Informatik

- **Kapselung:**
  - Kontrollierter Zugriff auf Methoden bzw. Attribute von Klassen
  - → Speicherelemente Privat machen
  - → getter und setter Funktionen erstellen

- **Fliesskommazahlensysteme:**

$F(\beta, p, e_{min}, e_{max})$

- $\beta \geq 2$, Basis
- $p \geq 1$, Präzision, Stellenzahl
- e_{min}/e_{max}, kleinste/grösste Exponent

→ **normalisierte Darstellung:**

$\pm d_0.d_1 \dots d_{\beta-1} \times \beta^e \quad d_0 \neq 0$

→ eindeutige Darstellung

$\pm \sum_{i=0}^{p-1} d_i \beta^{-i} \cdot \beta^e$

- **Exponent verschieben:**

$0{,}000111 \cdot 2^3 \rightarrow 1{,}11 \cdot 2^{-1}$

- **Wertebereich herausfinden:**

$F^*(2, 3, -4, 4)$ → nur positive Zahlen werden dargestellt

1.) # volle binäre Zahlen (100, 101, 110, 111)
LoP 4 Zahlen ~~(001, 010, 011, 000)~~

2.) Exponent kann 9 Werte haben [-4, 4]

3) 4 · 9 = 36 Zahlen

- **Umrechnung dezimal → binär:**

→ Bsp: 1.1₁₀ $\quad b = \begin{cases} 1, & \text{Falls } x \geq 1 \\ 0, & \text{sonst} \end{cases}$

→ mit Kommazahl beginnen

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	1	0.1	0.2
0.2	0	0.2	0.4
0.4	0	0.4	0.8
0.8	0	0.8	1.6
1.6	1	0.6	1.2
1.2	1	0.2	0.4

→ $1.000\overline{11} \cdot 2^{\boxed{0}}$ → Achtung: Exponent periodisch

- **nicht endliche Binärdarstellung:**

1,1 ; 0,1 ; 1/40 ; 2/5

→ nur Vielfache von 2^{-n}

- **negative Binärzahlen:** (nur für 2^n)

$-16 \rightarrow \boxed{1}10000 \quad \sim -1$

- **maximale Präzision:**

→ entspricht der kleinsten Zahl
→ $\beta^{e_{min}} = 2^{-4}$

- **Strings:** string.length()

```
std::string name = roland;
name[0]  // r   | name.at(0)
name[0] - 'a' // r - a  17
```

- **Bitweise Invertierung:**

$-1 - sum$ (2er-Komplement)

- **in Grossbuchstaben umwandeln:**

```
gross = klein + ('a' - 'A')
```

- **grösste unsigned int Zahl:**

4294967295

- **Negativer Modulo:**

-6 % 5 = -1 $\quad$ -8 % 5 = -3
6 % -5 = 1 $\quad$ 8 % -5 = 3
-6 % -5 = -1 $\quad$ -8 % -5 = -3

- **Typ-Alias:**

```
using int_vec = std::vector<int>
```

- **Beispielprogramme:**
  - **Primzahlen:**

```
bool p(int n){
  int f=2;
  for(; n%f != 0; ++f);
  return n==f;
}
```

  - **KgV:**

```
int g(int a, int b){
  if(a==0) return b;
  return g(b%a, a);
}
```

  - **Dekonstruktor bei LinkedList:**

```
List::~List(){
  Node* node = first;
  while(node != 0){
    Node* tmp = node;
    node = node->next;
    delete tmp;
  }
}
```

  - **n-te Fibonacci-Zahl berechnen:**

```
int f(int x){
  if(x==0) return 0;
  if(x==1) return 1;
  return f(x-1) + f(x-2);
}
```

  - **Primfaktorenzerlegung:**

```
void prime_factors(int n){
for(int d=2; d<=n; ++d){
if(n%d==0){ std::cout<<d<<" "; n=n/d; --d; }}}
```

- **Allgemeines:**
  - **Vorward Declaration:** `int f(int);`

Es gibt eine zirkuläre Abhängigkeit zwischen den Funktionen. Daher muss mindestens eine Funktion vor ihrer Definition deklariert werden. Sonst wäre sie in mindestens einer anderen Funktion nicht sichtbar.

