Westfälische Wilhelms-Universität Münster
Fachbereich 10 – Mathematik und Informatik
Institut für Informatik

Masterarbeit

Garbage-Collection-Algorithmen und ihre Simulation

Phil Steinhorst

Erstgutachter und Betreuer Prof. Dr. Jan Vahrenhold

Zweitgutachter Prof. Dr. Markus Müller-Olm

Münster, 27. Februar 2019

Garbage-Collection-Algorithmen und ihre Simulation
Masterarbeit zur Erlangung des akademischen Grades *Master of Education*
in den Fächern Mathematik und Informatik
Erstgutachter und Betreuer: Prof. Dr. Jan Vahrenhold
Zweitgutachter: Prof. Dr. Markus Müller-Olm
Münster, 27. Februar 2019

Phil Steinhorst
Dürerstraße 1, 48147 Münster
p.st@wwu.de
Matrikelnummer: 382 837

Westfälische Wilhelms-Universität Münster
Fachbereich 10 – Mathematik und Informatik
Institut für Informatik
Einsteinstraße 62, 48149 Münster

Aus Gründen der besseren Lesbarkeit wird in dieser Arbeit auf die gleichzeitige Verwendung weiblicher und männlicher Sprachformen verzichtet und das generische Femininum verwendet. Sämtliche Personenbezeichnungen gelten gleichermaßen für alle Geschlechter.

Zusammenfassung

Die vorliegende Arbeit liefert eine Aufarbeitung verschiedener Ansätze für Garbage-Collection-Algorithmen. Nach einer kurzen Darstellung der zugrunde liegenden Problematik und deren praktischer Relevanz sowie der Vor- und Nachteilen einer automatischen Speicherverwaltung gegenüber einer manuellen Speicherverwaltung werden gängige Ansätze vergleichend vorgestellt. Hierbei erfolgt auch eine Beurteilung des Einsatzes und der Eignung in der Praxis. Als Gütekriterien dienen unter anderem Laufzeitbetrachtungen, Speicherbedarf und entstehende Verzögerungen im Programmablauf, die für ausgewählte Ansätze besonders detailliert untersucht werden.

Weiter wird eine Anwendung entworfen, mit der die Arbeitsweise der diskutierten Garbage-Collection-Ansätze visualisiert werden kann. Dies umfasst eine angemessene Visualisierung eines beschränkten Speicherbereichs durch eine optische Unterscheidbarkeit belegter Blöcke sowie der einzelnen Arbeitsphasen, die eine Garbage Collection ausführt. Dabei sind auch unterschiedliche Szenarien auswählbar, etwa verschiedene Speicherfüllstände und eine variable Anzahl bzw. Größe von Objekten, die im Speicher hinterlegt sind. Nennenswerte Auszüge aus dem Entwurfs- und Implementationsprozess werden detailliert hervorgehoben und erläutert.

Inhaltsverzeichnis

1 Einleitung

Die Möglichkeiten einer dynamischen Speicherverwaltung sind in den meisten modernen und höheren Programmiersprachen ein unverzichtbarer Bestandteil. Die Vorteile, einen Teil des dynamischen Speichers, der oft auch als *Heap* bezeichnet wird, zur Laufzeit eines Programms anfordern zu können, sind unbestreitbar: Speicherbereiche des Heaps dienen für Unterprogramme als Ablagemöglichkeit jenseits ihrer eigenen *Stacks*, sodass abgelegte Inhalte nach Terminierung erhalten und für weitere Unterprogramme zugänglich bleiben. Die Größe des angeforderten Speichers muss dabei nicht zur Übersetzungszeit bekannt sein, was die Realisierung dynamischer Datenstrukturen ermöglicht und die Gefahr zu klein oder zu groß gewählter fester Speicherbereiche reduziert.

Für die konkrete Verwendung einer dynamischen Speicherverwaltung sind grundsätzlich zwei diametrale Ansätze denkbar: Zum einen kann die Verantwortung für den korrekten Umgang mit dynamisch angefordertem Speicher gänzlich der Entwicklerin übertragen werden. Dies ist in der Regel mit zusätzlichem Aufwand verbunden (vgl. [Wil92, S. 1f]): Speicheradressen müssen manuell verwaltet werden, Anweisungen zur Anforderung und Freigabe von Speicher müssen in den eigentlichen Code integriert werden und entsprechende Ausnahmefälle bei Fehlschlägen müssen ordnungsgemäß abgefangen werden. Neben einer komplexer werdenden Codestruktur führt dies zu weiteren Fehlerquellen: Die Freigabe noch benötigten Speichers führt zu sogenannten *hängenden Zeigern* (engl. *dangling pointers*) – Referenzen, die *ins Leere zeigen* und in der Folge bestenfalls zu Programmabstürzen, schlimmstenfalls aber zu unerwartetem Verhalten und Datenverlust führen können. Nicht freigegebener, aber nicht mehr benötigter Speicher kann wiederum zu *Speicherlecks* (engl. *memory leaks*) und – bei hinreichend langer Laufzeit des Programms – zu einer Ausschöpfung des Speichers führen. *Double frees*, bei denen Speicherbereiche doppelt freigegeben werden, sind eine weitere Ursache für unerwünschtes Programmverhalten. Während die Anforderung von Speicher in der Regel unproblematisch ist, ist die Frage, wann und an welcher Stelle angeforderter Speicher wieder freigegeben werden kann, deutlich komplizierter, und fehlerhafte Verwendungen werden gegebenenfalls erst bei langfristiger Ausführung des Programms bemerkt.

Zum anderen existiert zur Vermeidung eben jener Schwierigkeiten der Ansatz, dem Compiler und der Laufzeitumgebung die adäquate Freigabe nicht mehr benötigten

Speichers zu überlassen. Zuständig hierfür ist dann ein Mechanismus, der gemeinhin als **Garbage Collection** (dt. *Abfallentsorgung*) bezeichnet wird. Eine Garbage Collection führt automatisch zu bestimmten Zeitpunkten – etwa regelmäßig oder wenn akuter Speichermangel besteht – eine Bereinigung des Speichers durch und gibt nicht mehr benötigte Speicherbereiche frei, ohne dass die Entwicklerin entsprechende Routinen in ihr Programm integrieren muss. Nichtsdestoweniger wird dieser Komfort nicht ohne Nachteile erworben: Wie jede Programmanweisung besitzt auch eine Garbage Collection einen gewissen Bedarf an Rechenzeit und Ressourcen, der sich negativ auf die Performance der eigentlichen Anwendung auswirken kann. Vor allem in Anwendungen, die einen hohen Durchsatz erreichen sollen oder in denen Deadlines und maximale Laufzeiten (engl. *worst case execution times*) um jeden Preis eingehalten werden müssen, spielt die Auswahl eines geeigneten Garbage-Collection-Algorithmus eine signifikante Rolle.

In dieser Arbeit werden gängige Ansätze zur Garbage Collection vorgestellt und miteinander verglichen. Dabei werden zunächst drei Klassen von Algorithmen fokussiert, die als Grundlage aller fortgeschrittenen Konzepte betrachtet werden können: markierende Algorithmen, referenzzählende Algorithmen und objektkopierende Algorithmen zur Kompaktierung des Heaps (vgl. [BCM04, S. 25]). Dazu wird aus jeder Klasse ein elementarer Algorithmus betrachtet, an dem das Grundprinzip der zugehörigen Algorithmen nachvollzogen werden kann. Durch einen Blick auf Performance und Ressourcenbedarf im Kontext verschiedener Anwendungsfälle ergeben sich dann Ansätze, die zu einer Auseinandersetzung mit elaborierteren Varianten motivieren. Im zweiten Teil der Arbeit wird der Entwurf und die Implementation einer Anwendung beschrieben, die die diskutierten Garbage-Collection-Algorithmen grafisch visualisiert und in einem vereinfachten Speichermodell simuliert. Anhand dieser Anwendung wird die Arbeitsweise der Algorithmen zusätzlich veranschaulicht, sodass ihre charakteristischen Merkmale erkennbar sind.

1.1 Terminologie

Bevor genauer darauf eingangen wird, was man unter einer Garbage Collection konkret versteht, erfolgt zunächst eine Einführung der nötigen Terminologie sowie eines Speichermodells, das im Fortgang dieser Arbeit benutzt wird. Dieses Speichermodell ist bewusst abstrakt gehalten, sodass es möglichst allgemeine Betrachtungen losgelöst von konkreten Programmiersprachen, Laufzeitumgebungen und Betriebssystemen ermöglicht, auch wenn an einigen Stellen exemplarisch Bezüge zu diesen hergestellt werden. Die eingeführten Begrifflichkeiten orientieren sich stark an der Terminologie aus der Fachliteratur (vgl. [JL96, Kap. 1]).

Objekt

Der Begriff **Objekt** bezeichnet stets eine konkrete Instanz eines definierten Datentyps, beispielsweise eines `struct` in C oder einer Java-Klasse. Ein Objekt besitzt eine festgelegte Anzahl von **Feldern**, die jeweils einen Wert eines festgelegten Datentyps – etwa ein Integer oder eine Referenz auf ein anderes Objekt im hier definierten Sinne – enthalten. Der in dieser Arbeit verwendete Objektbegriff ist wesentlich allgemeiner gehalten als in der Objektorientierung üblich: Auch einzelne Werte eines Basisdatentyps oder Arrays werden als Objekt aufgefasst, selbst wenn diese nicht Bestandteil eines im Programm definierten Datentyps sind.

Ferner wird vorausgesetzt, dass Objekte und ihre Felder *typisiert* sind. Das bedeutet, dass stets nachvollziehbar ist, aus welchen Feldern ein Objekt besteht und von welchem Datentyp diese sind. Insbesondere ist unterscheidbar, ob ein Feld eines Objekts eine Referenz enthält oder nicht. Weiter wird angenommen, dass jedes Objekt einen sogenannten *Header* besitzt. Dies sind separate Felder, die Metainformationen aufnehmen, die für den Compiler und die Laufzeitumgebung, nicht aber für die Entwicklerin, zugänglich sind. Diverse vorgestellte Algorithmen werden diesen Bereich nutzen, um für die Speicherverwaltung relevante Informationen zu hinterlegen.[1]

Heap

Der **Heap** ist derjenige Speicherbereich, in dem zur Laufzeit eines Programms Objekte in beliebiger Reihenfolge erzeugt und freigegeben werden können. Der Heap besteht aus *Wörtern* einer festen Größe, auf die über eine Speicheradresse zugegriffen werden kann; ein *Wort* ist dabei die kleinste zuweisbare Speichermenge und kann die Zustände *belegt* (bzw. *zugewiesen*) oder *frei* annehmen. Sofern nichts anderes vereinbart ist, wird davon ausgegangen, dass der Heap ein zusammenhängender linearer Speicherbereich ist (siehe Abbildung 1.1).[2]

Der Zugriff auf das `i`-te Feld eines Objekts `a` wird – analog zur Syntax der Programmierspache C – mit `a[i]` notiert. Ebenso bezeichnet `&a` die Adresse eines Objekts und `*p` die Dereferenzierung eines Zeigers `p`. Mit POINTERS(`a`) wird zudem die

[1] Technisch kann dies etwa dadurch realisiert werden, dass die angeforderte Speichermenge für ein Objekt vergrößert wird, sodass die Informationen des Headers am Anfang des Objekts aufgenommen werden können. Als Ergebnis der Speicheranforderung wird dann die Speicheradresse zurückgegeben, die auf den Bereich hinter dem Header verweist.

[2] Tatsächlich ist dies eine starke Vereinfachung. In der Praxis ist der Bereich des physikalischen Speichers, der von einer Anwendung verwendet wird, häufig fragmentiert und inhomogen. Die Speicherverwaltung eines Betriebssystems bildet diesen Bereich auf einen *virtuellen Speicher* ab, der der Anwendung zur Verfügung gestellt wird und aus ihrer Sicht linear zusammenhängend ist. Für einen Überblick hierzu siehe etwa [TB14, Kap. 3.3].

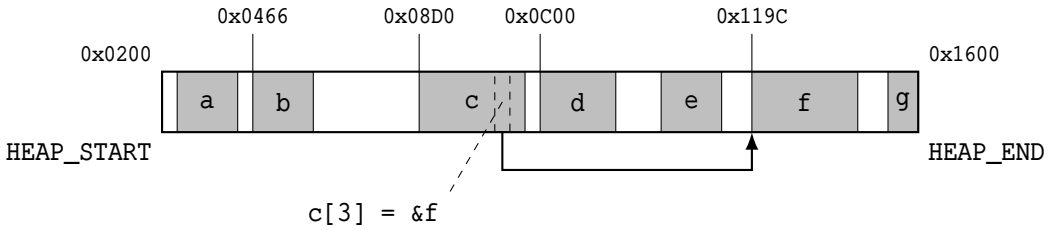


Abbildung 1.1.: Beispiel eines Heaps, der mit einigen Objekten gefüllt ist. Die Position eines Objekts ist durch die Adresse des ersten Worts gegeben, das von dem Objekt belegt wird.

Menge aller von `a` ausgehenden Referenzen auf andere Objekte des Heaps und mit POINTERFIELDS(`a`) die Menge aller Felder von `a`, die eine solche Referenz enthalten, bezeichnet. Ist also `x` $\in$ POINTERFIELDS(`a`), so bezeichnet `*x` entsprechend die im Feld gespeicherte Referenz und `**x` das Ziel dieser Referenz. In den Algorithmen wird in der Regel auf eine Überprüfung von Referenzen auf `null` verzichtet, bevor sie dereferenziert werden.[3]

Allokator, Mutator und Kollektor

Aufgabe des **Allokators**, der zur Laufzeitumgebung eines Programms gehört, ist zum einen die Zuweisung von Heapspeicher bei dynamischer Instanziierung eines neuen Objekts und zum anderen die Freigabe von Objekten. Der Allokator führt somit Buch über die belegten und freien Wörter des Heaps. Die genaue Realisierung dieser Mechanismen werden in dieser Arbeit weitestgehend außen vor gelassen, jedoch wird in gewissen Situationen das Vorhandensein bestimmter Funktionalitäten vorausgesetzt. Beispielsweise wird verlangt, dass eine Prozedur *new* zur Verfügung steht, die bei der Erzeugung eines neuen Objekts Speicher reserviert und die entsprechende Speicheradresse zurückgibt. Die Funktionsweise von *new* kann dabei vom verwendeten Garbage-Collection-Algorithmus abhängen. In Algorithmus 1.1 ist etwa eine naive Implementation zu sehen.

Nach DIJKSTRA et al. besteht die Laufzeitumgebung eines Programms zudem aus zwei funktional unterscheidbaren Bestandteilen [Dij+78, S. 967]: Der **Mutator** ist derjenige Thread (bzw. eine Menge von Threads), die den eigentlichen Programmcode ausführen. Bedeutsam sind dabei vor allem Programmanweisungen, die in Feldern von Objekten vorhandene Referenzen manipulieren und somit ursächlich

[3] In vielen Algorithmen, die in dieser Arbeit vorgestellt werden, werden Objekte als Parameter an Hilfsprozeduren übergeben. Ob es an der Stelle sinnvoll ist, Objekte *by value* oder *by reference* zu übergeben, ist ein Implementationsdetail, das die Entwicklerin im Anbetracht der verwendeten Programmiersprache festlegt. In dieser Arbeit soll die Übergabe eines Objekts die Intention erzeugen, dass die im Objekt gespeicherten Informationen relevant sind, während die Übergabe einer Objektadresse dessen Position im Speicher in den Fokus rückt. Ähnliches gilt für Mengen von Objekten, die in manchen Algorithmen vorkommen.

Algorithmus 1.1 Prozedur *new* zur Erzeugung eines neuen Objekts. Die Garbage Collection wird hier bei Bedarf ausgelöst, wenn nicht genügend freier Speicher verfügbar ist.

```
1: new():
2:     adr ← allocate()                          ▷ Versuche Zuweisung von Speicher
3:     if adr = null                             ▷ Nicht genügend freier Speicher
4:         collect()                             ▷ Aufruf der Garbage Collection
5:         adr ← allocate()                      ▷ Neuer Versuch
6:         if adr = null
7:             error("Nicht genügend Speicher")
8:     return adr
```

für die Entstehung nicht mehr benötigter Objekte sind. Im Gegensatz dazu ist die Aufgabe des **Kollektors**, die nicht mehr benötigten Objekte zu identifizieren und ihre Freigabe zu veranlassen. Der Kollektor ist demnach derjenige Thread (bzw. eine Menge von Threads), die einen Garbage-Collection-Algorithmus ausführen.

1.2 Problemstellung

Nachdem die nötigen Grundbegriffe eingeführt wurden, kann nun definiert werden, was unter einer Garbage Collection verstanden wird. Anschließend folgt eine Spezifikation von Eigenschaften, die im Kontext dieser Arbeit von einem Garbage-Collection-Algorithmus gefordert werden.

Definition 1.1 (Lebendigkeit)**:**
Ein Objekt heißt zu einem bestimmten Zeitpunkt im Programmablauf **lebendig**, wenn der Mutator im weiteren Programmablauf lesend oder schreibend auf dieses zugreift. Andernfalls wird das Objekt als **nicht mehr benötigt** bezeichnet.

Definition 1.2 (Garbage Collection)**:**
Eine **Garbage Collection** ist ein Algorithmus zur automatischen Wiederverwendung bereits genutzten Heapspeichers durch Identifikation und Freigabe von Objekten, die im weiteren Programmverlauf nicht mehr benötigt werden.

Sobald der Mutator auf eine Objektinstanz im weiteren Programmverlauf nicht mehr zugreift – weder lesend, noch schreibend – ist ein Überschreiben des Objekts unproblematisch. Demzufolge darf eine Garbage Collection die Freigabe des entsprechenden Speicherbereichs veranlassen, sobald eine Stelle im Programmcode erreicht wurde, ab der der Bezeichner eines Objekts (bzw. eine Referenz auf dieses Objekt) nicht mehr verwendet wird – auch, wenn theoretisch noch darauf zugegriffen werden könnte. Allerdings ist die Frage, ob dies der Fall ist oder nicht, nicht beantwortbar:

Satz 1.1 (Unentscheidbarkeit von Lebendigkeit)**:**
Es existiert kein Algorithmus, der die Lebendigkeit von Objekten entscheidet.

Beweisskizze: Dies ist ein Korollar aus der Unentscheidbarkeit des Halteproblems: Angenommen, es gäbe einen Algorithmus, der für ein beliebiges Programm entscheidet, ob Objekte zu einem bestimmten Zeitpunkt lebendig sind. Dieser müsste insbesondere entscheiden, dass der Teil eines Programms, in dem ein Objekt lebendig ist, terminiert. Ein solcher Algorithmus existiert jedoch im Allgemeinen nicht (vgl. [Sip13, Kap. 4.2]). □

Aus diesem Grund wird nachfolgend eine schwächere Eigenschaft von Objekten betrachtet: die Erreichbarkeit über Referenzen. Dafür wird von einer Menge ROOTS von **Basisobjekten** (engl. *root objects*) ausgegangen, die nicht zum Heap gehören. Diese sind dadurch gekennzeichnet, dass der Mutator unmittelbaren Zugriff auf sie hat, ohne dafür zunächst ihre Adresse aus den Feldern anderer Objekte beschaffen zu müssen. Hierzu zählen zum Beispiel statische Objekte, deren Positionen im Speicher bereits zur Übersetzungszeit bekannt sind, oder Objekte, die sich in Aufrufstapeln von Unterprogrammen befinden. Basisobjekte selbst werden durch die Garbage Collection nicht verwaltet, allerdings werden ihre Header von manchen Algorithmen dennoch zur Speicherung von Informationen genutzt. Alle weiteren Objekte, die zur Laufzeit dynamisch erzeugt werden, gelten als erreichbar, wenn auf sie über eine Folge von Referenzen zugegriffen werden kann, wobei diese in den Feldern von Objekten gespeichert sind und die erste Referenz von einem Basisobjekt ausgeht. Einfacher ausgedrückt: Ein Objekt ist erreichbar, wenn der Mutator die Möglichkeit hat, mittelbar oder unmittelbar über Referenzen auf das Objekt zugreifen zu können. Formal wird diese Eigenschaft wie folgt definiert:

Definition 1.3 (Erreichbarkeit)**:**
Jedes Element der Menge $\mathcal{R}$ der **erreichbaren Objekte** ist durch endlich häufige Anwendung der folgenden beiden Regeln konstruiert:

(1) Ist $\texttt{a} \in$ ROOTS, so folgt $\texttt{a} \in \mathcal{R}$.

(2) Ist $\texttt{a} \in \mathcal{R}$, b ein weiteres Objekt und existiert ein $\texttt{i} \in \mathbb{N}_0$ mit $\texttt{*a[i]} = \texttt{b}$, so folgt $\texttt{b} \in \mathcal{R}$.

Gilt $\texttt{*a[i]} = \texttt{b}$, so schreibt man auch $\texttt{a} \rightarrow \texttt{b}$. Existiert für zwei Objekte $\texttt{a}, \texttt{b}$ eine endliche Folge von Objekten $\texttt{a}_1, \ldots, \texttt{a}_n$ mit $\texttt{a} \rightarrow \texttt{a}_1 \rightarrow \cdots \rightarrow \texttt{a}_n \rightarrow \texttt{b}$, so wird dies zudem mit $\texttt{a} \xrightarrow{*} \texttt{b}$ notiert.

Diese Definition garantiert zwar nicht, dass jedes erreichbare Objekt auch lebendig ist. Davon ausgehend, dass unerreichbare Objekte auch nicht *wiedergefunden* werden

können und verwaist bleiben, lässt sich jedoch mit Sicherheit sagen, dass unerreichbare Objekte nicht mehr verwendet werden und gefahrlos durch den Kollektor freigegeben werden dürfen.

Die Erreichbarkeit von Objekten lässt sich mithilfe eines sogenannten **Objektgraphen** visualisieren. Jedes existierende Objekt korrespondiert dabei mit einem Knoten des Graphen, sodass das Objekt mit seinem Knoten identifiziert werden kann (und umgekehrt). Besitzt ein Objekt `a` in mindestens einem seiner Felder eine Referenz auf ein weiteres Objekt `b`, so wird dies durch eine gerichtete Kante zwischen den entsprechenden Knoten dargestellt. Ein Objekt ist somit nicht erreichbar, wenn es im Objektgraphen keinen Pfad zu ihm gibt, der in einem Basisobjekt startet. Objektgraphen werden im Rahmen dieser Arbeit bei der Veranschaulichung der vorgestellten Algorithmen dienlich sein.

Definition 1.4 (Objektgraph)**:**
Sei O eine Menge von Objekten. Ein gerichteter Graph $G = (V, E)$ mit Knotenmenge V und Kantenmenge $E \subseteq V \times V$ heißt **Objektgraph** zu O, wenn eine bijektive Abbildung $\varphi\colon O \to V$ existiert, sodass für je zwei Objekte $\mathtt{a}, \mathtt{b} \in O$ gilt:

$$\mathtt{a} \to \mathtt{b} \quad \Leftrightarrow \quad (\varphi(\mathtt{a}), \varphi(\mathtt{b})) \in E.$$

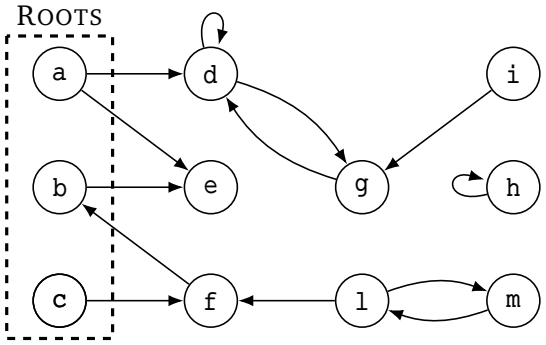


Abbildung 1.2.: Beispiel eines Objektgraphen. Die Objekte `a`, `b` und `c` sind Basisobjekte. Die Objekte `h`, `i`, `l` und `m` sind in dieser Konstellation nicht erreichbar.

An dieser Stelle folgt die Formulierung eines Korrektheitskriteriums für Garbage-Collection-Algorithmen. Dieses besteht aus der Anforderung, dass keine noch benötigten Daten zerstört werden.

Definition 1.5 (Korrektheit von Garbage-Collection-Algorithmen)**:**
Ein Garbage-Collection-Algorithmus ist **korrekt**, wenn er keine lebendigen Objekte freigibt.

Gemäß Definition 1.3 ist es folglich hinreichend zu zeigen, dass nur nicht erreichbare Objekte freigegeben werden, um Korrektheit nachzuweisen.

An dieser Stelle ist erwähnenswert, warum nicht vorausgesetzt wird, dass die Ausführung eines Garbage-Collection-Algorithmus die Freigabe *sämtlicher* nicht erreichbaren Objekte anfordert. Tatsächlich wird zu sehen sein, dass es aus Performancegründen vorteilhaft sein kann, nur einen Teil des nicht mehr benötigten zugewiesenen Speichers zu bereinigen, um längere Wartezeiten zu vermeiden. Ein solches Kriterium wäre daher zu restriktiv.

Teil I

Algorithmen und Ansätze

2 Mark-Sweep-Algorithmen

Dieses Kapitel beginnt mit einer Vorstellung des ersten Garbage-Collection-Algorithmus, der auf McCarthy zurückgeht [McC60, S. 191–193]. Im Rahmen eines im Jahr 1960 veröffentlichten Artikels über die Berechnung rekursiver Funktionen auf dem *IBM 704* mithilfe des *LISP Programming Systems* erläutert McCarthy die Speicherung von Daten in einer Listenstruktur. Diese besteht aus Paaren, deren erster Eintrag `car` die zu speichernde Information enthält[1], während im zweiten Eintrag `cdr` die Registeradresse des nachfolgenden Paares zu finden ist (siehe Abbildung 2.1). Diese Struktur hat den Vorteil, dass ein in mehreren Listen vorkommendes Datum nur ein einziges Register belegt. Register, die aktuell nicht zur Speicherung von Daten genutzt werden, befinden sich in einer *free storage list*. Bei der Anforderung von Speicher für ein zu speicherndes Datum werden Register aus dieser Liste entfernt. Durch die Manipulation der Registeradressen können Paare verwaisen, was zu Speicherlecks führt. Zur Auflösung dieser Problematik bietet LISP als erste Programmiersprache ihrer Zeit eine automatische Speicherverwaltung, die von McCarthy wie folgt grob umschrieben wird: Im Falle von Speicherknappheit wird – ausgehend von einer Menge von Basisregistern – ermittelt, welche Register über eine Folge von `cdr`-Einträgen erreichbar sind. Nicht erreichbare Register enthalten überschreibbare Inhalte, sodass diese zurück in die *free storage list* eingefügt werden können und wieder als freie Speicherplätze zur Verfügung stehen. Diese zweischrittige Vorgehensweise – das Erkennen nicht mehr benötigter Speicherbereiche und die anschließende Freigabe eben jener – bildet die Grundlage des **Mark-Sweep-Algorithmus**.

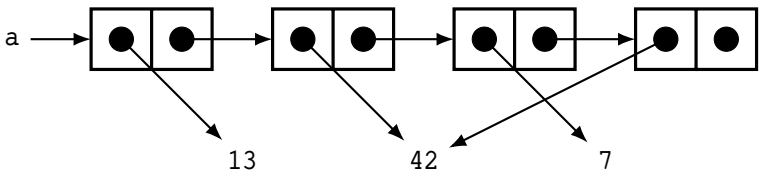


Abbildung 2.1.: Visualisierung der LISP-Liste `a = (13,42,7,42)` als *Box-and-Pointer*-Diagramm (vgl. [ASS96, Kapitel 3.3]).

2.1 Naiver Mark-Sweep-Algorithmus

Der **naive Mark-Sweep-Algorithmus** arbeitet in zwei Schritten: Zunächst wird bestimmt, welche Objekte im Speicher unerreichbar sind, weil sie von keinem anderen

[1] Genauer: Die Adresse des Registers, in dem die zu speichernde Information gelagert wird.

erreichbaren Objekt referenziert werden. Diese Objekte können gefahrlos freigegeben werden, da auf ihre Informationen nicht mehr zugegriffen werden kann. Der zweite Schritt besteht aus einer Traversierung des gesamten Heaps. Dabei werden alle existierenden Objekte besucht und diejenigen freigegeben, die im ersten Schritt als unerreichbar identifiziert werden konnten. Die entsprechenden Speicherbereiche stehen anschließend wieder für neue Objekte zur Verfügung.

Algorithmus 2.1 Naives Mark and Sweep – Markierungsphase (vgl. [JL96, Kap. 2.2])

```
 1: collect():
 2:     markStart()
 3:     sweep()

 4: markStart():
 5:     toDo ← ∅                                  ▷ Noch abzuarbeitende Objekte
 6:     for each obj ∈ ROOTS                      ▷ Beginne mit Basisobjekten
 7:         if not isMarked(obj)
 8:             setMarked(obj)                    ▷ Objekt als erreichbar markieren
 9:             add(toDo, obj)
10:             mark()                            ▷ Abarbeitung starten

11: mark():
12:     while toDo ≠ ∅
13:         obj ← remove(toDo)                    ▷ Hole nächstes Objekt
14:         for each ref ∈ POINTERS(obj)          ▷ Hole nächste Referenz auf Objekt
15:             if not isMarked(*ref)
16:                 setMarked(*ref)
17:                 add(toDo, *ref)
```

Die Markierungsphase (engl. *mark*) startet mit Auslösung der Garbage Collection durch den Allokator und funktioniert wie folgt: Ein Objekt zu markieren bedeutet, es als erreichbar zu kennzeichnen, indem in seinem Header ein entsprechender Wert – etwa ein Bit – gesetzt wird. Zunächst wird mittels der Prozedur *markStart* eine Menge `toDo` erzeugt, die diejenigen Objekte enthält, die bereits als erreichbar erkannt, aber selbst noch nicht verarbeitet wurden (Zeile 5 in Algorithmus 2.1). In diese werden alle bislang unmarkierten Basisobjekte der Menge ROOTS eingefügt und markiert, da sie in jedem Fall erreichbar sind (Zeile 6 bis 9). Ist ein Basisobjekt bereits markiert worden, so wurde es schon entdeckt – etwa, weil es durch ein zuvor abgearbeitetes Objekt referenziert wird. Daraus folgt, dass es ebenfalls bereits abgearbeitet wurde oder sich noch in der Menge `toDo` befindet. In beiden Fällen muss es folglich nicht erneut zu `toDo` hinzugefügt werden.

Bereits nach dem Hinzufügen des ersten Basisobjekts wird die Prozedur *mark* aufgerufen, welche die `toDo`-Menge abarbeitet. Für jedes Objekt in `toDo` werden diejenigen Felder betrachtet, die eine Referenz auf ein Objekt enthalten (Zeile 13 und 14). Wenn dieses Objekt noch nicht markiert wurde, wird es in diesem Augenblick zum ersten

Mal entdeckt. Da es somit erreichbar ist, kann es markiert und zu `toDo` hinzugefügt werden (Zeile 16 und 17), um zu einem späteren Zeitpunkt abgearbeitet zu werden. Verweist die Referenz hingegen auf ein Objekt, das bereits markiert wurde, wurde dieses schon zuvor entdeckt. Auch hier ist ein erneutes Hinzufügen zu `toDo` überflüssig. Sobald `toDo` leer ist, erfolgt die Rückkehr zur Prozedur *markStart*, sodass gegebenenfalls das nächste Basisobjekt abgearbeitet wird. Abbildung 2.2 zeigt die Arbeitsweise des Algorithmus an einem Beispiel.

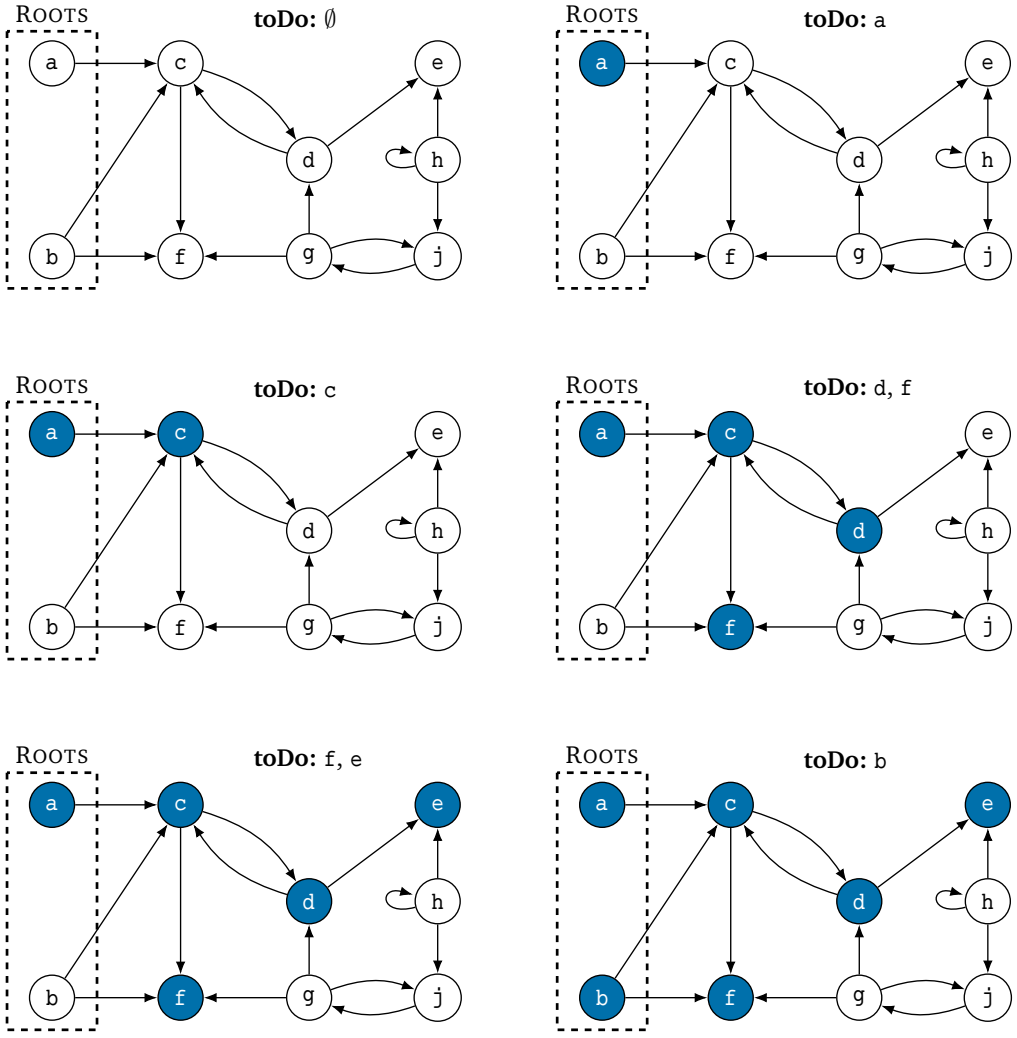


Abbildung 2.2.: Beispielhafte Ausführung von Algorithmus 2.1.

