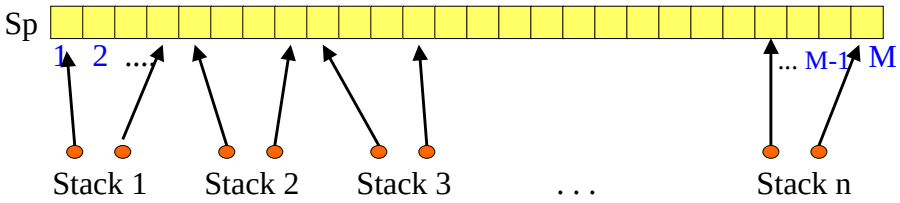
Beispiel 2.3: Implementierung von mehreren Stapeln / Stacks

Aufgabe: Wir wollen eine ***Multistapelverwaltung*** in einem eindimensionalen Feld durchführen, d.h.:

Es sollen n Stacks verwaltet werden. Insgesamt steht hierfür ein linearer Speicher Sp(1..M) zur Verfügung.

Spontane Idee: Jeder Stack erhält gleichviele Speicherplätze M/n:



Diese Verweise werden wir durch Indizes realisieren.

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

```
generic
   M: natural := 2002;                  -- Initialisierung willkürlich
   type element is private;
package stack is
   procedure newstack;                  -- Leeren des Stacks
   function isempty return Boolean;     -- Ist der Stack leer?
   function isfull return Booelan;      -- Ist der Stack voll?
   function top return element;         -- Oberstes Stackelement
   procedure push (x: in element);      -- Füge Element x oben an
   procedure pop;                       -- Lösche oberstes Element
   function length return natural;      -- Aktuelle Stacklänge
   unterlauf, ueberlauf: exception;     -- Ausnahmebehandlung
end stack;
```

Hieran schließt sich der Modulrumpf an:

```
package body stack is
   type speicher is array (1..M) of element;
   Sp: speicher;
   index: integer range 0..M := 0;
   procedure newstack 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 stack;
```


Eine Instanz kann nun lauten:

```
package Ganzzahlkeller is
        new stack (M => 50000, element => integer);
```

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

Aufgabe: Realisieren Sie diesen Fall selbst!

Allgemeiner Fall: **n ≥ 3**. Vorhandener Speicher Sp(1..M).

Hier gibt es mindestens zwei Varianten:

- *Variante 1:* Jeder Stack 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

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

Dieser Fall ist wie der Fall n=1 zu behandeln, indem überall die Nummer des Stacks hinzugefügt wird und jeder Stack unabhängig von allen anderen ist. Beispielsweise muss es dann zwei Felder Base, Index: **array** (1..n) **of** natural geben mit $0 \le \text{Index}(i)-\text{Base}(i) \le \text{Max}(i)$ für i = 1, 2, ..., n. (Details siehe unten.)

- *Variante 2:* Die Größe der einzelnen Stacks ist nicht vorab beschränkt und alle Stacks zusammen sollen den Speicherplatz der Größe M möglichst gut nutzen. Hier muss es ebenfalls zwei Felder Base, Index: **array** (1..n) **of** natural geben, für die zu jedem Zeitpunkt gilt

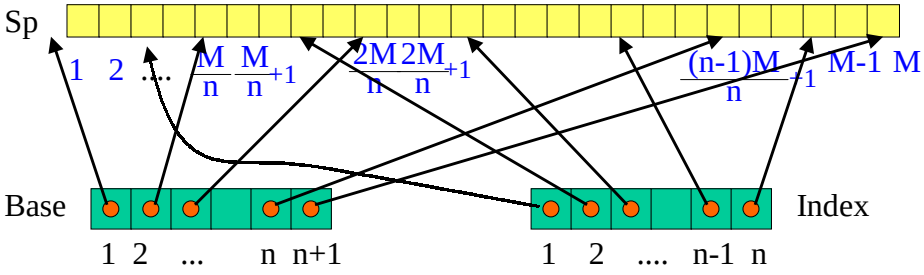
$$\sum_{i=1}^{n} \text{Index}(i)-\text{Base}(i) \le M.$$

Wenn also einer der Stacks überläuft (d.h.: Index(i) = Base(i+1)) und andere Stacks nutzen den ihnen zugewiesenen Bereich noch nicht voll aus, so muss der Speicherplatz neu auf die Stacks verteilt werden.