  - **Ebnf programmieren:**

Funktion nur direkt aufrufen, falls sie nichts fälschlich verändert
→ lookahead verwenden

  - **Konstruktor:**

Konstruktoren verwenden
→ Kopierkonstruktor

  - **Ebnf: Darstellung:**

| Alternative
{ } beliebige Wiederholung: 0, 1, ..., n
[ ] Option → 0 oder 1 mal

⚠ Index von Arrays beginnt bei 0!

  - **void Funktionen:**

→ können durch Pointer oder Referenzen auch Datenstrukturen verändern

  - **Kurzschlussauswertung (short-circuit evaluation):**

&& und || : linker Operand zuerst ausgewertet
→ Ergebnis steht dann schon fest
→ rechter Operand wird nicht ausgewertet

  - **XOR:**

x	y	XOR(x,y)
0	0	0
0	1	1
1	0	1
1	1	0

$(x||y)\ \&\&\ !(x\ \&\&\ y) \dots 1\,111 \dots 10001$

  - **negative Binärzahl:**

-(-1) $\quad$ 15
-1 $\quad$ 14
in 2er System $\quad$ 1110
0→1, 1→0 $\quad$ 0001

Typen und Werte:

→ 0xf * 0xa → int, 150

→ unsigned int u = 0 ; u - 5 < 6
→ bool, false → Überlauf

→ double d = 1.5 ; 4/d
→ double, 2.666

→ unsigned int u = 8 ; u - 9 == -1
→ bool, true → Überlauf gleich gross

→ bool a, b ; a || !b && !a || b
→ bool, true → Präzedenzen (!b&&!a)

→ 0xf + 0x2 → int, 30 (2-15)

→ unsigned int u = 1, int i = 10
u - i + 12 → unsigned int, 3
→ u - 1 = 0 → Überlauf - 9 + 9 = 0 → 0+3 = 3

→ 1e4 == 10000

→ 5u / 2u → unsigned int, 2

```
int A[3] = {1,0,2};
int *X = &A[0];
```
→ X[A[1]] → int, 1

```
char s[] = "OTI";
char* n = s; int t[] = {1,0,-1};
int* i = t;
*(n++) += *(i++);
*(++n) += *(++i);
std::cout << s;
```
→ ETH

```
struct S{
  S* n;
  int v;
  S(int V, S* N): n(N), v(V){};
};
S s = S(10, new S(20, &s1));
std::cout << (s.n->n->v);
```
→ 10

```
int y[] = {2,0,1};
int* x = &y[1];
```
→ x[1] == y[0] → bool, false

```
double d = 1.5
while(d != 1){
  std::cout << d << "...";
  d -= 0.1;
}
```
→ infinite loop

- **Konversion bool ⇔ int:**

true → 1 $\quad$ ≠0 → true
false → 0 $\quad$ 0 → false

- **De Morgansche Regeln:**

!(a && b) == (!a || !b) $\quad$!(reich und schön)
!(a || b) == (!a && !b) $\quad$ ==(arm oder hässlich)

- **Implizite Konversion:** (32 bit)

bool → char → int → unsigned int → float → double ~ (64 bit)

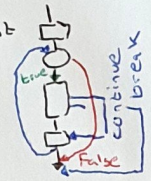
→ mit 1 füllen, bis alle gefärbten Bits voll sind

Precedence:

		Assoziativität
+, -	Vorzeichen	←
!	Negation	
*	Dereferenzierung	
&	Adresse	
*, /, %	Multiplikation, Division, Rest	→
+, -	Addition, Subtraktion	→
<, <=, >, >=	kleiner, kleinergleich, grösser, grössergleich	→
==, !=	gleich, ungleich	→
&&	und	→
||	oder	→
=, +=, -=, *=	Zuweisungen	←

Kontrollfluss:

- For:



Const:

```
const int & i = n  // i: Lese-Alias von n
int & j = n        // j: Lese/Schreib-Alias von n
```

`const int * const a` // int const * const a

- Lo Wertkonstante
- Lo Pointerkonstante

können nicht verändert werden

```
int numerator() const
{return n;}
```

→ const bei einer Memberfunktion bewirkt, dass eine Instanz nicht über diese Funktion verändert wird

Konstruktoren:

```
public:
  rational(int num, int den)
      : n(num), d(den)
      {assert(den != 0)}
```

- default-constructor:

```
public:
  rational()
      : n(0), d(1) {}
```

- this:

Dereferenzieren eines Zeigers gefolgt von einem Memberzugriff

"Folge dem Zeiger zur Membervariable"

- Iterator:

Const Iterator muss bei const Vektoren verwendet werden

Vektoren:

```
#include <vector>          {v1, v2, v3, v4}
std::vector<typ> Name(Grösse, Wert)
```

v.size(), v.push_front(x), v.at(x)
v.push_back(x), v.pop_back(),
v.clear(), → entfernt das letzte Element
v.front(), v.back() → Referenz auf das
v.begin(), v.end() → erste/letzte Element
→ Pointer

- 3er Regel:

Dekonstruktor, Copyconstructor und Zuweisungsoperator definieren

Bit, Byte:

- Bit: eine Stelle einer Binärzahl, entweder 0 oder 1
- Byte: binäre Folge aus 8 Bit

Operatorüberladung:

- binär/unär/Vergleich: +, -, ==, <=, %

```
T operator+(T a, T b){}
```

- Zuweisung/Prefix: -=, ++

```
T& operator+=(T& a, T b){}
```

- Ausgabeoperator:

```
std::ostream & operator<<
(std::ostream & out, T a){
  return out << evtl. etwas ausg.;
}
```

→ Eingabeoperator: istream

Pointer:

```
int a = 2;          i = *(myArray+5);
int * b = &a;   → & Adresse
int c = *b;     → * Dereferenzierung
```

Call by Reference:

```
void swap(int& x, int& y){}
int main(){ swap(a, b)}
```

Set:

```
#include <set>
```

→ speichert eindeutige Elemente
→ Wert des Elements ist auch dessen Identifikation
→ Werte des Elements in einem Set

s.begin(), s.end() // Iterator zu begin, end
s.empty(), s.size(), s.insert(),
s.erase, s1.swap(s2), s.clear
s.find(value)

- Iterator:

```
typedef std::set<User*>
  ::const_iterator cit;
for(cit u = followers.begin(),
u != followers.end(); ++u){
(*u)->receive(message);}
```

Vergleichsoperatoren:

```
a < b   == a < b
a > b   == b < a
a <= b  == !(b < a)
a >= b  == !(a < b)
```

Fliesskommazahlen

→ 1.1 - 1.0 < 0.11 → true

→ jede 32-bit Zahl vom Typ unsigned int kann ohne Wertänderung in den Typ Double konvertiert werden

→ a + c * d + b == d * c + a + b
→ true → gleiche Operationen in gleicher Reihenfolge

→ a - 1 == a → true
→ a ist ein spezieller Double

→ 1.1 und 1.1f haben nicht den gleichen Wert, da 1.1 eine unendliche Binärdarstellung hat

Syntax:

```
int x, y;
std::cin >> x >> y;
std::cout << " bla" << x << "bla"
    << "\n";
```

Vererbung:

```
class Person{
  public:
   Person(string Name)
       : name(Name){}
   string getName(){return name;}
   private:
   string name;
};
class Mitarbeiter: public Person{
   public:
   Mitarbeiter(string Name, int
   Gehalt): Person(Name),
    gehalt(Gehalt){}
   int gehalt;
};
```

[begin, end) is a valid range

Binär	Dezimal	Hexadezimal
1	1	1
10	2	2
11	3	3
100	4	4
101	5	5
110	6	6
111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

