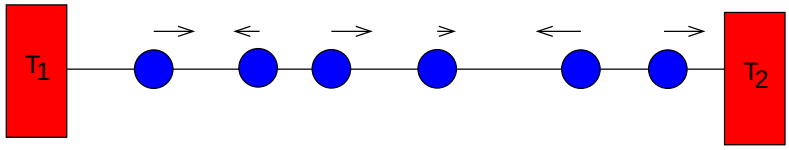
Block 9 (Dienstag 27.2.2024)

10 Event-driven Simulations

VIDEO: `video09a_ereignis_modell` | VIDEO: `video09a_ereignis_zeit`
VIDEO: `video09a_ereignis_bearbeitung`

10.1 One-dimensional chain of hard particles

Model: "Line" of n hard particles i wth mass m_i, positions x_i, velocities v_i



Walls at $x = 0/x = L$ with heat baths (temperatures T_1/T_2).
Interaction of particles $i.i+1$: ideal collision (before v_i, after v_i')

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

[Activator]

What happens if all particles have the same mass?

Interaction with walls :
velocities according to "Maxwell distribution" [5] .

$$P_{1/2}(v) = \theta(\pm v) \frac{mv}{T_{1/2}} \exp(-mv^2/2T_{1/2}) \tag{50}$$

[Activator]

How does one draw random numbers according to $P_{1/2}$?

Aim: investigate heat flow between baths.

10.2 Events

[Activator]

How would you simulate the modell?
Think about the question for 2 minutes, then discuss with your bench neighbor.

For each particle: store position $x_i(t_i)$ at previous collision at time t_i

$$\begin{aligned} x_i(t) &= x_i(t_i) + (t - t_i)v_i \\ x_{i+1}(t) &= x_{i+1}(t_{i+1}) + (t - t_{i+1})v_{i+1} \end{aligned} \tag{51}$$

At collision: $x_i(t^*) = x_{i+1}(t^*) \Rightarrow$
collision time:

$$t^* = \frac{(x_i(t_i) - t_i v_i) - (x_{i+1}(t_{i+1}) - t_{i+1}v_{i+1})}{v_{i+1} - v_i}$$

10.3 Implementation

[Activator]

Perform preliminary considerations about the program design:
Which data structures do you need?
Which fundamental functions do you have to implement?

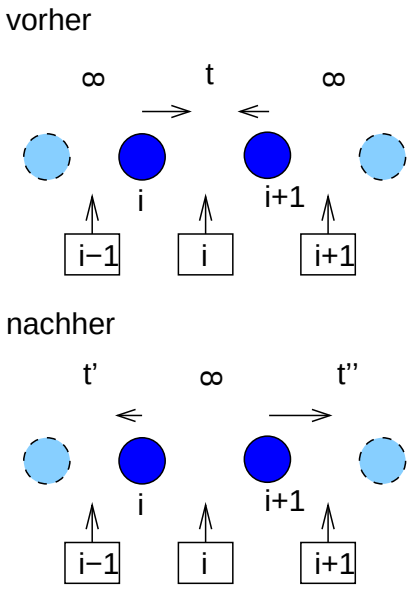
Particles:

```
/** 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;
```

Initialisation: distribute particles uniformly between $x = 0$ and $x = L$, velocities randomly in $[-1, 1]$. Special: walls are particles 0, $n + 1$, at $x = 0$,

$X = L$ with no velocities.

Events:
Event i describes collision between particles i and $i+1$. If no collision currently: collision time ="∞".
typical situation:




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

(Contains so far just the collision time, will be extended later on.)
Each step, only the *next* event is treated $\rightarrow$ one has to search all events to find the one with the smallest time (LATER: better implementation with *heap*).
Treatment of an event :
For event i the "neighboring" events $i-1$ and $i+1$ (special case: with walls) are recomputed, also new collision time for event i = "∞.
Function `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);
  }
}
```

Attention: possibly no event for a particle (neither collision with left nor with reight neighbor), but is no problem.
VIDEO: `video09a_ereignis_dichte`

10.4 Density

Meaure quantity: density as a function of position (one can also measure heat conduction etc)
Realisation: (`glob.L`= size of system)

```
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;
```

Measurement (`part[p]`= data for particle `p`, `glob.n`= number of particles):

```
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;
}
```

Structure of main function. Plan with *Pseudocode*

algorithm main()
begin
  initialisation
  t =first event
  while $t < t_{\text{end}}$
  begin
    measurements
    treat event
    t =next event

