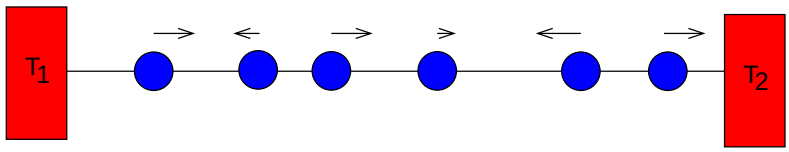
Vorlesung 10 (Freitag 2.3.2018)

9 Ereignisgesteuerte Simulationen

9.1 Eindimensionale Kette harter Teilchen

Modell: "Kette" von n harten Teilchen i mit Masse m_i, Ort x_i, Geschwindigkeit v_i



Wände bei $x = 0/x = L$ mit Wärembädern (Temperatur T_1/T_2).
Wechselwirkung der Teilchen $i.i+1$: idealer Stoß (vorher v_i, nachher v_i')

$$\begin{aligned} v_i' &= \frac{m_i - m_{i+1}}{m_i + m_{i+1}} v_i + \frac{2m_{i+1}}{m_i + m_{i+1}} v_{i+1} \\ v_{i+1}' &= \frac{2m_i}{m_i + m_{i+1}} v_i - \frac{m_i - m_{i+1}}{m_i + m_{i+1}} v_{i+1} \end{aligned}$$

[Selbsttest]

Was passiert wenn alle Teilchen die gleiche Masse haben?

Wechselwirkung mit Wänden:
Geschwindigkeit gemäß "Maxwell-Verteilung" verteilt [8] .

$$P_{1/2}(v) = \theta(\pm v) \frac{mv}{T} \exp(-mv^2/2T_{1/2}) \tag{71}$$

[Selbsttest]

Wie lost man Zufallszahlen gemäß $P_{1/2}$ aus?

Ziel: Untersuchung des Wärmetransports zwischen den Bädern.

9.2 Ereignisse

[Selbsttest]

Wie würden Sie generell das Modell simulieren?
Überlegen Sie 2 Minuten alleine und diskutieren Sie dann mit Ihrem Nachbarn.

9.3 Implementierung

[Selbsttest]

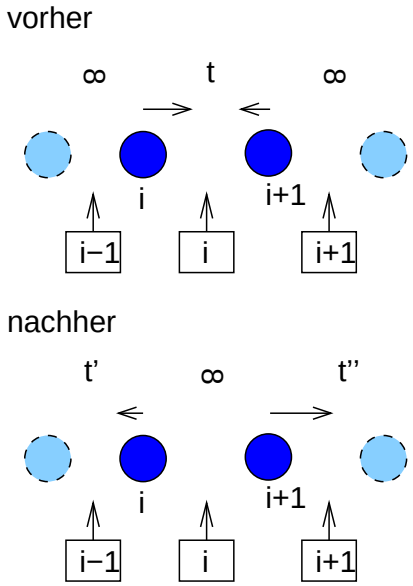
Stellen Sie Vorüberlegungen zur Programmdesign an:
Welche Datenstrukturen braucht man
Welche grundlegenden C-Funktionen muss das Programm beinhalten?

Teilchen:

```
/** data structure for one particle **/
typedef struct
{
  double  m;                              /* mass */
  double  x;       /* position at last collision*/
  double  t;          /* time of last collision */
  double  v;   /* velocity after last collision */
} particle_t;
```

Initialisierung: Teilchen gleichmäßig zwischen $x = 0$ und $x = L$ verteilen, Geschwindigkeiten zufällig in $[-1, 1]$. Speziell: Wände sind Teilchen 0,$n + 1$, bei $x = 0$, $X = L$ ohne Geschwindigkeit.

