Todo list

4. Punkt-Verknüpfungen
5. not $\rightarrow$ **not**, Empty $\rightarrow \emptyset$
6. Transitive Hülle
7. Übergänge zwischen Kapiteln
8. Methode
9. Bildungssprache
10. Seitenumbrüche

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

Masterarbeit

Garbage-Collection-Algorithmen und ihre Simulation

Phil Steinhorst

Erstgutachter und Betreuung Prof. Dr. Jan Vahrenhold

Zweitgutachter Prof. Dr. Markus Müller-Olm

Münster, 21. Januar 2019

Garbage-Collection-Algorithmen und ihre Simulation
Masterarbeit zur Erlangung des akademischen Grades *Master of Education*
in den Fächern Mathematik und Informatik
Erstgutachter und Betreuung: Prof. Dr. Jan Vahrenhold
Zweitgutachter: Prof. Dr. Markus Müller-Olm
Münster, 21. Januar 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

Zusammenfassung

Die vorliegende Arbeit soll eine Aufarbeitung verschiedener Ansätze für Garbage-Collection-Algorithmen liefern. Nach einer kurzen Darstellung der zugrunde liegenden Problematik und deren praktische Relevanz sowie den Vor- und Nachteilen einer automatischen Speicherverwaltung gegenüber einer manuellen Speicherverwaltung werden gängige Ansätze vergleichend vorgestellt sowie Einsatz und Eignung in der Praxis beurteilt. Als Gütekriterien dienen hier beispielsweise 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. Dazu gehört eine angemessene Visualisierung eines beschränkten Speicherbereichs, etwa durch eine optische Unterscheidbarkeit belegter Blöcke, sowie der einzelnen Arbeitsphasen, die eine Garbage Collection ausführt. Dabei sollen auch unterschiedliche Szenarien auswählbar sein, etwa verschiedene Speicherfüllstände und eine variable Anzahl bzw. Größe von Objekten, die im Speicher hinterlegt sind.

Am Ende nochmal schauen, ob das wirklich so ist :D

Abstract

Englisch einfügen.

Inhaltsverzeichnis

1 Einleitung

Die Möglichkeiten einer dynamischen Speicherverwaltung haben sich in den meisten modernen Programmiersprachen etabliert. Die Vorteile, einen Teil des dynamischen Speichers – oft auch als *Heap* bezeichnet – 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 Überschreitung hartkodierter Speicherbereiche vermeidet.

JV: In welchen Sprachen nicht?

s. JV

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 so genannten *hängenden Zeigern* (engl. *dangling pointer*) – 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 Komfortgewinn 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 wollen oder in denen Deadlines um jeden Preis eingehalten werden müssen, spielt die Auswahl eines geeigneten Garbage-Collection-Algorithmus eine signifikante Rolle.

JV: evtl. auf WCET in EmbSys hinweisen?

In dieser Arbeit werden wir gängige Ansätze zur Garbage Collection vorstellen und miteinander vergleichen. Dabei soll auch ein Augenmerk auf Performance und Ressourcenbedarf gelegt sowie die Eignung in verschiedenen Anwendungsfällen beurteilt werden. 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 soll die Arbeitsweise der Algorithmen veranschaulicht werden.

Konkretisieren

1.1 Terminologie

Bevor wir genauer darauf eingehen, was unter einer Garbage Collection konkret verstanden wird, sollen zunächst die nötige Terminologie sowie ein Speichermodell eingeführt werden, das im Fortgang dieser Arbeit benutzt wird. Dieses Speichermodell ist bewusst so abstrakt gehalten, dass es möglichst allgemeine Betrachtungen lösgelöst von gängigen 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

Unter einem **Objekt** verstehen wir 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.

Wir setzen ferner voraus, 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 nehmen wir an, dass jedes Objekt einen so genannten *Header* besitzt. Dies ist ein separates Feld, das Metainformationen aufnimmt, die für den Compiler und die Laufzeitumgebung, nicht aber aus Sicht des Entwicklers, zugänglich sind. Diverse vorgestellte Algorithmen werden diesen Bereich nutzen, um für die Speicherverwaltung relevante Informationen zu hinterlegen.

evtl genauer eingehen

Den Zugriff auf das `i`-te Feld eines Objekts `a` notieren wir – analog zur Syntax der Programmierspache C – mit `a[i]`. Ebenso bezeichnen wir mit `&a` die Adresse eines Objekts und mit `*p` die Dereferenzierung eines Zeigers `p`. Mit POINTERS(`a`) bezeichnen wir zudem die Menge aller Felder eines Objekts `a`, die eine Referenz enthalten können.

Relativisieren

Heap

Als **Heap** bezeichnen wir denjenigen Speicherbereich, in dem zur Laufzeit eines Programms Objekte in beliebiger Reihenfolge erzeugt und freigegeben werden können. Der Heap besteht aus Blöcken einer festen Größe, auf die über eine Speicheradresse zugegriffen werden kann; ein *Block* ist dabei die kleinste zuweisbare Speichermenge und kann die Zustände *belegt* (bzw. *zugewiesen*) oder *frei* annehmen. Sofern nichts anderes vereinbart ist, gehen wir davon aus, dass der Heap ein zusammenhängender linearer Speicherbereich ist.[1]

Grafik zu Heap und Objekten einfügen

[1] 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 zu Verfügung gestellt wird und aus ihrer Sicht linear zusammenhängend ist. Für einen Überblick hierzu siehe etwa [TB14, Kap. 3.3].

Allokator, Mutator und Kollektor

Aufgabe des **Allokators**, der zur Laufzeitumgebung eines Programms gehört, ist zum einen die Zuweisung von Heapspeicher bei dynmaischer Instanziierung eines neuen Objektes und zum anderen die Freigabe von Objekten. Der Allokator führt somit Buch über die belegten und freien Blöcke des Heaps. Die genaue Realisierung dieser Mechanismen werden in dieser Arbeit weitestgehend außen vor gelassen, jedoch setzen wir in gewissen Situationen das Vorhandensein bestimmter Funktionalitäten voraus. Beispielsweise verlangen wir, dass eine Prozedur *new* zu 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 (siehe Algorithmus 1.1).

