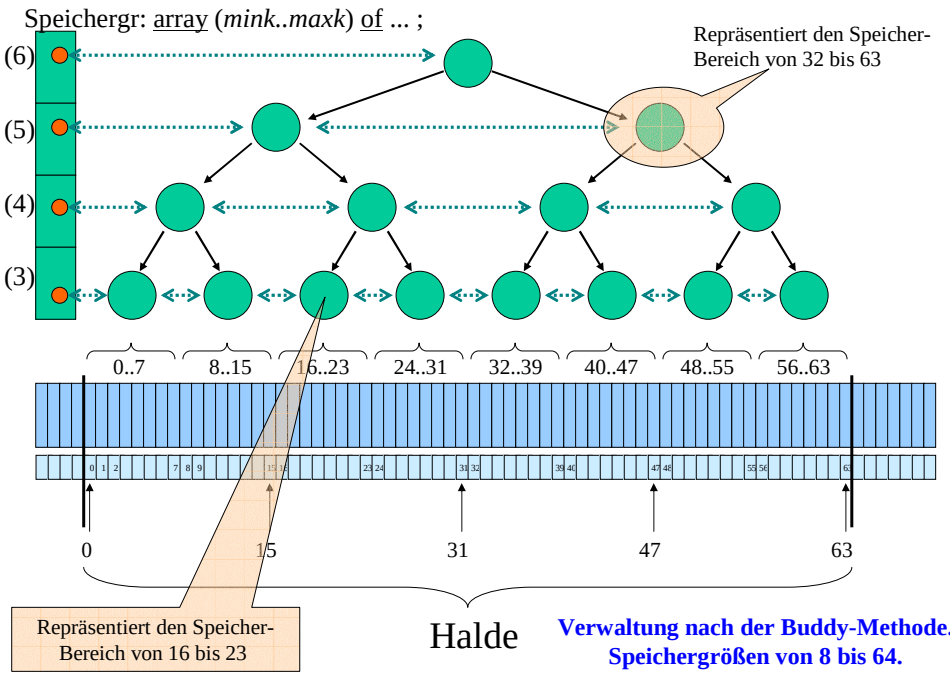
3.1.3.4 Buddy-Verfahren

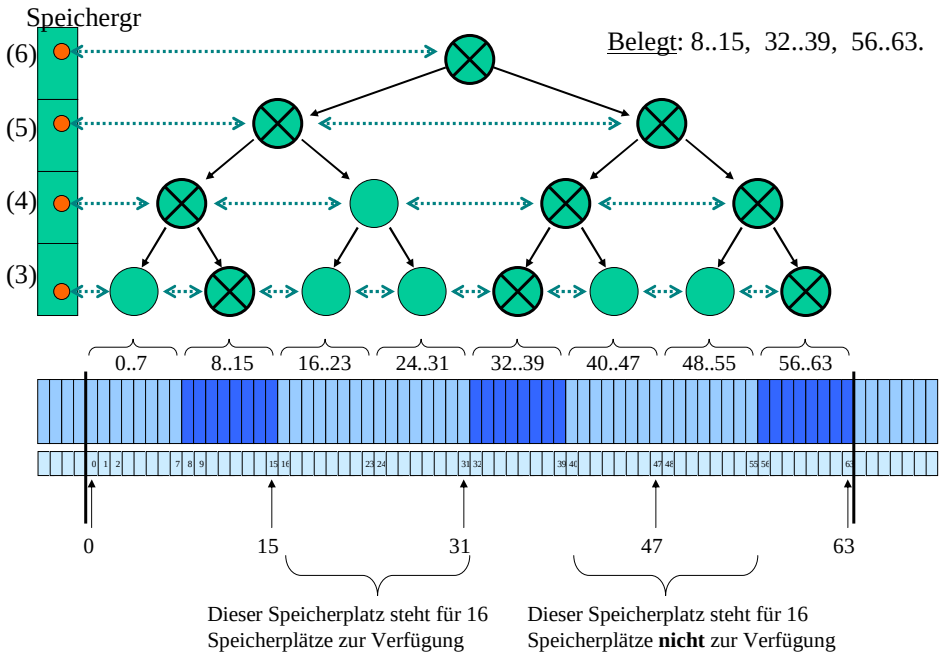
Wir stellen kurz einen alten Vorschlag zur Verwaltung des freien Speicherplatzes vor, der leicht implementiert werden kann, aber gewisse Nachteile besitzt, die sog. Buddy-Methode. (Buddy = (engl.) Kamerad, Kumpel.)

