Garbage Collection

Maik Theisen
Betreuer: Guido Tack

Proseminar Programmiersysteme WS 03/04
Prof. Dr. Gert Smolka
Programming Systems Lab, Universität des Saarlandes

Garbage Collection (GC) hat sich mittlerweile als wichtiger Teil des Speicherverwaltungssystems von vielen Programmiersprachen etabliert. Dabei gibt es zwei große Problemstellungen: wie wird GC durchgeführt und wann wird GC durchgeführt. Nach einer allgemeinen Einführung in Kapitel 1 werden in Kapitel 2 und 3 die wichtigen Techniken zur Copying GC bzw. Generational GC behandelt. In Kapitel 4 wird die Problematik des Schedulings, also der Frage nach dem wann, angesprochen und zuletzt gibt es in Kapitel 5 einen kurzen Ausblick auf weitere Techniken, die sich noch in Forschung und Entwicklung befinden.

1. Einführung

Garbage Collection wurde zuerst in der Programmiersprache LISP eingesetzt und lange Zeit war GC nur auf funktionale Sprachen beschränkt. Heute allerdings setzen viele Programmiersprachen –funktionale und imperative- diese Technik zur Speicherbereinigung ein, z.B. Java, Alice, ML und Smalltalk Das Ziel von GC ist es, *lebende Knoten* auf dem Heap zu finden und von *toten Knoten* zu unterscheiden. Diese toten Knoten werden nach Abschluss der Collection als wieder verfügbarer Speicher freigegeben. Somit stellt sich die Frage, woran man einen toten Knoten von einem Lebenden unterscheidet. Grundsätzlich kann man sagen, dass jeder Knoten, der von den *root*-Knoten aus erreichbar ist, lebt (s. Abb. 1.1). Root-Knoten stellen die Ausgangsknoten für die verschiedenen Techniken von GC dar und sind meist diejenigen Knoten auf dem Heap, die direkt durch Referenzen aus dem Stack angesprochen werden. Alle Root-Knoten bilden zusammen das *root set*. Dementsprechend sind tote Knoten diejenigen Knoten, die nicht durch Referenzen oder Pointer von lebenden Knoten erreicht werden können.

