Einführung in die Informatik

Universität Stuttgart, Studienjahr 2005/06

Gliederung der Grundvorlesung

1. ~~Einführung in die Sprache Ada95~~
2. ~~Algorithmen und Sprachen~~
3. ~~Daten und ihre Strukturierung~~
4. ~~Begriffe der Programmierung~~
5. ~~Abstrakte Datentypen~~ Zurückgestellt
6. ~~Komplexität von Algorithmen und Programmen~~
7. ~~Semantik von Programmen~~
8. ~~Suchen~~
9. ~~Hashing~~
10. ~~Sortieren~~
11. ~~Graphalgorithmen~~
12. Speicherverwaltung

Gliederung des Kapitels

Stand: 24.7.06

12 Verwaltung von Datenstrukturen

12.1 Keller und Halde
12.2 Kellerverwaltung
12.3 Freispeicher- und Haldenverwaltung
12.4 Historische Hinweise

12.1 Keller und Halde

12.1.1 Überblick über die wichtigsten Speicher: Wie in 3.4.3 vorgestellt benötigen Programme mindestens drei Typen von Speichern:

1. Zur Übersetzungszeit bekannter ("statischer") Speicher.
Am Ende der Übersetzung des Programms liegen fest:
1.1: Der Platz, den das übersetzte Programm braucht.
1.2: Der Platz für alle Konstanten und für alle Variablen, die zu einem Datentyp gehören, dessen Speicherplatzbedarf von vornherein feststeht, z.B.: elementarer Datentyp, Aufzählungstyp, Unterbereiche hiervon, records, die nur aus solchen Komponenten bestehen, Felder hierüber von konstanter Länge, access-Datentypen (nur der Zeiger, nicht die hiermit aufgebaute Liste).

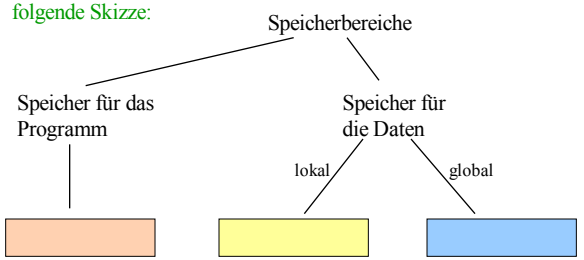
2. Zur Laufzeit beim Abarbeiten der Deklarationen weiterhin erforderlicher dynamischer Speicher (Keller-Speicher).

Viele Variablen, die in klammerartig ineinander geschachtelten Blockstrukturen oder in aufgerufenen Unterprogrammen oder anderen Einheiten stehen oder die parametrisiert sind (dynamische Felder, Records mit Diskriminanten), erhalten ihren Speicherplatz erst zur Laufzeit zugewiesen. Der Platz für solche Objekte wird beim Erreichen der zugehörigen Deklarationen hinten an den zum Programm gehörenden (lokalen) Speicher angehängt und er wird am Ende ihrer Lebensdauer wieder frei gegeben. Wegen der Klammerstruktur, in die die Deklarationen eingebunden sind, ist dieser dynamische Speicher ein Kellerspeicher (Pushdown, Keller), siehe 4.1.4.

Keller-Speicher in der Praxis:
In der Regel legt man auch die Daten, die zum statischen Speicher zählen, in diesem Kellerspeicher (ganz am Anfang) ab, wodurch man die Deklarationen zur Laufzeit einheitlich abarbeiten kann.

3. Zur Laufzeit durch Anweisungen erzeugte Daten, deren Lebensdauer sich nicht an den Programmeinheiten orientiert.
Die Speicherplätze für solche dynamisch erzeugten Daten werden mittels new angefordert und zugeordnet ("allokiert"). Dieser Speicher pulsiert nicht kellerartig, daher liegen diese Speicherplätze in einem allgemeinen Speicherbereich, der Halde. Die Halde kann von vielen Programmen gleichzeitig genutzt werden. Die Speicherplätze in der Halde werden meist explizit freigegeben, sobald die Daten nicht mehr gebraucht werden, oder sie werden mit einer Speicherbereinigung entfernt, sobald die Halde überzulaufen droht.

So erhält man folgende Skizze:



Programmspeicher (für jedes Programm gesondert): Wird beim Ablauf des Programmes nicht verändert (wie bei Turing- und Registermaschine).

Keller (für jedes Programm gesondert): Pulsiert mit dem Wechsel zwischen Blöcken, Prozeduren und weiteren Einheiten.

Halde: In einem großen Speicherbereich werden referenzierte Daten abgelegt. Im Allgemeinen für viele Programme. Wird häufig aktualisiert.

12.1.2 Erinnerung an Pointer/Zeiger/Listen:

Listen sind Folgen von Elementen des gleichen Datentyps. Sie werden durch *Zeiger (pointer, access-Datentypen)* realisiert. Die Verkettung kann einfach oder doppelt, die Anordnung sequentiell oder ringförmig sein. Das erste und/oder das letzte Element sind von außen über einen Zeiger erreichbar (auch *Anker* der Liste genannt). Der Zugriff erfolgt *sequenziell*; man durchläuft also die Liste von vorne nach hinten bzw. von hinten nach vorne, um nach einem Element zu suchen oder um ein Element einzufügen. Die mit einer Liste üblicherweise verbundenen Operationen finden sich unter 3.5.