Es ist wesentlich, dass Objekte bereits markiert werden, wenn sie der Menge `toDo` hinzugefügt werden, und nicht etwa, nachdem sie abgearbeitet wurden (Zeile 8 und 9 bzw. 16 und 17). Andernfalls besteht bei zyklischen Referenzen die Gefahr einer Endlosschleife, da unmarkierte Objekte mehrfach hinzugefügt würden. Präziser lässt sich festhalten, dass `toDo` zu jedem Zeitpunkt ausschließlich bereits markierte Objekte enthält. Da keine Objekte hinzugefügt werden, die bereits markiert wurden (Zeile

7 und 15), wird kein Objekt mehrfach verarbeitet. Da zudem mit jeder Iteration der **while**-Schleife mindestens ein Objekt aus `toDo` entfernt wird (Zeile 13), die Anzahl aller Objekte endlich ist und vorausgesetzt wird, dass während der Ausführung des Kollektors keine neuen Objekte entstehen, wird sowohl die **while**-Schleife, als auch die **for each**-Schleife nach endlich vielen Schritten terminieren.

Algorithmus 2.2 Naives Mark and Sweep – Bereinigungsphase (vgl. [JL96, Kap. 2.2])

```
1:  sweep():
2:    pos ← nextObject(HEAP_START)          ▷ Beginne bei erstem Objekt im Heap
3:    while pos ≠ null
4:      if isMarked(*pos)
5:        unsetMarked(*pos)
6:      else free(pos)
7:      pos ← nextObject(pos)
8:    for each obj ∈ ROOTS                  ▷ Markierung der Basisobjekte zurücksetzen
9:      unsetMarked(obj)
```

Die Bereinigungsphase (engl. *sweep*) beginnt unmittelbar nach der Markierungsphase durch Aufruf der Prozedur *sweep*. Die Variable `pos` wird mit der Speicheradresse initialisiert, an der sich das erste Objekt im Heap befindet. Hier wird davon ausgegangen, dass eine Prozedur *nextObject* des Allokators zur Verfügung steht, die anhand einer übergebenen Speicheradresse die Adresse des nachfolgenden Objektes oder `null` zurückgibt, wenn dieses nicht existiert. Dadurch wird der Heap linear traversiert; nicht markierte Objekte werden freigegeben, während die Markierung erreichbarer Objekte zurückgesetzt wird. Damit auch die Markierung der nicht dem Heap zugehörigen Basisobjekte zurückgesetzt wird, erfolgt zuletzt eine Iteration über ROOTS (Zeile 8f).

Wir halten zunächst fest, dass der Mark-Sweep-Algorithmus in seiner Gänze terminiert und korrekt ist, sofern während der Garbage Collection das laufende Programm angehalten wird:

Satz 2.1 (Korrektheit des naiven Mark-Sweep-Algorithmus)**:**
Der Mark-Sweep-Algorithmus terminiert und ist korrekt, wenn der Mutator während der Arbeit des Kollektors angehalten wird.

Beweis: Wie oben erläutert, terminiert die Markierungsphase in jedem Fall, da bei angehaltenem Mutator keine neuen Objekte erstellt werden. Gleiches gilt für die Bereinigungsphase, in der alle Objekte des Heaps in endlicher Zeit besucht werden. Somit terminiert der gesamte Algorithmus.

Weiter werden lediglich nicht markierte Heapobjekte freigegeben (Zeile 4 bis 6 in Algorithmus 2.2). Bleibt ein Objekt `obj` unmarkiert, so ist `obj` kein Basisobjekt,

da *markStart* alle Basisobjekte markiert. Somit bleibt kein Basisobjekt unmarkiert. Wir zeigen, dass es nach der Markierungsphase zudem keine zwei Objekte `a`, `b` mit $\texttt{a} \rightarrow \texttt{b}$ gibt, sodass `a` markiert und `b` unmarkiert ist: Da `a` markiert ist, wurde `a` auch der Menge `toDo` hinzugefügt (Zeile 8f bzw. 16f in Algorithmus 2.1). Entsprechend gab es eine Iteration der **while**-Schleife mit $\texttt{obj} = \texttt{a}$. Gilt nun $\texttt{a} \rightarrow \texttt{b}$, so ist $\texttt{\&b} \in \textsc{Pointers}(\texttt{a})$. Folglich wird in einer Iteration der **for each**-Schleife mit $\texttt{ref} = \texttt{\&b}$ auch $\texttt{*(\&b)} = \texttt{b}$ markiert, falls `b` nicht schon zuvor markiert wurde. Es existiert somit keine Referenz von einem erreichbaren auf ein unmarkiertes Objekt, weswegen keine erreichbaren Objekte freigegeben werden. □

Die Bedingung, dass der Mutator während des Markierens pausiert wird, ist tatsächlich notwendig, um zu vermeiden, dass fälschlicherweise erreichbare Objekte entfernt werden, wie folgendes Beispiel zeigt (vgl. [Dij+78, S. 969]): Man betrachte etwa die Situation, dass zwei Basisobjekte `a` und `b` alternierend auf ein Objekt `c` verweisen, das ausschließlich über `a` oder `b` erreichbar ist. Während der Kollektor aktiv ist, führe der Mutator folgenden Code aus:

```
1: b.ref ← &c
2: a.ref ← null
3: a.ref ← &c
4: b.ref ← null
```

Es könnte passieren, dass der Kollektor gerade Objekt `a` abarbeitet, unmittelbar nachdem Zeile 2 ausgeführt wurde. Es wird dann keine Referenz auf Objekt `c` vorgefunden. Wenn der Kollektor nun Objekt `b` betrachtet, nachdem bereits Zeile 4 abgearbeitet wurde, wird Objekt `c` weiterhin nicht entdeckt. Insgesamt wird Objekt `c` somit nicht markiert, obwohl es über `a` erreichbar ist. In der Folge würde `c` irrtümlich freigegeben werden, sodass ein hängender Zeiger entsteht oder sogar Datenverlust verursacht wird – der Algorithmus arbeitet also nicht korrekt.

Situationen, in denen die nebenläufige Ausführung von Programmsegmenten zu unvorhersehbarem Verhalten führt, werden als *race conditions* bezeichnet. Algorithmen, die zur Vermeidung von *race conditions* zwischen Kollektor und Mutator die Arbeit des letzteren unterbrechen, werden auch als *Stop-the-World-Algorithmen* (vgl. [LP06, S. 2]) bezeichnet.

2.2 Drei-Farben-Abstraktion

Da das Anhalten des Mutators während eines gesamten Garbage-Collection-Zyklus zu großen Verzögerungen führen kann, ist eine Optimierung der Markierungspha-

se wünschenswert. Zielführend ist, Kollektor und Mutator möglichst häufig eine nebenläufige Ausführung zu ermöglichen, ohne dabei die Korrektheit der Garbage Collection zu gefährden. Wir haben gesehen, dass die Manipulation von Referenzen während der Markierungsphase dazu führt, dass Verweise von markierten auf unmarkierte Objekte entstehen können, sodass erreichbare Objekte unmarkiert bleiben. Um dies zu vermeiden, könnte man als ersten Ansatz beim Schreiben einer neuen Referenz in ein Feld eines Objekts das Ziel dieser Referenz sofort markieren. Dies ist jedoch nur scheinbar eine Lösung: Da das Ziel ebenfalls Referenzen auf unmarkierte Objekte bereithalten könnte, entstehen dadurch möglicherweise neue Referenzen von markierten auf unmarkierte Objekte. Von DIJKSTRA et al. stammt ein Ansatz, der diese Idee aufgreift und um ein *Zwischenstadium* erweitert [Dij+78, S. 969f]. Diese ermöglicht es, markierte Objekte, die bereits komplett abgearbeitet wurden, von solchen zu unterscheiden, die bislang lediglich entdeckt wurden. Dazu werden Objekte mit drei verschiedenen Farben markiert:

weiß: Das Objekt wurde bislang nicht als erreichbar identifiziert. Bleibt es nach Ende der Markierungsphase weiß, kann es freigegeben werden.

grau: Das Objekt ist erreichbar, allerdings wurden die Felder des Objekts noch nicht auf Referenzen zu weiteren Objekten überprüft.

schwarz: Das Objekt ist erreichbar und alle Felder des Objekts wurden bereits überprüft.

Zu Beginn des Algorithmus sind alle existierenden Objekte weiß. Der ursprüngliche Mark-Sweep-Algorithmus 2.1 wird nun so modifiziert, dass Objekte bei ihrer Entdeckung grau und nach Abarbeitung ihrer Felder schwarz markiert werden. Hierfür existiert eine atomare Prozedur *setColor*, die die Markierung eines Objekts auf eine bestimmte Farbe `WHITE`, `GRAY` oder `BLACK`[2] setzt. Auf diese Art bleiben nicht mehr erreichbare Objekte weiß und können anschließend als löschbar identifiziert werden. Analog wird die *sweep*-Prozedur in Algorithmus 2.2 so abgeändert, dass Objekte gelöscht werden, wenn sie weiß markiert sind. Andernfalls wird ihre Markierung zurück auf weiß gesetzt.

Mithilfe dieser **Drei-Farben-Abstraktion** (engl. *tri-color abstraction*, Algorithmus 2.3) können nun Referenzmanipulationen zugelassen werden, die parallel zur Markierungsphase der Garbage Collection stattfinden: Wird in einem Objekt `a` eine Referenz auf ein Objekt `b` hinterlegt, so kann dies dazu führen, dass `b` erreichbar wird. Infolgedessen müssen auch alle von `b` referenzierten Objekte als erreichbar identifiziert werden. Somit ist es zielführend, `b` beim Setzen der Referenz grau zu markieren und zur Menge `graySet` hinzuzufügen, sofern dies noch nicht der Fall ist oder die Felder

[2] Die Farbinformation kann dadurch realisiert werden, dass jede Farbe mit einer eindeutigen Konstante identifiziert wird. Entsprechend ist darauf zu achten, dass dies mehr Speicherplatz zur Verwaltung der Markierungsinformationen benötigt.

Algorithmus 2.3 Markierung mit Drei-Farben-Abstraktion (vgl. [Dij+78, S. 970])

```
 1: markStart():
 2:   graySet ← ∅                                  ▷ Grau markierte Objekte
 3:   for each obj ∈ ROOTS
 4:     if isWhite(obj)
 5:       setColor(obj, GRAY)
 6:       add(graySet, obj)
 7:       mark()

 8: mark():
 9:   while graySet ≠ ∅
10:     obj ← remove(graySet)
11:     setColor(obj, BLACK)                       ▷ Objekt wird nun abgearbeitet
12:     for each ref ∈ POINTERS(obj)
13:       if isWhite(*ref)
14:         setColor(*ref, GRAY)                   ▷ Referenzierte Objekte grau markieren
15:         add(graySet, *ref)

16: sweep():
17:   pos ← nextObject(HEAP_START)
18:   while pos ≠ null
19:     if isWhite(*pos)
20:       free(pos)
21:     else setColor(*pos, WHITE)
22:     pos ← nextObject(pos)
23:   for each obj ∈ ROOTS
24:     setColor(obj, WHITE)
```

von `b` nicht bereits verfolgt wurden. Dies kann etwa mit einer *Schreibbarriere* (engl. *write barrier*) realisiert werden, die genau dann zum Einsatz kommt, wenn während eines Kollektionszyklus eine Referenz in ein Feld eines Objektes geschrieben wird (siehe Algorithmus 2.4). Auf diese Art kann es jedoch vorkommen, dass Objekte unerreichbar werden, nachdem sie bereits grau oder schwarz markiert wurden. Entsprechend verbleiben sie zunächst im Speicher und werden erst im nächsten Garbage-Collection-Zyklus entfernt.

Algorithmus 2.4 Schreibbarriere zur Manipulation von Referenzen in Objekten. `obj` bezeichnet das Objekt, in das die Referenz geschrieben wird, `i` den Index des Feldes und `ref` die zu schreibende Referenz.

```
1: atomic writeRef(obj, i, ref):
2:   if isWhite(*ref)
3:     setColor(*ref, GRAY)
4:     add(grayList, *ref)
5:   obj[i] ← ref
```

An dieser Stelle gehen wir auf die Terminierung und Korrektheit des modifizierten Mark-Sweep-Algorithmus ein.

Satz 2.2 (Korrektheit der Drei-Farben-Abstraktion)**:**
Der Mark-Sweep-Algorithmus mit Drei-Farben-Abstraktion terminiert und ist korrekt, sofern während der Arbeit des Kollektors keine neuen Objekte erzeugt werden.

Beweis der Terminierung: Zunächst ist festzuhalten, dass Objekte während der Markierungsphase ausschließlich *dunkler* gefärbt werden: In Algorithmus 2.3 werden lediglich weiß markierte Objekte grau gefärbt (Zeile 5 und 14); eine Weißfärbung geschieht in dieser Phase grundsätzlich nicht. Analog zum ursprünglichen Algorithmus 2.1 enthält `graySet` nur grau markierte Objekte. Jedes Objekt befindet sich dadurch höchstens einmal in dieser Menge. Da `graySet` nach jeder Iteration der **while**-Schleife um ein Objekt reduziert wird und keine neuen Objekte erzeugt werden, terminieren auch hier **while**- und **for each**-Schleife nach endlich vielen Schritten. Zuletzt terminiert auch die Bereinigungsphase (siehe Satz 2.1).

Zur Korrektheit: Erneut ist zu zeigen, dass keine Objekte unmarkiert bleiben, die erreichbar sind. Dies wäre genau dann der Fall, wenn nach der Markierungsphase ein schwarz markiertes Objekt eine Referenz auf ein weiß markiertes Objekt besitzt. Daher ist die Gültigkeit folgender Schleifeninvariante für die **while**-Schleife (Zeile 9 bis 15) zu zeigen:

(A) Es existieren keine zwei Objekte `a` und `b` mit $\texttt{a} \rightarrow \texttt{b}$, sodass `a` schwarz markiert und `b` weiß markiert ist.

Da zu Beginn alle Objekte weiß markiert sind und bis zum Eintritt in die **while**-Schleife Objekte nur grau markiert werden, gilt (A) trivialerweise, da keine schwarz markierten Objekte existieren. Gelte nun (A) zu Beginn einer Schleifeniteration und sei $\texttt{obj} = \texttt{a}$ dasjenige Objekt, das der Menge `graySet` entnommen und schwarz gefärbt wird (Zeile 10f). Existiert nun ein weiteres Objekt `b` mit $\texttt{a} \rightarrow \texttt{b}$, so ist $\texttt{\&b} \in \textsc{Pointers}(\texttt{a})$. Ist `b` bereits grau oder schwarz markiert, so geschieht nichts. Andernfalls wird $\texttt{*(\&b)} = \texttt{b}$ grau markiert (Zeile 14). In beiden Fällen ist `b` am Ende der Schleife nicht weiß markiert. Da `a` das einzige Objekt ist, das in dieser Iteration schwarz gefärbt wird, ist (A) nach der Iteration weiterhin gültig. Da die Schleife terminiert, die Invariante aufrecht erhalten bleibt und in der Bereinigungsphase nur weiß markierte Heapobjekte freigegeben werden, ist der Algorithmus korrekt. □

Die Atomizität der Schreibbarriere *writeRef* in Algorithmus 2.4 ist in der Tat notwendig, um die Korrektheit zu gewährleisten. Käme es unmittelbar nach der Markierung des Referenzziels, aber vor dem eigentlichen Setzen der Referenz in Zeile 5, zu einem kompletten Garbage-Collection-Zyklus, so würde die Markierung durch den Kollektor wieder aufgehoben. Damit bestünde im Anschluss die Möglichkeit, dass

die Invariante (A) verletzt wird. Allerdings kann die Invariante (A) abgeschwächt und damit eine *feingranularere* Lösung ermöglicht werden. Für Details hierzu sei auf die Publikation von DIJKSTRA et al. verwiesen [Dij+78, S. 972ff]. PIRINEN liefert darüber hinaus einen Überblick über Vor- und Nachteile verschiedener Lese- und Schreibbarrieren im Kontext der Drei-Farben-Abstraktion [Pir98].

2.3 Verzögerte Bereinigung

Die im vorigen Abschnitt vorgestellte Drei-Farben-Markierung dient vor allem dazu, die durch die Markierungsphase entstehenden Verzögerungen im Programmablauf zu reduzieren. In diesem Abschnitt wird eine Reduktion der Kosten der Sweep-Phase angestrebt. Meist können Mutator und Kollektor während der Bereinigung gleichzeitig tätig sein: Nicht mehr erreichbare Objekte sind ohne Nebenwirkungen zerstörbar und Markierungsinformationen werden so hinterlegt, dass sie für den Mutator grundsätzlich nicht zugänglich sind. *Race conditions* sind dadurch prinzipiell ausgeschlossen. Ferner ergibt sich hierdurch die Möglichkeit, die Bereinigungsphase nicht unmittelbar nach der Markierungsphase ausführen zu müssen. Für Systeme ohne hardware- oder softwareseitige Unterstützung von Multithreading, in denen die Aufgaben des Mutators und Kollektors nicht gleichzeitig bearbeitet werden können, bietet sich hingegen die **verzögerte Bereinigung** (engl. *lazy sweeping*) nach HUGHES [Hug82] an.

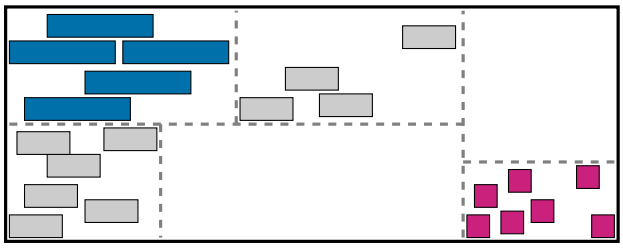


Abbildung 2.3.: Veranschaulichung eines in Blöcken aufgeteilten Heaps. Jeder Block kann Objekten einer festgelegten Größenordnung zugewiesen werden. Für eine Größenordnung können allerdings mehrere Blöcke verwendet werden.

Bei der verzögerten Bereinigung ist der Heapspeicher in **Blöcke** eingeteilt. Jeder Block enthält Objekte einer festgelegten Größenordnung; zu jeder Größenordnung können mehrere Blöcke existieren (Abbildung 2.3). Die Idee ist nun, das Sweeping durch den Allokator ausführen zu lassen, wenn Speicher angefordert wird. Dabei werden die zuvor markierten Objekte freigegeben, allerdings nur in denjenigen Blöcken, die der angeforderten Speichermenge zugeordnet sind. Die Bereinigung beschränkt sich dadurch auf einen Bruchteil des Heaps.

Dieser Ansatz geht mit einigen marginalen Änderungen an den bisher betrachteten Algorithmen einher. Zum einen wird vorausgesetzt, dass beim Markieren eines

Algorithmus 2.5 Verzögertes Bereinigen des Heaps (vgl. [JHM11, S. 25]).

```
 1: new(size):
 2:   adr ← allocate(size)
 3:   if adr = null
 4:     lazySweep(size)                          ▷ Starte Lazy-Sweep-Zyklus
 5:     adr ← allocate(size)                     ▷ Zweiter Versuch
 6:     if adr = null
 7:       collect()          ▷ Nach weiteren unerreichbaren Objekten suchen
 8:       lazySweep(size)                        ▷ Zweiter Lazy-Sweep-Zyklus
 9:       adr ← allocate(size)                   ▷ Dritter Versuch
10:       if adr = null
11:         error("Nicht genügend Speicher")
12:   return adr

13: collect():
14:   markStart()                          ▷ Siehe Algorithmus 2.2 bzw. 2.3
15:   for each blk ∈ BLOCKS
16:     if not isMarked(blk)
17:       free(blk)                  ▷ Gib komplett verwaisten Block sofort frei
18:     else add(toSweep, blk)

19: lazySweep(size)
20:   do
21:     blk ← remove(toSweep, size)
22:     if sweep(start(blk), end(blk))
23:       return                   ▷ Beende, sobald Speicher gewonnen wurde
24:   while blk ≠ null
25:   createBlock(size)        ▷ Initialisiere neuen Block, wenn nichts freigegeben
```

Objekts auch der entsprechende Block markiert wird, in dem sich das Objekt befindet. Dies kann recht einfach realisiert werden, indem bei der Initialisierung eines Blocks durch den Allokator zusätzlich Platz für einen Header reserviert wird, welcher Metadaten wie die Größenordnung der enthaltenen Objekte aufnimmt. Weiter wird verlangt, dass der Allokator zusätzlich Informationen über Anzahl, Lage und Größenordnung der existierenden Blöcke besitzt. Zuletzt wird die Prozedur *sweep* aus den Algorithmen 2.2 und 2.3 dahingehend abgewandelt, dass sie nur einen eingeschränkten Teil des Heaps traversiert – etwa, indem zwei Parameter für Start- und Endadresse übergeben werden. Zudem soll *sweep* den Wahrheitswert `true` zurückgeben, wenn zumindest ein Objekt entfernt wurde, und andernfalls `false`.

Nach Abschluss der Markierungsphase kann anhand der zusätzlichen Markierung der Blöcke festgestellt werden, welche nur verwaiste Objekte enthalten. In diesem Fall kann der Allokator einen Teilbereich des Heaps en bloc freigeben (Zeile 17 von Algorithmus 2.5). Hingegen werden Blöcke, die mindestens ein erreichbares Objekt enthalten und somit markiert sind, nach Abschluss der Markierungsphase einer Menge `toSweep` zugefügt (Zeile 18). In dieser wird Buch geführt, welche

Blöcke differenzierter bereinigt werden müssen, da sie sowohl erreichbare als auch unerreichbare Objekte enthalten können.

Im Gegensatz zu den bisher betrachteten Mark-Sweep-Algorithmen beginnt die Bereinigungsphase nicht unmittelbar nach Abschluss der Markierung, sondern erst bei Anforderung von Speicher mittels *new*. Falls dabei kein freier Speicher geeigneter Größe gefunden werden kann, weil etwa die entsprechenden Blöcke ausgeschöpft sind, beginnt ein *Lazy-Sweep-Zyklus* (Zeile 20 bis 24). Dabei werden nach und nach Blöcke aus `toSweep` entfernt und bereinigt, die Objekte der angeforderten Speichergröße verwalten. Sobald ein Block um ein Objekt reduziert werden konnte, wird der Zyklus beendet und der Allokator kann den gewonnenen Speicher neu zuordnen (Zeile 22f). Falls aus keinem Block ein Objekt freigegeben werden konnte, bedeutet dies, dass alle Blöcke der entsprechenden Objektgröße mit erreichbaren Objekten gefüllt sind. In diesem Fall muss der Allokator einen neuen Block initialisieren, der neue Objekte aufnehmen kann (Zeile 25).

Falls nicht genügend freier Speicher zur Verfügung steht, um einen weiteren Block zu erzeugen, schlägt auch der zweite Allokationsversuch fehl (Zeile 5f). Der letzte Versuch besteht nun darin, die Markierungen der Objekte zu aktualisieren, indem eine Markierungsphase gestartet wird (Zeile 7). Da die letzte Markierungsphase nicht unmittelbar vor der Speicheranforderung, sondern möglicherweise wesentlich früher stattgefunden hat, könnten in der Zwischenzeit weitere Objekte verwaist sein. Diese werden nachfolgend in einem zweiten Lazy-Sweep-Zyklus freigegeben (Zeile 8), sodass zuletzt Platz innerhalb eines Blocks bzw. für einen neuen Block geschaffen werden kann.

Während die Einschränkung der Bereinigung auf einzelne Blöcke zwar einen Performancevorteil mit sich bringt, besitzt diese Variante des Mark-Sweep-Algorithmus auch einige Nachteile: Eine ungünstige Kombination von Speicheranforderungen und Sweep-Zyklen könnte dazu führen, dass viele Blöcke derselben Größenordnung angelegt werden, die jeweils nur wenige Objekte enthalten und größtenteils ungenutzt sind. Wenn jeder Block allerdings mindestens ein erreichbares Objekt enthält, kann er nicht in Gänze freigegeben werden. Eine hohe Zahl dieser ineffizient genutzter Blöcke führt dazu, dass möglicherweise nicht mehr genügend freier Speicher zur Verfügung steht, um Blöcke anderer Größenordnung zu initialisieren, obwohl ausreichend Speicher vorhanden wäre, der nicht durch Objekte belegt wird. Eine Lösung dieses Problems wäre eine zusätzliche Konsolidierungsphase, bei der Blöcke gleicher Größenordnung zusammengelegt werden. Dies bedeutet jedoch zusätzlichen Aufwand für die Verschiebung von Objekten (siehe Kapitel 4). Der zweite Nachteil ist, dass nicht mehr erreichbare Objekte eventuell erst sehr spät freigegeben werden, wenn primär Objekte anderer Größenordnungen angefordert werden. In so einem Fall können auch Blöcke, die keine erreichbaren Objekte mehr enthalten, über lange

Zeit im Speicher verweilen, sodass diese für einen gewissen Zeitraum ein Speicherleck darstellen. Zuletzt kann auch die Aufteilung des Heaps nach Objektgrößen nicht zielführend sein: Je nach Anwendungsfall und verwendeter Programmiersprache besteht die Möglichkeit, dass zwischen den meisten Objekten keine signifikanten Größenunterschiede bestehen und die Blöcke einer bestimmten Größenordnung besonders stark beansprucht werden. Der Vorteil, nur ein beschränktes Gebiet des Heaps zu bereinigen, würde damit deutlich geschmälert werden.

2.4 Weitere Varianten und Komplexität

Der in diesem Kapitel betrachtete Mark-Sweep-Algorithmus gilt als der erste Algorithmus zur automatischen Speicherverwaltung, weswegen zahlreiche Varianten existieren, die für bestimmte Anwendungen und Systeme optimiert sind. Allen gemein ist das Auffinden erreichbarer Objekte durch Verfolgung von Referenzen in der Markierungsphase. Obwohl die erreichbaren Objekte weniger Speicher belegen als während der Bereinigungsphase traversiert werden muss, ist die Markierungsphase in der Praxis die aufwendigere. Häufig befinden sich Objekte, die aufeinander verweisen, nicht in unmittelbarer Nähe zueinander. Das Auffinden erreichbarer Objekte verursacht daher schwer vorhersehbare Speicherzugriffe. Caching- und Prefetch-Mechanismen, die durch vorweggenommenes Bereitstellen von Speicherinhalten Zugriffe beschleunigen, profitieren hingegen von *zeitlicher* und *räumlicher Lokalität* von Daten, die in einer Beziehung zueinander stehen. In der Markierungsphase können sie daher Speicherzugriffe nicht so effektiv beschleunigen wie in der Bereinigungsphase [JHM11, S. 21f]. Um diesen Makel zu beheben, kann die Zugriffsreihenfolge auf Objekte variiert werden. Anstatt etwa im naiven Algorithmus 2.1 die Abarbeitung der `toDo`-Menge zu beginnen, sobald das erste Basisobjekt erfasst wurde, können stattdessen auch zunächst alle Basisobjekte zu `toDo` hinzugefügt und die Prozedur *mark* im Anschluss aufgerufen werden. Je nachdem, wie `toDo` in der Praxis realisiert wird – zum Beispiel in Form eines Stacks – kann die Traversierung der Objekte – und damit die Performanz des Kollektors in der Markierungsphase – signifikant beeinflusst werden (vgl. [GBF07]).

Statt Markierungsinformationen im Header zu speichern, können diese auch in Bitmaps oder tabellenähnlichen Strukturen außerhalb eines Objekts verwaltet werden. Dies vermeidet schreibende Zugriffe auf Objekte während der Markierungsphase, da Objekte ohne Referenzfelder übersprungen werden können und die Garbage Collection das Caching weniger stark beeinträchtigt. Im Falle eines in Blöcken eingeteilten Heaps bietet es sich beispielsweise an, pro Block eine Bitmap anzulegen. Somit kann während der Bereinigung schnell erkannt werden, ob größere Teile eines Blocks freigegeben werden können, ohne ihn komplett traversieren zu müssen.

Dieses Kapitel schließt mit einer Betrachtung der Komplexität der vorgestellten Mark-Sweep-Varianten ab. In seiner einfachsten Form erweist sich der Mark-Sweep-Algorithmus als vergleichsweise platzsparend: Werden Markierungsinformationen unmittelbar in den Objekten gespeichert, wird pro Objekt lediglich ein einzelnes Bit oder Byte beansprucht. Zwischen verschiedenen Mark-Sweep-Zyklen müssen ferner keinerlei zusätzliche Informationen bereitgehalten werden. Während eines Zyklus muss jedoch die Menge `toDo` der noch zu untersuchenden Objekte verwaltet werden. Da bereits festgestellt wurde, dass keine Objekte mehrfach zu `toDo` hinzugefügt werden, ist ihre Mächtigkeit durch die Anzahl $|\mathcal{R}|$ der erreichbare Objekte beschränkt. Davon ausgehend, dass in `toDo` ausschließlich Referenzen konstanter Größe gespeichert werden müssen, ergibt sich insgesamt ein Platzbedarf von $\mathcal{O}(|\mathcal{R}|)$.

Während der Speicherbedarf der Markierungsphase also linear mit der Anzahl der erreichbaren Objekte wächst, spielt für die Laufzeit auch die Anzahl der Referenzen eine Rolle. Für jedes erreichbare Objekt müssen alle enthaltenen Referenzen verfolgt werden, um auszuschließen, dass kein erreichbares Objekt unmarkiert bleibt. Die Komplexität dieser Operation ist analog zur vollständigen Traversierung eines beliebigen Graphen (V, E), die durch $\mathcal{O}(|V| + |E|)$ gegeben ist [Cor+09, Kap. 22]. V entspricht hierbei der Menge $\mathcal{R}$ der erreichbaren Objekte und E die Menge der von ihnen ausgehenden Referenzen.

Satz 2.3 (Speicherbedarf und Komplexität des Mark-Sweep-Algorithmus)**:**
Für den naiven Mark-Sweep-Algorithmus gelten folgende Eigenschaften:

(1) Der Speicherbedarf des gesamten Algorithmus liegt in $\mathcal{O}(|\mathcal{R}|)$.

(2) Die Laufzeit der Markierungsphase liegt in $\mathcal{O}\left(|\mathcal{R}| + \sum_{a \in \mathcal{R}} |\textsc{Pointers}(a)|\right)$.

(3) Die Laufzeit der Bereinigungsphase liegt in $\mathcal{O}(|\mathbb{H}|)$, wobei $|\mathbb{H}|$ die Größe des Heaps bezeichnet.

Für die Drei-Farben-Abstraktion ergeben sich identische asymptotische Laufzeiten; in der Praxis ermöglicht die Nebenläufigkeit von Kollektor und Mutator jedoch geringere Verzögerungen im Programmablauf. Die Kosten der Bereinigungsphase des Lazy Sweeping nach HUGHES sind nur schwer abschätzbar, da sie unter anderem von der Strukturierung des Heaps in Blöcken abhängen, welche wiederum je nach Anwendungsfall einen starken Einfluss auf die Ausführungshäufigkeit der Garbage Collection haben kann. Grundsätzlich ist es für Mark-Sweep-Ansätze empfehlenswert, einen größeren Puffer für die Arbeit des Kollektors zu reservieren, um eine hohe Zahl an Speicheranforderungen seitens des Allokators bewältigen zu können. Andernfalls besteht die Gefahr, dass die Garbage Collection zu häufig ausgelöst wird und den Mutator nach und nach verdrängt (vgl. [JL96, S. 70]).

Während wir in diesem Kapitel mit dem Mark-Sweep-Algorithmus und seinen Optimierungen ein erstes Verfahren zur Bewältigung des Ausgangsproblems eingeführt haben, wird im nächsten Kapitel ein fundamental anderer Ansatz vorgestellt, der zunächst größere Unterbrechungen im Programmablauf vermeidet. Dabei wird auch diskutiert, welche Vor- und Nachteile sich im Vergleich zum Mark-Sweep-Algorithmus ergeben und welche Optimierungsmöglichkeiten hier ausgeschöpft werden können.

3 Referenzzählung

Der zweite Garbage-Collection-Algorithmus, der in dieser Arbeit vorgestellt wird, stammt von COLLINS und aus demselben Jahr wie der Mark-Sweep-Algorithmus. COLLINS betrachtet wie MCCARTHY das *LISP Programming System* auf Großrechnern und das Problem, wann Listenelemente, die in mehreren Listen vorkommen können, wieder freigegeben werden dürfen. MCCARTHYS Ansatz bezeichnet er allerdings als „elegant but inefficient“ [Col60, S. 655], da dieser sowohl zeitaufwändig sei, als auch den Speicherplatz für Nutzdaten einschränke. Stattdessen schlägt COLLINS vor, zusätzlich zu einem Datum die Anzahl der Referenzen auf dieses zu speichern. Anhand dieses Zählers, der bei der Manipulation von Referenzen aktualisiert wird, kann unmittelbar festgestellt werden, ob ein Datum verwaist ist und der entsprechende Speicherplatz freigegeben werden kann [Col60, S. 656f]. Dieser Ansatz – die **Referenzzählung** (engl. *reference counting*) – wird in diesem Kapitel ausführlich betrachtet.

3.1 Naive Referenzzählung

Zur Realisierung des Ansatzes von COLLINS im eingangs betrachteten Speichermodell wird erneut der Header eines Heappbjekts $\mathtt{a}$ genutzt, um einen ganzzahligen, nicht negativen Wert $\mathtt{rc}(\mathtt{a})$ zu hinterlegen, der als **Referenzzähler** von $\mathtt{a}$ bezeichnet wird. Dieser gibt an, wie viele Referenzen von anderen Objekten auf $\mathtt{a}$ existieren und wird bei der Erzeugung von $\mathtt{a}$ mit dem Wert 0 initialisiert. Auf diese Art erhält man unmittelbar eine notwendige Bedingung für die Erreichbarkeit eines Heapobjekts: Ist $\mathtt{a} \in \mathcal{R}$, so existiert mindestens ein Objekt $\mathtt{b} \neq \mathtt{a}$ mit $\mathtt{b} \rightarrow \mathtt{a}$ und somit gilt $\mathtt{rc}(\mathtt{a}) \geq 1$. Damit dieses Kriterium auch von Basisobjekten ausgehende Referenzen einschließt, wird im Folgenden die Menge ROOTS der Basisobjekte selbst als Objekt aufgefasst. Entsprechend wird mit POINTERS(ROOTS) die Menge der Referenzen auf Objekte des Heaps bezeichnet, die von Basisobjekten ausgehen. Für Basisobjekte selbst, die nach Konvention nicht zum Heap gehören, wird kein Referenzzähler geführt, da sie per definitionem stets erreichbar sind.

Lemma 3.1 (Notwendige Bedingung für Erreichbarkeit)**:**
Für ein Objekt $\mathtt{a}$ des Heaps gilt: Ist $\mathtt{a} \in \mathcal{R}$, so folgt $\mathtt{rc}(\mathtt{a}) \geq 1$.

Diese Bedingung ist allerdings nicht hinreichend, da die Objekte, von denen die Referenzen ausgehen, gegebenenfalls nicht erreichbar sind und die Erreichbarkeit von `a` somit nicht garantiert werden kann. Jedoch folgt als Kontraposition von Lemma 3.1 sofort, dass ein Heapobjekt `a` mit $\texttt{rc}(\texttt{a}) = 0$ nicht erreichbar ist und freigegeben werden kann.