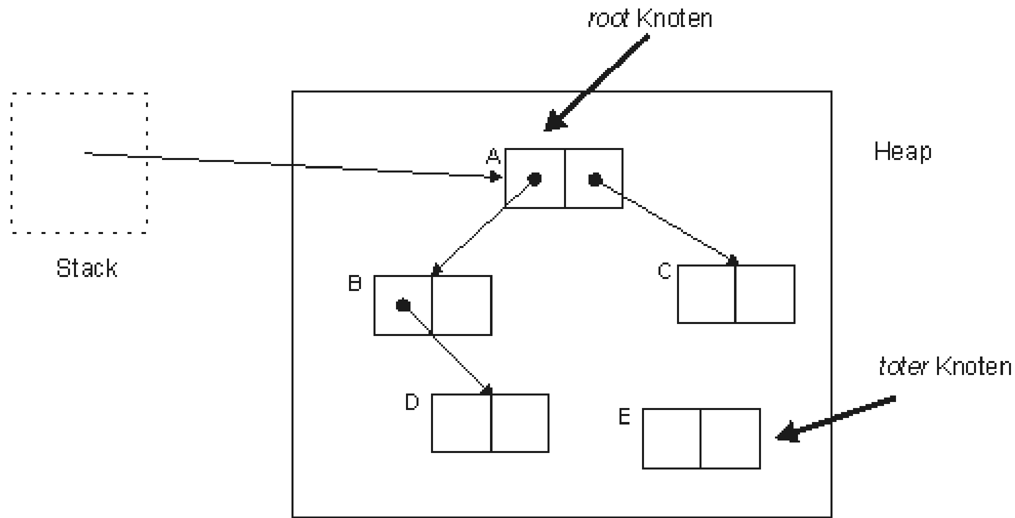


Abb. 1.1 *root* und *tote* Knoten

Weiterhin stellt sich die Frage, warum GC überhaupt eingesetzt wird. In C/C++ verzichtet man beispielsweise auf diese Techniken und überlässt dem Programmierer die Verwaltung des Speichers. Dies führt jedoch häufig zu Fehlern, wenn man z.B. einen Pointer löscht, der noch gebraucht wird. Als Beispiel für diese Fehler, ihre Entstehung und ihre Auswirkungen soll eine kleine einfach verkette Liste dienen (s. Abb. 1.2). Wenn nun ein Pointer auf eine andere Zelle gelöscht wird (s. Abb. 1.3), entsteht ein Speicherleck. Falls mehrere dieser Löcher im Programmablauf entstehen, kann es passieren, dass irgendwann kein freier Speicher mehr vorhanden ist. Sollte eine ganze Zelle gelöscht werden (s. Abb. 1.4) entsteht zusätzlich zu dem Speicherleck (Zelle 3 ist nicht mehr erreichbar) noch ein *dangling pointer*. Dieser Pointer kann ein Problem darstellen; wenn die Speicherzelle, auf die dieser Pointer zeigt, überschrieben wird, kann sich das Verhalten des gesamten Programms ändern. Die Suche nach diesen Fehlern ist sehr zeitraubend. Durch den Einsatz von GC wird diese oft mühselige Arbeit dem Programmierer abgenommen und er kann sich auf wesentlichere Dinge seiner Arbeit konzentrieren.

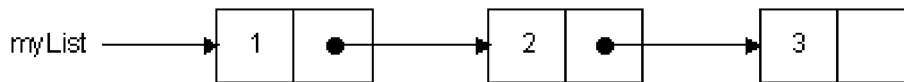


Abb. 1.2 Eine einfache Liste

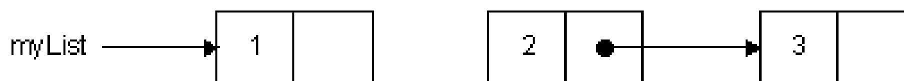


Abb. 1.3 Pointerlöschung führt zu einem Speicherleck

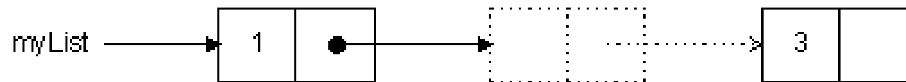


Abb. 1.4 Löschen einer Zelle erzeugt ein Speicherleck und einen „dangling Pointer“

GC steht in dem Ruf, die Ausführung von Programmen stark zu verlangsamen. In der Vergangenheit war dies auch zutreffend, aber heutige moderne Algorithmen haben den Overhead durch GC dermaßen reduziert, dass GC sogar in Sprachen zur Systemprogrammierung, wie z.B. Modula, eingesetzt wird. Man kann sagen, dass moderne GC einen Overhead von durchschnittlich etwa zehn Prozent verursacht (vgl. [1]).

2. **Copying Garbage Collection**

Copying GC unterteilt den Heap in zwei Bereiche, die sogenannten *semi-spaces*. Auf einem Bereich, dem *Fromspace,* wird gearbeitet. Hier befinden sich alle lebendigen Pointer und Knoten und auch Speicherallokationen finden nur im *Fromspace* statt. In dem anderen, inaktiven Bereich, dem *Tospace*, werden keine Operationen durchgeführt. Sobald die GC ausgelöst wird, werden die lebenden Knoten und Pointer aus dem *Fromspace* in den *Tospace* kopiert. Anschließend vertauscht man die Rolle der beiden Spaces. Die Größe der *semi-spaces* kann dynamisch variiert werden, je nachdem, wie viele Daten jeweils kopiert werden. In Abb. 2.1 wird verdeutlicht, wie die Speicherbelegungen vor und nach dem Kopieren aussehen.

