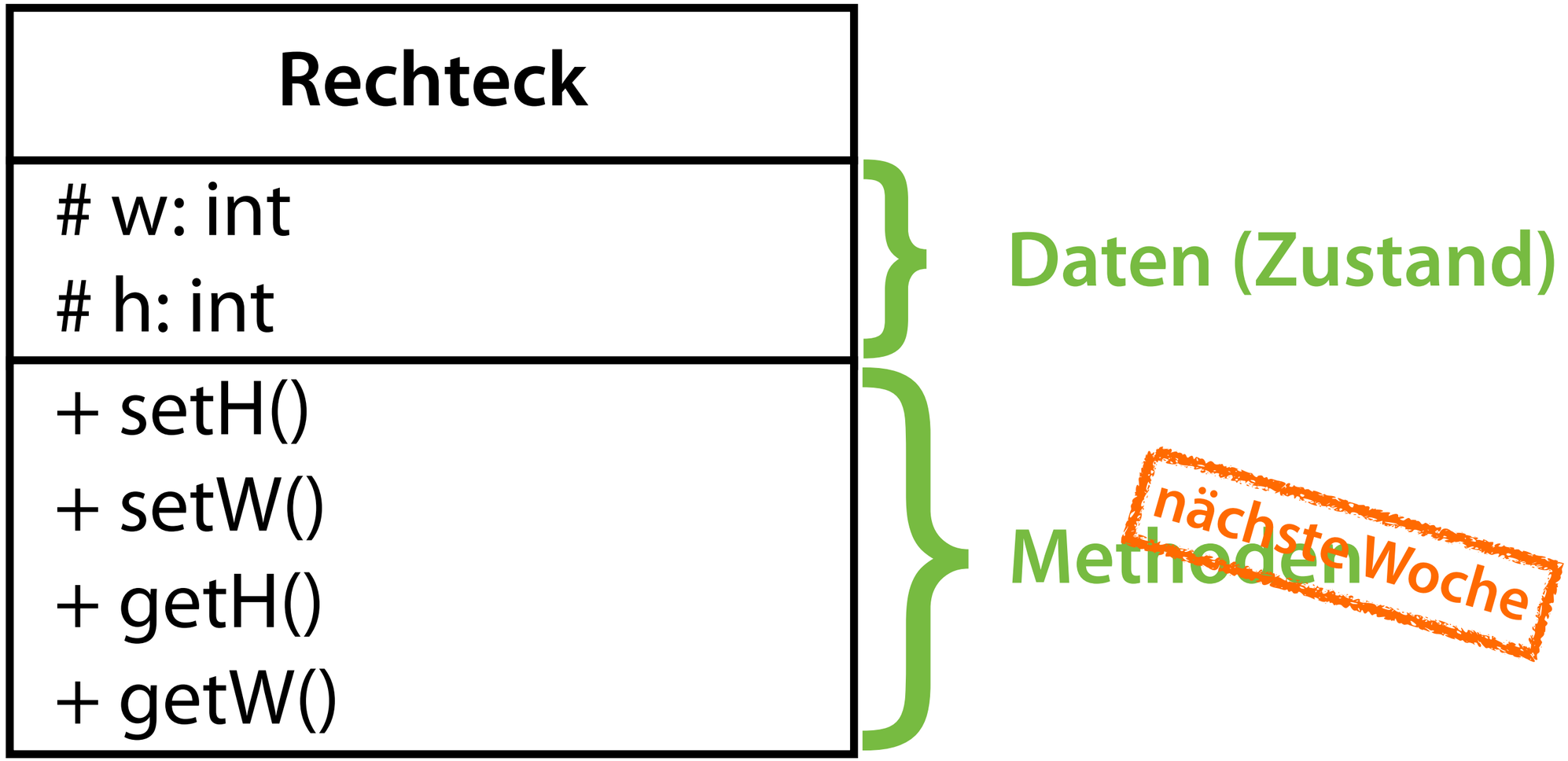
PT2

Unit03: Freispeicherverwaltung

Sven Köhler
Hasso-Plattner-Institut
2018-04-12

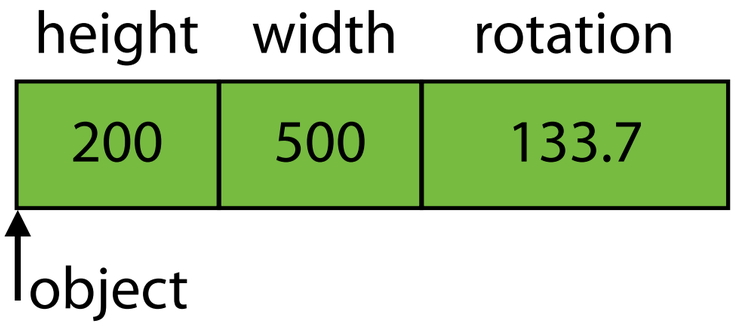

Rechteck
w: int
h: int
+ setH()
+ setW()
+ getH()
+ getW()
Daten (Zustand)
Methoden
nächste Woche

Wir brauchen eine neue Java-VM.

[https://www.youtube.com/watch?v=E3418SeWZfQ]

Variante 1 (C, C++, Java)
Attribute liegen hintereinander in einem **Speicherblock**.

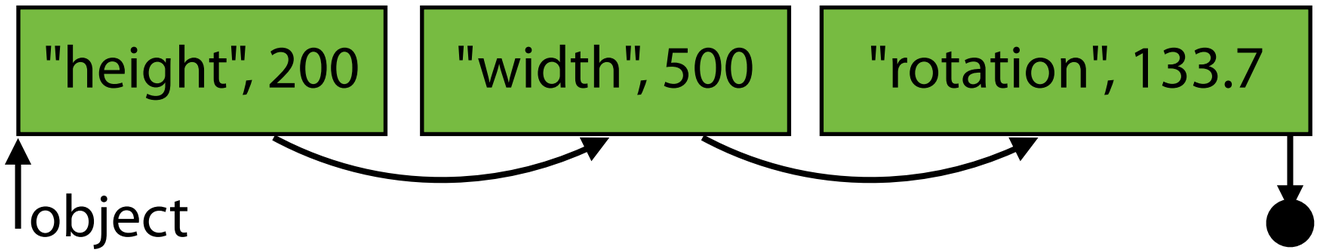
Jedes Attribut hat festen Abstand zum Objektanfang (in Bytes).



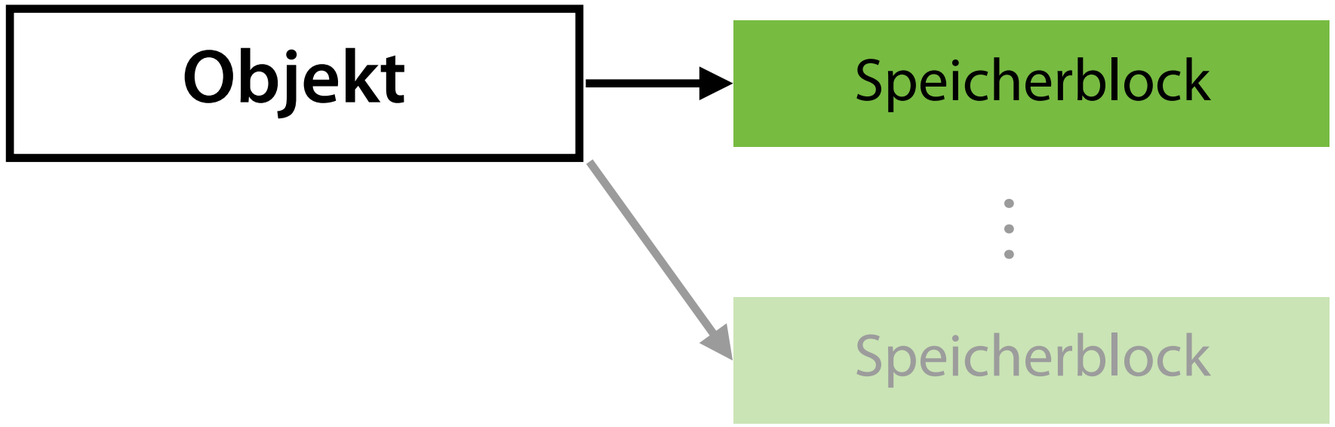
Einfache, effiziente Übersetzung in Maschinensprache.

Variante 2
Attribute liegen in einer **Container-Datenstruktur**

CLOS: Verkettete Liste (Name, Wert)-Paare
Python: Hashtabelle {Name: Wert}



Erlaubt Erweiterung mit zusätzlichen Attributen zur Laufzeit.



Objekt hat ein oder mehrere Speicherblöcke.
Jeder Block hat eine Anfangsadresse und Länge (implizit/explizit gegeben)

Anforderungen an eine **Freispeicherverwaltung**:

- Bereitstellung von Speicher für den Objektzustand
- Wiederverwendung freigegeben Speichers
- Freigabe nicht mehr verwendeter Objekte

heute

PT2

Unit03: Freispeicherverwaltung

Sven Köhler
Hasso-Plattner-Institut
2018-04-12

[https://xkcd.com/138/]

1 malloc & free

Allozierung ::

Bereitstellen von freiem Speicher.

```
void *malloc(size_t bytes);
new Foo(); // Java, C++
```

Eingabe: Benötigte Größe (explizit/implizit)
Rückgabe: Pointer auf Speicherblockanfang.

Deallozierung ::

Freigabe von belegtem Speicher (prüft evtl ob Speicher tatsächlich alloziert).

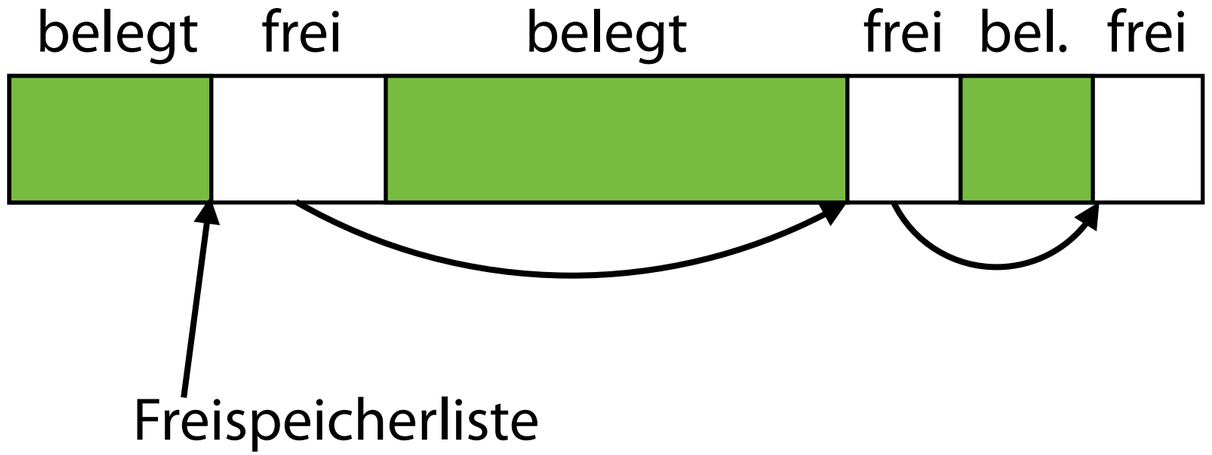
```
void free(void *data);
```

Eingabe: Pointer auf Anfang des Speicherblocks (evtl. Blockgröße)
Rückgabe: Keine

- Freispeicherverwaltung übl. Teil der Sprachumgebung/Standardbibliothek
- Alloziert großen Speicherblock vom Betriebssystem (sbrk, VirtualAlloc)
- Teilt den Speicher in kleinere Stücke, stellt Programm zur Verfügung
- Buchhaltung z.B. unbelegten Speichers ("Lücken") = Freispeicherliste

