KONZEPTE SYSTEMNAHER PROGRAMMIERUNG

Technische Hochschule Mittelhessen

Andre Rein

– Speicherverwaltung –

AKTUELLE SPEICHERALLOKATION

- Die jetzige Speicherverwaltung der VM verwendet `malloc()` um Speicher zu allokieren — Bei dem Erzeugen von Objekten wird Speicher auf dem Heap des Programms reserviert und `malloc()` kümmert sich automatisch um dessen *Verwaltung*.
  - Verwaltung bedeutet: Reservierung einer bestimmten Anzahl an Bytes und Rückgabe eines gültigen Zeigers.

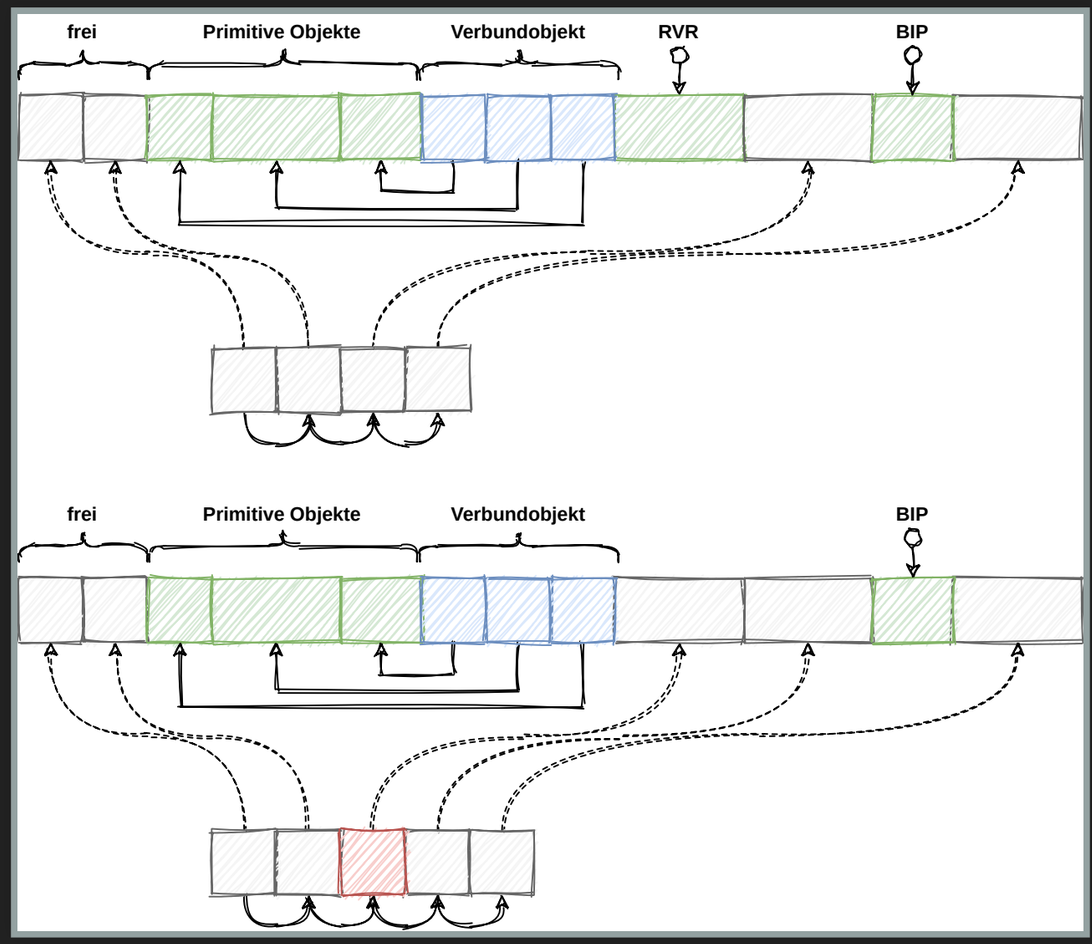
- Viele Objekte, die in der VM erzeugt und verwendet werden, sind *kurzlebige* Objekte.
  - Z.B.: **lokale Variablen** (Gültigkeitsdauer einer Funktionsaufrufs), Werte im **Rückgaberegister** (Gültigkeit endet nahezu direkt nach Funktionsaufruf), uvm.
- Die *kurzlebigen* Objekte werden aktuell nicht wieder freigegeben — Sie werden nur kurze Zeit benötigt, belegen aber Speicher über die komplette Programmlaufzeit.
  - Die VM allokiert also über den Zeitraum eines Programmdurchlaufs immer mehr und mehr Speicher.

SPEICHERVERWALTUNG: FREILISTE

Es gibt verschiedene Verfahren um Speicherverwaltung zu realisieren:

- **Freiliste**: Verwaltung einer Liste, in der alle freien Speicherbereiche verwaltet werden.
  - Vorwiegend verwendet bei Objekten konstanter Größe ansonsten sehr aufwendig (siehe `malloc()` in C).
- Aufwand: Verwaltung der Liste — inklusive Strategie welcher freie Bereich ausgewählt wird (*first-fit*, *best-fit*, *worst-fit*).

Freiliste: Speicherrückgabe RVR

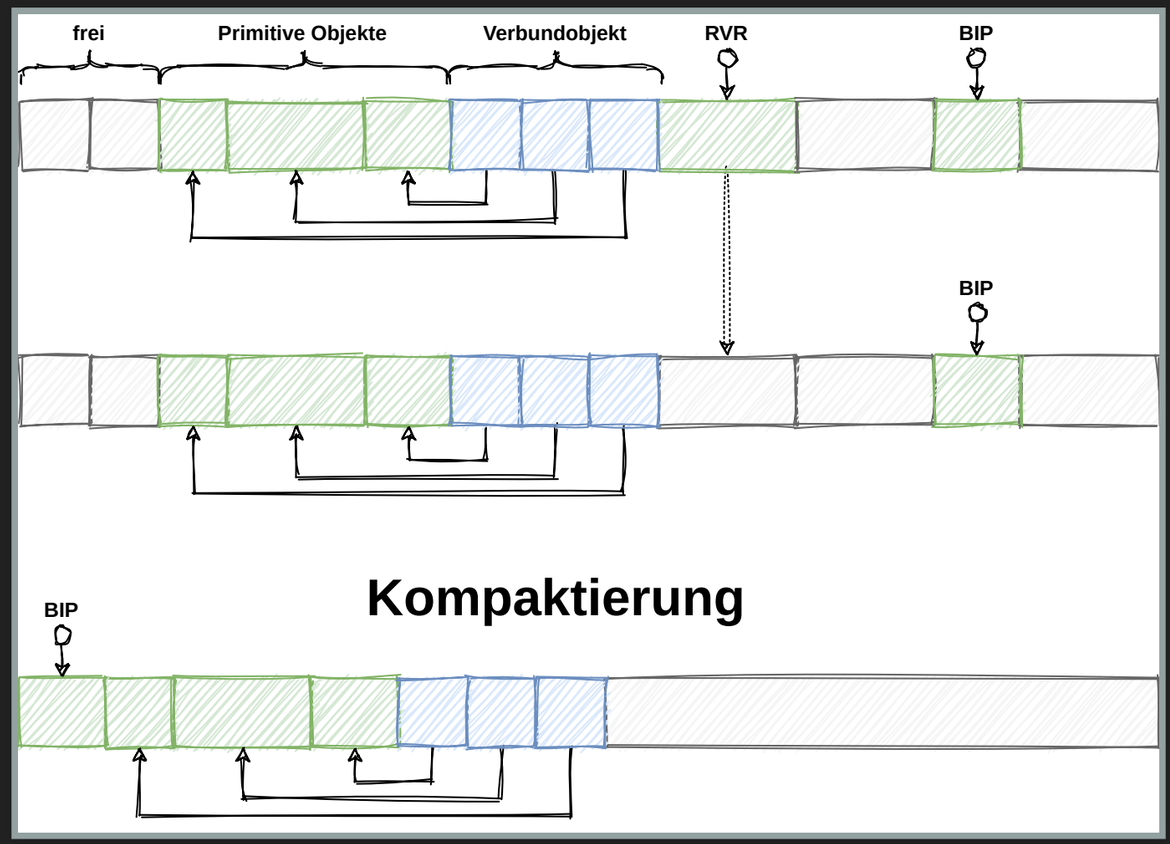


SPEICHERVERWALTUNG: FREIBEREICH

Es gibt verschiedene Verfahren um Speicherverwaltung zu realisieren:

- **Freibereich**: Verwaltung eines **zusammenhängenden** Bereichs von freiem Speicher
  - Auswahl von freiem Speichers sehr einfach — man nimmt den nächsten freien Platz.
  - Aufwand: Erfordert Speicherkompaktierung (zusammenlegen) von Speicherbereichen.