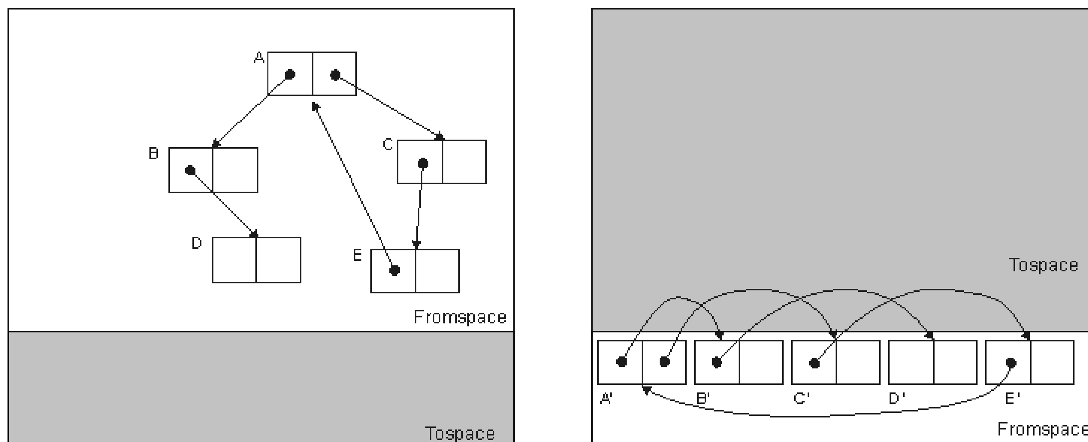


Abb. 2.1 Speicherbelegung vor und nach dem Kopieren

Durch das Bild wird bereits ein Vorteil von Copying GC deutlich: die kopierten Speicherzellen liegen kompakt beieinander; der freie Speicher wird nicht fragmentiert.

Der erste Copying Garbage Collector, der mit s*emi-spaces* arbeitete, wurde bereits 1969 von Robert Fenichel und Jerome Yochelson entwickelt (vgl. [1]). In deren System wurde der Speicherinhalt durch ein rekursives Verfahren kopiert. Dadurch wurde allerdings zusätzlicher Stack-Speicherplatz benötigt, der im schlimmsten Fall so groß werden konnte, wie der Inhalt des Heaps. Da die Größe des Stacks im Regelfall aber beschränkt ist, musste noch zusätzlich auf Stack-Overflows geachtet werden.

Im Jahr 1970 entwickelte C. J. Cheney einen Algorithmus (vgl. [3]), der den Speicherinhalt iterativ kopieren konnte (s. Abb. 2.2). Dadurch wurde zusätzlicher Stack-Speicherplatz wie in obigem Verfahren nicht benötigt. Der Algorithmus durchmustert den Heap in einer Breitensuche und kopiert alle erreichten Zellen des *Fromspace* in den *Tospace*. Die Implementierung von Cheney ist zur Zeit das effizienteste Verfahren zur Durchführung von Copying GC.

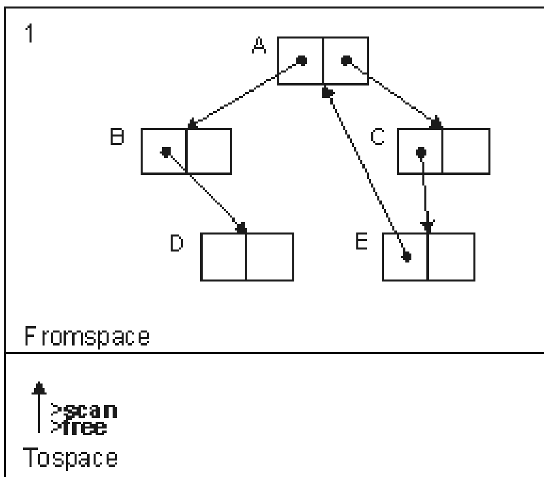
```
flip() =
     Fromspace, Tospace = Tospace, Fromspace
     top_of_space = Tospace + space_size
     scan = free = Tospace
     for R in Roots
          R = copy(R)

     while scan < free
          for P in Children(scan)
               P = copy(*P)
          scan = scan + size (scan)


copy(P) =
     if forwarded(P)
          return forwarding_adress(P)
     else
          addr = free
          move(P,free)
          free = free + size(P)
          forwarding_adress(P) = addr
          return addr
```

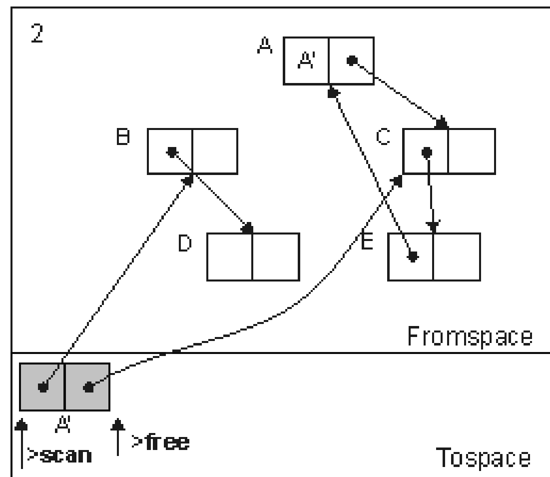
Abb. 2.2 Cheneys Algorithmus in Pseudocode