Ereignisse:
Ereignis i beschreibt den Stoß zwischen Teilchen i und $i+1$. Stoßzeit ="∞", falls kein Stoß.
typische Situation:




```
/** event= particle p1 hits on particle **/
/** p2 at time t                        **/
typedef struct
{
  double   t;                  /* time of event */
} event_t;
```

(Zunächst nur die Stoßzeit, wir später noch erweitert.)
Es wird immer das *nächste* Ereignis ausgeführt → man muß alle Ereignisse durchsuchen und das mit der kleinsten Zeit finden (SPÄTER: bessere Implementierung mit *Heap*).
Abarbeitung eines Ereignisses:
Beim Ereignis i werden Ereignisse $i-1$ und $i+1$ (Sonderfall Wände) neu ausgerechnet, neue Stoßzeit für Ereignis $i =$ "∞.
Routine `treat_event()`

```
/****************** treat_event() *****************/
/** Treat event 'ev' from 'event' array:           **/
/** calculate new velocities of particles ev,ev+1  **/
/** recalculate events ev-1, ev, ev+1              **/
/** PARAMETERS: (*)= return-paramter               **/
/**       glob: global data                        **/
/**       part: data of particles                  **/
/**      event: array of events                    **/
/*          ev: id of event                        **/
/** RETURNS:                                       **/
/**    nothing                                     **/
/**************************************************/
void treat_event(global_t *glob, particle_t *part, event_t *event, int ev)
{
  int pl, pr;               /* particles of collision */
  double vl, vr;           /* velocities of particles */

  pl = ev;
  pr = ev+1;

  part[pl].x += (event[ev].t-  part[pl].t)*part[pl].v;
  part[pr].x += (event[ev].t - part[pr].t)*part[pr].v;
  part[pl].t = event[ev].t;
  part[pr].t = event[ev].t;

  if(pl==0)                 /* collision w. left wall */
  {
    part[pr].v = generate_maxwell(part[pr].m, glob->T1);
    event[pl].t = glob->t_end+1;
    event[pr].t = event_time(pr, pr+1, glob, part);
  }
  else if(pr==(glob->n+1)) /* collision w. right wall */
  {
    part[pl].v = -generate_maxwell(part[pl].m, glob->T2);
    event[pl].t = glob->t_end+1;
    event[pl-1].t = event_time(pl-1, pl, glob, part);
  }
  else
  {
    vl = part[pl].v; vr = part[pr].v;
    part[pl].v = ( (part[pl].m-part[pr].m)*vl + 2*part[pr].m*vr )/
      (part[pl].m + part[pr].m);
    part[pr].v = ( 2*part[pl].m*vl - (part[pl].m-part[pr].m)*vr )/
      (part[pl].m + part[pr].m);
    event[pl-1].t = event_time(pl-1, pl, glob, part);
    event[pl].t = glob->t_end+1;
    event[pr].t = event_time(pr, pr+1, glob, part);
  }
}
```

Achtung: möglicherweise zeitweise KEIN Ereignis für ein Teilchen (weder Stoß rechts noch links), ist aber kein Problem.

9.4 Dichte

Meßgröße: Dichte als Funktion des Ortes. (auch möglich: Wärmeleitung etc)
Realisierung:(`glob.L`= Größe des Systems)

```
double *density;              /* for measuring rho(x) */
int bin, num_bins;
double delta_x;

...

num_bins = 50;
delta_x = glob.L/num_bins;
density = (double *) malloc(num_bins*sizeof(double));
for(bin=0; bin<num_bins; bin++)
  density[bin] = 0;
```

Messung (`part[p]`= Daten für Teilchen `p`, `glob.n`= Anzahl der Teilchen):

```
for(p=1; p<=glob.n; p++)
{
  bin = (int) floor(
     (part[p].x+(t_measure-part[p].t)*part[p].v)/
     delta_x);
  density[bin]+= 1/delta_x;
}
```

Aufbau der Hauproutine. Grobplanung durch *Pseudocode*

algorithm main()
begin
  Initialisierung
  t =erstes Ereignis
  while $t < t_{\text{end}}$
  begin
    Messungen;
    bearbeite Ereignis;
    t =nächstes Ereignis
  end
end