Idee: Das Verfahren legt einen gleichverzweigten binären Baum über den Speicher (also über die Halde). Das Aufspalten und das Verschmelzen sind aber nicht beliebig möglich.

Vorgehen: Die Halde wird in Datenblöcke unterteilt, deren Länge jeweils eine Zweierpotenz ist. Jeder Datenblock muss genau 2^k Speicherplätze belegen für eine natürliche Zahl k mit $mink \leq k \leq maxk$. (Man schränkt in der Praxis k ein z.B. zwischen $mink = 9$ und $maxk = 20$.) Jedem Datenblock wird genau einer seiner beiden Nachbarn als "Buddy" zugeordnet.



Verwaltung nach der Buddy-Methode. Speichergrößen von 8 bis 64.



Wir nummerieren die Halde also von 0 bis $2^{maxk} - 1$ durch. Die kleinste Blockgröße sei 2^{mink}. Alle Speicherblöcke mit festem $mink \le k \le maxk$ stehen in einer Liste, erreichbar über den Zeiger des Feldelements Speichergr(k).

Zu jedem Speicherblock, der an der Adresse x beginnt und die Größe 2^k besitzt, sei **$buddy_k(x)$** die Anfangsadresse seines Buddy (dieser liegt entweder links oder rechts von ihm und besitzt die gleiche Größe).

Es gilt:

$$\textbf{buddy}_k(x) = \begin{cases} x + 2^k, & \text{falls } x = 0 \mod 2^{k+1} \\ x - 2^k, & \text{falls } x = 2^k \mod 2^{k+1} \end{cases}$$

Wir programmieren die Freispeicherverwaltung nicht aus, sondern geben nur die Vorgehensweisen an.

Beginn: Anfangs werden alle Speicherbereiche als frei markiert.

Speicheranforderung: Ein Programm fordert einen Datenblock der Größe m an. Berechne k, so dass $2^{k-1} < m \leq 2^k$ gilt.

Suche: Die Speicherverwaltung durchläuft dann die Liste, die über Speichergr(k) erreichbar ist. Diese Liste hat 2^{maxk-k} Knoten.

Zuordnung und Aktualisierung: Wird hier ein freier Speicherbereich gefunden, so wird er dem Programm zugewiesen; zugleich werden dieser Bereich, alle Knoten in seinem Unterbaum und seine Vorgänger im Baum bis zur Wurzel als belegt markiert.

Ablehnung und "Wiedervorlage": Wird kein freier Speicherbereich gefunden, so lege die Speicheranforderung in einer Warteschlange des Systems ab, sende dem Programm einen "Wartehinweis" und prüfe später erneut.

Beispiel: Wird in der Situation der obigen Folie ein Bereich der Größe 14 angefordert, so ist k = 4 und es wird ausgehend von Speichergr(4) die Liste der Speicherblöcke der Größe $2^4 = 16$ durchsucht. Bereits der zweite Block ist frei, so dass dem anfordernden Programm der Bereich 16..31 zugewiesen wird.

Speicherfreigabe: Ein Programm gibt einen Datenblock der Größe 2^k wieder zurück. Dieser Block wird in der Liste zu Speichergr(k) als frei markiert. Ist sein Buddy frei, so wiederhole diesen Vorgang mit seinem Vorgängerknoten.

Beispiel: Wird in der Situation der obigen Folie der Bereich 8..15 der Größe 8 freigegeben, so kann dieser Block, aber auch sein Vorgänger und dessen Vorgänger frei gegeben werden, so dass anschließend die linken drei als belegt markierten Knoten im Baum wieder als frei markiert sind.

Vorteile der Buddy-Methode:

- Einfach zu handhaben und leicht zu programmieren.
- Das Verfahren bewährt sich in der Praxis hinreichend gut.