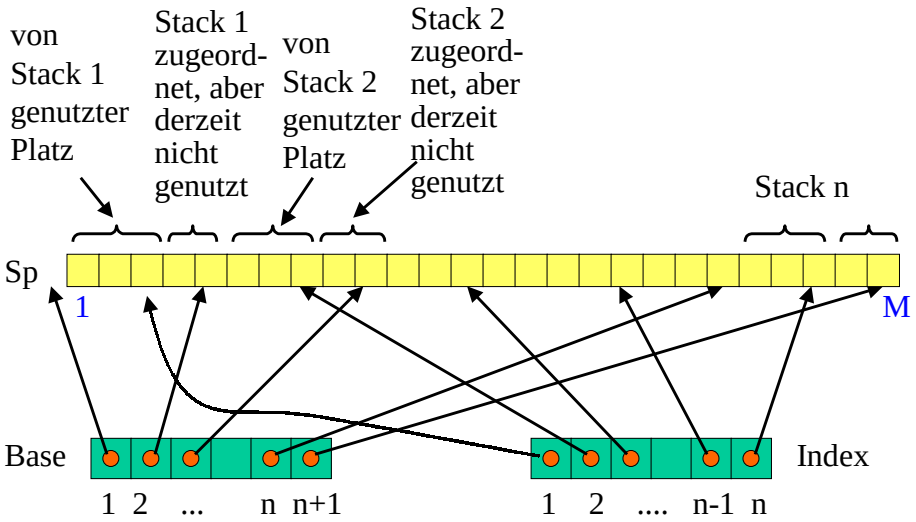
Hier sind wieder mehrere Untervarianten möglich, siehe unten.

Variante 1: Jeder Stack erhält den gleichen Platz der Größe M/n.

Wir realisieren dies über zwei Zeiger bzw. Indizes:
Base(i) zeigt auf den Speicherplatz, der unmittelbar vor dem Bereich für den i-ten Stack liegt;
Index(i) zeigt auf den Speicherplatz, auf dem sich das oberste Element des i-ten Stacks befindet.



Nochmals zur Illustration: Aufteilung des Speichers Sp



Ada Deklarationen hierzu: (MSV = Multistackverwaltung)

```
generic
   M: natural := 20000;            -- willkürlicher Default-Wert
   n: natural;                     -- Anzahl der Stacks, n ≥ 2.
   type Element is private;        -- Datentyp der Stackelemente
package MSV1 is
   type NN is natural range (1..n+1);  -- für Zugriff auf Stacks
   procedure newstack (i:NN);              -- Leeren des Stacks
   function isempty (i:NN) return Boolean; -- Ist der Stack leer?
   function isfull (i:NN) return Booelan;  -- Ist der Stack voll?
   function top (i:NN) return element;  -- Oberstes Stackelement
   procedure push (i: in NN; x: in element);  -- Füge x oben an
   procedure pop (i:NN);           -- Lösche oberstes Element
   function length (i:NN) return natural;  -- Aktuelle Stacklänge
   unterlauf (i:NN), ueberlauf (i:NN): exception;
end MSV1;
```


Paketrumpf hierzu: (MSV = Multistackverwaltung)

```
package body MSV1 is
type Adressen is range 0..M+1;   -- Speicher“adressen“
Sp: array (Adressen) of Element; -- Speicher
Base: array (NN) of Adressen;    -- Beginn der Stacks
Index: array (NN) of Adressen;   -- Aktueller Stand jedes Stacks
procedure newstack (i:NN) is
   begin  Index(i) := Base(i);  end newstack;
function isempty (i:NN) return Boolean is
  begin  return  Base(i) = Index(i);  end isempty;
function isfull (i:NN) return Booelan is
  begin  return  Base(i+1) = Index(i);  end isfull;
function top (i:NN) return element is
  begin  return Sp(Index(i));  end top;
```

Paketrumpf MSV1(Fortsetzung)

```
procedure push (i: in NN; 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:NN) is
  begin  if isempty(i) then raise unterlauf(i);
           else Index(i) := Index(i) - 1; end if; end pop;
function length (i:NN) return natural is
  begin return Index(i)-Base(i); end length;
exception    when ... =>.....
end MSV1;
```


Eine konkrete Instanz könnte dann sein:

```
package Zahlenkellerei is
        new MSV1(M => 50000; n => 10; Element => integer);
use Zahlenkellerei;
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 Multistackverwaltung zusammenbricht, falls irgendein Stack 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:

```
        procedure umordnen (i:NN);  ...
```

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

Möglichkeit 2:
Suche denjenigen Stack j mit maximal viel freiem Platz, d.h., Index(j)-Base(j) ist maximal, und tritt dann die Hälfte dieser Plätze an den Stack i ab. Konkret muss dann der Speicherbereich zwischen den Stacks i und j um q Speicherplätze verschoben werden, wenn q die Hälfte der freien Plätze von Stack j ist.

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

Möglichkeit 1:

```
procedure umordnen (i: NN) is
    k: NN;  j, q: Adressen;
begin  k :=  i;
    if (i=1) and (Index(2) < Base(3)) then k := 2;
    elsif (i=n) and (Index(n-1) < Base(n)) then k := n-1;
    elsif Base(i) - Index(i-1) < Base(i+2) - Index(i+1)
        then k := i+1; else k := i-1; end if;
                -- Stack k dient nun als Platz-Lieferant
    if k=i then raise ueberlauf;
    elsif k<i then
        q := (Base(k+1)-Index(k)+1)/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:

```
procedure umordnen (i: NN) is
   k: NN;  j, q: Adressen;
begin  k := 1;          -- Suche Stack 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) then raise ueberlauf;
   elsif k<i then
      q := (Base(k+1)-Index(k)+1)/2;   -- Hälfte des freien Platzes
      for j in k..i loop
         Base(j) := Base(j) - q; Index(j) := Index(j) - q; end loop;
      for j in Base(k)..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 Stacks ihren Platz kaum benötigen. Denn man prüft ja nur die benachbarten Stacks ab. Auch kann "umordnen" relativ rasch wieder aufgerufen werden.

Nachteil von Möglichkeit 2:
Eventuell wird die Prozedur "umordnen" nach q Schritten erneut aufgerufen.

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

Möglichkeit 3: (Garwick-Algorithmus)

- Berechne den insgesamt freien Platz aller Stacks ("sum").
- Berechne den gesamten Zuwachs seit dem letzten Umordnen.
- Verteile 10% des freien Platzes gleichmäßig an alle Stacks.
- 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 nur die positiven Werte hierbei aufaddieren. NewBase(j) ergibt sich aus den Newbase-Werten der darunter liegenden Stacks erhöht um den festen Anteil u, der jedem Stack zusteht, und dem Zuwachs-Anteil.
Dies ergibt folgende Prozedur "umordnen":

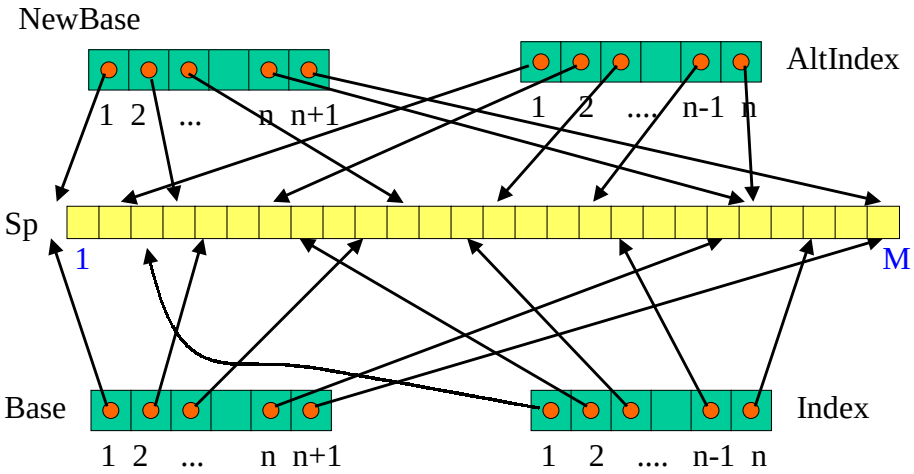
Garwick-Algorithmus: Füge zum "package body MSV1" hinzu:

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

procedure umordnen (i: NN) is
   k: NN;  j: Adressen;  sum, zuwachs, u: integer; v: float;
   Delta: array (NN) of Adressen;   -- Zuwachs jedes Stacks
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
                -- sum=0 wäre zu knapp wegen Rundungsfehlern
   else  zuwachs := 0;   -- ermittle Zuwächse seit letztem "umordnen"
      for j in NN loop Delta(j) := Index(j) - AltIndex(j);
        if Index(j)>Altindex(j) then Delta(j):=Index(j)-AltIndex(j);
                                     zuwachs:=zuwachs+Delta(j);
        else Delta(j) := 0; end if;      end loop;
```

```
if zuwachs >= 1 then
   u := INTEGER(0.1*FLOAT(sum)/FLOAT(n)  +  0.5);
          -- 10% Anteil (gleichmäßig für alle Stacks)
   v := INTEGER(FLOAT(sum) -
                  FLOAT(u)*FLOAT(n))/FLOAT(zuwachs));
else u := INTEGER(FLOAT(sum)/FLOAT(n)); v:=0; end if;
          -- dieser else-Fall darf eigentlich nicht eintreten
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 + Delta (j-1)*v;     end loop;
speicherumordnen;
for j in NN 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: NN;
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;
```



```
procedure verschieben (i: in NN) is  -- Unterprozedur zu speicherumordnen
   a: Adressen; d: integer;
begin d := NewBase(i) - Base(i);
      -- d gibt an, um wieviel Stellen Stack 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;
```