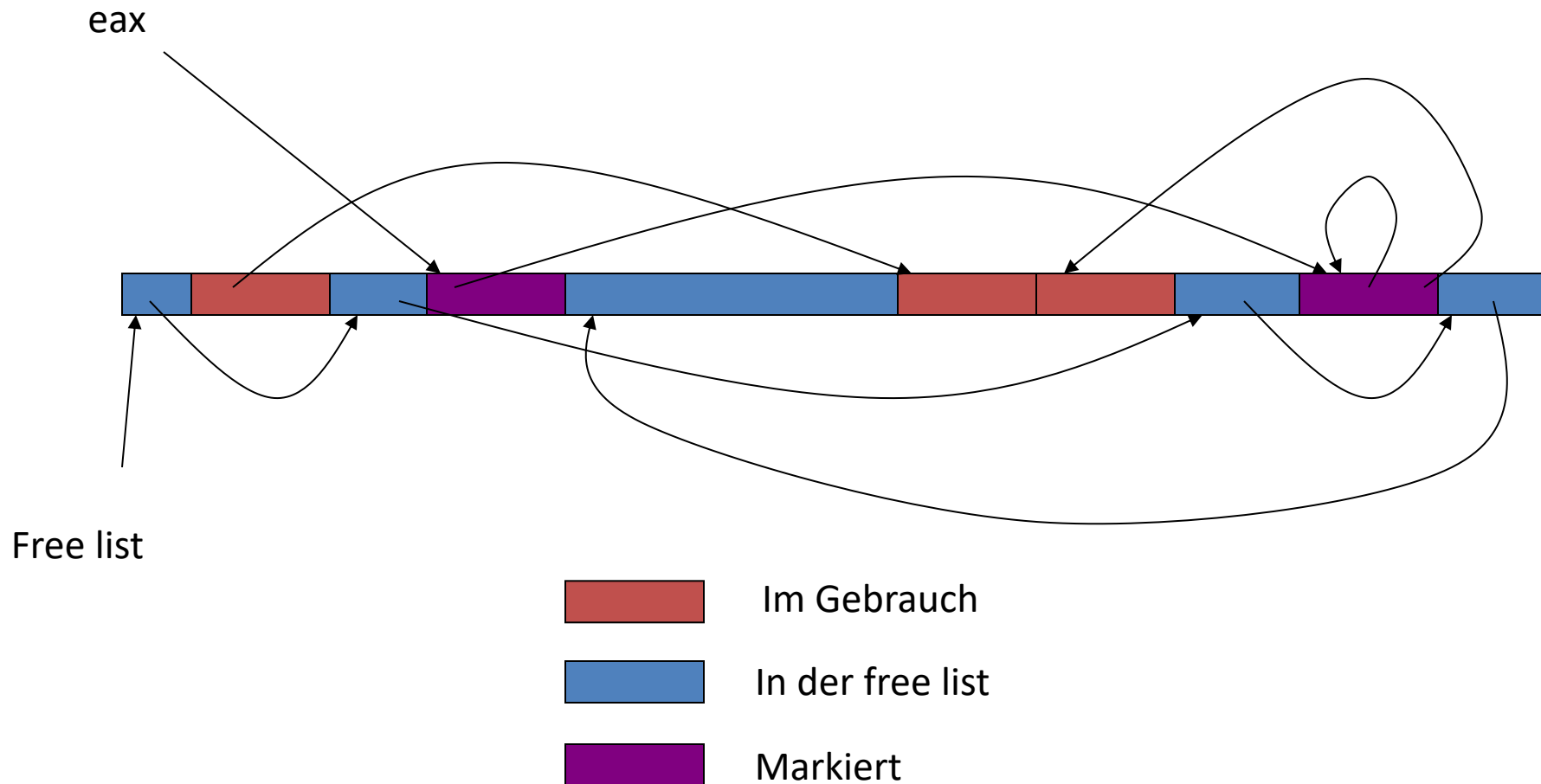
Mark & Sweep: Zeitlupe

Mark Phase: vom Root aus erreichbare Objekte werden markiert



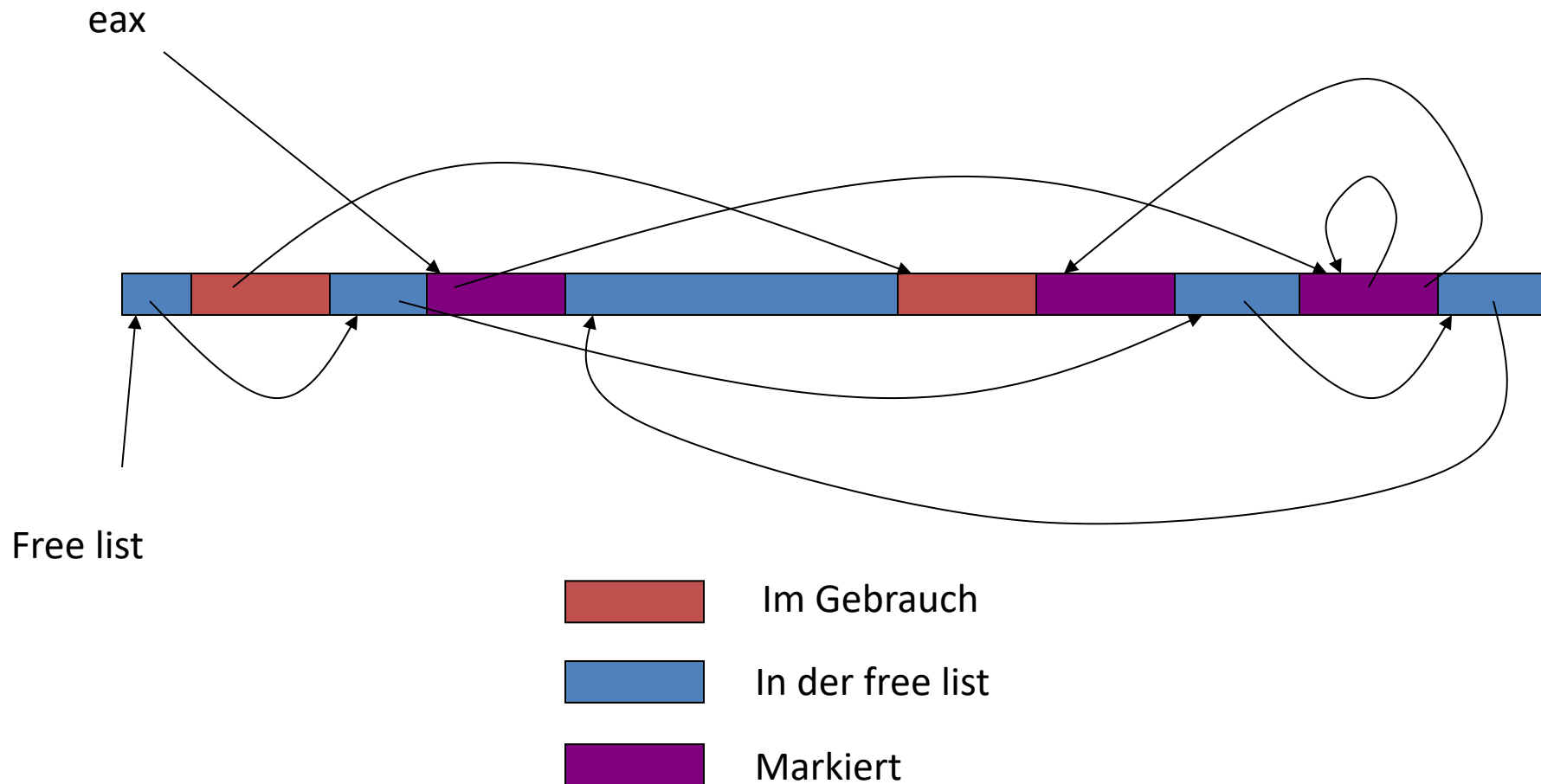
Mark & Sweep: Zeitlupe

Mark Phase: vom Root aus erreichbare Objekte werden markiert



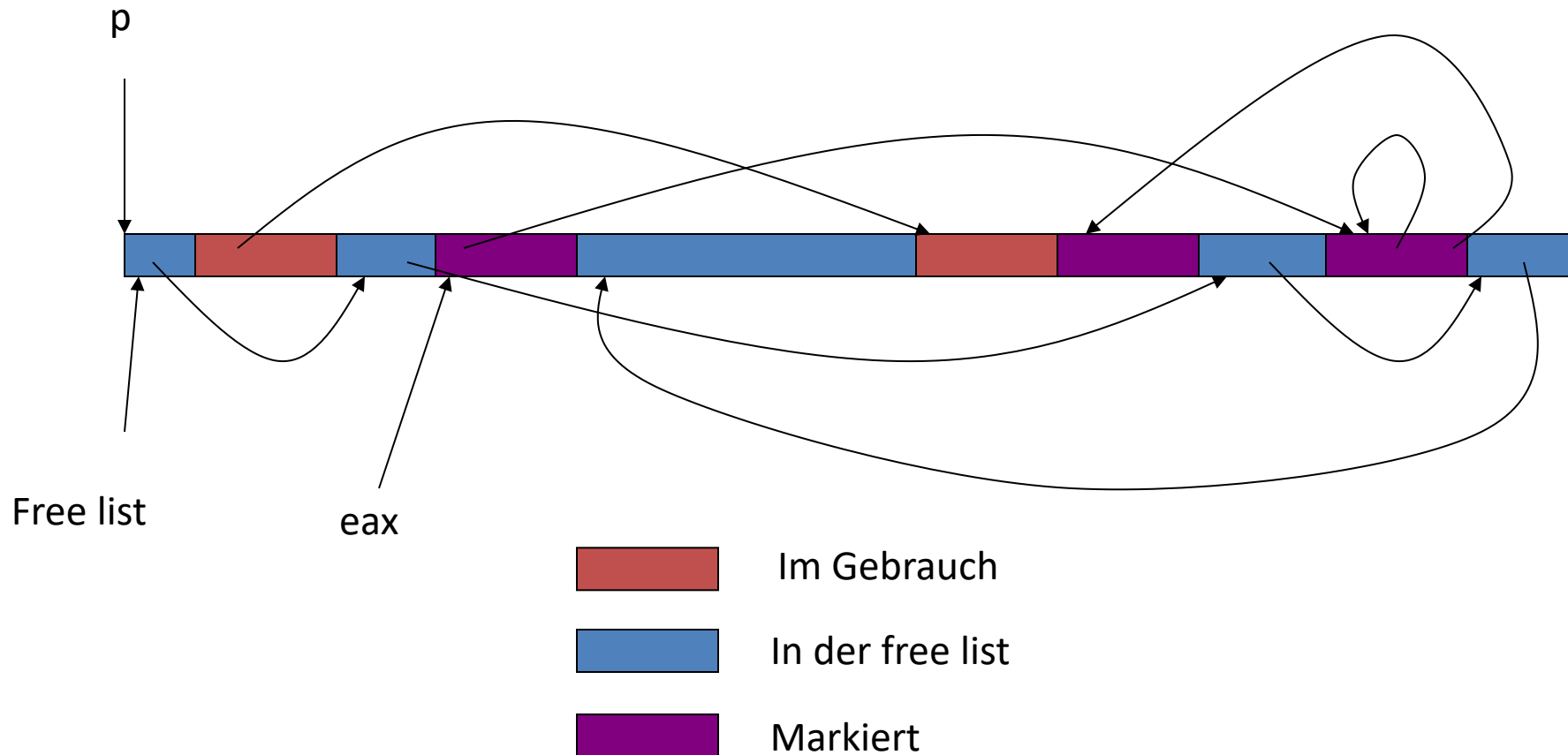
Mark & Sweep: Zeitlupe

Mark Phase: vom Root aus erreichbare Objekte werden markiert