Nachteile:

- Benachbarte Bereiche, die nicht Buddys sind, können nicht verschmolzen werden, und es gibt nicht genutzte Speicherbereiche (Fragmentierung), da immer nur Blöcke von der Länge einer Zweierpotenz zugewiesen werden können.
- Es können Verklemmungen entstehen, wenn zwei Programme Speicher nachfordern, die nicht verfügbar sind. (Man kann dies durch Überwachung durch das Betriebssystem abfangen, oder man kann verbieten, dass ein Programm eine zweite Speicheranforderung stellt, was aber bei rekursiven oder nebenläufigen Programmen nicht sinnvoll ist.)

3.1.3.5 Speicherbereinigung (garbage collection)

Wir nehmen nun an, die Programme legen immer mehr dynamische Datenstrukturen (also: verzeigerte Strukturen) in der Halde an. Es ist absehbar, dass in Kürze kein Speicherplatz mehr zur Verfügung steht.

Nun muss geprüft werden, ob die Datenobjekte, die in der Halde stehen, wirklich alle benötigt werden oder ob man sie löschen und auf diese Weise neuen Speicherplatz bereitstellen kann. Dies ist die Aufgabe der "Speicherbereinigung", die in der Regel automatisch erfolgt.

Standardtechniken zur Lösung dieser Aufgabe sind: Referenzzählung, Graphdurchlauf und Kopieren.

3.1.3.6 Verweiszählermethode ("Reference Counting")

Wenn die Zeigerstrukturen keine Kreise bilden (also "azyklisch" sind), dann kann man in jeden Knoten einen "Verweiszähler" aufnehmen, der angibt, wie oft auf dieses Objekt verwiesen wird. Wird ein Knoten mit k Zeigern hinzugefügt, so müssen die Verweiszähler der k Knoten, auf die diese Zeiger zeigen, jeweils um 1 erhöht werden. Wird ein Knoten gelöscht, so muss man die Verweiszähler in den k Objekten, auf die die Zeiger zeigten, um jeweils 1 erniedrigen. Wird ein Verweiszähler hierbei 0, dann muss man auch in allen ihren nachfolgenden Knoten den Verweiszähler um 1 erniedrigen. Entsprechende Operationen kann der Compiler in den übersetzten Code einfügen, ohne dass der Programmierer hiervon etwas bemerkt. Benötigt man irgendwann Speicherplatz, so kann man zu einem gegebenen Zeitpunkt genau alle die Knoten löschen, deren Verweiszähler 0 ist.

Bei zyklischen Strukturen funktioniert dieses einfache Verfahren aber nicht mehr. (Selbst durchdenken.)

3.1.3.7 Graphdurchlauf

Hier verfolgt man alle Zeiger, die von den lokalen Speichern der Programme ausgehen und markiert alle Datenobjekte, die auf diese Weise erreichbar sind. Die nicht-markierten kann man löschen.

Die Halde ja nichts anderes als ein großer Graph mit mehreren Einstiegspunkten (= den Zeigern in den lokalen Speichern der Programme). Ausgehend von diesen Einstiegspunkten wird der Graph mit den bekannten Techniken oder gewissen Varianten durchlaufen, wobei die erreichbaren Knoten mit "true" markiert werden.

Das Verfahren besteht aus zwei Teilen:
- Markiere alle erreichbaren Objekte,
- Entferne alle nicht-markierten Objekte.

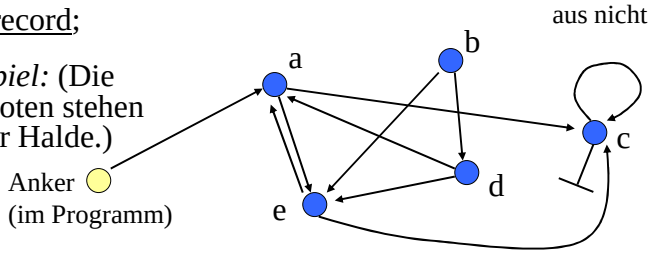
Wir behandeln nur den ersten Teil, da der zweite ein einfacher Halden-Durchlauf mit Umhängen in die Freispeicherliste ist.