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:         collectGarbage()                ▷ Aufruf der Garbage Collection
5:         adr ← allocate()                ▷ Neuer Versuch
6:         if adr = null
7:             error("Nicht genügend Speicher")
8:     return adr
```

Nach DIJKSTRA et al. besteht ein Programm 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. Für uns sind dabei vor allem Programmanweisungen von Bedeutung, die in Feldern von Objekten vorhandene Referenzen manipulieren und somit ursächlich für die Entstehung von nicht mehr benötigten Objekten sind. Im Gegensatz dazu ist es 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, können wir nun definieren, was wir unter einer Garbage Collection verstehen. Anschließend folgt eine Spezifikation von Eigenschaften, die wir von einem Garbage-Collection-Algorithmus fordern.

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 bezeichnen wir das Objekt als **nicht mehr benötigt**.

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.

Beweis: 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 nicht (vgl. [Sip13, Kap. 4.2]). □

Aus diesem Grund betrachten wir eine schwächere Eigenschaft von Objekten: die Erreichbarkeit über Referenzen. Dafür gehen wir von einer Menge ROOTS von **Basisobjekten** (engl. *root objects*) aus. 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 Position im Speicher bereits zur Compilezeit bekannt ist, oder Objekte, die sich in *stack frames* befinden. Alle weiteren Objekte, die zur Laufzeit dynamisch definieren 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 definieren wir diese Eigenschaft wie folgt:

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 $i \in \mathbb{N}$ mit $\texttt{*a[i]} = \texttt{b}$, so folgt $\texttt{b} \in \mathcal{R}$.

Gilt $\texttt{*a[i]} = \texttt{b}$, so schreiben wir auch $\texttt{a} \to \texttt{b}$. Existiert für zwei Objekte a, b eine endliche Folge von Objekten $a_1, \ldots, a_n$ mit $a \to a_1 \to \cdots \to a_n \to b$, so notieren wir dies zudem mit $a \xrightarrow{*} b$.

Diese Definition garantiert zwar nicht, dass jedes erreichbare Objekt auch lebendig ist. Davon ausgehend, dass unerreichbare Objekte auch nicht *wiedergefunden* werden können, können wir 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 zu einem Knoten des Graphen, sodass wir das Objekt mit seinem Knoten identifizieren können (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 ensprechenden Knoten dargestellt. Ein Objekt ist somit nicht erreichbar, wenn es im Objektgraphen keinen Pfad zu ihm gibt, der in einem Basisobjekt startet. Objektgraphen werden uns im Rahmen dieser Arbeit bei der Veranschaulichng 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 $\texttt{a}, \texttt{b} \in O$ gilt:

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

An dieser Stelle formulieren wir ein Korrektheitskriterium für Garbage-Collection-Algorithmen. Dieses besteht aus der Anforderung, dass keine noch benötigten Daten zerstört werden.

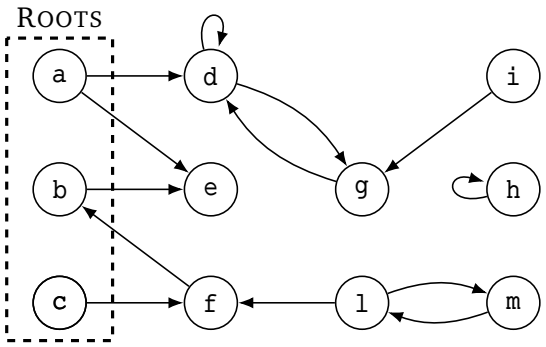


Abbildung 1.1.: Beispiel für einen Objektgraphen. Die Objekte `a`, `b` und `c` sind Basisobjekte. Die Objekte `h`, `i`, `l` und `m` sind in dieser Konstellation nicht erreichbar.

Definition 1.5 (Korrektheit für 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.

Man kann an dieser Stelle fragen, warum wir nicht voraussetzen, dass die Ausführung eines Garbage-Collection-Algorithmus die Freigabe *sämtlicher* nicht erreichbaren Objekte anfordert. Tatsächlich werden wir sehen, 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 and Sweep

Wir beginnen 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. Diese Struktur hat den Vorteil, dass ein Datum, das in mehreren Listen vorkommt, 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 zu Verfügung stehen. Diese zweischrittige Vorgehensweise – das Erkennen nicht mehr benötigter Speicherbereiche und die anschließende Freigabe eben jener – bildet die Grundlage des *Mark-Sweep-Algorithmus*.

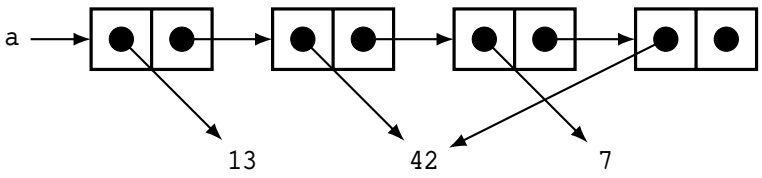


Abbildung 2.1.: Visualisierung der LISP-Liste `a = (13,42,7,42)` als *Box-and-Pointer*-Diagramm (vgl. [ASS96, Kapitel 3.3]).

2.1 Naives Mark and Sweep

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 zu Verfügung.

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

```
 1: collectGarbage():
 2:     markStart()
 3:     sweepHeap()

 4: markStart():
 5:     toDo ← ∅                                  ▷ Noch abzuarbeitende Objekte
 6:     for each obj ∈ ROOTS                      ▷ Beginne mit Basisobjekten
 7:         if isNotMarked(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 (ref ≠ null ∧ isNotMarked(*ref))
16:                 setMarked(*ref)
17:                 add(toDo, *ref)
```

Die Markierungsphase (engl. *mark*) 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, um zu einem späteren Zeitpunkt abgearbeitet zu werden (Zeile 15 und 16). 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 ggfs. das nächste Basisobjekt abgearbeitet wird.

Kleines Beispiel hier, großes in den Anhang?

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 können wir festhalten, dass `toDo` zu jedem Zeitpunkt ausschließlich bereits markierte Objekte enthält. Da keine Objekte hinzugefügt werden, die bereits markiert wurden (Zeile 7 und 16), 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 wir voraussetzen, dass während der Ausführung des Garbage Collectors 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 – Bereinigung (vgl. [JL96, Kap. 2.2])

```
1: sweep():
2:     pos ← nextObject(HEAP_START)
3:     while pos ≠ null
4:         if isMarked(*pos)
5:             unsetMarked(*pos)
6:         else free(pos)
7:         pos ← nextObject(pos)
```

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. Wir gehen davon aus, dass eine Prozedur *nextObject* des Allokators zu 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.

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:
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 Objekte freigegeben (Zeile 4–6 in Algorithmus 2.2). Bleibt ein Objekt `obj` unmarkiert, so ist `obj` kein Basisobjekt, da *markStart* alle Basisobjekte markiert. Somit wird kein Basisobjekt freigegeben. 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 erreichbaren Objekte entfernt werden, wie folgendes Beispiel zeigt (vgl. [Dij+78, S. 969]): Betrachten wir 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 erreichbar ist. In der Folge würde `c` irrtümlich freigegeben werden, sodass im schlimmsten Fall 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. [JHM11, S. 17]) bezeichnet.

2.2 Drei-Farben-Abstraktion

Da das Anhalten des Mutators während eines gesamten Garbage-Collection-Zyklus zu vergleichsweise großen Verzögerungen führt, ist eine Optimierung der Markierungsphase 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 auf die Idee kommen, beim Schreiben einer neuen Referenz in ein Feld eines Objekts das Ziel dieser Referenz sofort zu 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. Wir modifizieren den ursprünglichen Mark-Sweep-Algorithmus 2.1 so, 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.

[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.

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.

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

Vorbedingung: Alle Objekte sind weiß markiert.

```
 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 (ref ≠ null ∧ 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)
```

Mithilfe dieser **Drei-Farben-Abstraktion** (engl. *tri-color abstraction*) können wir nun Referenzmanipulationen zulassen, 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 zu `graySet` hinzuzufügen, sofern dies noch nicht der Fall ist oder die Felder von `b` bereits verfolgt wurden. Dies kann etwa mit einer *Schreibbarriere* (engl. *write barrier*) realisiert werden, die genau dann zum Einsatz kommt, wenn 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.

Input: Objekt `obj`, in dem Referenz gesetzt wird; Index des Feldes `i`; zu setzende Referenz `ref`

```
1: atomic writeRef(obj, i, ref):
2:   if (ref ≠ null ∧ 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:
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: Wir halten zunächst fest, 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. Wir zeigen daher die Gültigkeit folgender Schleifeninvariante für die **while**-Schleife:

(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 laut Vorbedingung 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 Objekte 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 einen 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 besteht die Möglichkeit, die Invariante (A) abzuschwächen und damit eine *feingranularere* Lösung zu ermöglichen. Für Details hierzu verweisen wir auf die Publikation von DIJKSTRA et al [Dij+78, S. 972ff]. PIRINEN liefert darüber hinaus einen Überblick über verschiedene Möglichkeiten von 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 für 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.

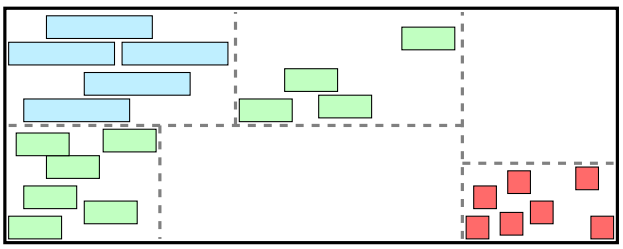


Abbildung 2.2.: Veranschaulichung eines in Regionen aufgeteilten Heaps. Jede Region kann Objekten einer festgelegten Größenordnung zugewiesen werden. Für eine Größenordnung können allerdings mehrere Regionen verwendet werden.

Bei der verzögerten Bereinigung ist der Heapspeicher in *Regionen* (engl. *blocks*) eingeteilt. Jede Region enthält Objekte einer festgelegten Größenordnung; zu jeder Größenordnung können mehrere Regionen existieren (Abbildung 2.2). 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 Regionen, die der angeforderten Speichermenge zugeordnet sind. Die Bereinigung beschränkt sich dadurch auf einen Bruchteil des Heaps.