Freibereich: Speicherrückgabe RVR



Die Speicherkompaktierung kann ein *separater* Schritt sein oder (bevorzugt) direkt bei der *Freigabe* von Objekten erfolgen.

SPEICHERVERWALTUNG: SPEICHERFREIGABE

Prinzipiell kann die Speicherfreigabe **implizit** oder **explizit** erfolgen.

- **Explizite Speicherfreigabe** — muss selbstständig programmiert werden.
  - Beispiele: `C`, `C++`
- **Implizite Speicherfreigabe** — erfolgt automatisch, da Teil der Speicherverwaltung der Programmiersprache.
  - Beispiel: *Java*

	Explizite Speicherfreigabe birgt ein hohes Risiko zur falschen Benutzung (*dangling pointer problem)* — Verwendung von Zeigern die auf nicht (*mehr*) gültige Objekte im Speicher zeigen.

SPEICHERFREIGABE: NINJA

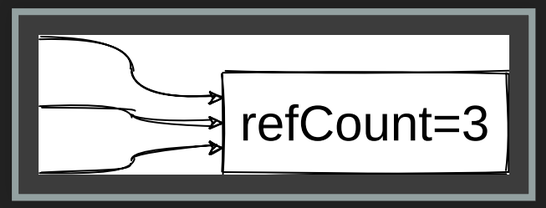
Die Speicherverwaltung und Freigabe von Speicher in Ninja selbst soll **implizit** erfolgen — also ähnlich wie in Java.

- Ähnlich wie in Java soll sich ein Programmierer, der in Ninja programmiert, nicht explizit um die Allokation und Freigabe von Speicher kümmern.

Das bedeutet, die **Ninja-VM** muss sich um die Allokation und Freigabe des Speichers kümmern!

METHODEN IMPLIZITER SPEICHERVERWALTUNG

Referenzzähler — Jedes Objekt beinhaltet einen Zähler, dessen Wert die Anzahl an Zeigern angibt, die auf ebendieses Objekt zeigen.



Fällt der Zähler auf `0`, kann der Speicher des Objekts freigegeben werden.

- **Vorteil**: Gleichmäßige Verteilung des zeitlichen Aufwands.
- **Nachteile**: 1. Zyklische Strukturen werden nicht erkannt und 2. der Aufwand ist relativ hoch, da bei jeder Zuweisung **die Zähler** aktualisiert werden müssen.

Vorher

```
// p1 und p2 sind Zeiger auf Objekte
p1=p2;
```

Nachher

```
p1->refCount--;
p1=p2;
p1->refCount++;
```

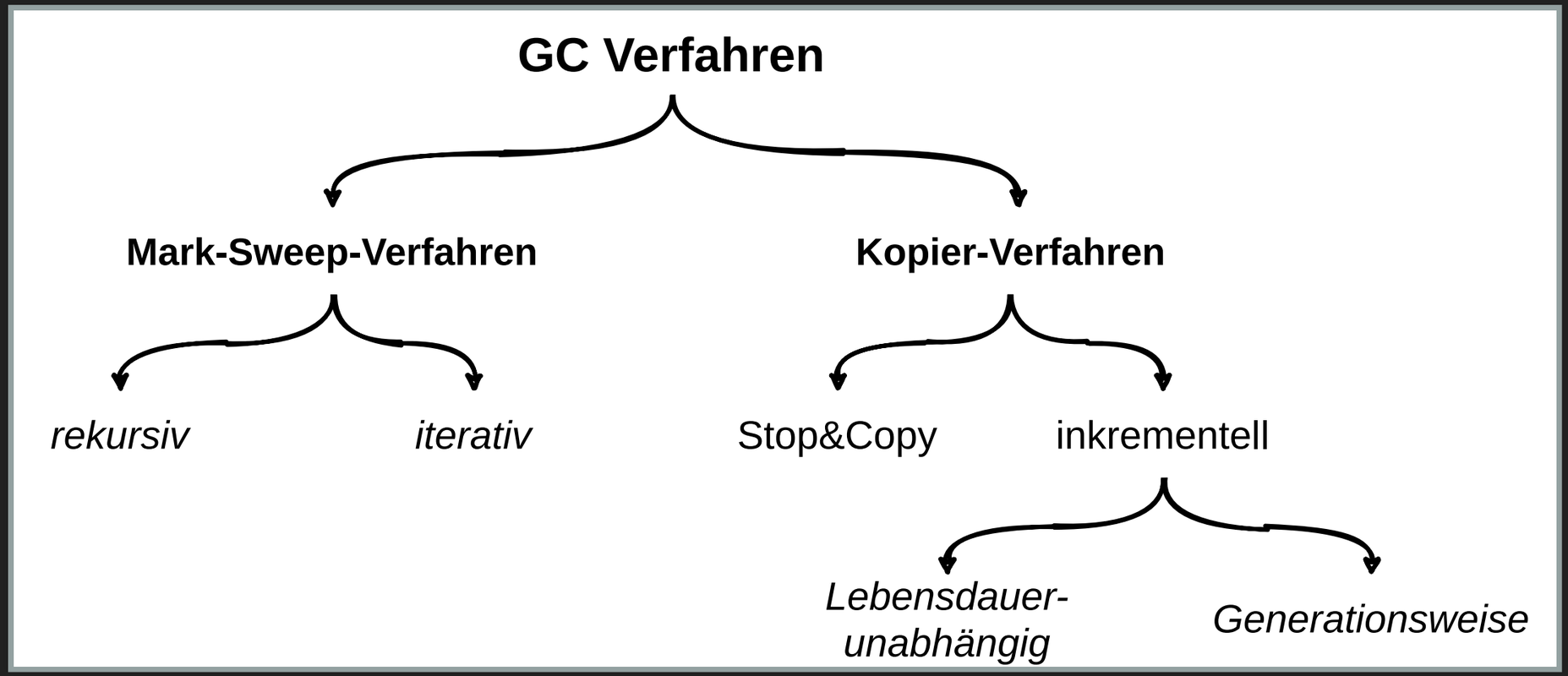
METHODEN IMPLIZITER SPEICHERVERWALTUNG

Müllsammeln (engl. Garbage Collection (**GC**)) — Zu einem bestimmten Zeitpunkt (z.B. wenn der freie Speicher knapp ist) wird der Speicher aller Objekte freigegeben, die nicht mehr erreichbar sind (*die nicht mehr lebendig sind*).

	Ausgehend von den **Registern**, der **SDA** und dem **Stack** der VM, werden alle erreichbaren Objekte identifiziert und alle anderen (nicht erreichbaren) Objekte freigegeben

- **Vorteile**: 1. Speicherverwaltung ist klar abgegrenzt vom Rest der Maschine und 2. zyklische Strukturen werden gesammelt.
- **Nachteile**: 1. Aufwand konzentriert sich auf bestimmte Zeitpunkte und 2. je nach Verfahren müssen die Berechnungen länger angehalten werden.

ÜBERSICHT GC-VERFAHREN



GC-VERFAHREN: MARK-SWEEP

Das Mark-Sweep-Verfahren ist ein nicht-kompaktierendes Verfahren und läuft in 2 Phasen (1. Mark, 2. Sweep) ab.

1. Phase **Mark**: Alle erreichbaren Objekte werden markiert — Z.B. durch das Setzen eines Flags im Objekt.
2. Phase **Sweep**: Durchgang durch **alle** Objekte. Der Speicherplatz von nicht markierten Objekten wird freigegeben. Weiterhin wird das Markierungsflag bei **allen** Objekten zurückgesetzt.