Wie kann `rc(a)` nun verwendet werden, um automatisch ein nicht erreichbares Objekt freizugeben? Immer, wenn eine Referenz auf `a` in ein Feld eines anderen Objekts geschrieben wird, so muss `rc(a)` inkrementiert werden. Gleichzeitig muss der Referenzzähler des Objekts `b`, welches das Ziel der überschriebenen Referenz war, dekrementiert werden. Wenn dieser anschließend den Wert 0 aufweist, kann `b` sofort freigegeben werden. Diese Manipulationen der Referenzzähler sind unmittelbar auszuführen, wenn eine Referenz manipuliert wird. Daher bieten sich zur Realisierung Schreibbarrieren an, die bereits in Abschnitt 2.2 eingeführt wurden.

Algorithmus 3.1 Naive Referenzzählung mittels Schreibbarriere (vgl. [JHM11, S. 58]).

```
 1: atomic writeRef(obj, i, ref):
 2:   if (ref ≠ null ∧ *ref ≠ obj)          ▷ Erhöhe rc, falls nicht null oder
 3:     rc(*ref) ← rc(*ref) + 1               Selbstreferenz geschrieben wird
 4:   if (obj[i] ≠ null ∧ *obj[i] ≠ obj)    ▷ Reduziere rc, falls nicht null oder
 5:     decRefCount(*obj[i])                  Selbstreferenz überschrieben wird
 6:   obj[i] ← ref

 7: decRefCount(obj):
 8:   rc(obj) ← rc(obj) − 1
 9:   if rc(obj) = 0
10:     for each ref ∈ POINTERS(obj)
11:       if (ref ≠ null ∧ *ref ≠ obj)      ▷ Reduziere rekursiv rc
12:         decRefCount(*ref)                 referenzierter Objekte
13:     free(&obj)
```

Die in Algorithmus 3.1 aufgeführte Schreibbarriere kommt genau dann zum Einsatz, wenn durch den Mutator mittels `obj[i]` ← `ref` in ein Feld eines Objekts eine Referenz auf ein Heapobjekt geschrieben wird. Dabei wird zunächst geprüft, ob eine Selbstreferenz oder `null` geschrieben werden soll (Zeile 2f). In beiden Fällen ist der Referenzzähler des Ziels nicht zu erhöhen, da die Operation die Erreichbarkeit des Ziels nicht beeinflusst. Analog wird überprüft, ob im betroffenen Feld bereits eine Referenz auf ein anderes Objekt gespeichert ist (Zeile 4f). Wenn ja, muss dessen Referenzzähler entsprechend dekrementiert werden. Im Anschluss kann die neue Referenz in das Feld des Objekts geschrieben werden.

Das Verringern des Referenzzählers wird durch die Prozedur *decRefCount* übernommen. Dabei wird zugleich geprüft, ob dieser anschließend 0 ist (Zeile 9). In diesem Fall wird das entsprechende Objekt `obj` freigegeben. Die Freigabe führt allerdings dazu, dass auch die Zähler derjenigen Objekte verringert werden müssen, die von

`obj` referenziert werden. Ensprechend wird für jede Referenz in den Feldern von `obj`, die nicht `null` oder eine Selbstreferenz ist, *decRefCount* rekursiv aufgerufen (Zeile 10 bis 12).

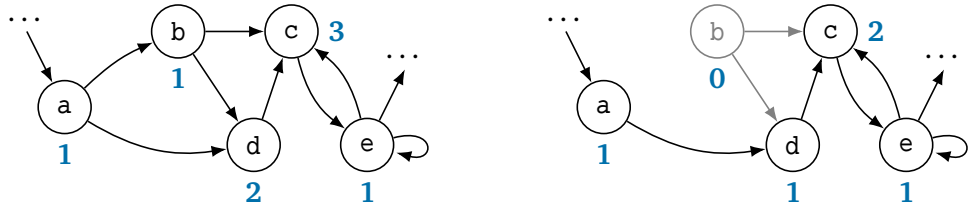


Abbildung 3.1.: Wird die Referenz $\texttt{a} \rightarrow \texttt{b}$ entfernt, so wird `b` freigegeben, da der Referenzzähler von `b` auf 0 fällt. In der Folge müssen auch die Referenzzähler von `c` und `d` angepasst werden.

Die Atomizität der Schreibbarriere verhindert die Entstehung von *race conditions* bei gleichzeitigen Zugriffen auf Referenzzähler und ist daher unabdingbar. Ebenso ist es wesentlich, dass der Referenzzähler des neuen Ziels zunächst inkrementiert wird, bevor der des alten Ziels dekrementiert wird. Andernfalls könnte es, wenn altes und neues Ziel identisch sind, passieren, dass der Zähler auf 0 reduziert würde. In der Folge würde das Ziel sofort freigegeben werden, obwohl es weiterhin erreichbar bliebe.

Satz 3.1:
Der Algorithmus 3.1 zur Referenzzählung ist korrekt und terminiert.

Beweis: Zu zeigen ist zunächst, dass die rekursiven Aufrufe von *decRefCount* in Zeile 12 terminieren. Angenommen, es gäbe eine nicht terminierende Folge von Aufrufen $\mathit{decRefCount}(\texttt{a}_1), \mathit{decRefCount}(\texttt{a}_2), \ldots$ für Objekte $\texttt{a}_i, i \in \mathbb{N}$. Da jedes Objekt nur endlich viele Referenzen auf andere Objekte besitzt, ist $\textsc{Pointers}(\texttt{a}_i)$ endlich für alle $i \in \mathbb{N}$. Weiter ist die Anzahl aller Objekte endlich; es muss also ein Objekt $\texttt{a}_j$ existieren, für welches $\mathit{decRefCount}(\texttt{a}_j)$ unendlich oft aufgerufen wird. Allerdings wird $\mathit{decRefCount}(\texttt{a}_j)$ nur aufgerufen, wenn ein Objekt `b` mit $\texttt{b} \rightarrow \texttt{a}_j$ existiert, für welches zuvor $\texttt{rc}(\texttt{b})$ auf 0 gesetzt wurde. Da während der rekursiven Aufrufe von *decRefCount* keine Referenzzähler erhöht werden, müssen also unendlich viele verschiedene Objekte `b` mit dieser Eigenschaft existieren. Das ist jedoch ein Widerspruch zur Endlichkeit der Anzahl aller Objekte.

Es bleibt zu zeigen, dass der Algorithmus korrekt ist. Die Freigabe eines Objekts `a` durch den Algorithmus erfolgt ausschließlich in Zeile 13. Diese wird nur ausgelöst, wenn $\texttt{rc}(\texttt{a}) = 0$ gilt. Aus Lemma 3.1 folgt, dass `a` nicht erreichbar ist. □

Die Garbage Collection mithilfe naiver Referenzzählung bietet im Vergleich zum Mark-Sweep-Algorithmus einige Vorteile, aber auch gewisse Nachteile (vgl. [LH06, S. 346]): Eine Freigabe von Objekten wird unmittelbar durch ihre Verwaisung ausgelöst und nicht etwa, wenn unzureichend freier Speicher zur Verfügung steht. Dadurch wird die Allokation von Speicher für neu erzeugte Objekte nicht verzögert. Verwaiste Objekte werden beim Löschen der letzten Referenz auf sie unmittelbar erkannt und freigegeben und nicht erst beim nächsten Mark-Sweep-Zyklus. Speichermangel kann de facto nur dadurch entstehen, dass der gesamte Speicher von erreichbaren Objekten belegt wird – in diesem Fall bietet allerdings kein Garbage-Collection-Algorithmus Abhilfe – oder der Heap zunehmend fragmentiert wird (siehe Kapitel 4). Referenzzählung benötigt zudem keinen Zugriff auf den vollständigen Objektgraphen: Da kein Aufspüren aller erreichbaren Objekte durchgeführt wird, kann auf eine Traversierung des Objektgraphen verzichtet werden. Dies ist besonders für verteilte Systeme von Vorteil, in denen die Markierungsphase sonst mehrere Knoten des Systems belasten und erhöhten Datenaustausch zwischen diesen verursachen würde. Weiter lässt sich erahnen, dass Referenzzählung weniger stark dazu neigt, Cache-Mechanismen zu beeinträchtigen. Im Gegensatz zu Mark and Sweep finden kaum unvorhersehbare Objektzugriffe statt. Wird der Referenzzähler eines Objektes erhöht, so wird gerade eine neue Referenz hierauf angelegt. Die Wahrscheinlichkeit, dass im Programmablauf kurz danach auch ein Zugriff auf dieses Objekt vorgesehen ist, ist daher relativ hoch. Zeitliche Lokalität ist somit gegeben.

Trotz dieser Vorteile ist die naive Referenzzählung jedoch kein Allheilmittel. Dadurch, dass bei jeglicher Referenzänderung eine Manipulation der Zähler erfolgt, werden Operationen des Mutators, die ursprünglich nur lesend auf Objekte zugreifen, zu Schreibzugriffen. Diese Problematik wird in Verbindung mit dynamischen Datenstrukturen besonders deutlich: Betrachtet man etwa die lineare Suche auf einer einfach verketteten Liste, so fällt auf, dass bei jeder Iteration die Referenzvariable auf das aktuell betrachtete Element neu gesetzt wird (Zeile 4 in Algorithmus 3.2). Dadurch werden zwei Schreibzugriffe notwendig, welche den Zähler des zuletzt betrachteten Objekts reduzieren und den des nächsten Objekts erhöhen, obwohl deren Erreichbarkeit tatsächlich nicht beeinflusst wird. Die Anzahl an Speicheroperationen wird also signifikant vergrößert. Durch den rekursiven Aufruf von *decRefCount* kann es zudem zu einer Kaskade an Freigaben und Zählermanipulationen kommen, sobald ein Objekt freigegeben wird. In Verbindung mit der Atomizität der Schreibbarriere können dadurch größere und unerwartete Verzögerungen im Programmablauf entstehen, die bei zeitkritischen Anwendungen um jeden Preis zu vermeiden sind. Die Speicherung eines Zählers für jedes Objekt, dessen Wert potenziell nur durch die Anzahl aller Referenzfelder begrenzt ist, kann in Anwendungen mit vielen kleineren Objekten zudem einen relevanten zusätzlichen Speicherbedarf hervorrufen.

Algorithmus 3.2 Lineare Suche in einer verketteten Liste. Obwohl der Algorithmus keine Objekte manipuliert, verursacht jede Änderung der Referenzvariable `cur` die Manipulation von zwei Referenzzählern.

```
1: linSearch(list, x):
2:     cur ← first(list)
3:     while (cur ≠ null ∧ *cur ≠ x)
4:         cur ← next(list)
5:     return cur
```

Ein weiteres Problem entsteht, wenn zyklische Datenstrukturen wie etwa doppelt verkettete Listen verwendet werden. Die gesamte Struktur kann unerreichbar sein, obwohl die Referenzzähler der einzelnen Objekte nicht 0 sind (siehe Abbildung 3.2). Die einzelnen Objekte werden dann nicht als löschbar erkannt, was zu einem Speicherleck führt. Eine mögliche Lösung dieser Problematik wird im nächsten Abschnitt betrachtet. Im Anschluss daran werden weitere Ansätze vorgestellt, die eine Effizienzsteigerung der naiven Referenzzählung anstreben.

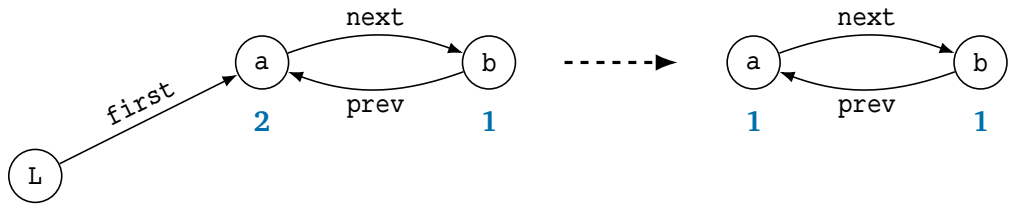


Abbildung 3.2.: Referenzzählung in zyklischen Datenstrukturen. Wird Objekt `L` entfernt, sind `a` und `b` nicht mehr erreichbar. Jedoch fallen ihre Referenzzähler nicht auf 0, sodass sie als Speicherlecks im Speicher verbleiben.

3.2 Zyklische Referenzen

Die fehlende Möglichkeit zur Erkennung und Freigabe zyklischer Strukturen kann – bei häufigem Auftreten – große Teile des Heaps unverwendbar werden lassen. Ein gelegentlich zusätzlich ausgeführter Mark-Sweep-Zyklus mag diese Problematik zwar beheben, macht allerdings die oben genannten Vorteile der Referenzzählung zunichte. Nachfolgend wird ein Algorithmus von MARTÍNEZ, WACHENCHAUZER und LINS vorgestellt, der die naive Referenzzählung um ein Verfahren ergänzt, mit welchem zyklische Strukturen zuverlässig erkannt werden können [MWL90]. Dieses basiert darauf, dass ein Zyklus von Objekten als **starke Zusammenhangskomponente** im Objektgraphen aufgefasst werden kann (vgl. [LH06, S. 348]). Das bedeutet, dass von jedem Objekt aus jedes andere Objekt innerhalb des Zyklus erreicht werden kann.

Definition 3.1 (Starke Zusammenhangskomponente)**:**
Sei O eine Menge von Objekten und $\mathtt{a} \in O$. $\langle \mathtt{a} \rangle \subseteq O$ heißt **starke Zusammenhangskomponente** von $\mathtt{a}$, wenn gilt:

(1) $\mathtt{a} \in \langle \mathtt{a} \rangle$

(2) Für alle $\mathtt{b}, \mathtt{c} \in \langle \mathtt{a} \rangle$ gilt $\mathtt{b} \xrightarrow{*} \mathtt{c}$ und $\mathtt{c} \xrightarrow{*} \mathtt{b}$.

(3) $\langle \mathtt{a} \rangle$ ist maximal, das heißt für jede Teilmenge $M \subseteq O$, die **(1)** und **(2)** erfüllt, gilt $M \subseteq \langle \mathtt{a} \rangle$.

Eine Referenz $\mathtt{x} \rightarrow \mathtt{a}$ auf $\mathtt{a}$ heißt **intern**, wenn $\mathtt{x} \in \langle \mathtt{a} \rangle$. Andernfalls heißt sie **extern**.

Man kann leicht sehen, dass die starke Zusammenhangskomponente eines Objekts eindeutig bestimmt ist und zwei starke Zusammenhangskomponenten entweder identisch oder disjunkt sind.[1]

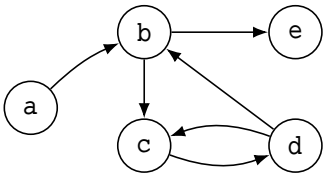


Abbildung 3.3.: Der abgebildete Objektgraph besitzt drei starke Zusammenhangskomponenten: $\langle \mathtt{a} \rangle = \{\mathtt{a}\}$, $\langle \mathtt{b} \rangle = \{\mathtt{b}, \mathtt{c}, \mathtt{d}\}$ und $\langle \mathtt{e} \rangle = \{\mathtt{e}\}$.

Passt man die Referenzzähler jedes Objekts so an, dass interne Referenzen unberücksichtigt bleiben, geben diese anschließend Auskunft darüber, wie viele externe Referenzen von Objekten außerhalb der Komponente auf Objekte innerhalb selbiger existieren. Besitzen alle so modifierten Zähler den Wert 0, so ist die Komponente nicht mehr erreichbar und kann entfernt werden. Anders formuliert: Besitzt die Komponente ein Objekt, das über eine externe Referenz erreichbar ist, so ist die gesamte Komponente von außen erreichbar.

Zusätzlich zum Referenzzähler wird pro Objekt eine Markierungsinformation gespeichert, die die Werte `WHITE`, `GRAY` und `BLACK` annehmen kann (vgl. Abschnitt 2.2). Nach Untersuchung der hypothetisch zyklischen Struktur sollen erreichbare Objekte weiß und nicht erreichbare schwarz markiert sein. Zu Beginn sind alle Objekte weiß markiert. Algorithmus 3.3 ergänzt die Prozedur *decRefCount* der naiven Referenzzählung um einen **else**-Fall, der ausgelöst wird, wenn der Referenzzähler `rc(obj)` nach Dekrementierung nicht 0 beträgt (vgl. Zeile 7 bis 10). In diesem Fall muss überprüft werden, ob `obj` das letzte Objekt einer starken Zusammenhangskomponente war,

[1] Betrachtet man die Relation $\sim$ auf O definiert durch $\mathtt{a} \sim \mathtt{b} :\Leftrightarrow \mathtt{a} = \mathtt{b} \vee \mathtt{a} \xrightarrow{*} \mathtt{b} \wedge \mathtt{b} \xrightarrow{*} \mathtt{a}$, so ist leicht zu sehen, dass diese eine Äquivalenzrelation auf O ist. Die Menge $\langle \mathtt{a} \rangle$ ist dann nichts anderes als die Äquivalenzklasse $[\mathtt{a}]_\sim$ von $\mathtt{a} \in O$ bezüglich $\sim$. Entsprechend lässt sich die Menge O in ihre starken Zusammenhangskomponenten partitionieren.

Algorithmus 3.3 Zyklische Referenzzählung nach MARTÍNEZ et al. (vgl. [MWL90, S. 32])

```
 1: decRefCount(obj):
 2:     rc(obj) ← rc(obj) − 1
 3:     if rc(obj) = 0
 4:         for each ref ∈ POINTERS(obj)
 5:             decRefCount(*ref)
 6:         free(&obj)
 7:     else
 8:         markGray(obj)          ▷ Entferne Zählung interner Referenzen
 9:         scan(obj)              ▷ Prüfe Erreichbarkeit
10:         collect(obj)           ▷ Entferne unerreichbare Strukturen
```

durch welches diese erreichbar war. Dazu wird nach einem dreischrittigen Verfahren vorgegangen, das im Folgenden beschrieben wird.[2]

Wir beginnen mit der Prozedur *markGray*, die zunächst für `obj` aufgerufen wird. Davon ausgehend, dass `obj` Bestandteil einer zyklischen Struktur sein kann, werden zunächst Zählungen von internen Referenzen entfernt, indem per Tiefensuche alle unmarkierten und von `obj` referenzierten Objekte betrachtet, grau markiert und ihre Referenzzähler angepasst werden (Zeile 3 bis 5 in Algorithmus 3.4). Durch einen rekursiven Aufruf in Zeile 6 wird somit die starke Zusammenhangskomponente ⟨`obj`⟩ behandelt und grau markiert; die Referenzzähler ihrer Objekte enthalten anschließend die Anzahl aller externen Referenzen.

Algorithmus 3.4 Zyklische Referenzzählung – Markierungsphase (vgl. [MWL90, S. 32f])

```
 1: markGray(obj):
 2:     if isWhite(obj)
 3:         setColor(obj, GRAY)
 4:         for each ref ∈ POINTERS(obj)        ▷ Entferne interne Referenz,
 5:             rc(*ref) ← rc(*ref) − 1         die von obj ausgeht
 6:             markGray(*ref)                  ▷ Reduziere rekursiv weitere rc

 7: scan(obj):
 8:     if isGray(obj)
 9:         if rc(obj) = 0                      ▷ Objekt besitzt keine externen Referenzen
10:             setColor(obj, BLACK)
11:             for each ref ∈ POINTERS(obj)
12:                 scan(*ref)                  ▷ Prüfe rekursiv referenzierte Objekte
13:         else unmark(obj)                    ▷ Objekt besitzt mind. eine externe Referenz
```

Im zweiten Schritt wird die Prozedur *scan* für `obj` aufgerufen, um die modifizierten Referenzzähler der grau markierten Objekte auszuwerten. Besitzt `rc(obj)` den Wert 0, so wird `obj` zunächst schwarz markiert (Zeile 9f). Das bedeutet, dass `obj` nur

[2] Wie bereits eingangs der Arbeit erwähnt, wird im Folgenden auf die Überprüfung von `null`-Referenzen verzichtet. Entsprechende **if**-Anweisungen lassen sich jedoch problemlos in die Algorithmen integrieren.

von Objekten aus erreichbar ist, die sich innerhalb der starken Zusammenhangskomponente befinden. Infolgedessen ist die Erreichbarkeit von `obj` von den anderen Objekten der Komponente abhängig: Besitzen diese ebenfalls nur interne Referenzen, ist die gesamte Komponente – und damit auch `obj` – unerreichbar. Um dies zu überprüfen, wird *scan* rekursiv für alle Objekte von ⟨`obj`⟩ aufgerufen (Zeile 11f).

Besitzt hingegen ein grau markiertes Objekt `a` einen Referenzzähler mit einem Wert ungleich 0, so existiert eine externe Referenz auf dieses Objekt – `a` ist von einem Objekt aus erreichbar, das nicht zu ⟨`a`⟩ gehört. In diesem Fall wird die Prozedur *unmark* aufgerufen, die `a` weiß markiert (Zeile 2 in Algorithmus 3.5). Ebenso müssen jedoch auch alle Objekte weiß markiert werden, die von `a` aus erreichbar sind, weswegen *unmark* rekursiv für diese Objekte aufgerufen wird (Zeile 6). Dabei werden gleichzeitig die zugehörigen Referenzzähler angepasst, sodass diese nach Beendigung wieder mit der Anzahl aller Referenzen auf ein Objekt übereinstimmen (Zeile 4).

Algorithmus 3.5 Zyklische Referenzzählung – Aufräumphase (vgl. [MWL90, S. 33])

```
 1: unmark(obj):
 2:     setColor(obj, WHITE)                          ▷ Objekt ist erreichbar
 3:     for each ref ∈ POINTERS(obj)
 4:         rc(*ref) ← rc(*ref) + 1                   ▷ Interne Referenz wieder berücksichtigen
 5:         if not isWhite(*ref)
 6:             unmark(*ref)

 7: collect(obj):
 8:     if isBlack(obj)                               ▷ Objekt ist nicht erreichbar
 9:         for each ref ∈ POINTERS(obj)
10:             collect(*ref)
11:         free(obj)
```

Nach Abarbeitung von *scan*(`obj`) sind alle zuvor grau markierten Objekte, die von `obj` aus erreichbar sind, schwarz markiert, wenn sie nicht über externe Referenzen erreichbar sind, oder andernfalls weiß markiert und ihre Referenzzähler wiederhergestellt. Im letzten Schritt können daher alle schwarz markierten Objekte freigegeben werden, was erneut rekursiv durch die Prozedur *collect* geschieht.

Um die Arbeit des Algorithmus zu veranschaulichen, wird nun exemplarisch der Objektgraph aus Abbildung 3.3 betrachtet und angenommen, dass die Referenz `a` → `b` entfernt wird, wodurch die Objekte `b`, `c`, `d` und `e` unerreichbar werden. Zu erwarten ist, dass der Aufruf von *markGray*(`b`), *scan*(`b`) und *collect*(`b`) alle vier Objekte freigibt.

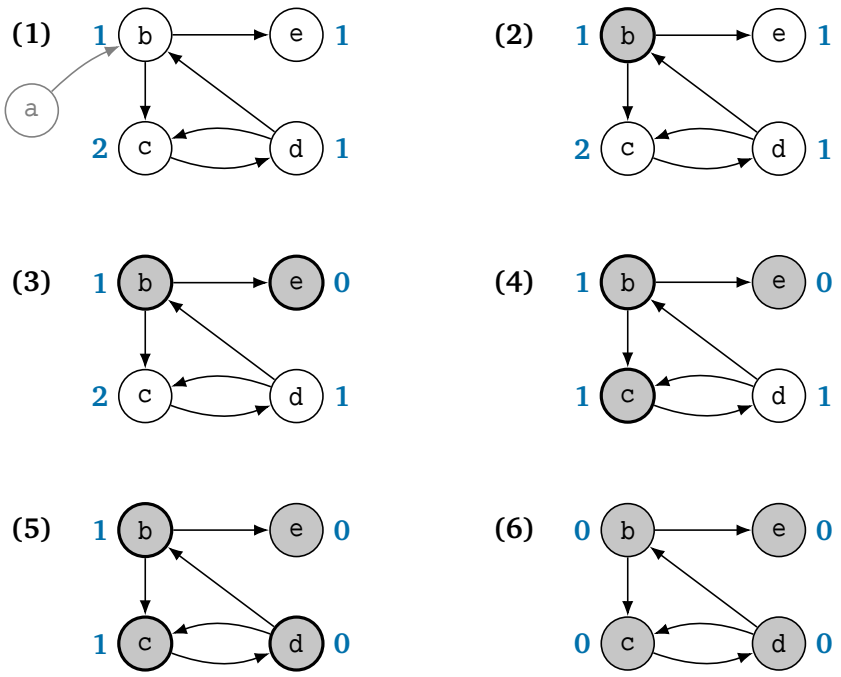


Abbildung 3.4.: Beispielhafte Ausführung von *markGray*(b) nach Entfernen der Referenz a → b. Dick umrandet sind diejenigen Objekte, für welche aktuell der rekursive Aufruf von *markGray* abgearbeitet wird.

Zunächst wird durch *markGray* Objekt b grau markiert (siehe Abbildung 3.4). Im Anschluss erfolgt für jedes von b referenzierte Objekt – hier e und c – eine Verringerung des zugehörigen Referenzzählers sowie ein rekursiver Aufruf von *markGray*. Der Aufruf *markGray*(c) erniedrigt den Referenzzähler von d; *markGray*(d) sorgt für eine erneute Modifikation von rc(b) und rc(c). An dieser Stelle geschehen jedoch keine rekursiven Aufrufe mehr, da alle von d referenzierten Objekte bereits grau markiert sind. Somit terminieren hier die Aufrufe von *markGray*(d), *markGray*(c) und *markGray*(b).

Zu diesem Zeitpunkt ist bereits erkennbar, dass alle Referenzzähler von ⟨b⟩ 0 sind, weswegen die starke Zusammenhangskomponente keine externen Referenzen besitzt und nicht mehr erreichbar ist. Dies wird nun von *scan* mittels einer weiteren Tiefensuche beginnend bei b überprüft. Da alle von b aus erreichbaren Objekte grau markiert sind und ihre Referenzzähler verschwinden, werden sie schwarz markiert; ein Aufruf von *unmark* findet nicht statt. Somit werden anschließend alle vier Knoten durch *collect* mittels einer dritten Tiefensuche freigegeben.

Was passiert jedoch, wenn der Zyklus erreichbar bleibt? Dazu wird das Beispiel um eine externe Referenz auf das Objekt d ergänzt (siehe Abbildung 3.5). Nach Terminierung von *markGray*(b) weist rc(d) dann den Wert 1 auf. Durch den Aufruf *scan*(b) werden zunächst b, e und c schwarz markiert. Da allerdings rc(d) $\neq 0$ gilt, folgt nun ein Aufruf von *unmark*(d), wodurch d weiß markiert wird. Hierdurch wird eine weitere Tiefensuche angestoßen, die die von d aus erreichbaren Objekte

traversiert, weiß markiert und ihre Referenzzähler zurücksetzt. Der anschließend noch abzuarbeitende Aufruf von *scan*(b) markiert lediglich graue Objekte schwarz. Somit sind letztlich keine schwarz markierten Objekte vorhanden und kein Objekt der starken Zusammenhangskomponente wird entfernt.

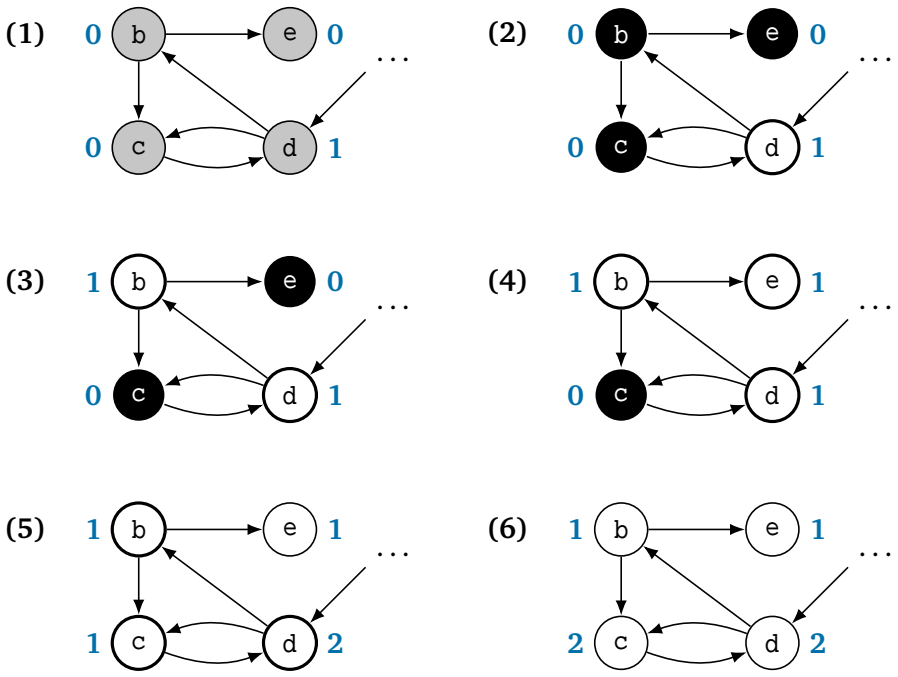


Abbildung 3.5.: Beispielhafte Ausführung von *unmark*(d) nach *markGray*(b).

Wir zeigen nun, dass die zyklische Referenzzählung nach MARTÍNEZ et al. korrekt ist, wobei weiterhin vorausgesetzt wird, dass der Algorithmus atomar ausgeführt wird.

Satz 3.2 (Korrektheit der zyklischen Referenzzählung)**:**
Die zyklische Referenzzählung nach MARTÍNEZ et al. ist korrekt und terminiert.

Beweis: Wir zeigen zunächst, dass die Aufrufe der Prozeduren *markGray*, *scan* und *collect* (Zeile 8 bis 10 in Algorithmus 3.3) terminieren: Für *markGray* ist dies leicht zu sehen, da *markGray* lediglich weiß markierte Objekte manipuliert und diese sofort grau färbt (Zeile 2f in Algorithmus 3.4). Analoges gilt für *collect* und schwarz gefärbte Objekte. *scan* wiederum verändert nur grau markierte Objekte. Ist der Referenzzähler eines Objektes 0, für welches *scan* aufgerufen wird, so wird dieses schwarz gefärbt. Andernfalls wird das Objekt durch *unmark* weiß markiert. In beiden Fällen ist damit sichergestellt, dass ein Objekt nicht mehrfach durch *scan* bzw. *unmark* bearbeitet wird und keine weiteren rekursiven Aufrufe stattfinden. Somit terminieren alle drei Aufrufe.

Wird nun ein Objekt a durch *collect* freigegeben, so wurde a zuvor schwarz markiert (Zeile 8 und 11 in Algorithmus 3.5). Dies passiert ausschließlich in der Prozedur *scan*, wenn a keine externen Referenzen besitzt. Wäre a erreichbar, so müsste folglich ein Objekt $b \in \langle a \rangle$ existieren, das ebenfalls erreichbar ist, also eine externe Referenz besitzt. In diesem Fall wird b jedoch durch *unmark* weiß markiert und rekursiv auch a, da $b \xrightarrow{*} a$ gilt. Das ist jedoch ein Widerspruch, da weiß markierte Objekte nach Terminierung von *markGray* nicht mehr schwarz markiert werden. Somit kann a nicht erreichbar sein. □

Der vorgestellte Algorithmus löst zwar das Problem brachliegender zyklischer Strukturen, allerdings zu einem hohen Preis. Jede Referenzlöschung, die einen Referenzzähler nicht auf 0 fallen lässt, löst eine Überprüfung auf zyklische Referenzen aus, die potenziell viele Manipulationen von Referenzzählern mit sich bringt. Im schlimmsten Fall wird dabei der gesamte Objektgraph drei Mal traversiert, ohne dass am Ende Speicher freigegeben wurde. Der Algorithmus eignet sich daher eher für Fälle, in denen mehrere Referenzen auf ein Objekt selten sind, aber weniger für objektorientierte Konzepte (vgl. [Lin92, S. 215]). Zyklische Referenzzählung kann somit unter Umständen wesentlich aufwendiger sein als der naive Mark-Sweep-Algorithmus. Als Verbesserung hat LINS eine Variante vorgestellt, bei welcher die für eine zyklische Struktur infrage kommenden Objekte zunächst in eine Warteschlange eingefügt werden. Die Überprüfung auf zyklische Referenzen wird damit erst bei Bedarf ausgelöst, etwa bei akutem Speichermangel oder voller Warteschlange. Falls zuvor bereits die letzte Referenz auf ein Objekt entfernt wird, kann das Verfahren übersprungen werden [Lin92]. Von LIN und HOU stammt darüber hinaus eine Variante, die mehrere zyklische Strukturen zu Teilgraphen eines Objektgraphen zusammenfasst. Mittels einer einzigen Traversierung kann dann die Zahl der externen Referenzen dieses Teilgraphen bestimmt werden; existieren keine, so kann der gesamte Teilgraph – und damit alle enthaltenen starken Zusammenhangskomponenten – freigegeben werden [LH06].

3.3 Optimierungsmöglichkeiten

Abschließend werden zwei Ansätze betrachtet, die eine Effizienzsteigerung der Referenzzählung bezwecken sollen. In Abschnitt 3.1 wurde bereits konstatiert, dass Löschkaskaden, die durch freigegebene Objekte entstehen, zu unerwarteten Programmunterbrechungen führen können und die Buchhaltung jeder einzelnen Referenzmanipulation enormen Overhead erzeugt. Von DEUTSCH und BOBROW stammt der Ansatz der **verzögerten Referenzzählung** (engl. *deferred reference counting*), der die Freigabe verwaister Objekte verzögert und auf eine Verfolgung von Referenzmanipulationen verzichtet, die in Feldern von Basisobjekten stattfinden [DB76]. Der

Referenzzähler eines Objekts wird dazu nicht im Header gespeichert. Stattdessen werden zwei Tabellen – die *zero count table* `ZCT` und die *multi reference table* `MRT` – angelegt. Erstere enthält alle Objekte, deren Referenzzähler 0 ist. Folglich werden diese nicht von anderen Objekten des Heaps referenziert, aber möglicherweise von Basisobjekten wie lokalen Variablen. Die eingangs des Kapitels formulierte notwendige Bedingung in Lemma 3.1 ist damit außer Kraft gesetzt. Die `MRT` enthält wiederum alle Objekte, die mehrfach von Heapobjekten referenziert werden, und weist diesen ihre Referenzzähler zu, was etwa mit einer Hashtabelle realisiert werden kann. Entsprechend wird ein Objekt genau einmal von einem Heapobjekt referenziert, wenn es sich weder in `ZCT` noch `MRT` befindet.