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 isNotMarked(blk)
17:       free(blk)                  ▷ Gib komplett verwaiste Region sofort frei
18:     else add(toSweep, blk)

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

Dieser Ansatz geht mit einigen marginale Änderungen an unseren bisher verwendeten Algorithmen einher. Zum einen setzen wir voraus, dass beim Markieren eines Objekts auch die entsprechende Region markiert wird, in dem sich das Objekt befindet. Dies kann recht einfach realisiert werden, indem bei der Initialisierung einer Region durch den Allokator zusätzlich Platz für einen Header reserviert wird, welcher Metadaten wie die Größenordnung der enthaltenen Objekte aufnimmt. Weiter verlangen wir, dass der Allokator zusätzlich Informationen über Anzahl, Lage und Größenordnung der existierenden Region besitzt. Zuletzt wandeln wir die Prozedur *sweep* aus den Algorithmen 2.2 und 2.3 dahingehend ab, 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 Regionen festgestellt werden, welche nur verwaiste Objekte enthalten. In diesem Fall kann der Allokator eine Region ad hoc freigeben (Zeile 17 von Algorithmus 2.5). Hingegen werden Regionen, 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 Regionen 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 Regionen ausgeschöpft sind, beginnt ein *Lazy-Sweep-Zyklus* (Zeile 20 bis 24). Dabei werden nach und nach Regionen aus `toSweep` entfernt und bereinigt, die Objekte der angeforderten Speichergröße verwalten. Sobald eine Region um ein Objekt reduziert werden konnte, wird der Zyklus beendet und der Allokator kann den gewonnenen Speicher neu zuordnen (Zeile 22f). Falls aus keiner Region ein Objekt freigegeben werden konnte, bedeutet dies, dass alle Regionen der entsprechenden Objektgröße mit erreichbaren Objekten gefüllt sind. In diesem Fall muss der Allokator eine neue Region initialisieren, die neue Objekte aufnehmen kann (Zeile 25).

Was geschieht nun, wenn nicht genügend freier Speicher zu Verfügung steht, um eine weitere Region zu erzeugen? In diesem Fall 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, sodass zuletzt Platz innerhalb einer Region bzw. für eine neue Region geschaffen werden kann (Zeile 8).

Während die Einschränkung der Bereinigung auf einzelne Regionen 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 Regionen einer Größenordnung angelegt werden, die jeweils nur wenige Objekte enthalten und größtenteils ungenutzt sind. Wenn jede Region allerdings mindestens ein erreichbares Objekt enthält, können sie nicht in Gänze freigegeben werden. Eine hohe Zahl dieser ineffizient

genutzter Regionen führt dazu, dass möglicherweise nicht mehr genügend freier Speicher zu Verfügung steht, um Regionen 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 Regionen gleicher Größenordnung zusammengelegt werden. Dies bedeutet jedoch zusätzlicher 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 Regionen, die keine erreichbare 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 Regionen einer bestimmten Größenordnung besonders stark beansprucht werden. Der Vorteil, nur einen 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. [JHM11, S. 19]).

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, sodass Objekte ohne Referenzfelder übersprungen werden können und die Garbage Collection das Caching weniger stark beeinträchtigt. Im Falle eines in Regionen eingeteilten Heaps bietet es sich beispielsweise an, pro Region eine Bitmap anzulegen. Somit kann während der Bereinigung schnell erkannt werden, ob größere Teile einer Region freigegeben werden können, ohne sie komplett traversieren zu müssen.

Wir beenden dieses Kapitel mit einer Betrachtung der Komplexität der vorgestellten Mark-Sweep-Varianten. 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 Mark-Sweep-Zyklus muss jedoch die Menge `toDo` der noch zu untersuchenden Objekte verwaltet werden. Da wir bereits festgestellt haben, 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 ein erreichbares Objekt nicht 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].

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 ist $\mathcal{O}(|\mathcal{R}|)$.

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

(3) Die Laufzeit der Bereinigungsphase ist $\mathcal{O}(|\mathbb{H}|)$, 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 Regionen 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]).

3 Referenzzählung

Der zweite Garbage-Collection-Algorithmus, der in dieser Arbeit vorgestellt wird, stammt von COLLINS und aus dem selben Jahr wie der Mark-Sweep-Algorithmus. COLLINS betrachtet wie MCCARTHY das *LISP Programming System* auf IBM-Großrechnern und das Problem, wann Listenelemente, die prinzipiell in mehreren Listen vorkommen können, wieder freigegeben werden dürfen. MCCARTHYS Ansatz bezeichnet er jedoch 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]. Diesen Ansatz – die *Referenzzählung* (engl. *reference counting*) – werden wir in diesem Kapitel ausführlich betrachten.

3.1 Naive Referenzzählung

Zur Realisierung des Algorithmus von Collins in unserem Speichermodell nutzen wir erneut den Header eines Objekts $\mathtt{a}$, um dort einen ganzzahligen, nicht negativen Wert $\mathtt{rc}(\mathtt{a})$ zu hinterlegen, den wir als *Referenzzähler* bezeichnen. 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 erhalten wir unmittelbar eine notwendige Bedingung für die Erreichbarkeit eines Objekts: 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 $\mathtt{rc}(\mathtt{a}) \geq 1$.

ROOTS als Objekt?

Lemma 3.1 (Notwendige Bedingung für Erreichbarkeit)**:**
Für ein Objekt $\mathtt{a}$ 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 $\mathtt{a}$ somit nicht garantiert werden kann. Jedoch folgt als Kontraposition von Lemma 3.1

sofort, dass ein Objekt 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, nachdem eine Referenz manipuliert wurde. 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]).

Input: Objekt obj, in dem Referenz gesetzt wird; Index des Feldes i; zu setzende Referenz ref

```
 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 mittels $\texttt{obj}[i] \leftarrow \texttt{ref}$ in ein Feld eines Objekts eine Referenz 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 Herabsetzen 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).

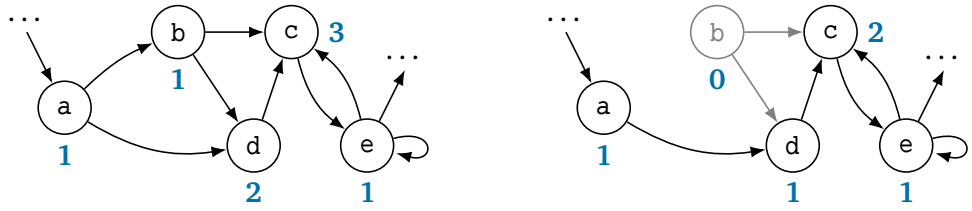


Abbildung 3.1.: Wird die Referenz $\texttt{a} \rightarrow \texttt{b}$ entfernt, so wird `b` freigegeben, da der Referenzzähler von `b` auf 0 fällt. In der Folge müssen auch die Referenzzähler von `c` und `d` angepasst werden.

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 naive Referenzzählung bietet im Vergleich zum Mark-Sweep-Algorithmus einige Vorteile, aber auch gewisse Nachteile (vgl. [LH06, S.

346]): Der Mechanismus zur Freigabe von Objekten wird durch die Entstehung von verwaisten Objekten ausgelöst und nicht etwa, wenn unzureichend freier Speicher zu Verfügung steht. Dadurch wird die Allokation von Speicher für neu erzeugte Objekte nicht verzögert. Verwaiste Objekte werden beim Löschen ihrer letzten Referenz unmittelbar erkannt und freigegeben und nicht erst beim nächsten Mark-Sweep-Zyklus. Speichermangel kann so de facto nur dadurch entstehen, dass der gesamte Speicher von erreichbaren Objekten belegt wird – in diesem Fall bietet allerdings kein Garbage-Collection-Algorithmus Abhilfe. 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: Betrachten wir 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, die die Zähler des zuletzt betrachteten Objekts reduzieren und 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.

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).

Algorithmus 3.2 Lineare Suche in einer verketteten Liste. Obwohl der Algorithmus keine Objekte manipuliert, verursacht jede Änderung an der Referenzvariablen `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
```

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.