Die Mark-Phase kann *rekursiv* also sog. *Depth-First-Search* (**DFS**) oder *iterativ* sog. Iterative *Depth-First-Search* (**IDFS**) implementiert werden.

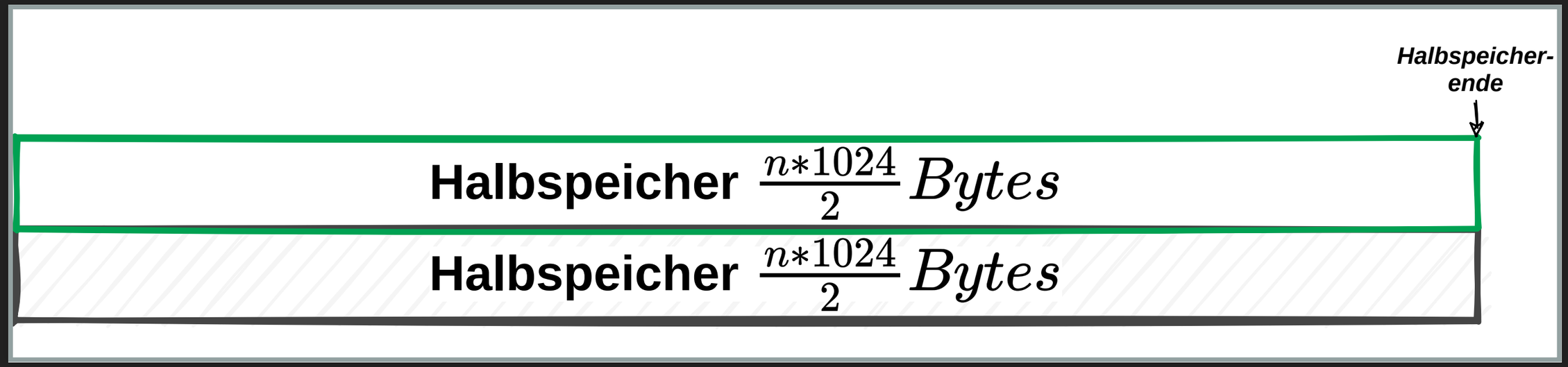
	Da Mark-Sweep-Verfahren nicht kompaktierend sind und sich eher für Strukturen konstanter Größe eigenen (da Speicherfragmentierung), wird ein anderes Verfahren in der Ninja-VM verwendet.

SPEICHERVERWALTUNG IN DER NINJA-VM

In der Ninja-VM wird das **Stop&Copy** GC-Verfahren eingesetzt werden. Hierbei handelt es sich um ein kompaktierendes Verfahren.

- **Vorteile**: 1. Einfach zu implementieren und 2. eignet sich sehr gut für Objekte mit unterschiedlichsten Größen.
- **Nachteil**: Alle Berechnungen müssen während des Kopiervorgangs vollständig angehalten werden.

SPEICHERVERWALTUNG IN DER NINJA-VM



- Im ersten Schritt wir der Zielspeicher angelegt. Dies erfolgt üblicherweise mit `malloc()` oder `calloc()`.
- Die Größe des Speichers soll in `KB` erfolgen, bei `n = 1` sollen also `1024` Byte belegt werden.
  - Der Aufruf wäre hierbei `malloc(n*1024)` oder `calloc(1, 1024)`.
- Dieser Speicher kann dann in 2 Halbspeicher aufgeteilt werden, die in diesem Beispiel jeweils `512 Byte` entsprechen würden.

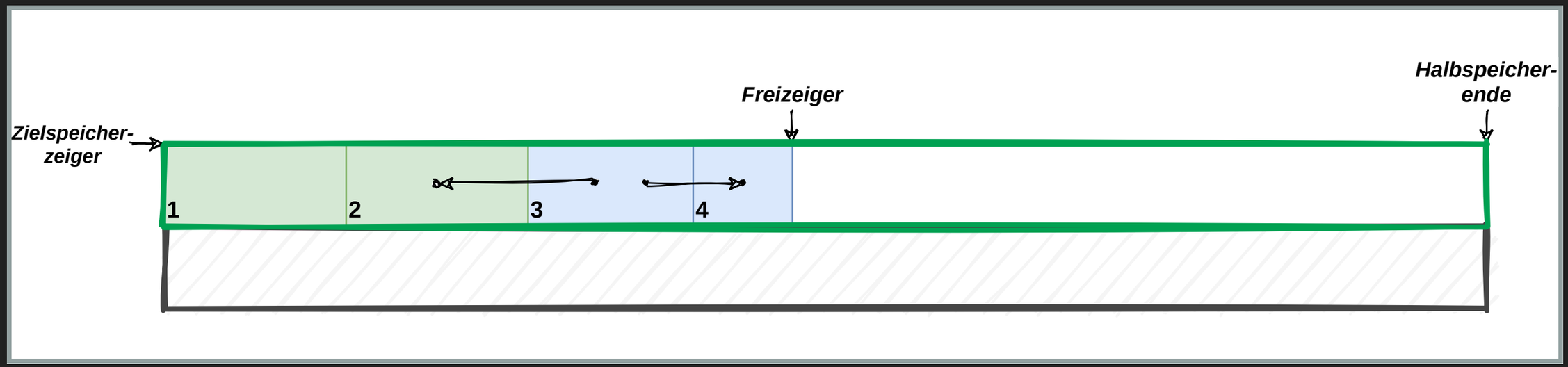
SPEICHERVERWALTUNG IN DER NINJA-VM

- Der erste Halbspeicher wird ausgewählt und wird zum aktiven Halbspeicher.
  - Die Speicherallokation erfolgt demnach ausschließlich in diesem Halbspeicher.
- Zusätzliche Verwaltungsstrukturen sind:
  - `Zielspeicherzeiger` — Zeigt auf den aktuell aktiven Zielspeicher.
  - `Freizeiger` — Zeigt die Speicherposition des nächsten Elements.
  - `Halbspeicherende` — Markiert das Ende des Halbspeichers ($Zielspeicherzeiger + \frac{n*1024}{2}$).

SPEICHERVERWALTUNG IN DER NINJA-VM

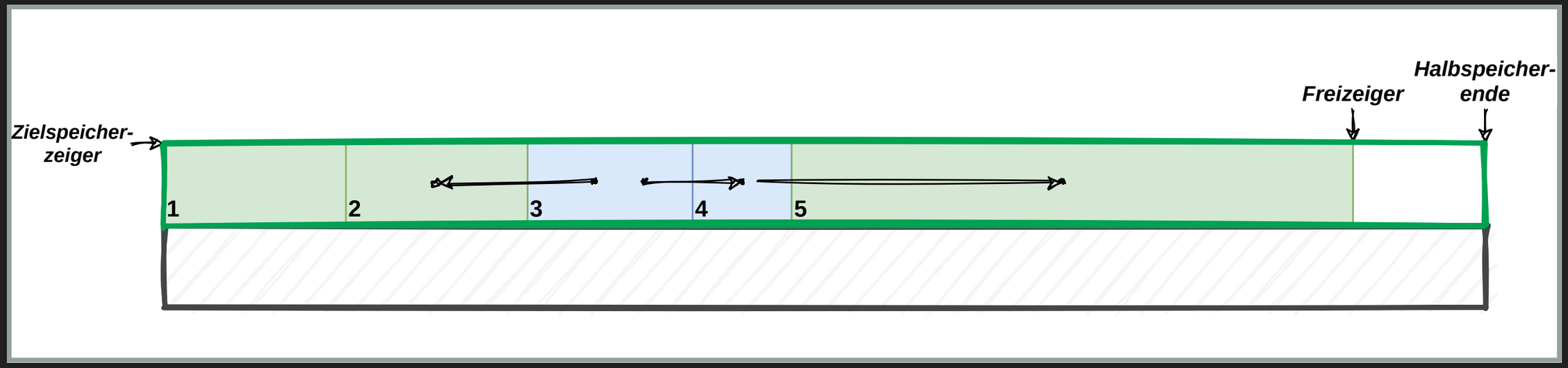
- Wenn ein Objekt Speicher benötigt, wird einfach der aktuelle `Freizeiger` verwendet (zurückgegeben), da dieser auf dieser auf den nächsten **freien** Speicherplatz zeigt.
  - Natürlich muss hierbei geprüft werden, das ein Objekt auch in den Speicher passt $- (Freizeiger + GrößeObjekt) < Halbspeicherende$.
- Passt das Objekt in den Speicher, wird der Freizeiger entsprechen verschoben — $Freizeiger + GrößeObjekt$

