Block 4 (Montag 19.2.2024)

5 Info exercises

The exercises take place on-site, in room W1 0-008. There are computers with Linux. You can also use your own laptops with Linux if they are prepared to have Shell, Editor, Compiler, gnuplot, gdb, (preferentially) valgrind. It is not recommended to use powerful and sophisticated development environments, because they create a lot of overhead for just a small benefit.
You can participate as single persons or in two-person groups, two-person groups are recommended.
The exercises are "in-time": The exercise sheets and the material are provided right before start in StudIP. The exercises are completed during the course time and graded at the end of each exercise.
General scheme: the exercises are typically based on partially written programs. You have to complete the programs, only sometimes to write full programs. Then you have often to perform simulations, and sometimes analyse or plot the data.
Towards the end of an exercise, you have to present in front of the screen your own code to the teacher, as well as the results of the simulations. All group members have to contribute to the presentation. The grading will result in up to 10 points, all group members receive the same number of points.
Typically, the simulations rely on a correct code. Thus, in the *worst case*, you might ask the teacher to look into your code and he/she try to find mistakes, which means the teacher also works on the exercise, without guarantee of success. Naturally, this leads to the decrease of the number of points, even if all tasks are completed in the end! Clearly, you are always allowed to ask general questions about programming, how to look for mistakes etc, without reduction of the number of points.
There are 8 exercises, each lasting about 4h, from 14:15 until (ca.) 17:45, depending how long the grading takes.

6 Algorithms

VIDEO: `video04a_complexity_r`

6.1 Easy, hard and unsolvable problems

$t(x)$= run time of a program $\mathcal{P}$ with input x. In theoretical Computer Science: run time is evaluated for model computers (e.g. *Turing Machines*).

$n = |x|$= "size" of input (e.g. number of bits needed to code the problem). *Time complexity* of a program $\mathcal{P}$ = slowest (*worst case*) run time as function of n:

$$T(n) = \max_{x:|x|=n} t(x) \tag{1}$$

Example: Run time

$T(n)$	$T(10)$	$T(100)$	$T(1000)$
$f(n)$	23000	46000	69000
$g(n)$	1000	10000	100000
$h(n)$	100	10000	1000000

□

[Selbsttest]

Which program is (asymptotically) the slowest?

O notation
$T(n) \in O(g(n))$: $\exists c > 0, n_0 \geq 0$ with $T(n) \leq cg(n)\ \forall n > n_0$. "$T(n)$ is at most of order $g(n)$."

Typical time complexities:
Polynomial problems (class P) are considered as "easy", exponential problems as "hard". Some some important problems: no polynomial algorithm is *known*. So far, there is no proof that there are no polynomial algorithms. *NP-hard* problems, run actually in polynomial time on a *non-deterministic*

Table 1: Growth of functions dependent on the input size n.

$T(n)$	$T(10)$	$T(100)$	$T(1000)$
n	10	100	1000
$n \log n$	10	200	3000
n^2	10^2	10^4	10^6
n^3	10^3	10^6	10^9
2^n	1024	1.3×10^{30}	1.1×10^{301}
$n!$	3.6×10^6	10^{158}	4×10^{2567}

(model) computer.

Decision problems: Only output "yes"/"no". Example: Is the ground state energy of a system $\leq E_0$ (E_0: given parameter value)?

For some problems one can prove that they are *undecidable*, i.e., there is *no general* algorithm, which answers "yes" or "no" for all possible inputs=problem instances. But such problems are often provable, i.e. one can prove one of the possible answers, but not both.

- Halting problem: Does a given (arbitrary) program stop after a finite time for a given (arbitrary) input? (If it stops it is easy to prove: let it run and wait)
- Correctness problem: Does a given (arbitrary) program created the desired output for *all* possible inputs (If not, it is "easy" to prove: let it run for an input where the output is not correct)

There are (academic) functions which are even *not computable*. Proof by diagonalization, works like Cantor's proof that there are non-rational numbers.
VIDEO: `video04b_rekursion_r`

6.2 Recursion and Iteration

Principle of *Recursion*: Subroutines calls itself. Natural approach for recursive function definitions.

Example: factorial $n!$:

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n \times (n-1)! & \text{else} \end{cases} \tag{2}$$

Simple translation to C function

```
double factorial(int n)
{
  if(n==1)
    return(1.0);
  else
    return(n*factorial(n-1));
}
```

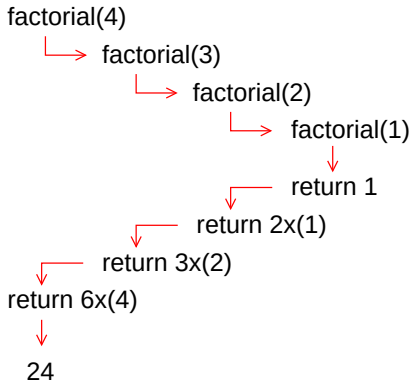


Figure 1: Hierarchy of recursive calls for the calculation of factorial(4).

Analysis of run time of `factorial()` using *recurrence equations*:

$$\tilde{T}(n) = \begin{cases} C & \text{for } n = 1 \\ C + \tilde{T}(n-1) & \text{for } n > 1 \end{cases} \tag{3}$$

Solve via differential equation: $\frac{d\tilde{T}}{dn} = \frac{\tilde{T}(n)-\tilde{T}(n-1)}{n-(n-1)} = C \rightarrow \tilde{T}(n) = CN + K$, with $\tilde{T}(1) = C \rightarrow$
Solution: $\tilde{T}(n) = Cn$, i.e. the run time is linear in n (fun fact: but exponentially in the length of input (=number of bits), because $n = 2^{\log_2 n}$).

[Selbsttest]

Propose an iterative calculation of the factorial by a loop. How is the time complexity?

VIDEO: `video04c_dvidide_and_conquer_r`

6.3 Divide-and-conquer (Divide and conquer)

[Selbsttest]

Think for yourself for 5 minutes about an algorithm (just the basic idea) which sorts the elements $\{a_0, \ldots, a_{n-1}\}$ "in increasing order".
Next, discuss your solution for five minutes with your bench neighbor.
What asymptotic run time $T(n)$ do you expect for your algorithm?

ATTENTION: Do not read on before you have thought about the problem!

Here: Principle (relies on recursion):

1. reduce problem to some smaller sub problems
2. solve all sub problems
3. construct the solution of the problem by using the solutions of the sub problems.

Example: Sorting of a set of elements $\{a_0, \ldots, a_{n-1}\}$ by *Mergesort*.
Basic idea: divide set into two sets of equal size, sort both sets and merge them such that the result is sorted.

```
/** sorts 'n' integer numbers in the array 'a' in ascending **/
/** order using the mergsort algorithm                      **/
void my_mergesort(int n, int *a)
{
  int *b,*c;                                    /* two arrays */
  int n1, n2;                      /* sizes of the two arrays */
  int t, t1, t2;                            /* (loop) counters */

  if(n<=1)                           /* at most one element ? */
    return;                                   /* nothing to do */
  n1 = n/2; n2 = n-n1;                /* calculate half sizes */

  /* array a is distributed to b,c. Note: one could do it   */
  /* using one array alone, but yields less clear algorithm */
  b = (int *) malloc(n1*sizeof(int));
  c = (int *) malloc(n2*sizeof(int));
  for(t=0; t<n1; t++)                              /* copy data */
      b[t] = a[t];
  for(t=0; t<n2; t++)
      c[t] = a[t+n1];

  my_mergesort(n1, b);                /* sort two smaller arrays */
  my_mergesort(n2, c);


  t1 = 0; t2 = 0;     /* assemble solution from subsolutions */
  for(t=0; t<n; t++)
    if( ((t1<n1)&&(t2<n2)&&(b[t1]<c[t2]))||
        (t2==n2))
      a[t] = b[t1++];
    else
      a[t] = c[t2++];

  free(b); free(c);
}
```

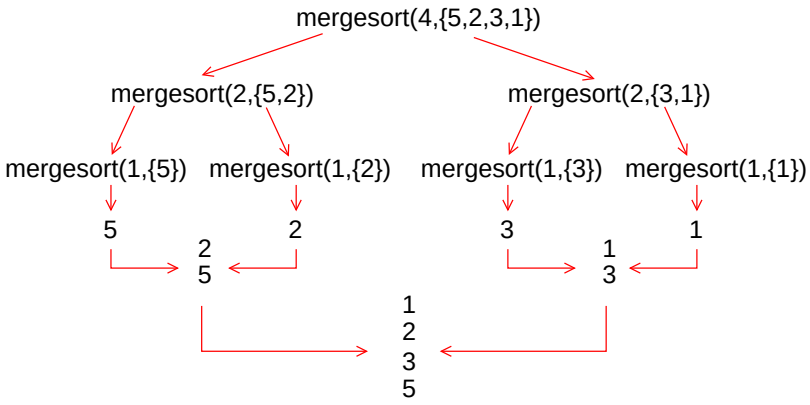


Figure 2: Aufruf von `mergesort`$(4, \{5, 2, 3, 1\})$.

Run time : division of set and reassemble: $O(n)$; recursive calls: $T(n/2)$.
Recurrence:

$$T(n) = \begin{cases} C & (n = 1) \\ Cn + 2T(n/2) & (n > 1) \end{cases} \tag{4}$$

Solution for large n: $T(n) = \frac{C}{\log 2} n \log n$. Proof by insertion

$$\begin{aligned} T(2n) &= C2n + 2T(2n/2) \\ &= C2n + 2(\frac{C}{\log 2} n \log n) \\ &= \frac{C}{\log 2} 2n \log 2 + \frac{C}{\log 2} 2n \log n \\ &= \frac{C}{\log 2} 2n \log(2n) \end{aligned}$$

VIDEO: `video04d_dynamic_programming_r`

6.4 Dynamic Programming

Fibonacci numbers:

$$\text{fib}(n) = \begin{cases} 1 & (n = 1) \\ 1 & (n = 2) \\ \text{fib}(n-1) + \text{fib}(n-2) & (n > 2) \end{cases} \tag{5}$$

E.g. $\text{fib}(4) = \text{fib}(3) + \text{fib}(2) = (\text{fib}(2) + \text{fib}(1)) + \text{fib}(2) = 3$,
$\text{fib}(5) = \text{fib}(4) + \text{fib}(3) = 3 + 2 = 5$.
Grows very quickly: $\text{fib}(10) = 55$, $\text{fib}(20) = 6765$, $\text{fib}(30) = 83204$, $\text{fib}(40) > 10^8$

Number of calls for a recursive implantation grows also exponentially with n, even faster than the function itself!

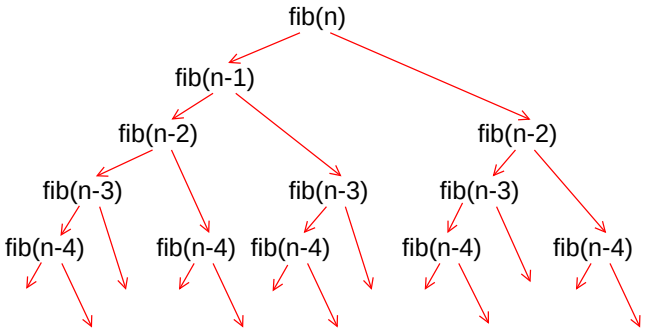


Figure 3: Hierarchie der Aufrufe für fib(n).

[Selbsttest]

How can you find a faster algorithm?

ATTENTION: Do not read on before you have thought intensively about the problem

Better: dynamic programming. Principle: calculate solutions for smaller problems, store them, and use them iteratively (not recursively) to obtain solutions for larger problems.

```
/** calculates Fibonacci number of 'n' dynamically **/
double fib_dynamic(int n)
{
  double *fib, result;
  int t;
  if(n<=2)                                    /* simple case ? */
    return(1);                          /* return result directly */
  fib = (double *) malloc(n*sizeof(double));
  fib[1] = 1.0;                       /* initialise known results */
  fib[2] = 1.0;

  for(t=3; t<n; t++)          /* calculate intermediate results */
    fib[t]=fib[t-1]+fib[t-2];

  result = fib[n-1]+fib[n-2];
  free(fib);
  return(result);
}
```

[Selbsttest]

What is the run time of the algorithm?
Competition: what is the largest value n, such that fib(n) results in a a finite output value: each one has to give an estimation.

Even faster: Formula

$$\text{fib}(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right) \tag{6}$$

Block 5 (Dienstag 20.2.2024)

VIDEO: `video04e_backtracking_r`

6.5 Backtracking

Basic idea: if you cannot calculate directly a solution: try systematically (all) different possible solutions. HOW?

Backtracking: Take one decision (assignment of variables etc) after the other. If already taken decisions prevent a solution → withdraw from some decisions in a systematic way and try other possibilities.