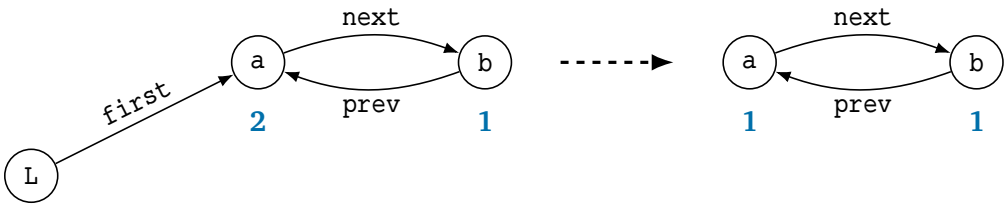


Abbildung 3.2.: Referenzzählung in zyklischen Datenstrukturen. Wird Objekt `L` entfernt, sind `a` und `b` nicht mehr erreichbar. Jedoch fallen ihre Referenzzähler nicht auf 0, sodass sie als Speicherlecks im Speicher verbleiben.

3.2 Zyklische Referenzen

Die fehlende Möglichkeit zur Erkennung und Freigabe zyklischer Strukturen kann – bei entsprechend häufiger Verwendung – große Teile des Heaps brachliegen lassen. Ein gelegentlich zusätzlich ausgeführter Mark-Sweep-Zyklus mag diese Problematik zwar beheben, macht allerdings die oben genannten Vorteile der Referenzzählung zunichte. Nachfolgend stellen wir einen Algorithmus von MARTÍNEZ, WACHENCHAUZER und LINS vor, der die naive Referenzählung 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 `a`, wenn gilt:

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

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

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

Eine Referenz $\texttt{x} \rightarrow \texttt{a}$ von a heißt **intern**, wenn $\texttt{x} \in \langle \texttt{a} \rangle$. 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]

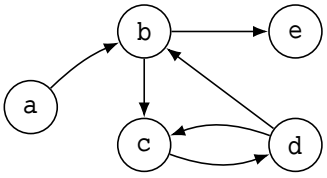


Abbildung 3.3.: Der abgebildete Objektgraph besitzt drei starke Zusammenhangskomponenten: $\langle \texttt{a} \rangle = \{\texttt{a}\}$, $\langle \texttt{b} \rangle = \{\texttt{b}, \texttt{c}, \texttt{d}\}$ und $\langle \texttt{e} \rangle = \{\texttt{e}\}$.

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 der Komponente 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.

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:       if (ref ≠ null ∧ *ref ≠ obj)
 6:         decRefCount(*ref)
 7:     free(&obj)
 8:   else
 9:     markGray(obj)          ▷ Entferne Zählung interner Referenzen
10:     scan(obj)              ▷ Prüfe Erreichbarkeit
11:     collect(obj)           ▷ Entferne unerreichbare Strukturen
```

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

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

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 8 bis 11). In diesem Fall muss überprüft werden, ob `obj` das letzte Objekt einer Struktur war, durch welches diese erreichbar war. Dazu wird nach einem dreischrittigen Verfahren vorgegangen, das im Folgenden beschrieben wird.

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)
 5:       if (ref ≠ null ∧ *ref ≠ obj)          ▷ Entferne interne Referenz,
 6:         rc(*ref) ← rc(*ref) − 1                  die von obj ausgeht
 7:         markGray(*ref)                      ▷ Reduziere rekursiv weitere rc

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

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 6 in Algorithmus 3.4). Durch einen rekursiven Aufruf in Zeile 7 werden somit die starke Zusammenhangskomponente $\langle$`obj`$\rangle$ behandelt und grau markiert; ihre Referenzzähler enthalten anschließend die Anzahl aller externen Referenzen.

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 10f). Das bedeutet, dass `obj` nur 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 $\langle$`obj`$\rangle$ aufgerufen (Zeile 12 bis 14).

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 $\langle$`a`$\rangle$ 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 7). Dabei werden gleichzeitig die zugehörigen Referenzzähler angepasst, sodass nach Beendigung diese wieder mit der Anzahl aller Referenzen auf ein Objekt übereinstimmen (Zeile 5).

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:     if (ref ≠ null ∧ *ref ≠ obj)
 5:       rc(*ref) ← rc(*ref) + 1                   ▷ Interne Referenz wiederherstellen
 6:       if isNotWhite(*ref)
 7:         unmark(*ref)

 8: collect(obj):
 9:   if isBlack(obj)                               ▷ Objekt ist nicht erreichbar
10:     for each ref ∈ POINTERS(obj)
11:       if (ref ≠ null ∧ *ref ≠ obj)
12:         collect(*ref)
13:     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, betrachten wir den Objektgraphen aus Abbildung 3.3 und nehmen an, dass die Referenz `a` $\rightarrow$ `b` entfernt wird, wodurch die Objekte `b`, `c`, `d` und `e` unerreichbar werden. Wir erwarten daher, dass der Aufruf von *markGray*(`b`), *scan*(`b`) und *collect*(`b`) alle vier Objekte freigibt.

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`).

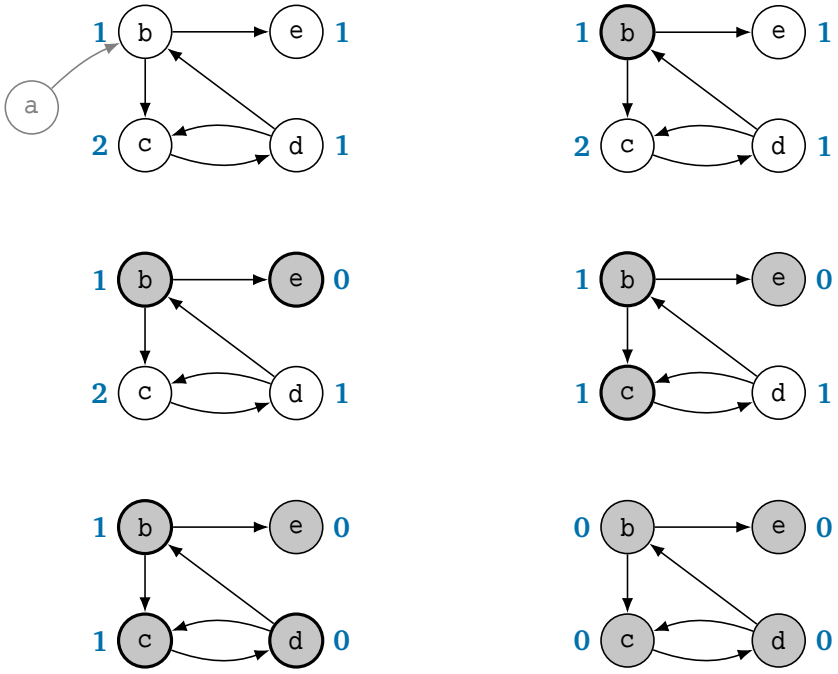


Abbildung 3.4.: Beispielhafte Ausführung von *markGray*(b) nach Entfernen der Referenz a → b. Dick umrandet sind diejenigen Objekte, für welchs gerade der rekursive Aufruf von *markGray* abgearbeitet wird.

An dieser Stelle können wir bereits erkennen, dass alle Referenzzähler von ⟨b⟩ 0 sind, weswegen der Zyklus 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? Wir ergänzen das Beispiel um eine externe Referenz auf das Objekt d (siehe Abbildung 3.5). Nach Terminierung von *markGray*(b) weist rc(d) dann den Wert 1 auf. Durch den Aufruf *scan*(b) werden zunächst wie zuvor 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 des Zyklus wird entfernt.

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

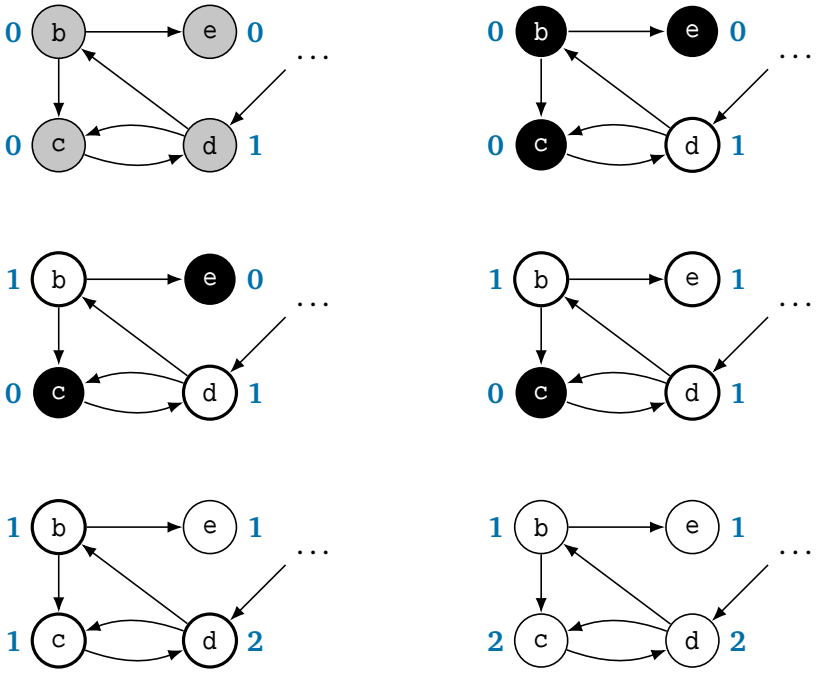


Abbildung 3.5.: Beispielhafte Ausführung von *unmark*(d) nach *markGray*(b).

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 9 bis 11 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 9 und 13 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 $\mathtt{b} \in \langle \mathtt{a} \rangle$ 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 $\mathtt{b} \xrightarrow{*} \mathtt{a}$. 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 naives Mark and Sweep. 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 – entsorgt werden [LH06].

3.3 Optimierungsmöglichkeiten

Abschließend gehen wir auf zwei Ansätze ein, die eine Effizienzsteigerung der Referenzzählung bezwecken sollen. Wir haben bereits festgestellt, 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:       if (ref ≠ null ∧ *ref ≠ obj)
 5:         incRefCount(*ref)
 6:       if (obj[i] ≠ null ∧ *obj[i] ≠ obj)
 7:         decRefCount(*obj[i])
 8:   obj[i] ← ref

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

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

21: atomic collect():
22:   for each ref ∈ POINTERS(ROOTS)         ▷ nicht gezählte Referenzen hinzufügen
23:     if ref ≠ null
24:       incRefCount(*ref)
25:   for each obj ∈ ZCT
26:     for each ref ∈ POINTERS(obj)
27:       if (ref ≠ null ∧ *ref ≠ obj)
28:         decRefCount(*ref)
29:     free(obj)
30:   for each ref ∈ POINTERS(ROOTS)         ▷ vorige Anpassung korrigieren
31:     if ref ≠ null
32:       decRefCount(*ref)
```