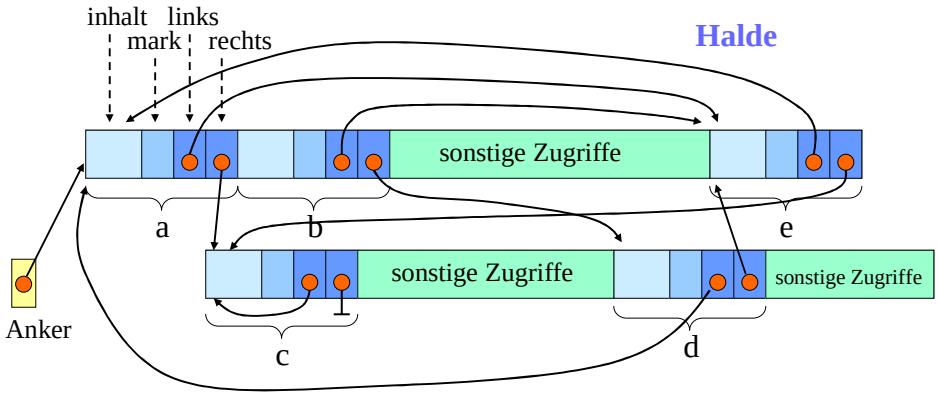
Vereinfachung: Um das Vorgehen zu erläutern, genügt es, Datenobjekte mit zwei Zeigern zu betrachten, also Objekte des folegnden Typs "Knoten":

```
type Knoten;
type Kante is access Knoten;
type Knoten is record
        inhalt:  ...
        mark: Boolean;
        links, rechts: Kante;
end record;
```

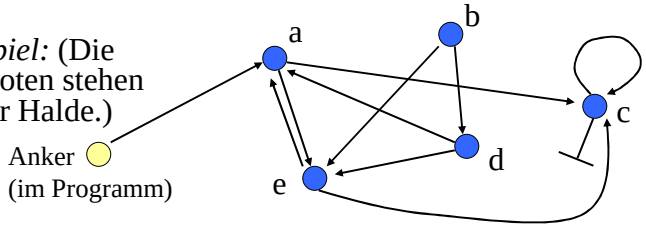
Die Knoten b und d sind vom Programm aus nicht erreichbar.

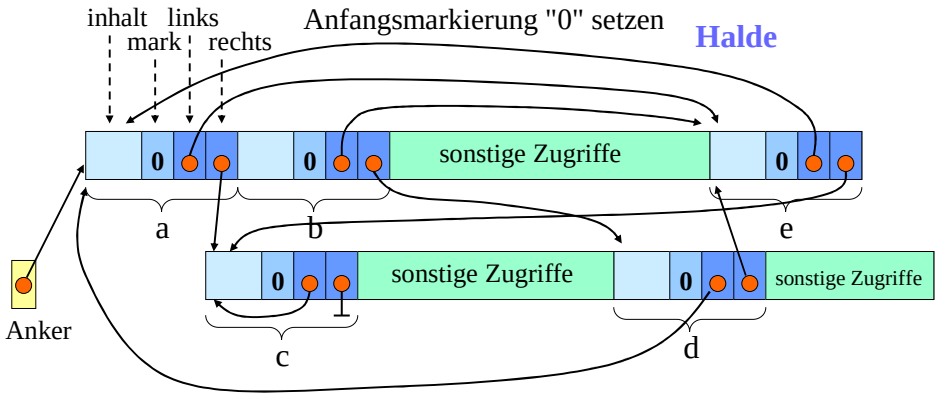
Beispiel: (Die 5 Knoten stehen in der Halde.)



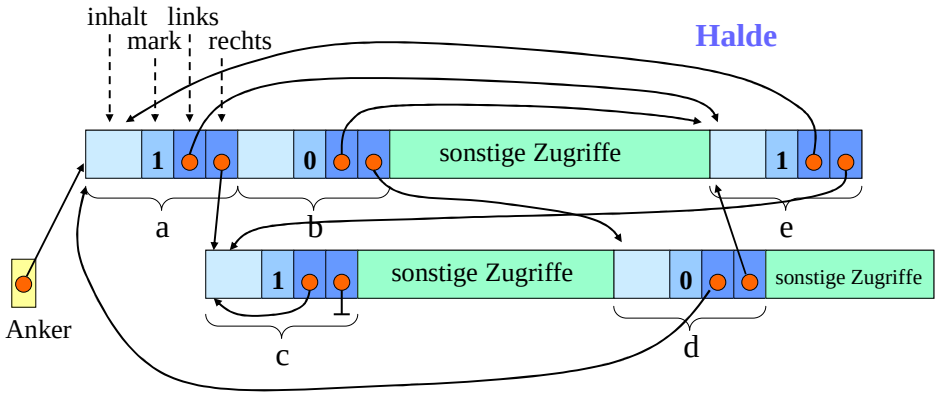


Beispiel: (Die 5 Knoten stehen in der Halde.)