BS1

Bei Freigabe aus der Reihe => Fragmentierung



Aber wie implementieren wir nun malloc+free? *

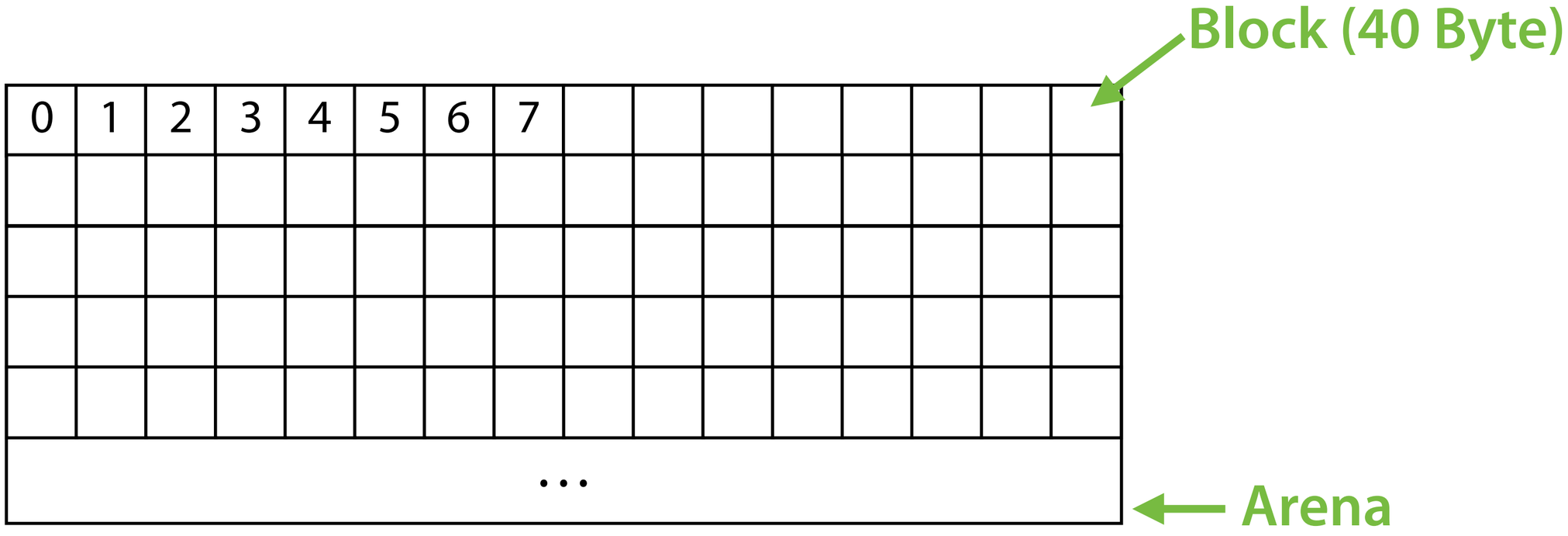
*oder eine andere, vergleichbare Schnittstelle,
die wir für unsere neue VM brauchen, da leider alle
Java-Implementierungen immer noch verschwunden sind.

1.1 Strategie I: Allokator für Objekte fester Größe

Annahmen:

Alle Objekte sind immer genau 40 Byte groß.

Gesamtspeicher liegt in einem kontinuierlichen Stück vor ("Arena")

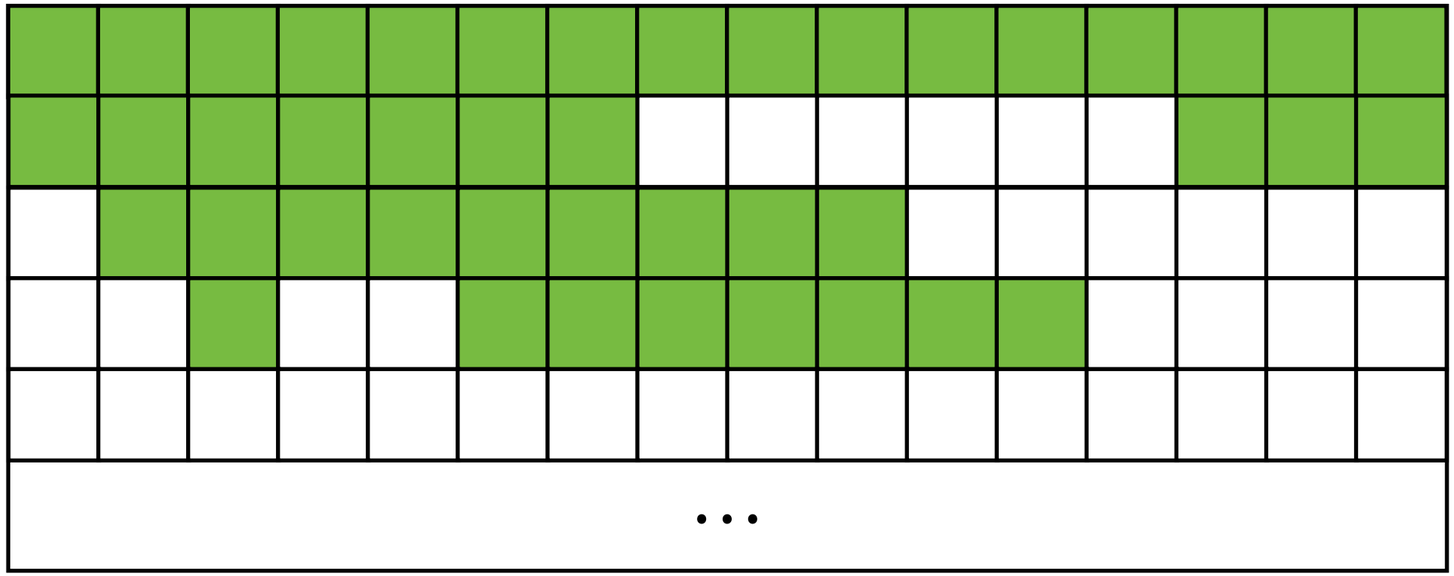


Arena wird in nummerierte 40 Byte-Blöcke unterteilt. Ein Block je Objekt.
Für jeden Block wird in einer zweiten Struktur gespeichert, ob er belegt ist.

< Welche Datenstruktur könnte die Belegung speichern? >

```
int isAllocated[NUM_BLOCKS];
```

Platzverschwendung!
Für jeden Block nur 1 Bit nötig.




```
1111111111111111
1111111000000111
0111111111000000
0010011111111110000
0000000000000000
```

"Bitmap"

Bits werden hintereinander in Bitmap gespeichert, z.B. in uint16_t.
Speicherbedarf der Bitmap: Zahl der Blöcke / 8 Byte.
Anfangs sind alle Blöcke frei, also Bits auf 0 gesetzt.

Algorithmus Allozierung

Suchen eines freien Bits in der Bitmap:
- Schleife über alle Worte der Bitmap
  - Schleife über alle Bits im Wort
    - Wenn Bit nicht gesetzt ist (=freier Block):
      - Bit setzen, Blockadresse zurückgeben

Wenn kein freies Bit gefunden,
ist der Speicher erschöpft.

Algorithmus Deallozierung

Ermitteln der Blocknummer
Ermitteln des entspr. Worts mit Bit zum Block
Ermitteln des Bits innerhalb des Worts.
Löschen des Bits (Freigabe des Blocks)

Optional:
- Test, ob gegebener Block einer gültigen Blocknummer entspricht.
- Test, ob Block tatsächlich alloziert ist.

Bit-Operationen in C:

1 << n: Erzeugen einer Bitmaske mit genau einem gesetzten Bit an Position n (== 2^n)
x | y: Bitweises Oder
x ^ y: Bitweises Exklusiv-Oder
x & y: Bitweises Und
~x: Bitweise Negation (Einerkomplement)

1.II

Strategie II: Freispeicherlisten

```
void *malloc(size_t bytes);
```

Größe gegeben

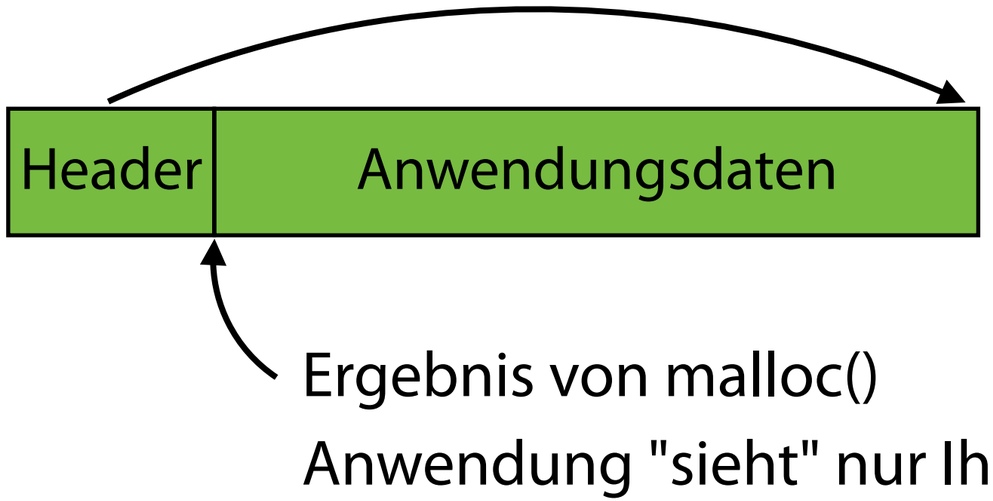
```
void free(void *data);
```

Größe nicht gegeben

Speicherverwaltung muss Blockgröße separat speichern

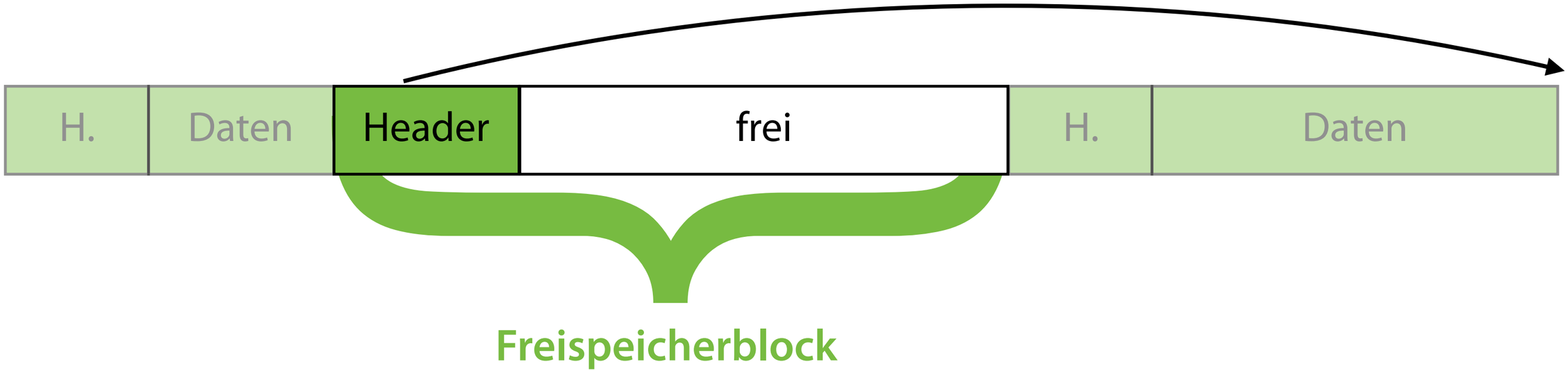
Idee: Blockgröße wird unmittelbar vor dem Speicherblock abgelegt

Bonus:
Keine extra Liste allozierter Speicherblöcken nötig, da Anwendung Blöcke explizit freigibt.



Problem: Freispeicherverwaltung benötigt Speicher für Liste freier Blöcke.

Idee: Jeder Freispeicherblock enthält eigene Listenverkettung
= Verweist auf nächsten freien Block.



Freispeicherheader benötigt Größe und Verweis auf nächsten Block.
(auf 64-bit Systemen also minimale Blockgröße von 16 Byte).

```
struct AllocatedHeader {
  int size;
};
```

```
struct FreeHeader {
  int size;
  struct FreeHeader* next;
};
```

Alignment ::
Anfangsaddresse des folgenden Datenblocks muss evtl. durch Wortbreite teilbar sein (z.B. 4 Byte auf 32-bit Systemen).
Header wird entsprechend "aligned".

Im Header oft auch Magic numbers, oder Prüfsummen enthalten um versehentliches Überschreiben (Heap-Smashing) zu erkennen.

Algorithmus Allozierung

- Gegeben: Größe des angeforderten Speichers
- Gesucht: Speicherblock, mindestens so groß wie der geforderte Block
- *<Strategie oder Magie>*
- Zerlegung eines großen Blocks in allozierten Block und freien Restblocks.
- Header anlegen und Zeiger auf Daten liefern.

```
void *malloc(size_t num_bytes)
{
  AllocatedHeader *found = magic();
  found->size = num_bytes;
  return (void *) found + 1;
}
```

Algorithmus Deallozierung

- Zurückrechnen auf Anfang des Headers.
- Block in Freispeicherliste eintragen.