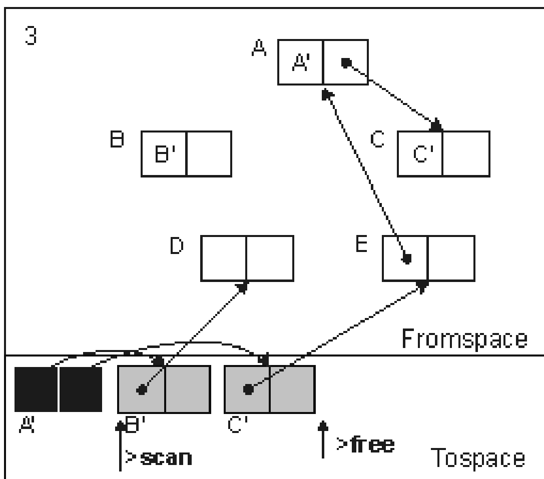
Cheney benutzt zwei Pointer, **>free** und **>scan**, die in den *Tospace* zeigen. **>free** zeigt hinter das zuletzt in den *Tospace* kopierte Objekt und damit auf freien Speicher. **>scan** zeigt auf das nächste vom Collector zu bearbeitende Objekt im *Tospace* (s. Abb. 2.3).

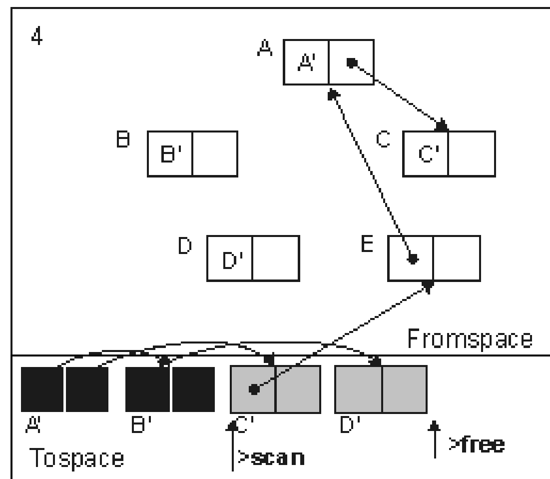
Abb. 2.3 **>scan, >free** und *forwarding pointer*

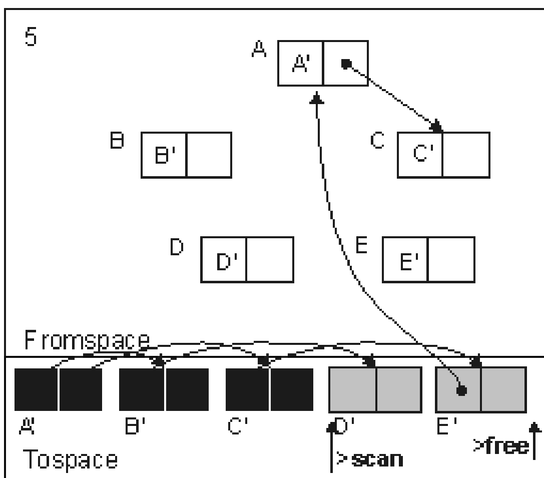
Die Vorgehensweise des in Abb. 2.2 beschriebenen Algorithmus wird in Abb. 2.4 in kompakter Form illustriert. Cheney verwendet zur Illustration eine Dreifärbung der Knoten im Heap in weiß, grau und schwarz. Weiße Knoten werden am Ende der Collection als „Müll“ betrachtet. Graue Knoten liegen zwischen **>scan** und **>free** und müssen vom Collector noch bearbeitet werden. Schwarze Knoten liegen links von **>scan** und wurden somit schon völlig abgearbeitet.
Der Collector beginnt bei den Knoten des *root sets*, kopiert diese mitsamt ihren Pointern in den *Tospace* und durchmustert anschließend den Heap in einer Breitensuche. Jeder Knoten, der dabei erreicht wird, wird zusammen mit seinen Pointern in den *Tospace* kopiert und **>free** wird um die Größe des Knotens verschoben. Anschließend wird ein *forwarding pointer* von dem alten Objekt auf seine Kopie erstellt, so dass eventuelle Pointer, die auf das alte Objekt zeigen, umgeleitet werden können. Ohne diese Pointer könnte der Collector nicht überprüfen, ob ein Objekt bereits kopiert ist. Wenn dann im weiteren Verlauf der GC ein Pointer auf dieses Objekt zeigt, müsste es wieder kopiert werden und der Collector befände sich in einer Endlosschleife.
Wenn sich im *Tospace* keine grauen Knoten mehr befinden, also wenn **>free** und **>scan** wieder gleich sind, ist die GC abgeschlossen und die Rollen von *Fromspace* und *Tospace* werden anschließend vertauscht.