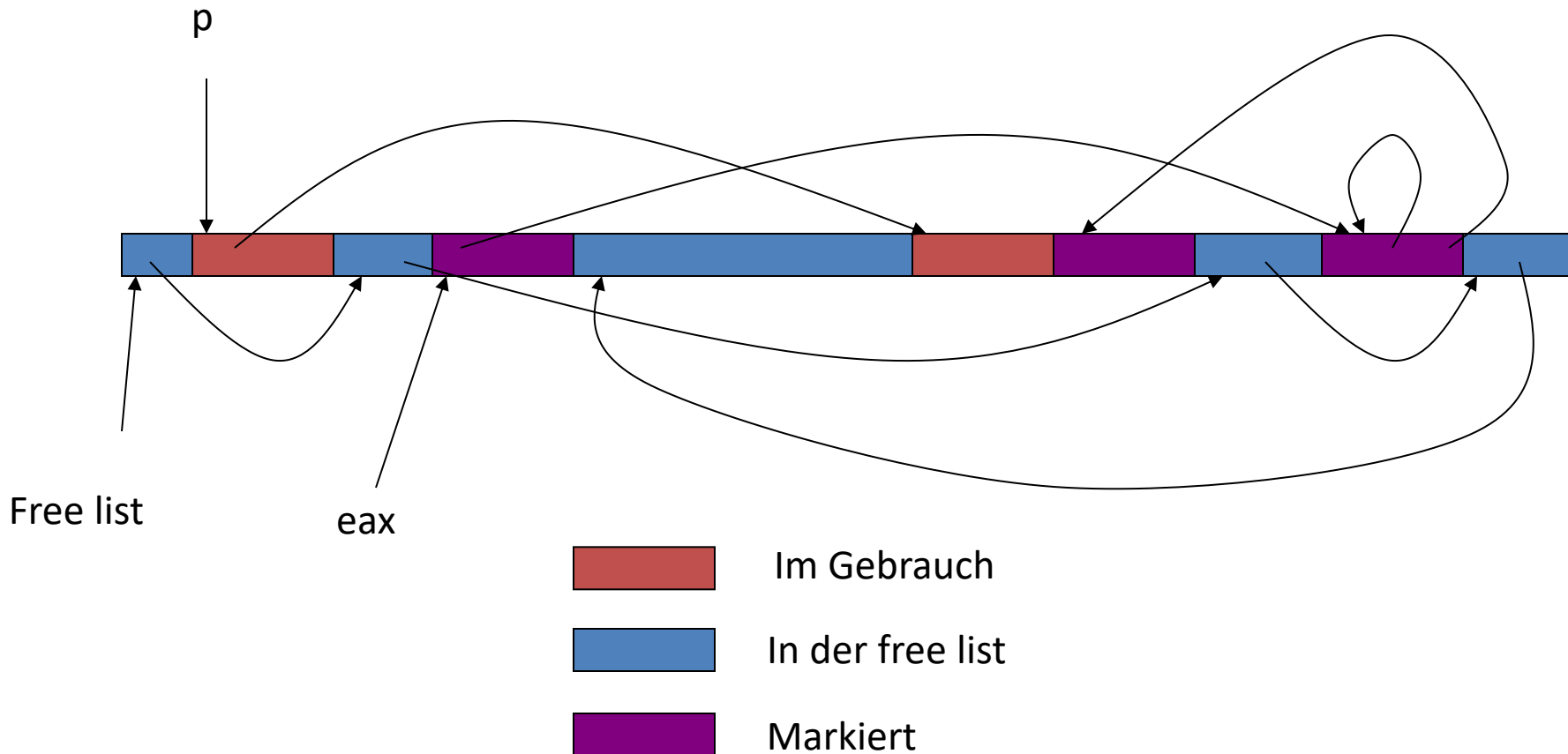
Mark & Sweep: Zeitlupe

Sweep Phase: Sweep Pointer p aufsetzen; Sweep starten



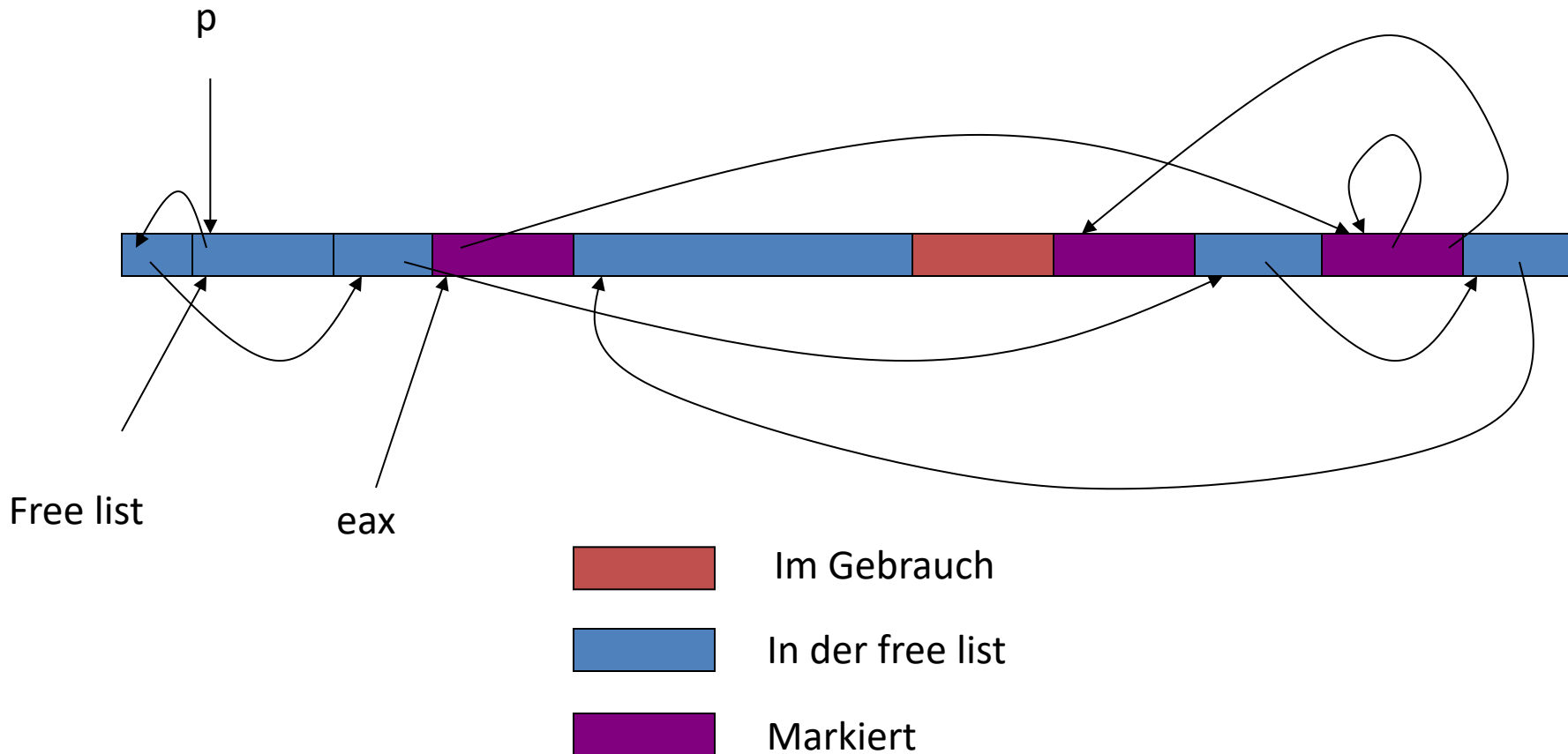
Mark & Sweep: Zeitlupe

Sweep Phase: nicht markierte Objekte der free list zufügen



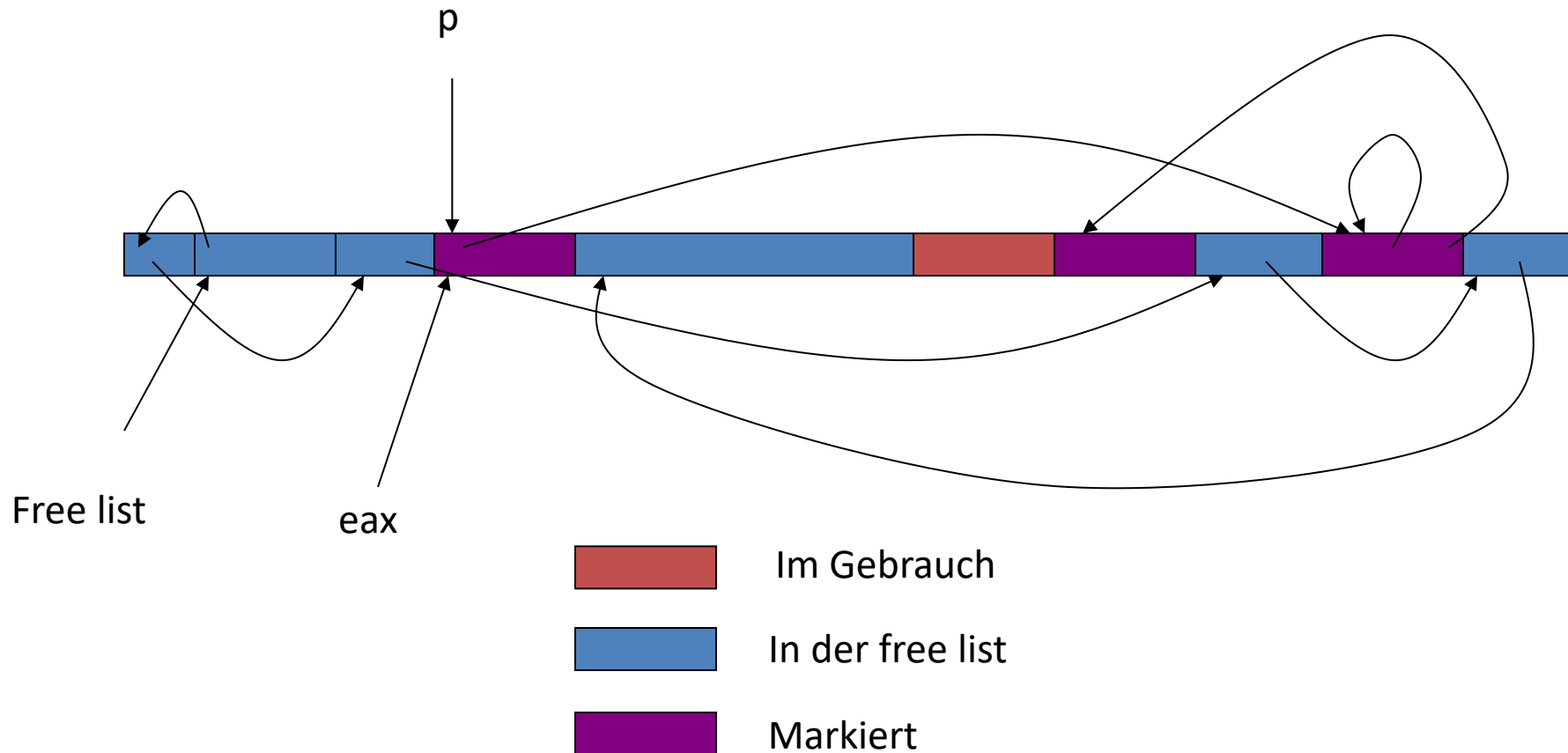
Mark & Sweep: Zeitlupe

Sweep Phase: nicht markierte Objekte der free list zufügen