Fragmentierung vermeiden:
Hintereinanderliegende freie Blöcke zu Größeren verschmelzen.
Empfehlung: Freispeicherliste nach Addressen aufsteigend sortiert halten.
Verschmelzung muss nach "vorn" und "hinten" möglich sein.

```
void free(void *data)
{
  FreeHeader *freeH = (FreeHeader *)
    ((AllocatedHeader*)data - 1);
  // naives eintragen in die Liste
  freeH->next = freelist;
  freelist = freeH;
}
```

Advanced

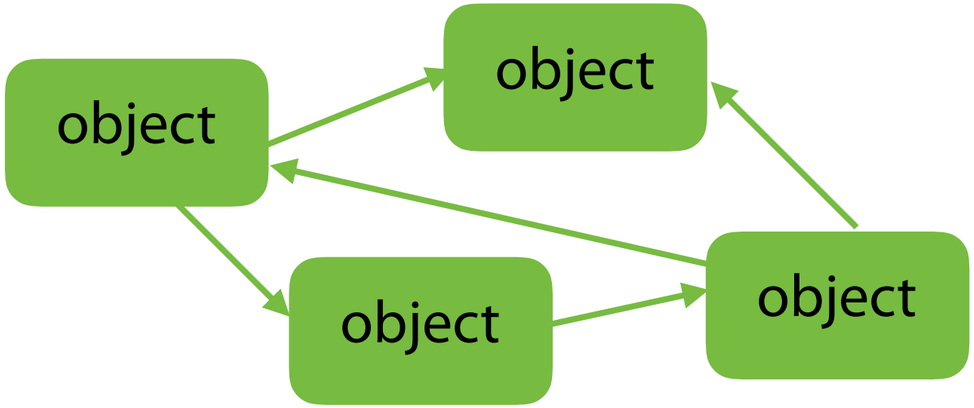
Strategien zum effizienten Finden freier Blöcke passender Größe sind immer noch aktueller Forschungsgegenstand. Zusammenführung analog.
Ideen: Pool-Allokatoren, Suchbäume, …

Moderne Implementierungen müssen **threadsicher** sein (gleichzeitige Manipulation der Freispeicherliste aus mehreren Threads ermöglichen).
Ideen: Seperate Liste je Thread, Atomare Operationen, …

— *If Java had an efficient garbage collector,*
it would collect itself.

2

Automatische Speicherverwaltung



Objekte verweisen mit Referenzen
auf andere Objekte => Objektgraph

Explizite Freigabe in C: free, C++: delete
Freigeben eines Objekts führt zur Nicht-Erreichbarkeit anderer Objekte.

Problem: Mehrfache Verweise auf dasselbe Objekte

dangling pointer: Speicher ist bereits freigegeben,
aber es gibt noch Pointer auf das Objekt

double free: Objekt mehrfach freigegeben

Automatische Speicherverwaltung: Allozierung explizit,
Deallozierung automatisch

In Java gibt es inherent lebendige Objekte:

- globale Variablen
- **GC roots**
  - laufende Threads
  - lokale Variablen aller aktiven Funktionen
  - Systemklassen (z.B. java.lang.String)

Ein Objekt ist erreichbar, wenn es ein GC root ist,
oder von einen erreichbaren Objekt referenziert wird.

Erreichbare Objekte sind u.U. nicht mehr verwendet.
Eine automatische Speicherverwaltung muss diese dennoch aufheben.

Jedes Objekt enthält einen Zähler
der Referenzen auf dieses Objekt
(aus anderen Objekten o. lok./glob. Variablen)

Objekt entsteht mit Referenzzähler = 1.
Erzeugen einer Referenz, erhöht den Zähler.
Ändern einer Referenz senkt den Zähler.

Wird der **Referenzzähler 0**,
wird das Objekt **freigegeben**.

In Folge werden Referenzzähler aller
referenzierten Objekte gesenkt.

A Method for Overlapping and Erasure of Lists

George E. Collins, *IBM Corp., Yorktown Heights, N. Y.*

Abstract. An important property of the Newell-Shaw-Simon scheme for computer storage of lists is that data having multiple occurrences need not be stored at more than one place in the computer. That is, lists may be "overlapped." Unfortunately, overlapping poses a problem for subsequent erasure. Given a list that is no longer needed, it is desired to erase just those parts that do not overlap other lists. In LISP, McCarthy employs an elegant but inefficient solution to the problem. The present paper describes a general method which enables efficient erasure. The method employs interspersed reference counts to describe the extent of the overlapping.

Introduction

The wide applicability of list processing is just beginning to be fully realized. This utility and versatility is due in large part to the ingenious scheme of representing lists in a computer storage which was devised by Newell, Shaw and Simon (see [4], for example). The primary merit of this scheme is that allocation of storage space for data is synthesized with the actual generation of the data. This is a practical necessity in many applications for which the quantities of data associated with some variables is highly unpredictable. This scheme achieves a sort of local optimization in that each "basic item" of data occupies a minimal amount of space.

A secondary, but often extremely important, merit of the scheme is that data having multiple occurrences often need not be stored more than one place in the computer. One may visualize this situation as the overlapping or intersection of lists, and its utilization may be regarded as a step toward global optimization of storage allocation.

However, the overlapping of lists leads to difficulties and complications in the erasure of lists (that is, in the return of words to the list of available storage). Given the location of a list that is no longer needed, it is desired to erase just those parts that do not overlap other lists. There is no general method of doing this short of making a survey of all lists in memory.

Two methods have been described so far in the literature for the solution of this difficulty, each of which has certain disadvantages. [illegible] motivated to devise a new method, described in this paper.

McCarthy's solution is very elegant, but unfortunate it contains two sources of inefficiency. First and m important, the time required to carry out this reclamati process is nearly independent of the number of regist reclaimed.[1] Its efficiency thereby diminishes drastically the memory approaches full utilization.[2] Second, method as used by McCarthy required that a bit (M Carthy uses the sign bit) be reserved in each word tagging accessible registers during the survey of accessi ity. If, as in our own current application of list processi much of the atomic data consists of signed integers floating-point numbers,[3] this results in awkwardness a further loss of efficiency.[4]

The second method, described by Gelernter et al. consists of adopting the convention that each list com nent is "owned" by exactly one list, and "borrowed" all other lists in which it appears. A "priority bit" is th used in each "location word" to indicate whether the entry referenced is owned or borrowed. An erasing rout is used which erases only those parts of a list that owned. This system obviously achieves the desired res only so long as no list is erased that owns a part that been borrowed by some other list not yet erased. In so applications (Gelernter's, in particular) this restrict may be easy to satisfy. In other applications (our own, particular) avoiding its violation is so difficult that method is quite useless.

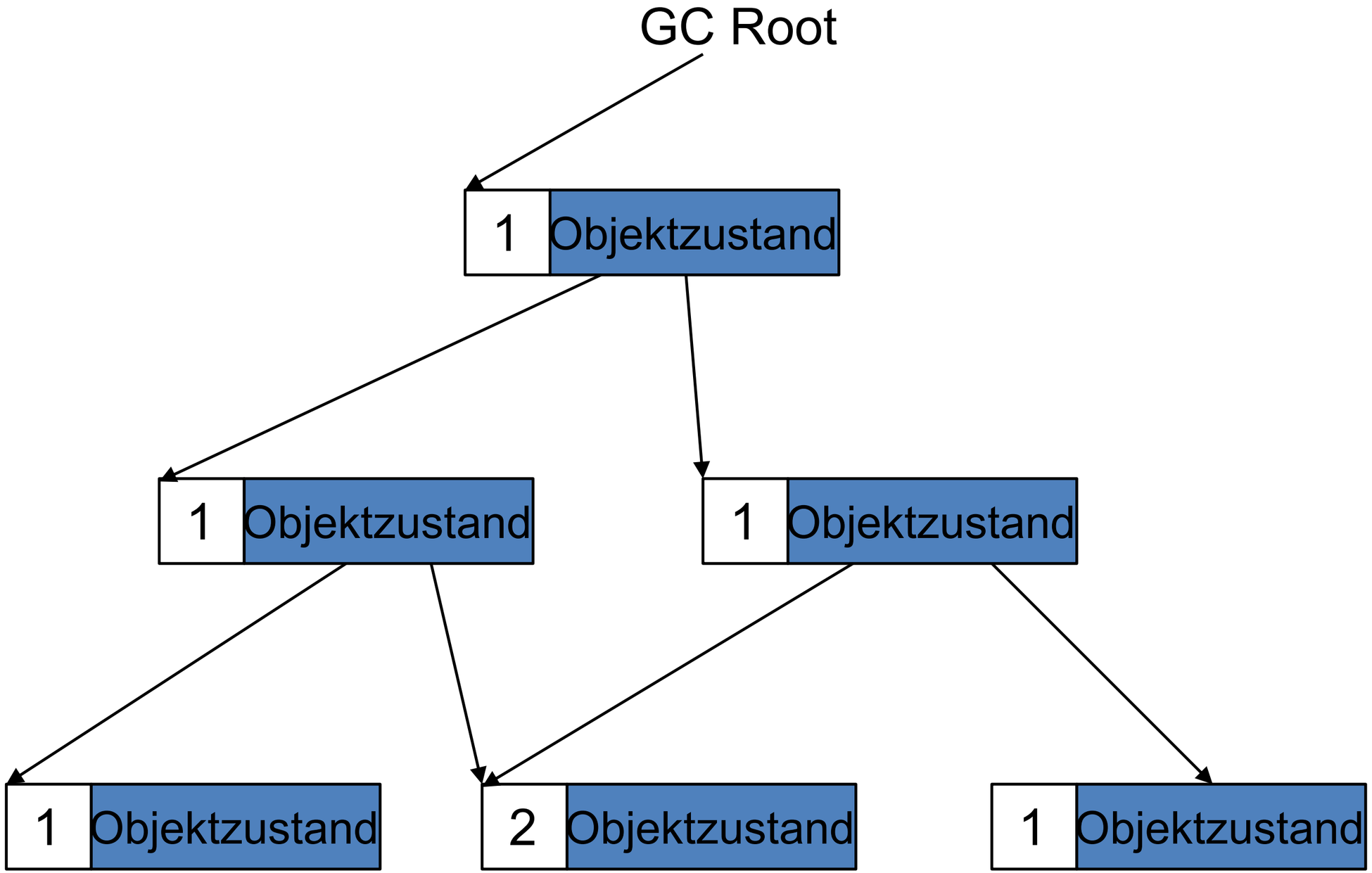
The method that we have devised consists, briefly, allowing the arbitrary interspersion (in lists) of words c taining reference counts. Viewed in terms of the conv tional diagrams of lists, such a reference count is a tally the number of arrows pointing to the box containing