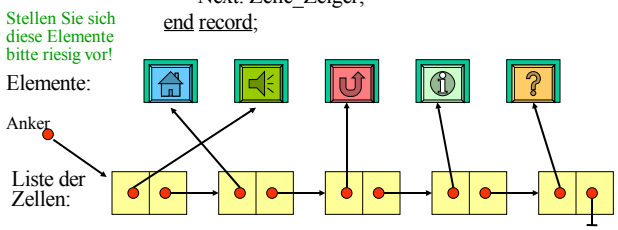
Das Schlüsselwort für Zeiger lautet in Ada access. Typische Definition einer Liste in Ada:

```
type Element is ....;      -- Definiere den Datentyp der Listenelemente
type List_Element;         -- Vorwärtsverweis
type List_Element_Zeiger is access List_Element;
type List_Element  is  record
                          Inhalt: Element;
                          Next: List_Element_Zeiger;
                       end record;
```

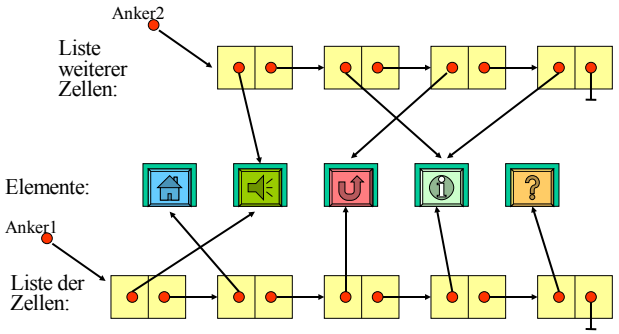
Dies ist eine Liste, in der die Elemente "im Original" stehen. In der Praxis ist dies oft lästig, da ein Element in vielen Listen auftreten kann und dann auch in allen Listen gespeichert und simultan geändert werden muss. Also:


```
type Element is ....;      -- Definiere den Datentyp der Listenelemente
type Element_Zeiger is access Element;
type Zelle;                -- Vorwärtsverweis
type Zelle_Zeiger is access Zelle;
type Zelle is   record
                   Inhalt: Element_Zeiger;
                   Next: Zelle_Zeiger;
                end record;
```

Stellen Sie sich diese Elemente bitte riesig vor!



Will man die Elemente in mehreren Listen gleichzeitig verwenden, so kann dies ohne Kopien der Elemente geschehen:



Vorteile dieser Zellen-Darstellung:

- Elemente können in verschiedenen Listen sein.
- Elemente werden in allen Listen gleichzeitig geändert (da nur das Original geändert werden muss).
- Das Einfügen in andere Listen ist einfach.
- Es lassen sich weitere Zugriffsstrukturen leicht aufbauen.

Nachteile dieser Zellen-Darstellung:

- Der Zugriff auf Elemente dauert evtl. etwas länger.
- Man braucht evtl. mehr Speicherplatz (als z.B. mit arrays).
- Es muss die Halde als Speicher verwaltet werden (meist keine direkte Kontrolle über die dortigen Abläufe).

Hinweis: Listen werden in der Halde abgelegt. Nur die Anker stehen im statischen Bereich oder lokalen Keller des Programms, sofern sie deklarierte Variablen sind.

12.2 Kellerverwaltung

Hiermit ist nicht die (recht einfache) Verwaltung eines einzelnen Kellers (3.5.4) gemeint, sondern die Verwaltung vieler Keller in einem beschränkten linearen gemeinsamen Speicherbereich.

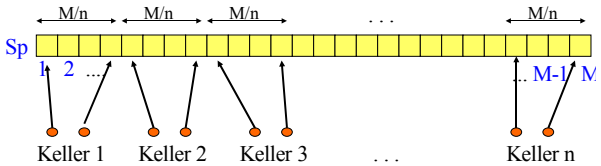
In einem Computer werden gleichzeitig viele Programme ("jobs") verwaltet. Jedes besitzt einen lokalen Keller (die dynamischen Daten werden in der Halde abgelegt, siehe 12.3).

Alle Keller liegen in einem vorgegebenen Speicherbereich Sp(1..M). Manche schwanken stark, andere nur wenig. In der Regel wird der Gesamt-Speicherplatz nicht überschritten, aber wenn man jedem Programm einen festen Bereich zuweisen würde, so wird es oft einen Speicherüberlauf geben. Daher muss man den Gesamt-Speicherplatz geeignet auf die jeweils laufenden Programme verteilen. Allgemeine Formulierung des Problems:

12.2.1: Implementierung von mehreren Kellern

Aufgabe: Wir wollen eine ***Multikellerverwaltung*** in einem eindimensionalen Feld durchführen, d.h.:
Es sollen n Keller verwaltet werden. Insgesamt steht hierfür ein linearer Speicher Sp(1..M) zur Verfügung.

Spontane Idee: Jeder Keller erhält gleich viele Speicherplätze, nämlich M/n:



Diese Verweise werden wir durch Indizes realisieren.

Einfachster Fall: **n = 1**. Es liegt ein einzelner Keller vor. Die Ada-Formulierung hierfür ist ein generisches Paket, z.B.:

```
generic
   M: Positive := 5_000_000;          -- Initialisierung willkürlich
   type Element is private;
package Keller is
   procedure newkeller;               -- Leeren des Kellers
   function isempty return Boolean;   -- Ist der Keller leer?
   function isfull return Booelan;    -- Ist der Keller voll?
   function top return Element;       -- Oberstes Kellerelement
   procedure push (x: in Element);    -- Füge Element x oben an
   procedure pop;                     -- Lösche oberstes Element
   function length return Natural;    -- Aktuelle Kellerlänge
   unterlauf, ueberlauf: exception;   -- Ausnahmebehandlung
end Keller;
```


Hieran schließt sich der Modulrumpf an:

```
package body Keller is
   type Speicher is array (1..M) of Element;
   Sp: Speicher;
   index: Integer range 0..M := 0;
   procedure newkeller is begin index := 0; end;
   function isfull return Booelan is
      begin return index >= M; end;
   procedure push (x: in Element) is
      begin  if isfull then raise ueberlauf;
             else index := index + 1; Sp(index) := x; end if; end;
   ...
   < selbst schreiben: die Prozedur pop, die Funktionen
isempty,    length, top und die Ausnahmen unterlauf und
ueberlauf >
end Keller;
```


Eine Instanz kann nun lauten:

```
package Ganzzahlkeller is
        new Keller (M => 800_000, Element => Integer);
```

Nächster Fall: **n = 2**. Wenn man zwei Keller auf einem linearen Speicher Sp der Größe 1 .. M unterbringen möchte, so wird man den ersten Keller von 1 an aufwärts und den zweiten Keller mit M beginnend abwärts implementieren. Auf diese Weise wird der Speicher Sp optimal genutzt.

Aufgabe: Realisieren Sie diesen Fall selbst!