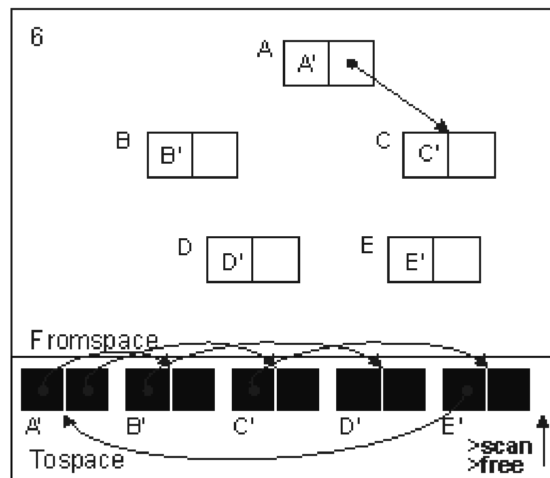


Abb. 2.4 Darstellung der Funktion von Cheneys Algorithmus

Cheneys Algorithmus ist der effizienteste Copying GC Algorithmus, der bisher entwickelt wurde. Er läuft in der Komplexitätsklasse *O(n)*, wobei n die Anzahl der *lebenden* Objekte im Heap ist (vgl. [1]). Folglich ist es unwichtig, wie viele tote Objekte sich im Heap befinden, da dieser Algorithmus nur auf den lebenden Knoten arbeitet.

Ein weiterer Faktor, der die Effizienz von GC-Algorithmen beeinflusst, ist die Größe des Heaps. Je größer dieser ist, desto seltener muss der Collector arbeiten und desto weniger Ausführungszeit des Programms geht verloren. Ebenfalls zu berücksichtigen ist die Größe der Objekte im Heap. Da es recht teuer ist, große Objekte immer wieder zu kopieren, gibt es Techniken, um dies zu verhindern, die sogenannte *multiple-area collection.* Dabei werden diese großen Objekte speziell behandelt, in dem sie in einen anderen Bereich des Speichers geschrieben werden und dieser Bereich durch eine andere GC-Technik bereinigt wird.

Die hauptsächlichen Einsatzgebiete von Copying GC sind Systeme mit kleinen, kurzlebigen Objekten und Systeme bei denen Pausen oder Verzögerungen unkritisch sind.
Bei erstgenannten Systemen kommt die Stärke von Copying GC voll zur Geltung. Da die Objekte nur klein sind, verursachen sie beim Kopieren nur geringe Kosten und aufgrund ihrer kurzen Lebensdauer werden sich zum Zeitpunkt der Collection nicht viele lebende Objekte auf dem Heap befinden.
Cheneys Algorithmus stellt einen sogenannten *start-stop-collector* dar. Das bedeutet, dass zum Zeitpunkt der Collection keine Aktivitäten von Seiten des Benutzers durchgeführt werden dürfen. Daher wird Copying GC auch hauptsächlich in zeitlich unkritischen Systemen eingesetzt.
Ein weiteres Einsatzgebiet sind Systeme, deren Speicherverwaltung von Allokationen dominiert wird. Aufgrund des Pointers **>free** kann sehr leicht neuer Speicher zugewiesen werden; **>free** zeigt schließlich auf den nächsten verfügbaren freien Speicherplatz. Wenn der Pointer dann noch um die Größe des neu zugewiesenen Objektes verschoben wird, kann er auch weiterhin seine Rolle erfüllen.

Ein Nachteil der Copying GC ist, dass durch die Breitensuche die *Lokalität* verloren geht. Objekte, die logisch voneinander abhängen, werden kurz hintereinander erzeugt und liegen auch im Speicher dicht zusammen. Am Ende der GC können die Objekte allerdings so weit auseinanderliegen, dass *page faults* in der virtuellen Speicherverwaltung auftreten können und dadurch viel Laufzeit verloren geht.

Ebenfalls ein großes Problem der Copying GC sind Objekte mit langer Lebensdauer. Diese werden bei jedem Aufruf des Copying Collectors wieder kopiert und kosten somit Laufzeit. Um dieses Problem zu umgehen wurden spezielle Techniken entwickelt, die als Generational Garbage Collection bekannt sind.