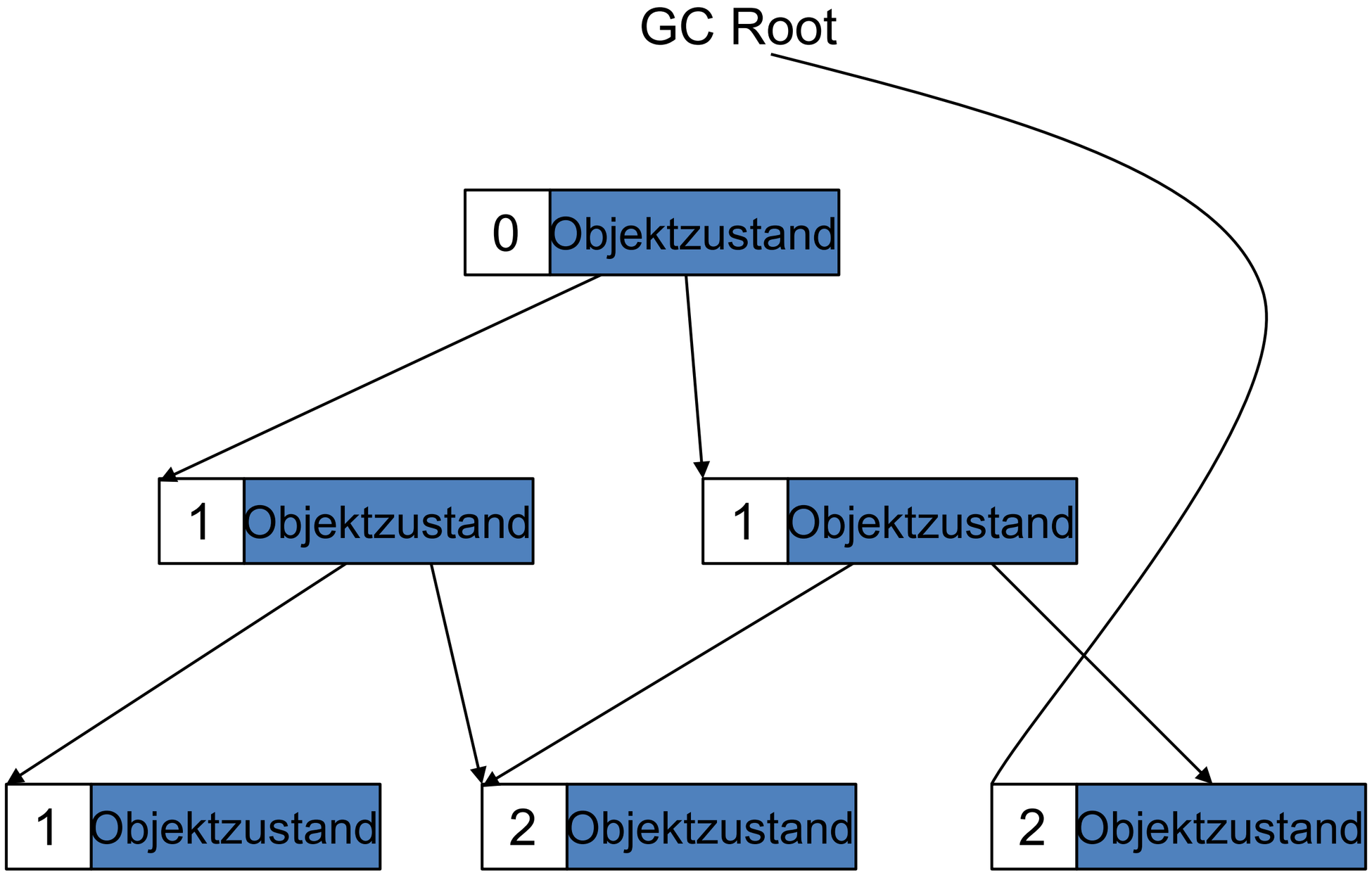
[1] This will of course depend on the computer, the applicati the skill of the programmer, and other factors. Our statemen based on estimates we have made for the IBM 7090, which indic that total reclamation time may even vary inversely with number of registers reclaimed.

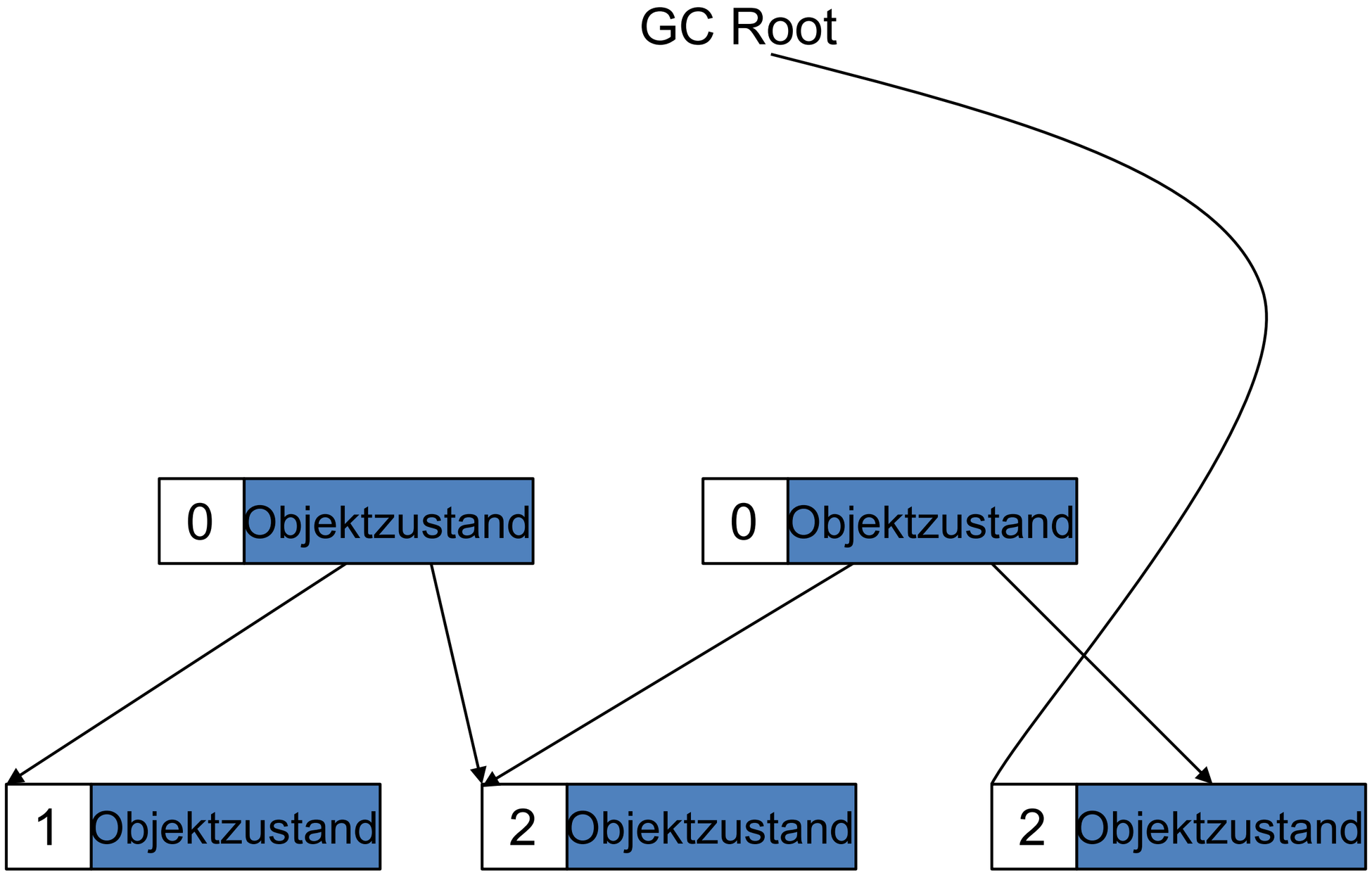
[2] Our estimates suggest that, even at a level of half-utilizat [illegible]

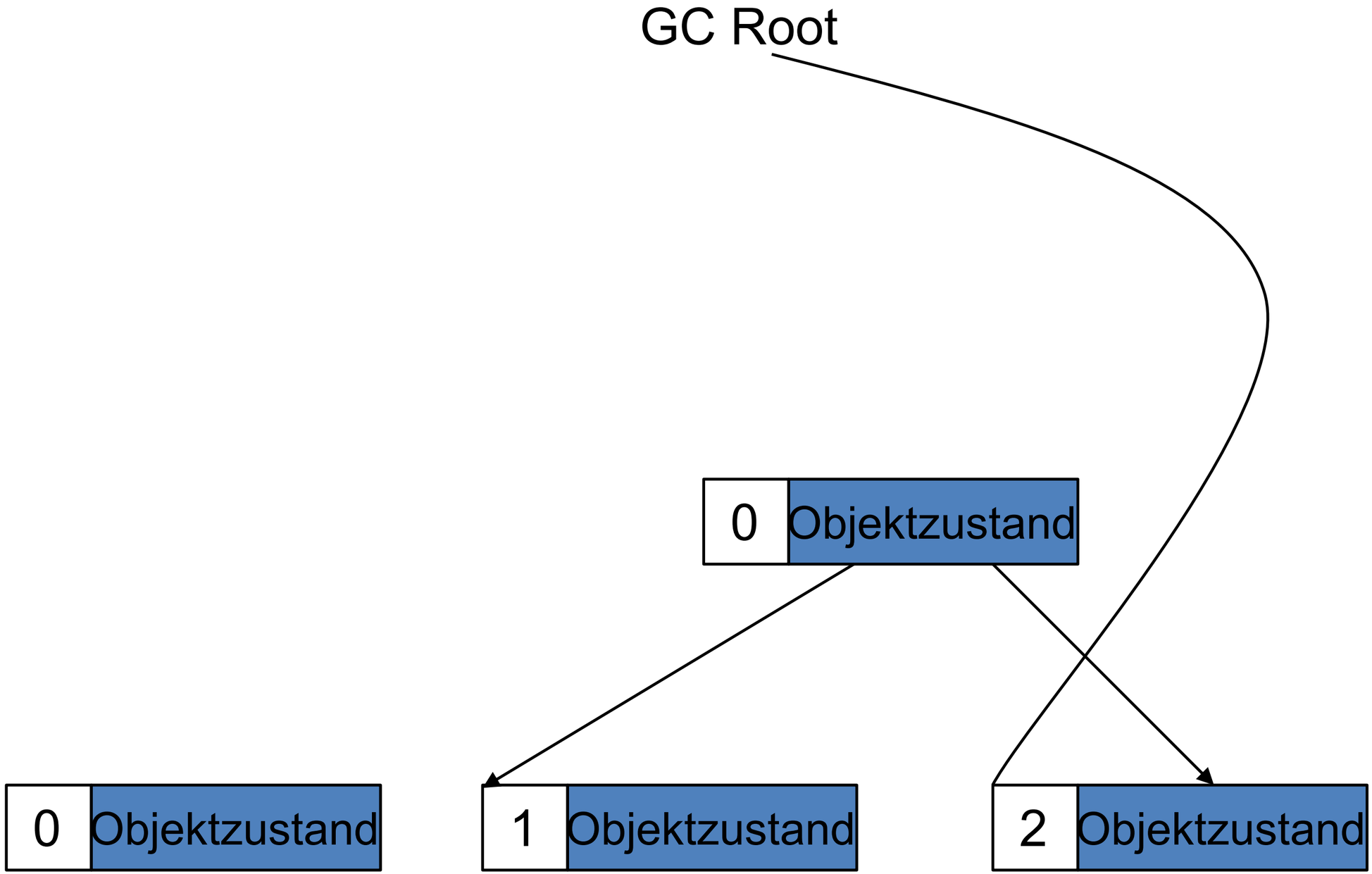
[3] This applicatio[illegible] in the development of a program fo modified form of A. Tarski's decision method for the element