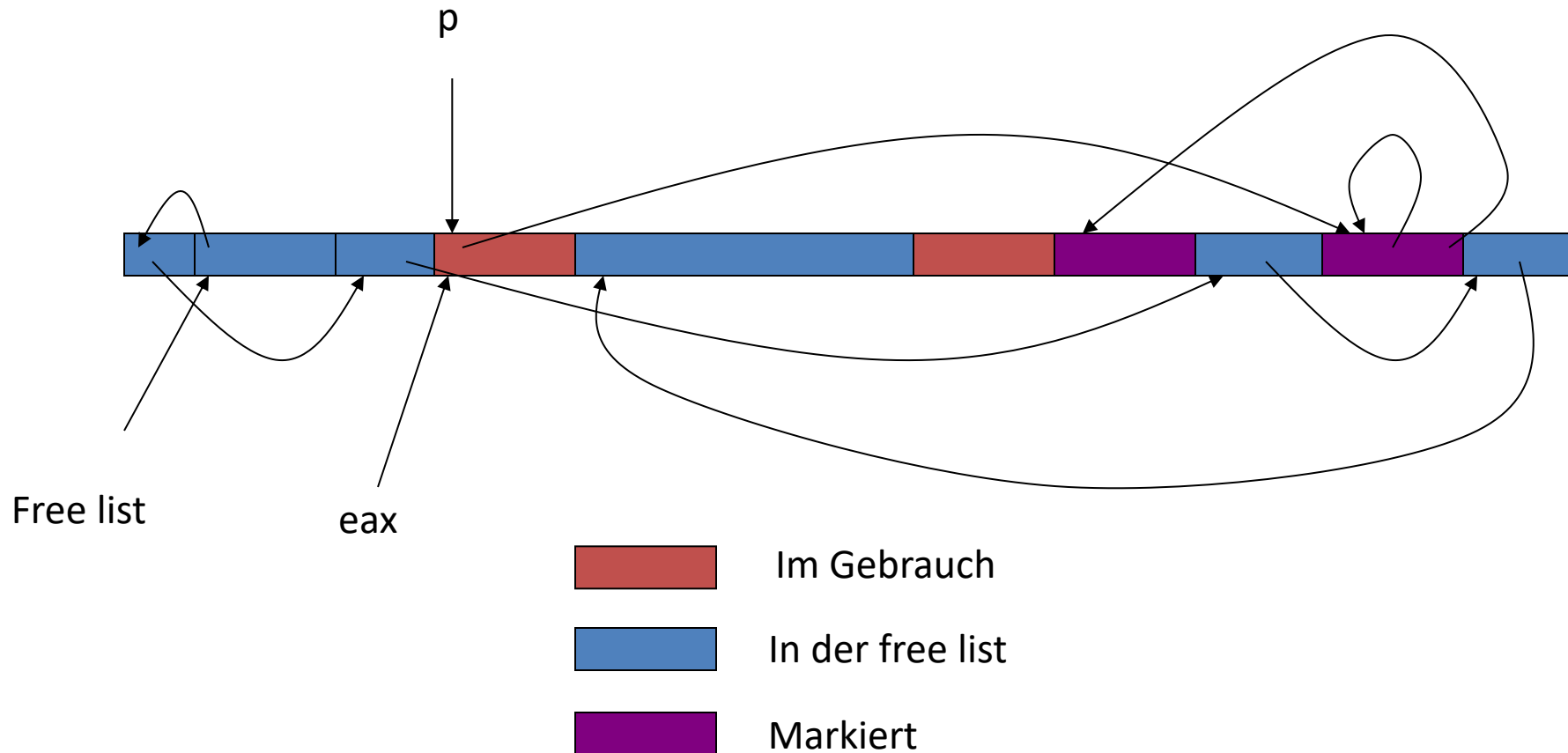
Mark & Sweep: Zeitlupe

Sweep Phase: nicht markierte Objekte der free list zufügen



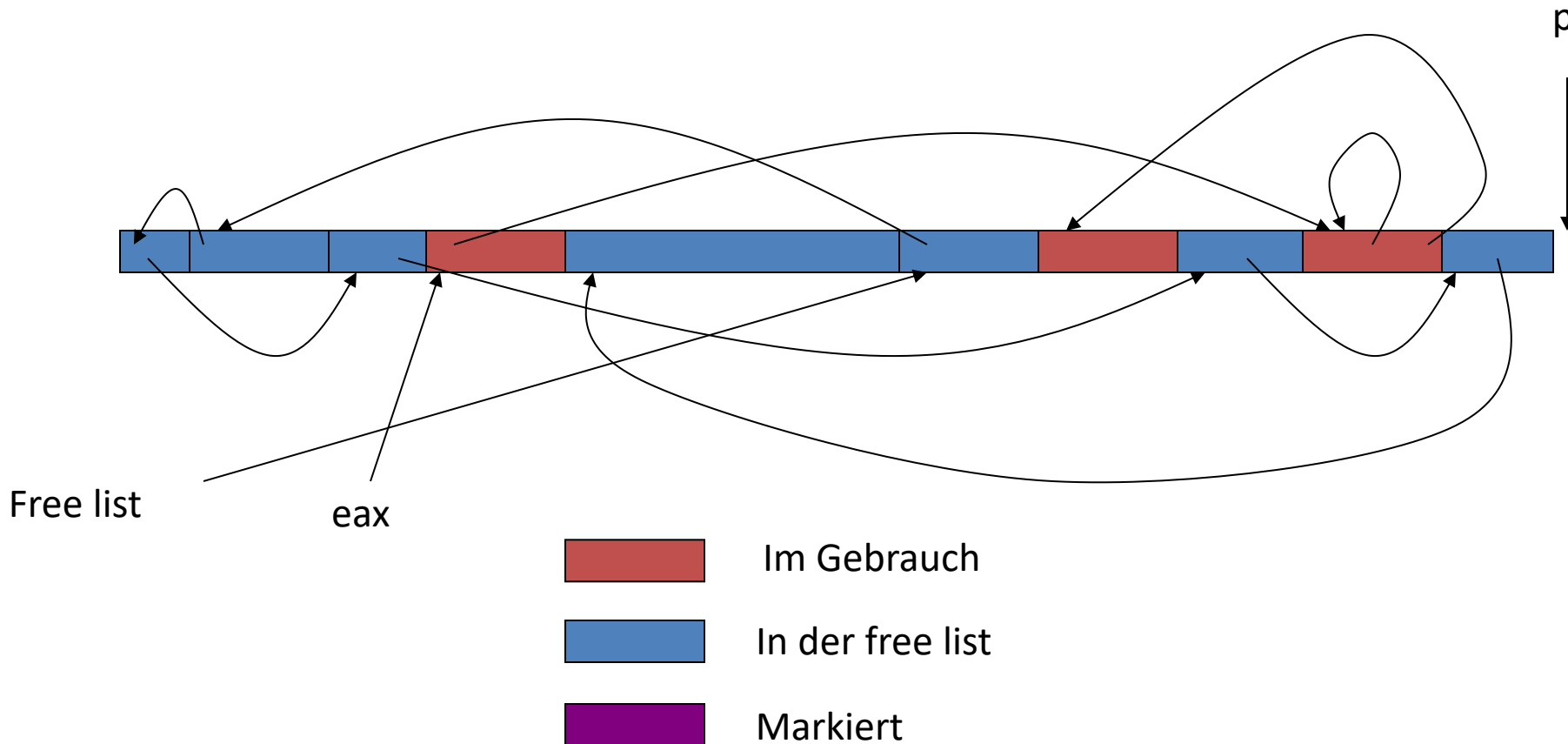
Mark & Sweep: Zeitlupe

Sweep Phase: nicht markierte Objekte der free list zufügen



Mark & Sweep: Zeitlupe

Sweep Phase: GC fertig, wenn Ende des Heaps erreicht wird; Ausführung des Programm kann wieder aufgenommen werden



Bemerkungen: Mark & Sweep GC

Vorteile

- GC wird «in situ» durchgeführt
- Kein Extra Speicherplatz nötig

Nachteile

- Fragmentierung kann ein Problem sein
- Programm muss während des GCs gestoppt werden
- Allokation kann langsam sein: Passender Block muss in der free list gesucht werden
- Sweep-Phase muss den ganzen Heap überqueren
 - Algorithmus kann noch optimiert werden