(siehe `main()` in `chain.c`)

Hier: alternierende Massen ($m^a = 1/m^b = 2.6$)
$n = 100$ Teilchen, Laufzeit $t_{\text{end}} = 100$. Messung der Dichte nach der Hälfte der Laufzeit alle 10 Zeiteinheiten. System noch nicht equilibriert:

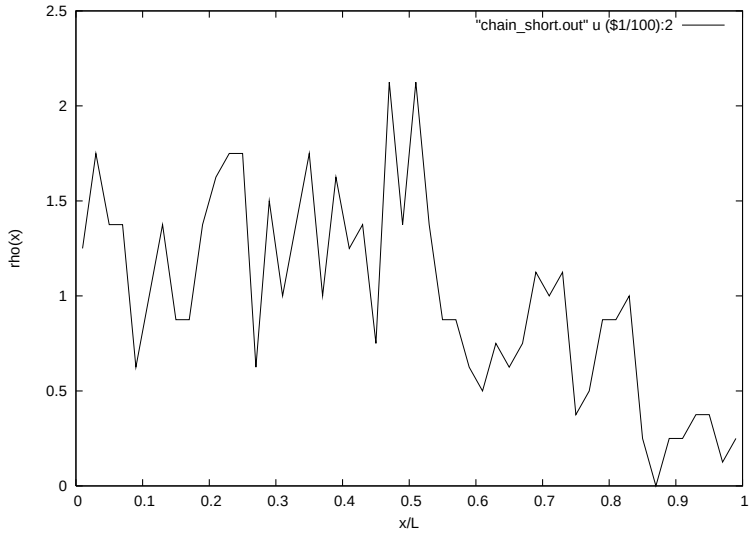


Figure 21: mittlere Dichte als Funktion des Ortes im Zeitintervall $[50, 100]$.

$t_{\text{end}} = 10000$.

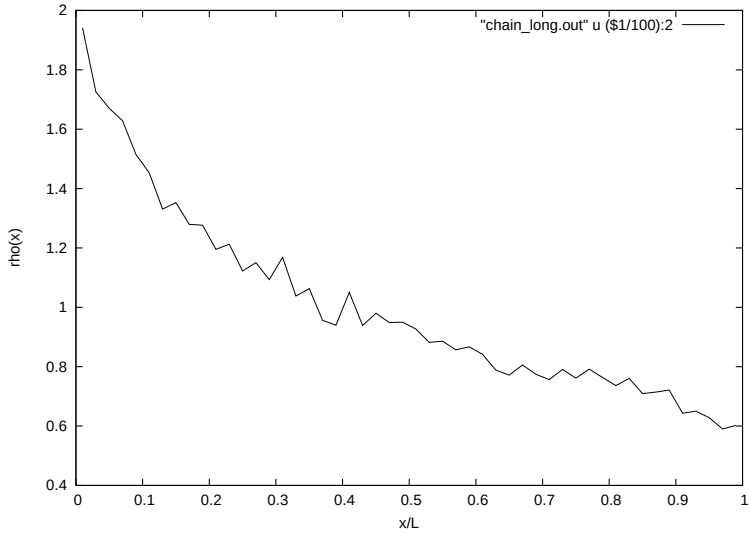


Figure 22: Mittlere Dichte als Funktion des Ortes im Zeitintervall $[5000, 10000]$.

Dicht geringer, dort wo die Temperatur höher ist. Weitere Ergebnisse siehe [8].

9.5 Heaps

Laufzeit des Programms:

Anzal der Stöße pro Zeiteinheit: $O(n)$
Suche des nächsten Ereignisses: $O(n)$
$\Rightarrow O(n^2) =$ "langsam".

Verbesserung: $O(n \log n)$, wenn man einen Heap verwendet.