```
    end
end
```

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

Here: alternating masses ($m^a = 1/m^b = 2.6$)
$n = 100$ particles, run time $t_{\text{end}} = 100$. Measurement of density after half of model time every 10 time units. Result: system not yet in steady state:

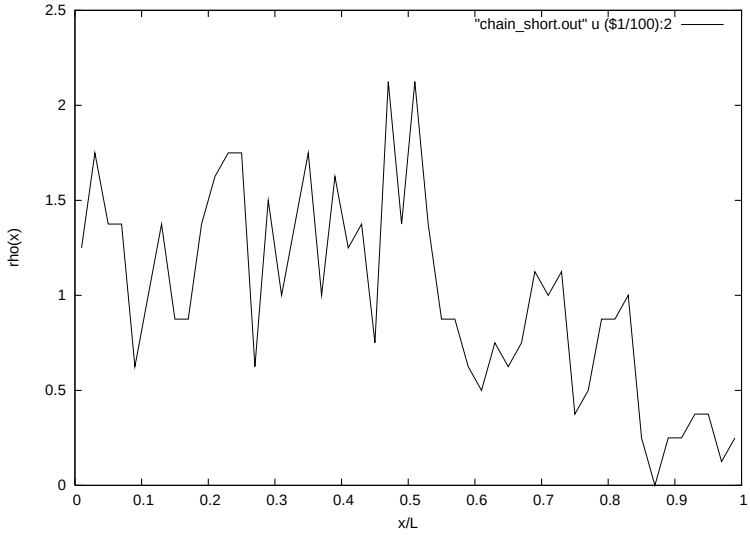


Figure 17: Average density as function of position in time interval $[50, 100]$.

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

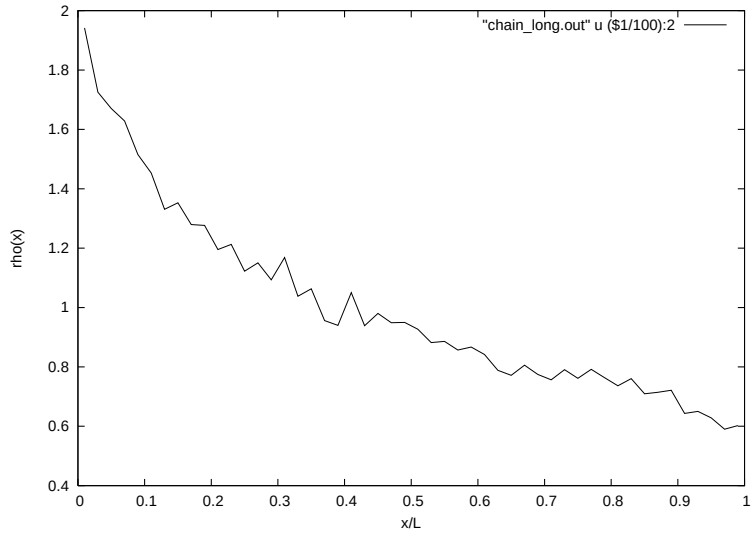


Figure 18: Average density as function of position in time interval $[5000, 10000]$.

Density is smaller where temperature is higher. More results see A. Dahr, Phys. Rev. Lett. **86**, 3554 (2001) [5].
VIDEO: `video09a_ereignis_heaps`

10.5 Heaps

Run time of programs:

Number of collisions per time unit: $O(n)$
Search for next event: $O(n)$
$\Rightarrow O(n^2) =$ "slow".

Improvement: $O(n \log n)$, when using heaps.

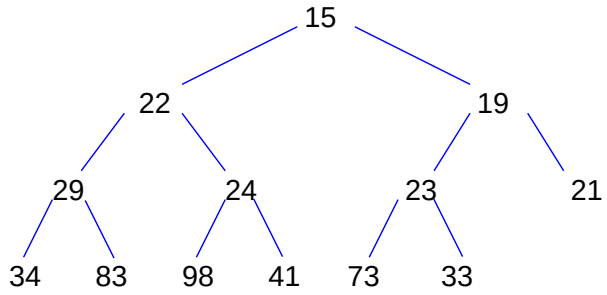
Preview:
Run time example: $n = 500, t_{\text{end}} = 10000$.