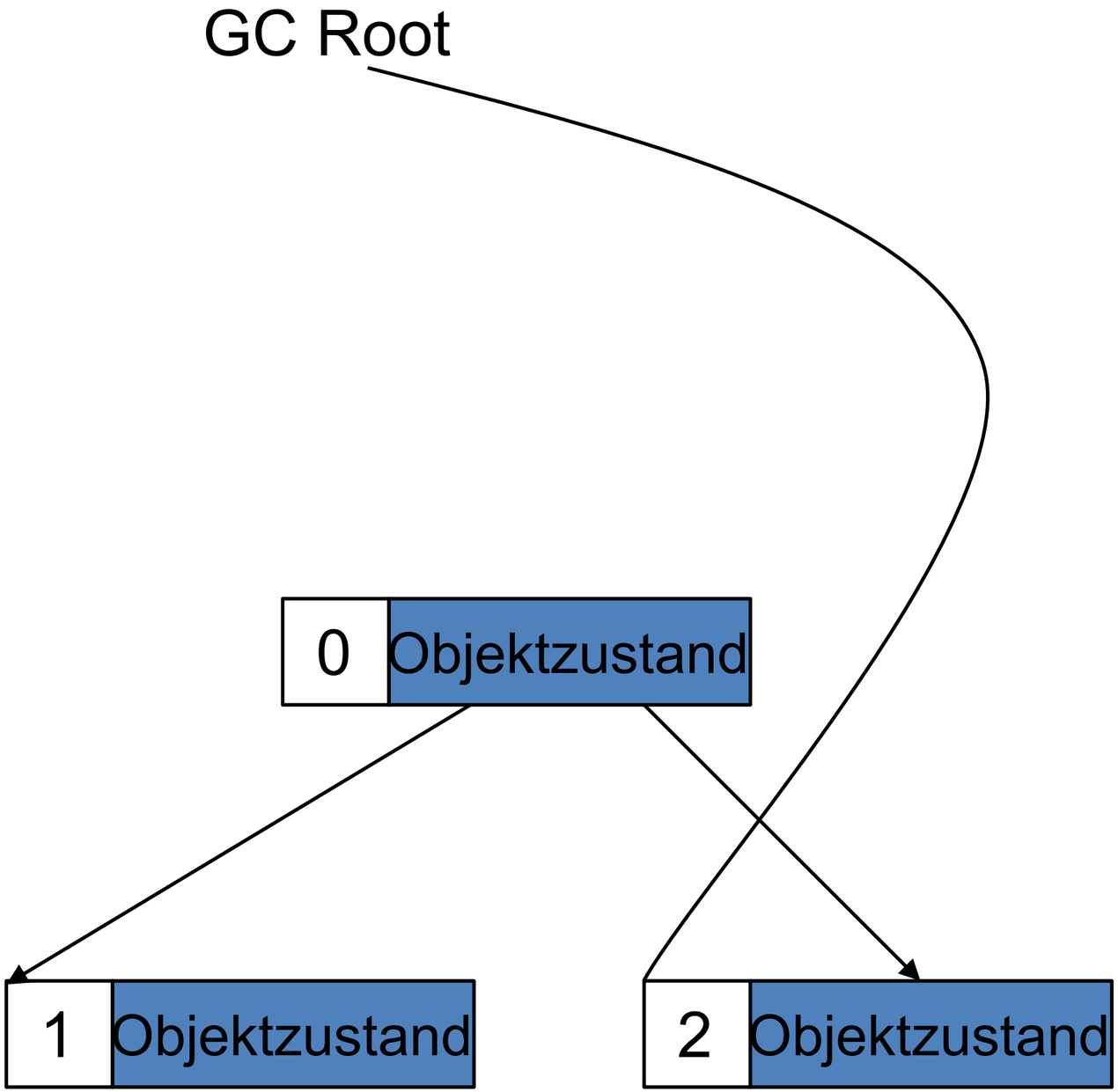
Referenzzählung (1960)

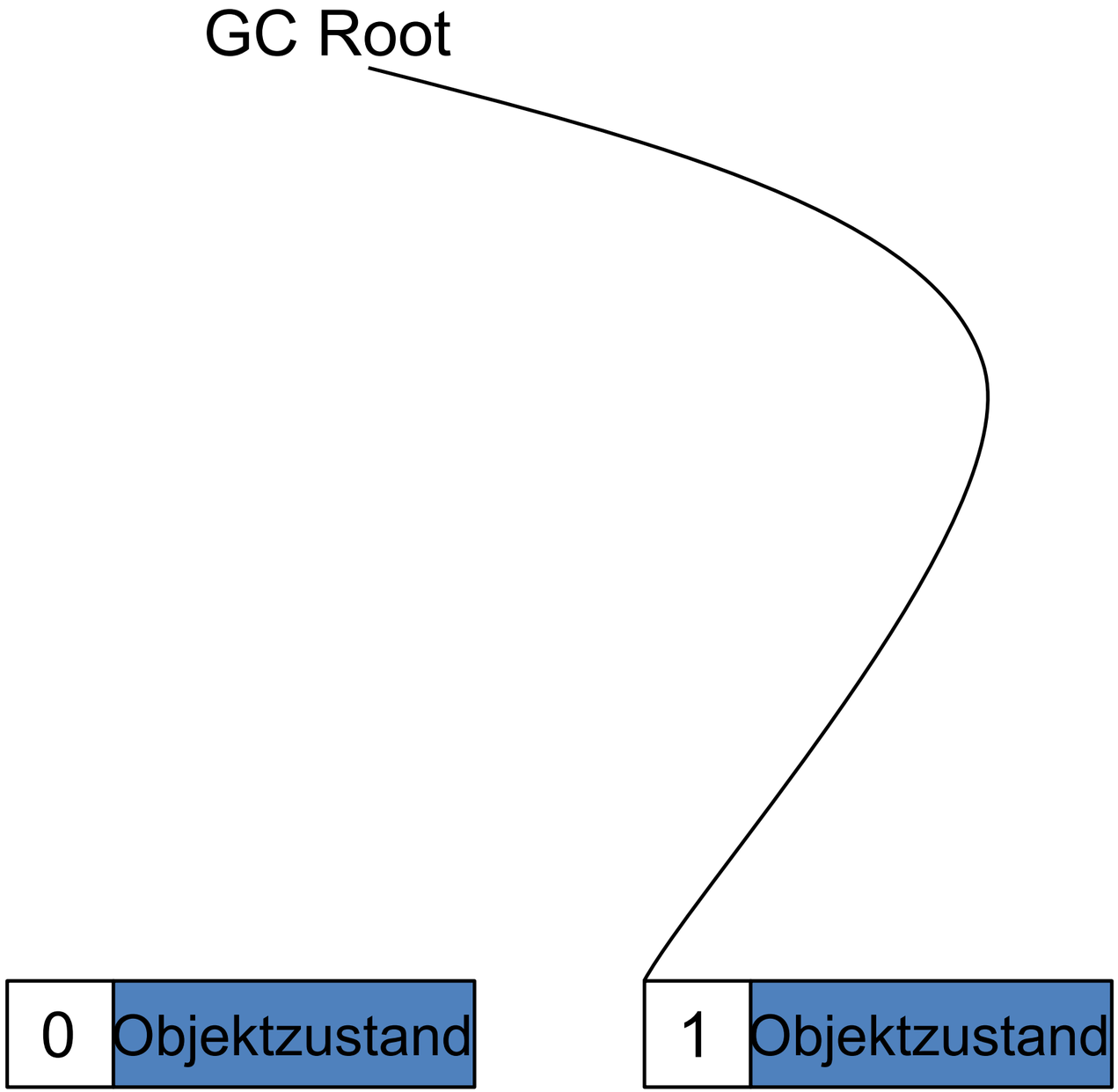

GC Root
1
Objektzustand
1
Objektzustand
1
Objektzustand
1
Objektzustand
2
Objektzustand
1
Objektzustand

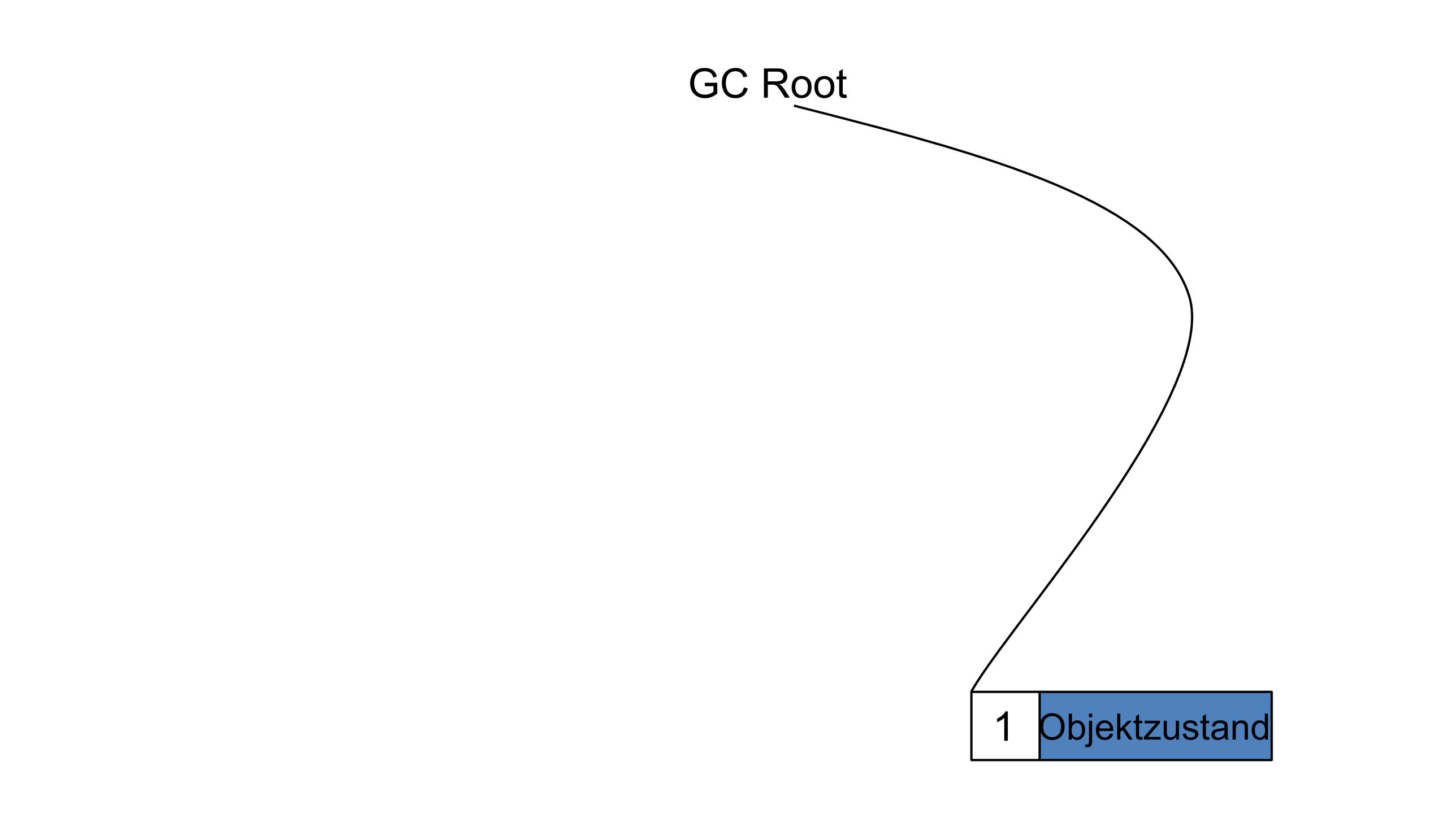

GC Root
0 Objektzustand
1 Objektzustand
1 Objektzustand
1 Objektzustand
2 Objektzustand
2 Objektzustand


GC Root
0
Objektzustand
0
Objektzustand
1
Objektzustand
2
Objektzustand
2
Objektzustand


