Rekursjon

- Hva er *rekursjon*?
- Hvordan programmere rekursivt?

Dagens programmer finnes på
https://www.uio.no/studier/emner/matnat/ifi/IN1010/v25/programmer/rekursjon/.

Å beregne fakultet

Den matematiske funksjonen **n fakultet** (med notasjonen **n!**) er definert slik:

$$
\begin{aligned}
1! &= 1 &&= 1 \\
2! &= 1 \times 2 &&= 2 \\
3! &= 1 \times 2 \times 3 &&= 6 \\
4! &= 1 \times 2 \times 3 \times 4 &&= 24 \\
n! &= 1 \times 2 \times \cdots \times n = &&
\end{aligned}
$$

En løsning med løkke

Fakultet1.java

```
class Fakultet1 {
    static long fak (int n) {
        long res = 1;
        for (int i = 1;  i <= n;  i++)
            res = res * i;
        return res;
    }

    public static void main (String[] args) {
        int nPar = Integer.parseInt(args[0]);
        for (int i = 1;  i <= nPar;  i++)
            System.out.println(i + "! = " + fak(i));
    }
}
```

```
$ java Fakultet1 6
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
```

En alternativ løsning

$$
\begin{aligned}
1! &= 1 && = && 1\\
2! &= 1 \times 2 \underbrace{1}_{1!} \times 2 && = && 2\\
3! &= 1 \times 2 \times 3 \underbrace{1 \times 2}_{2!} \times 3 && = && 6\\
4! &= 1 \times 2 \times 3 \times 4 \underbrace{1 \times 2 \times 3}_{3!} \times 4 && = && 24\\
&\vdots\\
n! &= 1 \times 2 \times \cdots \times n \underbrace{1 \times 2 \times \cdots \times n-1}_{(n-1)!} \times n = (n-1)! \times n
\end{aligned}
$$

Fakultet2.java

```
class Fakultet2 {
    static long fak (int n) {
        long res;
        if (n == 1) res = 1;
        else res = fak(n-1) * n;
        return res;
    }

    public static void main (String[] args) {
        int nPar = Integer.parseInt(args[0]);
        for (int i = 1;  i <= nPar;  i++)
            System.out.println(i + "! = " + fak(i));
    }
}
```

```
$ java Fakultet2 6
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
```

Rekursjon

Programkode som kaller seg selv, kalles **rekursiv kode**.

- Noen problemer er rekursive av natur (for eksempel en kompilator eller Hanois tårn)
- Rekursiv programmering er spesielt nyttig for programmer som skal søke ulike veier til en mulig løsning (for eksempel en labyrint eller sjakkspill)
- Noen programmerere foretrekker rekursive løsninger.

Hvordan tenke om rekursjon

Programmerere tenker ulikt om rekursjon.

Noen ser ikke problemet i det hele tatt.

En metode er en metode, og ferdig med det!

Sitat fra læreboken *Big Java*:

The key to the successfull design of a recursive method is *not to think about it*.

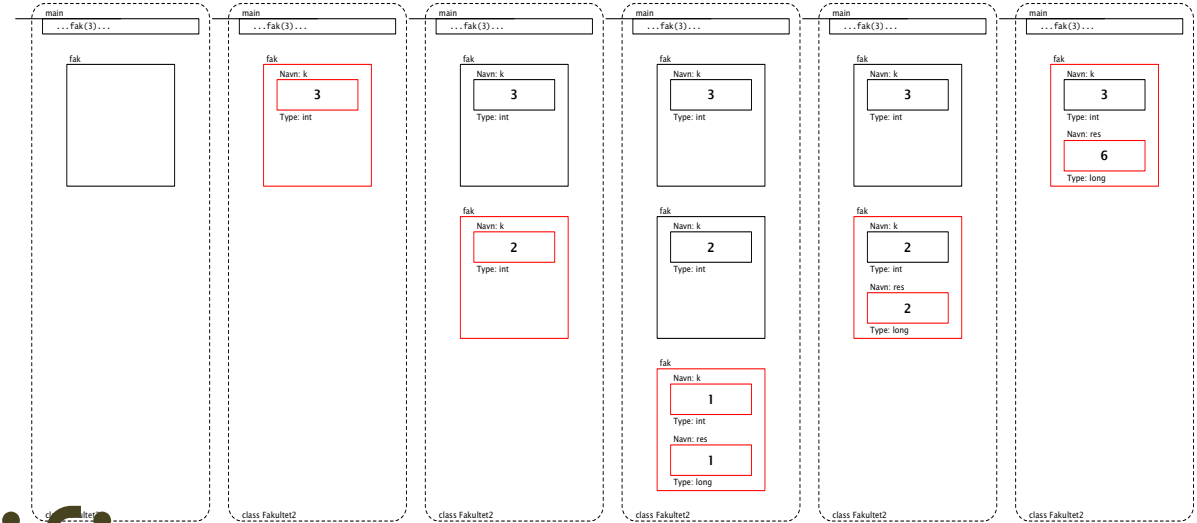
En visuell illustrasjon av rekursjon

Noen tenker rekursjon visuelt ved at et bilde gjentas i bildet, innover og innover . . .