12.2.2 Allgemeiner Fall: **n ≥ 3**. Vorhandener Speicher Sp(1..M). Hier gibt es mindestens zwei Varianten:

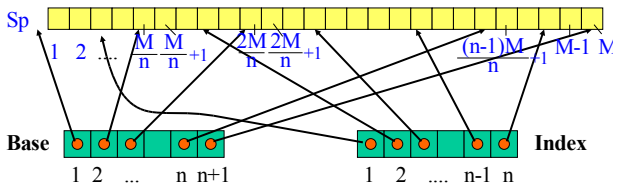
- *Variante 1:* Jeder Keller hat seine eigene maximale Größe, die in Max: array (1..n) of Natural abgelegt ist (einfachster Fall: Max(i) = ⌊M/n⌋ für alle i) und für die gilt

$$\sum_{i=1}^{n} \text{Max}(i) \le M.$$

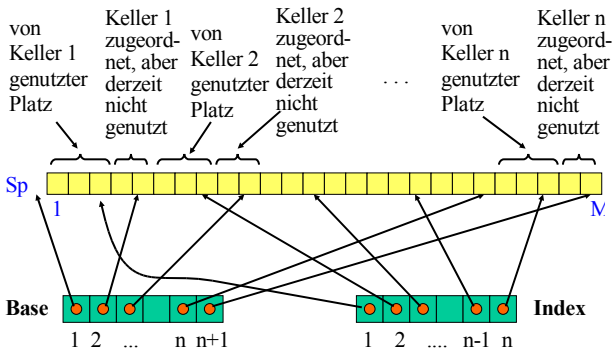
Dieser Fall wird wie "n=1" behandelt, indem überall die Nummer des Kellers hinzugefügt wird und jeder Keller unabhängig von den anderen ist. Es gibt dann zwei Felder für die Adresse vor dem Beginn des i-ten Kellers ("Base") und für seine aktuelle oberste Position ("Index") mit
$0 \le \text{Index}(i) - \text{Base}(i) \le \text{Max}(i)$ für i = 1, 2, ..., n.
Wir formulieren dies nun genau aus.

Implementierung: Jeder Keller erhält den gleichen Platz der Größe M/n. Wir verwenden hierfür zwei Zeiger bzw. Indizes:

Base(i) zeigt auf den Speicherplatz, der unmittelbar vor dem Bereich für den i-ten Keller liegt;
Index(i) zeigt auf den Speicherplatz, auf dem sich das oberste Element des i-ten Kellers befindet.



Nochmals zur Illustration: Aufteilung des Speichers Sp



12.2.3 Ada Deklarationen hierzu: (MKV = Multikellerverwaltung)

```
generic
   M: Positive := 5_000_000;      -- willkürlicher Default-Wert
   n: Positive;                   -- Anzahl der Keller, n ≥ 2
   type Element is private;       -- Datentyp der Kellerelemente
package MKV1 is
   type ZK is Positive range (1..n+1);      -- für Zugriff auf Keller
   procedure newkeller (i: in ZK);          -- Leeren des Kellers
   function isempty (i: ZK) return Boolean; -- Ist der Keller leer?
   function isfull (i: ZK) return Booelan;  -- Ist der Keller voll?
   function top (i: ZK) return Element;     -- Oberstes Kellerelement
   procedure push (i: in ZK; x: in Element); -- Füge x oben an
   procedure pop (i: in ZK);                -- Lösche oberstes Element
   function length (i: in ZK) return Natural; -- Aktuelle Kellerlänge
   unterlauf (i: ZK), ueberlauf (i: ZK): exception;
end MKV1;
```


Paketrumpf hierzu: (MKV = Multikellerverwaltung)

```
package body MKV1 is
type Adressen is Natural range 0..M;  -- Speicher"adressen"
Sp: array (Adressen) of Element;      -- Speicher
Base: array (ZK) of Adressen;         -- Beginn jedes Kellers
Index: array (ZK) of Adressen;        -- Aktueller Stand jedes Kellers
procedure newkeller (i: in ZK) is
   begin  Index(i) := Base(i);  end newkeller;
function isempty (i: ZK) return Boolean is
  begin  return  Base(i) = Index(i);  end isempty;
function isfull (i: ZK) return Boolean is
  begin  return  Base(i+1) >= Index(i);  end isfull;
function top (i: ZK) return Element is
  begin  return Sp(Index(i));  end top;
```


Paketrumpf MKV1 (Fortsetzung)

```
procedure push (i: in ZK; x: in Element) is
  begin  if isfull(i) then raise ueberlauf (i);
         else  Index(i) := Index(i)+1;
               Sp(Index(i)) := x; end if;
  end push;
procedure pop (i: in ZK) is
  begin  if isempty(i) then raise unterlauf(i);
         else Index(i) := Index(i)-1; end if; end pop;
function length (i: ZK) return Natural is
  begin return Index(i)-Base(i); end length;
exception   when ... =>.....
end MKV1;
```


Eine konkrete Instanz für zehn Keller könnte dann sein (hier wird der voreingestellte Wert von M genommen):

```
package Zahlenkeller is new MKV1(n => 10; Element => Integer);
use Zahlenkeller; ...
for i in 1..n loop Base(i) := (i-1)*(M/n); Index(i):=Base(i); end loop;
Base(n+1) := M; ...
```

Nachteilig ist, dass die Multikellerverwaltung zusammenbricht, falls irgendein Keller überläuft. In der Regel stehen ja noch weitere Speicherplätze in Sp zur Verfügung.

Es gibt diverse nahe liegende Veränderungen. Diese ersetzen alle „raise ueberlauf(i)" durch den Prozeduraufruf „umordnen(i)", um weiteren Speicherplatz bereitzustellen (siehe unten in Variante 2):

```
procedure umordnen (i: in ZK);  ...
```


12.2.4 *Variante 2:* Die Größe jedes einzelnen Kellers ist nicht vorab beschränkt und alle Keller zusammen sollen den Speicherplatz der Größe M möglichst gut nutzen. Hierfür muss es wiederum zwei Felder Base, Index: array (1..n) of Natural geben, für die zu jedem Zeitpunkt gilt

$$\sum_{i=1}^{n} \text{Index}(i)-\text{Base}(i) \le M.$$

Wenn einer der Keller überläuft (d.h. push-Operation bei Index(i) = Base(i+1)) und wenn zugleich andere Keller den ihnen zugewiesenen Bereich noch nicht voll ausnutzen, *so muss der Speicherplatz neu auf die Keller verteilt werden.*

Hier sind mehrere Untervarianten möglich.