Wird nun eine Referenz ref in ein Feld obj[i] geschrieben, kann der Schreibzugriff unsynchronisiert ausgeführt werden, wenn 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 10f). Ist es in der MRT, so ist der zugehörige Referenzzähler MRT(obj) zu erhöhen (Zeile 12f).

Ist beides nicht der Fall, so muss `obj` zur `MRT` hinzugefügt werden; der Referenzzähler wird entsprechend mit 2 initialisiert (Zeile 14). Analoge Fallunterscheidungen sind nötig, wenn ein Referenzzähler reduziert wird (Zeile 16 bis 20).

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 nicht entfernt werden dürfen, müssen zunächst ihre Referenzzähler angepasst werden, indem über alle Referenzen von Basisobjekten iteriert wird (Zeile 22 bis 24). Objekte, die im Anschluss in `ZCT` verbleiben, sind tatsächlich verwaist und können bedenkenlos entfernt werden, ohne die Korrektheit des Algorithmus zu gefährden (Zeile 25 bis 29).

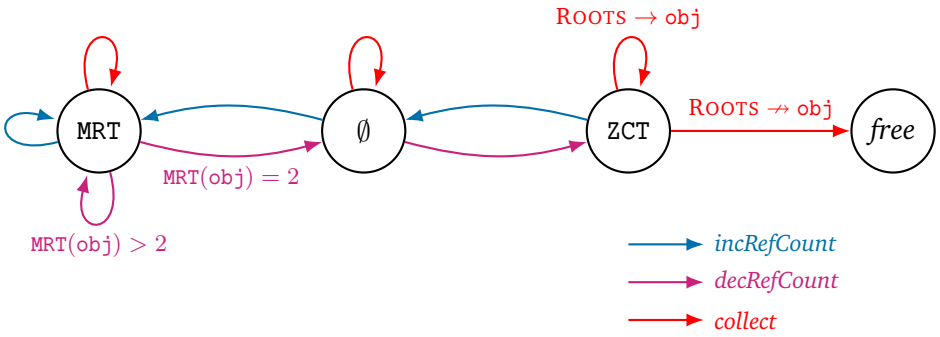


Abbildung 3.6.: Schema zur Veranschaulichung der verzögerten Referenzzählung.

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 unmmittelbar, 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 durch Datenstrukturen zu Gute. 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.[2]

[2]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 zu Verfügung stehenden Speicher einschränkt (vgl. [Cor+09, S. 253])

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 des selben Feldes unnötige Anpassungen von Referenzzählern verursachen, die sich gewissermaßen *kürzen* lassen [LP06, S. 4ff]. Betrachten wir etwa einen Mutator, der folgenden Code ausführt, und die dadurch ausgelösten Anpassungen von Referenzzählern:

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

Offensichtlich wird die Erhöhung der Referenzzähler von $\mathtt{a}_1$ bis $\mathtt{a}_{n-1}$ in der jeweils nächsten Anweisung annuliert, sodass es sinnvoll ist, diese Manipulationen zu aggregieren und lediglich die Zähler von $\mathtt{a}_0$ und $\mathtt{a}_n$ angepasst werden. 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 wir hier nicht näher eingehen, lässt sich dieser Ansatz durch feingranularere Synchronisationen optimieren, etwa indem Logtabellen threadweise angelegt werden. Dadurch kann vermieden werden, dass das gesamte Programm angehalten wird. Für mehr Details verweisen wir auf die Publikation von LEVANONI und PETRANK [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:             if *field ≠ null
10:                 incRefCount(**field)     ▷ vgl. Algorithmus 3.6
11:             if log(field) ≠ null
12:                 decRefCount(*log(field)) ▷ vgl. Algorithmus 3.6
13:         unsetModified(field)             ▷ Markierung entfernen
14:         removeLog(field)                 ▷ Entferne Eintrag aus Tabelle
15:     collect()                            ▷ vgl. Algorithmus 3.6
```

4 Kompaktierung

Mit dem Mark-Sweep-Algorithmus und der Referenzzählung haben wir zwei grundlegende Ansätze kennen gelernt, die eine Wiederverwendung bereits genutzten Speichers durch Freigabe nicht mehr benötigter Objekte realisieren. Außen vor gelassen wurde bislang jedoch ein Phänomen, das durch 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. 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.

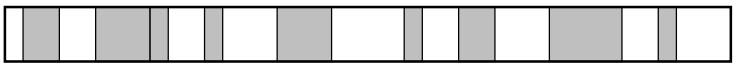


Abbildung 4.1.: Ein stark fragmentierter Heap erschwert die Allokation größerer Speichermengen, auch wenn ein Großteil des Heaps ungenutzt ist.

Die Vermeidung von Heapfragmentierung durch optimale Allokation ist nicht zielführend, da die Prognose zukünftiger Allokationen in der Regel unmöglich, zumindest aber NP-schwer ist (vgl. [Rob80]). Allerdings existieren akzeptable Lösungen, die die Fragmentierung des Heaps durch geschickte Allokation in Grenzen halten.[1] Nichtsdestoweniger betrachten wir in diesem Kapitel ausschließlich Algorithmen, 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 Garbge 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 und können die Bereinigungsphase ersetzen, indem aus den Markierungsinformationen die zukünftigen Positionen der Objekte generiert werden. Im Folgenden werden wir mit dem *LISP-2-Algorithmus* und *The Compressor* zwei Exemplare dieser Gattung vorstellen. 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. Hier werden wir exemplarisch einen Algorithmus betrachten, der den Heap in zwei Halbräume aufteilt.

4.1 LISP-2-Kompaktierung

Zunächst betrachten wir einen Algorithmus, 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 und können durch zukünftige Allokationen überschrieben werden.

Wie schon in Kapitel 2 setzen wir voraus, dass eine Prozedur zu Verfügung steht, die die Adrese 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.

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

[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]).

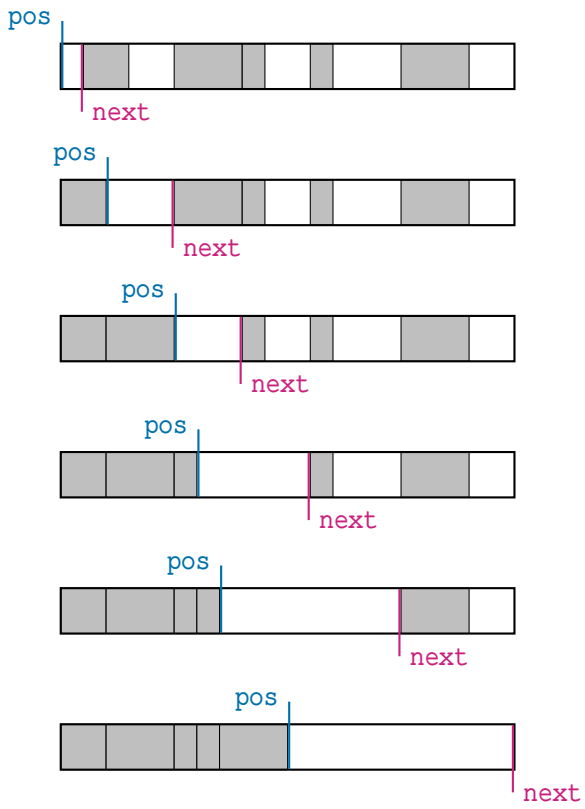


Abbildung 4.2.: Veranschaulichung des LISP-2-Algorithmus.

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)
 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.

Die Laufzeit des gesamten Algorithmus lässt sich wie folgt einschätzen: 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. Wir können somit folgende Abschätzungen festhalten:

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

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

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

Der Speicherplatzbedarf, der durch die Kompaktierung zusätzlich 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 bedarf. 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 seine 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 werden 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 angelegt 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.

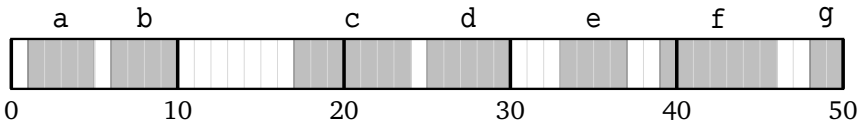


Abbildung 4.3.: Ein Bitvektor speichert zu jedem Wort des Heaps, ob es durch ein erreichbares Objekt belegt wird.

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 Speicheradresse eines Objekts kann nun wie folgt anhand der alten Speicheradresse

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

`adr` ermittelt werden (vgl. Prozedur *newAddress*): Zunächst wird anhand des Blocks, in dem sich die Speicheradresse befindet, der Offset ermittelt. 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.

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

