Todo list

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, 22. Dezember 2018

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, 22. Dezember 2018

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.

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.

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

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

[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].

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 Methode *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 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:

definieren

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 $\texttt{b} \in \mathcal{R}$. In diesem Fall schreiben wir auch $\texttt{a} \to \texttt{b}$.

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. 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:

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

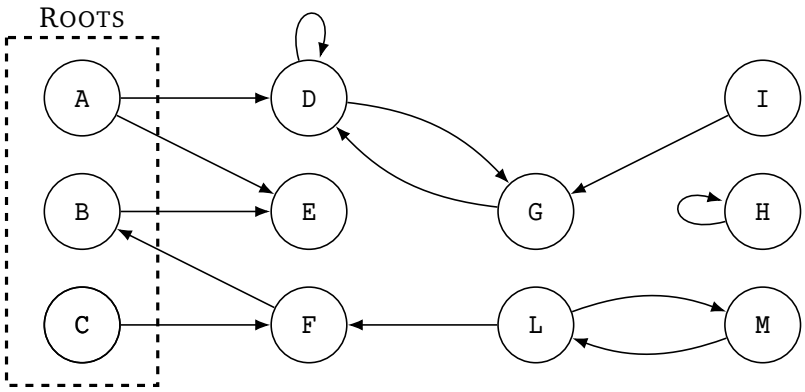

Abbildung 1.1.: Beispiel für einen Objektgraphen. Die Objekte A, B und C sind Basisobjekte. Die Objekte H, I, L und M sind in dieser Kostellation nicht erreichbar.

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

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 John 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*.

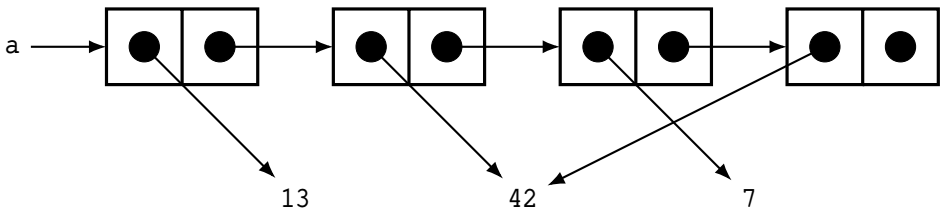

Abbildung 2.1.: Visualisierung der LISP-Liste `a = (13,42,7,42)`.

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 der 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 adr ∈ POINTERS(obj)          ▷ Hole nächste Referenz auf Objekt
15:             if (adr ≠ null ∧ isNotMarked(*adr))
16:                 setMarked(*adr)
17:                 add(toDo, *adr)
```

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 Methode *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 Methode *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 Methode *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 Methode *sweep*. Die Variable `pos` wird mit der Speicheradresse initialisiert, an der sich das erste Objekt im Heap befindet. Wir gehen davon aus, dass eine Methode *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 $\texttt{adr} = \texttt{\&b}$ 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 adr ∈ POINTERS(obj)
13:       if (adr ≠ null ∧ isWhite(*adr))
14:         setColor(*adr, GRAY)                  ▷ Referenzierte Objekte grau markieren
15:         add(graySet, *adr)

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.

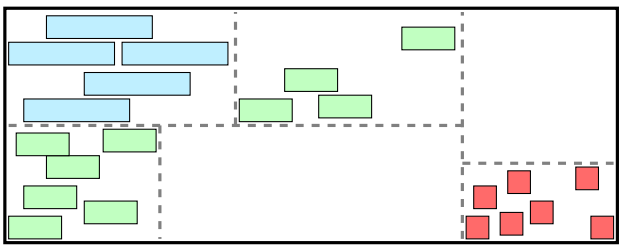

Abbildung 2.2.: Visualisierung eines in Regionen aufgeteilten Heaps. Jede Region kann Objekten einer festgelegten Größenordnung zugewiesen werden. Für eine Größenordnung können allerdings mehrere Regionen verwendet werden.

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 Methode *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. 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.

Ref zu Defrag

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 statt dessen auch zunächst alle Basisobjekte zu `toDo` hinzugefügt und die Methode *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 George 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 $\mathtt{rc}(\mathtt{a}) \geq 1$.

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 $\mathtt{a}$ mit $\mathtt{rc}(\mathtt{a}) = 0$ 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 `obj`$[i] \leftarrow$ `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).

Beispiel

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 *decRefCount*$(\mathtt{a}_1)$, *decRefCount*$(\mathtt{a}_2), \ldots$ für Objekte $\mathtt{a}_i, i \in \mathbb{N}$. Da jedes Objekt nur endlich viele Referenzen auf andere Objekte besitzt, ist $\textsc{Pointers}(\mathtt{a}_i)$ endlich für alle $i \in \mathbb{N}$. Weiter ist die Anzahl aller Objekte endlich; es muss also ein Objekt $\mathtt{a}_j$ existieren, für welches *decRefCount*$(\mathtt{a}_j)$ unendlich oft aufgerufen wird. Allerdings wird *decRefCount*$(\mathtt{a}_j)$ nur aufgerufen, wenn ein Objekt $\mathtt{b}$ mit $\mathtt{b} \rightarrow \mathtt{a}_j$ existiert, für welches zuvor $\mathtt{rc}(\mathtt{b})$ auf 0 gesetzt wurde. Da während der rekursiven Aufrufe von *decRefCount* keine Referenzzähler erhöht werden, müssen also unendlich viele verschiedene Objekte $\mathtt{b}$ 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 $\mathtt{a}$ durch den Algorithmus erfolgt ausschließlich in Zeile 13. Diese wird nur ausgelöst, wenn $\mathtt{rc}(\mathtt{a}) = 0$ gilt. Aus Lemma 3.1 folgt, dass $\mathtt{a}$ nicht erreichbar ist. □

Die Garbage Collection mithilfe naive Referenzzählung bietet im Vergleich zum Mark-Sweep-Algorithmus einige Vorteile (vgl. [JHM11, S. 58–60]): 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 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.

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
```

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 3.1). Die einzelnen Objekte werden dann nicht als löschbar erkannt, was zu einem Speicherleck führt.

Ausblick auf nächste Abschnitte

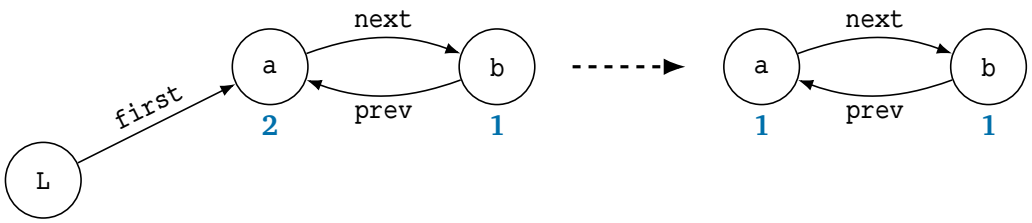

Abbildung 3.1.: Referenzzählung in zyklischen Datenstrukturen. Wird Objekt `L` entfernt, sind `a` und `b` nicht mehr erreichbar. Jedoch fallen ihre Referenzzähler nicht auf 0, sodass sie als Speicherlecks im Speicher verbleiben.

4 Kompaktierung

5

Generationelle Garbage Collection

6 Fazit und Vergleich

Teil II

Entwurf und Realisierung eines
Garbage-Collection-Simulators

7 Modellierung

Designentscheidungen

Anhang

Test

A

blablubb

Literatur

[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 Seite 22).

[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).

[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, 26, 27).

[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).

[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).

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

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

[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).

Diese Masterarbeit wurde mit LaTeX 2_ε 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)