SPEICHERVERWALTUNG IN DER NINJA-VM



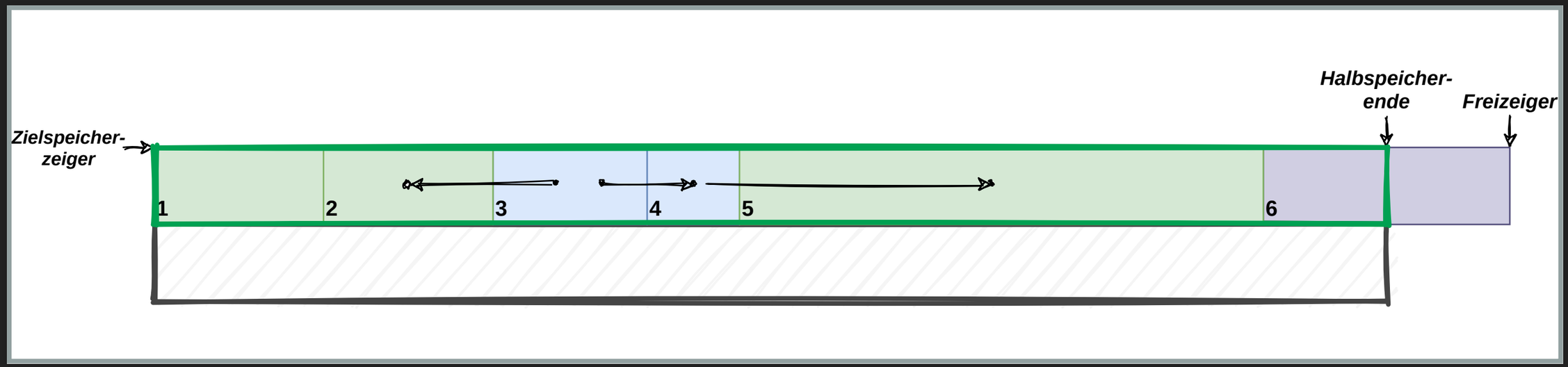
- So füllt sich sukzessive der Halbspeicher mit Objekten, die nacheinander angelegt werden.
  - Objekt 1 und 2 sind hierbei **primitive Objekte** und Objekt 3 und 4 sind **Verbundobjekte**.
- Hierbei muss darauf geachtet werden, die Objektgröße abhängig vom Objekttyp korrekt zu berechnen.

SPEICHERVERWALTUNG IN DER NINJA-VM



- Dies ist vorerst der Endzustand. Der Speicher wird langsam knapp.
- Es sind nun 5 Objekte im Speicher:
- Objekt 1, 2, 5 sind **primitive Objekte** und Objekt 3 und 4 sind **Verbundobjekte**.
  - Das *Verbundobjekt 3* zeigt außerdem auf das *primitive Objekt* 2 und das *Verbundobjekt 4*.
  - Das *Verbundobjekt 4* zeigt wiederum auf das *primitive Objekt* 5.

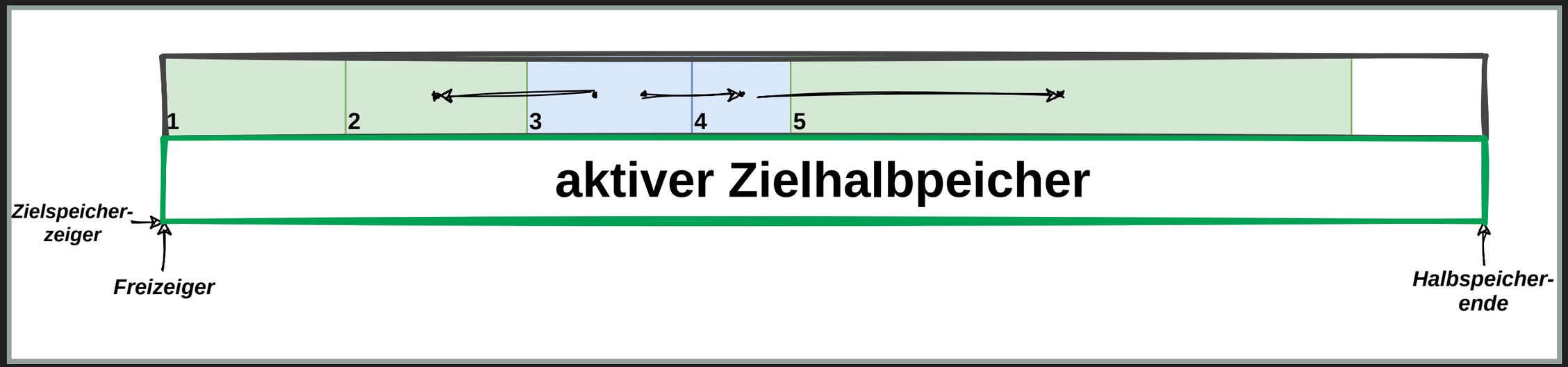
SPEICHERVERWALTUNG IN DER NINJA-VM



- Nun soll ein weiteres primitives Objekt 6 im Halbspeicher untergebracht werden
  — Offensichtlich passt diese Objekt aber nicht mehr in den Halbspeicher, da gilt:
  - $(Freizeiger + GrößeObjekt) > Halbspeicherende$

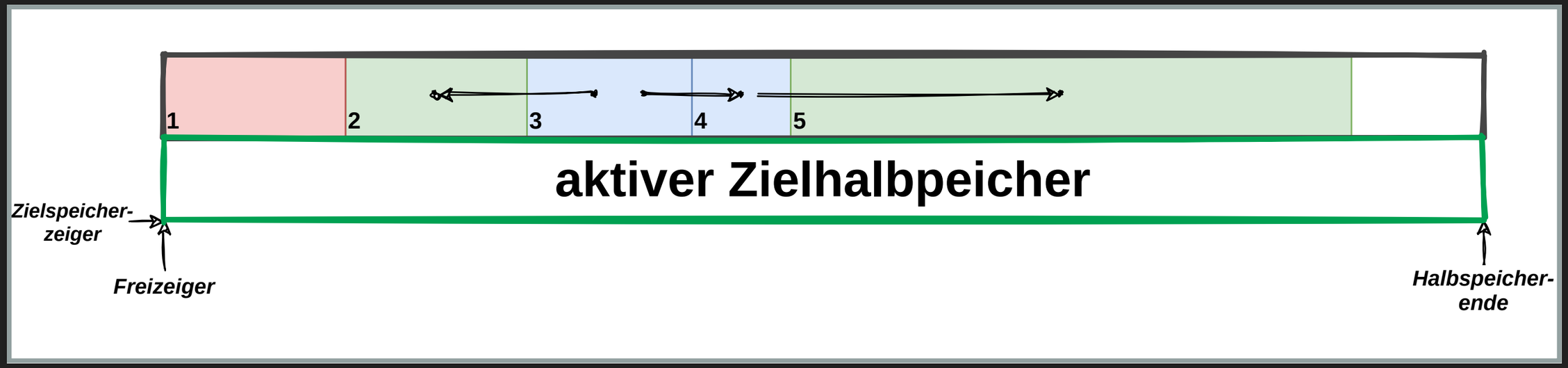
Nun muss zuerst Platz für das Objekt geschaffen werden und es kommt zum ersten Aufruf des **Garbage Collectors** (GC).