```
 1: calculateOffsets():
 2:   offset(0) ← HEAP_START
 3:   for k ∈ {1, ..., |BLOCKS|}
 4:     offset(k) ← offset(k − 1) + usedWords(i − 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 auch die neue Speicheradresse. 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 pro Wort Speicherplatz, während der Offsetvektor pro Block ein Wort belegt. Für die oben genannte Wortbreite und Blockgröße ergibt sich daher

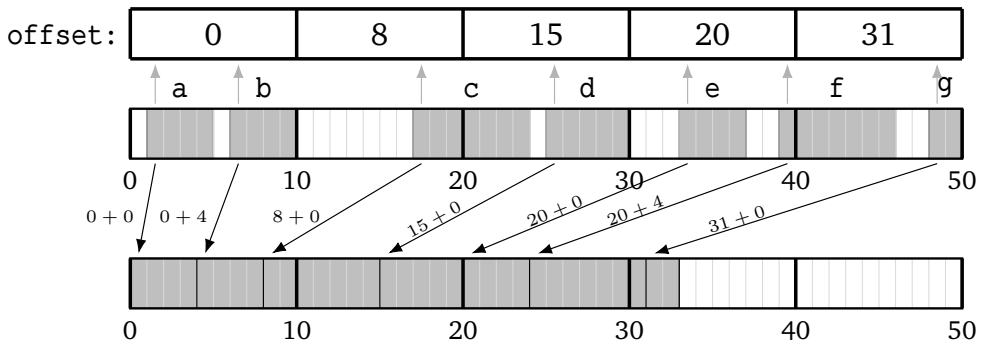


Abbildung 4.4.: Beispiel zum Compressor-Algorithmus. Die neue Speicheradresse eines Objekts berechnet sich aus dem Offset des zugehörigen Blocks und dem Speicherplatz, der durch die davorliegenden Objekte im Block beansprucht wird.

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 4.1), 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%

Tabelle 4.1.: 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 weitergeschoben, 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 Kategorie 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 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]); wir betrachten im Folgenden eine Anlehnung an eine Variante von CHENEY, die ohne Rekursion auskommt (vgl. [Che70]).

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

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

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

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

Zur Vorbereitung wird der Heap zu Beginn 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` zu 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 8 bis 10). 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 eine neue Speicheradresse in `newAdr` hinterlegt wurde (Zeile 19). Falls nein, 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 20 bis 22). 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 des untersuchten Feldes auf die neue Speicheradresse aktualisiert werden (Zeile 24). 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 *collectGarbage* kann anschließend mit der Abarbeitung der Objekte in `toDo` beginnen (Zeile 11 bis 15).

Abbildung 4.5 veranschaulicht die Arbeitsweise des Algorithmus an einem Beispiel: Nachdem die Rollen der Halbräume `source` und `target` vertauscht wurden, wird zunächst das einzige Basisobjekt `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)**.