3. Generational Garbage Collection

Generational GC wurde entwickelt, um Objekte mit langer Lebensdauer spezieller zu behandeln. Meistens werden in diesem Bereich Copying GC-Algorithmen eingesetzt, aber es ist nicht ungewöhnlich, dass auch andere Verfahren zum Einsatz kommen. Die Techniken der Generational GC basieren auf der *weak generational hypothesis*, welche besagt, dass die meisten Objekte nur eine sehr kurze Lebenszeit im Speicher haben (vgl. [1]). Untersuchungen ergaben, dass 80 bis 98 Prozent der Objekte sterben, bevor nur ein weiteres Megabyte an Heap-Speicher belegt wurde (vgl. [1]).

Die Idee der Generational GC ist es, den Heap in zwei oder mehr *Generationen* aufzuteilen, in denen die Objekte je nach unterschiedlichem Alter gespeichert werden (s. Abb. 3.1). Wenn die jüngste Generation voll ist, wird eine sogenannte *minor collection* ausgelöst, die nur über die jüngste Generation läuft und lebende Knoten unter gewissen Voraussetzungen in die nächstältere Generation kopiert. Sollte eine ältere Generationen voll sein, startet eine *major collection*, die über alle Generationen läuft und dadurch auch die älteren Generationen bereinigt.

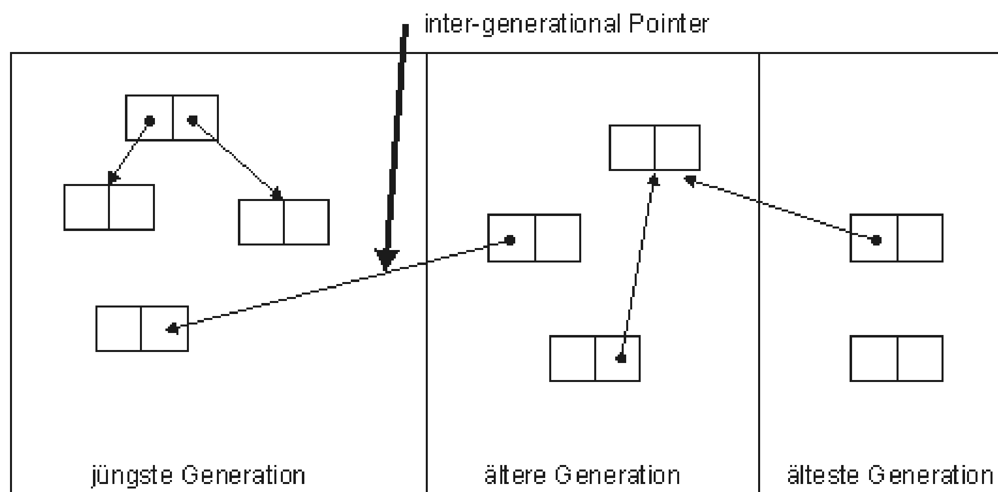


Abb. 3.1 *Generationen* und *inter-generational pointer*

Ein wichtiger Faktor bei Generational GC sind die *inter-generational pointers*. Diese Pointer zeigen von einer älteren Generation in eine jüngere und gehören dadurch zum *root set* der jüngeren Generation. Deshalb müssen diese Pointer speziell überwacht werden. Pointer von junge Generationen in alte Generation hingegen sind nicht gesondert zu beachten, da bei einer Collection, welche die alte Generation bereinigt, auch sämtliche jüngeren Generationen mitbereinigt werden.

Inter-generational pointer können nur auftreten, indem schreibend auf eine Zelle in einer älteren Generation zugegriffen wird. Die schreibenden Zugriffe kann man durch eine *write barrier* überwachen. Diese kontrolliert das Speichern von Pointern und passt gegebenenfalls das root set für die jüngste Generation an. Dadurch wird der Schreibzugriff auf Speicherzellen natürlich teuer, aber in Sprachen, in denen nicht sehr oft auf schreibend auf Speicherzellen zugegriffen wird, wie z. B. in funktionalen Sprachen, ist dies nicht weiter störend.
Für Sprachen, die allerdings sehr oft schreibend auf den Speicher zugreifen, ist diese Technik eher ungeeignet. Dort würde sich der Einsatz von *entry tables* empfehlen (s. Abb. 3.2). Dabei besitzt jede Generation einen speziellen Bereich, der als Umleitung für inter-generational Pointer dient. Diese Pointer zeigen auf eine vorgesehene Zelle und von dort wird dann ein weiterer Pointer auf das eigentliche Ziel gesetzt. Dadurch wird der Schreibzugriff nur minimal teurer, allerdings kann der Lesezugriff durch entsprechend viele Umleitungen teuer werden.