Algorithmus 3.6 Verzögerte Referenzzählung nach DEUTSCH und BOBROW (vgl. [DB76, S. 523f]).

```
 1: writeRef(obj, i, ref):
 2:   if obj ∉ ROOTS                    ▷ Schreibbarriere für Basisobjekte aussetzen
 3:     atomic
 4:       incRefCount(*ref)
 5:       decRefCount(*obj[i])
 6:   obj[i] ← ref

 7: incRefCount(obj):
 8:   if obj ∈ ZCT
 9:     remove(ZCT, obj)                ▷ Anzahl Referenzen ist nun 1
10:   else if obj ∈ MRT
11:     MRT(obj) ← MRT(obj) + 1         ▷ Referenzzähler erhöhen
12:   else add(MRT, obj)                ▷ Anzahl Referenzen ist nun 2

13: decRefCount(obj):
14:   if obj ∈ MRT
15:     if MRT(obj) > 2
16:       MRT(obj) ← MRT(obj) − 1       ▷ Referenzzähler erniedrigen
17:     else remove(MRT, obj)           ▷ Anzahl Referenzen ist nun 1
18:   else add(ZCT, obj)                ▷ Anzahl Referenzen ist nun 0

19: atomic collect():
20:   for each ref ∈ POINTERS(ROOTS)    ▷ nicht gezählte Referenzen hinzufügen
21:     incRefCount(*ref)
22:   while ZCT ≠ ∅                     ▷ nicht erreichbare Objekte löschen
23:     obj ← remove(ZCT)
24:     for each ref ∈ POINTERS(obj)
25:       decRefCount(*ref)
26:     free(obj)
27:   for each ref ∈ POINTERS(ROOTS)    ▷ vorige Anpassung korrigieren
28:     decRefCount(*ref)
```

Wird nun eine Referenz `ref` in ein Feld `obj[i]` geschrieben, kann der Schreibzugriff unsynchronisiert ausgeführt werden, sofern `obj` ein Basisobjekt ist, da keine Referenzzähler angepasst werden (Zeile 2 in Algorithmus 3.6). Andernfalls sind mehrere

Fälle zu unterscheiden: Wird ein Referenzzähler eines Objekts `obj` erhöht, das sich in `ZCT` befindet (etwa, da es bislang nur von lokalen Variablen referenziert wird), so muss es aus `ZCT` entfernt werden, da der Referenzzähler nun 1 beträgt (Zeile 8f). Ist es in der `MRT`, so ist der zugehörige Referenzzähler `MRT(obj)` zu erhöhen (Zeile 10f). Ist beides nicht der Fall, so muss `obj` zur `MRT` hinzugefügt werden; der Referenzzähler wird entsprechend mit 2 initialisiert (Zeile 12). Analoge Fallunterscheidungen sind nötig, wenn ein Referenzzähler reduziert wird (Zeile 14 bis 18).

Die Freigabe von Objekten kann nun zu geeigneten, unkritischen Zeitpunkten mittels der Prozedur *collect* ausgelöst werden, sofern diese vorhersehbar sind, oder alternativ bei akutem Speichermangel. Da die Objekte in der `ZCT` gegebenenfalls noch von lokalen Variablen referenziert werden und in diesem Fall nicht entfernt werden dürfen, müssen zunächst ihre Referenzzähler angepasst werden, indem über alle Referenzen von Basisobjekten iteriert wird (Zeile 20f). Objekte, die im Anschluss in der `ZCT` verbleiben, sind tatsächlich verwaist und können bedenkenlos entfernt werden, ohne die Korrektheit des Algorithmus zu gefährden (Zeile 22 bis 26).

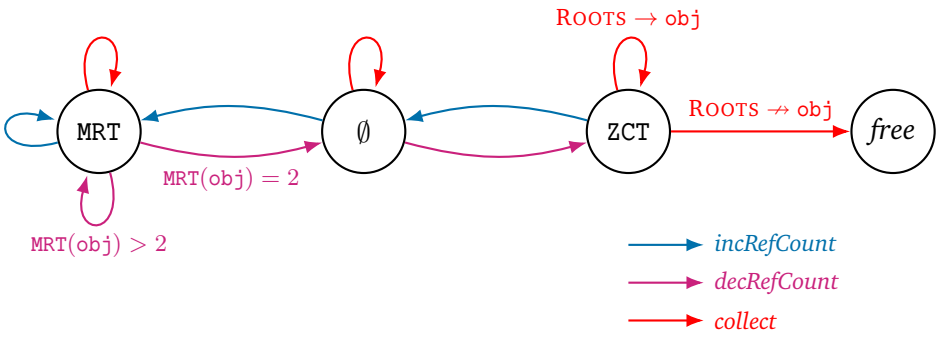


Abbildung 3.6.: Schema zur Veranschaulichung der verzögerten Referenzzählung.

Abbildung 3.6 veranschaulicht die Arbeitsweise des Algorithmus und die Verschiebung eines Objekts `obj` in bzw. aus `ZCT` und `MRT`. Der maßgebliche Performancegewinn entsteht dadurch, dass Referenzmanipulationen in lokalen Variablen nicht unmittelbar, sondern erst während einer Bereinigungsphase vermerkt werden. Somit wirken sich die durch die Garbage Collection bedingten Schreibbarrieren nicht auf Berechnungen aus, die vorwiegend auf lokale Variablen zugreifen. Dies kommt etwa Iterationen über Datenstrukturen zu Gute. Je nach Implementation der `ZCT` können außerdem Löschkaskaden, die während einer Kollektionsphase entstehen können, auf den nächsten Kollektionszyklus verschoben werden, um Unterbrechungen minimal zu halten. Ein weiterer Vorteil ergibt sich für Anwendungen, in denen die meisten Objekte höchstens einmal referenziert werden und sich zu keinem Zeitpunkt in der `MRT` befinden. Für diese entfällt die Notwendigkeit, Referenzzähler zu hinterlegen und für die `MRT` muss weniger Speicher reserviert werden, was den Speicherbedarf des gesamten Verfahrens reduziert. Für Anwendungen, in denen diese Bedingung nicht zutrifft, ergibt sich jedoch die Problematik, die `MRT` angemessen

zu realisieren. Im Falle einer Hashtabelle muss etwa eine geeignete Hashfunktion gewählt werden, um Schlüsselkollisionen und damit verbundene Performanceeinbußen zu vermeiden.[3]

Eine weitere Optimierungsmöglichkeit zielt darauf ab, die Auswirkung einer mehrfachen Änderung eines Referenzfeldes eines Objekts zusammenzufassen, anstatt jede einzelne Änderung nachzuverfolgen. LEVANONI und PETRANK beobachten, dass viele einzelne Referenzmanipulationen innerhalb desselben Feldes unnötige Anpassungen von Referenzzählern verursachen, die sich gewissermaßen *kürzen* lassen [LP06, S. 4ff]. Man betrachte etwa einen Mutator, der folgenden Code ausführt, und die dadurch ausgelösten Anpassungen von Referenzzählern:

	Referenzzähleranpassungen:	
`obj.ref` $\leftarrow$ `&a`$_0$		
`obj.ref` $\leftarrow$ `&a`$_1$	*incRefCount*(`a`$_1$)	*decRefCount*(`a`$_0$)
`obj.ref` $\leftarrow$ `&a`$_2$	*incRefCount*(`a`$_2$)	*decRefCount*(`a`$_1$)
`obj.ref` $\leftarrow$ `&a`$_3$	*incRefCount*(`a`$_3$)	*decRefCount*(`a`$_2$)
...	...	
`obj.ref` $\leftarrow$ `&a`$_n$	*incRefCount*(`a`$_n$)	*decRefCount*(`a`$_{n-1}$)

Offensichtlich wird die Erhöhung der Referenzzähler von `a`$_1$ bis `a`$_{n-1}$ durch die jeweils nächste Anweisung des Mutators rückgängig gemacht, sodass es sinnvoll ist, diese Manipulationen zu aggregieren und lediglich die Zähler von `a`$_0$ und `a`$_n$ anzupassen. Die **aggregierte Referenzzählung** (engl. *coalesced reference counting*) realisiert dies, indem in regelmäßigen Abständen überprüft wird, welche Referenzfelder von Objekten seit der letzten Überprüfung verändert wurden. Dazu wird im Rahmen einer Schreibbarriere ein Objektfeld als modifiziert gekennzeichnet – etwa durch Setzen eines Bits im Header des Objekts – und die Adresse des Objektfeldes zusammen mit dem alten Inhalt vor der ersten Modifikation in einer Tabelle `log` hinterlegt (Zeile 3f in Algorithmus 3.7). Wird nun über die in `log` eingetragenen Felder iteriert, so wird geprüft, ob sich die neue im Feld gespeicherte Referenz `*field` von der zwischengespeicherten Referenz `log(field)` unterscheidet (Zeile 8). Nur dann sind Anpassungen der jeweiligen Referenzzähler nötig. Anschließend wird die Markierung des Feldes zurückgesetzt und der Eintrag aus der Tabelle entfernt.

Mittels Methoden für verteilte Systeme, auf die hier nicht näher eingegangen wird, lässt sich dieser Ansatz durch feingranularere Synchronisationen optimieren, etwa indem Logtabellen threadweise angelegt werden. Dadurch kann vermieden werden,

[3] Kommt es zu vielen Schlüsselkollisionen, kann die Komplexität eines Zugriffs auf ein einzelnes Tabellenelement in $\mathcal{O}(n)$ liegen, wobei n die Anzahl der Einträge ist. Eine Vergrößerung der Schlüsselmenge zur Vermeidung von Kollisionen kann wiederum zu einem hohen Speicherbedarf führen, was den für den Heap zur Verfügung stehenden Speicher einschränkt (vgl. [Cor+09, S. 253]).

dass das gesamte Programm angehalten wird. Für mehr Details sei auf die Publikation von LEVANONI und PETRANK verwiesen [LP06, S. 19].

Algorithmus 3.7 Aggregierte Referenzzählung nach LEVANONI und PETRANK (vgl. [LP06, S. 14ff]).

```
 1: atomic writeRef(obj, i, ref):
 2:     if not isModified(&obj[i])
 3:         addLog(&obj[i], obj[i])          ▷ Speichere alten Feldinhalt in Tabelle
 4:         setModified(&obj[i])             ▷ Markiere Feld als verändert
 5:     obj[i] ← ref

 6: atomic processLog():
 7:     for each field ∈ log
 8:         if *field ≠ log(field)           ▷ Prüfe, ob Feldinhalt verändert wurde
 9:             incRefCount(**field)         ▷ vgl. Algorithmus 3.6
10:             decRefCount(*log(field))     ▷ vgl. Algorithmus 3.6
11:         unsetModified(field)             ▷ Markierung entfernen
12:         removeLog(field)                 ▷ Entferne Eintrag aus Tabelle
13:     collect()                            ▷ vgl. Algorithmus 3.6
```

Mit dem Mark-Sweep-Algorithmus und der Referenzzählung haben wir nun zwei zunächst diametral verschiedene Ansätze zur automatischen Speicherverwaltung kennen gelernt. Die zuletzt eingeführten Optimierungsmöglichkeiten, in denen die Referenzzählung von Kollektionsphasen begleitet wird, lassen bereits erahnen, dass beide Verfahren nicht gänzlich unvereinbar sind. Bevor dieser Gedanke weiter verfolgt wird, erfolgt im nächsten Kapitel die Auseinandersetzung mit dem Phänomen der Speicherfragmentierung, das beide Ansätze betrifft und bisher nicht zur Sprache gekommen ist.

4 Kompaktierung

Mit dem Mark-Sweep-Algorithmus und der Referenzzählung sind in den vorigen Kapiteln zwei grundlegende Ansätze eingeführt worden, die eine Wiederverwendung bereits genutzten Speichers durch Freigabe nicht mehr benötigter Objekte realisieren. Außen vor gelassen wurde bislang jedoch ein Phänomen, das mit zunehmender Laufzeit eines Programms in Erscheinung tritt: die **Fragmentierung** des Heaps. Je häufiger eine Garbage Collection zum Einsatz kommt, umso wahrscheinlicher ist es, dass die erreichbaren Objekte keinen zusammenhängenden Speicherbereich mehr bilden (siehe Abbildung 4.1). In der Folge ist auch der freie Speicher in mehrere unzusammenhängende Teile unterschiedlicher Größe aufgeteilt. Dies kann sich nachteilig auf die Performance einer Anwendung auswirken: Bildet der freie Speicher einen möglichst großen zusammenhängenden Bereich, so können viele aufeinanderfolgende Speicheranforderungen in hoher Geschwindigkeit erfüllt werden. Andernfalls benötigen Speicherallokationen mehr Zeit, da eine hinreichend große *Lücke* gefunden werden muss, die die angeforderte Speichermenge aufnehmen kann. Im schlimmsten Fall schlägt die Allokation fehl, da keine geeignete Lücke gefunden werden kann, obwohl in der Summe genügend freier Speicher vorhanden wäre. Dies wiederum führt zu weiteren Auslösungen der Garbage Collection, was die Performance weiter beeinträchtigt. Zudem benötigt eine Anwendung mit stark fragmentiertem Heap einen größeren Teil des gesamten Arbeitsspeichers. Eine Kompaktierung lebendiger Objekte kann ferner die räumliche Lokalität verbessern.

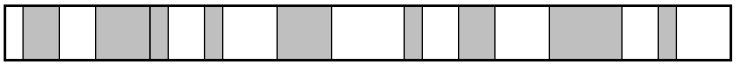


Abbildung 4.1.: Ein stark fragmentierter Heap erschwert die Allokation größerer Speichermengen, auch wenn ein Großteil des Heaps ungenutzt ist.

Die Vermeidung von Heapfragmentierung durch optimale Allokation ist nicht zielführend, da die Prognose zukünftiger Allokationen in der Regel unmöglich und das Finden einer entsprechenden Allokationsstrategie NP-schwer ist (vgl. [Rob80]). Allerdings existieren akzeptable Lösungen, die die Fragmentierung des Heaps durch geschickte Allokation zumindest in Grenzen halten.[1] Nichtsdestoweniger werden in diesem Kapitel ausschließlich Algorithmen betrachtet, die den Heap im Rahmen einer Garbage Collection **kompaktieren**, da die Durchführung eines Kollektionszyklus ursächlich für die Entstehung von Fragmentierung ist.

[1] Für einen Überblick siehe etwa [JHM11, Kap. 7].

Algorithmen, die im Rahmen der Garbage Collection die Fragmentierung des Heaps beseitigen oder verhindern, lassen sich grob in zwei Kategorien einteilen: Die erste Kategorie der **Mark-Compact-Algorithmen** baut auf der Markierungsphase eines Mark-Sweep-Kollektors auf, indem aus den Markierungsinformationen die zukünftigen Positionen der Objekte generiert werden und die Bereinigung durch eine Kompaktierungsphase ersetzt werden kann. Im Folgenden werden mit dem *LISP-2-Algorithmus* und *The Compressor* zwei Exemplare dieser Gattung vorgestellt. Die zweite Kategorie der **kopierenden** Algorithmen – gelegentlich auch *Mark-Copy-Algorithmen* genannt – vereinen Markierungs- und Kompaktierungsphase, indem als erreichbar identifizierte Objekte unmittelbar an eine neue Position kopiert werden. In dieser Arbeit wird exemplarisch ein Algorithmus betrachtet, der den Heap in zwei Halbräume aufteilt.

4.1 LISP-2-Kompaktierung

Zunächst wird ein Algorithmus behandelt, der auf dem Mark-Sweep-Ansatz aufbaut und in seiner ursprünglichen Form für *LISP 2*[2] konzipiert wurde, weswegen er in der Literatur häufig als **LISP-2-Algorithmus** bezeichnet wird (vgl. [JHM11, Kap. 3.2] und [Sty67]). Die grundlegende Idee ist, die Bereinigungsphase nach der Markierung erreichbarer Objekte durch eine Kompaktierungsphase zu ersetzen, die alle markierten Objekte an den Beginn des Heaps verschiebt. Dabei werden verwaiste Objekte entweder überschrieben oder befinden sich anschließend außerhalb des kompaktierten Bereichs, sodass sie durch zukünftige Allokationen überschrieben werden können.

Wie schon in Kapitel 2 wird vorausgesetzt, dass eine Prozedur zur Verfügung steht, die die Adresse des nachfolgenden Objekts liefert, sofern existent. Entsprechend kann leicht eine Prozedur *nextMarkedObject* realisiert werden, die nur markierte Objekte liefert. Weiter soll eine Prozedur *moveObject*(`old`, `new`) existieren, die das Objekt an der Speicheradresse `old` an die Speicheradresse `new` verschiebt, wobei eine Überschreibung von Daten an der Zielposition ausdrücklich zugelassen ist.

Der Algorithmus verfährt nun recht intuitiv (siehe Abbildung 4.2): Beginnend am Anfang des Heaps werden der Reihe nach alle markierten Objekte besucht und an die vorderste Stelle bewegt, an der noch kein markiertes Objekt steht. Dabei wird für jedes verschobene Objekt die alte und neue Speicheradresse in einer Tabelle `log` hinterlegt (Zeile 6 in Algorithmus 4.1). Nach jeder Verschiebung wird der Zeiger `pos` auf das neue Ende des kompaktierten Bereichs und `next` auf das nächste markierte

[2] LISP 2 war als Nachfolger von LISP gedacht und sollte einige Einschränkungen beheben, die bereits von McCarthy aufgeführt wurden. Aus Kostengründen wurde die Entwicklung von LISP 2 letztlich eingestellt (vgl. [McC79]).

Objekt gesetzt (Zeile 8f). Die erreichbaren Objekte befinden sich nun lückenlos am Anfang des Heaps.

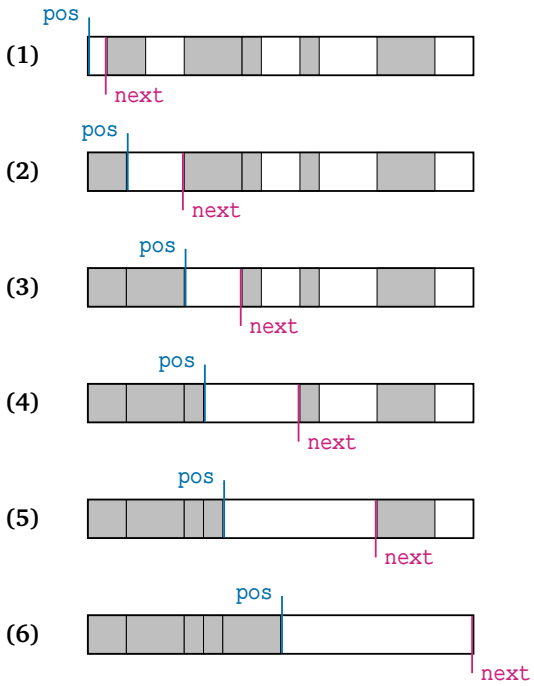


Abbildung 4.2.: Veranschaulichung des LISP-2-Algorithmus.

Algorithmus 4.1 LISP-2-Kompaktierung (vgl. [JHM11, S. 35] und [Sty67, S. 7ff]).

```
 1: atomic compact():
 2:     pos ← HEAP_START                    ▷ Zeiger auf Ende des kompaktierten Bereichs
 3:     next ← nextMarkedObject(HEAP_START)  ▷ Zeiger auf nächstes Objekt
 4:     while next ≠ null
 5:         if pos ≠ next                   ▷ Tue nichts, wenn noch im defragmentierten Bereich
 6:             addLog(next, pos)           ▷ Notiere alte und neue Adresse
 7:             moveObject(next, pos)       ▷ Verschiebe Objekt im Speicher
 8:         pos ← pos + sizeOf(*pos)        ▷ Zeiger weitersetzen
 9:         next ← nextMarkedObject(next)
10:     updateRefs()

11: updateRefs():
12:     for each field ∈ PointerFields(Roots)
13:         if *field ∈ log                 ▷ Felder von Basisobjekten anpassen
14:             *field ← log(*field)
15:     pos ← nextMarkedObject(HEAP_START)  ▷ Felder von Heapobjekten anpassen
16:     while pos ≠ null
17:         for each field ∈ PointerFields(*pos)
18:             if *field ∈ log
19:                 *field ← log(*field)
20:         pos ← nextMarkedObject(pos)
21:     clear(log)                          ▷ Tabelle aufräumen
```

Im Anschluss müssen alle Referenzen auf verschobene Objekte in den Feldern aller Objekte aktualisiert werden. Die Prozedur *updateRefs* iteriert dabei zunächst über alle Felder von Basisobjekten (Zeile 12 bis 14). Falls der Inhalt `*field` eines Felds als Schlüssel in der Tabelle auftaucht, so handelt es sich um die alte Adresse eines verschobenen Objekts. Entsprechend muss der Feldinhalt auf die neue Adresse `log(*field)` gesetzt werden. Um auch die Objekte des Heaps anzupassen, wird dieser ein weiteres Mal linear traversiert und analog die Felder aller markierten Objekte angepasst (Zeile 15 bis 20). Zuletzt wird die Tabelle mit den zwischengespeicherten Adressen geleert. Dadurch, dass der Heap in aufsteigender Reihenfolge durchlaufen wird, die Objekte jedoch in entgegengesetzte Richtung verschoben werden, werden keine lebendigen Daten überschrieben und der Algorithmus arbeitet korrekt.

Die Laufzeit des gesamten Algorithmus lässt sich wie folgt abschätzen: Im schlimmsten Fall läuft der Zeiger `pos` in der Prozedur *compact* über den gesamten Heap. Davon ausgehend, dass das Hinzufügen eines Adresspaares zur Tabelle `log` in $\mathcal{O}(1)$ möglich ist (siehe auch Abschnitt 3.3), ergibt sich dadurch eine Komplexität von $\mathcal{O}(|\mathbb{H}|)$ für die Verschiebung der erreichbaren Objekte an den Anfang des Heaps. Für die Aktualisierung der Referenzen ergibt sich die Laufzeit aus der Anzahl der Felder der Basisobjekte sowie einer weiteren Traversierung über den Heap. Somit können folgende Abschätzungen festgehalten werden:

Satz 4.1 (Komplexität des LISP-2-Algorithmus)**:**
Für den LISP-2-Algorithmus gelten folgende Eigenschaften:

(1) Die Laufzeit der Verschiebungsphase liegt in $\mathcal{O}(|\mathbb{H}|)$.

(2) Die Laufzeit von *updateRefs* liegt in $\mathcal{O}\left(\sum_{a \in \text{Roots}} |\text{PointerFields}(a)| + |\mathbb{H}|\right)$.

Der Speicherplatzbedarf, der zusätzlich durch die Kompaktierung anfällt, ist im Wesentlichen durch die Realisierung der Tabelle `log` bestimmt. Alternativ – und näher am ursprünglichen Algorithmus – kann zur Speicherung der neuen Adresse eines Objekts auch ein zusätzliches Feld im Header jedes Objekts eingerichtet werden. In diesem Fall ist allerdings ein weiterer Durchlauf über den Heap notwendig, da die Referenzen in den Feldern aller Objekte anpasst werden müssen, bevor die Objekte an ihre neue Position verschoben werden können (vgl. [MUI13, S. 16]).

Ein wesentlicher Nachteil des Algorithmus ist, dass er selbst bei geringer Fragmentierung des Heaps viele Speicheroperationen benötigt. So werden etwa auch dann fast alle Objekte verschoben, wenn nur ein kleiner Bereich zu Beginn des Heaps fragmentiert ist. Im Falle von großen Heaps mit vielen Objekten ist die Ausbeute der Kompaktierung gering, der Aufwand zur Aktualisierung von Speicheradressen allerdings sehr hoch. Insofern sollte abgewägt werden, ob die Ausführung bei

jedem Garbage-Collection-Zyklus zielführend ist. PRINTEZIS präsentiert beispielsweise ein heuristisches Kriterium und eine Umsetzung für Java, um zur Laufzeit zu entscheiden, ob eine Kompaktierung des Heaps erfolgen sollte. Wenn der Heap vergrößert wird oder aufeinanderfolgende Speicherallokationen nicht mehr schnell genug erfüllt werden können, ist eine Kompaktierung sinnvoll, andernfalls genügt der Mark-Sweep-Algorithmus (vgl. [Pri01, Kap. 3.4]). Andere Algorithmen vermeiden diese Problematik, indem sie den Heap von beiden Seiten traversieren und hinten stehende Objekte in freie Speicherbereiche im vorderen Teil des Heaps verschieben (vgl. [JHM11, S. 32f]). Dieses Vorgehen hat jedoch den entscheidenden Nachteil, dass die Reihenfolge der Objekte im Heap zerstört wird. Damit wird die Arbeit von Allokatoren, die verwandte Objekte zur Optimierung der räumlichen Lokalität möglichst benachbart anlegen, zunichte gemacht.

4.2 Compressor-Algorithmus

Der von KERMANY und PETRANK vorgestellte **Compressor-Algorithmus** zielt darauf ab, die durch die Kompaktierung entstehenden Verzögerungen vor allem für größere Heaps zu minimieren, indem in einem einzigen Durchlauf Objekte an ihre neue Speicheradresse verschoben und sämtliche Referenzen angepasst werden [KP06]. Dazu wird davon ausgegangen, dass der Heapspeicher in gleich große Blöcke eingeteilt ist und Objekte an *Wörtern* ausgerichtet angelegt werden.[3] Während der Markierungsphase kann dann ein Bitvektor erzeugt werden, der angibt, ob ein Wort durch ein erreichbares Objekt belegt ist (siehe Abbildung 4.3). Alle weiteren Berechnungen können dann mithilfe des Bitvektors realisiert werden, anstelle auf den Heap zuzugreifen.

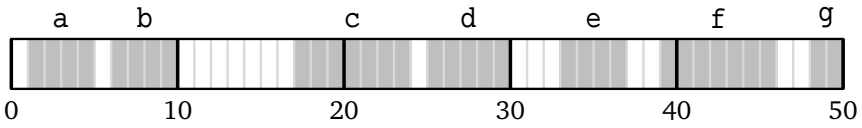


Abbildung 4.3.: Ein Bitvektor speichert zu jedem Wort des Heaps, ob es durch ein erreichbares Objekt belegt wird. Dargestellt ist ein Heapausschnitt mit 5 Blöcken à 10 Wörtern.

Zu Beginn der Kompaktierungsphase wird durch die Prozedur *calculateOffsets* in Algorithmus 4.2 ein Vektor `offset` angelegt, der für jeden Block $\texttt{blk}_k$ angibt, an welche Speicheradresse das erste Objekt des Blocks verschoben werden muss. Diese ergibt sich aus dem Offset des vorigen Blocks $\texttt{blk}_{k-1}$ und dem Speicherplatz, den die Objekte belegen, deren Beginn im Block $\texttt{blk}_{k-1}$ liegt. Diese Information kann durch Zählung der korrespondierenden Bits im Bitvektor erhalten werden. Die neue

[3] KERMANY und PETRANK gehen exemplarisch von einer Wortbreite von 4 Bytes (32 Bit) und einer Blockgröße von 512 Bytes aus.

Speicheradresse eines Objekts kann nun wie folgt anhand der alten Speicheradresse `adr` ermittelt werden (vgl. Prozedur *newAddress*): Zunächst wird anhand des Blocks, in dem sich die Speicheradresse befindet, der Offset bestimmt. Die genaue Zieladresse ergibt sich nun durch den Speicherplatz der Objekte, deren Beginn im gleichen Block liegt, sich aber vor `adr` befinden (Zeile 8f). Dies kann ebenfalls durch Zählung der belegten Bits erfasst werden. Somit ist sichergestellt, dass die Reihenfolge der Objekte im Heap erhalten bleibt und keine erreichbaren Objekte zerstört werden.

Algorithmus 4.2 Der Compressor-Algorithmus nach KERMANY und PETRANK ([KP06]).

```
 1: calculateOffsets():
 2:     offset(0) ← HEAP_START
 3:     for k ∈ {1, ..., |BLOCKS| − 1}
 4:         offset(k) ← offset(k − 1) + usedWords(k − 1)

 5: newAddress(adr):
 6:     result ← offset(block(adr))          ▷ Ermittle Offset des zugehörigen Blocks
 7:     for each obj ∈ block(adr)
 8:         if &obj < adr                    ▷ Addiere Anzahl Wörter, die vor adr belegt sind
 9:             result ← result + sizeOf(obj)
10:     return result

11: compactAndUpdate():
12:     for each field ∈ POINTERFIELDS(ROOTS)
13:         *field ← newAddress(*field)
14:     pos ← nextMarkedObject(HEAP_START)
15:     while pos ≠ null                     ▷ Passe Objektfelder an
16:         for each field ∈ POINTERFIELDS(*pos)
17:             *field ← newAddress(*field)
18:         moveObject(pos, newAddress(pos))
19:         pos ← nextMarkedObject(pos)
```

Betrachten wir beispielhaft, wie die neuen Speicheradressen der Objekte `a` bis `d` aus Abbildung 4.3 bestimmt werden, nachdem der Offsetvektor erstellt wurde (Abbildung 4.4). Für Objekt `a` wird zunächst der Offset 0 nachgeschlagen, da sich `a` im ersten Block befindet. Da `a` das erste Element des Blocks ist, ist dieser Offset auch die neue Speicheradresse von `a`. Für `b` wird ebenfalls Offset 0 ermittelt, allerdings wird dieser um 4 korrigiert, da sich `a` vor `b` befindet und vier Wörter belegt. Analog zu `a` wird für `c` die neue Speicheradresse 8 ermittelt, da `c` das einzige Element im zweiten Block ist. Für `d` ergibt sich als Offset und Speicheradresse der Wert 15: Zwar befindet sich auch der hintere Teil von `c` im dritten Block, allerdings wurde die Größe von `c` bereits bei der Offsetberechnung berücksichtigt, sodass der Offset hier nicht korrigiert werden muss.

Der Performancegewinn durch den Verzicht auf eine zweite Heaptraversierung ist offenkundig, wird jedoch mit zusätzlichem Speicherbedarf erkauft. Der Bitvektor benötigt ein Bit Speicherplatz pro Wort, während der Offsetvektor pro Block ein

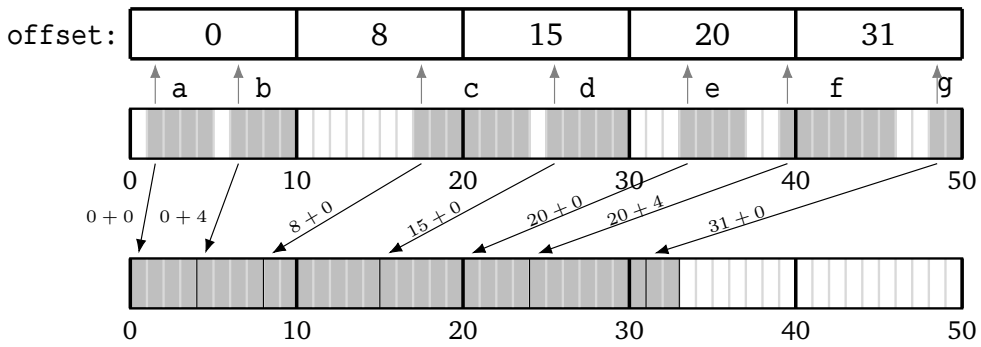


Abbildung 4.4.: Beispiel zum Compressor-Algorithmus. Die neue Speicheradresse eines Objekts berechnet sich aus dem Offset des zugehörigen Blocks (erster Summand) und dem Speicherplatz, der durch die davorliegenden Objekte im Block beansprucht wird (zweiter Summand).

Wort belegt. Für die von KERMANY und PETRANK vorgeschlagene Wortbreite und Blockgröße ergibt sich daher ein Overhead von etwa 4% des Heaps. Eine Erhöhung der Wortbreite w, an der Objekte im Heap ausgerichtet werden, kann diesen Anteil zwar reduzieren (siehe Tabelle 4.5), aber auch eine ineffizientere Speichernutzung verursachen, da jedes Objekt stets ein Vielfaches von w an Speicher belegt. Größere Blöcke wiederum erhöhen den Aufwand zur Berechnung der neuen Speicheradressen, da eine höhere Anzahl an Objekten pro Block zu erwarten ist.

$b \downarrow \diagdown w \rightarrow$	**16**	**32**	**64**	**128**
256	7.0%	4.7%	4.7%	7.0%
512	6.6%	3.9%	3.1%	3.9%
1024	6.4%	3.5%	2.3%	2.3%
2048	6.3%	3.2%	2.0%	1.6%
4096	6.3%	3.2%	1.8%	1.2%

Abbildung 4.5.: Verhältnis M des Speicherbedarfs des Compressor-Algorithmus zum gesamten Heap in Abhängigkeit von Wortbreite w in Bit und Blockgröße b in Byte. Es gilt $M = \frac{1}{w} + \frac{w}{8 \cdot b}$.

Die Korrektheit der beiden vorgestellten Mark-Compact-Algorithmen ergibt sich aus der Korrektheit der Markierungsphase. Wurden alle erreichbaren Objekte erfasst, werden auch die Zeiger `pos` und `next` im LISP-2-Algorithmus korrekt weitergesetzt, ohne dass es zur Überschreibung erreichbarer Objekte kommt. Mit gleichem Argument sind auch die berechneten Speicheradressen im Compressor-Algorithmus korrekt.

4.3 Kopierende Garbage Collection

Die zweite Klasse der Algorithmen, die eine Kompaktierung des Heaps erreichen, bilden die **kopierenden** Algorithmen. Hier werden Markierungs- und Kompaktierungsphase vereint: Analog zu Mark-Sweep-Ansätzen werden Referenzen verfolgt, um erreichbare Objekte zu identifizieren. Im Gegensatz zu den in den vorigen beiden Abschnitten vorgestellten Mark-Compact-Algorithmen werden erreichbare Objekte jedoch nicht durch eine zusätzliche lineare Traversierung des Heaps zusammengefasst, sondern unmittelbar bei Entdeckung an eine neue Position kopiert. Dazu wird der Heap in zwei Halbräume (engl. *semispaces*) aufgeteilt, zwischen denen die Objekte hin- und herkopiert werden und die nach jedem Kollektionszyklus ihre Rollen tauschen. Diese Idee wurde ursprünglich von FENICHEL und YOCHELSON für LISP entwickelt (vgl. [FY69]); im Folgenden wird eine Anlehnung an eine Variante von CHENEY betrachtet, die ohne Rekursion auskommt (vgl. [Che70]).