SPEICHERVERWALTUNG IN DER NINJA-VM



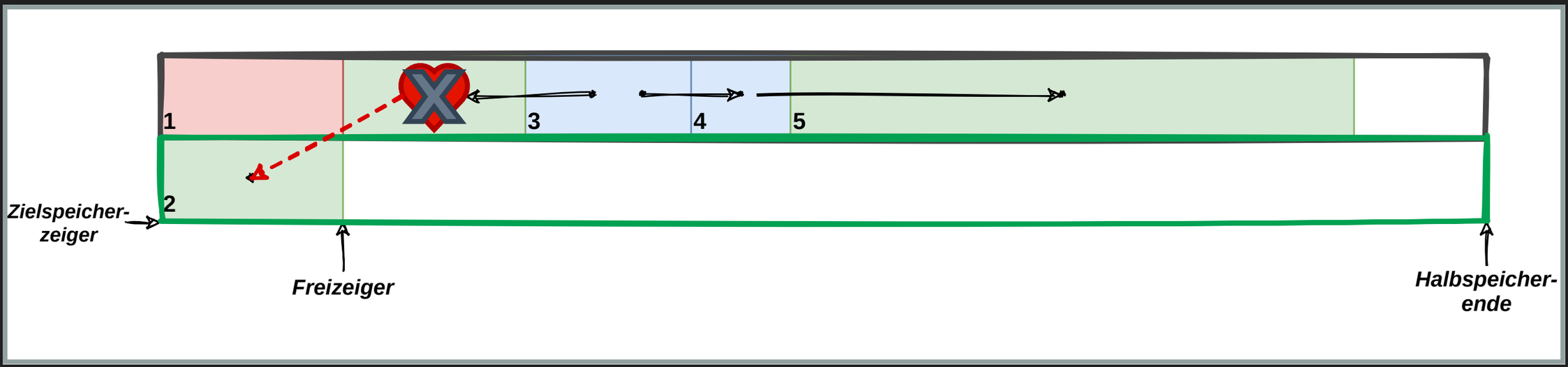
- Es kommt zum **Halbspeicherwechsel**. Der bisher aktive Halbspeicher wird der neue passive Speicher und der bisher passive Speicher wird der neue aktive.
- Hierzu müssen die Verwaltungsstrukturen entsprechend angepasst werden.
  - `Zielspeicherzeiger` und `Freizeiger` — Können auf **altes** Halbspeicherende gesetzt werden.
  - `Halbspeicherende` — $aktueller Zielspeicherzeiger + \frac{n*1024}{2}$.

SPEICHERVERWALTUNG IN DER NINJA-VM



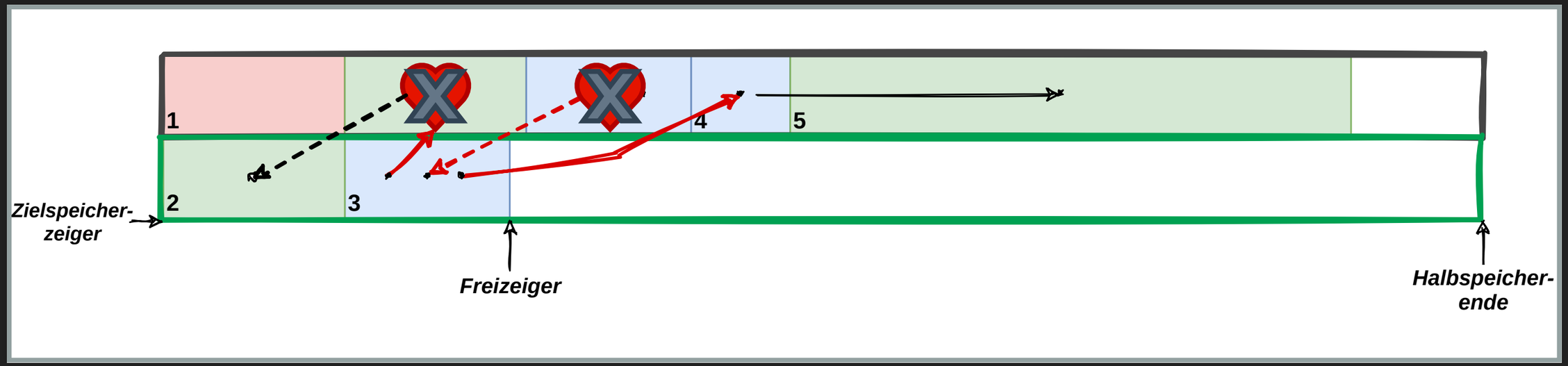
- Es folgt die Phase in der alle sog. Root-Objekte (Wurzelobjekte) kopiert werden — das sind die Objekte, die **direkt** von einem Register, dem BIP, einem Stackslot oder von Variablen in der SDA referenziert werden.
- Im aktuellen Beispiel sind die Root-Objekte `2` und `3`.
- **Objekt 1** wird weder von einem Root-Objekt, noch von einem Verbundobjekt referenziert und wird somit nicht weiter beachtet.
- Das erste Objekt was nun kopiert wird, ist das *primitive Objekt* `2`

SPEICHERVERWALTUNG IN DER NINJA-VM



- Die Kopie eines Objekts erfolgt über die Kopie des Speicherinhaltes vom passiven in den aktiven Halbspeicher — typischerweise wird hierfür die Funktion `void *memcpy(void *dest, const void *src, size_t n);` verwendet.
- Weiterhin wird im "alten" Objekt das sog. `brokenheart_flag` gesetzt, dies signalisiert, dass dieses Objekt bereits kopiert wurde (*und somit nicht mehr gültig ist*)
- Zusätzlich muss im alten Objekt der neue Speicherplatz im aktiven Halbspeicher hinterlegt werden — Dies erfolgt im sog. `forward_pointer`.
- Letztendlich wird der Freizeiger angepasst, um auf die nächste freie Speicherposition zu zeigen.

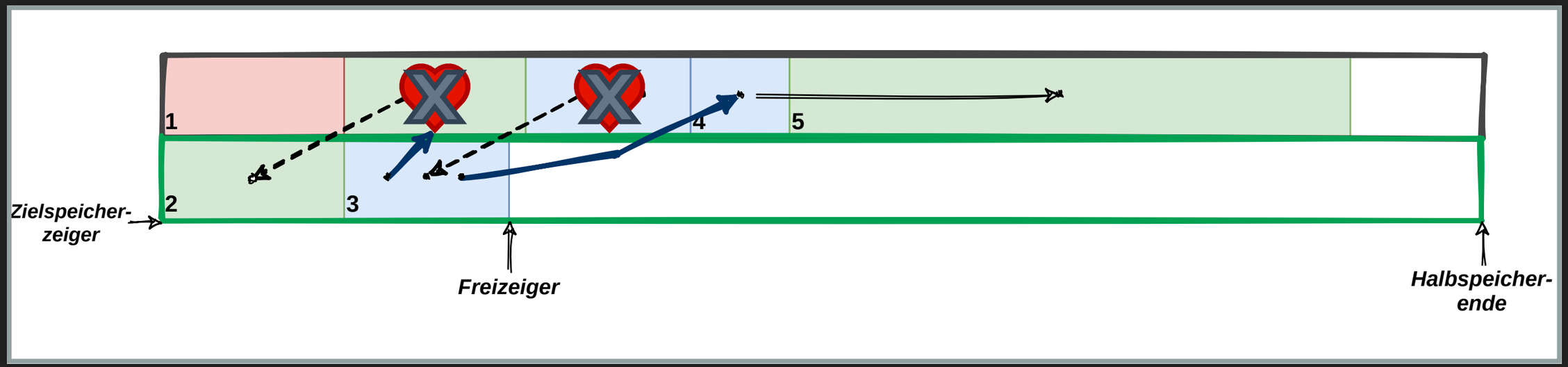
SPEICHERVERWALTUNG IN DER NINJA-VM



- Die Kopie von primitiven Objekten und Verbundobjekten ist äquivalent —
  1. Die Kopie des Objektes erfolgt mit `memcpy`.
  2. Das `brokenheart_flag` und der `forward_pointer` werden im "alten" Objekt entsprechend gesetzt.
  3. Der Freizeiger wird angepasst und zeigt auf die nächste freie Speicherposition.