Example: N Queens Problem

N queens shall be placed on a $N \times N$ chess board such that no queens checks against any other queen. This means that in each column, each row and each diagonal at most one queen is placed.

□

[Activator]

Please think for yourself for 4 minutes about a suitable algorithm (basic idea) which solves the N Queens Problem.
Next, discuss for 3 minutes with your bench neighbor about your solutions.

ATTENTION: Do not read beyond this point before you thought about the algorithm.

Basic idea of the approach: put on each column `c` in a systematic way one queen at position (row) (`pos[c]`, $c = 0, \ldots, N-1$). If this does not lead to a solution, then *backtrack*.
Additional variables for occupation of rows and diagonals. This need additional memory, which is in theory redundant but makes test faster whether rows or diagonals are available.
Since $x, y = 0, \ldots, N-1 \rightarrow$ downward diagonals: $x + y \in 0, \ldots, 2N-2$, upward diagonals: $x - y \in -N+1, \ldots, N-1$ (for C Array: add $N-1$ to start at 0)

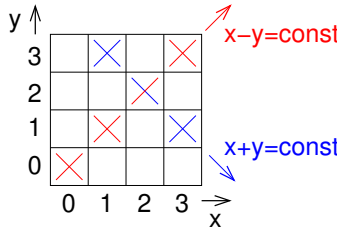


Figure 4: Test variables, whether there are queens placed in the diagonals.

```
void queens(int c, int N, int *pos, int *row,
            int *diag_up, int *diag_down)
{
  int r, c2;                                    /* loop counters */
  if(c == -1)                                 /* solution found ? */
  {
      /* omitted here */                        /* print solution */
  }
  for(r=N-1; r>=0; r--)   /* place queen in all rows of column c */
  {
    if(!row[r]&&!diag_up[c-r+(N-1)]&&!diag_down[c+r]) /*  place ?*/
    {
      row[r] = 1; diag_up[c-r+(N-1)] = 1; diag_down[c+r] = 1;
      pos[c] = r;
      queens(c-1, N, pos, row, diag_up, diag_down);
      row[r] = 0; diag_up[c-r+(N-1)] = 0; diag_down[c+r] = 0;
    }
  }
  pos[c] = 0;
}
```

Initially: `pos[i]=row[i]=diag_down[i]=diag_up[i]=0` for all `i` and call : `queens(N-1,N,pos,row,diag_up,diag_down)`.

[Activator]

Solve the 4×4 problem. How many solutions do exist?

If one counts for small values of N the number of solutions $\rightarrow$ grows exponentially as function of N.

VIDEO: `video05a_lists_r`

7 Advanced Data Structures

To perform elementary operations (store, search, read out and delete of data).

By using sophisticated data structures: faster simulations, thus, usually larger systems can be treated.

7.1 Lists

Lists = generalizations of arrays: have also linear order but more flexible
Example: Removal of array elements $O(N)$, (linked) lists: $O(1)$.

Here: single-connected lists. Data structure:

```
/* data structures for list elements */
struct elem_struct
{
    int                 info;               /* holds ‘‘information’’ */
    struct elem_struct *next;  /* points to successor (NULL if last) */
};

typedef struct elem_struct elem_t;   /* define new type for elements */
```

Double-linked lists: have also an entry `struct elem_struc *prev`
Generation and deletion of elements:

```
/****************** create_element() ****************/
/** Creates an list element an initialized info      **/
/** PARAMETERS: (*)= return-paramter                 **/
/**        value: of info                            **/
/** RETURNS:                                         **/
/**     pointer to new element                       **/
/*******************************************************/
elem_t *create_element(int value)
{
  elem_t *elem;

  elem = (elem_t *) malloc (sizeof(elem_t));
  elem->info = value;
  elem->next = NULL;
  return(elem);
}

/****************** delete_element() ****************/
/** Deletes a single list element (i.e. only if it   **/
/** is not linked to another element)                **/
/** PARAMETERS: (*)= return-paramter                 **/
/**         elem: pointer to element                 **/
/** RETURNS:                                         **/
/**    0: OK, 1: error                               **/
/*******************************************************/
int delete_element(elem_t *elem)
{
  if(elem == NULL)
  {
    fprintf(stderr, "attempt to delete ‘nothing‘\n");
    return(1);
  }
  else if(elem->next != NULL)
  {
    fprintf(stderr, "attempt to delete linked element!\n");
    return(1);
  }
  free(elem);
  return(0);
}
```

Actual access to list = pointer to first element:

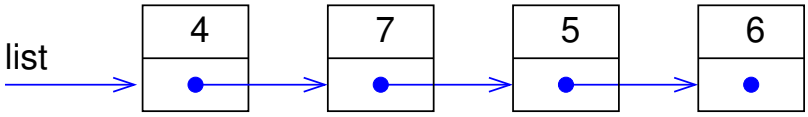


Figure 5: A single-linked list.

Generation of lists: insert elements, one after the other, either a) at beginning or b) after an existing element:

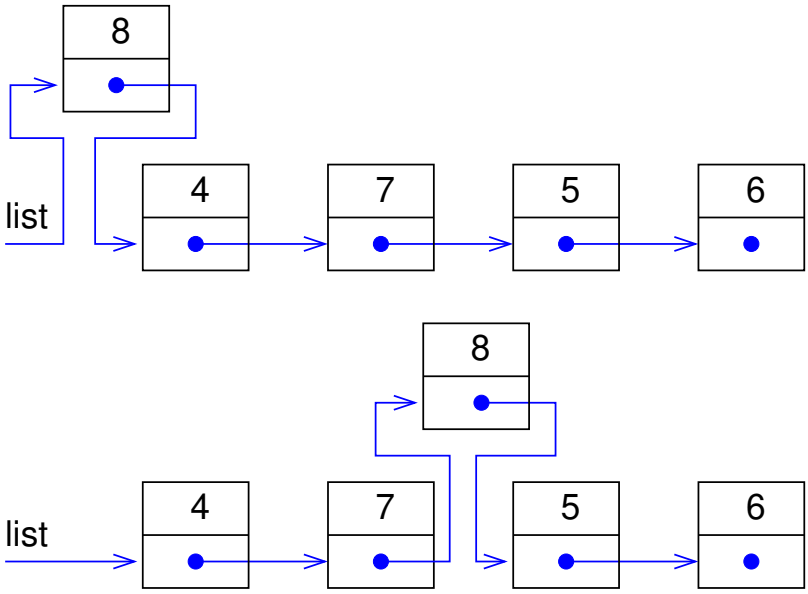


Figure 6: Insert an element into the list.

```
/****************** insert_element() ****************/
/** Inserts the element ‘elem‘ in the ‘list          **/
/** BEHIND the ‘where‘. If ’where’ is equal to NULL  **/
/** then the element is inserted at the beginning of **/
/** the list.                                        **/
/** PARAMETERS: (*)= return-paramter                 **/
/**         list: first element of list              **/
/**         elem: pointer to element to be inserted  **/
/**        where: position of new element            **/
/** RETURNS:                                         **/
/**  (new) pointer to the beginning of the list      **/
/****************************************************/
elem_t *insert_element(elem_t *list, elem_t *elem, elem_t *where)
{
  if(where==NULL)                    /* insert at beginning ? */
  {
    elem->next = list;
    list = elem;
  }
  else                                    /* insert elsewhere */
  {
    elem->next = where->next;
    where->next = elem;
  }
  return(list);
}
```

Print a list: iterate through all elements:

```
/****************** print_list() ****************/
/** Prints all elements of a list               **/
/** PARAMETERS: (*)= return-paramter            **/
/**           list: first element of list       **/
/** RETURNS:                                    **/
/**          nothing                            **/
/************************************************/
void print_list(elem_t *list)
{
  while(list != NULL)              /* run through list */
  {
    printf("%d ", list->info);
    list = list->next;
```

```
  }
  printf("\n");
}
```

[Activator]

Write a function `elem_t *list_last(elem_t *list)`, which returns a pointer to the last element of the list.

Removal of elements: a) first element b) other elements:

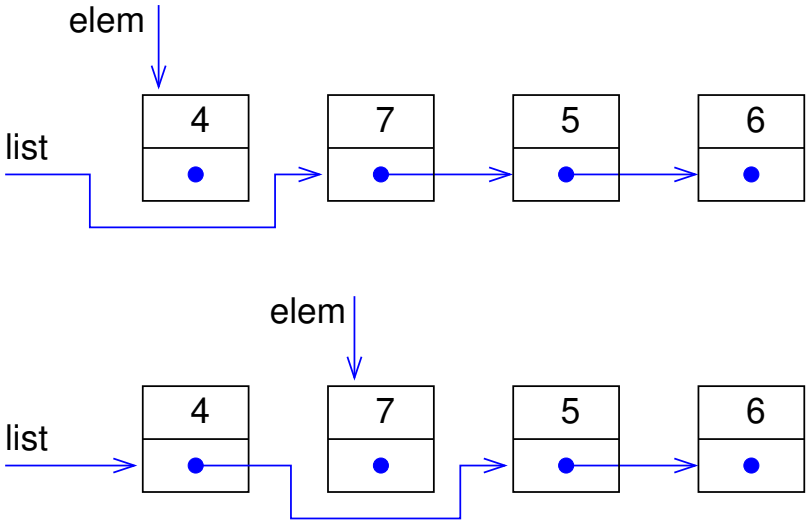


Figure 7: Removal of a list element.

One has to obtain the element bevor the element which is to be removed. Simpler: double-linked lists.

VIDEO: `video05b_tree1_r` | VIDEO: `video05c_tree2_r`

7.2 Binäry Search Trees

Search operations for lists: $O(N)$

Better: binary search trees:

binary trees:
each element (called node) has up to two successors.
Element without predecessor = the root of the tree
each node = root of a sub tree, consisting of the node + all direct and indirect successors.

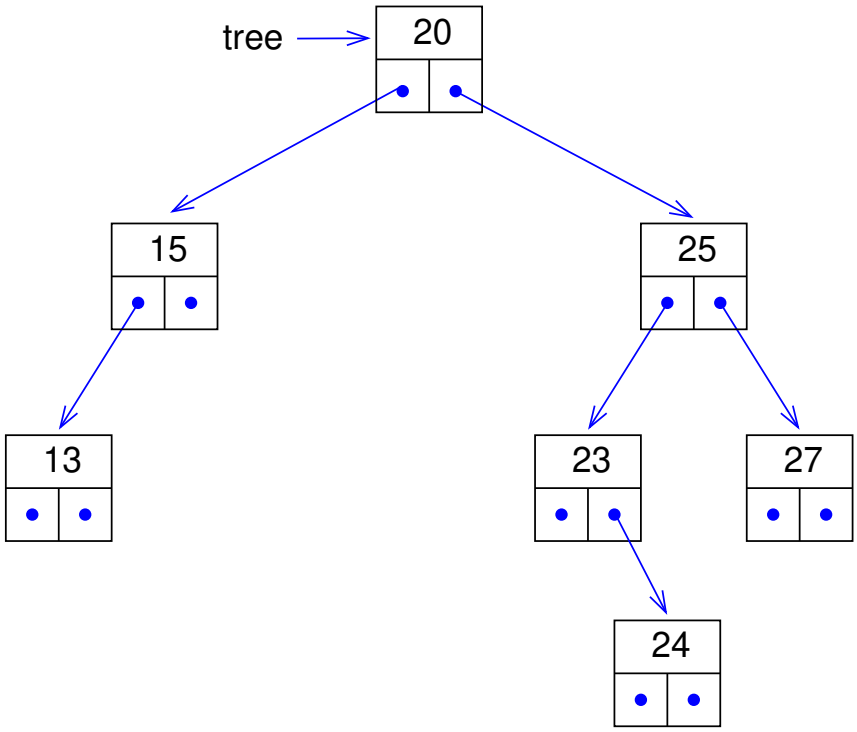


Figure 8: Binary search tree.

Search tree:
left sub tree: elements are "smaller" than root
right sub tree: elements are "larger" than root
holds also for all sub trees
→ search an element: compare with root. Either element is found, or search in the left (element smaller than root) or right sub tree (repeated within a loop). If at one point sub tree does not exist: element is not contained in tree.
→ search runs in $O(\log N)$ (typically).
Access to tree: pointer to root
nodes without successor = *leaves*

Basic data structure

```
/* data structures for tree elements */
struct node_struct
{
    int                  info;                  /* holds ‘‘information’’ */
    struct node_struct  *left;  /* points to left subtree (NULL if none) */
    struct node_struct *right; /* points to right subtree (NULL if none) */
};

typedef struct node_struct node_t;          /* define new type for nodes */
```

Creation and deletion of nodes: like for lists.