Ziel muss es nun sein, die Knoten b und d als löschbare Knoten zu erkennen. Hierzu durchläuft man ausgehend von "Anker" alle Zeiger und markiert die erreichten Knoten mit true (oder einer "1"). Dies geschieht für alle "Anker", die aus den lokalen Speichern in die Halde verweisen. Danach löscht man alle mit false (oder "0") markierten Objekte und schiebt ggf. den Speicher zusammen.



Ergebnis des Durchlaufs: Ausgehend von "Anker" wurden alle Zeiger nachverfolgt und die hierbei erreichten Knoten mit einer "1" markiert; die nicht erreichbaren bleiben mit "0" markiert.

Anschließend kann man den Speicherbereich neu organisieren, sofern man möglichst große Freispeicherbereiche benötigt.

3.1.3.8 Wir kommen nun zum Algorithmus zur Markierung der erreichbaren und der unerreichbaren Knoten:

Schritt 1:
Markiere alle Knoten in der Halde mit "false" (bzw. mit 0).

Schritt 2:
Markiere alle Knoten in der Halde, die von einem der lokalen Speicher direkt erreicht werden können, mit "true" (bzw. mit 1).

Schritt 3 (eigentlicher Algorithmus): Die Halde möge von Adresse 0 bis Adresse M im Speicher nummeriert sein. Jeder Knoten möge genau r Speicherplätze (Adressen) belegen.
u, v sind vom Typ Knoten, i und j sind Adressen in der Halde.

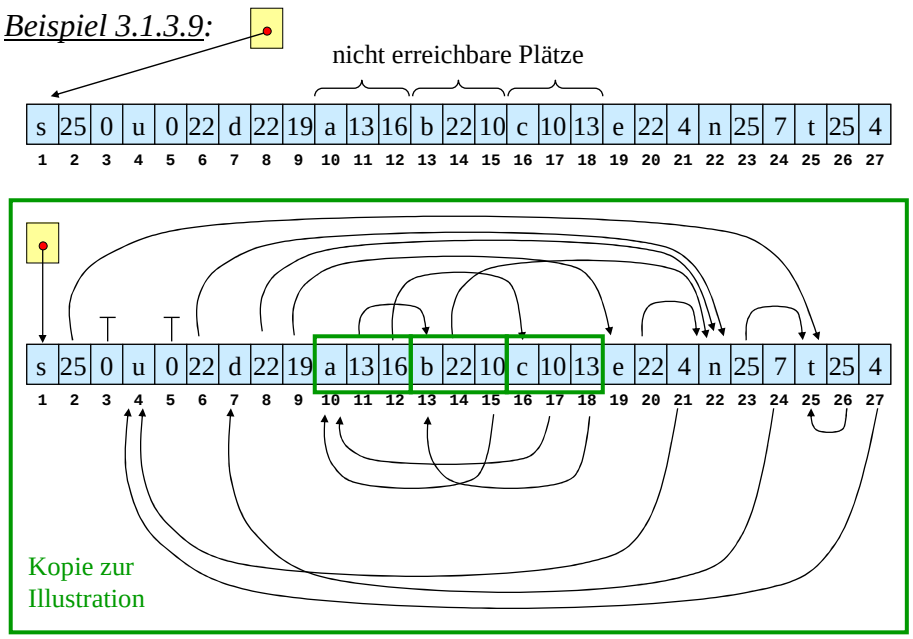

```
i := 0;                    -- i ist die Adresse des betrachteten Knotens
while i <= M loop
   j := i + r;             -- j wird die Adresse des nächsten Knotens
   if der Knoten mit Adresse i ist mit "true" markiert
      and then der Knoten mit Adresse i besitzt mindestens einen
           Nachfolger (d.h.: (links /= null) or (rechts /= null))
   then  if (links /= null) and then (der Knoten u, auf den links
              verweist, ist mit "false" markiert)
         then  markiere den Knoten u mit "true";
              j := Minimum (j, Adresse von u);  end if;
         if (rechts /= null) and then (der Knoten v, auf den rechts
              verweist, ist mit "false" markiert)
         then  markiere den Knoten v mit "true";
              j := Minimum (j, Adresse von v);  end if;
   end if;
   i := j;                 -- zum nächsten Knoten gehen
end loop;
```