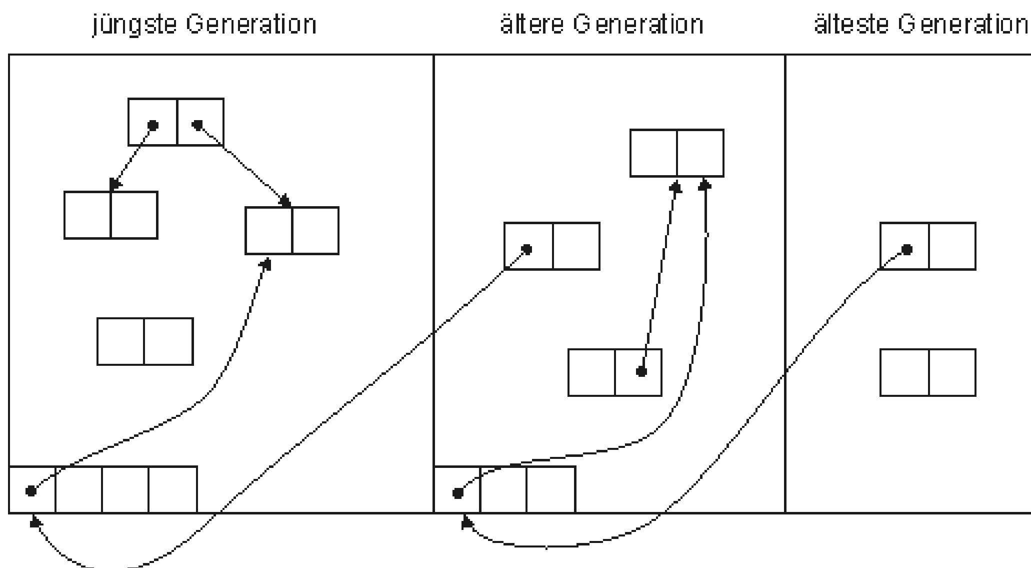


Abb. 3.2 *entry tables*

Es gibt bei Generational GC viele Möglichkeiten und Techniken, wie die Generationen und das Alter der Objekte behandelt werden.
Man kann z. B. eine *Beförderungsschwelle* einführen, so dass ein Objekt, welches in einer Generation überlebt hat, nicht direkt in die nächste Generation befördert wird. Anstatt dessen wird in dem Objekt markiert, wie viele Collections es überlebt hat und erst nach einer vorgegebenen Anzahl davon wird es befördert. Dadurch wird vermieden, dass kurzlebige Objekte zu früh befördert werden und somit ältere Generationen aufblähen.
Wenn man dennoch alle lebenden Objekte direkt kopiert, ergibt sich daraus der Vorteil, dass nur die älteste Generation zwei s*emi-spaces* benötigt. Für die jüngeren Generationen

ist dann einfach die nächstältere Generation der *Tospace*. Selbstverständlich werden in diesem speziellen Fall die Rollen von *Fromspace* und *Tospace* nach Abschluss der Collection nicht vertauscht. Außer es handelt sich um eine *major collection*, welche alle Generationen bereinigt. Dabei wird in der ältesten Generation, welche zwei *semi-spaces* beinhaltet, wieder *Fromspace* mit *Tospace* vertauscht. Die Größe der verschiedenen Generationen ist nach dem Alter zunehmend, also die jüngste Generation ist die kleinste. Dies hat den Vorteil, dass sehr oft schnelle GC ausgelöst wird; wenn die Objekte, wie angenommen, keine lange Lebensdauer haben, werden nicht viele Daten kopiert und somit die alten Generationen nicht aufgebläht.. Aufgrund der Größe der älteren Generationen wird eine *major collection* nur selten nötig und eventuell zu früh kopierte junge Objekte haben Zeit, wieder inaktiv zu werden.

Wenn Generational GC mit Hilfe von Copying Algorithmen implementiert wird, entsteht eine ähnliche Effizienz. Generational GC ist effektiv schneller, da -meist- nur ein kleinerer Teil des Heaps betrachtet wird, als bei reiner Copying GC. Allerdings wird der Generational Collector auch öfter aufgerufen, da die jüngste Generation auch wieder schnell gefüllt ist. Aber aufgrund der benötigten *write barrier* zum Abfangen der *inter-generational pointer* kann die Laufzeit der Programme wiederum leiden.