GC Root
0
Objektzustand
0
Objektzustand
1
Objektzustand
2
Objektzustand


GC Root
0
Objektzustand
1
Objektzustand
2
Objektzustand

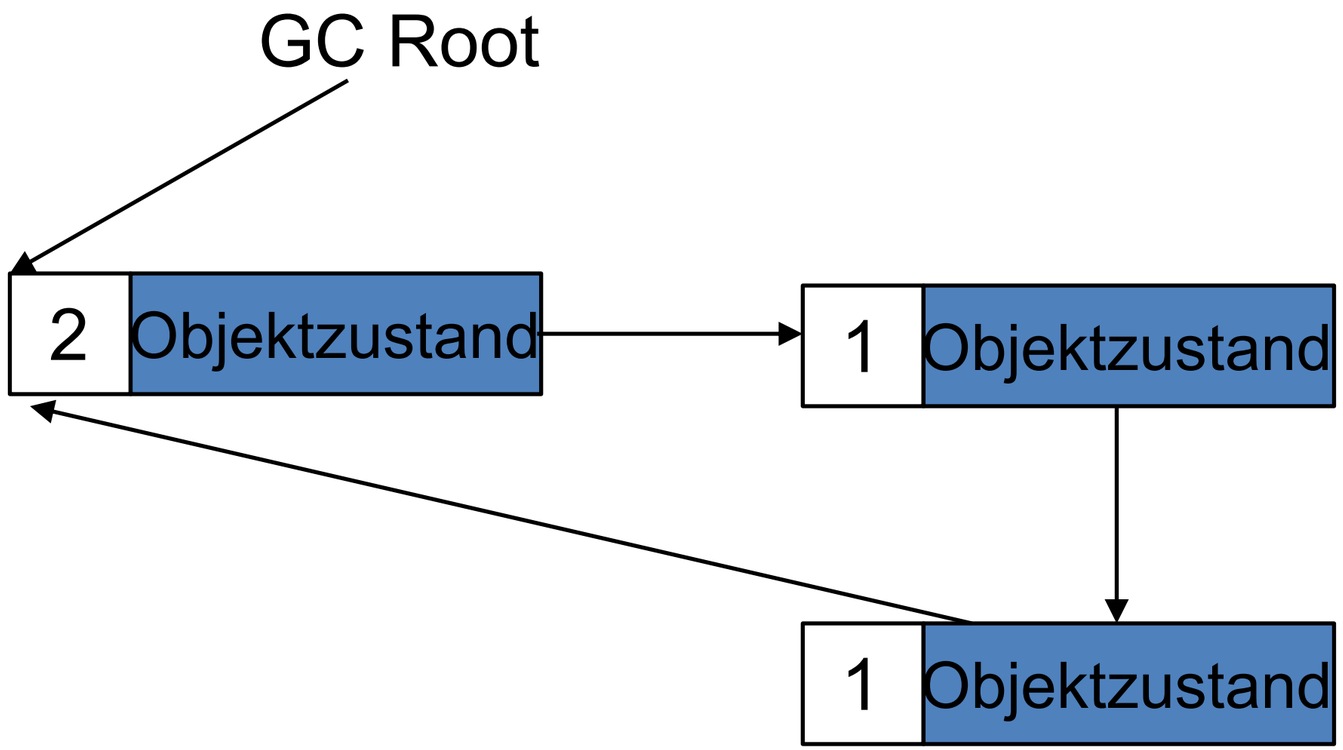

GC Root
0
Objektzustand
1
Objektzustand


GC Root
1
Objektzustand

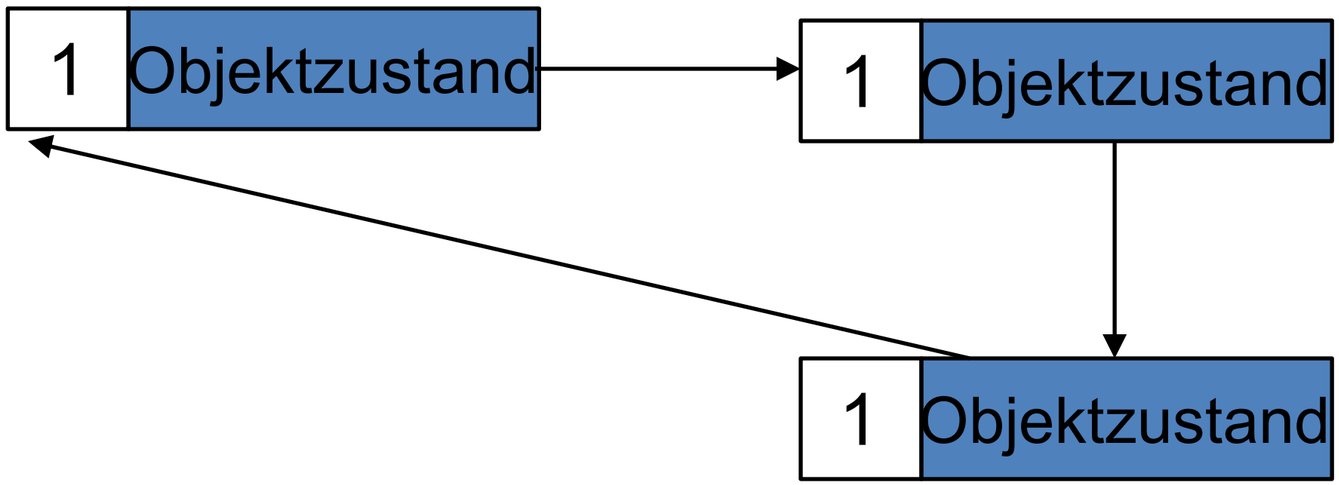
< Was ist ein Problem an dieser Methode? >

Zyklische Referenzen ::
Objekte verweisen gegenseitig aufeinander
(z.B. doppelt verkette Liste, Bäume mit Elternreferenz in jedem Knoten)

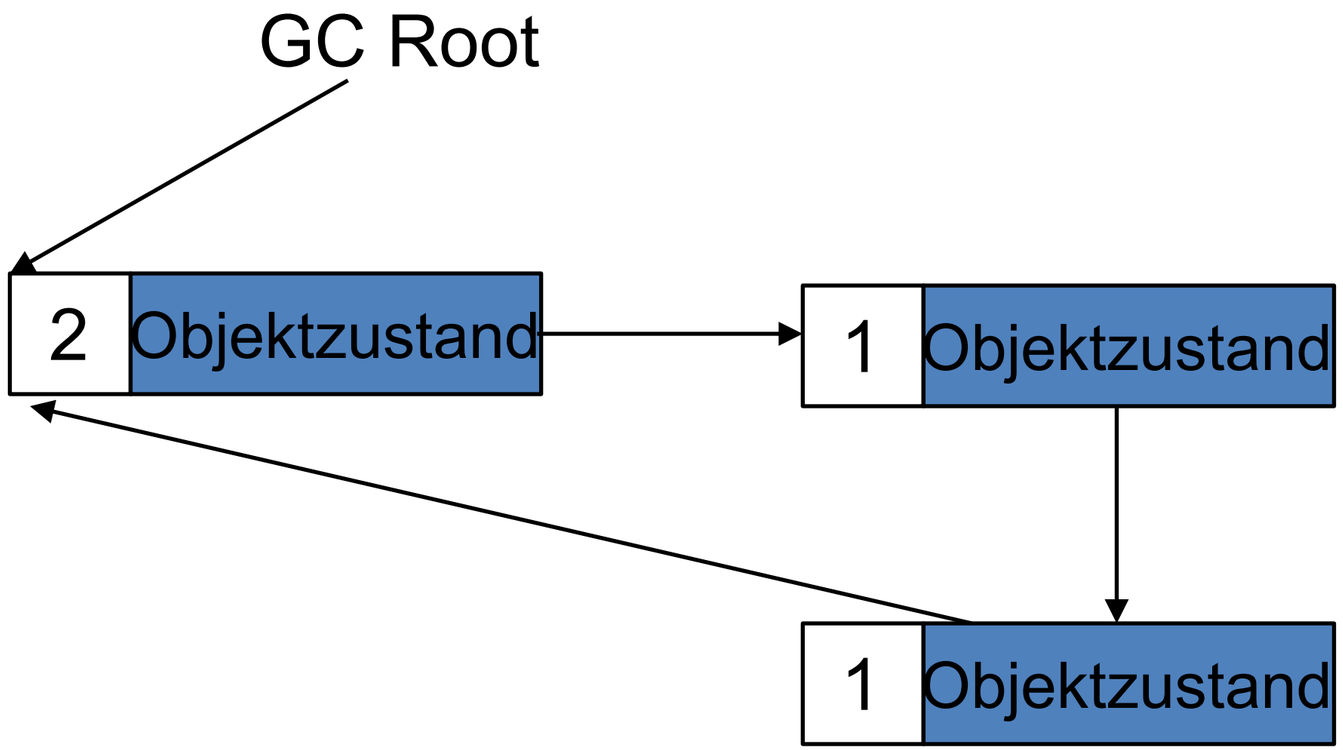
Referenzzähler kann nicht 0 werden.


GC Root
2
Objektzustand
1
Objektzustand
1
Objektzustand

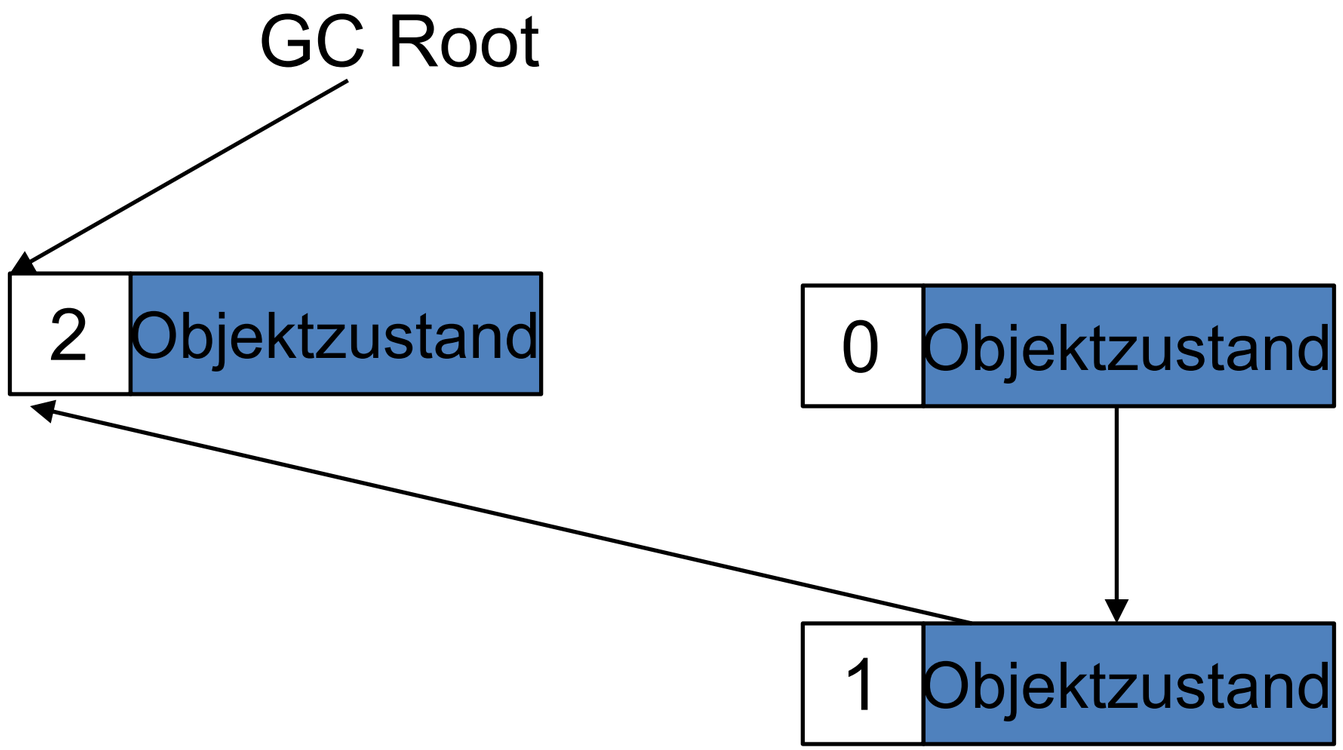
GC Root (null)



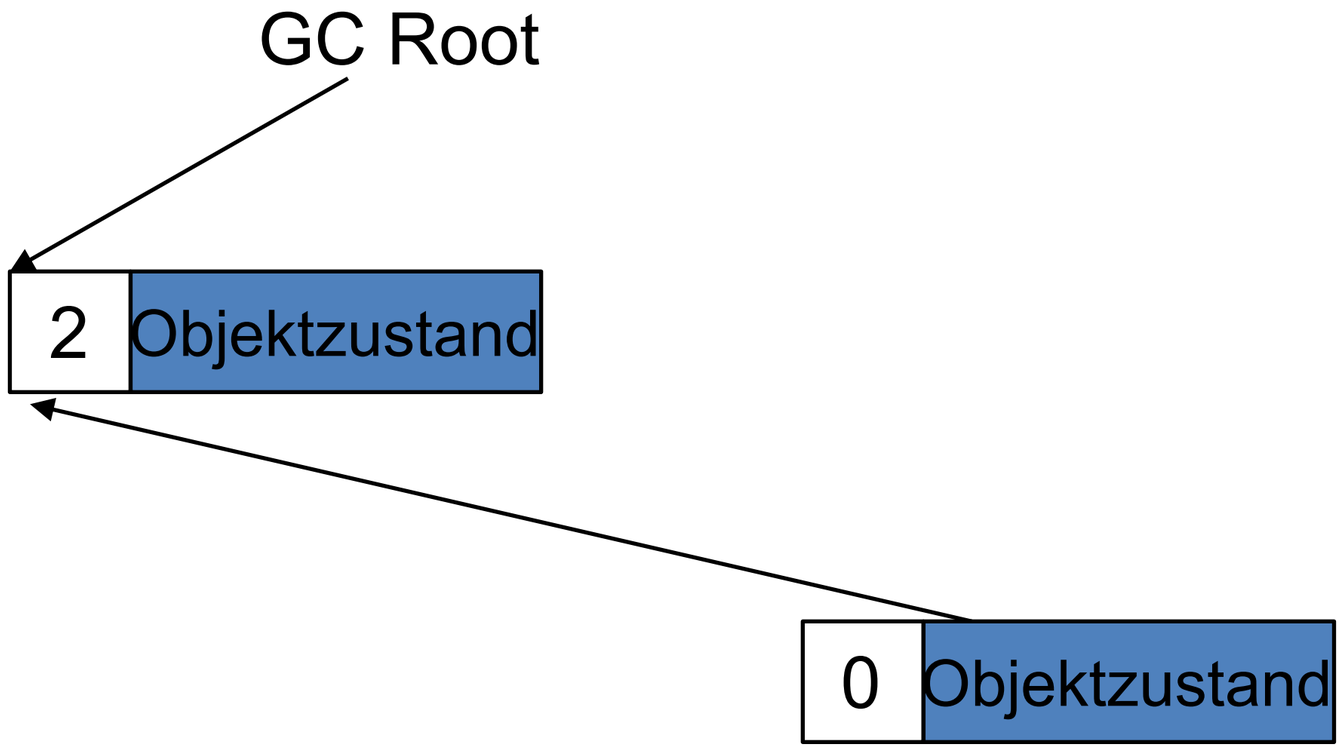
Lösung: Zyklus explizit aufbrechen (GCRoot.next = null)