Möglichkeit 1:
Schaue nach, ob der rechte oder linke Nachbar des Kellers i noch genügend freien Platz hat und tritt dann die Hälfte dieser Plätze an den Keller i ab.

Möglichkeit 2:
Suche einen Keller j mit maximal viel freiem Platz, das heißt, Index(j)-Base(j) ist maximal, und tritt die Hälfte dieser Plätze an den Keller i ab. Konkret muss dann der Speicherbereich zwischen den Kellern i und j um q Speicherplätze verschoben werden, wobei q die Hälfte der freien Plätze von Keller j ist.

Möglichkeit 3:
Berechne den Speicherplatz, den jeder Keller bekommen soll, neu, indem jedem Keller eine Mindestzahl an Plätzen und weitere Plätze entsprechend seines bisherigen Wachstums zugewiesen werden, und ordne den Speicher dann komplett um.

Möglichkeit 1: (nur die benachbarten Keller betrachten)

```
procedure umordnen (i: in ZK) is       -- n muss mindestens 3 sein
   k: ZK;  q: Adressen;
begin  k := i;   -- Teste auch, ob beim Keller k mindestens 2 Plätze frei sind
   if (i=1) and (Index(2)+1 < Base(3)) then k := 2;
   elsif (i=n) and (Index(n-1)+1 < Base(n)) then k := n-1;
   elsif Base(i)-Index(i-1)+1 < Base(i+2)-Index(i+1)
      then k := i+1;
      elsif Index(i-1)+1< Base(i) then k := i-1; end if;
                -- Keller k dient nun als Platz-Lieferant
   if k=i then raise ueberlauf;
   elsif k<i then
      q := (Base(k+1)-Index(k))/2;      -- Hälfte des freien Platzes
      Base(i) := Base(i) - q; Index(i) := Index(i) - q;
      for j in Base(i)..Index(i) loop Sp(j) := Sp(j+q); end loop;
   else  <das Gleiche, aber nach oben verschieben; selbst einfügen> end if;
end umordnen;
```


Möglichkeit 2: (Keller, der maximal viel Platz abgeben kann, suchen)

```
procedure umordnen (i: in ZK) is
   k: ZK;  q: Adressen;
begin  k := 1;                          -- Suche Keller k mit maximal freiem Platz
   for j in 2..n loop
      if (Base(j+1)-Index(j)) > (Base(k+1)-Index(k))
      then k := j; end if;
   end loop;
   q := (Base(k+1)-Index(k))/2;   -- Hälfte des freien Platzes
   if q <= 0 then raise ueberlauf;
   elsif k<i then
      for j in k+1..i loop
         Base(j) := Base(j)-q; Index(j) := Index(j)-q; end loop;
      for j in Base(k+1)..Index(i) loop Sp(j) := Sp(j+q); end loop;
   else  < das Gleiche, nur nach oben verschieben; selbst einfügen > end if;
end umordnen;
```

Nachteil der Möglichkeit 1:
Ein Abbruch kann geschehen, obwohl noch irgendwelche anderen Keller ihren Platz kaum benötigen. Denn man prüft ja nur die benachbarten Keller ab. Auch kann "umordnen" relativ rasch wieder aufgerufen werden.

Nachteil von Möglichkeit 2:
Eventuell wird die Prozedur "umordnen" nach q Schritten erneut aufgerufen, vor allem, wenn ein Keller schnell wächst. Komplettes Neuverteilen der Speicherbereiche vermeidet dies.

Vorteil:
Die Prozedur "umordnen" wird sehr schnell abgearbeitet.

12.2.5 Möglichkeit 3: (Garwick-Algorithmus)

- Berechne den insgesamt freien Platz aller Keller ("sum").
- Berechne den gesamten Zuwachs seit dem letzten Umordnen.
- Verteile 10% des freien Platzes gleichmäßig an alle Keller.
- Verteile 90% des freien Platzes proportional zum Zuwachs.

Um den Zuwachs zu berechnen, muss man sich in einem array ***AltIndex*** merken, welches die Indexpositionen *unmittelbar nach* dem letzten Umordnen waren. Um die Umordnung durchzuführen, muss man die neuen Basispositionen in einem array ***NewBase*** notieren. Der Zuwachs ergibt sich dann aus der Summe der Werte (Index(j)-AltIndex(j)), wobei man nur die positiven Werte addieren darf. NewBase(j) ergibt sich aus den Newbase-Werten der darunter liegenden Keller erhöht um den festen Anteil u, der jedem Keller zusteht, und dem Zuwachs-Anteil.

Die Formeln wollen wir zunächst genau angeben.

Zu berechnende Größen (u und w gerundet, weil ganzzahlig):

sum := gesamter freier Speicherplatz aller n Keller

zuwachs :=
Summiere die nichtnegativen Werte von Index(j) - Altindex(j)
Hier werden nur die Keller berücksichtigt, die seit dem letzten Umordnen gewachsen sind.

$u := \lfloor (sum/10) / n \rfloor$
u ist 10% der Zahl freier Speicherplätze anteilig für jeden Keller; mindestens u Speicherplätze werden jedem Keller als freie Plätze zugewiesen.
$u*n$ sind die freien Speicherplätze, die vorab an alle Keller verteilt werden.
sum-$u*n$ ist der restliche Speicherplatz, der nach Zuwachs zuzuordnen ist.

$v := (sum - u*n)/zuwachs$, $w := \lfloor v*\delta \rfloor$
Wenn ein Keller um δ Plätze gewachsen ist, so erhält er diese w zusätzlichen Speicherplätze, aber nur im Falle $\delta > 0$. (v ist der Faktor je Zuwachseinheit.)