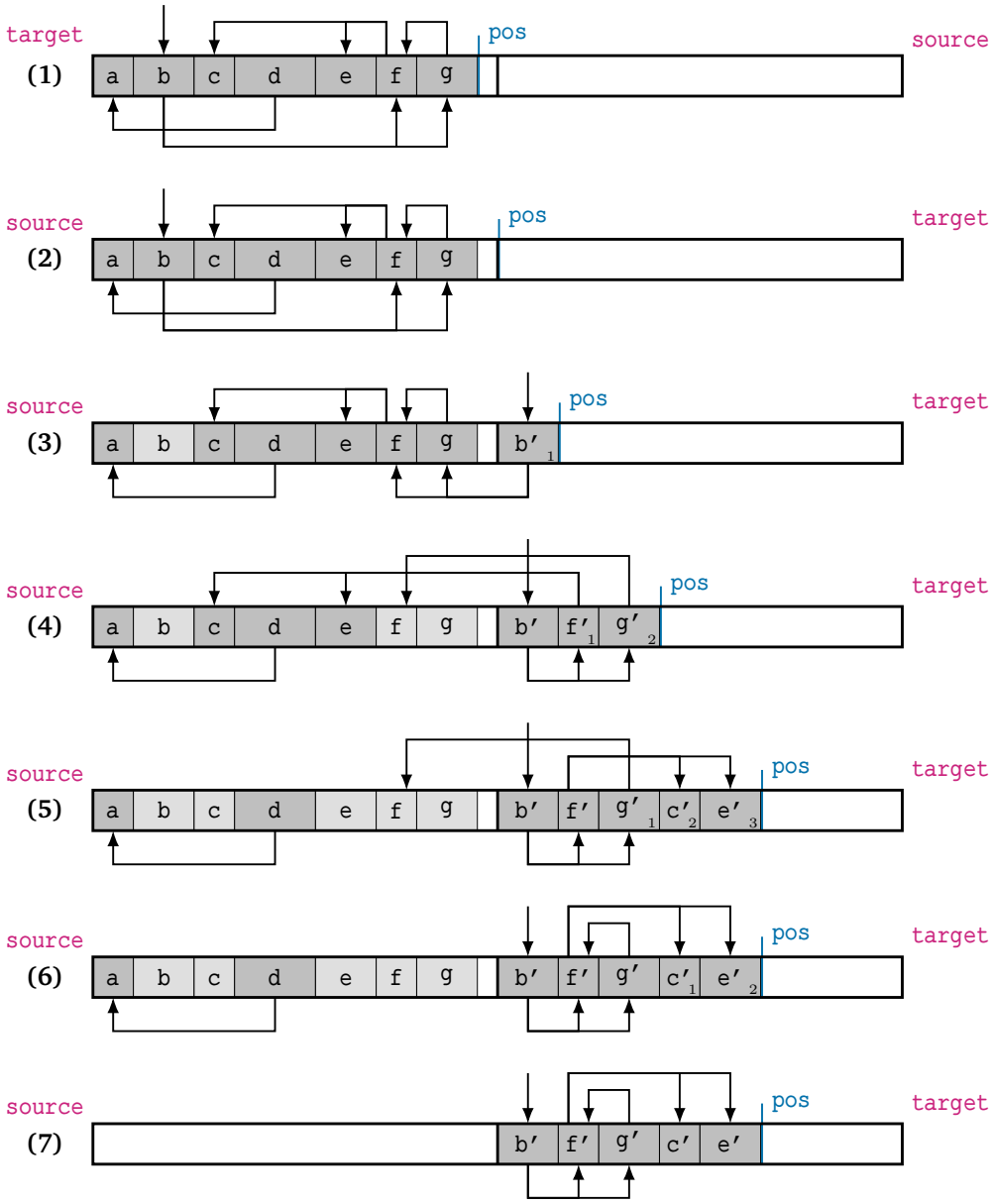


Abbildung 4.5.: Beispielhafte Ausführung von Algorithmus 4.3. Die Nummerierung der Kopien zeigt die aktuelle Position in `toDo` an. Referenzen der ursprünglichen Objekte aus `source` werden hier aus Gründen der Übersichtlichkeit nicht mehr aufgeführt, sobald das Objekt nach `target` kopiert wurde.

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. 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 8 bis 10) 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 23). Foglich kam es zu einer Iteration der **while**-Schleife (Zeile 11) 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 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 wiederum bietet die Möglichkeit, Objekte innerhalb des Heaps neu anzuordnen: Abbildung 4.5 lässt bereits erkennen, dass zwei Objekte, die vom selben Objekt referenziert werden, nebeneinander kopiert werden können. Dies 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, gegebenenfalls erneut zu `toDo` hinzugefügt werden, indem geeignete Lese- und Schreibbarrieren verwendet 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 haben wir gesehen, 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** 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` dann eine solche auf `ã`. Ändert sich die Position von `a` im Speicher, ist lediglich eine Anpassung der Speicheradresse in `ã` nötig (vgl. [Bro84]).

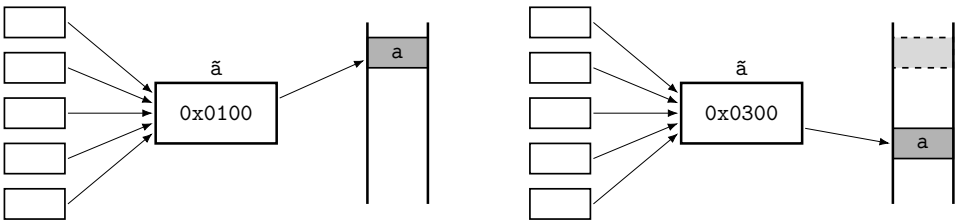


Abbildung 4.6.: Der Handle `ã` enthält die Speicheradresse des Objekts `a`. Erfolgt der Zugriff auf `a` ausschließlich über `ã`, so genügt es, lediglich die Adresse in `ã` anzupassen.

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 durch den 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:         collectGarbage()
 5:         adr ← allocate()
 6:     handle ← allocateHandle()          ▷ Anforderung im separaten Heapbereich
 7:     if handle = null
 8:         collectGarbage()
 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 $\tilde{a}$ im Header von `a` bereithält. Diese ist notwendig, damit nach Verschiebung von `a` der zugehörige Handle 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 `header(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 trifft beispielsweise auf die Programmiersprache C++ zu, in welcher unter anderem Adress- und Dereferenzierungsoperator überladen werden können (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 oben betrachteten Algorithmen verwaiste Objekte nicht explizit freigegeben, sondern 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 einen 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 Bereiche ermöglicht eine feingranularere Auswahl an Garbage-Collection-Strategien. Die verschiedenen Bereiche können unterschiedlich häufig, durch passend abgestimmte Algorithmen und/oder eine Kombination verschiedener Ansätze bereinigt werden. Im Folgenden werden wir daher zunächst auf diverse Kriterien zur Heappartitionierung eingehen, bevor anschließend eine Auswahl dieser hybriden Algorithmen betrachtet wird.

5.1 Algorithmus von Lieberman und Hewitt

Eines der ersten Verfahren, der die Lebensdauer der Objekte als Kriterium für einen Kollektionszyklus nutzt, stammt von LIEBERMAN und HEWITT [LH83]. Der Heap wird dazu ebenfalls 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 generationenweise 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]). Um dies zu errei-

chen, 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].

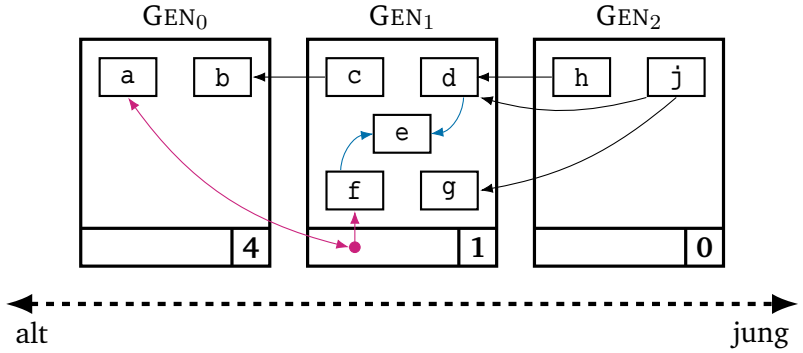


Abbildung 5.1.: Beispielhafte Objektkonstellation zwischen drei Generationen. Der Zähler unten rechts gibt die akuelle Version einer Generation an. Neben Referenzen von jüngeren auf ältere Objekten existieren auch interne Referenzen zwischen gleich alten Objekten (blau) und indirekte Referenzen von älteren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

Der Kollektor geht bei der Bereinigung der Generation k (`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.

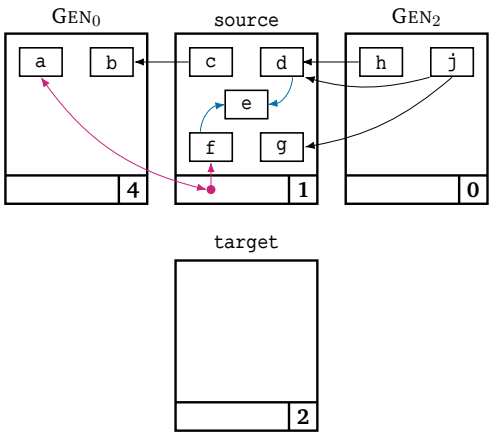
Das Kopieren eines Objekts wird stets durch die Prozedur *copy* bewerkstelligt. Genau wie in der Prozedur *update* im Halbraumverfahren (Algorithmus 4.3) wird ein Objekt

[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.