[Activator]

How could an algorithm for inserting an element look like? Think for yourself for 3 minutes, then discuss with your bench neighbor.

ATTENTION: Do not read beyond this point before you thought about the algorithm.

Insertion of a node:
Search for node/value
If found: stop
If not found: insert node as a leaf where the search terminated unsuccessfully.

```
/******************** insert_node() *****************/
/** Inserts 'node' into the 'tree' such that the     **/
/** increasing order is preserved                    **/
/** if node exists already, nothing happens          **/
/** PARAMETERS: (*)= return-paramter                 **/
/**         tree: pointer to root of tree            **/
/**         node: pointer to node                    **/
/** RETURNS:                                         **/
/**   (new) pointer to root of tree                  **/
/****************************************************/
node_t *insert_node(node_t *tree, node_t *node)
{
  node_t *current;

  if(tree==NULL)
    return(node);
```

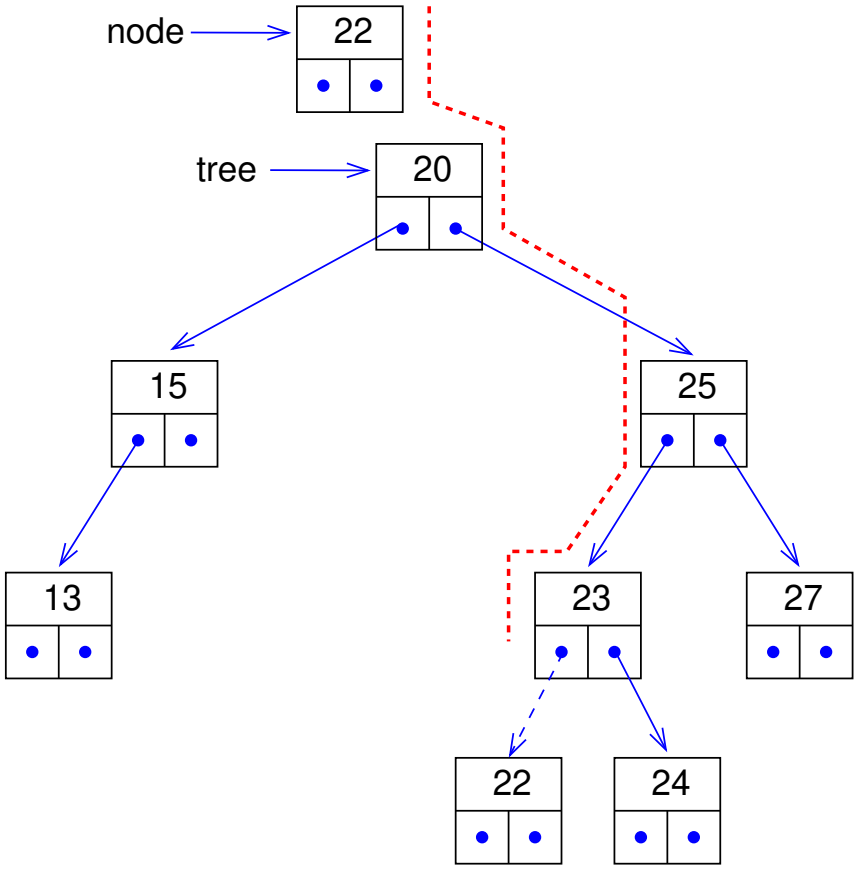


Figure 9: Inserting a new element into a search tree

```
current = tree;
while( current != NULL)                    /* run through tree */
{
  if(current->info==node->info)  /* node already contained ? */
    return(tree);
  if( node->info < current->info)             /* left subtree */
  {
    if(current->left == NULL)
    {
      current->left = node;                      /* add node */
      return(tree);
    }
    else
      current = current->left;          /* continue searching */
  }
  else                                        /* right subtree */
```

```
    {
      if(current->right == NULL)
      {
        current->right = node;                    /* add node */
        return(tree);
      }
      else
        current = current->right;         /* continue searching */
    }
  }
}
```

[Activator]

How can one print a tree? Think for yourself for three minutes, then discuss in groups of two or three.

ATTENTION: Do not read beyond this point before you thought about the algorithm.

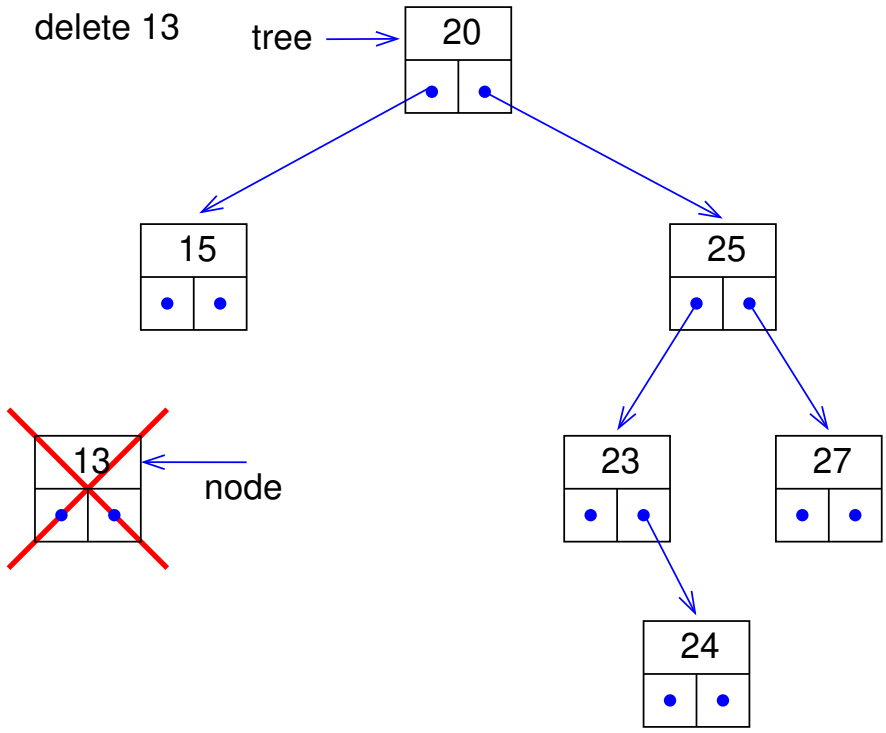
Ordered output of a tree: recursively

```
/******************** print_tree() ******************/
/** Prints tree in ascending order recursively.      **/
/** PARAMETERS: (*)= return-paramter                 **/
/**         tree: pointer to root of tree            **/
/** RETURNS:                                         **/
/**   nothing                                        **/
/*****************************************************/
void print_tree(node_t *tree)
{
  if(tree != NULL)
  {
    print_tree(tree->left);
    printf("%d ", tree->info);
    print_tree(tree->right);
  }
}
```

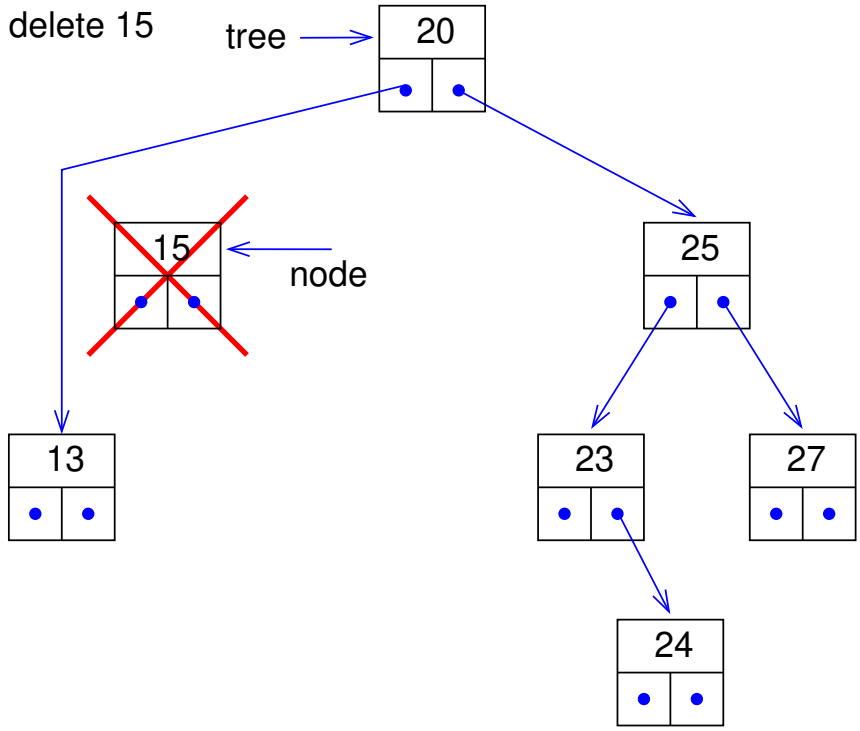
Remark: Also preorder (First print node, then left sub tree, then right sub tree) and postorder ... are possible.

Removal of a node with value x. Three cases:

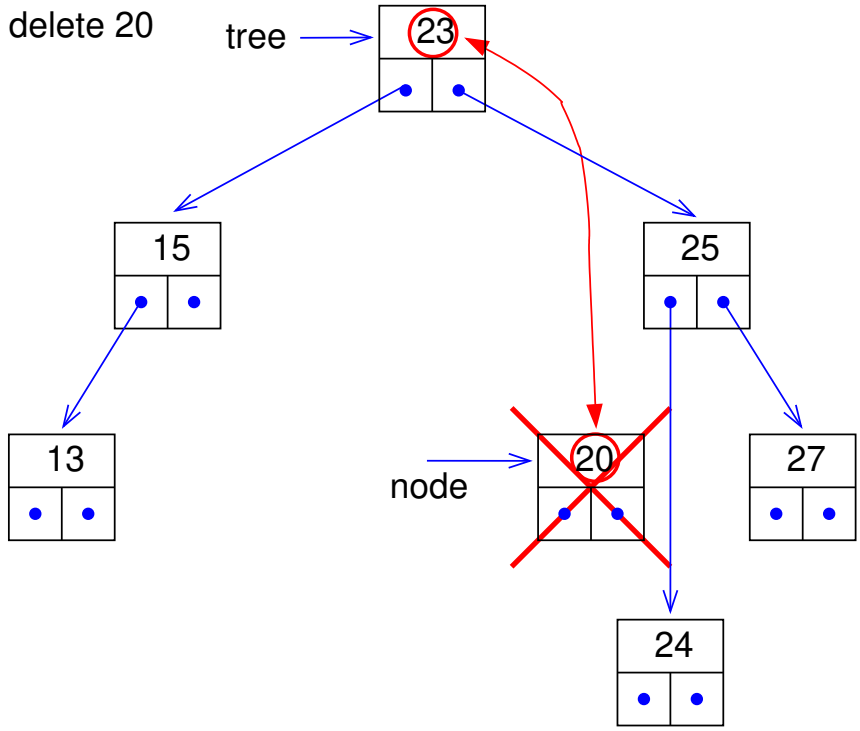
- Value is contained in a leaf (no successor): Simply remove.



- Node with x has one successor: link from predecessor to successor, omitting the node



- Node with x has two successors:
  Search for node n_2 which contains the smallest value y in the right sub tree (i.e. n_2 has no left successor).
  Exchange the values x and y. Now remove n_2 (which now contains x and has at most one successor) as in cases one or two .



[Activator]

Write a *recursive* function `int tree_size(node_t *tree)` which calculates the number of nodes in a tree.

Remark: Binary trees can be unbalanced (also called degenerate) (if two sub tree differ in height by more than one level)
Example: Iterated input to `insert_node` is ordered → tree becomes actually a list, thus, search takes $O(N)$ steps (worst case).
Solution: Balanced trees (e.g.. "red-black trees"): If a tree is unbalanced, it is balanced (e.g.. by "rotations") → all operations take only $O(\log N)$.

Block 6 (Donnerstag 22.2.2024)

VIDEO: `video06a_grundlagen_r`

8 Data Analysis and Random Numbers

Statistical data analysis: for deterministic simulations (Molecular Dynamics, DEs) and also for stochastic simulations.
For the latter ones also: Application of random numbers in computer simulations:

- Systems with random interactions (e.g. "Spin glasses")
- simulations at finite temperature with Monte Carlo algorithms
- randomised algorithms (modfied deterministic algorithms).

Generation of true random numbers in computer is possible (e.g. thermally-induced fluctuations of voltage measured at transistor etc). Advantage: random. Disadvantage: Statistical properties unknown and cannot be controlled.
Thus: pseudo random numbers = not random, but *as far as possible* equal statistical properties (distribution, correlations).

8.1 Fundamentals of Probability Theory

Ω: Set of outcomes of random experiment.