Algorithmus 4.3 Kopierende Garbage Collection zwischen Halbräumen nach FENICHEL, YOCHELSON und CHENEY (vgl. [FY69] und [Che70]).

```
 1: initialize():
 2:   target ← HEAP_START                                  ▷ Erste Hälfte
 3:   source ← (HEAP_START + HEAP_END)/2                   ▷ Zweite Hälfte
 4:   pos ← target                                         ▷ Füllstand

 5: atomic collect():
 6:   swap(target, source)                                 ▷ Halbräume tauschen
 7:   pos ← target
 8:   for each field ∈ PointerFields(Roots)                ▷ Beginne mit Basisobjekten
 9:     update(field)                                      ▷ Feld aktualisieren und Ziel zu toDo hinzufügen
10:   while toDo ≠ ∅                                      ▷ Solange kopierte, nicht aktualisierte
11:     ref ← remove(toDo)                                  Objekte existieren...
12:     for each field ∈ PointerFields(*ref)
13:       update(field)                                    ▷ ... aktualisiere ihre Felder
14:   clear(newAdr)

15: update(field):
16:   oldAdr ← *field                                     ▷ Adresse aus Feld holen
17:   if newAdr(oldAdr) = null                            ▷ Falls Ziel noch nicht kopiert wurde...
18:     newAdr(oldAdr) ← pos                              ▷ ... definiere neue Position
19:     pos ← pos + sizeOf(*oldAdr)
20:     moveObject(oldAdr, newAdr(oldAdr))                ▷ Kopiere Objekt an neue Position
21:     add(toDo, newAdr(oldAdr))                         ▷ Füge Kopie zu toDo hinzu
22:   *field ← newAdr(oldAdr)                             ▷ Feld aktualisieren
```

Zur Vorbereitung wird der Heap zu Programmbeginn in zwei Hälften eingeteilt, die durch zwei Zeiger `source` und `target` markiert werden (Prozedur *initialize* in Algorithmus 4.3). Der durch `target` referenzierte Halbraum ist stets derjenige, der neu angelegte bzw. zu kopierende Objekte aufnimmt. Ein dritter Pointer `pos` zeigt den

Füllstand von `target` und damit die Position an, an der ein neues Objekt angelegt bzw. kopiert wird. Der Allokator ist entsprechend so zu modifizieren, dass neue Objekte an der Stelle `pos` angelegt werden und `pos` anschließend angepasst wird, sowie die Garbage Collection bereits dann ausgelöst wird, wenn der freie Speicher in `target` zur Neige geht.

Kommt es zu einem Kollektionszyklus, werden zunächst die Rollen von `source` und `target` vertauscht (Zeile 6). Der Füllstandsanzeiger `pos` wird auf den Beginn des nun leeren Halbraums `target` gesetzt. Um alle erreichbaren Objekte aus `source` nach `target` zu kopieren, werden erst alle Felder von Basisobjekten auf Referenzen zu Heapobjekten untersucht (Zeile 8f). Diese Felder und die entsprechend referenzierten Objekte werden durch die Prozedur *update* bearbeitet. Diese schlägt in einer Tabelle `newAdr` nach, ob das Objekt bereits nach `target` kopiert und in ihr eine neue Speicheradresse hinterlegt wurde (Zeile 17). Falls nicht, wird die neue Adresse des Objekts auf die Position von `pos` gesetzt, das Objekt an diese Stelle kopiert und `pos` hinter das Objekt gesetzt (Zeile 18 bis 20). Damit alle Objekte erfasst werden können, die vom kopierten Objekt referenziert werden, wird dieses zu `toDo` hinzugefügt. In `toDo` befinden sich demnach alle Objekte, die bereits kopiert, aber deren ausgehende Referenzen noch nicht untersucht wurden. Schließlich kann die Referenz im untersuchten Feld auf die neue Speicheradresse aktualisiert werden (Zeile 22). Ist in `newAdr` bereits eine Speicheradresse hinterlegt worden, so wurde das Ziel bereits nach `target` kopiert; in diesem Fall genügt es, den Feldinhalt zu aktualisieren. Die Prozedur *collect* kann anschließend mit der Abarbeitung der Objekte in `toDo` beginnen (Zeile 10 bis 13).

Abbildung 4.6 veranschaulicht die Arbeitsweise des Algorithmus an einem Beispiel: Nachdem die Rollen der Halbräume `source` und `target` vertauscht wurden, wird zunächst das einzige von Basisobjekten referenzierte Objekt `b` kopiert **(2)**. Anschließend wird es zu `toDo` hinzugefügt, was durch eine Nummerierung der Kopie `b'` angezeigt wird **(3)**. Da nun alle Basisobjekte kopiert wurden, wird `toDo` abgearbeitet. Daher werden alle Felder von `b'` untersucht, die Ziele der enthaltenen Referenzen – hier `f` und `g` – nach `target` kopiert und die Kopien zu `toDo` hinzugefügt **(4)**. Analog wird mit `f'` verfahren **(5)**. Bei der Abarbeitung von `g'` tritt nun die Besonderheit auf, dass das referenzierte Objekt `f` bereits kopiert wurde. Entsprechend wird hier lediglich die Referenz angepasst **(6)**. Sobald `toDo` vollständig abgearbeitet wurde, sind alle erreichbaren Objekte kopiert worden und der Halbraum `source` kann in Gänze verworfen werden **(7)**.

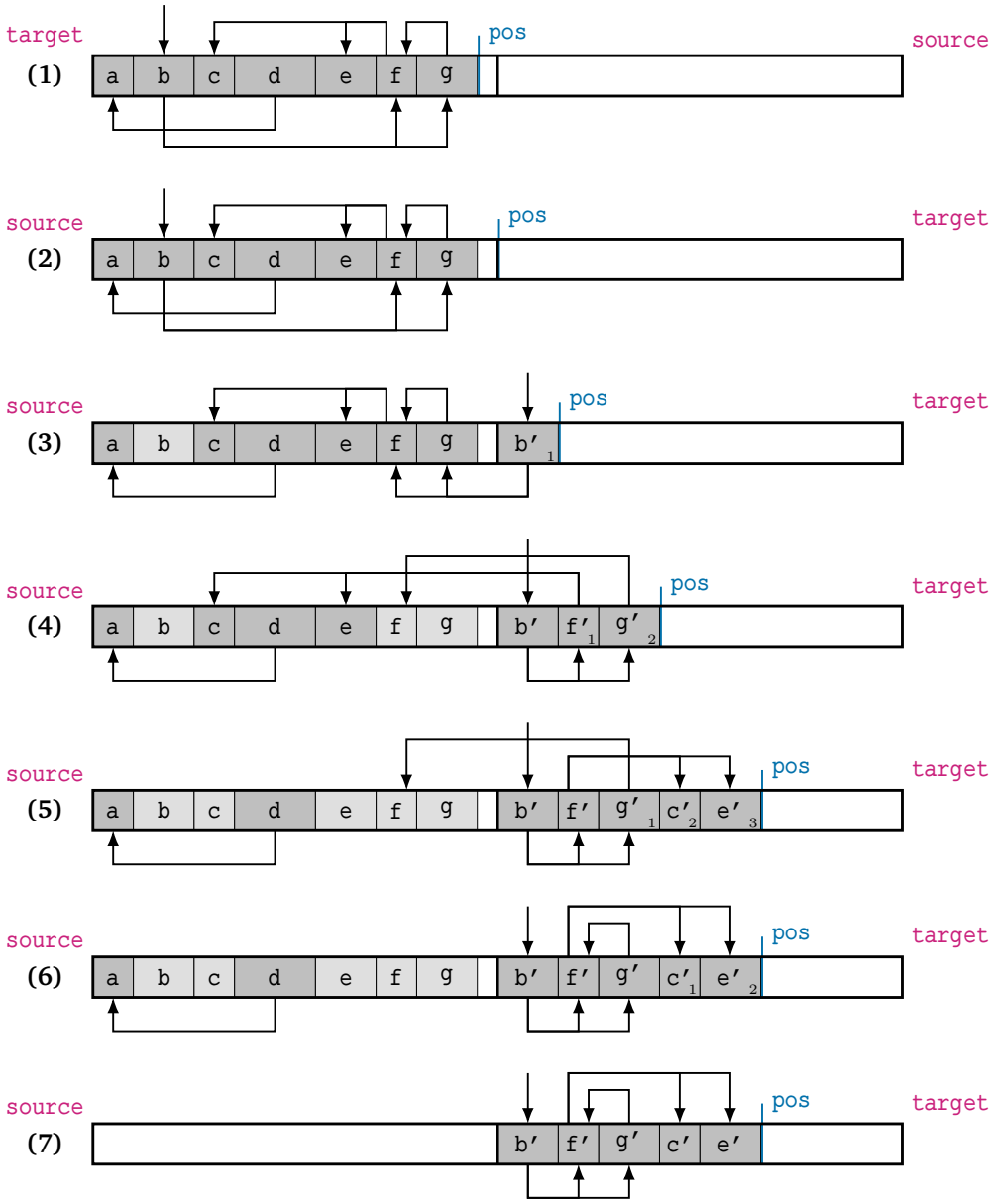


Abbildung 4.6.: Beispielhafte Ausführung von Algorithmus 4.3. Die Nummerierung der Kopien zeigt ihre aktuelle Position in `toDo` an. Referenzen der ursprünglichen Objekte aus `source` werden hier aus Gründen der Übersichtlichkeit nicht mehr aufgeführt, sobald das Objekt nach `target` kopiert wurde.

Die kopierten Objekte, die sich in `toDo` befinden, sind vergleichbar mit den grauen Objekten der Drei-Farben-Abstraktion (siehe Abschnitt 2.2): Sie wurden bereits durch den Kollektor entdeckt und nach `target` kopiert, aber ihre Felder wurden noch nicht auf Referenzen zu noch unentdeckten Objekten untersucht. Entsprechend analog zu Satz 2.2 lässt sich die Terminierung des Algorithmus beweisen, da kein Objekt mehrfach zu `toDo` hinzugefügt wird.

Satz 4.2 (Korrektheit der kopierenden Garbage Collection)**:**
Die kopierende Garbage Collection nach FENICHEL, YOCHELSON und CHENEY ist korrekt.

Beweis: Wir beweisen, dass nach Terminierung des Algorithmus kein erreichbares Objekt im Halbraum `source` verbleibt. Sei also `a` $\in \mathcal{R}$ ein erreichbares Objekt des Heaps. Dann gilt `b[i]` $=$ `&a` für Feld `b[i]` eines Objektes `b`. Ist `b` $\in$ ROOTS, so ist `&b[i]` $\in$ POINTERFIELDS(ROOTS) und `a` wird in einer Iteration der **for each**-Schleife (Zeile 8f) nach `target` kopiert sowie `b[i]` angepasst.

Ist `b` $\notin$ ROOTS, so sei `b` ohne Einschränkung aus dem Halbraum `target`.[4] Da `b` also nach `target` kopiert wurde, wurde `&b` auch zuvor zur Menge `toDo` hinzugefügt (Zeile 21). Foglich kam es zu einer Iteration der **while**-Schleife (Zeile 10) mit `ref` $=$ `&b` und, da `&b[i]` $\in$ POINTERFIELDS(`b`), wurde auch hier `a` nach `target` kopiert und `b[i]` angepasst. Somit werden alle erreichbaren Objekte nach `target` kopiert. □

Die Laufzeit der kopierenden Garbage Collection ist vergleichbar mit der der Markierungsphase des Mark-Sweep-Algorithmus (siehe Satz 2.3), da alle erreichbaren Objekte und ihre Felder aufgesucht bzw. aktualisiert werden müssen. Hinzu kommen allerdings Verzögerungen, die durch das Kopieren aller erreichbaren Objekte entstehen. Dies bietet jedoch die Möglichkeit, Objekte innerhalb des Heaps neu anzuordnen: Abbildung 4.6 lässt bereits erkennen, dass zwei Objekte, die vom selben Objekt referenziert werden, nebeneinander kopiert werden können. Das kann sich positiv auf die räumliche Lokalität von zueinander in Beziehung stehenden Objekten auswirken. Wie bereits in Abschnitt 2.4 angedeutet, hängt die Ausprägung dieses Effekts stark von der Traversierungsreihenfolge der Objekte – und damit von der Realisierung der Menge `toDo` – ab, die optimalerweise an den konkreten Anwendungsfall angepasst wird (vgl. [JHM11, Kap. 4.2]). Darüber hinaus bietet die Analogie zur Drei-Farben-Abstraktion auch hier die Möglichkeit, die Atomizität des Algorithmus aufzuweichen, indem Objekte, deren ausgehende Referenzen während der Kollektion manipuliert werden, durch geeignete Lese- und Schreibbarrieren gegebenenfalls erneut zu `toDo` hinzugefügt werden (vgl. [KNL07, S. 129], [Hos06, S. 41]).

Größter und offensichtlicher Nachteil des Algorithmus ist die permanente Halbierung des potenziell verfügbaren Heapspeichers, da ein Halbraum ausschließlich während eines Kollektionszyklus verwendet wird. Dies erhöht potenziell die Häufigkeit von

[4] Andernfalls ist `b` selbst ein erreichbares Objekt im Halbraum `source` und die Behauptung folgt per Induktion.

Garbage-Collection-Zyklen. Hinzu kommt während der Arbeit des Kollektors zusätzlicher Speicherbedarf zur Verwaltung von `toDo` (siehe Abschnitt 2.4) und `newAdr`. Letzterer lässt jedoch vermeiden, indem etwa die neue Speicheradresse des kopierten Objekts an der ursprünglichen Position hinterlegt wird.

4.4 Optimierung von Referenzanpassungen

In allen drei vorgestellten Algorithmen war zu sehen, dass eine Anpassung von Referenzen notwendig ist, wenn ein Objekt im Speicher verschoben oder kopiert wird. Dies erfordert im Allgemeinen eine Anpassung aller Felder von Objekten, die die ursprüngliche Speicheradresse enthalten. Um den Aufwand hierfür zu reduzieren, können **Handles** (dt. *Griff*, *Henkel*) verwendet werden. Ein Handle `ã` eines Objekts `a` ist ein weiteres Objekt, das lediglich die Speicheradresse von `a` enthält und eine feste Position im Speicher besitzt. Alle Objekte besitzen anstelle einer Referenz auf `a` eine solche auf `ã`. Ändert sich die Position von `a` im Speicher, ist ausschließlich eine Anpassung der Speicheradresse in `ã` nötig (vgl. [Bro84]).

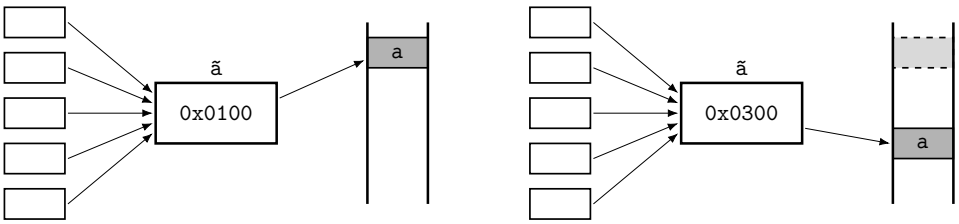


Abbildung 4.7.: Der Handle `ã` enthält die Speicheradresse des Objekts `a`. Erfolgt der Zugriff auf `a` ausschließlich über `ã`, so genügt es, lediglich die Adresse in `ã` anzupassen.

Die Realisierung dieses Mechanismus kann auf unterschiedlichen Wegen erfolgen. Denkbar ist unter anderem eine zentrale Tabelle, in der zu jedem Objekt die aktuelle Position aufgeführt wird, die mittels einer eindeutigen ID des Objekts ermittelt werden kann. Anstelle von Adressen wird vom Mutator dann die ID zur Referenzierung von Objekten verwendet. Wie bereits in Abschnitt 3.3 angedeutet, kann sich das Nachschlagen in großen Tabellen jedoch negativ auf die Performance auswirken. Alternativ empfehlen KALIBERA und JONES, Handles analog zu Objekten zu verwalten und im Heap abzulegen. Da Handles eine unveränderliche Position besitzen, bietet es sich etwa an, bestimmte Bereiche des Heaps ausschließlich für Handles zu reservieren, die von Kompaktierungsmechanismen ausgespart werden (vgl. [KJ11, S. 90f]).

Die in Algorithmus 4.4 modifizierte Variante von *new* zur Erzeugung neuer Objekte legt für jedes erzeugte Objekt `a` gleichzeitig einen Handle `ã` an, dessen Adresse zurückgegeben wird. In `forward(ã)` wird die Adresse von `a` gespeichert, während

Algorithmus 4.4 Erzeugung von Objekt und Handle mittels *new*. Anstelle einer Referenz auf das eigentliche Objekt erhält der Mutator eine Referenz auf den Handle des Objekts.

```
 1: new():
 2:     adr ← allocate()
 3:     if adr = null
 4:         collect()
 5:         adr ← allocate()
 6:     handle ← allocateHandle()          ▷ Anforderung im separaten Heapbereich
 7:     if handle = null
 8:         collect()
 9:         handle ← allocateHandle()
10:     if adr = null ∨ handle = null
11:         error("Nicht genügend Speicher")
12:     forward(*handle) ← adr             ▷ Objektadresse im Handle eintragen
13:     handle(*adr) ← handle              ▷ Handle-Adresse im Objekt eintragen
14:     return handle                      ▷ Adresse des Handles zurückgeben
```

`handle(a)` eine Referenz auf ã im Header von `a` bereithält. Diese ist notwendig, damit nach Verschiebung von `a` der zugehörige Handle gefunden und angepasst werden kann.

Aus Sicht des Mutators soll die Existenz der Handles im Verborgenen bleiben: Müsste die Entwicklerin auf den korrekten Umgang mit Handles achten, stünde dies im Widerspruch zur Idee einer Garbage Collection, die Entwicklerin von der Speicherverwaltung zu entlasten. Insofern sollte der Zugriff auf `handle(a)` nur dem Kollektor ermöglicht werden. Je nach Programmiersprache müssen dafür auch gewisse Operatoren angepasst werden, wenn sie durch den Mutator verwendet werden. So muss etwa die Dereferenzierung einer Speicheradresse nicht den Handle, sondern das vom Handle referenzierte Objekt liefern (siehe Algorithmus 4.5). Dies kann beispielsweise in der Programmiersprache C++ durch Überladung des Adress- und Dereferenzierungsoperators realisiert werden (vgl. [Str13, Kap. 18]).

Algorithmus 4.5 Sollen Handles für den Mutator verborgen bleiben, müssen Operatoren überschrieben werden, die mit Speicheradressen von Objekten arbeiten.

```
1: mutatorOperator*(ref):
2:     return *forward(*ref)          ▷ Dereferenziere Speicheradresse im Handle

3: mutatorOperator&(obj):
4:     return handle(obj)             ▷ Gib Adresse des Handles zurück
```

Die Verwendung von Handles vereinfacht die betrachteten Mark-Compact-Algorithmen deutlich, da auf eine kostspielige Aktualisierung aller Objektfelder verzichtet und eine Anpassung des Handles unmittelbar nach Verschiebung eines Objekts vorgenommen werden kann. Im LISP-2-Algorithmus wird damit die Prozedur *updateRefs* obsolet und die Buchführung über alte und neue Adressen ist nicht länger notwendig (siehe Algorithmus 4.6). Allerdings besitzen diese Vorteile auch hier ihren Preis:

Da jedes Objekt im Heap einen Handle besitzt, der ebenfalls dynamisch erzeugt wurde, verdoppelt dieser Ansatz die Zahl der Objekte und reduziert den verfügbaren Heapspeicher. Wie bereits diskutiert kann dies zu häufigeren Garbage-Collection-Zyklen führen, weswegen Handles in Kombination mit der kopierenden Garbage Collection eher unattraktiv sind. Zudem müssen Handles durch eine eigene Garbage Collection behandelt werden, da sie nicht verschoben werden dürfen. Zwar ist ein Handle genau dann erreichbar, wenn das zugehörige Objekt erreichbar ist. Allerdings werden in den zuletzt betrachteten Algorithmen verwaiste Objekte nicht explizit freigegeben, sondern durch andere Objekte überschrieben bzw. dadurch überschreibbar, dass sie sich außerhalb des kompaktierten Bereichs befinden. Ein gleichzeitiges Freigeben von Handle und Objekt ist somit nicht realisierbar. Weiter verursacht die feste Position der Handles eine Fragmentierung des Heaps, die eigentlich beseitigt werden soll. Das Problem lässt sich jedoch in Grenzen halten: Falls Handles ausschließlich eine einzige Speicheradresse enthalten, belegen sie gegebenenfalls gleich viel Speicher. Neu erzeugte Handles fügen sich somit passgenau in entstandene Lücken ein. Nicht zuletzt sollte auch beachtet werden, dass sich Handles auf die Performance des Mutators auswirken können. Wird auf eine Folge $\mathtt{a}_1 \rightarrow \mathtt{a}_2 \rightarrow \cdots \rightarrow \mathtt{a}_n$ von Referenzen von Objekten zugegriffen, so erfolgt zwischen zwei Objekten stets ein Zugriff auf die jeweiligen Handles. Da Handles und Objekte in verschiedenen Bereichen des Heaps aufbewahrt werden, lässt sich hier keine räumliche Lokalität herstellen, die von Caching-Mechanismen ausgenutzt werden könnte.

Algorithmus 4.6 Optimierung des LISP-2- und Compressor-Algorithmus mit Handles.

```
 1: compact():                                          ▷ LISP-2-Algorithmus
 2:     pos ← HEAP_START
 3:     next ← nextMarkedObject(HEAP_START)
 4:     while next ≠ null
 5:         if pos ≠ next
 6:             forward(*handle(*next)) ← pos           ▷ Adresse im Handle anpassen
 7:             moveObject(next, pos)
 8:         pos ← pos + sizeOf(*pos)
 9:         next ← nextMarkedObject(next)

10: compactAndUpdate():                                 ▷ Compressor-Algorithmus
11:     pos ← nextMarkedObject(HEAP_START)
12:     while pos ≠ null
13:         forward(*handle(*pos)) ← newAddress(pos)    ▷ Handle anpassen
14:         moveObject(pos, newAddress(pos))
15:         pos ← nextMarkedObject(pos)
```

5 Hybride und generationelle Ansätze

In den vorigen Kapiteln wurden hauptsächlich Algorithmen betrachtet, deren Anwendung stets den gesamten Heap betrifft. An zwei Stellen war jedoch zu erkennen, dass es zielführend sein kann, auf diesen Grundsatz zu verzichten: Die verzögerte Bereinigung nach HUGHES (Abschnitt 2.3) teilt den Heap nach Objekten gleicher Größe ein, um das Sweeping nur bei Bedarf auf einem Bruchteil des Heaps ausführen zu müssen. Die Einführung von Handles in Abschnitt 4.4 legt die Speicherung eben jener in einem separaten Bereich nahe. Da die Handles im Gegensatz zu den eigentlichen Objekten nicht verschoben werden, muss dieser Bereich mit einem anderen Garbage-Collection-Ansatz verwaltet werden.

Die Partitionierung des Heaps in disjunkte Bereiche ermöglicht eine feingranularere Auswahl der Garbage-Collection-Strategien. Die verschiedenen Bereiche können unterschiedlich häufig, durch abgestimmte Algorithmen und/oder eine Kombination verschiedener Ansätze bereinigt werden. Im Folgenden werden exemplarisch zwei Algorithmen vorgestellt, die diese Idee aufgreifen. Der erste unterscheidet Objekte nach ihrer Lebensdauer, was eine feine Abstimmung der Kollektionshäufigkeit ermöglicht. Der zweite Algorithmus verwendet für unterschiedlich alte Objekte verschiedene Ansätze, die in den vorigen Kapiteln dieser Arbeit behandelt wurden. Abschließend erfolgt ein kurzer Überblick über weitere Möglichkeiten zur Partitionierung des Heaps.

5.1 Algorithmus von Lieberman und Hewitt

Eines der ersten Verfahren, die die Lebensdauer der Objekte als Kriterium für einen Kollektionszyklus nutzen, stammt von LIEBERMAN und HEWITT [LH83]. Der Heap wird dazu in Blöcke eingeteilt, die durch zwei Zahlen charakterisiert werden: Die **Generation** eines Blocks ist ein Indikator für die Lebensdauer der enthaltenen Objekte und wird regelmäßig inkrementiert, sodass zu jeder Generation genau ein Block gehört. Blöcke, die jünger sind, besitzen somit eine höhere Generationenzahl und neue Objekte werden immer in der zuletzt erzeugten Generation gespeichert. Die **Version** eines Blocks gibt an, wie oft er durch die Garbage Collection bereinigt wurde. Die Idee ist nun (analog zu Abschnitt 2.3), die Garbage Collection generationenwei-

se arbeiten zu lassen und die Laufzeit des Algorithmus für jüngere Generationen möglichst kurz zu halten. Dies geht zurück auf eine Hypothese von DEUTSCH und BOBROW, wonach ein Großteil der Objekte entweder frühzeitig verwaist oder langfristig erreichbar bleibt (vgl. [DB76, S. 523]). Dazu wird die Konvention eingeführt, dass nur direkte Referenzen innerhalb einer Generation oder von Objekten einer jüngeren Generation auf solche einer älteren erlaubt sind. Eine Referenz `a` $\rightarrow$ `b` eines älteren Objekts `a` auf ein jüngeres Objekt `b` wird indirekt über einen Handle realisiert, welcher in einem separaten Bereich der Generation von `b` gespeichert wird und – neben einer internen Referenz auf `b` – zusätzlich eine *Rückreferenz* `origin` auf das entsprechende Feld von `a` besitzt (siehe Abbildung 5.1)[1].

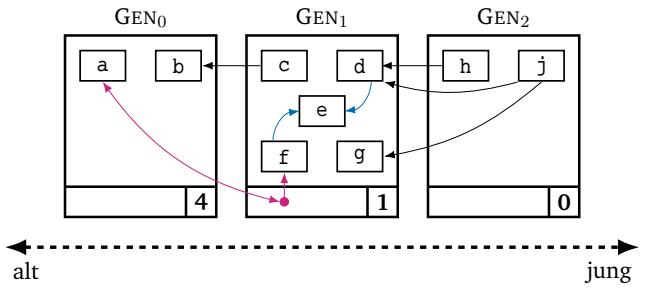


Abbildung 5.1.: Beispielhafte Objektkonstellation zwischen drei Generationen. Der Zähler unten rechts gibt die akuelle Version einer Generation an. Neben Referenzen von jüngeren auf ältere Objekte existieren auch interne Referenzen zwischen gleich alten Objekten (blau) und indirekte Referenzen von älteren auf jüngere Objekte über Handles (purpur).

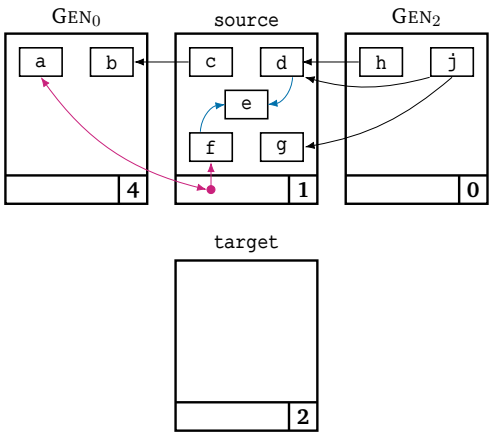
Der Kollektor geht bei der Bereinigung der Generation $k \in \mathbb{N}_0$ (`source`) in mehreren Schritten vor (siehe Algorithmus 5.1): Als erstes wird ein Block für eine neue Version `target` der betrachteten Generation angelegt (Zeile 3). Nun werden die erreichbaren Objekte in `source` nach `target` kopiert, vergleichbar mit dem Halbraumverfahren aus Abschnitt 4.3. Zunächst werden dazu alle Objekte kopiert, die über Handles erreichbar sind (Prozedur *updateHandles*). Dabei wird für jeden auffindbaren Handle in `source` das entsprechende Ziel nach `target` kopiert, ein neuer Handle in `target` angelegt und das Feld des älteren Objekts angepasst, das eine Referenz auf den ursprünglichen Handle enthält (Zeile 12 bis 14). Weiter werden alle direkten Referenzen auf Objekte der k-ten Generation durch die Prozedur *updateFields* aktualisiert. Aufgrund der oben eingeführten Konvention genügt es, alle Blöcke mit höherer Generationenzahl zu untersuchen (Zeile 16). Auch hier werden dann Referenzen auf Objekte aus der k-ten Generation nach `target` kopiert und die korrespondierenden Felder angepasst. Damit hierbei auch Referenzen berücksichtigt werden, die von Basisobjekten ausgehen, wird ROOTS als *Generation* ∞, also als jüngste aller Generationen, identifiziert.

[1] Wir gehen im Folgenden davon aus, dass jeder Handle eine einzelne Referenz verwaltet. Natürlich können auch alle Referenzen auf `b` über einen einzigen Handle realisiert werden; dieser muss entsprechend Rückreferenzen auf alle Felder speichern, die `b` referenzieren. Der Algorithmus ist entsprechend anzupassen.

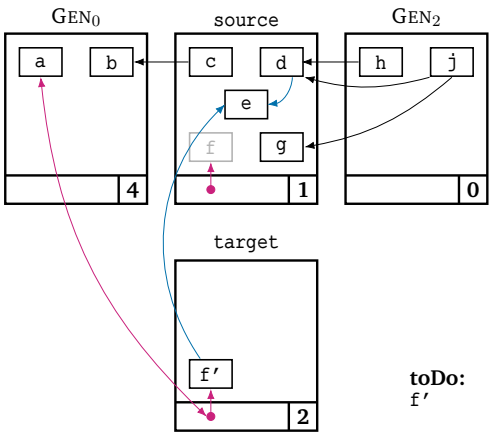
Algorithmus 5.1 Generationelle Garbage Collection nach LIEBERMAN und HEWITT (vgl. [LH83, S. 421ff]).

```
1:  collect(k):
2:    source ← GEN_k
3:    target ← newVersion(GEN_k)        ▷ Neue Version der Generation initialisieren
4:    GEN_k ← target
5:    pos ← target
6:    updateHandles(source, target)                          ▷ Handles anpassen
7:    updateFields(source)                                   ▷ Übrige Objekte kopieren
8:    clear(newAdr)
9:    free(source)                                           ▷ Alte Version freigeben

10: updateHandles(source, target):
11:   for each handle ∈ HANDLES(source)     ▷ Handles der Generation anpassen
12:     obj ← *forward(handle)
13:     copy(obj, pos)
14:     *origin(handle) ← newHandle(target, newAdr(obj))

15: updateFields(source):
16:   for each j > generation(source)                        ▷ Referenzen aus jüngeren
17:     for each field ∈ POINTERFIELDS(GEN_j)                  Generationen anpassen
18:       if **field ∈ source
19:         copy(**field, pos)
20:         *field ← newAdr(**field)
21:   while toDo ≠ ∅
22:     ref ← remove(toDo)
23:     for each field ∈ POINTERFIELDS(*ref)     ▷ Interne Referenzen anpassen
24:       if **field ∈ source
25:         copy(**field, pos)
26:         *field ← newAdr(**field)

27: copy(obj, pos):
28:   if newAdr(obj) = null                          ▷ vgl. update in Algorithmus 4.3
29:     newAdr(obj) ← pos
30:     moveObject(&obj, pos)
31:     pos ← pos + sizeOf(obj)
32:     add(toDo, newAdr(obj))
```

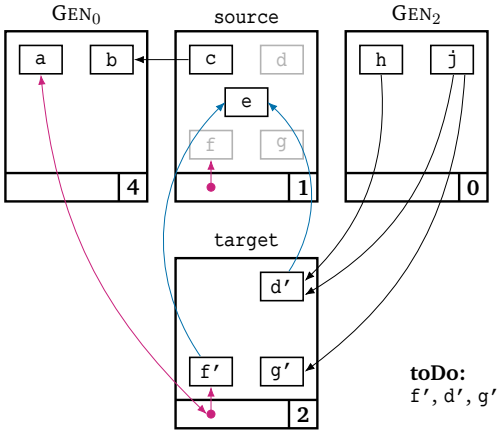
Das Kopieren eines Objekts wird stets durch die Prozedur *copy* bewerkstelligt. Genau wie in der Prozedur *update* im Halbraumverfahren (Algorithmus 4.3) wird ein Objekt nur dann kopiert, wenn es nicht zuvor bereits kopiert wurde. Andernfalls genügt es, lediglich die im Feld gespeicherte Adresse anzupassen. Da kopierte Objekte ebenfalls Referenzen auf Objekte derselben Generation enthalten können, werden sie zu `toDo` hinzugefügt. Bei der Abarbeitung von `toDo` (Zeile 21 bis 26) werden somit auch Objekte aufgespürt, die nur durch generationeninterne Referenzen erreichbar sind. Abbildung 5.2 zeigt die Arbeitsweise am Beispiel der oben dargestellten Objektkonstellation.



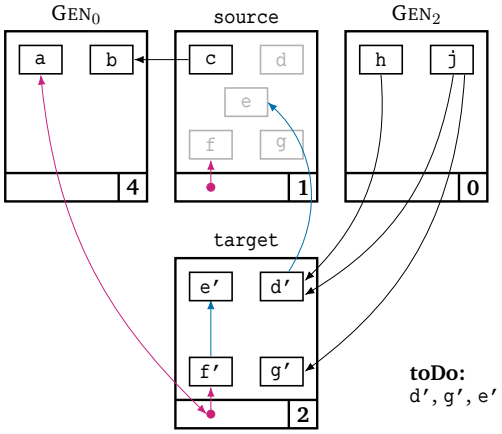
(a) Anlegen einer neuen Version.



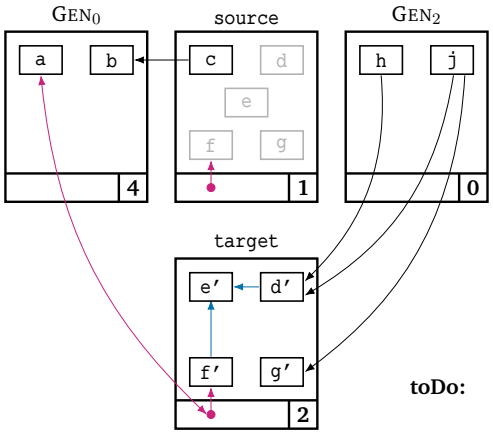
(b) Kopieren der von älteren Generationen referenzierten Objekte und Erzeugung neuer Handles.



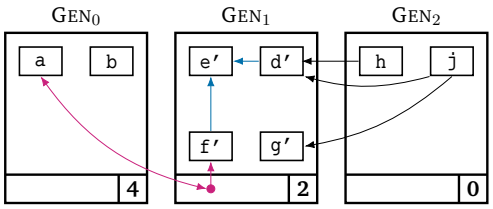
(c) Kopieren der von jüngeren Generationen referenzierten Objekte.



(d) Abarbeitung der `toDo`-Menge zur Aktualisierung interner Referenzen.



(e) `toDo` ist abgearbeitet. Nicht kopierte Elemente in `source` sind verwaist.



(f) `source` wird en bloc freigegeben.

Abbildung 5.2.: Beispielhafte Ausführung von Algorithmus 5.1 für Generation $k = 1$.

Die Stärke des Algorithmus besteht darin, die Häufigkeit der Garbage Collection von der Generation von Objekten und der Versionszahl abhängig zu machen. Sehr junge Generationen niedriger Version enthalten nach der Hypothese von DEUTSCH und BOBROW potenziell viele Objekte, die bereits verwaist sind. Entsprechend ist es attraktiv, diese zu bereinigen, da sie eine vielversprechende Ausbeute bieten. Die erwartete Laufzeit des Kollektors hält sich dabei gleichzeitig in Grenzen, da nur wenige Nachfolgegenerationen existieren, die Referenzen auf die zu reinigende Generation besitzen könnten. Ältere Generationen, die schon häufig bereinigt wurden, enthalten wiederum tendenziell weniger verwaiste Objekte. Da bei ihrer Bereinigung viele jüngere Generationen betrachtet werden müssen, sollten diese eher selten stattfinden (vgl. [LH83, S. 423]). Basisobjekte wiederum sind tendenziell häufigen Referenzmanipulationen unterworfen. Die Identifikation von ROOTS mit GEN_∞ ist somit zweckmäßig, damit bei der Bereinigung einer Generation diese Manipulationen berücksichtigt werden.