Damit sind die Formeln in folgender Prozedur "umordnen" erklärt (diese Prozedur braucht keine Parameter mehr):

Garwick-Algorithmus: Füge zum "package body MKV1" hinzu:

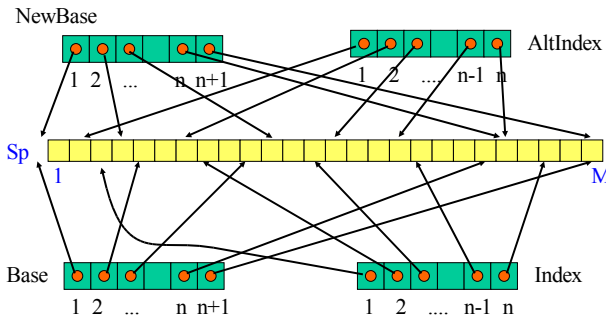
```
NewBase: array (ZK) of Adressen;     -- Neuer Beginn der Kellers
AltIndex: array (ZK) of Adressen;    -- Alter Stand jedes Kellers

procedure umordnen is -- bei Möglichkeit 3 brauchen wir keinen Parameter i
   k: ZK;  sum, zuwachs, u, h: Integer; v: Float;  -- n ist global
   Delta: array (ZK) of Adressen;       -- für den Zuwachs jedes Kellers
begin  sum := 0;                        -- Addiere freien Platz in "sum" auf
   for j in 1..n loop  sum:=sum+Base(j+1)-Index(j); end loop;
   if sum <= n then raise ueberlauf;
      -- Nicht genug Platz frei; bei sum > n vermeidet man eine unendliche
      -- Schleife durch ständig erneutes Aufrufen des Garwickalgorithmus
   else  zuwachs := 0;    -- ermittle Zuwächse seit letztem "umordnen"
      for j in ZK loop h := Index(j) - AltIndex(j);
         if h > 0 then Delta(j) := h; zuwachs := zuwachs+h;
         else Delta(j) := 0; end if;
      end loop;
```



```
   if zuwachs >= 1 then
      u := INTEGER(0.1*FLOAT(sum)/FLOAT(n));
            -- 10% Anteil, der jedem Keller zusteht)
      v := (FLOAT(sum) - FLOAT(u*n))/FLOAT(zuwachs);
   else u := INTEGER(FLOAT(sum)/FLOAT(n)); v:=0; end if;
      -- Dieser else-Fall darf eigentlich nicht eintreten, da ja ein Keller
      -- überläuft; man könnte also auch eine exception erwecken.
   NewBase(1) := 0; NewBase(n+1) := M;
   for j in 2..n loop
   NewBase(j) := NewBase(j-1) + Index(j-1) - Base(j-1)
                 +u+INTEGER(Delta(j-1)*v-0.5);  end loop;
   speicherumordnen;
   for j in ZK loop AltIndex(j) := Index(j); end loop;
 end if;
end umordnen;
-- Hier kann am Ende wegen Rundungsfehlern freier Speicherplatz übrig
-- bleiben (INTEGER(...-0.5)). Wie viel? Wie kann man dies berücksichtigen?
```


Garwick-Algorithmus: Verwaltung des Speichers Sp



Unterprozedur zu "umordnen":

```
procedure speicherumordnen is
   m, j, k: ZK;
begin  j := 2;
   while (j <= n) loop
      k := j;
      if  NewBase(k) < Base(k)  then  verschieben(k);
      else   while NewBase(k+1) > Base(k+1) loop
                                  k := k + 1;  end loop;
                        -- Diese Schleife endet spätestens für k = n
           for m in reverse j..k loop  verschieben(m);  end loop;
      end if;
      j := k+1;
   end loop;
end speicherumordnen;
```


Unterprozedur zu "speicherumordnen":

```
procedure verschieben (i: in ZK) is
d: Integer;
begin d := NewBase(i) - Base(i);
            -- d gibt an, um wie viele Stellen Keller i verschoben werden muss
   if (d /= 0) then
      if d > 0 then
         for a in reverse Base(i)+1 .. Index(i) loop
            Sp(a+d) := Sp(a);  end loop;
      else
         for a in Base(i)+1 .. Index(i) loop
            Sp(a+d) := Sp(a);  end loop;        -- beachte hier d<0
      end if;
      Index(i) := Index(i)+d;
      Base(i) := NewBase(i);
   end if;
end verschieben;
```


Hinweis: Wo kommen die Konstanten 10 und 90 im Garwickalgorithmus her? Warum nicht 30 und 70?

Dies sind Erfahrungswerte: Man hat die obige Umordnung der Kellerspeicherbereiche für eine feste Menge von Programmen inmer wieder für verschiedene Prozentzahlen durchgeführt und hierbei festgestellt, dass der Wert 10% für die Festzuweisung (und somit 90% für den Zuwachs) den besten Durchsatz, d.h., im Mittel die geringste Zahl an Umordnungen ergab.

12.2.6 Verwaltung durch ein Betriebssystem

- Der gesamte Kellerspeicher wird vom Betriebssystem verwaltet.
- Jedes Programm, das neuen Speicherplatz benötigt oder alten zurückgibt oder auf seine Daten zu greifen möchte, schickt eine Anfrage an das Betriebssystem mit "Nummer des Programms i" und "Relative Speicherplatzadresse a".
- Die zugehörige absolute Speicherplatzadresse im Rechner ist dann Base(i) + a. Ist dieser Wert kleiner oder gleich Base (i+1), stellt das Betriebssystem den Wert zur Verfügung bzw. speichert ihn ab bzw. legt ihn neu an oder gibt ihn frei; anderen-falls wird die Prozedur umordnen durchgeführt und erst danach entweder die Anfrage bedient oder, falls neuer Speicherplatz nicht mehr zugewiesen werden kann, die weitere Bearbeitung des Programms i unterbrochen und der Konfliktfall irgendwie gelöst (notfalls mit dem Abbruch mindestens eines Programms).

12.3 Freispeicher- und Haldenverwaltung

Daten oder Objekte lassen sich durch Zeiger miteinander verflechten. Früher sprach man dann von "Geflechten", die im Speicher aufzubauen sind. Es handelt sich hierbei um *Graphen* (vgl. 3.8).

Um solche Geflechte oder Vernetzungen aufzubauen, muss in jedem Datenobjekt mindestens ein Zeiger existieren.

Existiert genau ein Zeiger, so kann man nur Listen aufbauen. Ab zwei Zeigern lassen sich stark vernetzte Strukturen realisieren. Beispiel: Binäre Bäume. Siehe hierzu Abschnitte 3.7 und 8.2.

Erläuternder Hinweis:

In der Implementierung werden "Zeiger" stets durch die "Adresse eines Speicherplatzes", ab der die referenzierte Struktur beginnt, dargestellt.

