Grundvorlesung Informatik

Universität Stuttgart, Studienjahr 2002/03

~~0. Vorbemerkungen~~ (14.10.02)

~~1. Grundlagen der Programmierung~~ (17.10.02 - 5.5.03)

~~2. Interaktionen~~ (8.5. - 2.6.03)

3. Grundlegende Verfahren (5.6. - 25.7.03)

Klausur (Orientierungsprüfung) am 5. August 2003

Hochschullehrer: Volker Claus, Fakultät 5 "I, E u. I"
Institut für Formale Methoden der Informatik (FMI)

Teil 3 der Grundvorlesung

3. Grundlegende Verfahren

3.1 Speicherverwaltung
3.2 Suchverfahren
3.3 Hashing
3.4 Sortieren
3.5 Algorithmen auf Graphen
3.6 Zufallszahlen

Gliederung des Kapitels

3.1 Speicherverfahren

3.1.1 Keller und Halde
3.1.2 Kellerverwaltung
3.1.3 Haldenverwaltung

3.1.1 Keller und Halde

3.1.1.1 Überblick über die wichtigsten Speicher: Wie in 1.3.4 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, Unterbereich hiervon, records, die nur aus solchen Komponenten bestehen, Felder hierüber von konstanter Länge, access-Datentypen (nur der Zeiger, nicht die hiermit aufgebaute Liste).

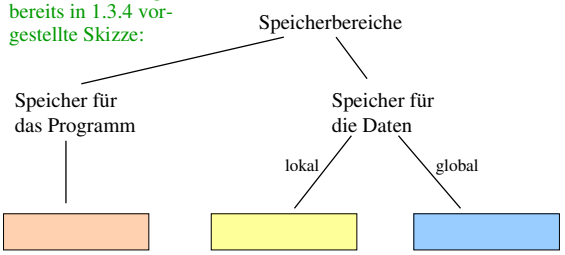
2. Zur Laufzeit beim Abarbeiten der Deklarationen weiterhin erforderlicher dynamischer Speicher (lokalerKeller).

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 1.4.1.4.

Lokaler Speicher in der Praxis:
In der Regel legt man auch die unter 1.2 genannten Daten, die zum statischen Speicher zählen, in diesem Kellerspeicher 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 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 bereits in 1.3.4 vorgestellte 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.

3.1.1.2 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 *sequentiell*; 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 1.3.3.1.

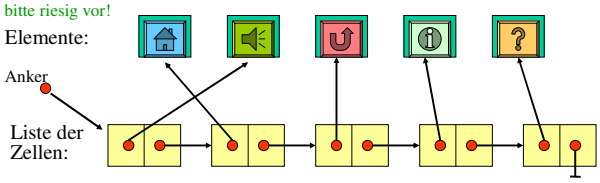
Das Schlüsselwort für Zeiger lautet in Ada access.
Typische Definition einer Liste in Ada:

```
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;
```