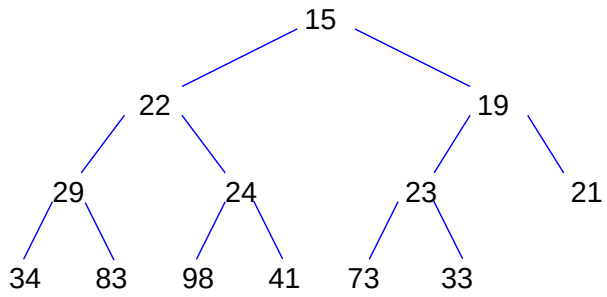
Vorschau:
Lauzeitbeispiel: $n = 500, t_{\text{end}} = 10000$.

```
time chain 500 10000
21.36user 0.07system 0:21.70elapsed 98%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (133major+20minor)pagefaults 0swaps
```

```
time chain_heap 500 10000
7.92user 0.01system 0:08.08elapsed 98%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (133major+23minor)pagefaults 0swaps
```

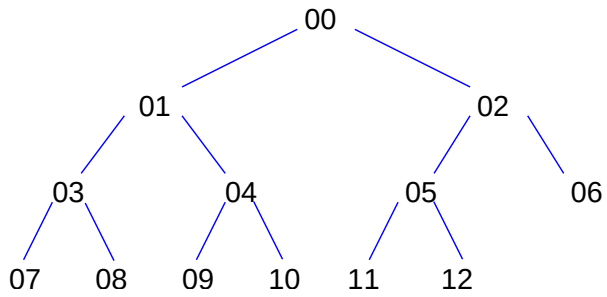
mit Heap → schnellere Programm → größere Systeme ($n = 16383$ zu $n = 1281$)
→ verlässlichere, ANDERE Ergebnisse [9] (Crossover).

Heap = teilgeordneter Baum, der für jeden Unterbaum das (hier) kleinste Element an der Wurzel stehen hat
→ jedes Element ist kleiner als seine Söhne
Bsp:



Damit: das erste Element ist IMMER das kleinste, also z.B. das nächste Ereignis → schneller Zugriff ($O(1)$).

Für Heaps: effiziente Realisierung als Array:



Knoten i:
Vater: $(i-1)/2$ (`int` Operation)
linker Sohn: $2i+1$
rechter Sohn: $2i+2$

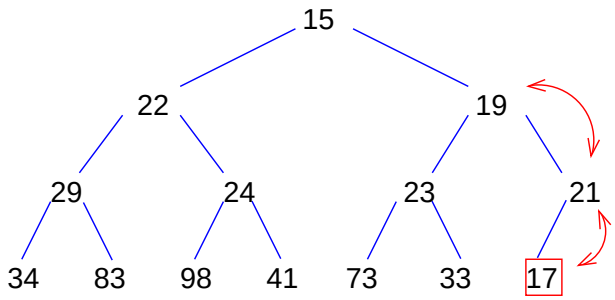
Grundlegende Heap Operationen:
Einfügen:

algorithm heap_insert()
begin
    füge Element am Ende hinzu;

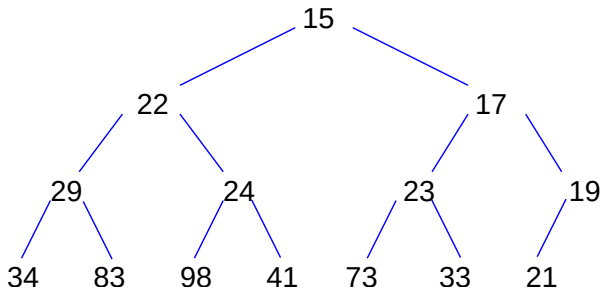
while (Element kleiner als Vater)
vertausche mit Vater;
end

(siehe `heap_insert()` in `chain_heap.c`)

Bsp: Einfügen von "17"