16^0	16^1	16^2	16^3
0	**16**	**256**	**4096**
1	16	256	4096
2	32	512	8192
3	48	768	12288
4	64	1024	16384
5	80	1280	20480
6	96	1536	24576
7	112	1792	28672
8	128	2048	32768
9	144	2304	36864
10	160	2560	40960
11	176	2816	45056
12	192	3072	49152
13	208	3328	53248
14	224	3584	57344
15	240	3840	61440

Binärzahlen <-> Dezimalzahlen

16	10000	32	100000	48	110000
17	10001	33	100001	49	110001
18	10010	34	100010	50	110010
19	10011	35	100011	51	110011
20	10100	36	100100	52	110100
21	10101	37	100101	53	110101
22	10110	38	100110	54	110110
23	10111	39	100111	55	110111
24	11000	40	101000	56	111000
25	11001	41	101001	57	111001
26	11010	42	101010	58	111010
27	11011	43	101011	59	111011
28	11100	44	101100	60	111100
29	11101	45	101101	61	111101
30	11110	46	101110	62	111110
31	11111	47	101111	63	111111

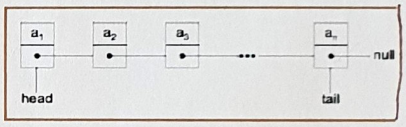
Linked List

```
//headerfile
class LinkedList {
   Node * head;
   public:
      LinkedList() //Konstruktor1
   {
      head = NULL;
   }
   LinkedList (int n); //Konstruktor2
   void print();
   void addNode(intv);
   void insert(intv, Node * after);
   void delete(Node * after);
   void find(intv);
};
//bodyfile
LinkedList :: LinkedList (int n) //vorneeinsetzen
{
   head = NULL;
   Node * newNode;
   int v;
   for (int i = 0; i < n; i++) {
      cout << "Enter number:";
      cin >> v;
      newNode = new Node;
      newNode- > data = v;
      newNode-> next = head;
      head = newNode;
   }
}
void LinkedList :: print() //Liste auf Konsole ausgeben
{
   Node * p = head;
   while (p != NULL) {
      cout << p-> data << "";
      p = p-> next;
   }
}
void LinkedList :: addNode (int v) //hinten einsetzen
{
   Node * newNode;
   Node * current;
   if (head == NULL) //Liste noch leer
   {
      newNode = new Node;
      newNode-> data = v;
      newNode-> next = NULL;
      head = newNode;
   } else {
      current = head;
      while (current-> next != NULL)
         current = current-> next;
      newNode = new Node;
      newNode-> data = v;
      newNode-> next = NULL;
      current-> next = newNode;
   }
}
```

```
void LinkedList :: insert (int v, Node * after) //einsetzen
{
   Node * newNode = new Node;
   newNode-> data = v;
   newNode-> next = NULL;
   newNode-> next = after-> next;
   after-> next = newNode;
}
void LinkedList :: delete (Node * after) //löschen
{
   Node * deleteNode = after-> next;
   after-> next = deleteNode-> next;
   delete deleteNode;
}
void LinkedList :: find (int v) //Element v finden
{
   Node * current = head;
   while (current != NULL) {
      if (current-> data == v)
         break;
      current = current-> next;
   }
   if (current == NULL)
      cout << "Elementnotfound\n";
}
```