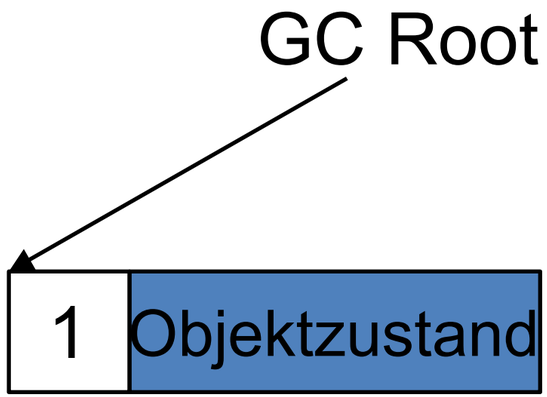
Lösung: Zyklus explizit aufbrechen (GCRoot.next = null)



Lösung: Zyklus explizit aufbrechen (GCRoot.next = null)



Lösung: Zyklus explizit aufbrechen (GCRoot.next = null)



Lösung: Zyklus explizit aufbrechen (GCRoot = null)

GC Root (null)

Lösung: Zyklus explizit aufbrechen (GCRoot = null)

GC Root (null)

Lösung 2: strong & weak references in der Sprache.

Starke Referenzen (auch shared) werden gezählt.
Schwache Referenzen verweisen auf ein Objekt ohne Zähler, erlauben aber keinen Zugriff. Erst bei Verwendung wird eine temporäre starke Referenz erzeugt.

Programmier sind für richtige Wahl verantwortlich.

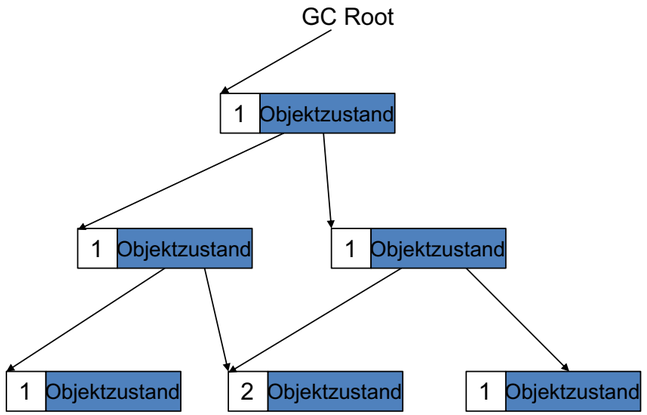
Beispiele: iOS-ARC, C++-STL (std::weak_ptr)

Vorteile Referenzzählung

- Existiert keine Referenz mehr, werden Objekte sofort freigegeben
  (meisten Objekte haben eine oder "wenige" Referenzen)
- Kosten der Speicherverwaltung gleichmäßig auf Programmoperationen verteilt.
  (potentiell echtzeitfähig, wenn Rekursionstiefe absehbar)

Nachteile Referenzzählung

- Explizite Behandlung von Zyklen nötig.
  CPython nutzt z.B. zusätzlich einen Garbage Collector für zyklische Strukturen
- Atomares Inkrementieren und Dekrementieren nötig für Threadsicherheit
- Zusätzlicher Speicher für Referenzzähler pro Objekt



1. Auffinden aller erreichbarer Objekte
   beginnend bei GC Roots
2. Freigeben aller nicht-erreichbarer Objekte
3. ???
4. Profit!

Optional: Fragmentierung beseitigen durch
Verschieben aller erreichbaren Objekte
(compacting GC): Alle Zeiger aktualisieren.

Garbage Collection

3

Wann Müll sammeln?

Speicher erschöpft

Regelmäßig nach n Sekunden

Regelmäßig nach m Allokationen

3.1

Strategie I: Mark-and-Sweep GC

Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I

John McCarthy, Massachusetts Institute of Technology, Cambridge, Mass. *

April 1960

1 Introduction

A programming system called LISP (for LISt Processor) has been developed for the IBM 704 computer by the Artificial Intelligence group at M.I.T. The system was designed to facilitate experiments with a proposed system called the Advice Taker, whereby a machine could be instructed to handle declarative as well as imperative sentences and could exhibit "common sense" in carrying out its instructions. The original proposal [1] for the Advice Taker was made in November 1958. The main requirement was a programming system for manipulating expressions representing formalized declarative and imperative sentences so that the Advice Taker system could make deductions.

In the course of its development the LISP system went through several stages of simplification and eventually came to be based on a scheme for representing the partial recursive functions of a certain class of symbolic expressions. This representation is independent of the IBM 704 computer, or of any other electronic computer, and it now seems expedient to expound the system by starting with the class of expressions called S-expressions and the functions called S-functions.

*Putting this paper in LaTeXpartly supported by ARPA (ONR) grant N00014-94-1-0775 to Stanford University where John McCarthy has been since 1962. Copied with minor notational changes from *CACM*, April 1960. If you want the exact typography, look there. Current address, John McCarthy, Computer Science Department, Stanford, CA 94305, (email: jmc@cs.stanford.edu), (URL: http://www-formal.stanford.edu/jmc/)

Zwei Phasen: Markieren und Durchwischen.

Mark:

Von den GC Roots werden alle Objekte durchwandert,
jedes gefundene markiert.

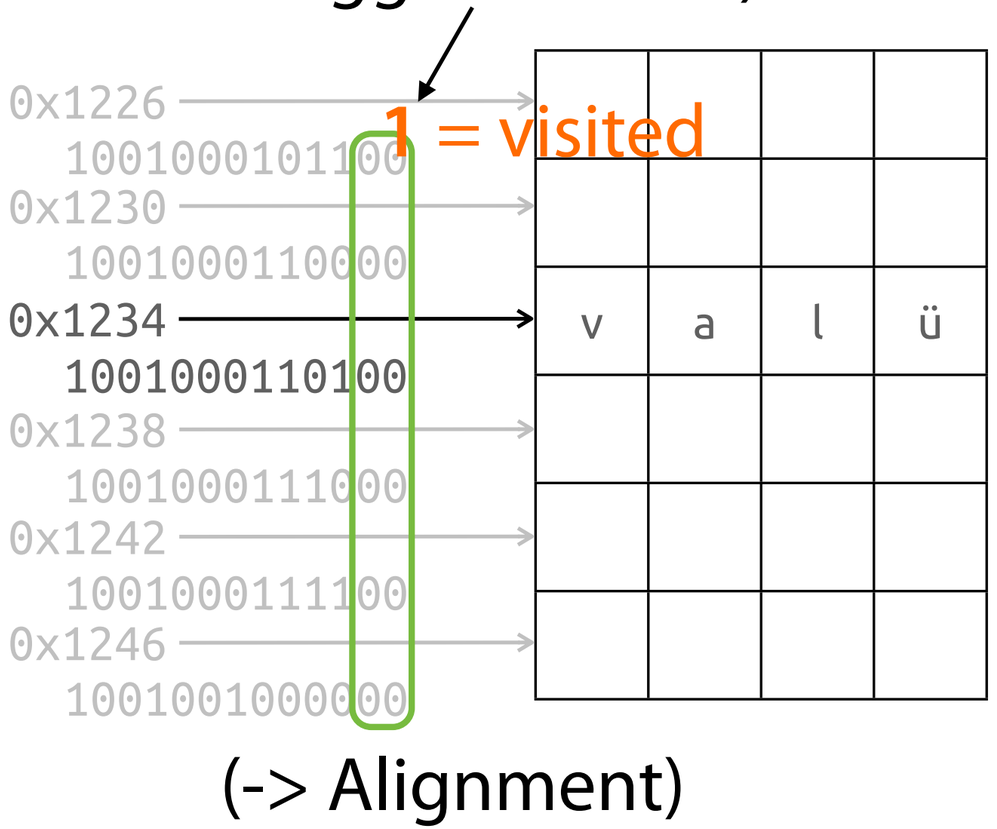
Alle Objektreferenzen rekursiv weiterverfolgt,
abbrechen wenn referenziertes Objekt bereits markiert ist.

Sweep:

Alle Objekte löschen, die nicht markiert sind.

Überlegungen zur Implementierung

Wie wird das Mark-Bit repräsentiert?
(seperater Speicher, freies Bit im Objekt, oder als Tagged Pointer)



(-> Alignment)

Rekursiv oder iterativ?
Rekursiv: Stacküberlauf bei tief verschachtelten Datenstrukturen.
Iterativ: Speicherung einer Liste noch zu bearbeitender Objekte.

Vorteile (gegenüber Referenzzählung)

- Kann automatisch Zyklen löschen
- Keine Zwischenkosten zur Laufzeit bei Pointermanipulation

Nachteile

- Stopp-Start-Algorithmus: Berechnung wird für GC unterbrochen
- Laufzeitkosten schwer vorhersagbar (abhängig von Zahl allozierter Objekte)

3.II

Strategie II: Copying GC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Project MAC
Cambridge, Massachusetts

Artificial Intelligence Project
Memo 58 (Revised)

Memorandum MAC-M-129
December 27, 1963

A LISP Garbage Collector Algorithm Using Serial Secondary Storage*

by M. L. Minsky

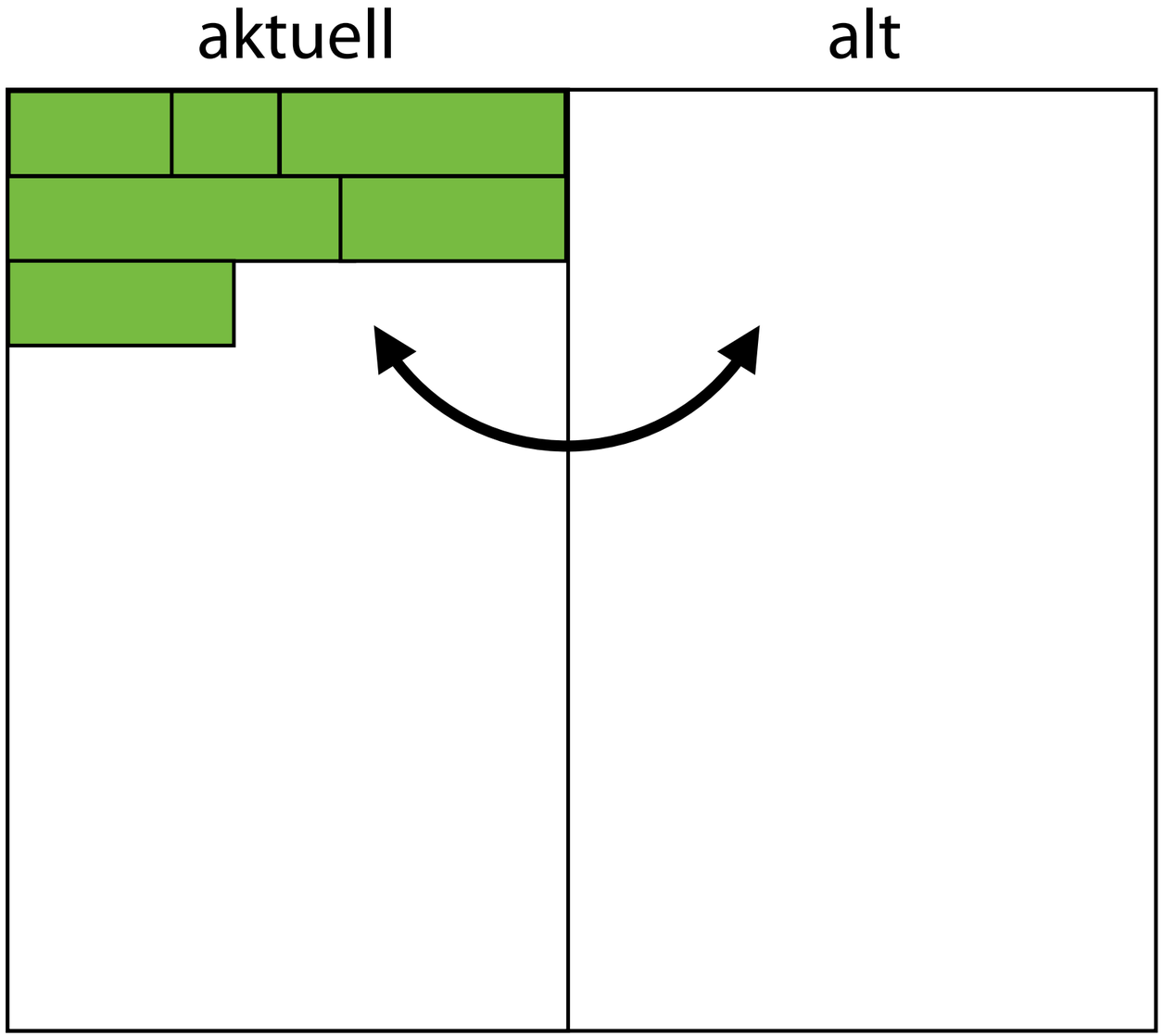
Paper to be presented at the First International LISP Conference, Mexico City, Mexico, December 30 - January 3, 1964.