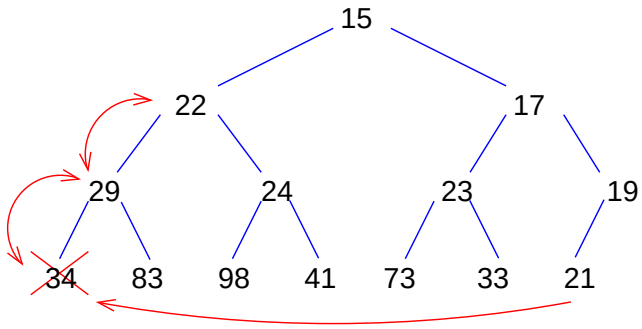
ergibt



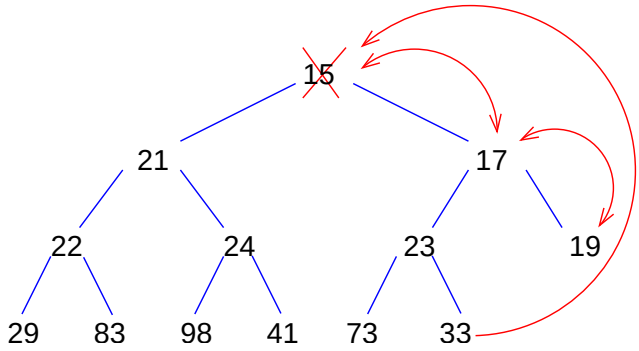
maximal ein Durchlauf von einem Blatt zur Wurzel
$\rightarrow$ Zeit $O(\log N)$
Entfernen:

algorithm heap_remove()
begin
ersetze Element durch letztes Element;
if (Element kleiner als Vater) **then**
while (Element kleiner als Vater)
vertausche mit Vater;
else
while (Element größer als ein Sohn)
vertausche mit kleinerem Sohn;
end

Fall A:



Fall B:



→ Zeit $O(\log N)$
Implementierungshinweis: Es werden auch Ereignisse mitten aus dem Heap entfernt (wenn sich die Zeiten benachbarter Ereignisse ändern).
→ damit das schnell geht, wird für jedes Ereignis in dem nach Ort geordnetem Ereignis Array auch seine Position im Heap gespeichert. (siehe Typ `heap_elem_t` in `chain_heap.c`). Diese Position muß bei jeder Verschiebung im Heap mit aktualisiert werden. (Ohne diese Abspeicherung müsste wieder der ganze Heap durchsucht werden, wenn ein Ereignis mitten aus dem Heap entfernt wird → wieder $O(N)$). Solche "Doppelverweise" (hier Heap → Array, Array → Heap) sind oft nötig, wenn man effiziente Programme schreiben will.
(siehe `heap_remove()` in `chain_cheap.c`)

Zugriff auf das erste Element im Heap: $O(1)$ (im Vergleich zu $O(N)$ bei der einfachen Implementierung).
→ Gesamtlaufzeit $O(N \log N)$.

References

[1] A. M. Ferrenberg, D. P. Landau, and Y. J. Wong. Monte Carlo simulations: Hidden errors from "good" random number generators. *Phys. Rev. Lett.*, 69:3382, 1992.

[2] B.J.T. Morgan. *Elements of Simulation*. Cambridge University Press, Cambridge, 1984.

[3] W. H. Press, S. A. Teukolsky, W.T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, Cambridge, 1995.

[4] A. K. Hartmann. *Practical Guide to Computer Simulations*. World Scientific, Singapore, 2009.

[5] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biophys.*, 5:115–133, 1943.

[6] D. Hebb. *Organisation of Behavior*. Wiley, New York, 1949.

[7] A. C. Maggs and V. Rossetto. Local simulation algorithms for coulomb interactions. *Phys. Rev. Lett.*, 88(19):196402, 2002.

[8] A. Dhar. Heat conduction in a one-dimensional gas of elastically colliding particles of unequal masses. *Phys. Rev. Lett.*, 86:3554, 2001.

[9] P. Grassberger, W. Nadler, and Lei Yang. Heat conduction and entropy production in a one-dimensional hard-particle gas. *Phys. Rev. Lett.*, 89:180601, 2002.