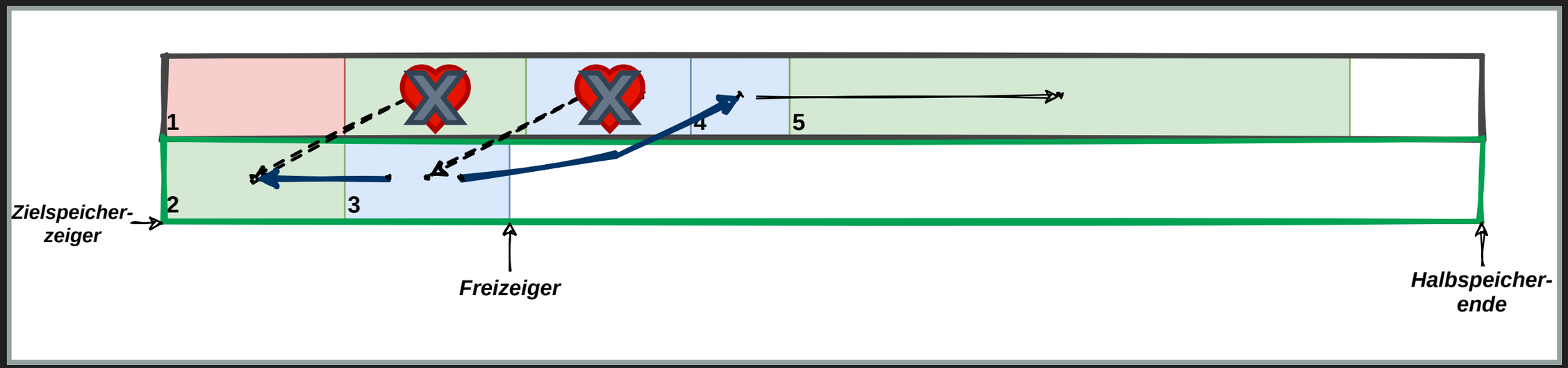
	Die Zeiger im Verbundobjekt selbst zeigen noch auf den "alten" Halbspeicher — Die Anpassung erfolgt in der sog. **Scan**-Phase.

SPEICHERVERWALTUNG IN DER NINJA-VM



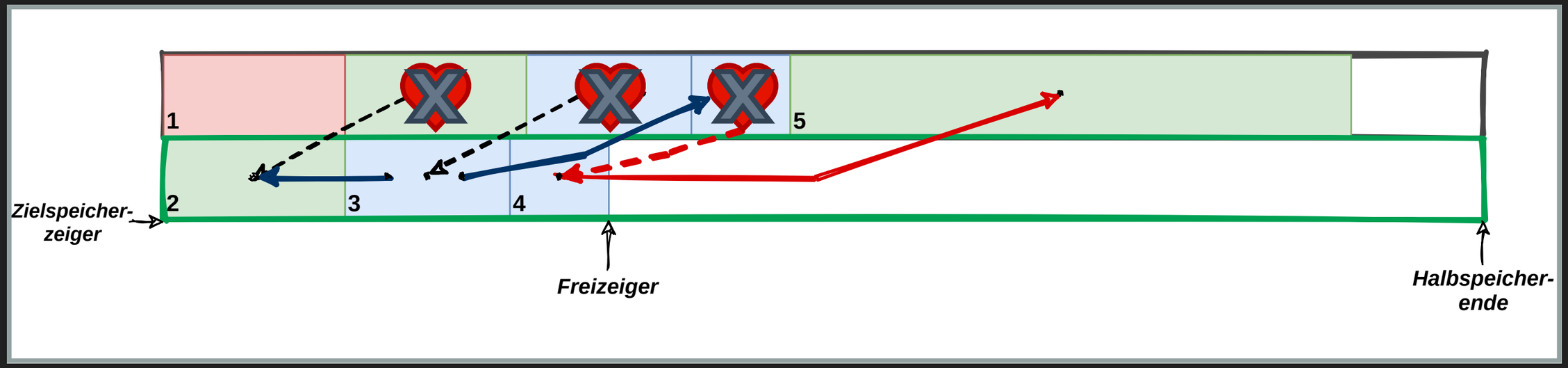
- Nachdem die Root-Objekte kopiert wurden, folgt die sog. **Scan**-Phase.
- Man durchläuft hierbei sukzessive den **aktiven** Halbspeicher und kopiert alle erreichbaren Elemente (ausgehend von den Root-Objekten).
- Bereits kopierte primitive Objekte werden bei dem Scan-Durchlauf übersprungen, da diese Objekte auf keine weiteren Objekte zeigen können.
- Verbundobjekte müssen im Gegensatz dazu im Detail betrachtet werden — konkret die Referenz auf das "*alte und kopierte*" Objekt `2` und das "*noch nicht kopierte*" Objekt `4`.

SPEICHERVERWALTUNG IN DER NINJA-VM



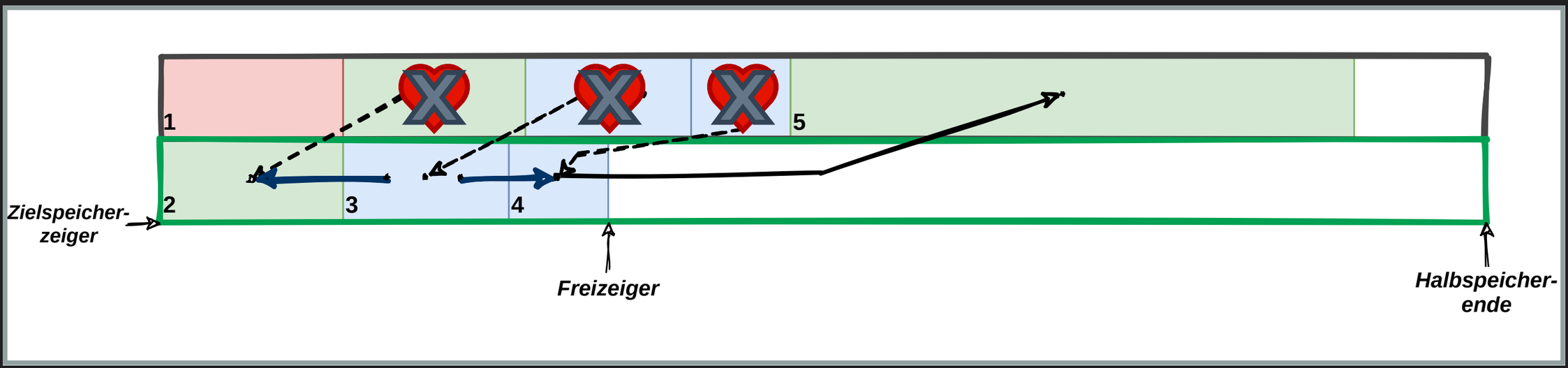
- Die erste Anpassung erfolgt für die Referenz auf das "alte Objekt 2".
- Wenn man dem Zeiger auf das "alte Objekt 2" folgt, stellt man fest, dass in diesem Objekt das `brokenheart_flag` gesetzt ist.
  - Dies bedeutet, dass das Objekt bereits kopiert ist und ein gültiger `forward_pointer` auf den aktiven Halbspeicher verfügbar ist.
- Der `forward_pointer` des alten Objektes kann somit direkt verwendet werden und wird im Verbundobjekt eingetragen.