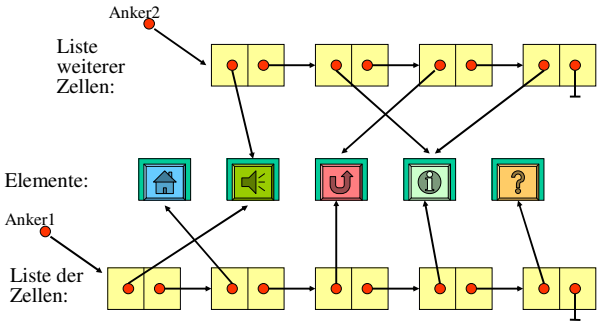
Dies ist eine Liste, in der die Elemente "im Original" stehen. In der Praxis ist dies oft lästig, da ein Element in vielen Listen auftreten kann und dann auch in allen Listen gespeichert und simultan geändert werden muss. Also:


```
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.

Bitte wiederholen aus Informatik I, WS 02/03, 1.3.3.2:

3.1.1.3 Keller (Stapel, Pushdown, Stack) sind Listen mit speziellen zulässigen Operationen, üblicherweise sind dies:

newkeller	-- Leeren des Stacks
isempty	-- Abfrage, ob der Stack leer ist
top	-- Oberstes Element im Stack
push	-- Füge ein Element oben an
pop	-- Lösche das oberste Element
length	-- aktuelle Länge des Stacks

3.1.2 Kellerverwaltung

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

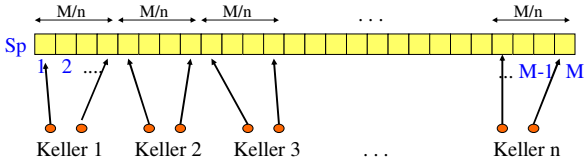
In einem Computer werden gleichzeitig viele Programme ("jobs") verwaltet. Jedes besitzt einen lokalen Keller (die dynamischen Daten werden in der Halde abgelegt, siehe 3.1.3).

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:

3.2.1.1: 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.

Aufgabe: Realisieren Sie diesen Fall selbst!

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

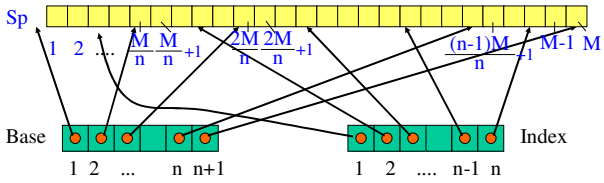
- *Variante 1:* Jeder Keller hat seine eigene maximale Größe, die in Max: array (1..n) of natural abgelegt ist (einfachster Fall: Max(i) = ⌊M/n⌋ für alle i) und für die gilt

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

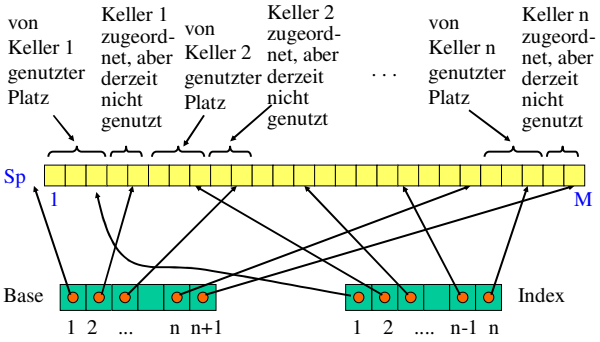
Dieser Fall wird wie "n=1" behandelt, indem überall die Nummer des Kellers hinzugefügt wird und jeder Keller unabhängig von den anderen ist. Es gibt dann zwei Felder für die Adresse vor dem Beginn des i-ten Kellers ("Base") und für seine aktuelle oberste Position ("Index") mit
$0 \le \text{Index}(i)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



3.1.2.3 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 der 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);  ...
```


3.1.2.4 *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
   k: ZK;  j, 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, nur 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;  j, 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;
   if Base(k+1) <= Index(k) + 1 then raise ueberlauf;
   elsif k<i then
      q := (Base(k+1)-Index(k))/2;   -- Hälfte des freien Platzes
      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. (Aber dann ist ohnehin nur noch wenig freier Speicherplatz vorhanden.)

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

3.1.2.5 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)), aber man darf hierfür nur die positiven Werte aufaddieren. 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 der 10% Anteil am freien Speicherplatz, der gleichmäßig jedem Keller zugewiesen wird.
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∗δ$\rfloor$
Wenn ein Keller um δ Plätze gewachsen ist, so erhält er diese w zusätzlichen Speicherplätze, aber nur im Falle δ > 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:

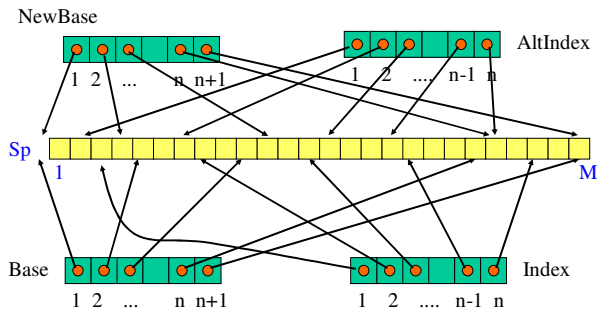
```
NewBase: array (ZK) of Adressen;    -- Neuer Beginn der Kellers
AltIndex: array (ZK) of Adressen;   -- Alter Stand jedes Kellers

procedure umordnen is -- bei Möglichkeit 3 brauchen wir keinen Parameter i
   k: ZK;  j: Adressen;  sum, zuwachs, u, h: integer; v: Float;
   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 (gleichmäßig für alle Keller)
         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;
```