Aufgrund der Tatsache, dass bei Generational GC meistens auf die Techniken der Copying GC zurückgegriffen wird, sind die Anwendungsgebiete der beiden Techniken die gleichen.
Auch wenn die Laufzeit von Generational GC besser ist, als die von anderen Algorithmen wie z.B. Mark-Compact, ist es nicht immer sinnvoll, Generational GC auch einzusetzen. Aufgrund des hohen Speicherbedarfs von Generational GC wird diese Technik z.B. in PDAs oder Palms nicht eingesetzt. Dort sind die Kosten für zusätzlichen Speicher groß genug, dass man auf etwas Laufzeit verzichtet und andere, nicht so speicherintensive Techniken einsetzt.

Wie bei Copying GC besteht auch bei Generational GC das Problem der großen Objekte.
Deshalb gibt es Techniken, die auf verschiedenen Heuristiken basieren und solche Objekte bereits bei ihrer Entstehung in den Bereich der ältesten Generation schreiben und sie somit aus der GC fast ausnehmen; sie werden nur noch bei den nicht so häufig auftretenden *major collections* beachtet.

4. **Scheduling**

Eine weitere wichtige Fragestellung im Bereich der Garbage Collection, ist die Frage nach dem *wann*. Wann soll Garbage Collection durchgeführt werden? Eine einfache Antwort darauf wäre, wenn der Speicher voll ist. Diese Antwort ist korrekt; allerdings ist dies nicht immer der optimale Zeitpunkt. Je nach verwendeter Technik gibt es Pausezeiten, in denen der Benutzer nicht mit dem System arbeiten kann. Dies ist allerdings oft nicht erwünscht, z. B. in Echtzeitsystemen. Folglich muss GC auch zu anderen Zeitpunkten ausgelöst werden können. Mit Hilfe verschiedener Heuristiken kann man versuchen, den optimalen Zeitpunkt zu finden. Wenn z. B. in einem System festgestellt wir, dass der Benutzer seit einiger Zeit inaktiv ist, könnte dies ein guter Zeitpunkt für GC sein, da der Benutzer wahrscheinlich noch einige Zeit inaktiv bleiben wird.
In einem System, in dem es nicht so sehr auf die Reduzierung der Pausezeiten ankommt, kann man auch nach Zeitpunkten suchen, zu denen GC am lohnendsten ist. Dies geschieht durch die Überwachung von sogenannten *key objects*. Sämtliche Speicherzellen, die von diesem Objekt abhängen, werden aus dem Bereich, der von der GC überwacht wird, ausgenommen. Sollte ein solches key object nicht mehr erreichbar, also tot sein, werden auch alle davon abhängenden Knoten als freier Speicher angesehen. Ein typisches Beispiel für ein key object ist die Wurzel eines Baumes.
Ebenfalls ein guter Zeitpunkt für GC wäre am Ende von längeren Berechnungen. Die Wahrscheinlichkeit, dass dann nur noch wenig lebende Objekte im Speicher sind, ist dann sehr groß und die zusätzliche Verzögerung durch die GC wäre nicht sehr stark bemerkbar.

5. Ausblick

Auch wenn schon viele gute Algorithmen und Techniken zur Garbage Collection existieren, wie z. B. Cheneys Copying Collector, ist die Forschung in diesem Bereich sehr aktiv und es befinden sich viele neue Techniken in der Entwicklung –zum Teil auch mit Rückgriff auf ältere Techniken, so wie auch Generational GC eine Weiterentwicklung von Copying GC ist. In diesem Rahmen sollten nur einige Beispiele nicht unerwähnt bleiben: Incremental Garbage Collection, welche kürzere Pausezeiten als ein *start-stop-collector* hat da das System nicht vollständig angehalten wird und Region Based Memory Management, welches den Speicher in verschiedene Regionen unterteilt (vgl. [6]).

Literatur

[1] Richard Jones & Rafael Lins
Garbage Collection: Algorithms for Automatic Dynamic Memory Management
John Wiley & Sons, New York, 1996

[2] John Mitchell
Concepts in Programming Languages
Cambridge University Press, 2003

[3] C. J. Cheney
A Nonrecursive List Compacting Algorithm
Communications of the ACM, 13(119):677-678, November 1970

[4] Andrew W. Appel
Simple Generational Garbage Collection and Fast Allocation
Software Practice and Experience, 1988

[5] David Ungar
Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm
SICPLAN Notices, 19(5):157-167, 1984

[6] Mads Tofte & Jean-Pierre Talpin
Region Based Memory Management
Information and Computation, 1997