Idee dieses Vorgehens: Durchlaufe die Knoten von vorne nach hinten in der Halde. Es interessieren nur die mit true markierten Knoten (nur sie sind bisher vom Programm aus erreichbar). Betrachte deren beide Nachfolgeknoten. Markiere sie mit true und setze das Verfahren an der minimalen Adresse der drei Knoten
- nächster Knoten in der Halde
- linker Nachfolgeknoten
- rechter Nachfolgeknoten

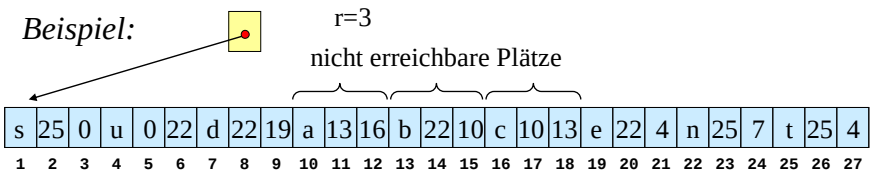
fort.

Auf diese Weise gelangt man schließlich an alle erreichbaren Knoten. Beachte: Dieser Algorithmus ist eine rekursionsfreie Variante des Graphdurchlaufs, der in 1.6.4.3 beschrieben wurde.

Beispiel 3.1.3.9:



Beispiel:



Ablaufdiagramm für i, j und "Markierung auf true setzen":

i	j	Mark.
		1
1	4	
	4	25
4	7	
7	10	
10	13	
13	16	
16	19	
19	22	
22	25	
25	28	

i	j	Mark.
	4	4
4	7	
	7	22
7	10	
10	13	
13	16	
16	19	
19	22	
22	25	
	7	7
7	10	

i	j	Mark.
	10	19
10	13	
13	16	
16	19	
19	22	
	22	
22	25	
25	28	
28		
Ende		

Dies ist bereits der worst case: Es gibt viele Verweise vom Ende der Halde nach vorne (vgl. Komplexitätsabschätzung unten).

Aufwand dieses Verfahrens? (Im worst case quadratisch in M)

Im ungünstigsten Fall beim Durchlauf durch die Halde ist die if-Bedingung erst beim letzten Knoten erfüllt und dessen Verweis führt auf den ersten Knoten zurück, siehe obiges Beispiel.
Nach dem zweiten Durchlauf geschieht das Gleiche mit dem vorletzten Knoten usw.
Wenn n die Zahl der Knoten in der Halde ist, so würde man also $n + (n-2) + (n-4) + ... + 3 + 1 \approx n \cdot (n+1)/4 = O(n^2)$ Schritte ausführen müssen. Wegen $n \approx M/r$ erhält man ein $O(M^2)$-Verfahren. Die Speicherplatzkomplexität ist dagegen konstant (M = Anzahl der Speicherplätze in der Halde).
In der Tat erweist sich dieser Algorithmus in der Praxis auch im Mittel als ein quadratisch mit M wachsendes Verfahren.

Hinweis:
Es ist klar, wie man dieses Verfahren auf Knoten, die mehr als zwei Nachfolger haben können oder deren Größe im Datenobjekt selbst gespeichert ist, erweitern kann:
- for all Nachfolgeknoten (im äußersten then-Teil),
- ersetze j := i+r durch j:=i+größe_des_aktuellen_Knotens.

Das oben genannte Verfahren eignet sich besonders dann, wenn man (fast) keinen freien Speicherplatz mehr zur Verfügung hat. Gibt es dagegen noch Speicherplatz, den man für einen Keller S nutzen kann, dann empfiehlt sich folgender deutlich schnellere Algorithmus, der die weiter zu verfolgenden Zeiger im Keller S ablegt (vgl. 1.6.4.3):