Übersicht

Reference Counting

Mark & Sweep GC

Mark & Copy GC

Mark & Compact GC

Generational GC

Mark & Copy

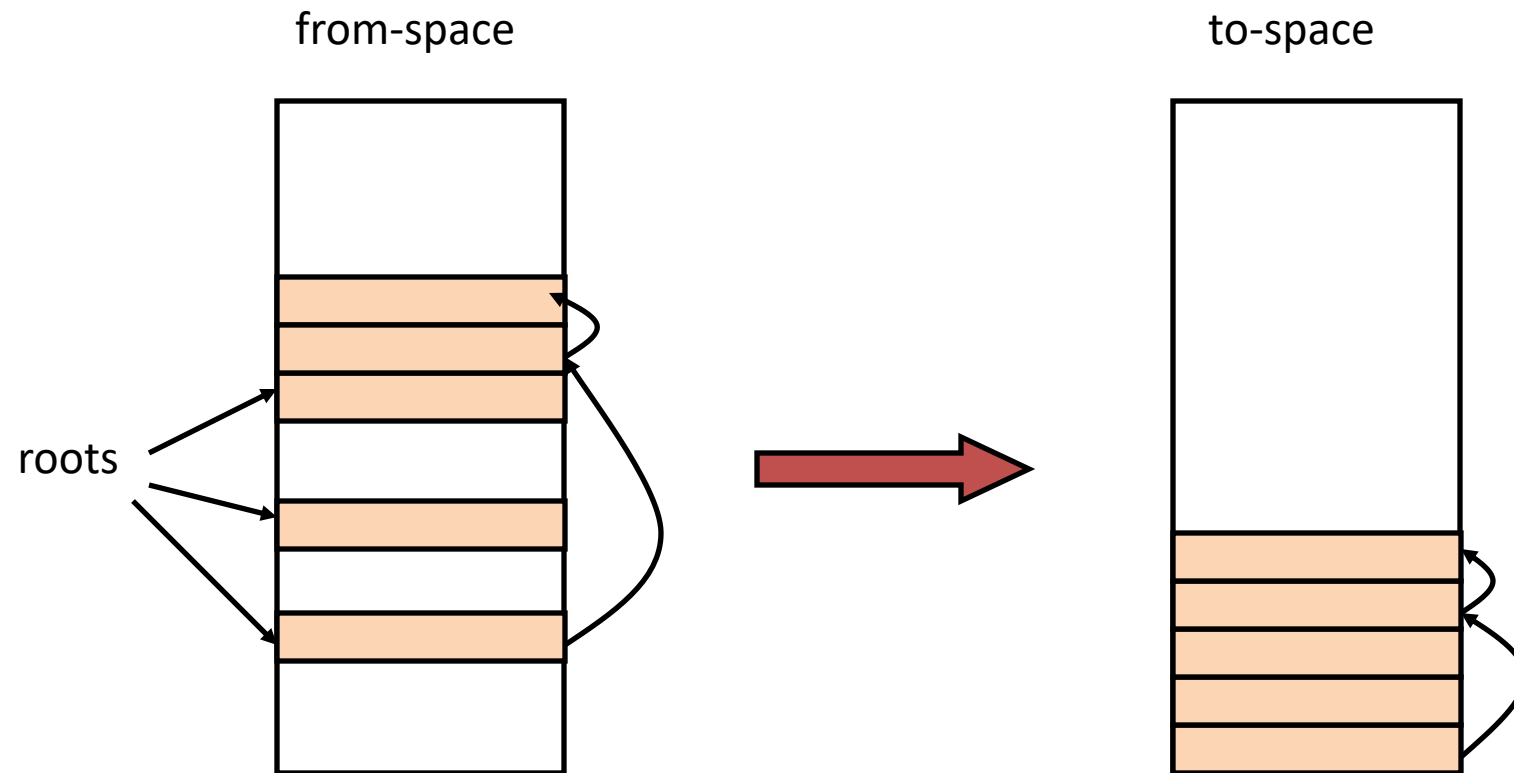
Idee: 2 Heaps werden verwendet

- Ein Heap (sog. from-space) wird vom Programm verwendet
- Der andere Heap (sog. to-space) nicht verwendet bis GC startet

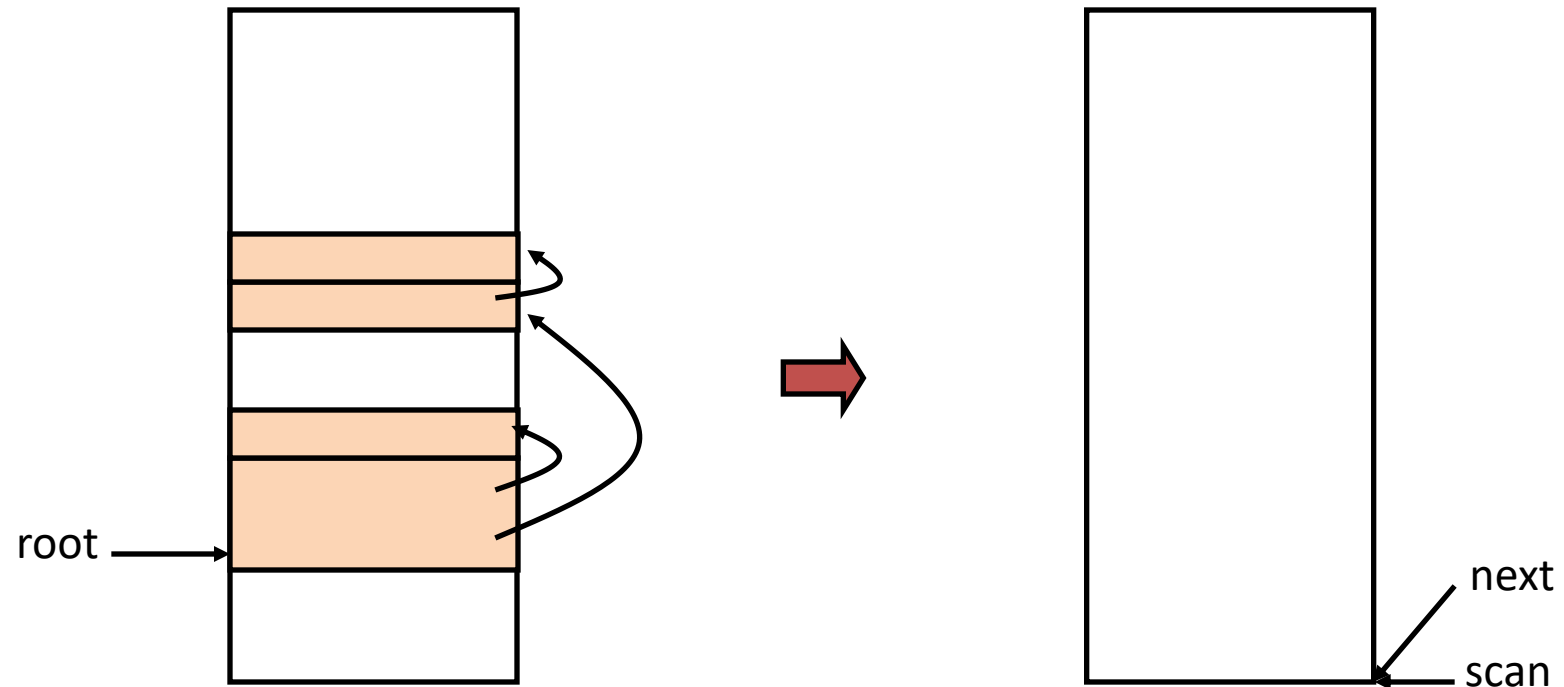
GC:

- Startet beim Root Set und traversiert den Objektgraphen
- Erreichbare Objekte werden vom from-space ins to-space kopiert
- Unerreichbare Objekte sind im from-space hinterlassen
- Die Rolle der Heaps wird gewechselt