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
a: Adressen; d: integer;
begin d := NewBase(i) - Base(i);
            -- d gibt an, um wieviel Stellen Keller i verschoben werden muss
   if (d /= 0) then
      if d > 0 then
         for a in reverse Base(i) .. 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;
```


3.1.3 Haldenverwaltung

Daten oder Objekte lassen sich durch Zeiger miteinander verflechten. Früher sprach man dann von "Geflechten", die im Speicher aufzubauen sind. Heute bezeichnet man diese Strukturen meist als Vernetzungen oder - mathematisch - als *Graphen* (vgl. 1.3.3.4).

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 "Einführung in die Informatik I", WS 02/03: 1.1.5.7, 1.6.4.4 und der Einschub in 1.6.4.3.

Erläuternder Hinweis:

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

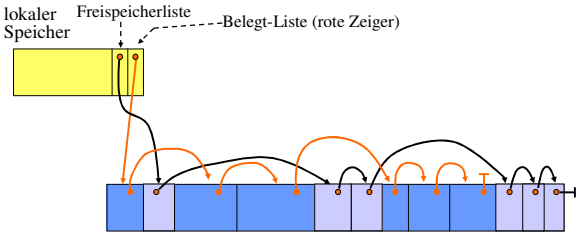
Grund: Man beachte, dass heutige Rechner in der Regel einen ein-dimensionalen Speicher besitzen, auf dessen Speicherplätze über einen Index von 0 bis 2^s-1 (für eine natürliche Zahl s) zugegriffen wird. Einen Zeiger implementiert man daher als die Adresse derjenigen Speicherzelle, ab der das Objekt,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

3.1.3.1 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 über eine Freispeicherliste.

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

3.1.3.2 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).

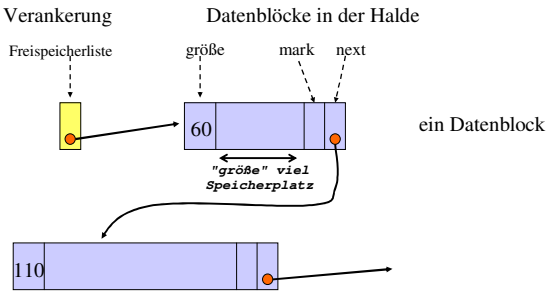
Um diese Aufgaben durchzuführen, muss die Freispeicherliste oft durchlaufen werden, wobei wir die Datenblöcke, die zur Freispeicherliste gehören, markieren, um sie später "erkennen" zu können. Ein Datenblock muss also neben dem Inhalt, den das jeweilige Programm hineinschreibt, mindestens seine Größe,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

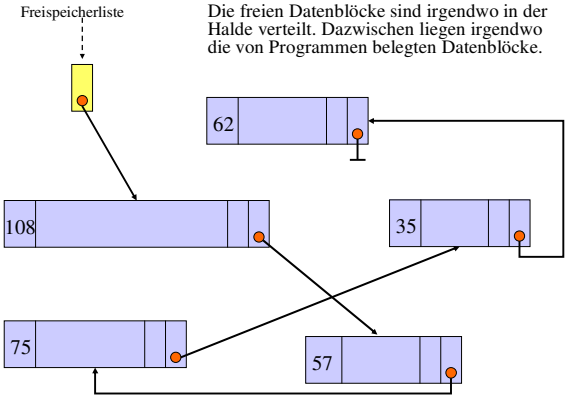
Wir legen daher folgenden Datentyp "DBlock" für Datenblöcke fest. Es sei eine natürliche Zahl "maxgröße" (= die maximale Größe an Speicherplätzen je Datenblock) vorgegeben:

```
type DBlock;
type DBlockzeiger is access DBlock;
type DBlock is record
        größe: 1..maxgröße;
        ... < Komponenten, die insgesamt genau "größe" viele
                Speicherplätze belegen > ...
        mark: Boolean;
        next: Blockzeiger;
        end record;
```

Jeder DBlock belegt also größe + x viele Speicherplätze in der Halde, wobei x die Zahl der Speicherplätze für "größe", "mark" und "next" bezeichnet. (generic-Formulierung in Ada? Selbst!)

Skizze:





3.1.3.3 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 auf früherer Folie).
Füge diese beiden Datenblöcke in die Freispeicherliste anstelle des DBlocks D ein.
Klinke den ersten dieser beiden Datenblöcke aus der Freispeicherliste aus und weise 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 auch 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 schneller.
- Lege einen binären Suchbaum über die Freispeicherliste. Die Suche erfolgt dann viel schneller. Nachteil: Weiterer Speicherplatz. Beachte: Diese Suchbäume müssen ebenfalls in der Halde untergebracht werden, da sie dynamische Datenstrukturen sind.