Ex: $\Omega = \{\text{head, tail}\}$ for coin toss.

Definition: A probability function P is a function $P : 2^\Omega \longrightarrow [0, 1]$ with

$$P(\Omega) = 1 \tag{7}$$

and for any sequence $A_1, A_2, A_3, \ldots$ of disjoint events ($A_i \cap A_j = \emptyset$ for $i \neq j$) holds:

$$P(A_1 \cup A_2 \cup A_3 \cup \ldots) = P(A_1) + P(A_2) + P(A_3) + \ldots \tag{8}$$

Properties (without proof), for $A, B \subset \Omega$, $A^c := \Omega \setminus A$ holds:

$$P(A^c) = 1 - P(A)\,. \tag{9}$$

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) \tag{10}$$

Forr independent random experiments holds:

$$P(A^{(1)}, A^{(2)}, \ldots, A^{(k)}) = P(A^{(1)})P(A^{(2)}) \ldots P(A^{(k)}) \tag{11}$$

Definition: The conditional probability of outcome A conditioned to C ist

$$P(A|C) = \frac{P(A \cap C)}{P(C)} \,. \tag{12}$$

[Activator]

What is P(dice 6|dice>3)?

Bayes' rule
Due to (12) holds $P(A|C)P(C) = P(A \cap C) = P(C \cap A) = P(C|A)P(A) \Rightarrow$

$$P(C|A) = \frac{P(A|C)P(C)}{P(A)} \,. \tag{13}$$

VIDEO: `video06b_ZVen_r`

8.2 Random Variables

Random variable (RV) X (sloppy): random experiment with $\Omega = \mathbb{R}$

Definition: Distribution function (DF) of a RV X is a function $F_X : \mathbb{R} \longrightarrow [0, 1]$ defined by

$$F_X(x) = P(X \leq x) \tag{14}$$

Ex: coin:

$$F(x) = \begin{cases} 0 & x < 0 \\ 0.5 & 0 \leq x < 1 \\ 1 & x \geq 1 \end{cases} . \tag{15}$$

x position of gas particle in a container $[0, L_x]$

$$F(x) = \begin{cases} 0 & x < 0 \\ x/L_x & 0 \leq x < L_x \\ 1 & x \geq L_x \end{cases} . \tag{16}$$

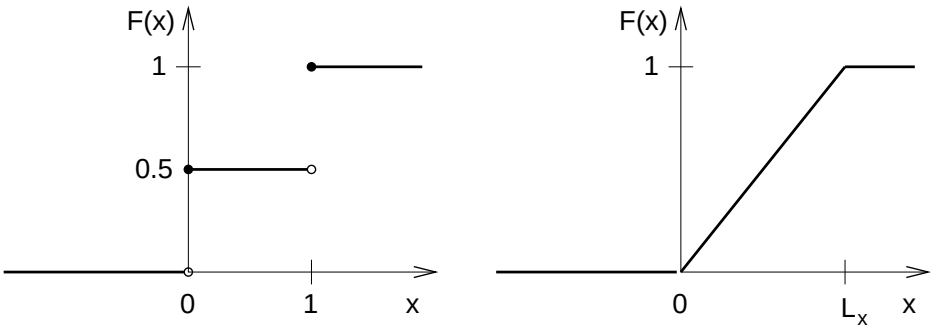


Figure 10: Distribution functions for coin and gas particle.

After random experiment corresponding to RV X with outcome x, next obtain $y = g(x)$:
Transformation of RV to $Y = g(X)$, in general:

$$Y = \tilde{g}\left(X^{(1)}, X^{(2)}, \ldots, X^{(k)}\right) . \tag{17}$$

Distribution function sometimes inconvenient $\rightarrow$ probability/density function:
VIDEO: `video06c_ZV_diskret_r`

8.2.1 Discrete Random Variables

Definition: The probability mass function (pmf) $p_X : \mathbb{R} \rightarrow [0, 1]$ is given by

$$p_X(x) = P(X = x) . \tag{18}$$

Distribution for n coin tosses (0/1):
Definition: The *Binomial Distribution* with parameters $n \in \mathbb{N}$ and p $(0 < p \leq 1)$ describes a RV X with pmf

$$p_X(x) = \begin{cases} \binom{n}{x} p^x (1-p)^{n-x} & (0 \leq x \leq n, x \in \mathbb{N}) \\ 0 & \text{else} \end{cases} \tag{19}$$

Notation $X \sim B(n, p)$.
Characterization of RVs:

$\{\tilde{x}_i\}$ set of possible values for which $p_X(\tilde{x}) > 0$ holds.
Definition:

- Expectation value

$$\mu \equiv \mathrm{E}[X] = \sum_i \tilde{x}_i P(X = \tilde{x}_i) = \sum_i \tilde{x}_i p_X(\tilde{x}_i) \tag{20}$$

- Variance

$$\sigma^2 \equiv \mathrm{Var}[X] = \mathrm{E}[(X - \mathrm{E}[X])^2] = \sum_i (\tilde{x}_i - \mathrm{E}[X])^2 p_X(\tilde{x}_i) \tag{21}$$

- Standard deviation

$$\sigma \equiv \sqrt{\mathrm{Var}[X]} \tag{22}$$

Properties:

$$\begin{aligned} \mathrm{E}[\alpha_1 X^{(1)} + \alpha_2 X^{(2)}] &= \alpha_1 \mathrm{E}[X^{(1)}] + \alpha_2 \mathrm{E}[X^{(2)}] && (23)\\ \sigma^2 = \mathrm{Var}[X] &= \mathrm{E}[X^2] - \mathrm{E}[X]^2 && (24)\\ \Leftrightarrow \quad \mathrm{E}[X^2] &= \sigma^2 + \mu^2 && (25)\\ \mathrm{Var}[\alpha_1 X^{(1)} + \alpha_2 X^{(2)}] &= \alpha_1^2 \mathrm{Var}[X^{(1)}] + \alpha_2^2 \mathrm{Var}[X^{(2)}] && (26) \end{aligned}$$

$\mathrm{E}[X^n]$: n'th moment
For the Binomial Distribution :

$$\begin{aligned} \mathrm{E}[X] &= np && (27)\\ \mathrm{Var}[X] &= np(1-p) && (28) \end{aligned}$$

VIDEO: `video06d_ZV_kont_r`

8.2.2 Continuous Random Variables

Definition: For a RV X with continuous DF F_X, the probability density function (pdf) $p_X : \mathbb{R} \to \mathbb{R}$ is given by

$$p_X(x) = \frac{dF_X(x)}{dx}\,. \tag{29}$$

Therefore:

$$F_X(x) = \int_{-\infty}^{x} p_X(\tilde{x})\, d\tilde{x} \tag{30}$$

Definition:

- Expectation value

$$\mathrm{E}[X] = \int_{-\infty}^{\infty} dx\, x\, p_X(x) \tag{31}$$

- Variance

$$\mathrm{Var}[X] = \mathrm{E}[(X - \mathrm{E}[X])^2] = \int_{-\infty}^{\infty} dx\,(x - \mathrm{E}[X])^2 p_X(x) \tag{32}$$

- Median $x_{\mathrm{med}} = \mathrm{Med}[X]$

$$F_X(x_{\mathrm{med}}) = 0.5 \tag{33}$$

Definition: Uniform distribution with parameters $a < b$, describes a RV X with pdf

$$p_X(x) = \begin{cases} 0 & x < a \\ \frac{1}{b-a} & a \le x < b \\ 0 & x \ge b \end{cases} \tag{34}$$

Notation: $X \sim U(a, b)$.
By using $g(X_{01}) = (b - a) * X_{01} + a$ one obtains $g(X_{01}) \sim U(a, b)$ if $X_{01} \sim U(0, 1)$.

[Activator]

Calculate $\mathrm{E}[X]$ and $\mathrm{Var}[X]$.

Most important:
Definition: The Gaussian distribution or Normal distribution with parameters μ and $\sigma > 0$, describes the RV X having pdf

$$p_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{35}$$

Notation: $X \sim N(\mu, \sigma^2)$.
Properties: $\mathrm{E}[X] = \mu$, $\mathrm{Var}[X] = \sigma^2$.
By using $g(X) = \sigma X_0 + \mu$ one obtains $g(X_0) \sim N(\mu, \sigma^2)$ falls $X_0 \sim N(0, 1)$.

Central limit theorem:
For independent RVs $X^{(1)}, X^{(2)}, \ldots, X^{(n)}$ with $E[X^{(i)}] = \mu$ and $\mathrm{Var}[X^{(i)}] = \sigma^2$, the RV

$$X = \sum_{i=1}^{n} X^{(i)} \tag{36}$$

is for large n distributed as $X \sim N(n\mu, n\sigma^2)$.

Densities of other important distributions:
Definition:

- Exponential distribution (for $x \geq 0$)

$$p_X(x) = \frac{1}{\mu} \exp\left(-x/\mu\right) \tag{37}$$

[Activator]
Calculate the distribution function for the Exponential distribution

---

Definition:

- Power-law distribution or Pareto distribution

$$p_X(x) = \begin{cases} 0 & x < 1 \\ \frac{\gamma}{\kappa}(x/\kappa)^{-\gamma-1} & x \geq 1 \end{cases} \tag{38}$$

For $\gamma > 1$ the expectation value is finite $\mathrm{E}[X] = \gamma\kappa/(\gamma-1)$, for $\gamma > 2$ $\mathrm{Var}[X] = \frac{\kappa^2\gamma}{(\gamma-1)^2(\gamma-2)}$

$$F_X(x) = 1 - (x/\kappa)^{-\gamma} \quad (x \geq 1) \tag{39}$$

- Fisher-Tippett distribution

$$p_X(x) = \lambda e^{-\lambda x} e^{-e^{-\lambda x}} \tag{40}$$

(also called Gumbel distribution for $\lambda = 1$) $\mathrm{E}[X] = \nu/\lambda$, $\nu \equiv 0.57721\ldots$, Maximum at $x = 0$, shift by $x \to (x - \mu)$

[Activator]
Can you read off the distribution function ?

Block 7 (Freitag 23.2.2024)

VIDEO: `video06e_linear_congruential_r`

8.3 Pseudo Random Numbers: Linear Congruential Generator (LCG)

[Activator]

Which approaches do you know to generate random numbers in a computer?

LCG: Generate series $I_1, I_2, \ldots$ of values between 0 and $m-1$, starting from given value I_0.

$$I_{n+1} = (aI_n + c) \mathrm{mod}\, m \tag{41}$$

$\rightarrow$ (Pseudo) random numbers x_n uniformly in interval $[0, 1)$: $x_n = I_n/m$. Arbitrary distribution: see below

Wanted: "chaotic" behavior. Aim: chose parameters a, c, m (and I_0), such that generator is "good" $\rightarrow$ criteria needed. Attention: Frequently results of simulations turned out to be (slightly) wrong due to bad random number generators (Ferrenberg et al., 1992) [1].

Program `linear_congruential.c` generates random numbers and creates histogram of the frequencies of occurrence:

```
/*** Linear congruential generator                                    ***/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define NUM_BINS 100
int main(int argc, char *argv[])
{
  int a, c, m, I;              /* parameter of random-number generator */
  double number;                                  /* generated number */
  int num_runs;                 /* number of generated random numbers */
  int histo[NUM_BINS];           /* histogram to measure distribution */
```

```
  double start_histo, end_histo;                  /* range of histogram */
  double delta;                                         /* width of bin */
  int bin;
  int t;                                                /* loop counter */

  m = 32768; c = 1; I = 1000;

  sscanf(argv[1], "%d", &num_runs);                  /* read parameters */
  sscanf(argv[2], "%d", &a);
  for(t=0; t<NUM_BINS; t++)                       /* initialise histogram */
      histo[t] = 0;
  start_histo = 0.0; end_histo = 1;
  delta = (end_histo - start_histo)/NUM_BINS;

  for(t=0; t<num_runs; t++)                                  /* main loop */
  {
    I = (a*I+c)%m;                       /* linear congruential generator */
    number = (double) I/m;                          /* map to interval [0,1) */
    bin = (int) floor((number-start_histo)/delta);
    if( (bin >= 0)&&(bin < NUM_BINS))                  /* inside range ? */
       histo[bin]++;                                      /* count event */
  }

  for(t=0; t<NUM_BINS; t++)                 /* print normalized histogram */
      printf("%f %f\n", start_histo + (t+0.5)*delta,
             histo[t]/(delta*num_runs));
  return(0);
}
```

Example: $a = 12351, c = 1, m = 2^{15}$ and $I_0 = 1000$ (values divided by m).
Distribution: is "uniform" in $[0, 1)$ (Fig. 11), but very regular.
Thus: correlations. Analysis: k-tuples of k successive random numbers $(x_i, x_{i+1}, \ldots, x_{i+k-1})$. Small correlation: k-dim space uniformly covered. LCGs: tuples are located on $k-1$-dim planes, their number is *at most* $O(m^{1/k})$ (B.J.T. Morgan, Elements of Simulation, 1984) [2]. Above parameter combinations $\rightarrow$ very few planes.