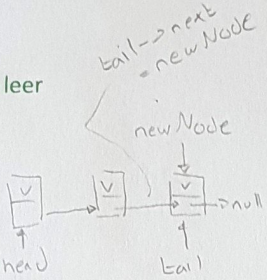
Queue

Prinzip nennt man \First In First Out"




```
struct nodeType {
   int data ;
   nodeType * next ;
};
typedef struct nodeType Node ;
Node * head = NULL ;
Node * tail = NULL ;
//headerfile
class Queue {
   public:
    Queue(); //Konstruktor
     ~Queue(); //Destruktor
    void put (int value); //neues Element ans Ende
    int get(); //erstes Element entfernen und zurückgeben
   private:
    Node * head;
    Node * tail;
};
//bodyfile
Queue :: Queue() //Konstruktor
{
   head = NULL;
   tail = NULL;
}
Queue :: ~Queue() //Destruktor
{
   Node * toDelete = head;
   while (toDelete != NULL) {
      head = head-> next;
      delete toDelete;
      toDelete = head;
   }
}
```

```
void Queue::put (int value) {
   Node * newNode = new Node;
   newNode-> data = value;
   newNode-> next = NULL;
   if (head == NULL) //Queue noch leer
   {
      head = newNode;
      tail = newNode;
   } else {
      tail-> next = newNode;
      tail = newNode;
   }
}
int Queue :: get() {
   if (head != NULL) {
      Node * toDelete = head;
      int result = head-> data;
      head = head-> next;
      if (head == NULL) //falls Queue jetzt leer
         tail = NULL;
      delete toDelete;
      return result;
   } else
      return 0;
}
```



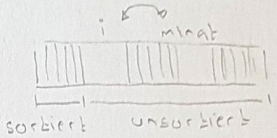
EBNF

```
bool Expression(std::istream & is);
// ExpressionList = Expression { ',' Expression }.
bool ExpressionList(std::istream & is) {
   if (!Expression(is)) return false;
   while (has(is, ', ')) {
      if (!Expression(is)) return false;
   }
   return true;
}
// Designator = Identifier {'.' Identifier | '(' [ExpressionList] ')'}.
bool Designator(std::istream & is) {
   if (!Identifier(is)) return false;
   while (true) {
      if (has(is, '.')) {
         if (!Identifier(is)) return false;;
      } else if (has(is, '(')) {
            bool ignore = ExpressionList(is);
            if (!has(is, ')')) return false;;
      }
      else
         return true;
   }
}
// Simple = Designator | Integer.
bool Simple(std::istream & is) {
   return Integer(is) || Designator(is);
}
// Expression = Simple [':' Simple].
bool Expression(std::istream & is) {
   if (!Simple(is)) return false;
   return !has(is, ': ') || Simple(is);
}
```

Selection Sort

Das kleinste Element wird gesucht und mit dem ersten Element der Liste vertauscht. Der Algorithmus wird ohne die bereits geordneten Elemente wiederholt.

```
void selectionSort (int* array, int length){
   int i, j, min, minat ;
   for (i = 0; i < (length-1); i++){
      minat = i; //Index vom Minimum
      min = array[i]; //Wert vom Minimum
      for (j = i+1; j < length; j++){ //aktuelles Min vergleichen
         if (min > array[j]){ //falls neues Minimum gefunden
            minat = j; //Index wechseln
            min = array[j]; //Wert wechseln
         }
      }
//Minimum mit 1. Wert der ungeordneten Liste vertauschen
      int temp = array[i];
      array[i] = array[minat];
      array[minat] = temp;
   }
}
```



Bubble Sort

Sortieren durch wiederholtes Vergleichen und eventuelles Vertauschen benachbarter array-Elemente (falls das Linke grösser als das Rechte). Nach dem ersten Durchlauf ist das grösste Element an der richtigen Position.

```
void bubbleSort (int* array, int length){
   for (int i = length-1; i > 0; i--){ //outerloop
      for (int j = 1; j <= i; j++){ //innerloop
         if (array[j-1] > array[j]){
            int t = array[j-1];
            array[j-1] = array[j];
            array[j] = t;
         }
      }
   }
}
```

grösstes Element wandert nach rechts

Insertion Sort

Ein Element nach dem anderen in der Liste wird betrachtet. Man sucht seine Position in der bereits sortierten Liste der vorhergehenden Elemente und fügt es ein. Die nachfolgenden Elemente müssen verschoben werden.

```
void insertionSort (int* array, int length){
   int i, j, tmp;
   for (i = 1; i < length; i++){
      j=i;
      while (j > 0 && array[j-1] > array[j]){
         tmp = array[j];
         array[j] = array[j-1];
         array[j-1] = tmp;
         j--;
      }
   }
}
```