SPEICHERVERWALTUNG IN DER NINJA-VM



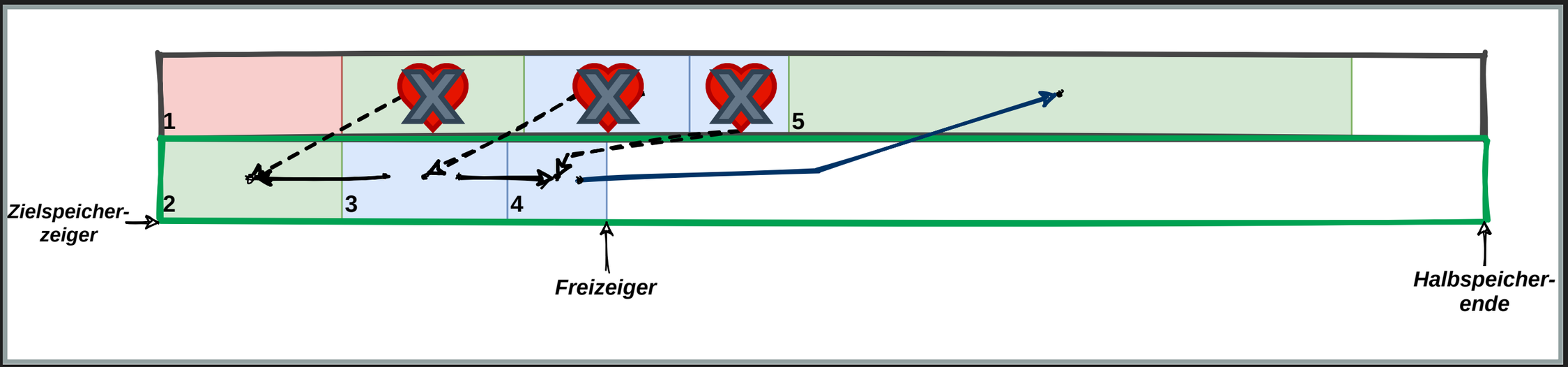
- Das Objekt 4 wurde noch nicht kopiert — Dies erkennt man daran, da im Objekt das `brokenheart_flag` nicht gesetzt ist.
- Somit muss das Objekt 4 zuerst in den aktiven Halbspeicher kopiert werden — der Ablauf ist abermals:
  1. Die Kopie des Objektes erfolgt mit `memcpy`.
  2. Das `brokenheart_flag` und der `forward_pointer` werden im "alten" Objekt entsprechend gesetzt.
  3. Der Freizeiger wird angepasst und zeigt auf die nächste freie Speicherposition.

SPEICHERVERWALTUNG IN DER NINJA-VM



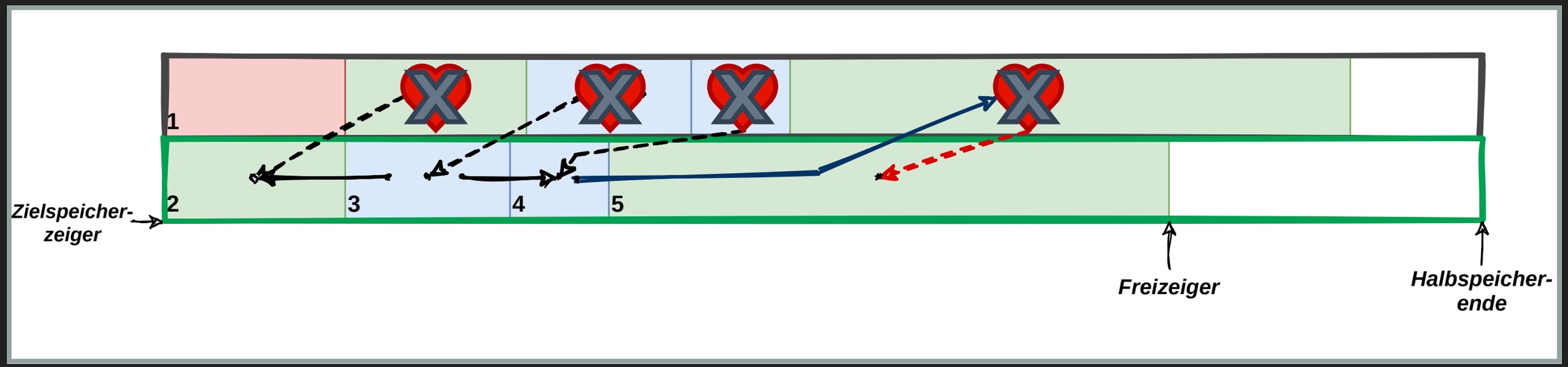
- Nun existiert im alten Objekt auch eine gültiger `forward_pointer` und das `brokenheart_flag` ist gesetzt.
- Objekt 3 wird nun angepasst, indem die Referenz im Objekt 3 auf den neuen Zeiger gesetzt wird, der dem `forward_pointer` im alten Objekt 4 entspricht und nun auf das kopierte Objekt 4 im aktiven Hauptspeicher verweist.

SPEICHERVERWALTUNG IN DER NINJA-VM



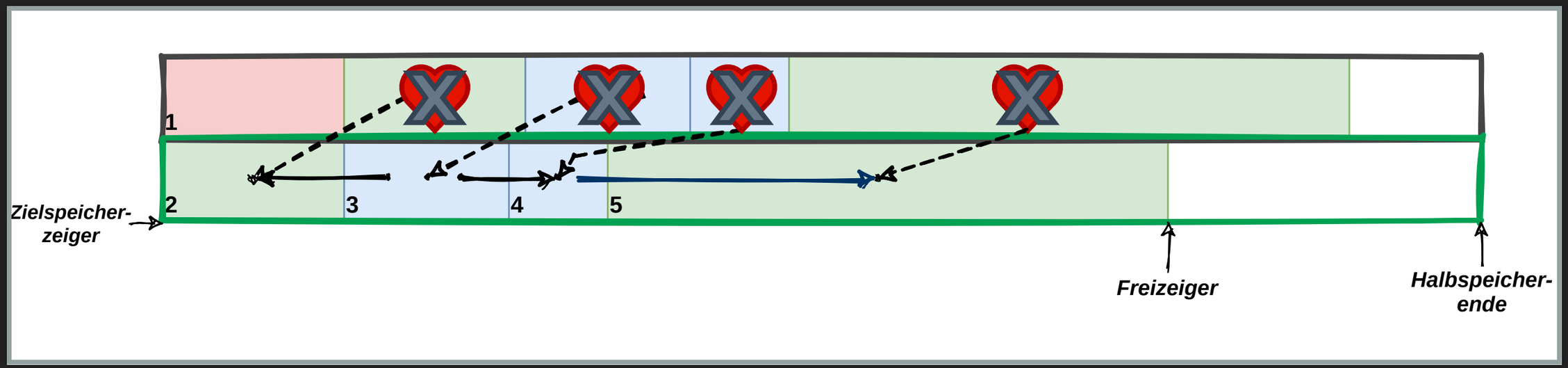
- Nun wird der Vorgang für das Verbundobjekt 4 wiederholt, d.h. alle vom Objekt referenzierten andere Objekte werden bei Bedarf erst kopiert und deren Zeiger angepasst.
- In diesem Fall gibt es noch eine Referenz auf das Objekt 5, welches noch nicht kopiert wurde.

SPEICHERVERWALTUNG IN DER NINJA-VM



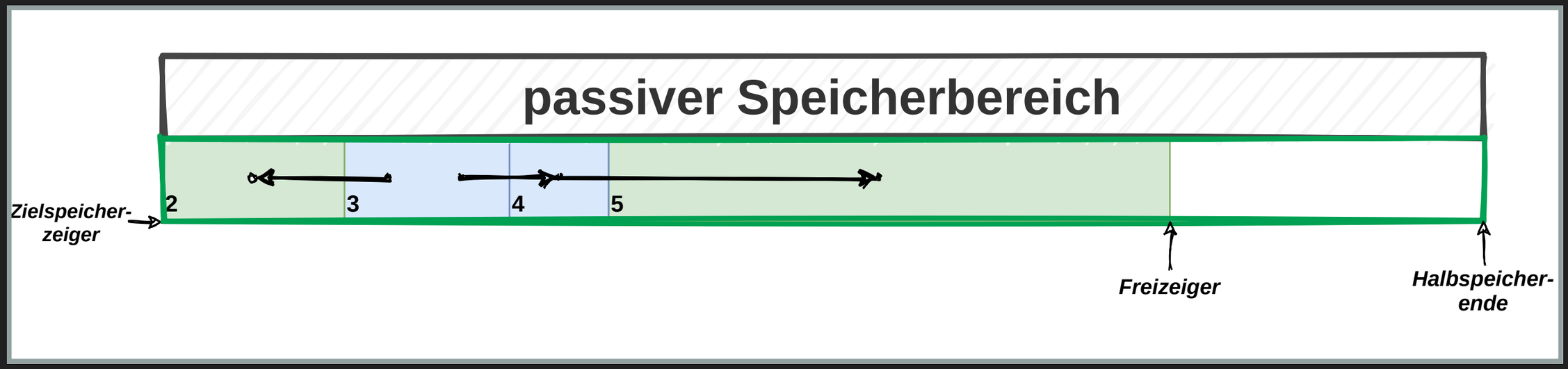
- Es folgt die Kopie von Objekt 5 in den aktiven Halbspeicher:
  1. Die Kopie des Objektes erfolgt mit `memcpy`.
  2. Das `brokenheart_flag` und der `forward_pointer` werden im "alten" Objekt entsprechend gesetzt.
  3. Der Freizeiger wird angepasst und zeigt auf die nächste freie Speicherposition.