Abgesehen von Generation und Versionszahl gibt es noch weitere Parameter zur Optimierung des Algorithmus. Eine periodische Erhöhung der Generationenzahl hat den Vorteil, dass Objekte, die fast zeitgleich erzeugt werden, in derselben Generation gespeichert werden, was sich für einige Datenstrukturen positiv auf die räumliche Lokalität auswirken dürfte. Alternativ kann jedoch auch eine neue Generation erst dann angelegt werden, wenn die aktuelle nicht mehr genügend freien Speicher besitzt, um eine Speicheranforderung zu erfüllen. Das verringert den möglichen Speicherverschnitt durch nur teilweise belegte Blöcke. Da neue Objekte stets in der jüngsten Generation angelegt werden, tendieren ältere Generationen dazu, mit jedem Kollektionszyklus zu schrumpfen. Aus demselben Grund bietet es sich daher an, die Größe jeder neuen Version einer Generation auf die Größe der aktuell enthaltenen Objekte zu setzen. Eine andere Möglichkeit wäre, zusätzliche Konsolidierungsphasen einzubauen, in welchen Generationen verschmolzen werden. Eine weitere, nicht zu verachtende Maßnahme ist die Parallelisierung des Algorithmus, indem mehrere Generationen gleichzeitig gepflegt werden (vgl. [LH83, S. 426]).

Zuletzt sollte beachtet werden, dass die eingangs erwähnte Hypothese über die Lebensdauer von Objekten in der Praxis nicht zutreffen muss, wenngleich sie auch in neueren Programmiersprachen empirisch bestätigt wurde (vgl. [JR08]). Ebenso mag die Annahme, dass nur wenige Referenzen von älteren auf jüngere Objekte zu Stande kommen und sich die Zahl der benötigten Handles damit in Grenzen hält, auf die von LIEBERMAN und HEWITT betrachteten LISP-Systeme zutreffen, da die Komponenten eines Objekts in den meisten Fällen unveränderlich sind und bereits vor Erzeugung des Objekts existieren (vgl. [LH83, S. 422]). In objektorientierten Programmiersprachen etwa sind Attribute von Objekten in der Regel mutabel, weswegen dieser Gedanke nicht notwendigerweise auf diese Fälle übertragbar ist. Die

Anzahl der Handles liegt dann viel höher, womit die in Abschnitt 4.4 angesprochenen Nachteile relevant werden können.

5.2 Verborgene Referenzzählung

Eine periodische Erzeugung neuer Heapabschnitte hat zwar den Vorteil, eine an die Lebensdauer der Objekte angepasste Garbage Collection zu ermöglichen, aber auch den Nachteil, zusätzliche Metainformationen über jede Generation verwalten zu müssen (siehe dazu auch die in Abschnitt 2.3 genannten Nachteile). Als Kompromiss kann es zielführend sein, gröber zwischen jüngeren und älteren Objekten zu entscheiden und sich somit auf zwei Altersklassen zu beschränken: Jüngere Objekte tendieren nicht nur zu einer hohen Verwaisungswahrscheinlichkeit, sondern auch zu einer höheren Veränderungsrate, während ältere Objekte eher seltener manipuliert werden. Diese Feststellung nutzen BLACKBURN und MCKINLEY in ihrem Konzept der **verborgenen Referenzzählung** (engl. *ulterior reference counting*), um die Vorteile markierender und referenzzählender Algorithmen zu vereinigen [BM03]. Der Heap wird dazu aufgeteilt in zwei (nicht notwendigerweise gleich große) Teile NURSERY (dt. etwa *Kinderzimmer*) und MATURE (dt. *erwachsen*, *ausgewachsen*). Neu erzeugte Objekte werden zunächst in NURSERY angelegt und verbleiben dort bis zum nächsten Kollektionszyklus. In MATURE befinden sich alle Objekte, die mindestens einen Kollektionszyklus überlebt haben.

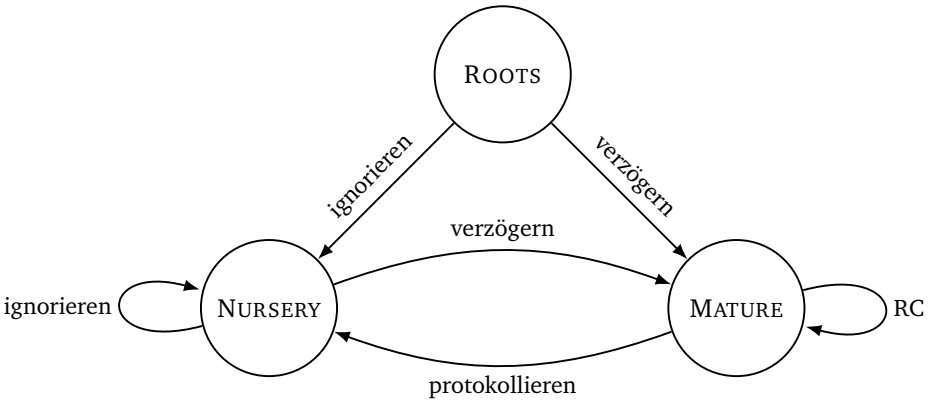


Abbildung 5.3.: Prinzip der verborgenen Referenzzählung. Das Schema zeigt, wie Referenzen zwischen den einzelnen Heapbereichen durch die Referenzzählung behandelt werden (vgl. [BM03, S. 346]).

Ob und in welcher Form die Referenzzählung zum Einsatz kommt, richtet sich nun danach, ob sich Quelle oder Ziel in ROOTS, NURSERY oder MATURE befinden (siehe Abbildung 5.3). Bereits in Abschnitt 3.1 wurde festgestellt, dass Referenzzählungen – bedingt durch nötige Schreibbarrieren – die Performance einer Anwendung beeinträchtigen, wenn viele Manipulationen von Referenzen stattfinden. Da junge Objekte in NURSERY tendenziell häufiger Manipulationen von eingehenden wie ausgehen-

den Referenzen unterworfen sind, bietet es sich an, diese von der Referenzzählung auszusparen. Referenzen von ROOTS nach NURSERY sowie innerhalb von NURSERY sind von der Schreibbarriere nicht betroffen; Referenzen von MATURE nach NURSERY werden zunächst protokolliert und während des nächsten Kollektionszyklus behandelt. Objekte in MATURE verändern sich tendenziell seltener, weshalb Markierungs- und Kompaktierungsalgorithmen keine hohe Ausbeute versprechen, aber dennoch aufwendig sind. Schreibbarrieren und Zähleranpassungen sind für Referenzen innerhalb MATURE daher tolerabel. Manipulationen, die Referenzen von ROOTS bzw. NURSERY nach MATURE betreffen, werden hingegen verzögert behandelt und ihre Auswirklungen auf Referenzzähler erst während einer Kollektionsphase beachtet.[2] Algorithmus 5.2 zeigt die entsprechende Schreibbarriere.

Algorithmus 5.2 Schreibbarriere der verborgenen Referenzzählung (vgl. [BM03, S. 346ff]).

```
1:  writeRef(obj, i, ref):
2:    if obj ∈ MATURE
3:      atomic
4:        if (*obj[i] ∈ NURSERY ∧ *ref ∉ NURSERY)
5:          removeLog(&obj[i])          ▷ Falls MATURE → NURSERY entfernt wird
6:        else if (*obj[i] ∉ NURSERY ∧ *ref ∈ NURSERY)
7:          addLog(&obj[i])             ▷ Falls MATURE → NURSERY erzeugt wird
8:        if *ref ∈ MATURE              ▷ Referenzen MATURE → MATURE behandeln
9:          incRefCount(*ref)
10:       if *obj[i] ∈ MATURE
11:         decRefCount(*obj[i])
12:   obj[i] ← ref
```

Kommt es nun zu einem Garbage-Collection-Zyklus, etwa weil der freie Speicher im NURSERY-Bereich zur Neige geht, werden alle erreichbaren Objekte aus NURSERY nach MATURE kopiert. Da die Objekte in NURSERY von der Referenzzählung ausgeschlossen sind, wird dies analog zum Halbraumverfahren aus Abschnitt 4.3 bewerkstelligt. Zunächst werden daher alle Felder von Basisobjekten untersucht (Zeile 2 bis 5 aus Algorithmus 5.3). Enthält ein Feld eine Referenz auf ein junges Objekt, wird es an die Prozedur *tenure* übergeben. Diese kopiert ein zuvor noch nicht entdecktes Objekt nach MATURE, fügt es zur Menge `toDo` hinzu und aktualisiert die Speicheradresse im übergebenen Feld (vgl. Prozedur *update* in Algorithmus 4.3). Im Anschluss wird – unabhängig davon, ob das Objekt kopiert wurde – der Referenzzähler des Objekts inkrementiert, da es sich nun in jedem Fall in MATURE befindet und mittels Referenzzählung verwaltet wird. Zudem wird hierdurch der Referenzzähler um die Anzahl an Referenzen aus ROOTS korrigiert (siehe Algorithmus 3.6).

Weiter werden alle Referenzen von älteren auf jüngere Objekte behandelt (Zeile 6 bis 9). Felder von Objekten in MATURE, die eine Referenz auf Objekte aus NURSERY

[2] Zur Erinnerung: Ein Referenzzählerwert $\texttt{rc}(\texttt{a}) = 0$ bedeutet somit nicht, dass a nicht erreichbar ist, sondern dass keine Referenzen auf a von Objekten aus MATURE existieren. Jedoch können Referenzen auf a von Objekten aus ROOTS oder NURSERY existieren (siehe Abschnitt 3.3).

enthalten, wurden zuvor von der Schreibbarriere zur Menge `log` hinzugefügt. Somit genügt eine Iteration über diese Menge, um die entsprechenden Objekte zu kopieren. Auch hier werden wieder alle betroffenen Referenzzähler korrigiert, da Referenzen von MATURE nach NURSERY nun Referenzen innerhalb MATURE sind. Anschließend erfolgt eine Abarbeitung der `toDo`-Menge, die alle Objekte enthält, die in diesem Kollektionszyklus nach MATURE kopiert wurden (Zeile 10 bis 15). Da diese Referenzen auf bislang unentdeckte Objekte in NURSERY besitzen können, werden ihre Felder entsprechend untersucht, die Ziele gegebenenfalls kopiert und die Referenzzähler korrigiert.

Algorithmus 5.3 Verborgene Referenzzählung nach BLACKBURN und MCKINLEY (vgl. [BM03, S. 346ff]).

```
 1: atomic collect():
 2:   for each field ∈ PointerFields(Roots)
 3:     if *field ∈ Nursery          ▷ Referenzen Roots → Nursery behandeln
 4:       tenure(field)              ▷ Ziel nach Mature kopieren und Feld anpassen
 5:     incRefCount(**field)                                  ▷ vgl. Algorithmus 3.6
 6:   for each field ∈ log          ▷ Referenzen Mature → Nursery behandeln
 7:     tenure(field)
 8:     incRefCount(**field)
 9:     removeLog(field)
10:   while toDo ≠ ∅
11:     ref ← remove(toDo)
12:     for each field ∈ PointerFields(*ref)
13:       if *field ∈ Nursery      ▷ Referenzen Nursery → Nursery behandeln
14:         tenure(field)
15:       incRefCount(**field)
16:   free(ZCT)                                  ▷ Objekte mit rc = 0 entfernen
17:   free(Nursery)                              ▷ Verbleibende junge Objekte entfernen
18:   for each ref ∈ Pointers(Roots)             ▷ RC-Anpassungen rückgängig machen
19:     decRefCount(*ref)                                     ▷ vgl. Algorithmus 3.6
```

Alle Objekte, die sich abschließend in MATURE befinden und deren Referenzzähler 0 beträgt, werden weder von Basisobjekten, noch von anderen Objekten aus MATURE referenziert und sind unerreichbar. Somit können sie durch Bereinigen der *zero count table* `ZCT` freigegeben werden (Zeile 16, siehe auch Algorithmus 3.6). Objekte, die sich in NURSERY befanden und erreichbar sind, wurden nach MATURE kopiert. Folglich kann der gesamte Bereich freigegeben werden (Zeile 17). Zuletzt müssen die Korrekturen der Referenzzähler rückgängig gemacht werden. Dazu genügt es, nochmals alle von ROOTS ausgehenden Referenzen zu betrachten, da diese nun lediglich auf Objekte aus MATURE verweisen.

Ein ausführliches Beispiel zum Ablauf des Algorithmus, das sämtliche Arten von Referenzen zwischen den einzelnen Heapbereichen berücksichtigt, befindet sich im Anhang A dieser Arbeit, weswegen an dieser Stelle abschließend auf die Vor- und Nachteile eingegangen wird. Ist die Aufteilung des Heaps in NURSERY und MATURE

geschickt gewählt, vermag der vorgestellte Algorithmus die Schwächen der in den vorigen Kapiteln vorgestellten Algorithmen zu reduzieren. Wie bereits erläutert, wird durch die Aussparung jüngerer Objekte der Performanceverlust durch Zählermanipulationen teilweise vermieden. Die Integration der verzögerten Referenzzählung nach DEUTSCH und BOBROW verhindert zudem Löschkaskaden und die unmittelbare Nachverfolgung von Referenzmanipulationen in Basisobjekten, etwa bei lokalen Variablen einer Prozedur. Die Verwaltung älterer Objekte mithilfe der Referenzzählung erspart wiederum die Kosten, die Markierungs- und Kompaktierungsansätze bei Betrachtung des gesamten Heaps mit sich bringen würden. Hier wirkt sich auch die Protokollierung derjenigen Felder in MATURE aus, die Referenzen nach NURSERY enthalten, da keine gesamte Traversierung des Heaps notwendig ist, um Referenzen auf kopierte Objekte anzupassen. Die Unterbrechungen im Programmablauf werden damit deutlich verringert. Beachtet werden muss jedoch die Schwierigkeit, das richtige Verhältnis von NURSERY und MATURE zu finden: Ist NURSERY zu klein angelegt, verursacht dies häufigere – wenn auch kürzere – Kollektionszyklen. Zudem besteht die Gefahr, dass Objekte frühzeitig nach MATURE verschoben werden, obwohl sie in naher Zukunft noch häufig manipuliert werden oder verwaisen. Ist NURSERY jedoch zu groß, verlängern sich potenziell die Kollektionsphasen, da mehr Referenzen verfolgt werden müssen. Zwei weitere Probleme bereffen vor allem die MATURE-Region: Zum einen vermag der Speicher in diesem Bereich nach und nach zu fragmentieren, was jedoch durch geschicktes Kopieren an geeignete Positionen in Grenzen gehalten werden kann. Zum anderen besitzt der vorgestellte Algorithmus keine Möglichkeit, verwaiste zyklische Referenzen älterer Objekte zu erkennen. BLACKBURN und MCKINLEY empfehlen daher, beim Erreichen eines gewissen Füllstandes des Heaps zusätzliche Erkennungen verwaister Zyklen einzustreuen (vgl. [BM03, S. 349]).

5.3 Weitere Partitionierungsmöglichkeiten

Neben dem Ansatz, Objekte nach *Lebensdauer* bzw. *Mutationsrate* zu partitionieren, sind auch andere Aufteilungen des Heaps denkbar (vgl. [JHM11, Kap. 8]). In Abschnitt 2.3 wurde der Heap in Blöcke eingeteilt, um die Bereinigungsphase zu beschleunigen. Alternativ bietet es sich an, Objekte unterschiedlicher *Größe* mit verschiedenen Techniken zu bereinigen. Große Objekte zu verschieben ist eventuell so zeitaufwendig, dass eine Defragmentierung nicht lohnenswert ist. Deswegen kann es ratsam sein, entsprechende Bereiche von einer Kompaktierung auszuschließen. Dieser Gedanke führt zum Ansatz, Objekte nach *Verschiebbarkeit* anzuordnen. Diesbezüglich wurde bereits in Abschnitt 4.4 der Vorschlag geäußert, nicht verschiebbare Handles in einer eigens reservierten Region zu lagern. Auch wenn keine Handles verwendet werden, können etwa häufig referenzierte Objekte als nicht verschiebbar deklariert werden, da eine Anpassung der betroffenen Referenzen zu kostspielig

wäre. Entsprechend können auch diese Bereiche von Kompaktierungsalgorithmen ausgespart werden.

In eine ähnliche Richtung geht der Ansatz, Objekte nach *Typ* zu sortieren. Zum Beispiel können Objekte, die keine ausgehenden Referenzen besitzen können, in eigene Bereiche gespeichert werden. Markierungsalgorithmen, die Referenzen verfolgen, profitieren gegebenenfalls davon, dies anhand der Speicheradresse zu erkennen und diese Objekte von einer weiteren Untersuchung auszuschließen. Ebenso können Objekte separiert werden, die dazu neigen, zyklische Referenzen zu bilden. Somit kann auch hier die kostspielige Überprüfung auf verwaiste Zyklen auf Teile des Heaps beschränkt werden.

Letztlich sollte nicht nur beachtet werden, nach welchen Kriterien der Heap partitioniert wird, sondern auch, ob eine Aufteilung generell sinnvoll ist, da eine Einschränkung der möglichen Positionen eines Objekts negative Auswirkungen auf die räumliche Lokalität haben kann. Insofern sollte für jeden konkreten Anwendungsfall abgewägt werden, ob die Vorteile einer Partitionierung den Aufwand und die entsprechenden Nachteile aufwiegen kann.

6 Qualitativer Vergleich

In den vorigen Kapiteln wurde eine Reihe sowohl grundlegender als auch elaborierter Algorithmen diskutiert, deren Vor- und Nachteile erläutert sowie Einsatzfälle genannt, in denen sie potenziell ihre Stärken ausspielen können. Zum Abschluss des ersten Teils sollen daher nochmals die Algorithmen bezüglich ausgewählter qualitativer Kriterien vergleichend gegenübergestellt werden. Diese Kriterien können je nach Anwendungsfall für eine erste Auswahl potenziell infrage kommender Garbage-Collection-Algorithmen herangezogen werden.

Durchsatz oder Responsivität?

Der Durchsatz ist ein Maß für die Performanz einer Software und beschreibt die Geschwindigkeit, mit der diese Aufgaben erledigt. Ein hoher Durchsatz verlangt, dass der Kollektor das eigentliche Programm möglichst wenig ausbremst und dem Mutator möglichst viel Prozessorzeit überlässt. Die Responsivität beschreibt hingegen die Antwortzeit, mit der eine Anwendung auf eingehende Anfragen reagiert. Wir haben gesehen, dass markierende Algorithmen häufig die Arbeit des Mutators unterbrechen, um fehlerhafte Markierungen zu vermeiden. In dieser Zeit können eingehende Anfragen nicht verarbeitet werden, da sie durch den Kollektor blockiert werden – die Responsivität sinkt. Referenzzählende Algorithmen verteilen ihre Arbeit hingegen auf die einzelnen Schreibzugriffe des Mutators, sodass hier Performanceeinbußen zu erwarten sind, während die Responsivität weitgehend erhalten bleibt. In ihrer naivsten Form scheint daher die Wahl klar zu sein: Strebt man eine Optimierung des Durchsatzes an, fällt die Entscheidung auf den Mark-Sweep-Algorithmus, da dieser außerhalb eines Kollektionszyklus keine zusätzliche Arbeit verursacht. Soll hingegen die Responsivität optimiert werden, führt kein Weg an der Referenzzählung vorbei, da sie keine längerfristigen Unterbrechungen verursacht. Allerdings war auch zu sehen, dass die fortgeschritteneren Varianten der beiden Algorithmen diese Klarheit relativieren. So kann die Drei-Farben-Abstraktion die Responsivität erhöhen, indem teilweise die Nebenläufigkeit von Mutator und Kollektor ermöglicht wird. Hierzu sind jedoch – wie bei der Referenzzählung – Schreibbarrieren nötig, die sich negativ auf den Durchsatz auswirken können. Soll die Referenzzählung auch zyklische Strukturen zuverlässig freigeben oder aus Durchsatzgründen auf einige

Schreibbarrieren verzichten, sind wiederum separate Bereinigungsphasen notwendig, die den Mutator anhalten und die Referenzzähler korrigieren.

Speicherbedarf

Da eine Garbage Collection häufig ausgelöst wird, wenn der freie Heapspeicher zur Neige geht, ist der Speicherbedarf der verschiedenen Konzepte ein weiterer Aspekt, der bei der Wahl des passenden Algorithmus beachtet werden sollte. Alle vorgestellten Algorithmen benötigen zusätzlichen Speicher pro Objekt, sei es, um Markierungsinformationen oder Referenzzähler innerhalb oder außerhalb von Objektheadern zu speichern. Einzelne Bits, die von markierenden Algorithmen verwendet werden, können gegebenenfalls in bereits vorhandene Header integriert werden, Referenzzähler können jedoch prinzipiell unbeschränkt groß werden. Für diese muss dann während der gesamten Laufzeit der Anwendung Speicher bereitgehalten werden. Der dadurch verursachte Overhead hängt vor allem von der Anzahl der Objekte und deren Größe ab. Auch die Aufteilung des Heaps in Blöcke macht die Speicherung zusätzlicher Informationen zur Verwaltung selbiger notwendig. Die Algorithmen selbst benötigen meist Hilfsdatenstrukturen wie Warteschlangen, um Adressen noch zu untersuchender Objekte zwischenzuspeichern sowie Änderungen von Speicheradressen nachzuverfolgen. Entsprechend viel Speicher muss ebenfalls reserviert werden, damit die Algorithmen ordnungsgemäß arbeiten können. Als besonders speicherhungrig erweisen sich hier der Compressor-Algorithmus (siehe Tabelle 4.5 auf Seite 47) sowie das Halbraumverfahren aus Abschnitt 4.3, welches die Heapgröße praktisch halbiert. Für Anwendungen, in denen wenig Speicher zur Verfügung steht, kann dies ein Ausschlusskriterium sein, da häufige Ausführungen der Garbage Collection den Mutator verdrängen könnten.

Kopieren oder nicht kopieren?

Ein weiteres Kriterium bei der Suche eines geeigneten Garbage-Collection-Algorithmus ist, ob der Algorithmus den genutzten Heapspeicher kompaktiert oder nicht. Bildet der freie Speicher einen möglichst großen zusammenhängenden Bereich, können aufeinanderfolgende Speicheranforderungen schnell erfüllt werden, da keine *passende Lücke* für die angeforderte Speichermenge gesucht werden muss. Allerdings sind kopierende Algorithmen vor allem für große Heaps tendenziell langsamer, da im schlimmsten Fall alle erreichbaren Objekte verschoben werden, sämtliche Referenzen auf verschobene Objekte angepasst werden müssen und der Mutator in dieser Zeit angehalten wird. Zumal besteht die Gefahr, dass der Kollektor *gegen den Allokator* arbeitet, da kopierende Algorithmen die Reihenfolge der Objekte verän-

dern könnte, die zuvor vom Allokator zur Ausnutzung räumlicher Lokalität optimiert wurde. Insofern ist es wichtig, den Kollektor auch auf die Arbeitsweise des Allokators abzustimmen.

Quantitative Resultate

Neben dieser qualitativen Gegenüberstellung ist es durchaus möglich, auch die quantitativen Unterschiede der verschiedenen Konzepte zu messen. Vor allem neuere Algorithmen werden dazu für die *Jikes Research Virtual Machine*[1] implementiert oder Benchmark-Frameworks wie *SPECjvm*[2] ausgesetzt, um Performanceunterschiede messbar zu machen und dabei verschiedene Ausprägungen von Objekten und Objektkonstellationen zu betrachten (siehe etwa [BM03], [BCM04], [KP06], [LP06], [GBF07]). Diese belegen durchaus, dass zum einen deutliche Performanceunterschiede zwischen verschiedenen Ansätzen in gewissen Anwendungsfällen existieren und zum anderen vermutlich kein Algorithmus existiert, der in jeder Situation perfekt ist (vgl. [FT00]). Dennoch sollte beachtet werden, dass derartige Untersuchungen unter gewissen *Laborbedingungen* stattfinden, da sie Anwendungsfälle lediglich simulieren. Somit lassen sich die Resultate nur eingeschränkt auf die Praxis übertragen, weswegen hier abschließend nur einer Empfehlung zugestimmt werden kann: „Know your application“ [JHM11, S. 80].

[1] `https://www.jikesrvm.org/`
[2] `https://www.spec.org/jvm2008/`

Teil II

Simulation ausgewählter Algorithmen

7 Modellierung und Implementation

Der zweite Teil der Arbeit beginnt mit einer Vorstellung des Entwicklungsprozesses des *Garbage-Collection-Simulators*. Nach einer kurzen Beschreibung der Anforderungen an diese Anwendung erfolgt ein Blick auf die Modellierung und Implementation der zentralen Programmkomponenten, wobei die wichtigsten Designentscheidungen motiviert werden. Abschließend wird die Realisierung der grafischen Ausgabe erläutert, die für eine nachvollziehbare Darstellung der ausgewählten Algorithmen wesentlich ist. Das darauffolgende Kapitel beschreibt schließlich, wie die Anwendung zu verwenden ist und wie sie um weitere Algorithmen ergänzt werden kann.

7.1 Spezifikation der Anforderungen

Der zu entwerfende Simulator hat das Ziel, die Ausführung ausgewählter Garbage-Collection-Algorithmen zu demonstrieren, sodass die einzelnen Arbeitsschritte eines Algorithmus klar erkennbar sind. Dazu wird eine Realisierung des in Kapitel 1 eingeführten Speichermodells angestrebt. Eine grafische Ausgabe visualisiert dabei den Heap als Blockgrafik: Belegte Teile des Heaps sind optisch von freien unterscheidbar, zudem sind Lage und Größe der einzelnen Objekte erkennbar. Weiter wird gefordert, dass die Anwenderin die Möglichkeit zur Konfiguration des Simulators hat: Neben einer obligatorischen Auswahl des zu verwendenden Garbage-Collection-Algorithmus ist auch die Anpassung der Größe des Heaps sowie der Größe der erzeugten Heapobjekte vorgesehen. Weitere Anpassungsmöglichkeiten sind der Verzweigungsgrad der Objekte untereinander sowie die Verwaisungswahrscheinlichkeit, sodass verschiedene Objektkonstellationen betrachtet werden können. Zuletzt ist durch eine Anpassbarkeit der Animationsgeschwindigkeit auch die Visualisierung konfigurierbar.

Aus Sicht der Softwareentwicklung ergeben sich zusätzliche Anforderungen, die für die Umsetzung der Anwendung relevant sind: Der Simulator soll als frei verwendbare Software im Rahmen der Lehre eingesetzt werden können und beliebig anpassbar und erweiterbar sein, etwa indem Entwicklerinnen weitere Garbage-Collection-Algorithmen ergänzen. Dies impliziert die Verwendung verschiedener Konzepte der Objektorientierung, unter anderem der Modularisierung in funktional unterscheidbare Pakete und Klassen, der Polymorphie und der generischen Program-

mierung. Darüber hinaus wird die Anwendung plattformunabhängig entwickelt, um sie möglichst vielen Anwenderinnen zugänglich zu machen.

7.2 Modellierung

Aufgrund der oben beschriebenen Anforderungen bietet sich die in Abbildung 7.1 dargestellte Modularisierung in einzelne Komponenten an, auf deren Funktion im Folgenden eingegangen wird. Bereits in Kapitel 1 wurde angemerkt, dass sich ein Programm grob in die drei Bestandteile Mutator, Allokator und Kollektor einteilen lässt. Insofern ist es sinnvoll, diese Einteilung möglichst auch in die Modellierung einfließen zu lassen, insbesondere um verschiedene Kombinationen von Allokator und Kollektor zu ermöglichen.

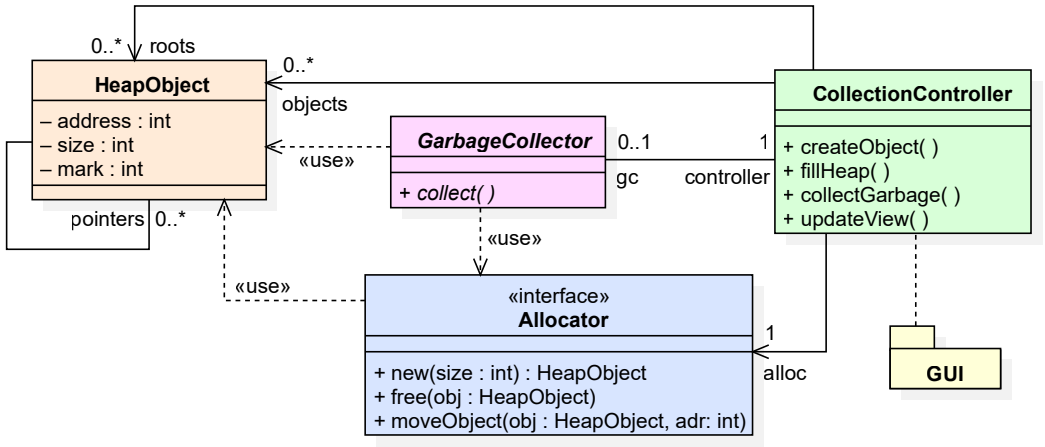


Abbildung 7.1.: Klassendiagramm zur Modellierung der Beziehung zwischen Mutator, Allokator und Kollektor. Im Rahmen der Implementation wurde dieses Modell um weitere Klassen, Methoden und Attribute ergänzt.

Heapobjekte werden als Instanzen einer Klasse `HeapObject` modelliert und beinhalten Speicheradresse (`address`), Größe (`size`), Markierungsinformation (`mark`) und eine Menge `pointers` an Referenzen auf weitere Instanzen von `HeapObject`, um die Menge POINTERS zu realisieren.

Die Schnittstelle `Allocator` stellt die wesentlichen Dienste eines Allokators zur Verfügung. Dazu zählt die Anforderung einer Speichermenge durch den Mutator, die Freigabe von Objekten und des durch sie belegten Speicherbereichs sowie das Verschieben eines Objekts. Ersteres geschieht durch Übergabe der gewünschten Speichermenge und Rückgabe einer neu erzeugten Instanz von `HeapObject`, die die entsprechende Speicheradresse enthält. Der Allokator ist diejenige Instanz, die Informationen über den Füllstand des Heaps und insbesondere über die Belegung einzelner Wörter besitzt. Die abstrakte Klasse `GarbageCollector` beschreibt die Funktionalität eines Kollektors, welche im Wesentlichen aus der Durchführung eines

Garbage-Collection-Zyklus besteht. Da eine Garbage Collection die Freigabe und Verschiebung von Objekten auslösen kann, benötigt sie die entsprechende Funktionalität von `Allocator`. Die Klasse `CollectionController` realisiert einerseits die Aufgabe des Mutators, das heißt die Anforderung von Speicher für neue Objekte sowie die (Simulation von) Referenzmanipulationen, die zur Verwaisung von Objekten führen. Daher besitzt sie zwei Mengen von Heapobjekten `objects` und `roots`. `objects` enthält dabei sämtliche Objekte des Heaps. `roots` enthält jedoch – im Gegensatz zur Menge ROOTS – keine Basisobjekte, sondern diejenigen Heapobjekte, die von Basisobjekten referenziert werden. Basisobjekte selbst werden somit in der Simulation ausgelassen. Zudem verwaltet `CollectionController` eine `Allocator`-Instanz, um Speicher anfordern zu können, sowie eine `GarbageCollector`-Instanz zur Auslösung der Garbage Collection. Letztere ist zudem in der Lage, über den `CollectionController` auf die beiden Objektmengen sowie den eingesetzten Allokator zuzugreifen. Andererseits fungiert `CollectionController` als Steuerklasse, um über die GUI eingehende Benutzerinteraktionen umsetzen und delegieren zu können (siehe Abschnitt 7.4).

Die Idee ist nun, beim Start des Simulators zunächst eine Instanz der Klasse `CollectionController` erzeugen. Diese legt wiederum – je nach ausgewählter Garbage Collection – eine geeignete `Allocator`-Instanz sowie einen `GarbageCollector` an und trägt sich beim Garbage Collector als Steuerinstanz ein. Dadurch greifen Kollektor, Allokator und Steuerklasse auf denselben simulierten Heap zu.

7.3 Implementation zentraler Komponenten

Im nun folgenden Abschnitt erfolgt die Vorstellung, wie die obige Modellierung unter Beachtung der spezifizierten Anforderungen realisiert wird. Dabei wird zunächst die Umsetzung des Speichermodells und der Programmlogik fokussiert. Abschnitt 7.4 widmet sich anschließend der Anbindung an die grafische Benutzerschnittstelle.

Im Sinne der Plattformunabhängigkeit wird dazu die Programmiersprache *Java* gewählt, wobei auf die *Java Language Specification 9*[1] zurückgegriffen wird. Der gesamte Programmcode wird durch das Versionskontrollsystem *Git*[2] verwaltet. Um plattformübergreifend eine einheitliche Entwicklung zu gewährleisten, kommt zudem das Build-Management-Tool *Apache Maven*[3] zum Einsatz. Dieses bildet die verschiedenen Arbeitsschritte der Entwicklung, wie etwa Kompilieren, Testen und Erzeugen von JAR-Dateien, auf automatisiert durchführbare Phasen ab und verwaltet dabei eigenständig die Abhängigkeiten von externen Bibliotheken. Zur Weiter-

[1] `https://docs.oracle.com/javase/specs/jls/se9/html/index.html`
[2] `https://git-scm.com/`
[3] `https://maven.apache.org/`

entwicklung des Systems genügt somit ein Klonen des Git-Repositorys und eine Ausführung von Maven, um alle benötigten Abhängigkeiten beschaffen zu lassen.

Bei der Entwicklung der einzelnen Komponenten wird grundsätzlich *testgetrieben* vorgegangen. Das bedeutet, dass zu jeder zu implementierenden Methode zunächst mehrere Testfälle erstellt werden, die das erwartete Verhalten des Programms beschreiben. Einzige Ausnahme bilden hier Methoden, die mit GUI-Komponenten interagieren, da diese nur mit zusätzlichem Aufwand automatisiert testbar sind. Als Testframework wird auf *JUnit 5*[4] zurückgegriffen.