3.1.3.10 Kellerverfahren

Schritt 1: Markiere alle Knoten in der Halde mit "false".
Schritt 2: Markiere alle Knoten in der Halde, die von einem lokalen Speicher direkt erreicht werden können, mit "true" und lege sie (bzw. ihre Adressen) im Keller S ab.
Schritt 3: K sei vom Typ Kante (="Zeiger auf Knoten").

```
while not isempty(S) loop
    while not isempty(S) and (top(S) hat keinen Nachfolger)
        loop pop(S); end loop;
    if not isempty(S) then
        K := top(S);  pop(S);
        if (K.links /= null) and then (not K.links.mark) then
            K.links.mark := true;  push(S, K.links);  end if;
        if (K.rechts /= null) and then (not K.rechts.mark) then
            K.rechts.mark := true;  push(S, K.rechts);  end if;
    end if;
end loop;
```

Aufwand dieses Keller-Verfahrens? (Im worst case linear in M)

Das Verfahren durchläuft jeden Zeiger, der in einem Knoten auftritt, höchstens einmal. Da es höchstens doppelt so viele Zeiger wie Knoten gibt, handelt es sich bei Schritt 3 also um ein O(n')-Verfahren (n' = Zahl der erreichbaren Knoten in der Halde).

Allerdings bezahlt man diese Schnelligkeit mit dem benötigten Speicherplatz für den Keller S. Dieser kann bis zu n/2 Knoten groß werden. Im Mittel wird man aber deutlich weniger Platz brauchen.

Die uniforme Zeitkomplexität dieses Keller-Verfahren wird also vor allem durch Schritt 1 bestimmt, welcher M/r Zeiteinheiten benötigt. Insgesamt ergibt sich damit ein O(M)-Verfahren sowohl bzgl. der Zeit als auch bzgl. des Platzes.

Man kann nun die beiden Algorithmen kombinieren:

Variante 1: Solange noch genügend Platz für den Keller S vorhanden ist, arbeite nach dem Kellerverfahren. Sobald der Keller überläuft, suche man die kleinste Adresse im Keller und schalte mit ihr beginnend auf das andere Verfahren um.

Variante 2: Man verwende statt des Kellers eine sortierte Liste, worin die Zeiger geordnet nach ihrer Adresse eingetragen werden und verwende stets den Zeiger mit der kleinsten Adresse als nächsten zu untersuchenden Knoten. Da die Liste bei jedem Eintrag durchlaufen werden muss, wird dieses Vorgehen zu einem $O(n'^2)$-Verfahren und damit letztlich zu einem $O(M^2)$-Verfahren.

Überlegen Sie sich weitere Kombinationen, Varianten oder Verbesserungen. Ist es z.B. sinnvoll, zuerst den Nachfolgeknoten mit der größeren Adresse in den Keller S zu legen?

Man erkennt nun auch die Abhängigkeit der Speicherbereinigung von der jeweiligen Programmiersprache: Man muss wissen, wie die Datenobjekte / Blöcke / Knoten usw. aufgebaut sind, um Zeiger auch als Zeiger erkennen zu können. Andererseits kann natürlich das Betriebssystem ein universelles Datenformat vorgeben, in dem zum Beispiel die Informationen über Zeiger und die Markierungen an vorgegebenen Stellen notiert werden müssen.

Siehe auch 3.1.4.

Nachdem wir nun die erreichbaren Knoten bzw. Datenblöcke mit "true" markiert haben, kann man alle anderen in die Freispeicherliste (oder in die Buddy-Verwaltung) eintragen und normal weitermachen.

Oft möchte man zusätzlich den Speicher "zusammenschieben" ("kompaktifizieren") und dabei die im Laufe der Rechnungen entstandenen kleinen Fragmente (nicht nutzbaren freien Speicherbereiche) beseitigen. Dieser Kompaktifizierungs-Algorithmus ist bei beliebiger Verzeigerung aufwändig. Man kann Verschiebe-Zeiger mitführen, die die Lage nach der Kompaktifizierung angeben. Dies ist insbesondere bei der Kopiermethode (3.1.3.11) vorteilhaft.

Diese und weitere Fragen zur Verwaltung von Programmen und Daten lernen Sie in Vorlesungen über Betriebssysteme oder auch in speziellen Praktika kennen.

3.1.3.11 Kopiertechniken

Man verwendet aktuell immer nur den halben Speicher. Läuft dieser über, so kopiert man, ausgehend von den Zeigern in den lokalen Speichern der Programme, die erreichbaren Objekte in die andere Hälfte des Speichers, wobei man die relative Anordnung unverändert lässt. Auf diese Weise werden die nicht-markierten Objekte zu Lücken im neuen Speicher und können in die Freispeicherliste eingetragen werden.