Change of program to measure 2-tuple correlations:

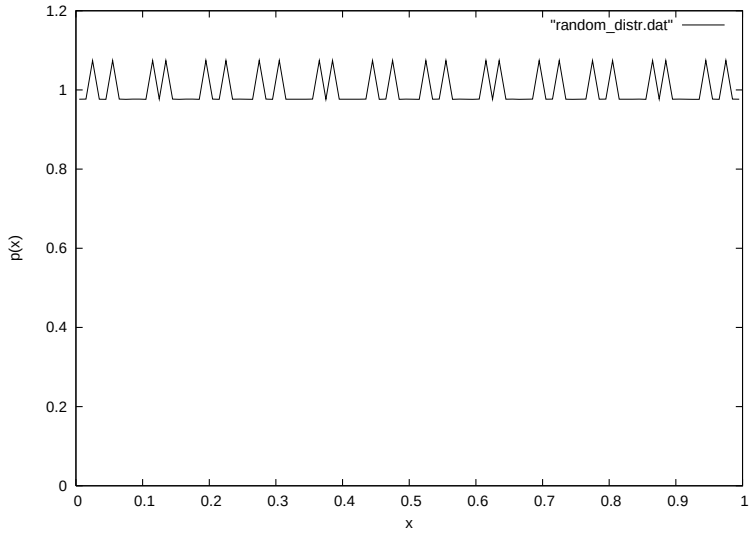


Figure 11: Distribution of random numbers in interval $[0, 1)$, generated by a linear congruential generator with parameters $a = 12351, c = 1, m = 2^{15}$.

```
double number_old;

number_old = (double) I/m;

for(t=0; t<num_runs; t++)                                  /* main loop */
{
  I = (a*I+c)%m;                      /* linear congruential generator */
  number = (double) I/m;                          /* map to interval [0,1) */
  bin = (int) floor((number-start_histo)/delta);
  printf("%f %f\n", number_old, number);
  number_old = number;
}
```

[Activator]

Generate random numbers in $[0, 1)$ with the provided program for :
a) $a = 12351, c = 1, m = 2^{15}$ and $I_0 = 1000$
b) $a = 12349$, $c = 1, m = 2^{15}$ and $I_0 = 1000$
Plot the 2-correlations using gnuplot.

Fig. a)

Fig. b)

Remarks:: The *GNU Scientific Library* (GSL) offers high-quality generators like the *Mersenne Twister*. For small experiments one can also use in Unix the `drand48()`.
VIDEO: `video06f_inversion_r`

8.4 Inversion Method

Given: `drand48()` (MS: `((double) rand())/(RAND_MAX)`) generates uniformly numbers in $[0, 1)$, denoted as U.

Target: random numbers Z distributed according to pdf $p(z)$, i.e. with distribution

$$P(z) \equiv \text{Prob}(Z \leq z) \equiv \int_{-\infty}^{z} dz' p(z') \tag{42}$$

Idea: look for function $g()$ with $Z = g(U)$. Assumption: g is strongly monotonous growing, i.e. it can be inverted→

$$P(z) = \text{Prob}(Z \leq z) = \text{Prob}(g(U) \leq z) = \text{Prob}(U \leq g^{-1}(z)) \tag{43}$$

With
1) $\text{Prob}(U \leq u) = F(u) = u$ if U uniformly in $[0, 1)$
2) Identification u with $g^{-1}(z)$
$\Rightarrow u = P(z) = g^{-1}(z) \Rightarrow z = g(u) = P^{-1}(u)$. (invert left and right)

Works if P can be obtained and inverted (possibly numerically).

Example: uniform distribution in $[2,4]$: $p(z) = 0.5$ for $z \in [2,4]$, 0 else. $\Rightarrow$ $P(z) = 0.5 \times (z-2)$ for $z \in [2,4]$. Equate to u and resolve with respect to z, thus: generated uniformly distributed number u and choose $z = 2 + 2 \times u$.

[Activator]

How does the generation look like for the exponential distribution: $p(z) = \lambda \exp(-\lambda z)$, $z \in [0, \infty)$?
Calculation:

Program `exponential.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_BINS 100

int main(int argc, char *argv[])
{
  int histo[NUM_BINS];                                    /* histogram */
  int bin;
  double start_histo, end_histo;                 /* range of histogram */
  double delta;                                        /* width of bin */
  int t;                                               /* loop counter */
  int num_runs;                  /* number of generated random numbers */
  double lambda;                            /* parameter of distribution */
  double number;                                    /* generated number */

  num_runs = atoi(argv[1]);                          /* read parameters */
  sscanf(argv[2], "%lf", &lambda);
  for(t=0; t<NUM_BINS; t++)                    /* initialise histogram */
      histo[t] = 0;
  start_histo = 0.0; end_histo = 10.0/lambda;
  delta = (end_histo - start_histo)/NUM_BINS;

  for(t=0; t<num_runs; t++)                               /* main loop */
  {
    number = -log(drand48())/lambda;     /* generate exp-distr. number */
    bin = (int) floor((number-start_histo)/delta);
```

```
    if( (bin >= 0)&&(bin < NUM_BINS))                /* inside range ? */
       histo[bin]++;                                    /* count event */
  }

  for(t=0; t<NUM_BINS; t++)                /* print normalized histogram */
      printf("%f %f\n", start_histo + (t+0.5)*delta,
             histo[t]/(delta*num_runs));
  return(0);
}
```

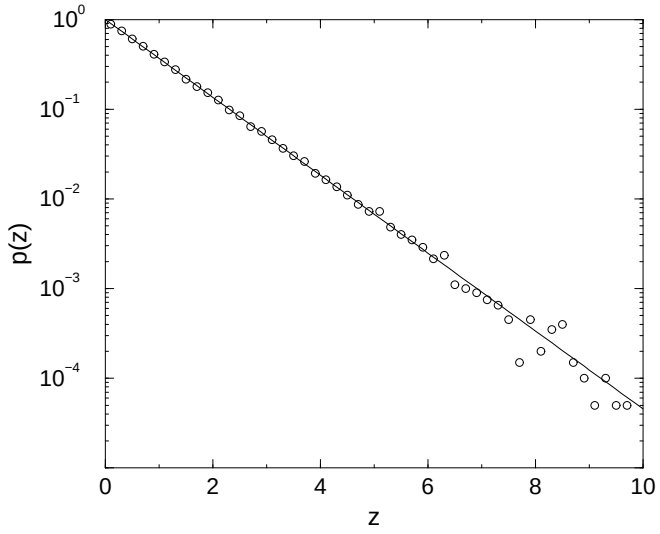


Figure 12: Histogram of random numbers, generated for exponential distribution ($\lambda = 1$) compared to pdf with logarithmic y-axis.

VIDEO: `video06g_reject_r`

8.5 Rejection Method

For (analytically) non-integrable pdfs, or (analytically) non-invertable distributions.

Simple variant: Condition: pdf $p(x)$ fits into box $[x_0, x_1) \times [0, p_{\max}]$, i.e. $p(x) = 0$ for $x \notin [x_0, x_1]$ and $p(x) \leq p_{\max}$.

Basic idea: generate random pairs (x, y), distributed uniformly in $[x_0, x_1) \times [0, p_{\max})$. Accept only those x with $y \leq p(x)$, i.e. the pairs below $p(x)$, see Fig. 13. The x value is the generated number for the pair.

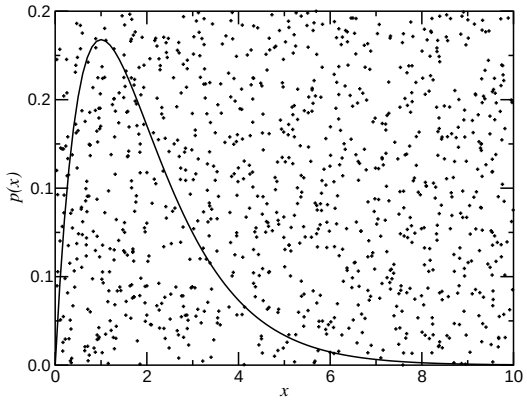


Figure 13: Rejection method: points (x, y) are uniformly distributed in rectangle. The probability that $y \leq p(x)$ is proportional to $p(x)$.

Implementierung as function (program `reject.c`):

```
/** gerenates random number for 'pdf' in the range **/
/** ['x0', 'x1'). condition: pdf(x)<='p_max' in    **/
//* the range ['x0', 'x1')                          **/
double reject(double p_max, double x0, double x1,
              double (* pdf)(double))
{
  int found;                 /* flag if valid number has been found */
  double x,y;                 /* random points in [x0,x1]x[0,p_max] */
  found = 0;
  while(!found)                    /* loop until number is generated */
  {
    x = x0 + (x1-x0)*drand48();            /* uniformly on [x0,x1] */
    y = p_max *drand48();                 /* uniformly in [0,p_max] */
    if(y <= pdf(x))                                  /* accept ? */
        found = 1;
  }
  return(x);
}
```

Beispiel:

```
/** artifical pdf **/
double pdf(double x)
{
  if( (x<0)||
      ((x>=0.5)&&(x<1))||
      (x>1.5) )
      return(0);
  else if((x>=0)&&(x<0.5))
      return(1);
  else
      return(4*(x-1));
}
```

results for 100000 random numbers is shown in Fig. 14.
Disadvantage: Possibly many random numbers are thrown away. Efficiency $1/(2p_{\max}(x_1 - x_0))$. (Factor 1/2 because at least two numbers (x, y) are needed for one final random number).

VIDEO: `video06i_schaetzwerte_r`

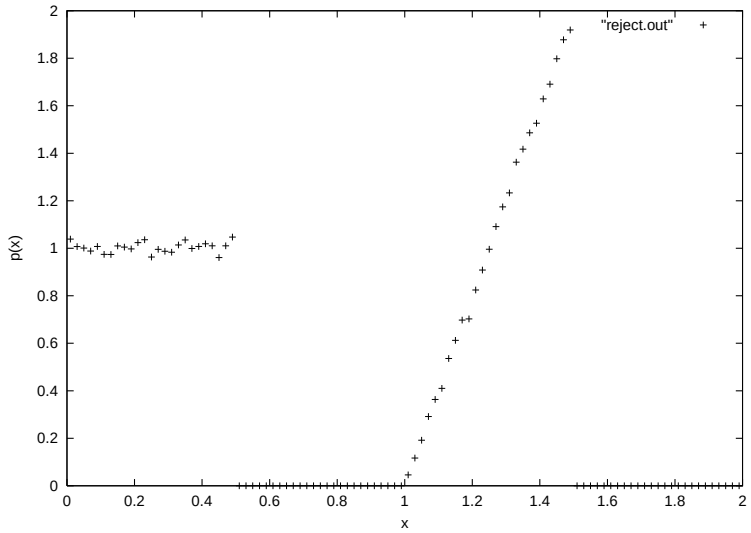


Figure 14: Rejection method: Histogram for the sample pdf.

8.6 Basic Data Analysis

Given: n data points ("sample") $\{x_0, x_1, \ldots, x_{n-1}\}$

Problem: underlying distribution $F(x)$ usually unknown.

8.6.1 Estimators

Estimators $h = h(x_0, x_1, \ldots, x_{n-1})$ are random variables as well: $H = h(X_0, X_1, \ldots, X_{n-1})$

- Mean (MW)

$$\overline{x} \equiv \frac{1}{n} \sum_{i=0}^{n-1} x_i \tag{44}$$

- Sample variance

$$s^2 \equiv \frac{1}{n} \sum_{i=0}^{n-1} (x_i - \overline{x})^2 \tag{45}$$

- Sample standard deviation

$$s \equiv \sqrt{s^2} \tag{46}$$

MW: To estimate the expectation value $\mu = \mathrm{E}[X]$. MW corresponds to RV $\overline{X} = \frac{1}{n} \sum_{i=0}^{n-1} X_i$. $\Rightarrow$

$$\mu_{\overline{X}} \equiv \mathrm{E}[\overline{X}] = \mathrm{E}\left[\frac{1}{n}\sum_{i=0}^{n-1} X_i\right] = \frac{1}{n}\sum_{i=0}^{n-1}\mathrm{E}[X_i] = \frac{1}{n} n\,\mathrm{E}[X] = \mathrm{E}[X] = \mu \quad (47)$$

$\rightarrow$ the mean unbiased.
Distribution for $\overline{X}$ has variance:

$$\begin{aligned}\sigma^2_{\overline{X}} &\equiv \mathrm{Var}[\overline{X}] = \mathrm{Var}\left[\frac{1}{n}\sum_{i=0}^{n-1} X_i\right] \overset{\mathrm{Var}[\alpha X]=\alpha^2\,\mathrm{Var}[X]}{=} \frac{1}{n^2}\sum_{i=0}^{n-1}\mathrm{Var}[X_i] \\ &= \frac{1}{n^2} n\,\mathrm{Var}[X] = \frac{\sigma^2}{n} \end{aligned} \quad (48)$$

$\rightarrow$ gets narrower for increasing n
$\rightarrow$ estimation gets more precises (while σ^2 unknown)

$\rightarrow$ wanted: unbiased estimator for σ^2 Attempt for $S^2 = \frac{1}{n}\sum_{i=0}^{n-1}(X_i - \overline{X})^2$:

$$\begin{aligned}\mathrm{E}[S^2] &= \mathrm{E}\left[\frac{1}{n}\sum_{i=0}^{n-1}(X_i - \overline{X})^2\right] = \mathrm{E}\left[\frac{1}{n}\sum_{i=0}^{n-1}(X_i^2 - 2X_i\overline{X} + \overline{X}^2)\right] \\ &\overset{\sum_i X_i = n\overline{X}}{=} \frac{1}{n}\left(\sum_{i=0}^{n-1}\mathrm{E}[X_i^2] - n\,\mathrm{E}[\overline{X}^2]\right) \overset{\mathrm{E}[Y^2]=\sigma_Y^2+\mu_Y^2}{=} \frac{1}{n}\left(n(\sigma^2+\mu^2) - n(\sigma^2_{\overline{X}} + \mu^2_{\overline{X}})\right) \\ &\overset{\sigma^2_{\overline{X}}=\frac{\sigma^2}{n}}{=} \frac{1}{n}\left(n\sigma^2 + n\mu^2 - n\frac{\sigma^2}{n} - n\mu^2\right) = \frac{n-1}{n}\sigma^2 \end{aligned} \quad (49)$$

S^2 is biased, but $\frac{n}{n-1}S^2$ is unbiased.

Advanced subjects: confidence intervals, resampling, hypothesis tests, principal component analysis, clustering, fits, ...
(see A.K. Hartmann, *Big Practical Guid to Computer Simulations*, (World-Scientific, 2015) [3])

Block 8 (Montag 26.2.2024)

VIDEO: `video07a_perkolation_r` | VIDEO: `video07b_cluster_r`

9 Percolation

Model for conducting material: partition a block of material into small cubes, where each cube is "non-conducting" with probability $1-p$. Remaining fraction p is "conducting". In general: sites of interest = "occupied", here the conducting ones.

Question: Value of p such that current can run from one side to the other? ("percolating") $\rightarrow$ Phase transition at critical concentration $p = p_c$ above which current runs. Percolation: important (toy) model for phase transitions. Many phase transitions can be explained by hidden percolation transitions.

Literature: Stauffer and Arahony, Percolation theory (1994) [4].

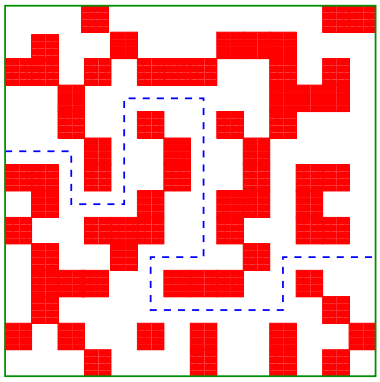


Figure 15: Percolation in two dimensions: there is a path of conducting sites (light), such that current runs through full system (broken line). The system "percolates".

Cluster := connected region of occupied sites $\Rightarrow$ Percolation = largest cluster spans full lattice.

Order parameter: size S of largest cluster divided by total number N of sites.

$p > p_c$: $S/N \to \text{const}(p) > 0$ for $N \to \infty$
$p < p_c$: $S/N \to 0$ for $N \to \infty$

The value of p_c depends on the dimension of the system and on the lattice structure.

[Activator]

How large is p_c in one dimension (system =line)?

For larger dimensions usually no analytical statement → computer simulations (square lattice: $p_c \approx 0.592746$, cubic: $p_c \approx 0.3116$, ... [4]). Corresponding algorithms (to find clusters): applied in MANY areas of computational physics.

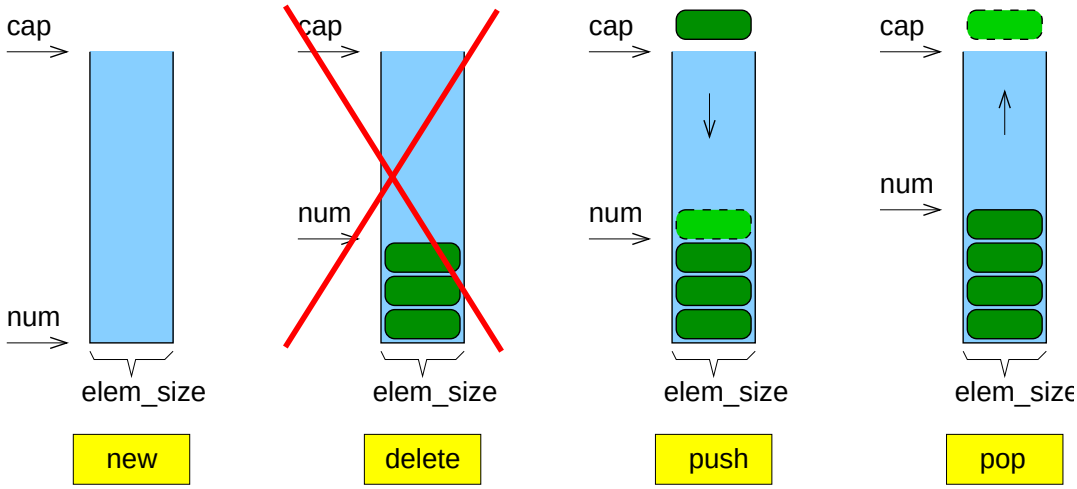
9.1 Stacks

For implementation needed: special Data type: *stacks*.

Stack = one can put elements on top and remove from the top.
→ LIFO (last in first out).
Remark: waiting queue = FIFO principle.

Basic operations:

- `lstack_new`: create stack of elements of given size with maximum number of elements..
- `lstack_delete`: delete stack.
- `lstack_push`: Put one element on top of the stack.
- `lstack_pop`: Remove top element from stack.



9.2 Cluster Algorithm

Representation of a d-dim systems in computer:
e.g., d-dim array `site`, e.g.. 3-dim `site[x][y][z]` = 1 if site (x,y,z) occupied. Also higher dimensions of theoretical interest, thus
`site [x1][x2][x3][x4][x5][x6][x7]` → too complicated, inflexible (also for changing lattice structure).

[Activator]

Think for 3 minutes about how one can represent a system betters.

ATTENTION: Do NOT read on before you found something

Solution: Number sites from 1 to N and use 1-dimensional (!!) array `site`:
`site[t]=1` if site `t` occupied.
Realization of lattice structure: variable '`num_n`' (=number of neighbors) and array `next`.
Each site '`t`' has `num_n` neighbors, stored in `next[t*num_n]`... `next[(t+1)*num_n-1]`.
order (simple cubic): +x,-x,+y,-y,... directions.
Access conveniently by macros:

```
#define INDEX(t, r, nn) ((t)*(nn)+r)
#define NEXT(t, r) next[INDEX(t, r, num_n)]
```

Attention: whenever using the macros, the variables `next` and `num_n` have be available with exactly these names. This can be made sure easily if all variables are used locally. The macros make the coding a bit less flexible but

more convenient and better readable (runs also faster as if using function calls).

Initialize array `next[]` only once in the beginning, can be used everywhere. Here: periodic boundary conditions:

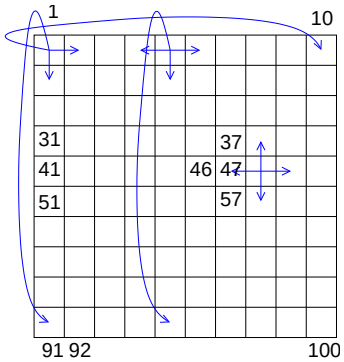


Figure 16: A 10x10 square lattice with periodic boundary conditions. The arrows point to the neighbors.

Example: $L \times L$ square lattice for $L = 10$. Site $i = 48$ has the neighbors $i + 1 = 49$ (+x), $i - 1 = 47$ (-x), $i + L = 58$ (+y) and $i - L = 38$. Site $i = 1$ has the neighbors 2 (+x), 10 (-x), 11 (+y), 91 (-y).
Main question of Percolation: is there a *percolating* cluster, i.e. running through the system:
First, determine *clusters* = connected regions (see below). Next:

a) Check whether there is cluster spanning through system

b) In the percolating region: average size of larger clusters grows linearly with size $N \to$: Calculate fraction of sites in largest cluster (for different system sizes N).

Here: first method b), easier to implement.

Idea for determination of clusters:

[Activator]

Think about a possible solution for three minutes, then discuss for 3 minutes with your neighbor.

ATTENTION: Do NOT read on before you head some idea.

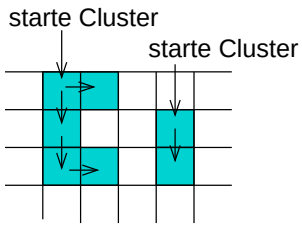
Starting point: a lattice with some sites occupied.
The occupied neighbors of an occupied site belong to the same cluster.
Their occupied neighbors also.
Implementation: The neighbors are store in a stack and the treated one after the other. Take care than every site is put on stack at most once.

algorithm Cluster
begin
  while there are "untreated" occupied sites t **do**
  begin
    push t on stack
    start new cluster with t
    while stack is not empty **do**
    begin
      pop one site c from stack
      for all neighbors n of c **do**
        if n occupied and not yet "treated" **then**
          push n on stack, add to cluster, mark as "treated"
    end
  end
end

Store in `cluster[]` array, cluster ID of each site (-1: not yet treated).

Since each site is treated only once: run time $O(N)$.

Example:



Subroutine for cluster determination:

```
/****************** percol_cluster() ******************/
/** Calculates the connected clusters of the lattice   **/
/** 'site'. Occupied sites have 'site[i]=1' (i=1..N).  **/
/** Neigbours of occupied sites form clusters.         **/
/** For each site, in 'cluster[i]' the id of the       **/
/** cluster (starting at 0) is stored                  **/
/** PARAMETERS: (*)= return-paramter                   **/
/**       num_n: number of neighbours                  **/
/**           N: number of sites                       **/
/**        next: gives neighbours (0..N)x(0..num_n-1)  **/
/**              0 not used here. Use NEXT() to access **/
/**        site: tells whether site is occupied        **/
/** (*) cluster: id of clusters sites are contained in **/
/** RETURNS: number of clusters                        **/
/******************************************************/
int percol_cluster(int num_n, int N, int *next,
                   short int *site, int *cluster)
{
  int num_clusters=0;
  int t, r;       /* loop counters over sites, directions */
  int current, neighbour;                        /* sites */
  lstack_t *members;      /* stack of members for cluster */
  for(t=1; t<=N; t++)    /* initialise all clusters empty */
    cluster[t] = -1;
  members = lstack_new(N, sizeof(int));
  for(t=1; t<=N; t++)              /* loop over all sites */
  {
    if((site[t] == 1)&&(cluster[t]==-1)) /* new cluster ? */
    {
      lstack_push(members, &t);          /* start cluster */
      cluster[t] = num_clusters;
      while(!lstack_is_empty(members))  /* extend cluster */
      {
        lstack_pop(members, &current);
        for(r=0; r<num_n; r++)   /* loop over neighbours */
        {
          neighbour = NEXT(current, r);
          if((site[neighbour]==1)&&(cluster[neighbour]==-1))
          {          /* neighbour belongs to same cluster */
            lstack_push(members, &neighbour);
            cluster[neighbour] = num_clusters;
          }
        }
      }
      num_clusters++;
    }
  }
  lstack_delete(members); return(num_clusters);
}
```

Example run for 2 dimensions, side length $L = 10$, $p = 0.5$). Output cluster IDs of site:

```
 0  0  0  0  0  0  0  0     0
 0                0     0  0  0
                  0        0
    1           0  0  0  0
    1     2           0  0     3
    1     2     4           5
       6     4  4  4  4
 0                            0  0
 0  0                  7        0  0
 0  0     0  0  0        0  0
```

9.3 Ergebnisse

Many program calls done by script `run_percol.scr`:

```
#!/bin/bash
L=$1
for p in 0.1 0.2 0.3 0.4 0.45 0.5 0.52 0.54 0.56 0.57 0.58 0.59 \
         0.60 0.61 0.62 0.64 0.66 0.68 0.7 0.8 0.9 1.0
do
  percolation1 $L $p 100
done
```

Possible you have to make the script executable by: `chmod u+x run_percol.scr`.

[Activator]

Compile the program by
`cc -o percolation1 percolation1.c percol.c stacks.c`
Run the script for different system sizes (e.g. 10, 20, 50, 100, 200) and redirect each time the results to a file, e.g., by `run_percol.scr 10 > perc10.dat`. Plot the data using `gnuplot`.

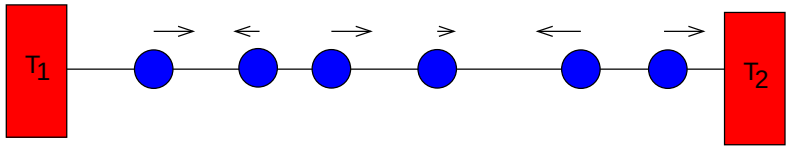
Block 9 (Dienstag 27.2.2024)

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?

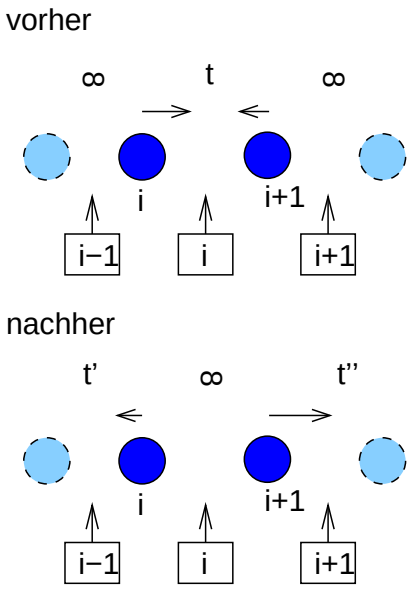
Particles:

```
/** 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:

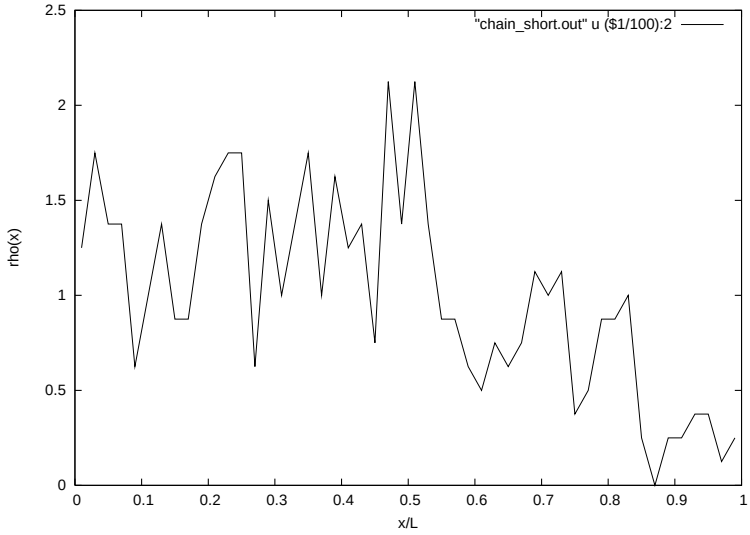


Figure 17: Average density as function of position in time interval $[50, 100]$.

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

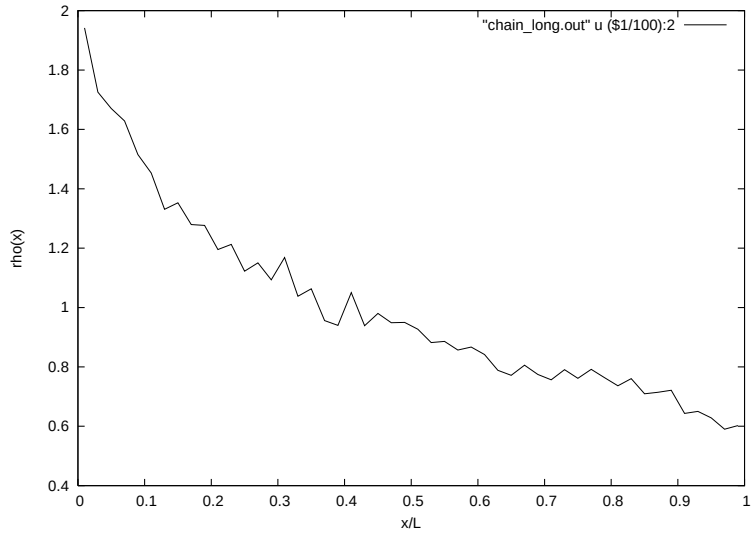


Figure 18: Average density as function of position in time interval $[5000, 10000]$.

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.

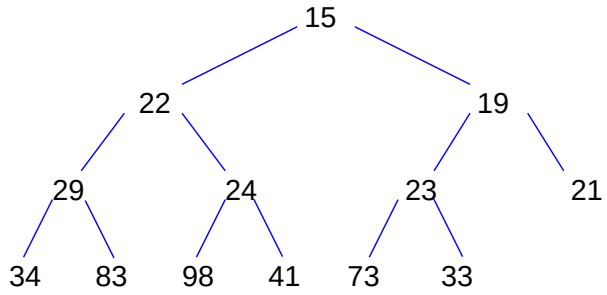
Preview:
Run time example: $n = 500, t_{\text{end}} = 10000$.

```
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
```

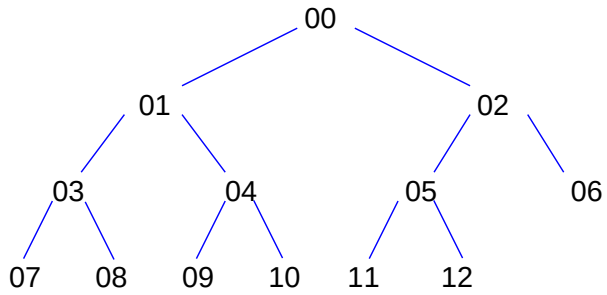
with heap → faster program → larger systems ($n = 16383$ vs. $n = 1281$)
→ more reliable, DIFFERENT results (Crossover), see P. Grassberger et al., Phys. Rev. Lett **89**, 180601 (2002) [6].

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:



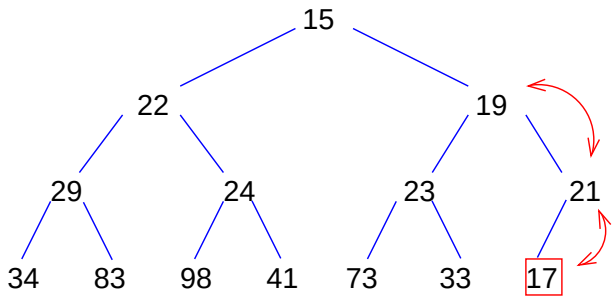
node i:
predecessor: $(i-1)/2$ (`int` division)
left descendant: $2i+1$
right descendant: $2i+2$

Basic heap operations:
Insert:

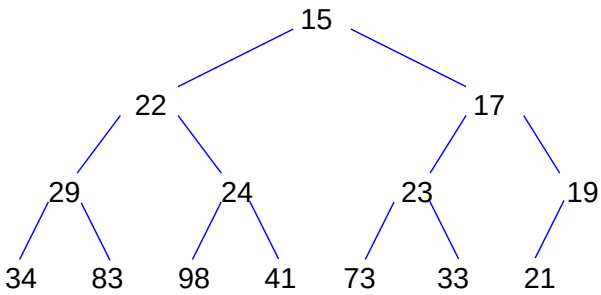
algorithm heap_insert()
begin
  add element at the end;
  while (element smaller than predecessor)
    exchange with predecessor;
end

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

Eaxmple: insert "17"



results in



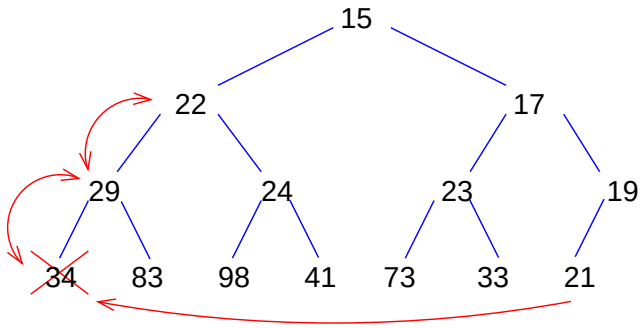
At most one sweep from leaf to root
$\rightarrow$ time $O(\log N)$

Removal:

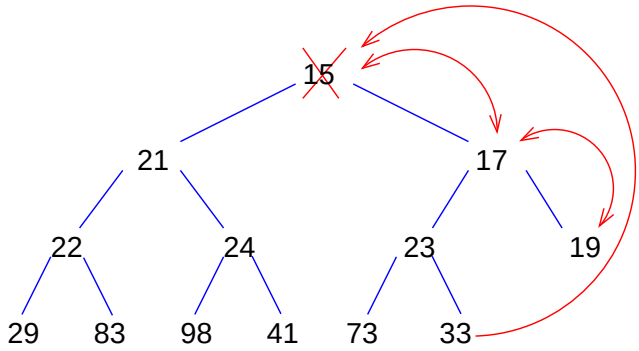
algorithm heap_remove()
begin
  replace element by last element;
  if (element smaller than predecessor) **then**
    while (element smaller than predecessor)
      exchange with predecessor;
  else
    while (element larger than a desendant)

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 $O(N \log N)$.

Block 10 (Donnerstag 29.2.2024)

11 Monte Carlo Simulations

VIDEO: `video08b_markov_ketten_r`

11.1 Markov Chains

Given; sytsem with (a finite number of) states $\underline{y} = \underline{y}_1, \underline{y}_2, \ldots, \underline{y}_K$ and probabilities $P(\underline{y})$.
Typical:K exponentially large in number N of particles and $P(\underline{y})$ significant only for an exponentially small fraction of states ($\sum_{\text{significant } \underline{y}} 1/K \to 0$ and $\sum_{\text{significant } \underline{y}} P(\underline{y}) \to 1$)
Example: Boltzmann distribution
Target: Estimation of expectation values of obervables $A(\underline{y})$

$$\langle A \rangle := \sum_{\underline{y}} A(\underline{y}) P(\underline{y}) \tag{52}$$

Assumption: K very large $\to$ impossible to enumerate all states.

Simple Sampling:

Generate M states $\{\underline{y}^i\}$ $(i = 1, \ldots, M)$ randomly, with uniform probability. Then:

$$\langle A \rangle \approx \overline{A}^{(1)} := \sum_{\underline{y}^i} A(\underline{y}^i) P(\underline{y}^i) / \sum_{\underline{y}^i} P(\underline{y}^i)$$

Drawback: for almost all states $P(\underline{y}^i)$ is exponentially small $\to$ $\overline{A}^{(1)}$ not accurate.
Importance Sampling:

Better: generate M configurations $\underline{y}^i$ according to $P(\underline{y}^i)$. (generate *most important states* more often), then:

$$\langle A \rangle \approx \overline{A}^{(2)} := \sum_{\underline{y}^i} A(\underline{y}^i) / M \tag{53}$$

But: usually no algorithm to generate y^i directly according to $P(\underline{y}^i)$ (distribution function cannot be obtained or inverted).

$\rightarrow$
Baisc idea: states $\underline{y}^i$ are *not* generated independently but by a *probabilistic dynamics* of states $\underline{y}(t)$ at discrete times $t = 0, 1, 2, \ldots$: $\underline{y}(0) \rightarrow \underline{y}(1) \rightarrow \underline{y}(2) \rightarrow \ldots$.

Assumption: states $\underline{y}(t+1)$ depend only on random numbers and on $\underline{y}(t)$. $\{\underline{y}(t)|t = 0, 1, 2, \ldots\}$ is called a *Markov chain*.
Description of transitions $\underline{y}(t) \rightarrow \underline{y}(t+1)$ by $W_{\underline{y}\underline{z}} = W(\underline{y} \rightarrow \underline{z})$ = probability to move from state $\underline{y}$ (at time t) to state $\underline{z}$ (at time $t+1$), where $W_{\underline{y}\underline{z}}$ does not depend on time.

[Activator]

Which properties has $W_{\underline{y}\underline{z}}$?

The state space plus the transition probabilities is called a *Markov process*.

Example: Two state system

Two states A,B and transition probabilities $W_{AA} = 0.6$, $W_{AB} = 0.4$, $W_{BA} = 0.1$, $W_{BB} = 0.9$.