7.3.1 Heapobjekte (`pst.gcsim.Objects`)

Die Implementation der Klasse `HeapObject` unterscheidet sich nur unwesentlich von der in Abschnitt 7.2 aufgeführten Modellierung. Erwähnenswert ist jedoch die zusätzlich implementierte Klasse `AddressComparator` (siehe Listing 7.1): Da es je nach Kontext sinnvoll ist, Objektmengen sortieren zu können (siehe etwa Abschnitt 7.3.4), vergleicht diese `Comparator`-Implementation Objekte anhand ihrer Speicheradresse. Damit zwei verschiedene Objekte mit gleicher Speicheradresse nicht fälschlicherweise als identisch eingestuft werden, werden sie im Zweifelsfall zusätzlich nach einer internen ID verglichen.[5] Diese ist ein zusätzliches Attribut in Form einer fortlaufenden Nummerierung, die beim Erstellen eines `HeapObject` vergeben wird.

```
1 public class AddressComparator implements Comparator<HeapObject> {
2   public int compare(HeapObject first, HeapObject second) {
3     if (first.getAddress() == second.getAddress())
4       return (Integer.compare(first.getId(), second.getId()));
5     else return Integer.compare(first.getAddress(), second.getAddress());
6   }
7 }
```

Listing 7.1: Die Methode `compare` der Klasse `AddressComparator` vergleicht zwei Objekte anhand ihrer Speicheradresse und ihrer ID.

7.3.2 Allokatorklassen (`pst.gcsim.Allocators`)

Der Allokator verwaltet die Information, welche Wörter des simulierten Heaps belegt sind. Aus diesem Grund wurde zunächst eine Klasse `AllocationBitmap` entworfen, die diese Information in einem `boolean`-Array speichert (siehe Abbildung 7.2). Der

[4] `https://junit.org/junit5/`

[5] Die Java 9 API-Spezifikation der Klasse `Comparator` empfiehlt, dass für zwei Objekte `a` und `b` der Ausdruck `compare(a,b)` genau dann zu `0` auswertet, wenn `a.equals(b)` zu `true` auswertet. Diese Bedingung ist etwa für sortierte Datenstrukturen wie `TreeSet` wichtig, die keine Duplikate zulassen und `compare` zur Überprüfung auf Gleichheit heranziehen.

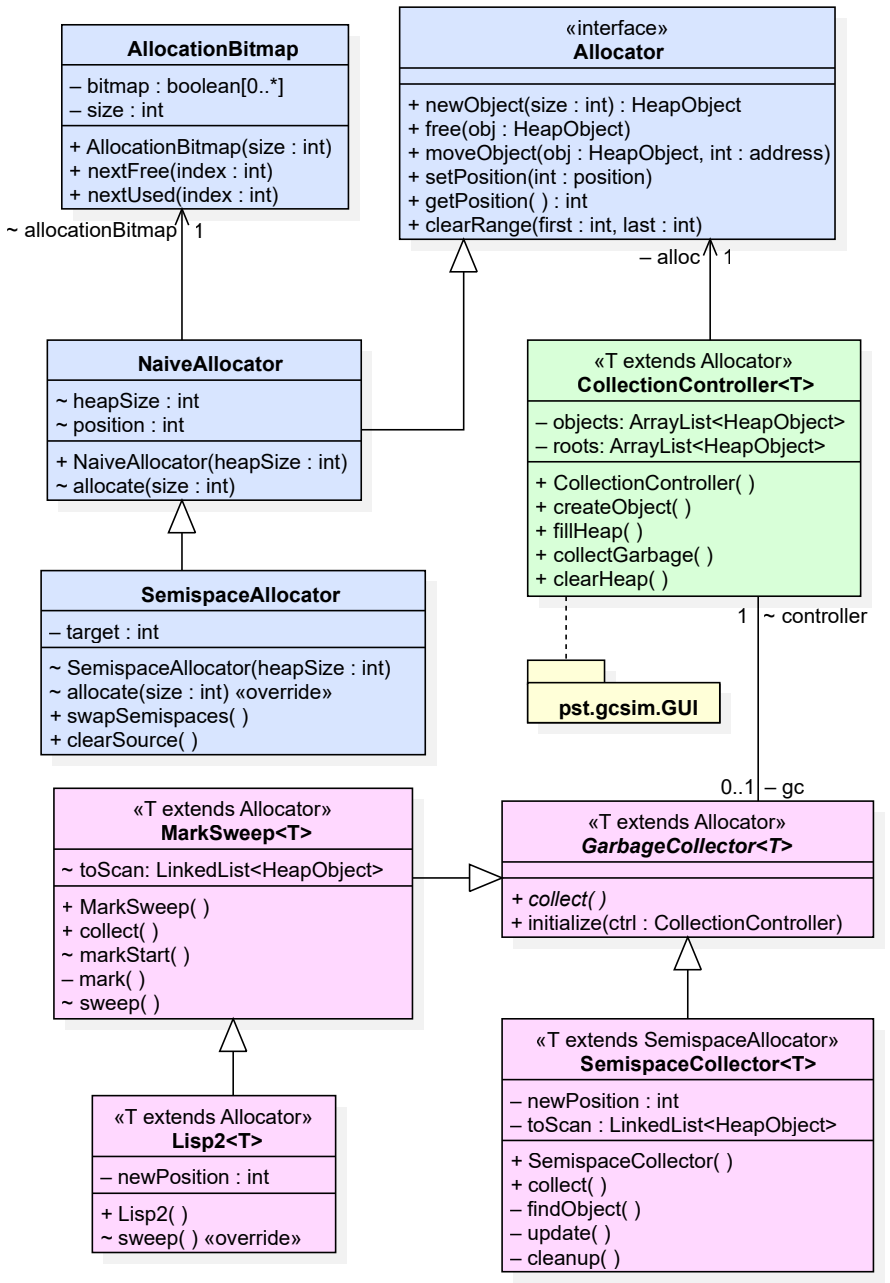


Abbildung 7.2.: Klassendiagramm (Auszug) des realisierten Modells aus Abbildung 7.1. Zwei Klassen mit gleicher Färbung gehören zum gleichen Paket.

i-te Eintrag dieses Arrays gibt an, ob das i-te Wort des Heaps durch ein Objekt belegt (`true`) oder frei (`false`) ist. Weiter stellt `AllocationBitmap` zwei Methoden `nextFree` und `nextUsed` zur Verfügung, welche ausgehend von einer übergebenen Speicheradresse die nächste freie bzw. belegte Stelle des Heaps liefern, indem das Array linear durchsucht wird. Diese Methoden sind essenziell, um für eine angeforderte Speichermenge eine hinreichend große freie Stelle zu finden.

Als Konkretisierung der Schnittstelle `Allocator` wurde zunächst ein naiver Allokator entwickelt. Dieser zeichnet sich dadurch aus, dass er – ausgehend von einer Startposition – mittels linearer Traversierung die nächste freie Stelle findet, die die angeforderte Speichermenge aufnehmen kann. Die Startposition ist dabei gegeben durch das Attribut `position`, welches stets die Adresse des ersten Worts hinter dem zuletzt angelegten Objekt enthält.

Das Herzstück der Klasse `NaiveAllocator` ist die Methode `allocate`, die intern durch `newObject` aufgerufen wird und das Auffinden einer geeigneten Position für ein neues Heapobjekt übernimmt (siehe Listing 7.2). Diese Methode sucht zunächst unter Zuhilfenahme von `nextFree` und `nextUsed` der Reihe nach alle freien Stellen hinter `position` ab, bis eine genügend große gefunden oder das Ende des Heaps erreicht wurde (Zeile 7 bis 15). Dabei wird auch berücksichtigt, dass sich hinter `position` eventuell kein freier Speicher befindet (Zeile 9f) oder ein freier Speicherbereich sich bis ans Heapende erstreckt (Zeile 12f). Wird so ein hinreichend großer Bereich freien Speichers gefunden, wird dieser entsprechend der angeforderten Speichermenge als belegt markiert und die Anfangsadresse des Bereichs zurückgegeben (Zeile 18 bis 23). Andernfalls wird auf analoge Weise der Bereich vor `position` durchsucht. Sollte auch hierbei kein geeigneter Speicherbereich gefunden werden können, wird als Ergebnis `-1` zurückgegeben, was `newObject` dazu veranlasst, kein neues Objekt zu erzeugen.

Der `SemispaceAllocator` wurde zur Realisierung des Halbraumverfahrens nach FENICHEL, YOCHELSON und CHENEY (siehe Abschnitt 4.3) als Spezialisierung des naiven Allokators entwickelt. Die Allokation läuft hier weitestgehend analog ab, allerdings wird der Allokationsversuch auf den aktuellen Zielhalbraum beschränkt, dessen Beginn im Attribut `target` notiert ist. Zusätzlich steht eine Methode `swapSemispaces` zur Verfügung, die die Rolle der beiden Halbräume tauscht, sowie eine Methode `clearSource`, die einen Halbraum in Gänze als unbelegt markiert.

7.3.3 Zentrale Steuerklasse `CollectionController`

Die Klasse `CollectionController` verwaltet als zentrale Steuerklasse zum einen die beiden Objektmengen `objects` und `roots`, auf die von den implementierten Garbage-Collection-Algorithmen per Getter-Methoden zugegriffen werden kann. Zum anderen bietet sie Methoden, um neue Objekte im Heap anzulegen bzw. bereits vorhandene Objekte verwaisen zu lassen. Die Entscheidung, ob zwischen zwei Objekten eine Referenz erzeugt wird oder ein Objekt zur Menge der Basisobjekte gehört, wird dabei zufällig gefällt. Die Wahrscheinlichkeit hierfür ist in der zentralen Konfigurationsklasse `Settings` (siehe Abschnitt 7.3.5) hinterlegt und kann zur Laufzeit verändert werden. Ebenso wird auch die Verwaisung eines Objekts simuliert (siehe

```
1  int allocate(int size) {
2    int gapSize = 0;         // Größe der aktuell betrachteten Lücke
3    int end = position;      // Beginne bei aktueller Position
4    int start = position;
5
6    // durchsuche Heap hinter position
7    while (gapSize < size && end != heapSize) {
8      start = allocationBitmap.nextFree(end);
9      if (start == -1)       // Am Heapende angekommen?
10       break;
11     end = allocationBitmap.nextUsed(start);
12     if (end == -1)         // Am Heapende angekommen?
13       end = heapSize;
14     gapSize = end - start;
15   }
16
17   // Prüfe, ob gefundene Lücke groß genug
18   if (gapSize >= size) {
19     position = (start + size) % heapSize;     // position anpassen
20     for (int i = start; i < start + size; i++)  // Bitvektor anpassen
21       allocationBitmap.setBit(i, true);
22     return start;
23   }
24
25   // Suche am Heapanfang fortsetzen
26   ...
27
28   return -1;       // keine hinreichend große Lücke gefunden
29 }
```

Listing 7.2: Auszug aus der Methode `allocate` der Klasse `NaiveAllocator`.

Listing 7.3): Vor Auslösung der Garbage Collection wird zunächst über die Menge der Basisobjekte iteriert und bei jeder Iteration eine zufällige Zahl im Intervall $[0, 1)$ bestimmt (Zeile 11). Liegt diese unterhalb einer eingestellten Wahrscheinlichkeit, wird das Objekt aus der Menge `roots` entfernt. Zur Vermeidung von Nebenläufigkeitskonflikten (`ConcurrentModificationException`), die bei Manipulation einer Datenstruktur auftreten, über die gleichzeitig iteriert wird, werden Löschkandidaten dabei zunächst zu einer separaten Liste hinzugefügt (Zeile 12). Gleiches geschieht für jedes Objekt mit der Menge `pointers` der ausgehenden Referenzen. Somit ist sichergestellt, dass jedes Objekt tatsächlich nach genügend Kollektionszyklen durch die Garbage Collection entsorgt wird, sofern die Verwaisungswahrscheinlichkeit größer als 0 ist.

Da die Steuerklasse den Garbage-Collection-Algorithmen den verwendeten Allokator zur Verfügung stellt, ist `CollectionController` als *generische Klasse* konzipiert. Der Typparameter `T` muss dabei die Schnittstelle `Allocator` implementieren. Würde das Attribut `alloc` stattdessen lediglich als Instanz von `Allocator` deklariert werden, gäbe es für die implementierten Garbage-Collection-Algorithmen keine Möglichkeit,

```
public void collectGarbage() {
  ctrlView.setInputs(false);    // Eingaben deaktivieren
  killRoots();                  // Basisobjekte vernichten
  killReferences();             // Referenzen vernichten
  gc.collect(true);             // Kollektor auslösen
}

private void killRoots() {
  ArrayList<HeapObject> toRemove = new ArrayList<>();
  for (HeapObject obj : roots)
    if (ThreadLocalRandom.current().nextDouble() <=
    ↪  Settings.REFERENCE_DELETION)
      toRemove.add(obj);
  toRemove.forEach(obj -> roots.remove(obj));
}

private void killReferences() {
  ArrayList<HeapObject> toRemove = new ArrayList<>();
  for (HeapObject obj : objects) {
    toRemove.clear();
    for (HeapObject ref : obj.getPointers())
      if (ThreadLocalRandom.current().nextDouble() <=
      ↪  Settings.REFERENCE_DELETION)
        toRemove.add(ref);
    toRemove.forEach(ref -> obj.getPointers().remove(ref));
  }
}
```

Listing 7.3: Methode `collectGarbage` der Klasse `CollectionController`.

speziellere Allokatoren mit zusätzlichen Methoden anzufordern, da sie lediglich Zugriff auf die durch die Schnittstelle definierten Methoden hätten. Aus gleichem Grund sind auch alle Kollektorklassen generisch, wobei jeder Kollektor zusätzliche Anforderungen an den Typparameter stellen kann.

7.3.4 Kollektorklassen (`pst.gcsim.GarbageCollectors`)

Die abstrakte Klasse `GarbageCollector` besitzt sämtliche Attribute und Methoden, die von allen konkret implementierten Garbage-Collection-Algorithmen gemeinsam verwendet werden. Neben der Anbindung an den `CollectionController` gehört dazu auch eine Methode `isInitialized`, die angibt, ob eine Verbindung zur Steuerinstanz besteht. Erst dann ist eine Auslösung der Garbage Collection möglich.

Konkret wurden im Rahmen dieser Arbeit der Mark-Sweep-Algorithmus mit Drei-Farben-Abstraktion, die Lisp-2-Kompaktierung und die kopierende Garbage Collection implementiert. Zunächst wurden die Algorithmen unabhängig von der geplanten grafischen Ausgabe umgesetzt. Dieses Vorgehen hat den Vorteil, dass sich die

```
void markStart() {
  toScan.clear();
  for (HeapObject obj : controller.getRoots())
    if (obj.getMark() == WHITE) {
      obj.setMark(GRAY);
      toScan.add(obj);
      mark();
    }
}

void mark() {
  while (!toScan.isEmpty()) {
    HeapObject obj = toScan.remove();
    obj.setMark(BLACK);
    for (HeapObject ref : obj.getPointers())
      if (ref.getMark() == WHITE) {
        ref.setMark(GRAY);
        toScan.add(ref);
      }
  }
}

void sweep() {
  // Objekte nach Adresse sortieren, um lineare Traversierung zu simulieren
  controller.getObjects().sort(new AddressComparator());
  ArrayList<HeapObject> toRemove = new ArrayList<>();  // Löschkandidaten
  for (HeapObject obj : controller.getObjects())
    if (obj.getMark() == WHITE)
      toRemove.add(obj);
    else obj.setMark(WHITE);
  toRemove.forEach(obj -> {
    controller.getAllocator().free(obj);
    controller.getObjects().remove(obj);
  });
}
```

Listing 7.4: Implementation der Drei-Farben-Abstraktion in der Klasse `MarkSweep` (vgl. auch Algorithmus 2.3).

vorgestellten Algorithmen auf recht kanonische Weise anhand ihrer Pseudocode-Beschreibung testgetrieben implementieren lassen. So weist etwa der Java-Code der `MarkSweep`-Klasse starke Ähnlichkeit zum Pseudocode in Algorithmus 2.3 auf (siehe Listing 7.4). Der einzige nennenswerte Unterschied ist die Umsetzung der linearen Heaptraversierung in der Bereinigungsphase: Die implementierten Allokatoren besitzen zwar die Information, welche Bereiche des Heaps belegt sind, nicht aber, an welcher Adresse ein Objekt beginnt. Daher wird die Traversierung simuliert, indem die Menge `objects` aller Heapobjekte vermöge des `AddressComparators` nach den Adressen der Objekte sortiert und anschließend über sie iteriert wird. Auch hier werden Löschkandidaten zunächst einer Liste `toRemove` hinzugefügt, um eine `ConcurrentModificationException` zu vermeiden.

```
1  public class Lisp2<T extends Allocator> extends MarkSweep<T> {
2    @Override
3    void sweep() {
4      controller.getObjects().sort(new AddressComparator());
5      int newPosition = 0;
6      ArrayList<HeapObject> toRemove = new ArrayList<>();
7      for (HeapObject obj : controller.getObjects())
8        if (obj.getMark() == WHITE)
9          toRemove.add(obj);
10       else {
11         controller.getAllocator().moveObject(obj, newPosition);
12         newPosition += obj.getSize();
13       }
14
15     toRemove.forEach(obj -> controller.getObjects().remove(obj));
16     // Heap hinter kompaktierten Bereich bereinigen
17     controller.getAllocator().clearRange(newPosition,
       ↪  controller.getAllocator().getHeapSize() - 1);
18     // Allokator-Position hinter kompaktierten Bereich setzen
19     controller.getAllocator().setPosition(newPosition);
20     controller.getObjects().forEach(obj -> obj.setMark(WHITE));
21   }
```

Listing 7.5: Auszug der Klasse `Lisp2`, die als Spezialisierung von `MarkSweep` die `sweep`-Methode überschreibt.

Da sich die Markierungsphase des LISP-2-Algorithmus nicht von der des Mark-Sweep-Algorithmus unterscheidet (siehe Abschnitt 4.1), ist eine Spezialisierung der Klasse `MarkSweep` zur Umsetzung naheliegend. In der Tat genügt es, in der Klasse lediglich die `sweep`-Methode zu überschreiben (siehe Listing 7.5). Der große Unterschied zum Pseudocode-Algorithmus 4.1 ist allerdings – neben der Simulation der linearen Traversierung – der weitestgehende Verzicht auf die `update`-Methode: Da die Verschiebung eines Objekts durch Manipulation des Attributs `address` bewerkstelligt wird, anstatt die Objekte tatsächlich im Speicher zu verschieben, ist hier eine Anpassung von Referenzen obsolet. Das bedeutet gleichzeitig, dass dieser Schritt im Kollektionszyklus nicht mithilfe der entworfenen Modellierung dargestellt werden kann. Die Kompaktierungsphase wird schließlich mit einer Freigabe des Heaps hinter dem kompaktierten Bereich und einer Verschiebung der Allokatorposition ans Ende dieses Bereichs abgeschlossen (Zeile 17 bis 19), wodurch erneut zu sehen ist, dass beim Lisp-2-Algorithmus auf explizite Freigabe von Objekten verzichtet werden kann. Allerdings muss jedes verwaiste Objekt im Hinblick auf eine grafische Ausgabe aus `objects` entfernt werden, um auch aus der Menge aller Heapobjekte zu weichen.

Zuletzt sei ein Blick auf die Implementation des Halbraumverfahrens in der Klasse `SemispaceCollector` geworfen. Auch hier zeigt sich zunächst eine deutliche Ähnlichkeit der Methode `collect` zum Pseudocode-Algorithmus 4.3, allerdings wird in der Methode `update` ebenfalls auf eine Anpassung von Referenzen verzichtet.

In Erinnerung daran, dass der ursprüngliche Algorithmus keine Objekte markiert, sondern unmittelbar bei Entdeckung in den anderen Halbraum verschiebt, fehlt somit zunächst die Möglichkeit optisch zu erkennen, ob ein Objekt bereits kopiert wurde. Zur Kompensation werden daher zusätzlich Markierungsinformationen genutzt, um zwischen unentdeckten, entdeckten (d.h. bereits kopierten) und fertig abgearbeiteten Objekten zu unterscheiden. Dies hat den zusätzlichen Vorteil, diese drei Status in der grafischen Ausgabe optisch unterscheidbar machen zu können. Auch beim Halbraumverfahren ist zuletzt eine Entfernung der verwaisten Objekte aus `objects` nötig, obschon der Quellhalbraum en bloc freigegeben werden kann.

```
1  public void collect() {
2    controller.getAllocator().swapSemispaces();
3    newPosition = controller.getAllocator().getTarget();
4    toScan.clear();
5
6    for (HeapObject obj : controller.getRoots())
7      update(obj);
8    while (!toScan.isEmpty()) {
9      HeapObject obj = toScan.remove();
10     for (HeapObject target : obj.getPointers())
11       update(target);
12     obj.setMark(DONE);
13   }
14   cleanUp();
15 }
16
17 void update(HeapObject obj) {
18   if (obj.getMark() == UNDISCOVERED) {
19     obj.setMark(DISCOVERED);
20     controller.getAllocator().moveObject(obj, newPosition);
21     newPosition += obj.getSize();
22     toScan.add(obj);
23   }
24 }
25
26 void cleanUp() {
27   ArrayList<HeapObject> toRemove = new ArrayList<>();
28   for (HeapObject obj : controller.getObjects())
29     if (obj.getMark() != DONE)
30       toRemove.add(obj);
31     else obj.setMark(UNDISCOVERED);
32
33   toRemove.forEach(obj -> controller.getObjects().remove(obj));
34   controller.getAllocator().clearSource();
35 }
```

Listing 7.6: Auszug der Klasse `SemispaceCollector`. Im Gegensatz zum ursprünglichen Algorithmus 4.3 wird zusätzlich die Markierungsinformation der Objekte genutzt.

7.3.5 Konfigurationsklasse `Settings`

Die Konfigurationsklasse `Settings` enthält wesentliche Einstellungen zum Betrieb des Simulators sowie vorkonfigurierte Standardwerte (siehe Listing 7.7). Dazu gehören etwa die Größe des Heaps (`HEAP_COLS`, `HEAP_ROWS`), die Größe erzeugter Objekte (`MIN_OBJECT_SIZE`, `MAX_OBJECT_SIZE`) und die Animationsgeschwindigkeit sowie Einstellungen, die Erreichbarkeit (`CONNECTIVITY`, `ROOT_PROBABILITY`) und Verwaisung (`REFERENCE_DELETION`) von Objekten beeinflussen. Die Werte können über die grafische Oberfläche verändert werden (siehe Abschnitt 8.1) und werden beim Beenden mittels der Klasse `java.util.Properties` in einer Datei `gcsim.config` gesichert bzw. beim Start aus dieser geladen. Zudem befindet sich in der Klasse `Settings` eine Auflistung der zur Verfügung stehenden Garbage-Collection-Algorithmen. In Abschnitt 8.2 wird erläutert, wie diese um weitere Algorithmen ergänzt werden kann.

```
1  public class Settings {
2    public static int HEAP_COLS = 40;
3    public static int HEAP_ROWS = 20;
4    public static int ANIMATION_SPEED = 200;
5    public static int MIN_OBJECT_SIZE = 2;
6    public static int MAX_OBJECT_SIZE = 30;
7    public static double CONNECTIVITY = 0.1;
8    public static double REFERENCE_DELETION = 0.3;
9    public static double ROOT_PROBABILITY = 0.2;
10
11   // Zur Verfügung stehende Kollektoren
12   public static final GarbageCollectorSelection[] collectors =
13     {marksweep, lisp2, semispace, naiveAlloc, semispaceAlloc};
14 }
```

Listing 7.7: Auszug aus der Konfigurationsklasse `Settings`.

7.4 Grafische Benutzerschnittstelle

Für die Umsetzung der grafischen Benutzerschnittstelle wird auf das Framework *Swing* sowie das *Abstract Window Toolkit* (AWT) zurückgegriffen, die sich für die Entwicklung plattformunahängiger Oberflächen eignen. Jedes Fenster wird dabei in einer eigenen Klasse umgesetzt (siehe Abbildung 7.3). Sogenannte *Event Listener*, die auf Benutzerinteraktionen wie Button-Klicks, Textfeldeingaben und Fensterschließungen reagieren, werden dabei konsequent durch innere Klassen realisiert.

Das Herzstück der grafischen Ausgabe des Heaps bildet die Klasse `CanvasView`. Diese besitzt neben einer Referenz auf die Menge `objects` der Steuerklasse und der aktuellen Position des Allokators zusätzlich die innere Klasse `CanvasPanel` als Spezialisierung der Swing-Komponente `JPanel` (siehe Listing 7.8). Durch Überschreiben der

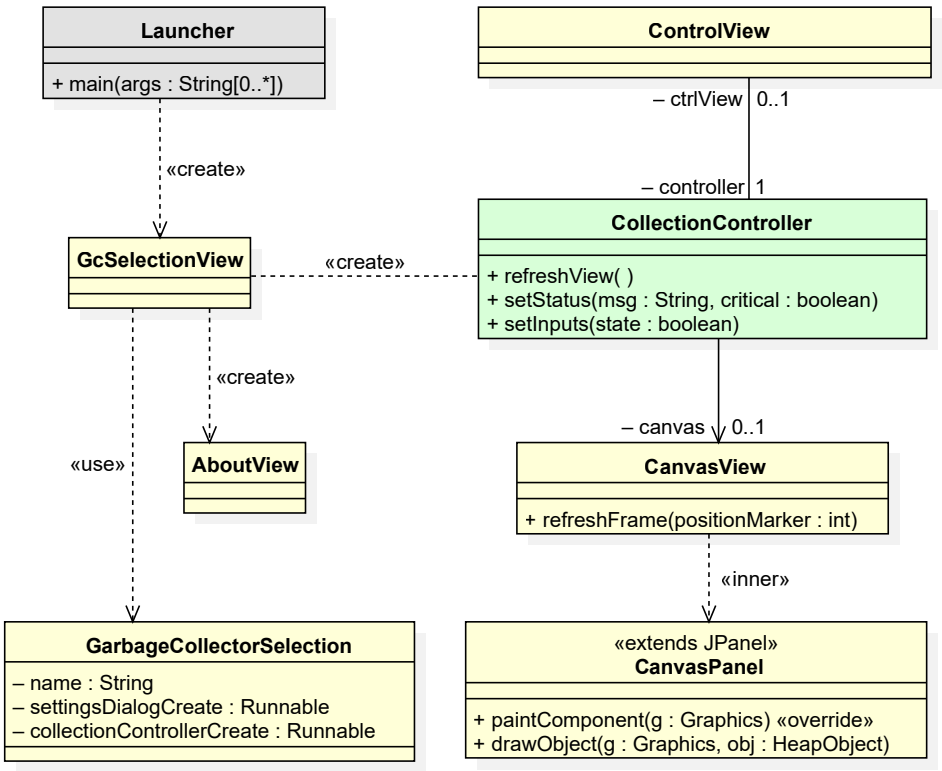


Abbildung 7.3.: Klassendiagramm (Auszug) des Pakets `pst.gcsim.GUI`. Zur Bedeutung der Klasse `GarbageCollectorSelection` siehe Abschnitt 8.2.

`paintComponent`-Methode können benutzerdefinierte Zeichnungen ausgegeben werden. Diese Methode wird zum Rendern des Fensterinhalts aufgerufen, beispielsweise wenn das Fenster über den Bildschirmrand hinausgeschoben wird. Ein manueller Aufruf durch den `CollectionController` ist über die Methode `refreshFrame` (Zeile 2 bis 6) möglich.

Da der Heap als zweidimensionale Grafik angezeigt wird, wird in der Methode `drawObject` zunächst anhand der Adresse des Heapobjekts `obj` die Startposition bestimmt (Zeile 22f). Zudem wird anhand der Markierung die verwendete Farbe festgelegt (Zeile 26 bis 34). Anschließend wird durch die `while`-Schleife das Objekt Zeile für Zeile gezeichnet (Zeile 37 bis 42). Dadurch wird berücksichtigt, dass sich das Objekt gegebenenfalls über mehrere Zeilen erstreckt. Zuletzt werden Linien gezeichnet, die Beginn und Ende des Objekts markieren. Ein Objekt erscheint somit als farbiges, geschlossenes Rechteck mit eventuellen Zeilenumbrüchen.

Zum Schluss gehen wir darauf ein, wie im entworfenen Simulator die Ausführung der Garbage-Collection-Algorithmen umgesetzt wird. Als nützliches Hilfsmittel hat sich hierfür die `Timer`-Komponente von Swing erwiesen. Mittels eines Swing-Timers kann eine Reihe von Anweisungen periodisch in einem eigenen Thread ausführt werden, wobei der zeitliche Abstand zwischen zwei Ausführungen definiert werden kann. Die Syntax dafür lautet:

```
1  public class CanvasView extends JFrame {
2    public void refreshFrame(int positionMarker) {
3      this.positionMarker = positionMarker;
4      this.revalidate();
5      this.repaint();    // ruft paintComponent auf
6    }
7
8    private class CanvasPanel extends JPanel {
9      @Override
10     public void paintComponent(Graphics g) {
11       super.paintComponent(g);
12       g.setColor(Color.WHITE);
13       g.fillRect(0, 0, Settings.canvasWidth(), Settings.canvasHeight());
14       drawGrid(g);                          // Zeichne Grundraster
15       for (HeapObject obj : objects)        // Zeichne jedes Objekt
16         drawObject(g, obj);
17       drawPositionMarker(g);                // Zeichne Allokatorposition
18     }
19
20     private void drawObject(Graphics g, HeapObject obj) {
21       // Bestimme Startzelle
22       int row = obj.getAddress() / Settings.HEAP_COLS;
23       int col = obj.getAddress() % Settings.HEAP_COLS;
24       int remaining = obj.getSize();   // noch zu zeichnende Zellen
25       Color color;
26       switch (obj.getMark()) {         // Farbe anhand Markierung festlegen
27         case MarkSweep.MS_WHITE:
28           color = Settings.COLOR_DEAD; break;
29         case MarkSweep.MS_GRAY:
30           color = Settings.COLOR_MARKED; break;
31         case MarkSweep.MS_BLACK:
32           color = Settings.COLOR_LIVE; break;
33         ...
34       }
35
36       // Zeichne Objekt Zeile für Zeile
37       while (remaining > Settings.HEAP_COLS - col) {
38         drawBlock(g, col, row, Settings.HEAP_COLS - col, color);
39         remaining = remaining - (Settings.HEAP_COLS - col);
40         ++row;
41         col = 0;
42       }
43
44       if (remaining > 0)    // Zeichne Reststück
45         drawBlock(g, col, row, remaining, color);
46       drawObjectStart(g, obj.getAddress() % Settings.HEAP_COLS,
47           obj.getAddress() / Settings.HEAP_COLS);
48       drawObjectEnd(g, col + remaining, row);
49     }
50   }
51 }
```

Listing 7.8: Auszug aus der Klasse `CanvasView`. Zu sehen sind die Methoden `paintComponent` und `drawObject`, welche die grafische Ausgabe erzeugen.

```
new Timer(<Verzögerung>, <Parametername> -> <Anweisungsblock>);
```

Dabei gibt der erste Parameter `<Verzögerung>` die Zeitspanne an, die zwischen zwei Ausführung von `<Anweisungsblock>` vergeht.[6] Durch den Lambda-Ausdruck `<Parametername> -> <Anweisungsblock>` wird hier ad hoc ein `ActionListener` definiert. Am Beispiel der Klasse `MarkSweep` lässt sich exemplarisch betrachten, wie ein derartiger Timer als Attribut in die Garbage-Collection-Algorithmen integriert werden kann (siehe Listing 7.9): Hier bietet es sich etwa an, mit jeder Timer-Auslösung das nächste Objekt aus `roots` zu markieren, sodass die Anwenderin sieht, wie nacheinander alle Basisobjekte markiert werden. Demzufolge wird die ursprüngliche `for`-Schleife der Methode `markStart` unter Zuhilfenahme eines Iterators der Menge `roots` in den Anweisungsblock von `gcTimer` integriert (Zeile 7 bis 13). Zusätzlich wird dabei die grafische Ausgabe des Heaps aktualisiert, um die erfolgte Markierung sichtbar zu machen (Zeile 12). Nachdem der Timer definiert wurde, wird er gestartet (Zeile 19). Sobald alle Basisobjekte abgearbeitet wurden, ist die Bedingung in Zeile 7 nicht mehr erfüllt, sodass sich der Timer im `else`-Block selbst anhält und die Methode `markTimed` aufruft. In dieser wird `gcTimer` analog so definiert, dass mit jeder Iteration ein Objekt der Menge `toScan` abgearbeitet wird, sodass auch hier die Behandlung jedes einzelnen Objekts beobachtet werden kann.

```
1  private void markStartTimed() {
2    toScan.clear();
3    // zunächst Iteration über roots
4    Iterator<HeapObject> iterator = controller.getRoots().iterator();
5    gcTimer = new Timer(Settings.ANIMATION_SPEED, action -> {
6      // mit jeder Timer-Iteration ein Basisobjekt abarbeiten
7      if (iterator.hasNext()) {
8        HeapObject obj = iterator.next();
9        if (obj.getMark() == WHITE) {
10         obj.setMark(GRAY);
11         toScan.add(obj);
12         controller.refreshView();      // Anzeige aktualisieren
13       }
14     } else {              // alle Basisobjekte abgearbeitet
15       gcTimer.stop();     // diesen Timer anhalten
16       markTimed();        // nächste Mark-Phase ausführen
17     }
18   });
19   gcTimer.start();            // starte erzeugten Timer
20 }
```

Listing 7.9: Ausführung des Mark-Sweep-Algorithmus mithilfe eines Swing-Timers. Mit jeder Auslösung des Timers wird ein weiteres Basisobjekt abgearbeitet. Im Anschluss an diese Phase wird der Timer angehalten und die Methode `markTimed` ausgeführt, in der `gcTimer` neu definiert und wieder gestartet wird.

[6] Standardmäßig findet jedoch die nächste Ausführung nicht vor Beendigung der vorigen statt, sodass sich zwei Ausführungszyklen nicht überlappen. Für Details sei hier auf die API-Spezifikation von `javax.swing.Timer` verwiesen.

Der Vorteil von Swing-Timern ist die nebenläufige Ausführung des Anweisungsblocks in einem eigenen Thread: Während ein Timer aktiv ist, wird dadurch der Rest der Anwendung nicht blockiert, sodass weiterhin Benutzereingaben möglich sind und die Animation der grafischen Ausgabe überhaupt sichtbar ist. Dies ermöglicht zudem, den Timer anzuhalten und einen Algorithmus zu unterbrechen. Offensichtlicher Nachteil ist die Integration in bereits implementierte Algorithmen, die im Zweifelsfall wenig intuitiv und fehlerträchtig ist. Zwar lassen sich die mit Timern versehenen Algorithmen ebenfalls mit JUnit testen, allerdings müssen die Tests künstlich verzögert werden, da andernfalls Ergebnisse überprüft werden, bevor die nebenläufige Ausführung beendet wurde.

Während in diesem Kapitel der Entwurf und die Implementation des Garbage-Collection-Simulators im Vordergrund steht, gehen wir im folgenden Kapitel in Kürze darauf ein, wie die Anwendung eingesetzt werden kann und wie aus ihrer Verwendung Erkenntnisse gewonnen werden können. Abschließend folgt ein kurzer Ausblick, welche Erweiterungsmöglichkeiten sich durch die umgesetzte Modellierung ergeben.