Mit etwas erhöhtem Aufwand können die Objekte beim Kopieren von vorne nach hinten nacheinander abgelegt werden, wodurch zugleich der Speicherplatz optimal genutzt wird. In mehreren Durchläufen können hierbei die Verweise entsprechend der neuen Anordnung umgesetzt werden. *(Überlegen Sie, wie so etwas geschehen könnte! Z.B. mit zusätzlichen Verweiszeigern, die den neuen Speicherplatz im alten Objekt speichern.)*

3.1.4: Historische Hinweise

Mit der Entwicklung der ersten Computer in den 1940er Jahren entstanden auch sogleich eindimensionale Felder, da diese genau die Speicherstruktur wiedergaben. Allgemeine Felder finden sich bereits in den Programmiersprachen der 1950er Jahre (Fortran 58, Algol 60, APL). Verbunde und Folgen von Buchstaben treten in Cobol auf (ab 1961). Verschiedene Konzepte der Datenstrukturen wurden in Algol 68 (Standard: 1975) zusammengeführt. Dessen "gut verständlicher" Anteil wurde von Nikolaus Wirth in die Sprache PASCAL (1972) eingebracht, die bis heute als "didaktisches Vorbild" für Programmiersprachen gilt. (Deren Sprachelemente gehören zum Kern des "Programmieren im Kleinen".)

Die Datenstruktur "Keller" entstand ca. 1954 mit ersten Arbeiten über die korrekte Auswertung von arithmetischen Ausdrücken. Sie wurde zugleich ab 1960 verwendet, um den Aufruf von (auch rekursiven) Prozeduren und generell alle klammerartigen Strukturen in Programmen und Programmiersprachen korrekt zu implementieren.

Listen bilden die Grundlage der Programmiersprache LISP (McCarthy, ab 1959). Statt der expliziten Zeiger wurden Operationen wie "head" und "tail" für den Zugriff auf das erste Element einer Liste bzw. auf die Restliste ohne das erste Element benutzt. In SIMULA und PL/I (beide ab 1965) konnten Zeiger verwendet werden. Die Unterscheidung zwischen einem statischen und einem dynamischen Speicher geschah bereits in den ersten Programmiersprachen; die Halde wurde in SIMULA (Koroutinenkonzept) und PL/I erforderlich.

Dass sehr allgemeine Datenstrukturen korrekt übersetzbar sind, demonstrierten die Compiler von SIMULA 67 und etwas später von PL/1. Probleme bereiteten aber die ganz allgemeinen Datenstrukturen von Algol 68, bei denen kartesische Produkte, Vereinigungen, Referenzen, Potenzmengen, Funktionenbildung usw. beliebig miteinander verknüpft werden können: Die Laufzeitsysteme wurden derart kompliziert, dass jeder Algol-68-Compiler gewisse Einschränkungen machen musste.

Für die automatische Speicherbereinigung des Betriebssystems ist es unverzichtbar, *einen Zeiger auch als Zeiger zu erkennen!* Hierzu legt der Compiler (unsichtbar für den Benutzer) für jeden Zeiger einen "Deskriptor" an, aus dem die Struktur des referenzierten Objektes (sein Typ einschl. der Weiterverweise und das Markierungsbit) zu ersehen ist.

Mitte der 1960er Jahre entstanden die ersten Betriebssysteme, die mehrere Programme gleichzeitig verwalten konnten. Ab dieser Zeit entwickelte man diverse Verfahren für die Speicherverwaltung (wie Multikeller, Freispeicher, Bereinigung usw.). Die in diesem Kapitel 3.1. behandelten Vorgehensweisen sind also bereits als "klassische Verfahren" einzustufen.

Für *Echtzeitsysteme* und für *Verteilte Systeme* wurden in den 80er und 90er Jahren neue Verfahren entwickelt. Echtzeitsysteme z. B. können nicht einfach unterbochen werden, so dass die Standardverfahren zu "inkrementellen Techniken" erweitert wurden: Während des Programmablaufs läuft ein Prozess mit, der nicht erreichbare Objekte aufspürt und in die Freispeicherliste einträgt. Hierbei muss natürlich ein wechselseitiger Ausschluss installiert werden, den der Compiler in das übersetzte Programm einfügt.