This paper presents an algorithm for reclaiming unused free storage memory cells in LISP. It depends on availability of a fast secondary storage device, or a large block of available temporary storage. For this price, we get

1. Packing of free-storage into a solidly packed block.
2. Smooth packing of arbitrary linear blocks and arrays.
3. The collector will handle arbitrarily complex re-entrant list structure with no introduction of spurious copies.
4. The algorithm is quite efficient; the marking pass visits words at most twice and usually once, and the loading pass is linear.
5. The system is easily modified to allow for increase in size of already fixed consecutive blocks, provided one can afford to initiate a collection pass or use a modified array while waiting for such a pass to occur.

collect[z;n] is the function that finds all list-structure depending from z, and puts out on drum the

Idee: Speicher in zwei Halbräume geteilt



Allokationen immer im "aktuellen" Halbraum.

GC: Halbräume tauschen ihre Rollen

lebendige Objekte werden vom alten Raum in den aktuellen Raum kopiert.

Referenzen in kopierten Objekten rekursiv verfolgt

Nicht kopierte Objekte sind einfach verworfen

Implementierungsaspekte

- Wie werden Objektadressen repräsentiert?
  - Globale Objekttabelle ("Handles"): Aktualisierung der Pointer vom alten Halbraum auf den aktuellen
  - Alternative: Altes Objekt kann Pointer auf ein neues Objekt enthalten
    - Bei Kopieren der zweiten Referenz, muss nur die Addresse aktualisiert werden.

Vorteile

- Effiziente Allozierung (Speicher stets am Ende des aktuellen Halbraums)
- Keine Speicherfragmentierung (Compacting GC)

Nachteile

- Ineffiziente Speichernutzung
- Kopieren zeitaufwendig (zum weiterlesen: Memory-Wall)

3.III

Varianten

- Mark-Compact: erreichbare Objekte werden in Lücken verschoben

- Objektgenerationen (*generational GC*)
  - "most objects die young" – junge Objekte verweisen i.d.R. auf alte, nicht umgekehrt
  - Objekte werden in Generationen unterteilt,
    GC betrachtet meist nur die “jüngste” Generation
  - erreichbare Objekte der jüngsten Generation (*survivors*)
    rücken in die nächste Generation auf (*promotion*)

- Vermeidung von Rekursion

- Tiefe-zuerst oder Breite-zuerst

- Garbage Collection im Hintergrund (multi-threading)
  - inkrementelle GC

GC-Varianten (Auswahl)

Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I

John McCarthy, Massachusetts Institute of Technology, Cambridge, Mass. *

April 1960

1 Introduction

A programming system called LISP (for LISt Processor) has been developed for the IBM 704 computer by the Artificial Intelligence group at M.I.T. The system was designed to facilitate experiments with a proposed system called the Advice Taker, whereby a machine could be instructed to handle declarative as well as imperative sentences and could exhibit "common sense" in carrying out its instructions. The original proposal [1] for the Advice Taker was made in November 1958. The main requirement was a programming system for manipulating expressions representing formalized declarative and imperative sentences so that the Advice Taker system could make deductions.

In the course of its development the LISP system went through several stages of simplification and eventually came to be based on a scheme for representing the partial recursive functions of a certain class of symbolic expressions. This representation is independent of the IBM 704 computer, or of any other electronic computer, and it now seems expedient to expound the system by starting with the class of expressions called S-expressions and the functions called S-functions.

*Putting this paper in LaTeXpartly supported by ARPA (ONR) grant N00014-94-1-0775 to Stanford University where John McCarthy has been since 1962. Copied with minor notational changes from *CACM*, April 1960. If you want the exact typography, look there. Current address, John McCarthy, Computer Science Department, Stanford, CA 94305, (email: jmc@cs.stanford.edu), (URL: http://www-formal.stanford.edu/jmc/)

Method for Overlapping and Erasure of Lists

George E. Collins, *IBM Corp., Yorktown Heights, N. Y.*

important property of the Newell-Shaw-Simon uter storage of lists is that data having multiple l not be stored at more than one place in the is, lists may be "overlapped." Unfortunately, s a problem for subsequent erasure. Given a list needed, it is desired to erase just those parts that her lists. In LISP, McCarthy employs an elegant ution to the problem. The present paper describes d which enables efficient erasure. The method rsed reference counts to describe the extent of

plicability of list processing is just beginning lized. This utility and versatility is due in he ingenious scheme of representing lists in orage which was devised by Newell, Shaw [4], for example). The primary merit of this t allocation of storage space for data is th the actual generation of the data. This is cessity in many applications for which the ata associated with some variables is highly This scheme achieves a sort of local optimiach "basic item" of data occupies a minimal e.

, but often extremely important, merit of that data having multiple occurrences often ored more than one place in the computer. alize this situation as the overlapping or lists, and its utilization may be regarded as global optimization of storage allocation.

he overlapping of lists leads to difficulties ions in the erasure of lists (that is, in the s to the list of available storage). Given the st that is no longer needed, it is desired to e parts that do not overlap other lists. There ethod of doing this short of making a survey emory.

s have been described so far in the literature n of this difficulty, each of which has certain or limitations. As a result we have been devise a new method, described in this

thod, due to McCarthy, is described in [3]. method individual lists are "abandoned"

McCarthy's solution is very elegant, but unfortunately it contains two sources of inefficiency. First and most important, the time required to carry out this reclamation process is nearly independent of the number of registers reclaimed.[1] Its efficiency thereby diminishes drastically as the memory approaches full utilization.[2] Second, the method as used by McCarthy required that a bit (McCarthy uses the sign bit) be reserved in each word for tagging accessible registers during the survey of accessibility. If, as in our own current application of list processing, much of the atomic data consists of signed integers or floating-point numbers,[3] this results in awkwardness and further loss of efficiency.[4]

The second method, described by Gelernter et al. [2], consists of adopting the convention that each list component is "owned" by exactly one list, and "borrowed" by all other lists in which it appears. A "priority bit" is then used in each "location word" to indicate whether the list entry referenced is owned or borrowed. An erasing routine is used which erases only those parts of a list that are owned. This system obviously achieves the desired result only so long as no list is erased that owns a part that has been borrowed by some other list not yet erased. In some applications (Gelernter's, in particular) this restriction may be easy to satisfy. In other applications (our own, in particular) avoiding its violation is so difficult that the method is quite useless.

The method that we have devised consists, briefly, in allowing the arbitrary interspersion (in lists) of words containing reference counts. Viewed in terms of the conventional diagrams of lists, such a reference count is a tally of the number of arrows pointing to the box containing the

[1] This will of course depend on the computer, the application, the skill of the programmer, and other factors. Our statement is based on estimates we have made for the IBM 7090, which indicate that total reclamation time may even vary inversely with the number of registers reclaimed.

[2] Our estimates suggest that, even at a level of half-utilization of memory, execution time per word reclaimed would be approximately three times as great for McCarthy's method as compared with our own.

[3] This application is in the development of a program for a modified form of A. Tarski's decision method for the elementary theory of real closed fields. See [1] and [5]. In our adaptation all formulas are normalized and arithmetized. Further, floating-point

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Project MAC
Cambridge, Massachusetts

Artificial Intelligence Project
Memo 58 (Revised)

Memorandum MAC-M-129
December 27, 1963

A LISP Garbage Collector Algorithm Using Serial Secondary Storage*

by M. L. Minsky

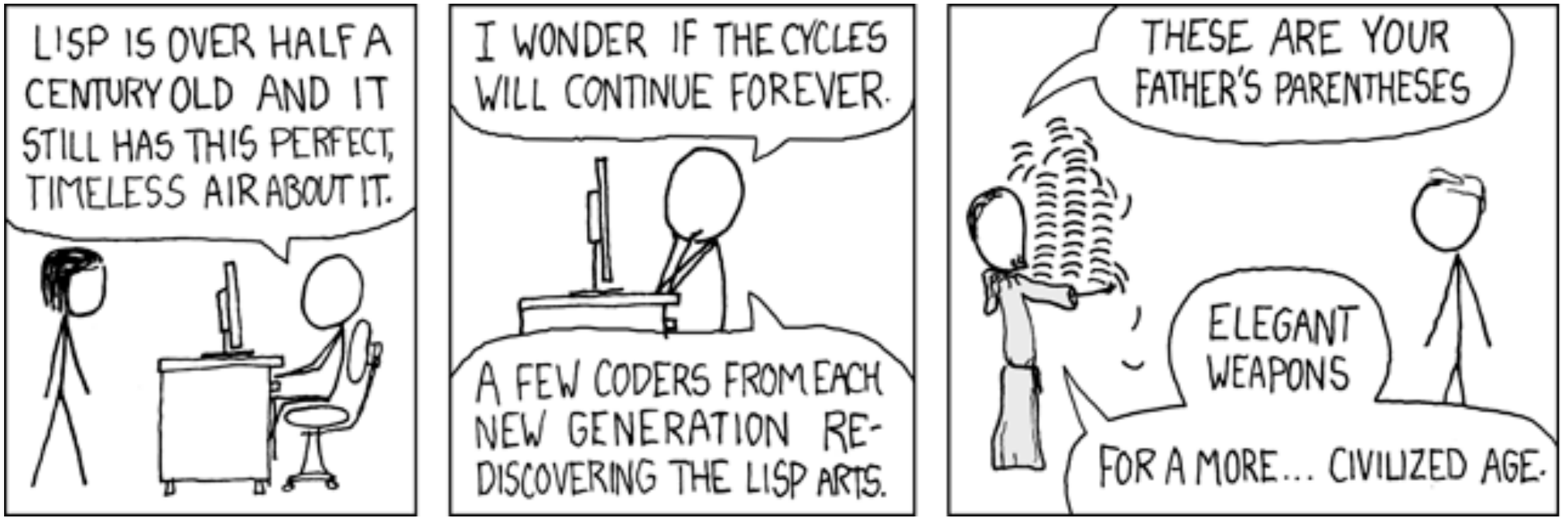
Paper to be presented at the First International LISP Conference, Mexico City, Mexico, December 30 - January 3, 1964.

This paper presents an algorithm for reclaiming unused free storage memory cells in LISP. It depends on availability of a fast secondary storage device, or a large block of available temporary storage. For this price, we get

1. Packing of free-storage into a solidly packed block.
2. Smooth packing of arbitrary linear blocks and arrays.
3. The collector will handle arbitrarily complex re-entrant list structure with no introduction of spurious copies.
4. The algorithm is quite efficient; the marking pass visits words at most twice and usually once, and the loading pass is linear.
5. The system is easily modified to allow for increase in size of already fixed consecutive blocks, provided one can afford to initiate a collection pass or use a modified array while waiting for such a pass to occur.

collect[z;n] is the function that finds all list-structure depending from z, and puts out on drum the

*Work reported herein was supported by MIT Project MAC and sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government.



[https://xkcd.com/297/]

ende