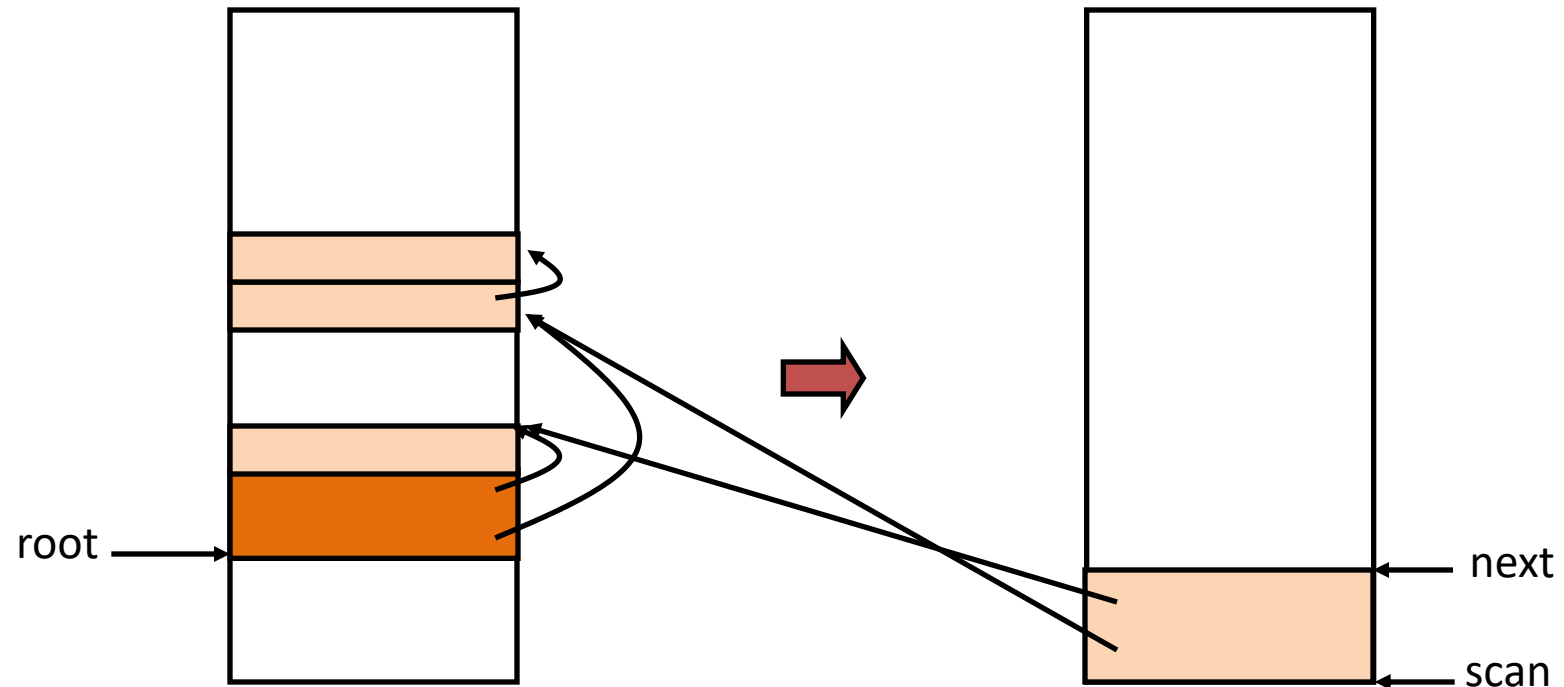
Mark & Copy



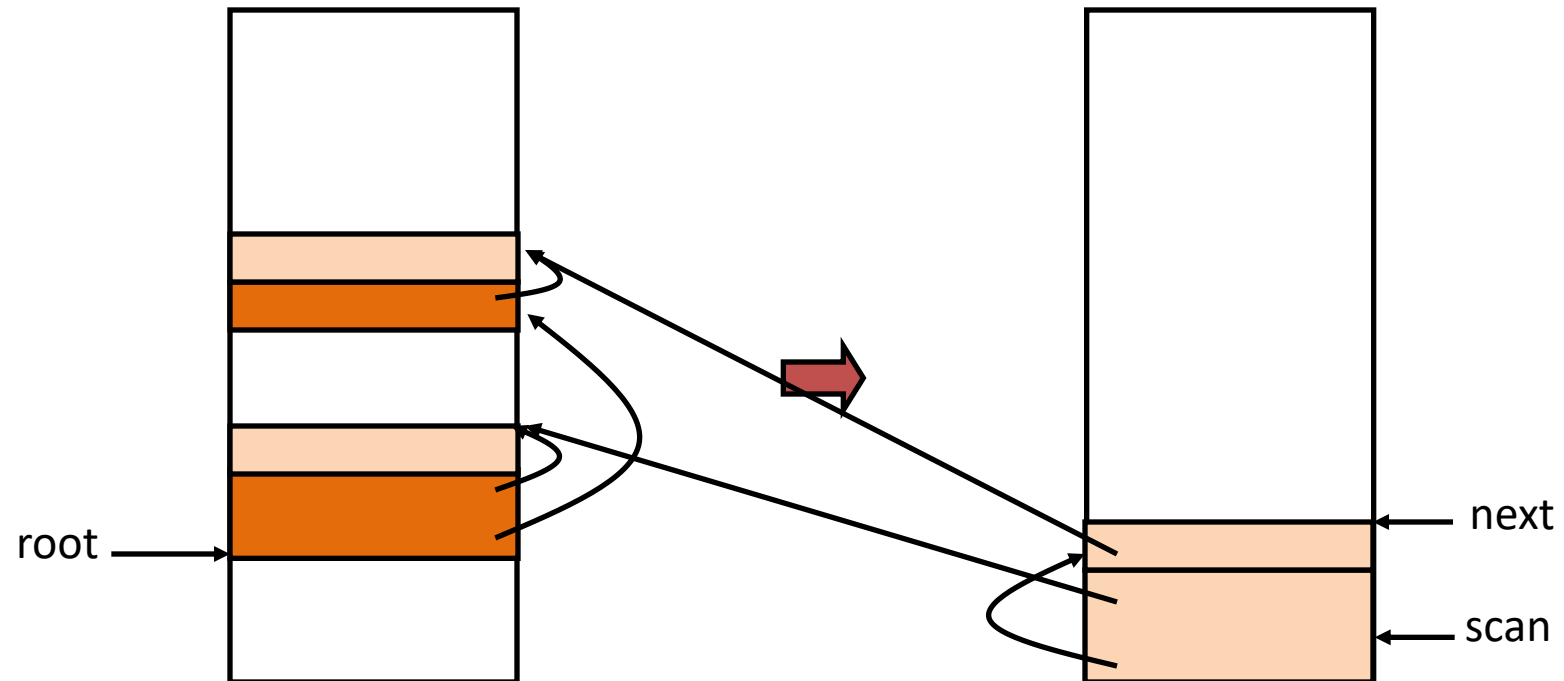
Mark & Copy: Zeitlupe



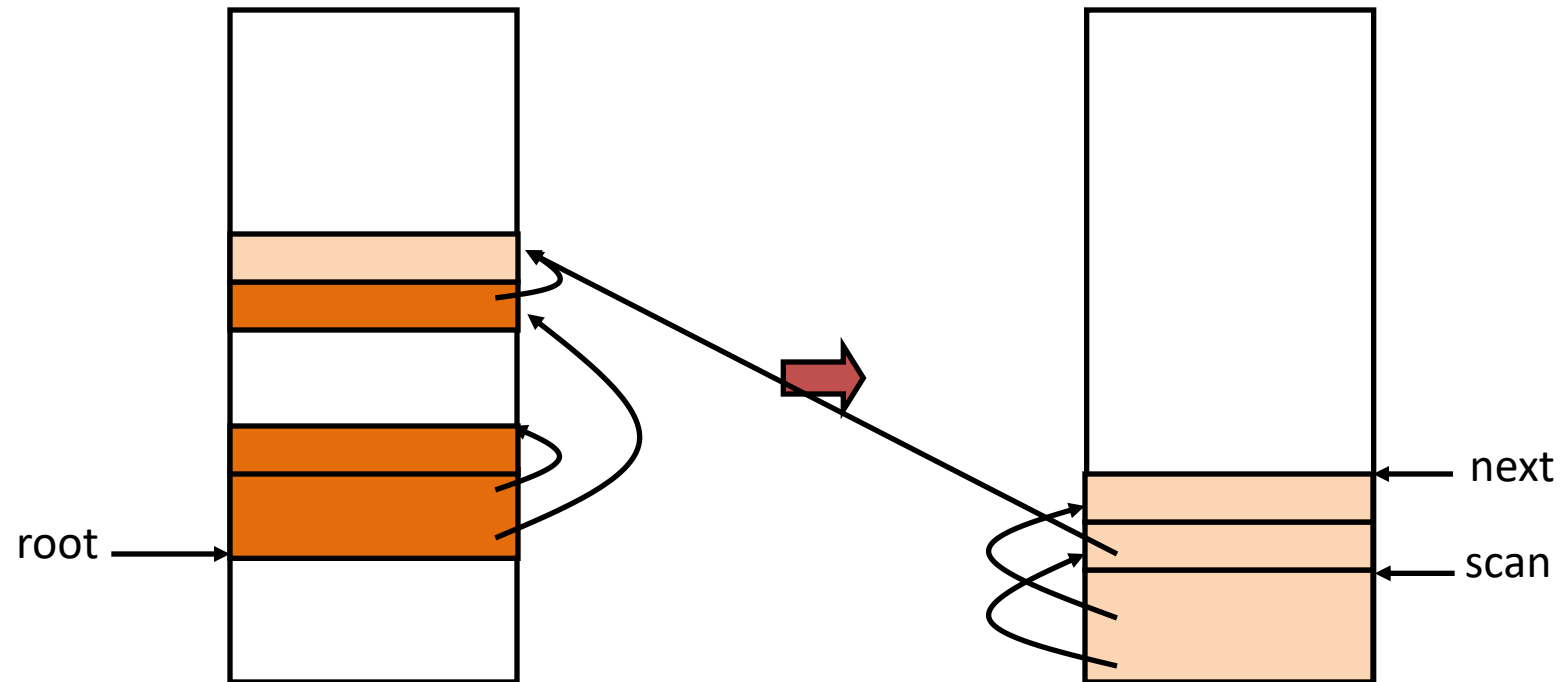
Mark & Copy: Zeitlupe



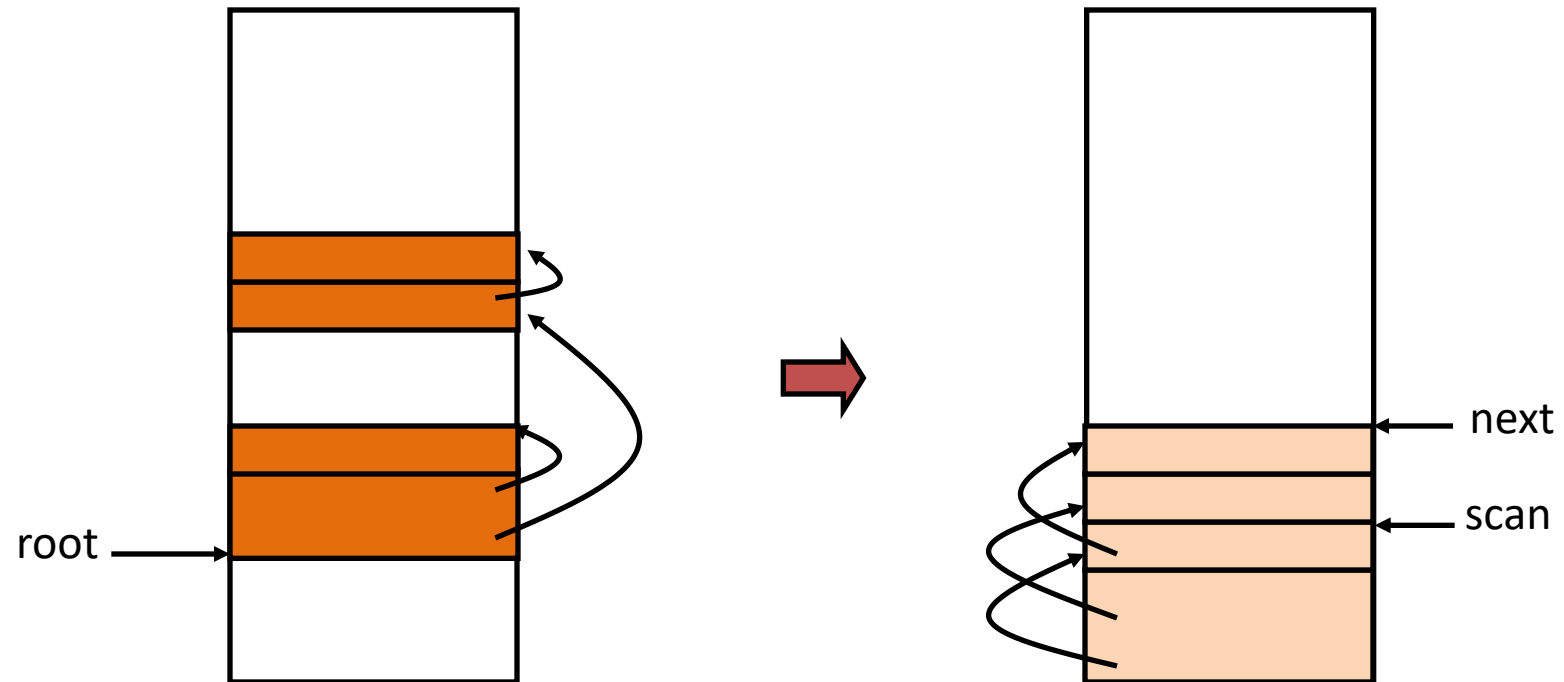
Mark & Copy: Zeitlupe



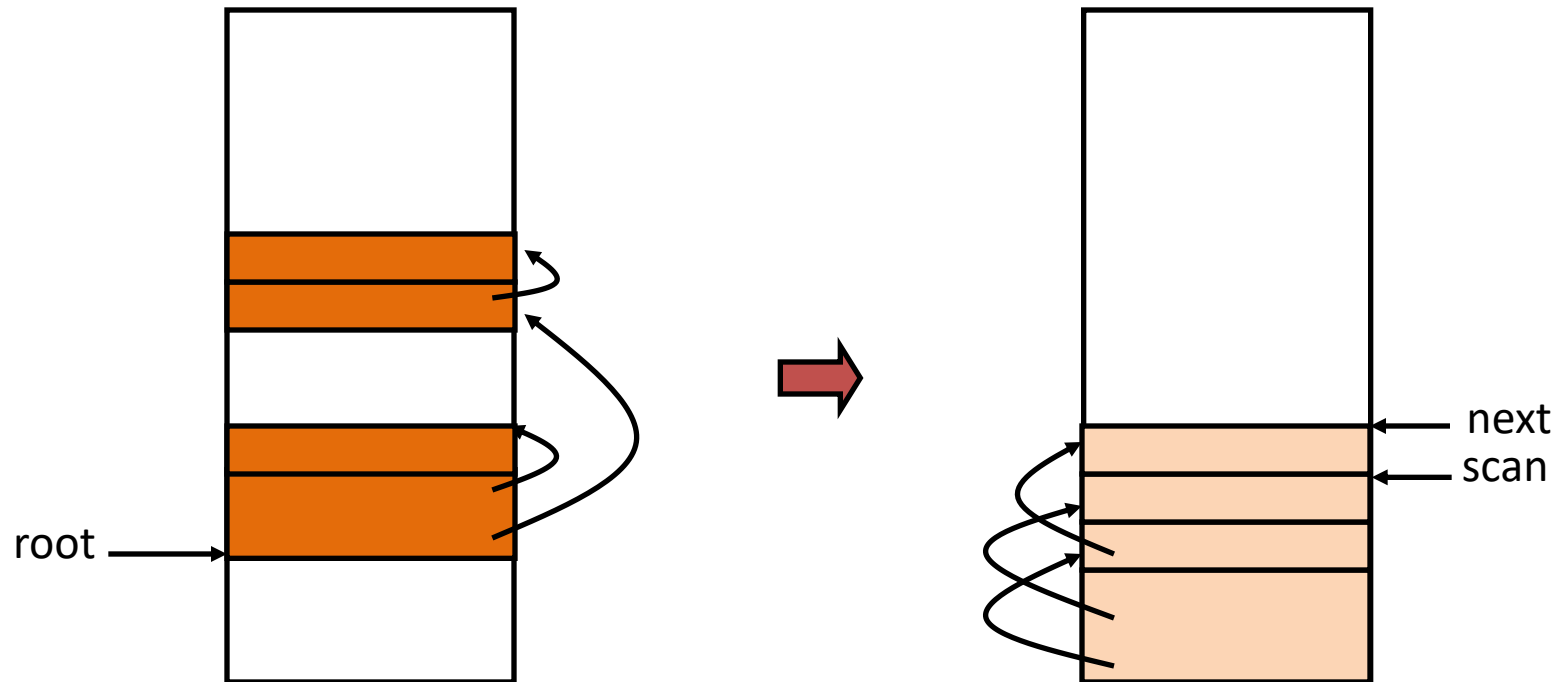
Mark & Copy: Zeitlupe



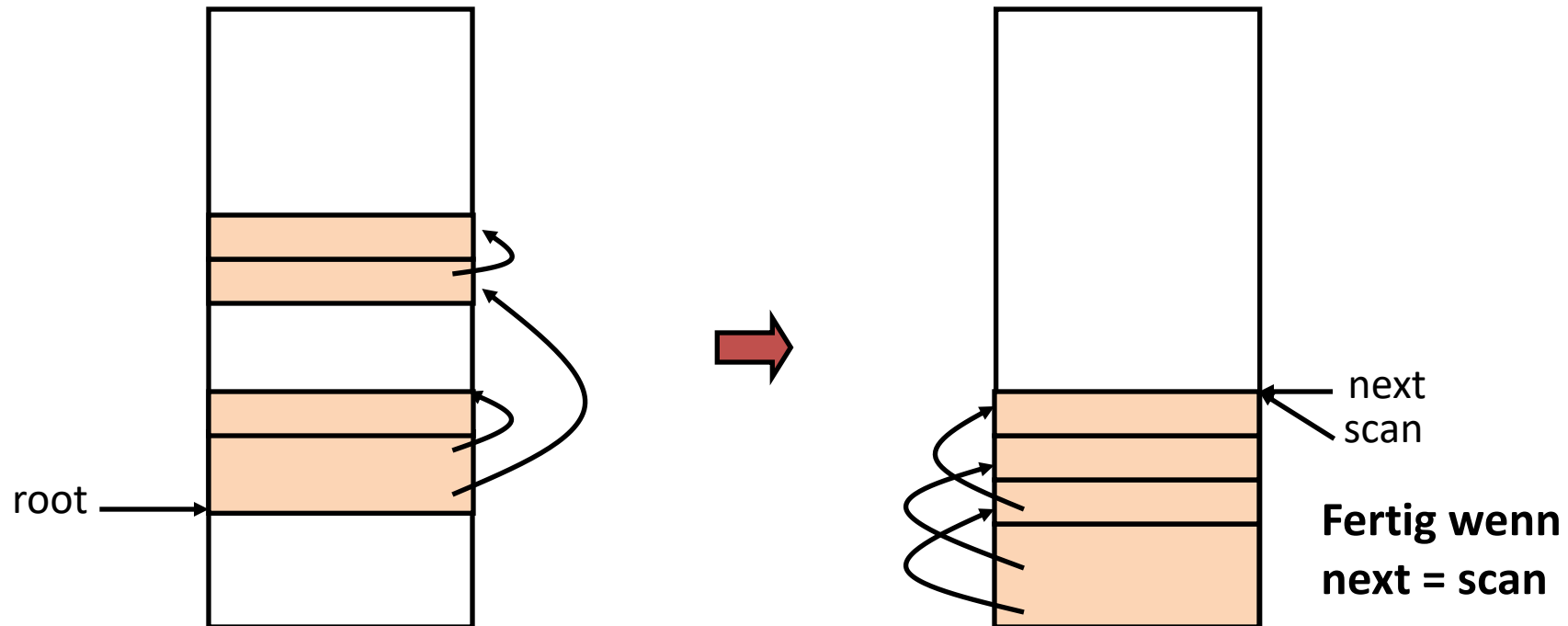
Mark & Copy: Zeitlupe



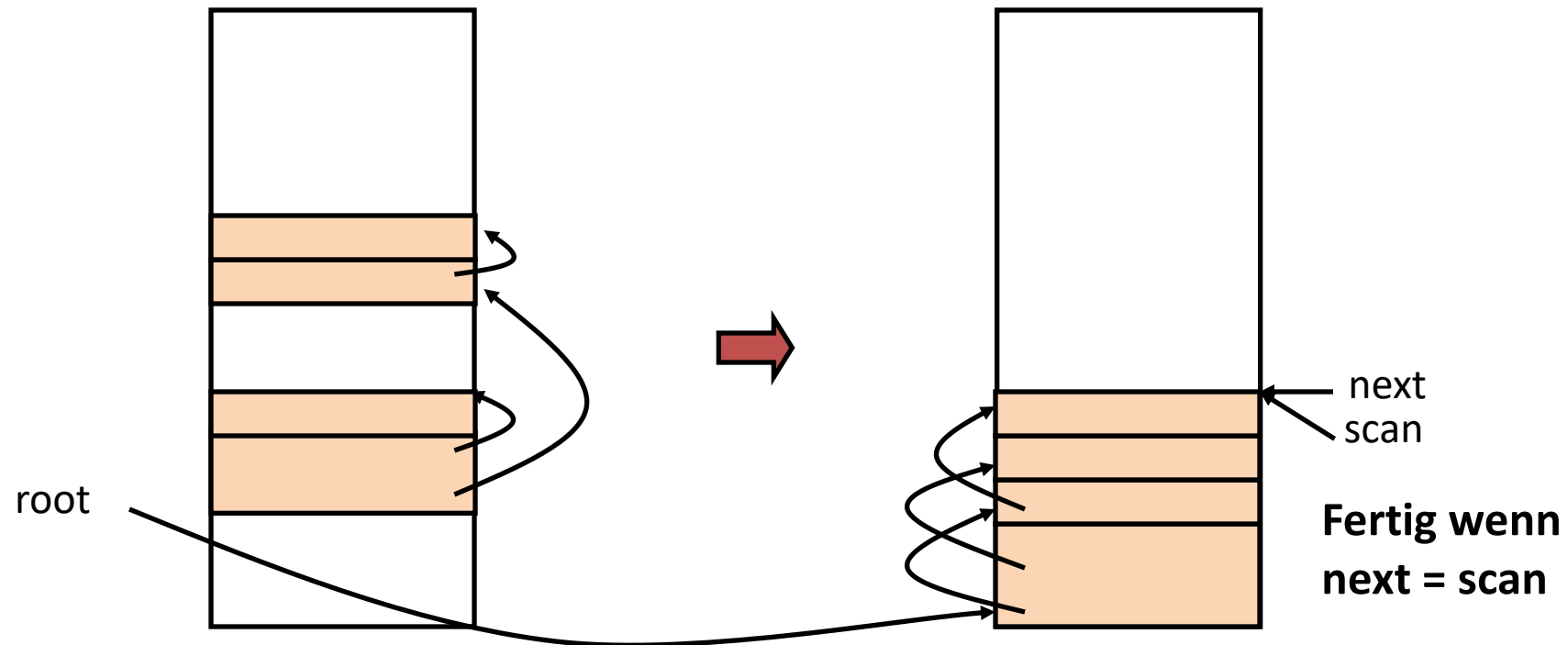
Mark & Copy: Zeitlupe



Mark & Copy: Zeitlupe



Mark & Copy: Zeitlupe



Bemerkungen (Mark & Copy)

Vorteile

- Einfach
- Eliminiert Fragmentierung
- Laufzeit proportional zur Anzahl erreichbaren Objekten