Grund: Man beachte, dass heutige Rechner in der Regel einen ein-dimensionalen Speicher besitzen, auf dessen Speicherplätze über einen Index von 0 bis 2^s-1 (für eine natürliche Zahl s) zugegriffen wird. Einen Zeiger implementiert man daher als die Adresse derjenigen Speicherzelle, ab der das Objekt, auf das verwiesen wird, steht.

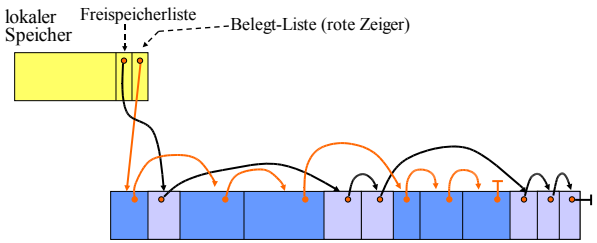
12.3.1 Halde *(engl.: Heap)*

Dies ist ein Speicherbereich, dessen Größe hinreichend groß ist, um die mittels new erzeugten Datenobjekte (Zugriff über Zeiger!) abzulegen. Wenn diese Datenobjekte im Laufe der Rechnungen nicht mehr gebraucht werden, sollten sie explizit wieder frei gegeben werden (in Ada mit Hilfe des pragmas "Controlled" und der Prozedur FREE). Die Verwaltung erfolgt oft über eine Freispeicherliste.

Werden die nicht mehr benötigten Speicherplätze nicht wieder frei gegeben, so liegen nach einiger Zeit in der Halde viele unnötige Datenobjekte herum (= Daten, auf die nicht mehr von irgendeinem Programm über seinen lokalen Speicher zugegriffen werden kann). So kann die Halde rasch voll werden. Um dann noch weiterarbeiten zu können, müssen die nicht mehr benötigten Datenobjekte erkannt, ihre Speicherplätze frei gegeben und die Halde in geeigneter Weise umorganisiert werden (Speicherbereinigung).

12.3.2 Freispeicherliste

Um die dynamischen Daten der Halde zu verwalten, tragen wir die freien Speicherplätze in eine "Freispeicherliste" ein, aber nicht jede Speicherzelle einzeln, sondern immer ganze "Datenblöcke".



Halde (blau = belegte Bereiche)

Benutzen mehrere Programme die Halde, so werden die "Freispeicherliste" und eventuell auch eine "Belegt-Liste" vom Betriebssystem verwaltet. Folgende Aufgaben sind unter anderen Fragestellungen zu lösen:

1. Ein Programm fordert einen Speicherplatzbereich der Größe "G" an. Weise dem Programm einen geeigneten Bereich in der Halde zu und modifiziere die Freispeicherliste.
2. Ein Programm gibt einen Speicherplatzbereich wieder frei. Füge diesen Bereich "geschickt" in die Freispeicherliste ein.
3. Verschmelze aneinander grenzende freie Datenblöcke der Halde zu größeren Einheiten.

4. Falls keine Zuweisung erfolgen kann, ordne die Halde so um, dass alle freien Bereiche nebeneinander liegen (das ist nicht trivial). Füge hierbei alle Datenblöcke, die nicht mehr benutzt werden, in die Freispeicherliste ein.
5. Falls auch dies nicht erfolgreich ist, führe einen Austausch der Speicherinhalte mit dem Hintergrundspeicher durch (Stichwort: Seitenaustauschstrategien, Paging; siehe Vorlesungen über Betriebssysteme).

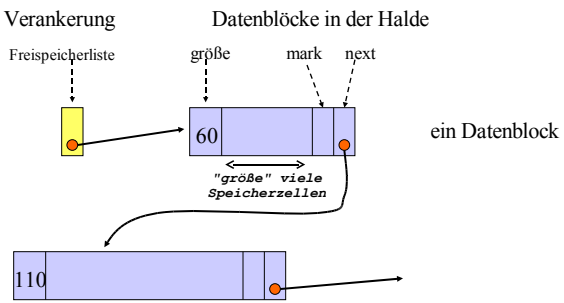
Um diese Aufgaben durchzuführen, muss die Freispeicherliste oft durchlaufen werden, wobei wir die Datenblöcke, die zur Freispeicherliste gehören, markieren, um sie später "erkennen" zu können. Ein Datenblock muss also neben dem Inhalt, den das jeweilige Programm hineinschreibt, mindestens seine Größe, ein Markierungsfeld und den Verweis auf den nächsten freien Datenblock enthalten.

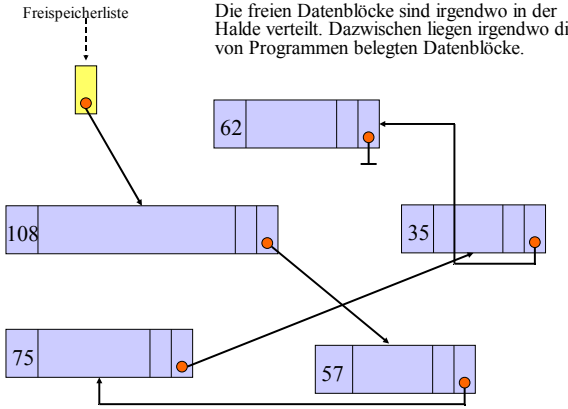
Wir legen daher folgenden Datentyp "DBlock" für Datenblöcke fest. Es sei eine natürliche Zahl "maxgröße" (= die maximale Größe an Speicherplätzen je Datenblock) vorgegeben (vgl. auch 1.12.3 "Diskriminanten für Größenangaben"):

```
type DBlock;
type DBlockzeiger is access DBlock;
subtype AnzahlZellen is Positive range 1..maxgröße;
type DBlock (größe: AnzahlZellen := maxgröße) is record
        inhalt: array (1..größe) of Speicherzelle;
        -- Komponenten, die genau "größe" viele Speicherzellen belegen
        mark: Boolean;
        next: DBlockzeiger;
        end record;
```

Jeder DBlock belegt also größe + x viele Speicherzellen in der Halde, wobei x die Zahl der Speicherzellen für "größe", "mark" und "next" bezeichnet.