8 Anwendung und Erweiterung

8.1 Betrieb des Simulators

Der Start der Anwedung erfolgt mithilfe des beigelegten Datenträgers in drei verschiedenen Möglichkeiten:

1. Ausführung der JAR-Datei `gcsim-1.0-full.jar` mittels Doppelklick.
2. Ausführung in der Konsole mittels `java -jar gcsim-1.0-full.jar`.
3. Kopieren des Maven-Projekts im Ordner `gcsim` und Ausführung eines Build-Zyklus mit `mvn package`. Anschließend kann der Simulator mit `mvn exec:exec` gestartet werden.

Wird zusätzlich der einzige gültige Parameter `default` übergeben, werden statt der gespeicherten Einstellungen in der Datei `gcsim.config` die Standardeinstellungen verwendet. Das geschieht auch, wenn keine gültigen gespeicherten Einstellungen vorhanden sind.

Nach dem Start der Anwendung erscheint das Auswahlfenster, in dem grundlegende Einstellungen wie der zu verwendende Garbage-Collection-Algorithmus und die Größe des Heaps festgelegt werden können (siehe Abbildung 8.1). Zudem kann die Größe der Darstellung des Heaps angepasst werden. Der Bereich rechts neben den Texteingabefeldern gibt dabei Auskunft über die zu erwartende Größe des Ausgabefensters. Die Schaltfläche *Einstellungen* ist aktiviert, wenn für den ausgewählten Algorithmus zusätzliche Konfigurationsmöglichkeiten verfügbar sind (siehe Abschnitt 8.2).

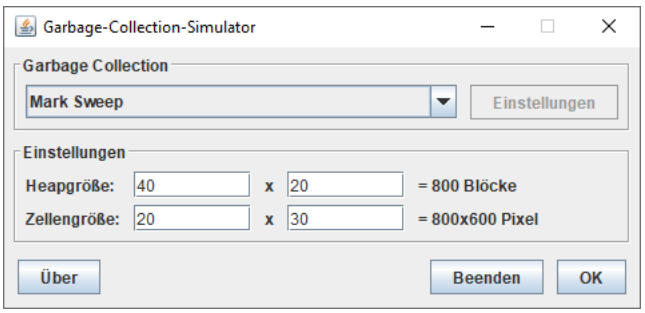
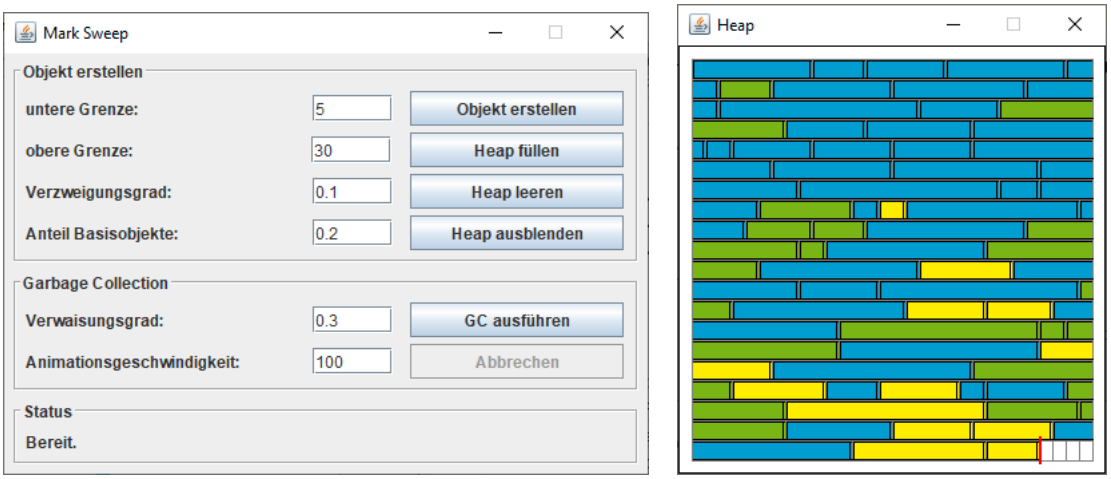


Abbildung 8.1.: Auswahlfenster des Simulators.

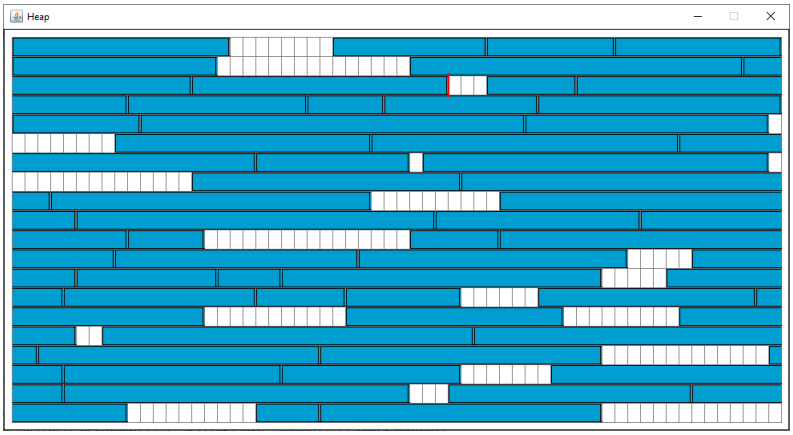
Sobald die Auswahl bestätigt wurde, beginnt der Simulationsmodus (siehe Abbildung 8.2). Mithilfe der Eingabekomponenten im oberen Bereich des Kontrollfensters können nun neue Objekte zum Heap hinzugefügt werden. Die beiden oberen Textfelder *untere Grenze* und *obere Grenze* geben dabei die Größenordnung an, in der sich neu erzeugte Objekte befinden. Das Textfeld *Verzweigungsgrad* gibt die Wahrscheinlichkeit an, mit der jedes bereits existierende Objekt eine Referenz auf ein neu erstelltes Objekt erhält, während der *Anteil an Basisobjekten* bestimmt, mit welcher Wahrscheinlichkeit ein erzeugtes Objekt ein Basisobjekt ist. Die Schaltfläche *Objekt erstellen* erzeugt ein einzelnes Objekt mit den spezifizierten Eigenschaften, während ein Klick auf *Heap füllen* solange Objekte erzeugt, bis ein Allokationsversuch fehlschlägt. Mittels der Schaltfläche *Heap leeren* kann der Heap zurückgesetzt werden. Dies ist etwa notwendig, wenn ein Kollektionszyklus unterbrochen wird und sich Objekte mit verschiedenen Markierungen im Heap befinden. Die Schaltfläche *Heap ausblenden* verbirgt die grafische Ausgabe des Heaps bzw. blendet sie wieder ein. Im unteren Bereich des Kontrollfensters lässt sich mit dem *Verwaisungsgrad* die Wahrscheinlichkeit einstellen, mit der eine Referenz entfernt wird (siehe auch Abschnitt 7.3.3). Im Textfeld *Animationsgeschwindigkeit* kann der zeitliche Abstand zweier Animationsschritte festgelegt werden (siehe auch Abschnitt 7.4). Ein Klick auf *GC ausführen* löst den Garbage-Collection-Algorithmus aus. Unentdeckte Objekte werden dabei zunächst hellblau dargestellt und bei Entdeckung gelb markiert. Nach erfolgter Abarbeitung werden sie grün gefärbt.



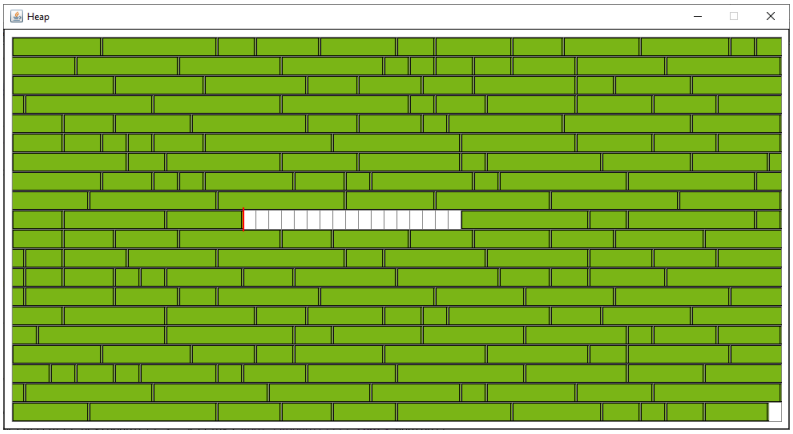
(a) Kontrollfenster **(b)** Darstellung des Heaps

Abbildung 8.2.: Simulationsmodus.

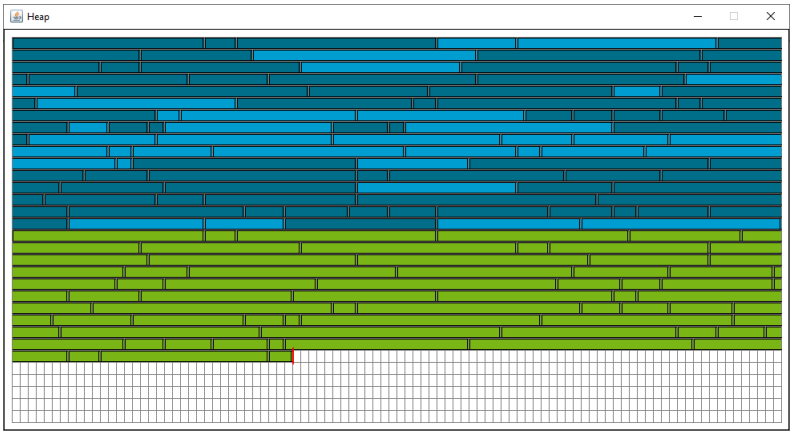
Durch wiederholte Ausführung von Füll- und Kollektionsphasen wird der langfristige Einsatz eines Algorithmus simuliert, wobei interessante Eigenschaften beobachtet werden können. Eine Anpassung der einstellbaren Parameter sorgt dabei für eine Simulation mannigfaltiger Szenarien (siehe Abbildung 8.3). Eine Verwendung unterschiedlich großer Objekte lässt etwa beim Mark-Sweep-Algorithmus die zunehmende Fragmentierung des ungenutzten Speichers erkennen, die Allokationsversuche



(a) Nach mehreren Kollektionzyklen lässt sich beim Mark-Sweep-Algorithmus deutlich eine langfristige Fragmentierung des Heaps erkennen.



(b) Bei geringem Verwaisungsgrad ist zu sehen, dass die Kompaktierungsphase des Lisp-2-Algorithmus sehr zeitaufwendig ist, obwohl die Ausbeute eher gering ist.



(c) Beim Halbraumverfahren ist die Aufteilung des Heaps in zwei Hälften deutlich zu sehen.

Abbildung 8.3.: Simulation der verschiedenen Algorithmen.

fehlschlagen lässt. Dieses Phänomen tritt jedoch nicht auf, wenn die Objektgröße konstant ist. Beim Lisp-2-Algorithmus ist wiederum zu sehen, dass bei geringem Verwaisungsgrad (bzw. hohem Verzweigungsgrad) die Ausbeute des Kollektors eher gering ist, aber bedingt durch die Kompaktierungsphase dennoch viele Objekte verschoben werden, was sehr zeitaufwendig sein kann. Eine ähnliche Kausalität kann ebenfalls beim Halbraumverfahren beobachtet werden, wobei hier zusätzlich die Halbierung des Speicherplatzes ins Auge springt. Objekte, die bei diesem Algorithmus bereits in den anderen Halbraum kopiert wurden, werden hier zusätzlich dunkelblau dargestellt.

8.2 Erweiterung um zusätzliche Algorithmen

Die in Kapitel 7 vorgestellte Modellierung und ihre Umsetzung ermöglicht, den Simulator um eigene Algorithmen zu ergänzen, indem die Klasse `GarbageCollector` spezialisiert bzw. die Schnittstelle `Allocator` implementiert wird. Die im Simulator auswählbaren Garbage-Collection-Algorithmen werden in der Klasse `Settings` durch das Array `collectors` definiert (siehe Listing 8.1). Dieses bildet die Datengrundlage für die `JComboBox` des Auswahlfensters. Somit steht jeder Eintrag des Arrays für einen auswählbaren Algorithmus. Wenn ein implementierter Garbage-Collection-Algorithmus zum Simulator hinzugefügt werden soll, muss das Array entsprechend um einen Eintrag der Klasse `GarbageCollectorSelection` ergänzt werden. Zur Erzeugung eines Objekts dieser Klasse ist folgende Syntax zu verwenden:

```
new GarbageCollectorSelection(<Name>, <Controller>, <Einstellfenster>);
```

Der erste Konstruktorparameter ist dabei der Name, der im Auswahlfenster angezeigt werden soll. Der zweite Parameter ist ein Objekt vom Typ `Runnable`, bei dessen Ausführung ein neuer `CollectionController` instanziiert werden muss. Dabei ist auch eine geeignete Allokator- und Kollektorinstanz zu erzeugen (siehe etwa Zeile 4 bis 7). Der dritte Parameter – ebenfalls vom Typ `Runnable` – ist optional und wird verwendet, um ein zusätzliches Einstellfenster zu erzeugen. Wird der Parameter spezifiziert, ist die Schaltfläche *Einstellungen* im Auswahlfenster des Simulators verfügbar. Das Einstellfenster kann dazu verwendet werden, um durch die Anwenderin weitere Konfigurationsmöglichkeiten festlegen zu lassen, die in der grafischen Benutzerschnittstelle bisher nicht vorgesehen sind.

```
private static GarbageCollectorSelection marksweep =
  new GarbageCollectorSelection(
    "Mark Sweep",
    () -> new CollectionController<NaiveAllocator>(
        false,
        new NaiveAllocator(Settings.heapSize()),
        new MarkSweep<NaiveAllocator>())
);

private static GarbageCollectorSelection lisp2 =
  new GarbageCollectorSelection(
    "Lisp 2",
    () -> new CollectionController<NaiveAllocator>(
        false,
        new NaiveAllocator(Settings.heapSize()),
        new Lisp2<NaiveAllocator>())
);

private static GarbageCollectorSelection semispace =
  new GarbageCollectorSelection(
    "Halbraumverfahren",
    () -> new CollectionController<SemispaceAllocator>(
        false,
        new SemispaceAllocator(Settings.heapSize()),
        new SemispaceCollector<SemispaceAllocator>())
);

...

public static final GarbageCollectorSelection[] collectors = {
    marksweep, lisp2, semispace, naiveAlloc, semispaceAlloc
};
```

Listing 8.1: Spezifikation der verfügbaren Garbage-Collection-Algorithmen in der Klasse `Settings`. Für jeden Algorithmus ist eine Instanz der Klasse `GarbageCollectorSelection` anzulegen und dem Array `collectors` hinzuzufügen.

9 Fazit

In dieser Arbeit wurden auf der Grundlage eines abstrakten Speichermodells verschiedene Garbage-Collection-Algorithmen zur automatischen Speicherverwaltung aufgearbeitet und analysiert. Mit dem Mark-Sweep-Algorithmus und der Referenzzählung haben wir zwei anfangs sehr unterschiedliche Ansätze kennen gelernt, welche die Freigabe nicht mehr benötigten Speichers bewerkstelligen. Durch einige Optimierungen war zu sehen, dass diese scheinbar konträren Ansätze keineswegs unvereinbar sind: Zusätzliche Kollektionsphasen neben der Referenzzählung erleichtern den Umgang mit zyklischen Strukturen und lindern gleichzeitig den Performanceverlust, der mit einer rigorosen Nachverfolgung von Referenzmanipulationen einhergeht. Mit der verborgenen Referenzzählung konnten wir schließlich sehen, dass eine Vereinigung beider Grundverfahren eine vielversprechende Lösung darstellt, um *das Beste zweier Welten* zu kombinieren. Eine wesentliche Rolle spielen dabei auch die zuvor betrachteten Algorithmen, die das zentrale Problem der Speicherfragmentierung durch eine Kompaktierung des Heaps beheben, sowie die Partitionierung des Heaps in disjunkte Speicherbereiche.

Der im zweiten Teil entworfene Garbage-Collection-Simulator ist in der Lage, vorgestellte Algorithmen angemessen zu visualisieren. Durch die angebeotenen Konfigurationsmöglichkeiten ermöglicht er dabei, verschiedene Anwendungsfälle zu simulieren und interessante Eigenschaften der implementierten Algorithmen zu entdecken. Durch eine geschickte Modellierung ist zudem eine Ergänzung um weitere Allokations- und Kollektionsverfahren zwecks ihrer Erprobung möglich. Einziger Wermutstropfen ist jedoch die fehlende Darstellung von Referenzen, wodurch eine Simulation von referenzzählenden Verfahren eher erkenntnisarm sein dürfte. Als Abhilfe ist eine zusätzliche Darstellung des Objektgraphen denkbar, sodass die Verwaisung von Objekten und die Anpassung von Referenzen in kompaktierenden Algorithmen direkt beobachtet werden kann. Diesbezüglich kann auch eine Erhöhung der Interaktivität in Betracht gezogen werden, indem die Anwenderin mithilfe des Objektgraphen selbst die Vernetzung und Verwaisung der Objekte bestimmt. Somit können verschiedene Algorithmen gezielt an festen Beispielszenarien erprobt werden, was mit der vom Zufall abhängigen Implementation aktuell nicht möglich ist. Eine weitere mögliche Ergänzung ist die Integration eines *Steppers*, mit dem die Anwenderin selbst den nächsten Arbeitsschritt eines Kollektors auslöst. Die verwen-

deten Swing-Timer, die auch einen nicht-wiederholenden Modus besitzen, bieten hierfür eine gute Grundlage.

Schlussendlich muss beachtet werden, dass die vorliegende Arbeit maximal einen Überblick über zentrale Algorithmen und Ansätze bieten kann. Allein die von Richard JONES gepflegte *Garbage Collection Bibliography*[1] zählt über 2.500 Publikationen zum Thema Garbage Collection. Darunter sind zahlreiche zu finden, die sich auf spezielle Anwendungsfälle beziehen oder Schwerpunkte wie Nebenläufigkeit, Parallelisierung und Echtzeitsysteme betonen. Auch, wenn diese Konzepte nicht im Detail betrachtet wurden, bietet diese Arbeit eine gute Einführung in die zugrunde liegende Thematik, sodass sie durchaus als Vorbereitung zur Auseinandersetzung mit elaborierteren Garbage-Collection-Algorithmen verstanden werden kann. Darüber hinaus liefert sie Hinweise auf relevante Aspekte, die für die Entscheidungsfindung für oder gegen einen Garbage-Collection-Algorithmus beachtet werden sollten. Dies spielt vor allem für Systeme und Programmiersprachen wie *Java* eine Rolle, bei denen die Entwicklerin aus verschiedenen vorimplementierten Algorithmen wählen kann. Mit dem Aufkommen neuer Anwendungsfälle entsteht auch der Bedarf an hochperformanten und ressourceneffizienten Algorithmen zur automatischen Speicherverwaltung, die an die jeweilige Situation angepasst sind. Insofern lässt sich abschließend festhalten, dass die Thematik der Garbage-Collection-Algorithmen auch in Zukunft von hoher Relevanz ist.

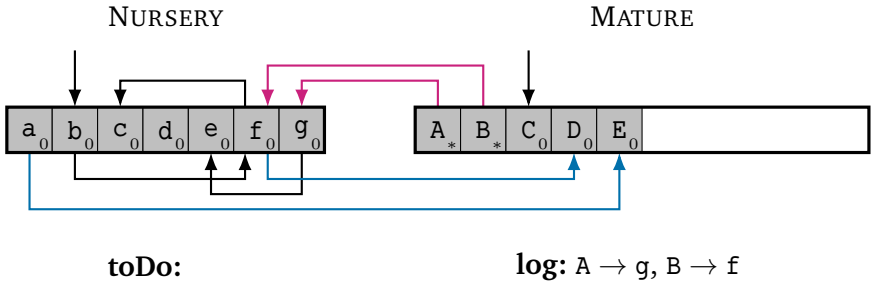
[1] `https://www.cs.kent.ac.uk/people/staff/rej/gcbib/gcbib.pdf`

Anhang

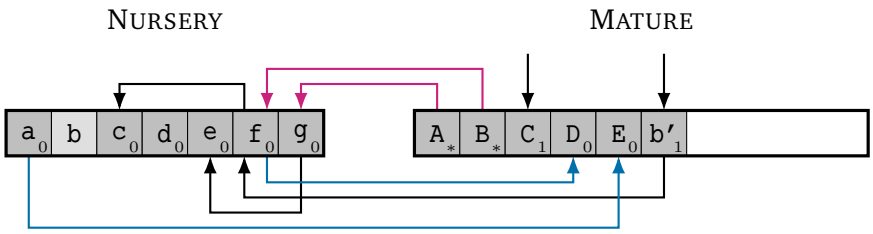
A

Beispiel zur verborgenen Referenzzählung

Es sei die folgende Konstellation von Objekten in NURSERY und MATURE gegeben. Zur Wahrung der Übersichtlichkeit sind Referenzen von NURSERY nach MATURE blau, Referenzen von MATURE nach NURSERY purpur und Referenzen innerhalb von NURSERY **schwarz** eingezeichnet. Die Objekte `b` und `C` seien über Referenzen aus ROOTS erreichbar. In der unteren rechten Ecke eines Objekts wird der Referenzzähler angezeigt. `A` und `B` seien über Referenzen innerhalb von MATURE erreichbar, besitzen also einen positiven Referenzzählerwert, der hier nicht konkret angegeben wird. Da diese Referenzen für Algorithmus 5.3 nicht relevant sind, wurden sie in der Abbildung fortgelassen. Ebenso wurde auf die Darstellung ausgehender Referenzen bereits kopierter Objekte verzichtet.

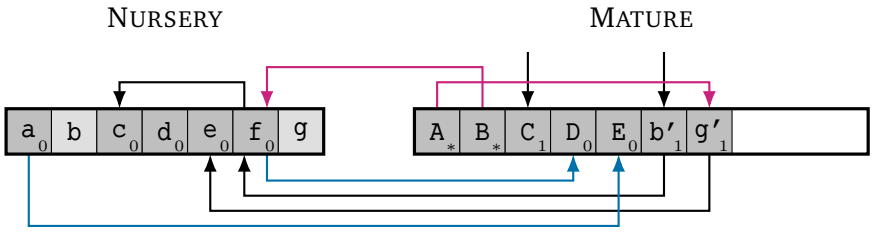


Zunächst werden alle von ROOTS ausgehenden Referenzen verfolgt, wobei die Objekte `b` und `C` entdeckt werden. `b` wird daher nach MATURE kopiert und der Referenzzähler erhöht. Die Kopie `b'` wird zu `toDo` hinzugefügt, um später `b` auf ausgehende Referenzen überprüfen zu können. Da sich `C` bereits in MATURE befindet, wird lediglich dessen Zähler inkrementiert. Damit sind alle Referenzen von Basisobjekten abgearbeitet und die entsprechenden Referenzzähler von Heapobjekten angepasst.

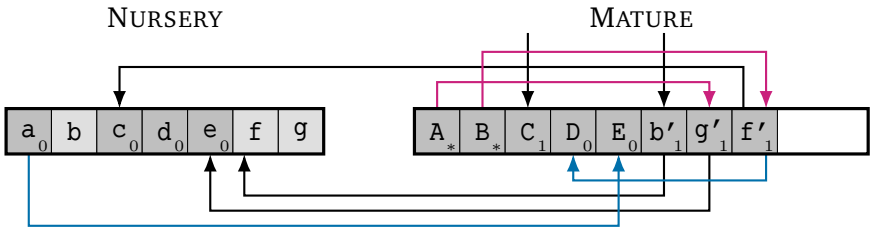


toDo: `b′` **log:** `A` → `g`, `B` → `f`

Nun werden alle Referenzen von MATURE nach NURSERY verfolgt, die in `log` verzeichnet sind. Daher werden die Objekte `g` und `f` nach MATURE kopiert, ihre Kopien zu `toDo` hinzugefügt und die Referenzzähler inkrementiert.

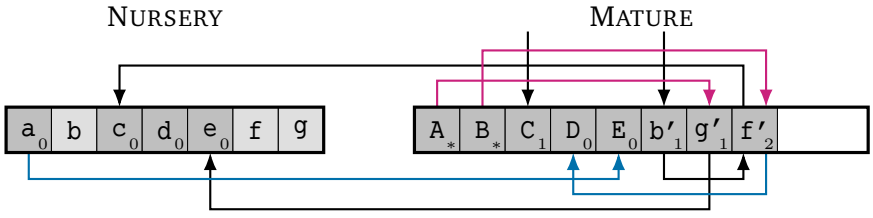


toDo: `b′`, `g′` **log:** `B` → `f`



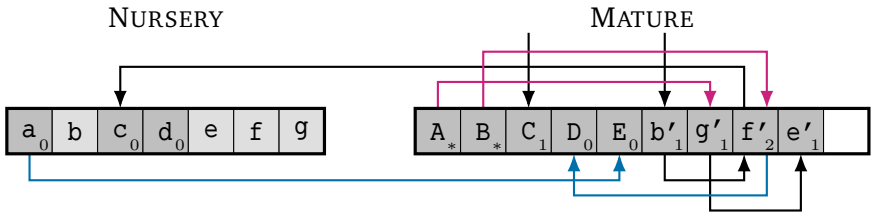
toDo: `b′`, `g′`, `f′` **log:**

Zuletzt erfolgt die Abarbeitung von `toDo`. `b′` enthält eine Referenz auf das bereits kopierte Objekt `f`, daher genügt es, diese Referenz anzupassen und den Referenzzähler von `f′` zu erhöhen.



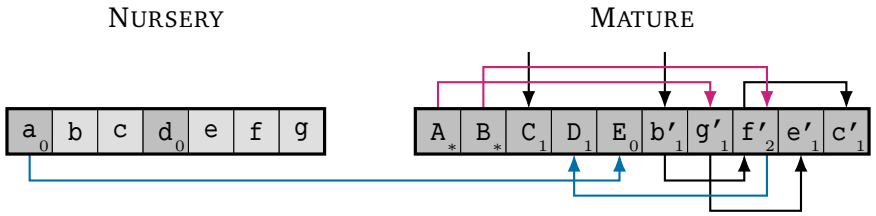
toDo: `g′`, `f′` **log:**

Objekt g′ besitzt eine Referenz auf e, sodass auch dieses Objekt nach MATURE kopiert wird.



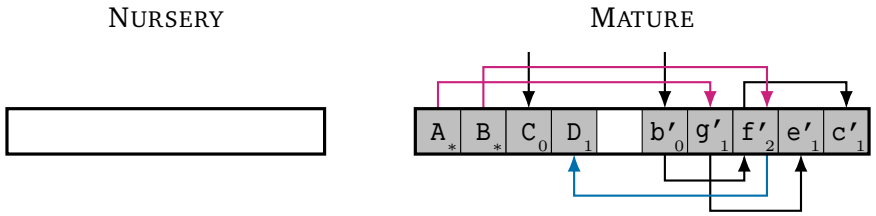
toDo: f′, e′ **log:**

f′ enthält eine Referenz auf das Objekt D, das sich bereits in MATURE befindet, sodass es hier wieder ausreicht, die Referenz anzupassen und den Referenzzähler von D zu inkrementieren. Objekt c, das von f′ referenziert wird, wird kopiert.



toDo: e′, c′ **log:**

Die Objekte e′ und c′ weisen keine Referenzen auf andere Objekte auf, sodass mit ihrer Abarbeitung toDo geleert wird. Es verbleiben a und d in NURSERY, die nicht kopiert wurden, unerreichbar sind und somit entfernt werden. Zugleich wird auch E freigegeben, da sein Referenzzähler 0 beträgt. Nachdem die Referenzzähler von C und b′ um die Referenzen aus ROOTS korrigiert wurden, terminiert der Algorithmus.



Literatur

[ASS96] Harold Abelson, Gerald Jay Sussmann und Julie Sussmann. *Structure and Interpretation of Computer Programs*. 2. Aufl. MIT Press, 1996.

[BCM04] Stephen M. Blackburn, Perry Cheng und Kathryn S. McKinley. „Myths and Realities: The Performance Impact of Garbage Collection“. In: *SIGMETRICS Performance Evaluation Review* 32.1 (2004), S. 25–36.

[BM03] Stephen M. Blackburn und Kathryn S. McKinley. „Ulterior Reference Counting: Fast Garbage Collection Without a Long Wait“. In: *SIGPLAN Notices* 38.11 (2003), S. 344–358.

[Bro84] Rodney A. Brooks. „Trading Data Space for Reduced Time and Code Space in Real-time Garbage Collection on Stock Hardware“. In: LFP ’84 (1984), S. 256–262.

[Che70] C. J. Cheney. „A Nonrecursive List Compacting Algorithm“. In: *Communications of the ACM* 13.11 (1970), S. 677–678.

[Col60] George E. Collins. „A Method for Overlapping and Erasure of Lists“. In: *Commununications of the ACM* 3.12 (1960), S. 655–657.

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein. *Introduction to Algorithms*. 3. Aufl. Cambridge, MA: MIT Press, 2009.

[DB76] L. Peter Deutsch und Daniel G. Bobrow. „An Efficient, Incremental, Automatic Garbage Collector“. In: *Commununications of the ACM* 19.9 (1976), S. 522–526.

[Dij+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten und E. F. M. Steffens. „On-the-fly Garbage Collection: An Exercise in Cooperation“. In: *Communications of the ACM* 21.11 (1978), S. 966–975.

[FT00] Robert Fitzgerald und David Tarditi. „The Case for Profile-directed Selection of Garbage Collectors“. In: *SIGPLAN Notices* 36.1 (2000), S. 111–120.

[FY69] Robert R. Fenichel und Jerome C. Yochelson. „A LISP Garbage-collector for Virtual-memory Computer Systems“. In: *Communications of the ACM* 12.11 (1969), S. 611–612.

[GBF07] Robin Garner, Stephen M. Blackburn und Daniel Frampton. „Effective Prefetch for Mark-sweep Garbage Collection“. In: *Proceedings of the 6th International Symposium on Memory Management*. ISMM ’07 (2007), S. 43–54.

[Hos06] Antony L. Hosking. „Portable, Mostly-concurrent, Mostly-copying Garbage Collection for Multi-processors“. In: ISMM ’06 (2006), S. 40–51.

[Hug82] Robert John Muir Hughes. „A semi-incremental garbage collection algorithm“. In: *Software: Practice and Experience* 12.11 (1982), S. 1081–1082.

[JHM11] Richard Jones, Antony Hosking und Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Boca Raton: CRC Press, 2011.

[JL96] Richard Jones und Rafael Lins. *Gargabe Collection: Algorithms for Automatic Dynamic Memory Management*. Chichester: John Wiley & Sons, 1996.

[JR08] Richard E. Jones und Chris Ryder. „A Study of Java Object Demographics“. In: ISMM ’08 (2008), S. 121–130.

[KJ11] Tomas Kalibera und Richard Jones. „Handles Revisited: Optimising Performance and Memory Costs in a Real-time Collector“. In: *SIGPLAN Notices* 46.11 (2011), S. 89–98.

[KNL07] Martin Kero, Johan Nordlander und Per Lindgren. „A Correct and Useful Incremental Copying Garbage Collector“. In: ISMM ’07 (2007), S. 129–140.

[KP06] Haim Kermany und Erez Petrank. „The Compressor: Concurrent, Incremental, and Parallel Compaction“. In: *SIGPLAN Notices* 41.6 (2006), S. 354–363.

[LH06] Chin-Yang Lin und Ting-Wei Hou. „A Lightweight Cyclic Reference Counting Algorithm“. In: *Advances in Grid and Pervasive Computing* (2006). Hrsg. von Yeh-Ching Chung und José E. Moreira, S. 346–359.

[LH83] Henry Lieberman und Carl Hewitt. „A Real-time Garbage Collector Based on the Lifetimes of Objects“. In: *Communications of the ACM* 26.6 (1983), S. 419–429.

[Lin92] Rafael D. Lins. „Cyclic reference counting with lazy mark-scan“. In: *Information Processing Letters* 44.4 (1992), S. 215–220.

[LP06] Yossi Levanoni und Erez Petrank. „An On-the-Fly Reference Counting Garbage Collector for Java“. In: *ACM Transactions on Programming Language and Systems* 28.1 (2006), S. 1–69.

[McC60] John McCarthy. „Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I“. In: *Communnications of the ACM* 3.4 (1960), S. 184–195.

[McC79] John McCarthy. *Histroy of LISP*. 1979. URL: `http://www-formal.stanford.edu/jmc/history/lisp/lisp.html` (besucht am 7. Jan. 2019).

[MUI13] Kazuya Morikawa, Tomoharu Ugawa und Hideya Iwasaki. „Adaptive Scanning Reduces Sweep Time for the Lisp2 Mark-compact Garbage Collector“. In: *SIGPLAN Notices* 48.11 (2013), S. 15–26.

[MWL90] Alejandro D. Martínez, Rosita Wachenchauzer und Rafael D. Lins. „Cyclic reference counting with local mark-scan“. In: *Information Processing Letters* 34.1 (1990), S. 31–35.

[Pir98] Pekka P. Pirinen. „Barrier Techniques for Incremental Tracing“. In: *SIGPLAN Notices* 34.3 (1998), S. 20–25.

[Pri01] Tony Printezis. „Hot-Swapping between a Mark&Sweep and a Mark&Compact Garbage Collector in a Generational Environment“. In: *Proceedings of the Java™Virtual Machine Research and Technology Symposium* (2001).

[Rob80] John Michael Robson. „Storage allocation is NP-hard“. In: *Information Processing Letters* 11.3 (1980), S. 119–125.

[Sip13] Michael Sipser. *Introduction to the Theory of Computation*. 3. Aufl. Cengage Learning, 2013.

[Str13] Bjarne Stroustrup. *The C++ Programming Language*. 4. Aufl. Addison-Wesley, 2013.

[Sty67] P. Stygar. *LISP 2 Garbage Collector Specification. TM-3417/500/00*. `http://www.softwarepreservation.org/projects/LISP/lisp2/TM-3417_500_00_LISP2_GC_Spec.pdf`. 1967.

[TB14] Andrew S. Tanenbaum und Herbert Bos. *Modern Operating Systems*. 4. Aufl. Pearson, 2014.

[Wil92] Paul R. Wilson. „Uniprocessor Garbage Collection Techniques“. In: IWMM ’92 (1992), S. 1–42.

Diese Masterarbeit wurde mit LaTeX 2$_\varepsilon$ unter Verwendung der Vorlage *Clean Thesis* von Ricardo LANGNER gesetzt. Für mehr Informationen siehe `http://cleanthesis.der-ric.de/`.

Abbildungsverzeichnis

Algorithmenverzeichnis

Listing-Verzeichnis

Eigenständigkeitserklärung

Hiermit versichere ich, dass die vorliegende Masterarbeit mit dem Titel *Garbage-Collection-Algorithmen und ihre Simulation* selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

Ich erkläre mich mit einem Abgleich der Arbeit mit anderen Texten zwecks Auffindung von Übereinstimmungen sowie mit einer zu diesem Zweck vorzunehmenden Speicherung der Arbeit in eine Datenbank einverstanden.

(Ort, Datum) (Unterschrift)