- Schnelle Allokation: Pointer wird mit Objektgröße inkrementiert

Nachteile

- Zusätzlicher Speicherplatz nötig
- Programm muss während des GCs gestoppt werden

Mark-Compact GC

Ähnlich wie Mark & Sweep

Unterschied: Nach der Mark Phase werden Objekte zum Anfang des Heaps umgelegt

Vorteile

- Keine Fragmentierung
- In-situ: Kein zweiter Heap nötig


Nachteile

- Zusätzliche Traversierung des Heaps nötig (3 Traversierungen insgesamt)
- Programm muss während des GCs gestoppt werden

Indiviualarbeit (siehe Übungsblatt)

Vergleich Speicherverwaltungsmethoden

Relevante Kriterien?



Algorithmus			
Reference Counting			
Mark & Sweep GC			
Mark & Copy GC			
Mark & Compact GC			

Diskussion: Wann kann GC passieren?

Beispielprogramm

```
class Foo {  
    public static void main(String args[]) {  
        Object v;  
        v = new Object();  
        System.out.println(v);  
        System.gc();  
    }  
}
```

Diskussion: Wann kann GC passieren?

Beispielprogramm

```
class Foo {  
    public static void main(String args[]) {  
        Object v;  
        Heap vorher →  
        v = new Object();  
        Heap nachher →  
        System.out.println(v);  
        System.gc();  
    }  
}
```