SPEICHERVERWALTUNG IN DER NINJA-VM



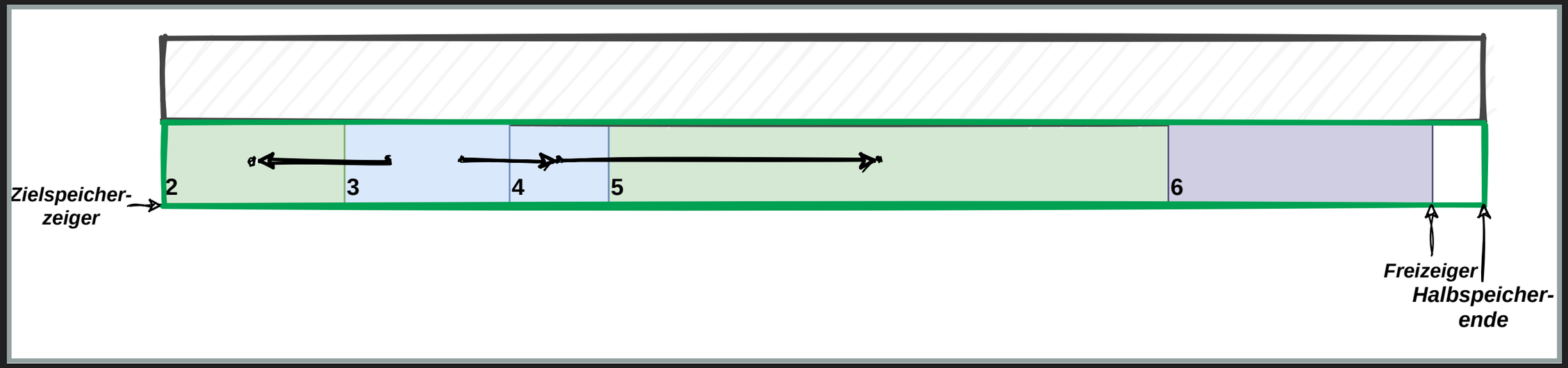
- Nun existiert im alten Objekt auch eine gültiger `forward_pointer` und das `brokenheart_flag` ist gesetzt.
- Objekt 3 wird nun angepasst, indem die Referenz im Objekt 3 auf den neuen Zeiger gesetzt wird, der dem `forward_pointer` im alten Objekt 4 entspricht und nun auf das kopierte Objekt 4 im aktiven Hauptspeicher verweist.
- Nach dieser finalen Anpassung ist der **Scan**-Vorgang vollständig abgeschlossen.

SPEICHERVERWALTUNG IN DER NINJA-VM



- Nach dem Scan-Vorgang sind alle erreichbaren Objekte kopiert. Somit kann der passiver Halbspeicher als nicht weiter relevant betrachtet werden.
- Damit ist der Durchlauf des GC vollständig abgeschlossen und neue Objekte können nun im aktiven Speicherbereich allokiert werden.

SPEICHERVERWALTUNG IN DER NINJA-VM



- Es wird nun erneut versucht Speicher für das primitive Objekt 6 zu allokieren.
- Da ausreichend Speicherplatz zur Verfügung steht, kann der Speicher allokiert werden und das Objekt im aktiven Halbspeicher untergebracht werden.

	Sollte nach dem Durchlauf des GC nicht ausreichend Speicherplatz vorhanden sein, dann muss das Programm mit einer Fehlermeldung abgebrochen werden, da nicht ausreichend Speicherplatz vorhanden ist.

SPEICHERVERWALTUNG: RELOKATION UND SCANPHASE

Relokation eines Objektes

```
ObjRef relocate(ObjRef orig){
    ObjRef copy;
    if (orig == NULL) {
        /* relocate(nil) = nil */
        copy = NULL;
    } else {
        if(orig->brokenHeart){
            /* Objekt ist bereits kopiert */
            copy=orig->forwardPointer;
        } else {
            /* Objekt muss noch kopiert werden*/
            copy = copyObjectToFreeMem(orig);
            orig->brokenHeart = TRUE;
            orig->forwardPointer = copy;
        }
    }
    return copy;
}
```

Scanphase des Garbage Collectors

```
void scan(void){
    scan = Zielspeicherzeiger;
    while (scan < freizeiger) {
        /* es gibt noch Objekte, die
         * gescannt werden müssen.
         */
        if (Objekt enthält Zeiger){
            /* Ist ein Verbundobjekt */
            for (jeder Zeiger q, auf das scan zeigt){
                scan->q = relocate(scan->q);
            }
        }
    scan += Größe des Objektes, auf das scan zeigt
    }
}
```

Es handelt sich absichtlich um Pseudocode, der bei korrekter Implementierung auch zuverlässig und korrekt arbeitet.

SPEICHERVERWALTUNG: DATENSTRUKTUREN

Die Implementierung des `brokenheart_flag` und des Forwardpointers, kann wieder verschieden erfolgen.

- Man kann das 2 höchste Bit der Komponente `ObjRef->size` verwenden um das `brokenheart_flag` abzubilden.
  - Dies ist sehr ähnlich zum beschriebenen Verfahren für die Unterscheidung zwischen primitiven Objekten und Verbundobjekten.
- Weiterhin kann der `forward_pointer` ebenfalls in der `ObjRef->size`-Komponente untergebracht werden.
  - Wenn ein Objekt bereits kopiert wurde, ist die Größe des Verbundobjektes nicht mehr relevant!

SPEICHERVERWALTUNG: DATENSTRUKTUREN

Alternativ können zur Verwaltung auch wieder eigene Komponenten in `ObjRef` angelegt werden.

```
typedef struct {
    void * forward_pointer;
    unsigned int size;
    bool brokenHeart;
    unsigned char data[1];
} *ObjRef;
```

- Verwendung einer neuen Komponente `ObjRef->forward_pointer` und `ObjRef->brokenHeart`.

SPEICHERVERWALTUNG: SONSTIGER CODE

- Zusätzlich benötigen Sie noch Code für folgende Funktionen:
  - Anlegen des Speichers und Aufteilung in 2 Halbspeicher.
  - Allokieren von Speicher im aktiven Speicher — inklusive Prüfung auf Überlauf und auslösen des GC.
  - Wechsel der Halbspeicher (passiv → aktiv, aktiv → passiv).
  - Durchlaufen und kopieren aller Root-Objekte in den aktiven Speicher.
  - Kopieren von Objekten vom passiven in den aktiven Speicher.

WEITERE GC-VARIANTEN UND VERBESSERUNGEN

Idee: Verteilung der Rechenzeit des Stop&Kopie-Algorithmus.

- Baker, 1978 (List processing in real time on a serial computer):
  - Bei jeder Speicherallokation wird nur ein kleiner Teil der Scan-Phase abgearbeitet. Hierdurch kann die Rechenzeit besser verteilt werden.
- Lieberman und Hewitt, 1980 (A Real-Time Garbage Collector Based on the Lifetimes of Objects):
  - Betrachtet zusätzlich zu Baker's Algorithmus die Lebenszeit von Objekten — Es wurde beobachtet, dass viele Objekte nur sehr kurzlebig sind → diese werden öfter "entmüllt".
    - Je öfter ein Objekt einen GC-Durchlauf "übersteht", wandert es in Speicherregionen, die weniger oft entmüllt werden.
  - Hierdurch reduziert sich letztendlich primär der Kopieraufwand

ABSCHLUSS

Alle Inhalte, die zur vollständigen Bearbeitung aller Praktikumsaufgaben nötig waren, sind nun abgeschlossen. Sie können nun alle Aufgaben und somit auch die Hausübung 2 vollständig fertigstellen.

Inhaltlich sind wir fast am Ende des Semester. Es fehlen noch ein paar wenige kleinere **klausurrelevante** Themen, die bis zum Ende des Semesters noch in Videos bereitgestellt werden.