Skizze:





Die freien Datenblöcke sind irgendwo in der Halde verteilt. Dazwischen liegen irgendwo die von Programmen belegten Datenblöcke.

12.3.3 Bearbeitungsstrategien: Ein Programm fordert einen Datenblock mit m Speicherplätzen an.

Algorithmus 1: **First Fit**
Gehe die Freispeicherliste durch, bis ein Datenblock D mit größe ≥ m gefunden ist.
Mache hieraus zwei Datenblöcke: Einen mit m+x und einen mit größe-m-x Speicherplätzen (x = Speicherplatz für größe, mark und next, siehe Datentyp DBlock in 12.3.2).
Füge diese beiden Datenblöcke in die Freispeicherliste anstelle des DBlocks D ein.
Klinke den ersten dieser beiden Datenblöcke aus der Freispeicherliste aus und ordne ihn dem Programm zu.
Hinweise: Falls der zweite Block "zu klein" ist, vermeide die Aufspaltung in zwei Datenblöcke und weise ganz D dem Programm zu. Falls kein geeigneter Block D existiert, rufe die Speicherbereinigung auf, siehe unten.

Algorithmus 2: **Best Fit**
Gehe die gesamte Freispeicherliste durch und ermittle den kleinsten Datenblock D mit größe ≥ m.
Fahre anschließend fort wie bei "First Fit".

Welche Strategie ist besser?

Bei beiden Methoden entstehen im Lauf der Zeit viele kleine Datenblöcke, die verstreut in der Halde liegen. Diese sog. "Fragmentierung" des Speichers erfordert häufige Aufrufe der Speicherbereinigung. In der Praxis erweist sich die Best-Fit-Strategie gegenüber der "First-Fit-Strategie" nach einiger Zeit als schlechter (!), da hierbei besonders kleine Datenblöcke entstehen; außerdem muss bei Best-Fit stets die gesamte Freispeicherliste durchlaufen werden.

Aus der Praxis weiß man: Solange der freie Speicher etwa ein Drittel der Halde ausmacht, ist die First-Fit-Strategie gut anwendbar. Wird aber der freie Platz geringer, so muss oft eine zeitaufwändige Speicherbereinigung durchgeführt werden, die zu Wartezeiten bei den "Kunden" führt.

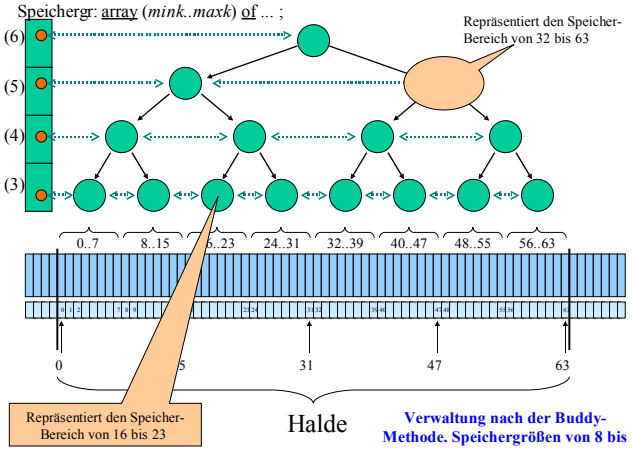
Recht nachteilig ist die Zeit, die beim Durchlaufen der Liste verstreicht. Zwei Ideen zur Verbesserung:

- Halte die Freispeicherliste stets nach der Größe der Datenblöcke sortiert. Nachteil: Das Einfügen freigegebener Speicherbereiche ist dann aufwändiger, dafür erfolgt aber die Suche nach einem passenden DBlock im Mittel schneller.
- Lege einen binären Suchbaum über die Freispeicherliste. Die Suche erfolgt dann schneller. Nachteile: Weiterer Speicher-platz; diese Suchbäume müssen ebenfalls in der Halde unter-gebracht werden, da sie dynamische Datenstrukturen sind; sie müssen eventuell ständig geändert werden.

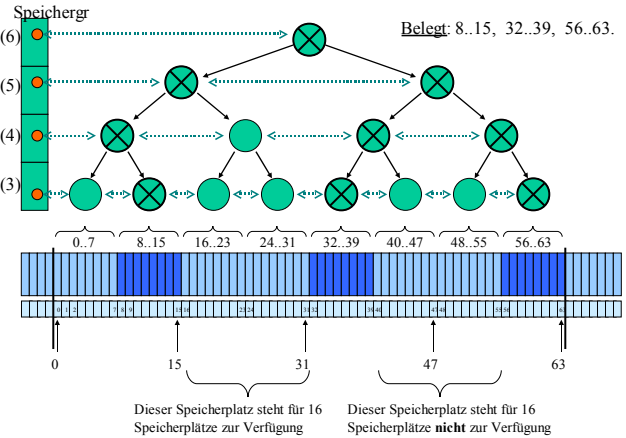
12.3.4 Buddy-Verfahren

Wir stellen kurz einen alten Vorschlag zur Verwaltung des freien Speicherplatzes anstelle einer Freispeicherliste 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 $mink \le k \le maxk$. (Man schränkt in der Praxis k ein z.B. zwischen $mink = 10$ und $maxk = 25$.) 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 $\mathbf{buddy_k(x)}$ die Anfangsadresse seines Buddy (dieser liegt entweder links oder rechts von ihm und besitzt die gleiche Größe).

Es gilt:

$$\mathbf{buddy_k(x)} = \begin{cases} x + 2^k, & \text{falls } x = 0 \bmod 2^{k+1} \\ x - 2^k, & \text{falls } x = 2^k \bmod 2^{k+1} \end{cases}$$

Wir programmieren diese Form der Speicherverwaltung 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 $2^{k-1} < m \le 2^k$ 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.

ok, aber ...

So realisiert man es natürlich nicht, sondern...? Und warum?

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.)

12.3.5 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: Verweiszählermethode, Graphdurchlauf und Kopieren.

12.3.6 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 Verfah-ren aber nicht mehr. (Klar!?)

12.3.7 Graphdurchlauf

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

Die Halde ist nichts anderes als ein großer Graph mit mehreren Einstiegspunkten (= den Zeigern der lokalen Speicher in den Programmen). Ausgehend von diesen Einstiegspunkten wird der Graph mit den bekannten Techniken (3.8.7 und 8.8.9) 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.