Assumption: One generates $N = 100$ Markov chains, which all start in A: $N(A/B, t)$ = number of chains which are in state A/B, at time step t. Let $N(A, 0) = 100$ and $N(B, 0) = 0$. How could the dynamics look like?

For (about) average values Mittelwerte: $N(A, 0)W_{AB} = 100 \times 0.4 = 40$ chains move from $A \rightarrow B$, while no transitions happen $B \rightarrow A$ at $t = 0$. An so on. Evolution:

$t = 0$: N(A)=100 $\underset{40}{\overset{0}{\rightleftarrows}}$ N(B)=0

$t = 1$: N(A)=60 $\underset{24}{\overset{4}{\rightleftarrows}}$ N(B)=40

$t = 2$: N(A)=40 $\underset{16}{\overset{6}{\rightleftarrows}}$ N(B)=60

$$t=3: \quad \boxed{\text{N(A)=30}} \overset{7}{\underset{12}{\rightleftarrows}} \boxed{\text{N(B)=70}}$$

$$t=4: \quad \boxed{\text{N(A)=25}} \overset{8}{\underset{10}{\rightleftarrows}} \boxed{\text{N(B)=75}}$$

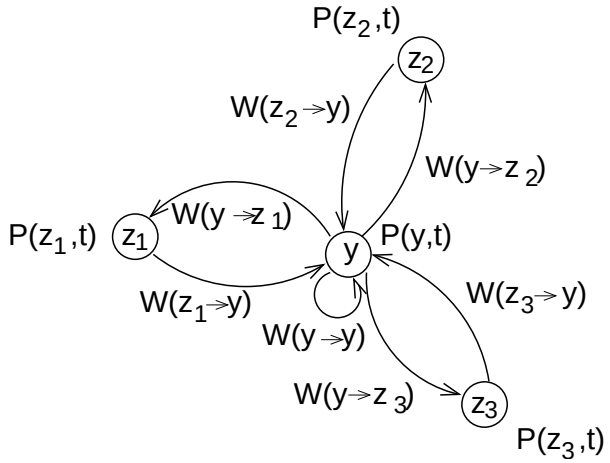
$$t=5: \quad \boxed{\text{N(A)=23}} \overset{8}{\underset{9}{\rightleftarrows}} \boxed{\text{N(B)=77}}$$

$$t=6: \quad \boxed{\text{N(A)=22}} \overset{8}{\underset{9}{\rightleftarrows}} \boxed{\text{N(B)=78}}$$

$$t=7: \quad \boxed{\text{N(A)=21}} \overset{8}{\underset{8}{\rightleftarrows}} \boxed{\text{N(B)=79}}$$

From now on about $N(A,t)W_{AB} = N(B,t)W_{BA}$. The transfer between states balances $\rightarrow N(A/B,t)$ =const, *stationary state* $\square$

General:
Let $P(\underline{y},t) = \langle N(\underline{y},t)/N \rangle$ the probability that system at time t is in state $\underline{y}(t) = \underline{y}$. Balance for state $\underline{y}$:



Therefore ("Master Equation"):

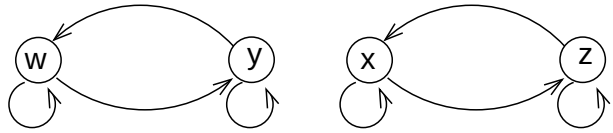
$$\Delta P(\underline{y},t) := P(\underline{y},t+1) - P(\underline{y},t) = \sum_{\underline{z}} W_{\underline{z}\underline{y}} P(\underline{z},t) - \sum_{\underline{z}} W_{\underline{y}\underline{z}} P(\underline{y},t) \quad \forall \underline{y} \quad (54)$$

Under specific conditions (in particular if there is only one eigen value $\lambda = 1$ of matrix $\hat{W}$ ($\hat{W}_{\underline{y}\underline{z}} = W_{\underline{y}\underline{z}} - \delta_{\underline{y}\underline{z}} \sum_{z'} W_{\underline{y}\underline{z}'}$), see L. Reichel, 1998 [7]), the probability distribution $\rightarrow$ *stationary* (time-independent) distribution.

$$P_{ST}(\underline{y}) := \lim_{t\to\infty} P(\underline{y}, t)$$

Independent of given initial state $\underline{y}(0)$! System is called *ergodic*.

Example for non-ergodic system:



Target: Choose $W_{\underline{y}\underline{z}}$ such that $P_{ST} = P$

Since $P(.)$ time-independent, from (54) with

$$0 = \Delta P(\underline{y}) = \sum_{\underline{z}} W_{\underline{z}\underline{y}} P(\underline{z}) - \sum_{\underline{z}} W_{\underline{y}\underline{z}} P(\underline{y}) \quad \forall \underline{y}$$

one has more conditions for transition probabilities.

Can be fulfilled, e.g. by

$$W_{\underline{z}\underline{y}} P(\underline{z}) - W_{\underline{y}\underline{z}} P(\underline{y}) = 0 \quad \forall \underline{y}, \underline{z} \tag{55}$$

(called *detailled balance*).

Thus: Markov process generates state distributed according to $P(.)$.
Hence averages (53) can be obtained.

Attention: at beginning states depend on $\underline{y}(0)$
$\Rightarrow$ states for $t < t_{equi}$ or omitted from average calculation ("equilibration").

$\underline{y}(t+1)$ typically "similar" to $\underline{y}(t)$
$\Rightarrow$ only distant states $\underline{y}(t), \underline{y}(t+\Delta t), \underline{y}(t+2\Delta t), \ldots$ are almost independent.

Remark: $t_{equi}, \Delta t$ depend STRONGLY on model, algorithm and parameters
$\Rightarrow$ have to be determined experimentally in simulations.
VIDEO: `video08c_metropolis_r`

11.2 Disordered (diluted) Ferromagnet

Model:

$$\mathcal{H} = -J \sum_{\langle i,j \rangle} e_i s_i e_j s_j \quad J > 0. \tag{56}$$

$\langle i, j \rangle$: sum over interacting neighbors (e.g. next neighbours)
$e_i = 0, 1$: site free/occupied
$s_i = \pm 1$: spin orientation "up"/ "down".
configuration: $\underline{x} = (s_1, s_2 \ldots, s_N)$ (spins with $e_i = 0$ are effectivly ignored)

Target: simulation in *canonical ensemble*:

$$P(\underline{x}) = \frac{1}{Z} \exp(-\mathcal{H}(\underline{x})/T)\,, \tag{57}$$

with $Z =$ partition function $Z = \sum_{\{\underline{x}\}} \exp(-\mathcal{H}(\underline{x})/T)$.

Method: use Markov chain.

Here:
Metropolis Algorithmus defined by transition probabilities $W_{\underline{yz}} = W(\underline{y} \to \underline{z})$.

Basic idea: (given $\underline{y} = \underline{y}(t)$)

1. Create *test configuration* $\underline{z}$ *randomly*, according to $A(\underline{y} \to \underline{z})$

2. With probability $\tilde{W}(\underline{y} \to \underline{z})$ $\underline{z}$ is *accepted*, i.e. $\underline{y}(t+1) = \underline{z}$.
   $\tilde{W}(\underline{y} \to \underline{z})$ is called *acceptance probability*

   With probability $1 - \tilde{W}(\underline{y} \to \underline{z})$ $\underline{z}$ is *rejected*, i.e. $\underline{y}(t+1) = \underline{y}$

$\to$ Full probability:

$$W(\underline{y} \to \underline{z}) = A(\underline{y} \to \underline{z}) \tilde{W}(\underline{y} \to \underline{z}) \quad (\underline{y} \neq \underline{z})\,. \tag{58}$$

(probability that state $\underline{y}$ remains: $W(\underline{y} \to \underline{y}) = 1 - \sum_{\underline{z} \neq \underline{y}} W(\underline{y} \to \underline{z})$).

Insert Eq. (58) into detailled balance condition

$$\frac{\tilde{W}(\underline{y} \to \underline{z})}{\tilde{W}(\underline{z} \to \underline{y})} = \frac{P(\underline{z})}{P(\underline{y})} \frac{A(\underline{z} \to \underline{y})}{A(\underline{y} \to \underline{z})}\,. \tag{59}$$

For Metropolis algorithmus [8], choice:

$$\tilde{W}(\underline{y} \to \underline{z}) = \min\left(1, \frac{P(\underline{z})}{P(\underline{y})} \frac{A(\underline{z} \to \underline{y})}{A(\underline{y} \to \underline{z})}\right) , \tag{60}$$

One sees easily that Eq. (59) holds.

Most simple: single-spin flip dynamics:
Let $\underline{y} = (s_1, s_2, \ldots, s_N)$. Choose a spin j randomly, then $\underline{z} = (s'_1, s'_2, \ldots, s'_N)$ with

$$s'_i = \begin{cases} -s_i & \text{for } i = j \\ s_i & \text{else} \end{cases}$$

All spins equal likely: $A(\underline{y} \to \underline{z}) = 1/N$
(better: choose among spins $e_i \neq 0$: $A(\underline{y} \to \underline{z}) = 1/(\sum_i e_i)$)

[Activator]

What is the result for the Boltzmann distribution when inserting into (Metropolis Acceptance probability) ?

Acceptance probability depends only on *energy change* $\Delta\mathcal{H} = \mathcal{H}(\underline{z}) - \mathcal{H}(\underline{y})$
$\Rightarrow$ easy to compute, because only neighbors $N(j)$ of j contribute:
$\Delta\mathcal{H} = \Delta\mathcal{H}(j) = 2J \sum_{i \in N(j)} e_i s_i e_j s_j$
Obervation: at low temperatures, changes which increase energy are rarely accepted.
Since at most on spin flipped: algorithm is slow, configurations change globaly only on large time scales.
Slowest: Close to phase transitions. (Solution here: cluster algorithms).
Summary:

algorithm MC Ferromagnet
begin
  for(MC iterations)
  begin
    select occupied site t randomly
    $\Delta H = 0$
    for(all neighbors of t)
      add contribution to ΔH
    flip spin with Metropolis probability(ΔH)
  end
end

VIDEO: `video08d_messgroessen_r.mkv`

11.3 Ferromagnet: oservables

Repetition:
Average magnetisation:

$$\langle m \rangle = \frac{1}{Z} \sum_{\{\underline{s}\}} m(\underline{s}) \exp(-\beta H(\underline{s})) \tag{61}$$

with $m = \sum_i s_i/N$, $Z = \sum_{\{\underline{s}\}} \exp(-\beta H(\underline{S}))$, $\beta = 1/k_B T$, $N = L^d$ spins.
Small magnetic field B: $H_B(\underline{s}) = -\sum_{\langle i,j \rangle} s_i s_j - B \sum_i s_i \rightarrow$

[Activator]

Calculate $\frac{\partial Z}{\partial B}\big|_{B=0}$

For the suszeptibility

$$\begin{aligned}
\chi &\equiv \left. \frac{\partial \langle m \rangle}{\partial B} \right|_{B=0} \\
&= \left. \frac{\partial}{\partial B} \right|_{B=0} \frac{1}{Z} \sum_{\{\underline{s}\}} m \exp(-\beta H(\underline{s})) \\
&= -\frac{1}{Z^2} \left. \frac{\partial Z}{\partial B} \right|_{B=0} \sum_{\{\underline{s}\}} m \exp(-\beta H(\underline{s})) + \frac{1}{Z} \left. \frac{\partial}{B} \right|_{B=0} \sum_{\{\underline{s}\}} m \exp(-\beta H(\underline{s})) \\
&= -\frac{1}{Z^2} \beta N \left(\sum_{\{\underline{s}\}} m \exp(-\beta H(\underline{s})) \right)^2 + \frac{1}{Z} \beta N \sum_{\{\underline{s}\}} m^2 \exp(-\beta H(\underline{s})) \\
&= \beta N \left(\langle m^2 \rangle - \langle m \rangle^2 \right) = \beta N \sigma^2
\end{aligned} \tag{62}$$

Determination of phase transition: *Binder cumulant* of magnetization

$$b(L,T) = 0.5 \left(3 - \frac{\langle m^4 \rangle}{\langle m^2 \rangle^2} \right) \tag{63}$$

One can show:

- $b(L,T)$ curves for different L intersect (almost) at T_c.
- Average absolute magnetisation per spin $M \equiv \langle |m| \rangle$ behaves (theory of phase transitions):

$$M(T) \sim |T - T_c|^\beta \quad (T < T_c) \tag{64}$$

For finite systems

$$M(T,L) = L^{-\beta/\nu} \tilde{\xi}_1(L/\xi) = L^{-\beta/\nu} \hat{\xi}_1(L^{1/\nu}(T - T_c))$$

where ν describes the divergence of the correlation length at the critical point.