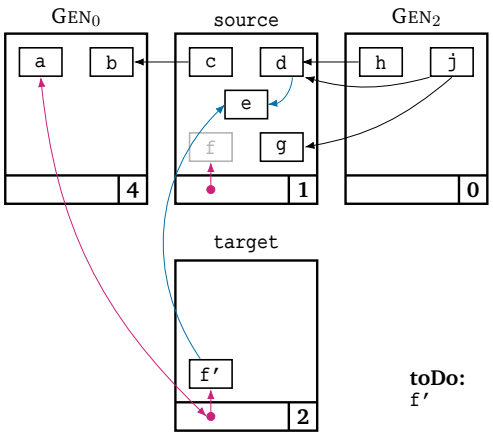
Algorithmus 5.1 Generationelle Garbage Collection nach LIEBERMAN und HEWITT (vgl. [LH83, S. 421ff]).

```
 1: collectGarbage(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 ≠ null ∧ **field ∈ source)
19:         copy(**field, pos)
20:         *field ← newAdr(**field)
21:   while toDo ≠ null
22:     ref ← remove(toDo)
23:     for each field ∈ POINTERFIELDS(*ref)     ▷ Interne Referenzen anpassen
24:       if (*field ≠ null ∧ **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))
```

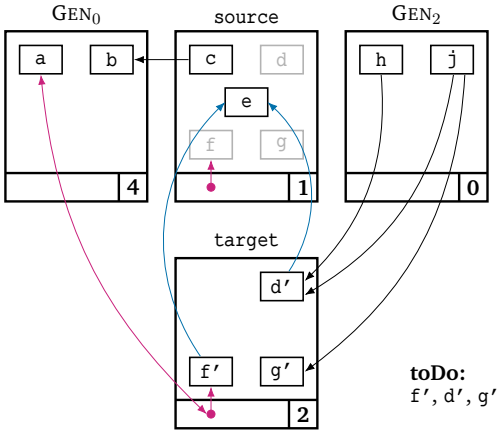
nur dann kopiert, wenn es nicht zuvor bereits kopiert wurde. Andernfalls genügt es, lediglich die im Feld gespeicherte Adresse anzupassen. Da kopierte Objekte ebenfalls Referenzen auf Objekte derselben Generation enthalten können, werden sie zu `toDo` hinzugefügt. Bei der Abarbeitung von `toDo` (Zeile 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.



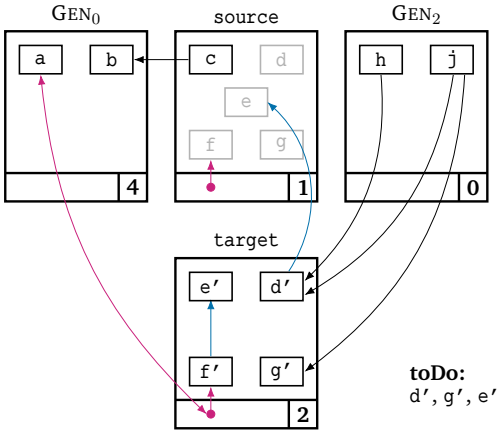
(a) Anlegen einer neuen Version.



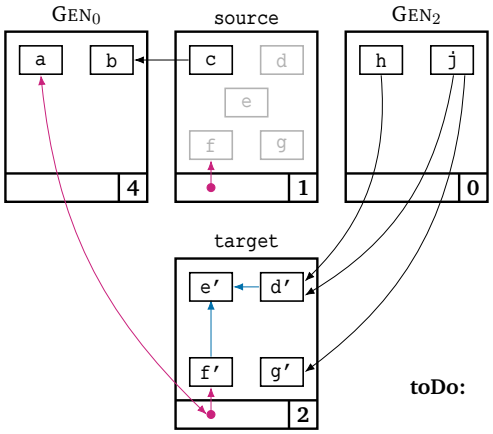
(b) Kopieren der von älteren Generationen referenzierten Objekte und Erzeugung neuer Handles.



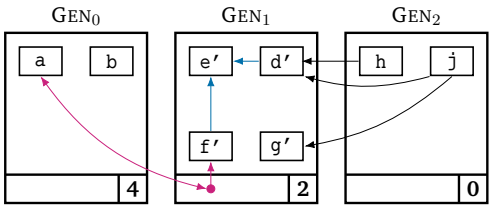
(c) Kopieren der von jüngeren Generationen referenzierten Objekte.



(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 nun 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, sollte diese eher selten stattfinden (vgl. [LH83, S. 423]).

Abgesehen von Generation und Versionszahl gibt es noch weitere Stellschrauben 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 aktuell erhaltenen 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.

6 Fazit und Vergleich

Teil II

Entwurf und Realisierung eines
Garbage-Collection-Simulators

7 Modellierung

Designentscheidungen

Anhang

Test

A

blablubb

Literatur

[ASS96] Harold Abelson, Gerald Jay Sussmann und Julie Sussmann. *Structure and Interpretation of Computer Programs*. 2. Aufl. MIT Press, 1996 (zitiert auf Seite 11).

[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 (zitiert auf Seite 52).

[Che70] C. J. Cheney. „A Nonrecursive List Compacting Algorithm“. In: *Communications of the ACM* 13.11 (1970), S. 677–678 (zitiert auf Seite 48).

[Col60] George E. Collins. „A Method for Overlapping and Erasure of Lists“. In: *Commununications of the ACM* 3.12 (1960), S. 655–657 (zitiert auf Seite 25).

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein. *Introduction to Algorithms*. 3. Aufl. Cambridge, MA: MIT Press, 2009 (zitiert auf den Seiten 22, 37).

[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 (zitiert auf den Seiten 35, 36, 55).

[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 (zitiert auf den Seiten 4, 14–16, 18).

[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 (zitiert auf Seite 48).

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

[Hug82] Robert John Muir Hughes. „A semi-incremental garbage collection algorithm“. In: *Software: Practice and Experience* 12.11 (1982), S. 1081–1082 (zitiert auf Seite 18).

[JHM11] Richard Jones, Antony Hosking und Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Boca Raton: CRC Press, 2011 (zitiert auf den Seiten 15, 19, 21, 22, 26, 41–43, 45, 51).

[JL96] Richard Jones und Rafael Lins. *Gargabe Collection: Algorithms for Automatic Dynamic Memory Management*. Chichester: John Wiley & Sons, 1996 (zitiert auf den Seiten 2, 12, 13, 23).

[JR08] Richard E. Jones und Chris Ryder. „A Study of Java Object Demographics“. In: ISMM ’08 (2008), S. 121–130 (zitiert auf Seite 59).

[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 (zitiert auf Seite 52).

[KNL07] Martin Kero, Johan Nordlander und Per Lindgren. „A Correct and Useful Incremental Copying Garbage Collector“. In: ISMM ’07 (2007), S. 129–140 (zitiert auf Seite 51).

[KP06] Haim Kermany und Erez Petrank. „The Compressor: Concurrent, Incremental, and Parallel Compaction“. In: *SIGPLAN Notices* 41.6 (2006), S. 354–363 (zitiert auf den Seiten 45, 46).

[LH06] Chin-Yang Lin und Ting-Wei Hou. „A Lightweight Cyclic Reference Counting Algorithm“. In: (2006). Hrsg. von Yeh-Ching Chung und José E. Moreira, S. 346–359 (zitiert auf den Seiten 27, 29, 35).

[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 (zitiert auf den Seiten 55, 57, 59).

[Lin92] Rafael D. Lins. „Cyclic reference counting with lazy mark-scan“. In: *Information Processing Letters* 44.4 (1992), S. 215–220 (zitiert auf Seite 35).

[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 (zitiert auf den Seiten 38, 39).

[McC60] John McCarthy. „Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I“. In: *Commununications of the ACM* 3.4 (1960), S. 184–195 (zitiert auf Seite 11).

[McC79] John McCarthy. *Histroy of LISP*. 1979. URL: `http://www-formal.stanford.edu/jmc/history/lisp/lisp.html` (besucht am 7. Jan. 2019) (zitiert auf Seite 42).

[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 (zitiert auf Seite 44).

[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 (zitiert auf den Seiten 29–32).

[Pir98] Pekka P. Pirinen. „Barrier Techniques for Incremental Tracing“. In: *SIGPLAN Notices* 34.3 (1998), S. 20–25 (zitiert auf Seite 18).

[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) (zitiert auf Seite 45).

[Rob80] John Michael Robson. „Storage allocation is NP-hard“. In: *Information Processing Letters* 11.3 (1980), S. 119–125 (zitiert auf Seite 41).

[Sip13] Michael Sipser. *Introduction to the Theory of Computation*. 3. Aufl. Cengage Learning, 2013 (zitiert auf Seite 5).

[Str13] Bjarne Stroustrup. *The C++ Programming Language*. 4. Aufl. Addison-Wesley, 2013 (zitiert auf Seite 53).

[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 (zitiert auf den Seiten 42, 43).

[TB14] Andrew S. Tanenbaum und Herbert Bos. *Modern Operating Systems*. 4. Aufl. Pearson, 2014 (zitiert auf Seite 3).

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

Backtracking entfernen

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

Tabellenverzeichnis

Algorithmenverzeichnis

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.

(Ort, Datum) (Unterschrift)