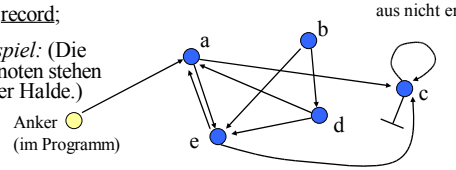
Wir behandeln nur den ersten Teil, da der zweite ein einfacher Halden-Durchlauf mit Umhängen in die Freispeicherliste ist.

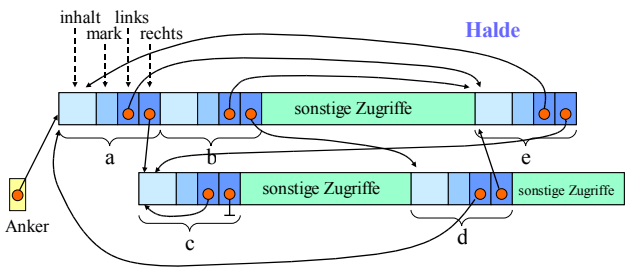
Vereinfachung: Um das Vorgehen zu erläutern, genügt es, Datenobjekte mit zwei Zeigern zu betrachten, also Objekte des folgenden Typs "Knoten":

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

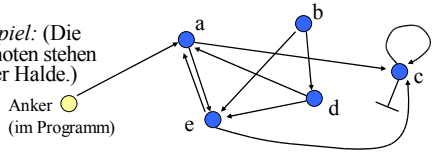
Die Knoten b und d sind vom Programm aus nicht erreichbar.

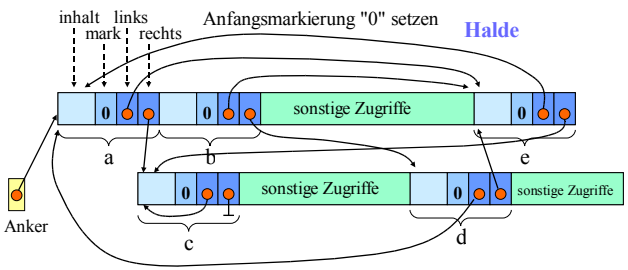
Beispiel: (Die 5 Knoten stehen in der Halde.)



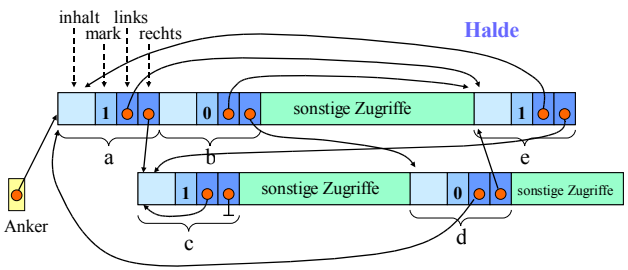


Beispiel: (Die 5 Knoten stehen in der Halde.)





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.

12.3.8 Wir kommen nun zum Algorithmus zur Markierung der erreichbaren und der unerreichbaren Knoten unter der Annahme, dass kein freier Speicherplatz für den Algorithmus zur Verfügung steht:

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.

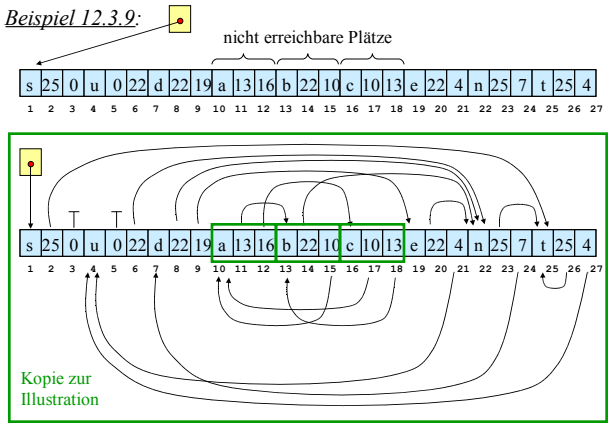

```
i := 0;                          -- i ist die Adresse des betrachteten Knotens
while i <= M loop
   j := i + r;                   -- j wird die Adresse des nächsten Knotens
   if der Knoten mit Adresse i ist mit "true" markiert
      and then der Knoten mit Adresse i besitzt mindestens einen
           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

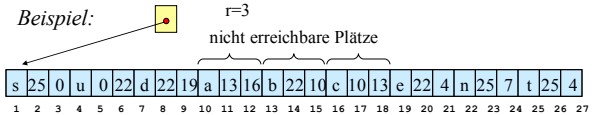
fort.

Auf diese Weise gelangt man schließlich an alle erreichbaren Knoten. Beachte: Dieser Algorithmus ist eine rekursionsfreie Variante des Graphdurchlaufs, der in 3.8.7 beschrieben wurde.

Beispiel 12.3.9:



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 (machen Sie sich klar: dies ist der Algorithmus GD aus 3.8.7 beschränkt auf Ausgangsgrad 2):

12.3.10 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 (in der Praxis: 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-Verfahrens 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 12.4.

Zum Durchlaufalgorithmus in Linearzeit ohne einen Keller vgl. den Durchlauf durch binäre Bäume (Stichwort: Schorr-Waite-Algorithmus, siehe Literatur) bzw. Hinweise in den Übungen.

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 (12.3.11) vorteilhaft.

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

12.3.11 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.)*

Um sich in dieses Problem hineinzudenken, sollten Sie die Frage lösen, wie man einen Graphen kopiert.

Dies sieht einfach aus, hat aber seine Tücken, wenn man keine Zusatzzeiger mitführen oder (wegen fehlendem Speicherplatz) keine Rekursion verwenden darf.

12.4 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" und sind auch in Ada enthalten.)

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 Programmier-sprachen 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 12 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 unterbrochen werden, so dass die Standard-verfahren zu "inkrementellen Techniken" erweitert wurden: Während des Programmablaufs läuft ein Prozess mit, der nicht erreichbare Objekte aufspürt und in die Freispeicherliste einträgt. Die beiden Prozesse dürfen allerdings nicht gleichzeitig auf gewisse Teile zugreifen ("wechselseitiger Ausschluss"), was der Compiler automatisch in das übersetzte Programm einfügt.