```
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
```

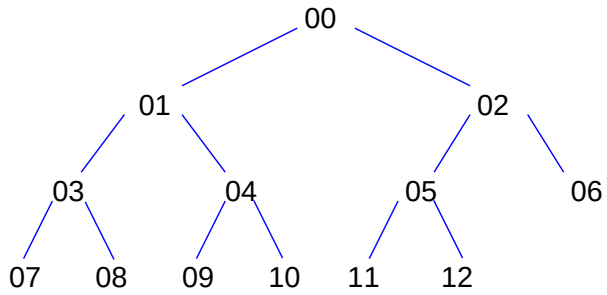
with heap → faster program → larger systems ($n = 16383$ vs. $n = 1281$)
→ more reliable, DIFFERENT results (Crossover), see P. Grassberger et al., Phys. Rev. Lett **89**, 180601 (2002) [6].

Heap = partially ordered tree, where for each sub tree the (here) smallest element is located at the root of the sub
→ jeach element is smaller than all decendants
Example:



Thus: the "top" root element is ALWAYS the smallest of all, e.g., the one containing the next event → faster access ($O(1)$).

For heaps: efficient realisation as array:



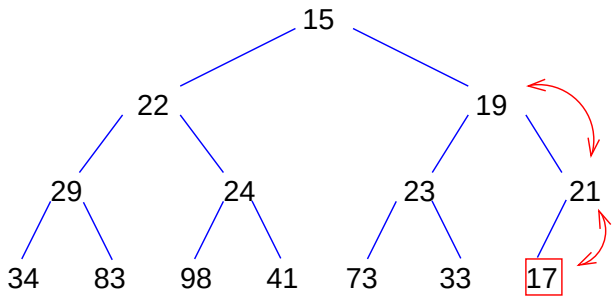
node i:
predecessor: $(i-1)/2$ (`int` division)
left descendant: $2i+1$
right descendant: $2i+2$

Basic heap operations:
Insert:

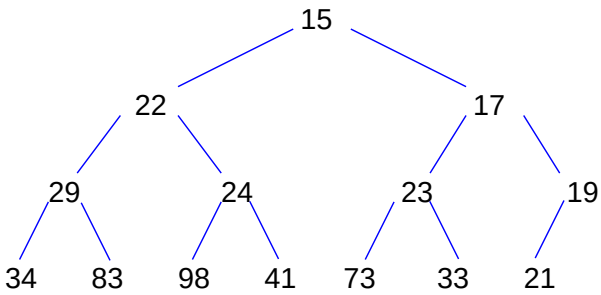
algorithm heap_insert()
begin
  add element at the end;
  while (element smaller than predecessor)
    exchange with predecessor;
end

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

Eaxmple: insert "17"



results in



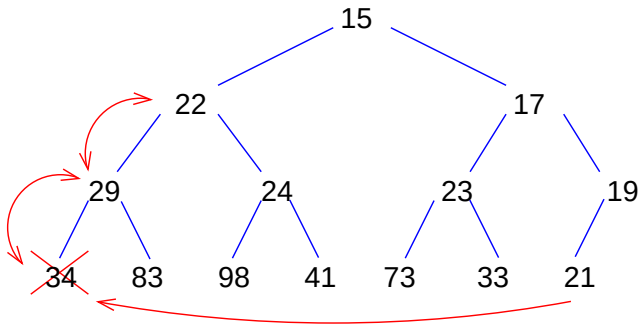
At most one sweep from leaf to root
$\rightarrow$ time $O(\log N)$

Removal:

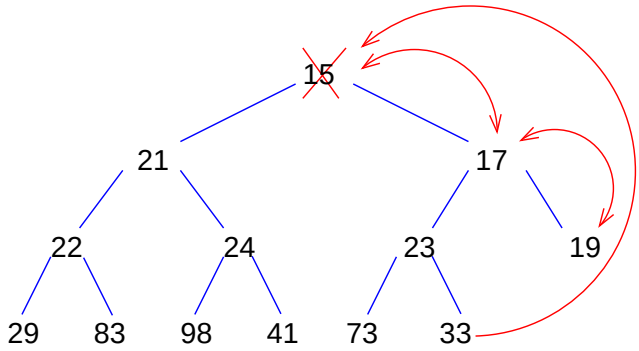
algorithm heap_remove()
begin
  replace element by last element;
  if (element smaller than predecessor) **then**
    while (element smaller than predecessor)
      exchange with predecessor;
  else
    while (element larger than a desendant)

exchange with the smaller descendant;
end

Case A:



Case B:



→ time $O(\log N)$
Implementation hints: events are removed also from within the heap (if velocities change).
→ to make it faster, for each event its current heap position is stored in the array of events (see type `heap_elem_t` in `chain_heap.c`). This position has to be updated if an event moves inside the heap! (Without this additional storage, one would have to search the full heap to find an arbitrary element → again $O(N)$ run time).
Such "double storage " (here heap → array, array → heap) is often needed to obtain efficient programs.
(see `heap_remove()` in `chain_cheap.c`)

Access to first element in heap: $O(1)$ (in contrast to $O(N)$ for the simple implementation).
→ total run time $O(N \log N)$.