Eksempel: En nederlandsk kakoboks



long res; if (k==1) res = 1; else res = fak(n-1)*n; return res;



Fibonacci-tallene

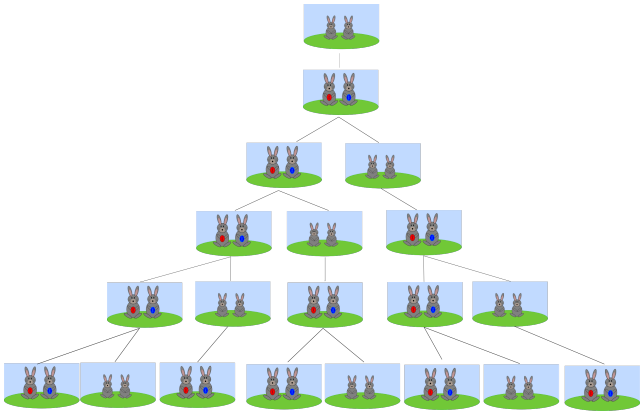
Tallsekvensen

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

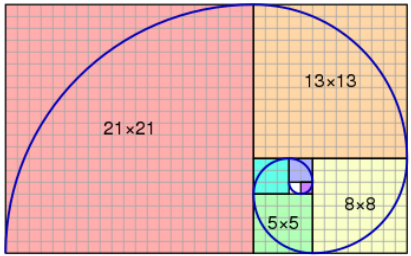
der hvert tall er summen av de to foregående, forekommer usedvanlig ofte i naturen.

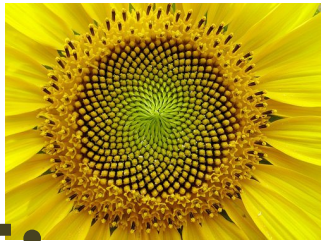
Leonardo Fibonacci
(ca 1170–ca 1250) fant denne tallsekvensen som svar på hvordan kaniner formerer seg.

- Kaniner blir kjønnsmodne etter 1 måned.
- Deretter får de et kaninpar hver måned.
- Kaniner lever evig.



Overraskende ofte finner vi disse tallene i naturen:









Vi skal altså programmere sekvensen gitt ved

$$F_n = F_{n-1} + F_{n-2} \quad \text{og} \quad F_1 = 1$$

Fibonacci.java

```
class Fibonacci {
    static int fib (int n) {
        if (n == 1) return 1;
        else return fib(n-1) + fib(n-2);
    }

    public static void main (String[] args) {
        int antall = Integer.parseInt(args[0]);
        for (int i = 1;  i <= antall;  i++)
            System.out.println("Fib(" + i + ") = " + fib(i));
    }
}
```

Dette gikk ikke bra:

```
$ java Fibonacci 10
Fib(1) = 1
Exception in thread "main" java.lang.StackOverflowError
        at Fibonacci.fib(Fibonacci.java:4)
        at Fibonacci.fib(Fibonacci.java:4)
        at Fibonacci.fib(Fibonacci.java:4)
        at Fibonacci.fib(Fibonacci.java:4)
        at Fibonacci.fib(Fibonacci.java:4)
        at Fibonacci.fib(Fibonacci.java:4)
        at Fibonacci.fib(Fibonacci.java:4)
              :
```

Problemet oppstår i kallet **`fib(2)`** som så kaller på

1. `fib(1)` (går OK)
2. `fib(0)` som kaller på `fib(-1)` som ...

Huskeregler for rekursiv programmering

Når vi programmerer rekursivt, må vi alltid huske på

- Det rekursive kallet må være et kall på et *enklere* problem.
- Vi må sørge for at det *trivielle* problemet håndteres skikkelig.

En riktig versjon av koden:

Fibonacci2.java

```
class Fibonacci2 {
    static int fib (int n) {
        if (n <= 2) return 1;
        else return fib(n-1) + fib(n-2);
    }

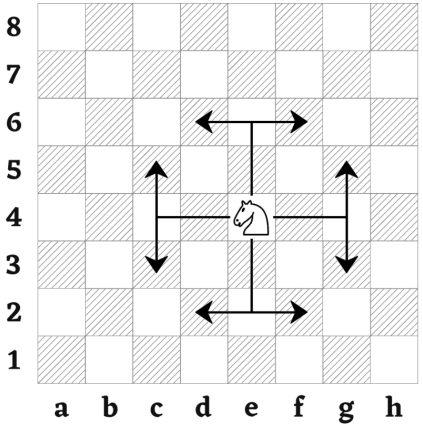
    public static void main (String[] args) {
        int antall = Integer.parseInt(args[0]);
        for (int i = 1;  i <= antall;  i++)
            System.out.println("Fib(" + i + ") = " + fib(i));
    }
}
```

En komplett springertur

Et vanlig hobbyproblem i sjakk er:

> *Flytt springeren slik at den er innom alle rutene på et sjakkbrett.*

Dette kan vi løse ved å prøve alle muligheter.



```
class KnightsTour {
  static int[][] brett;
  static int bredde, bredde2;
  static long antForsøk;

  public static void main (String[] args) {
    bredde = Integer.parseInt(args[0]);
    bredde2 = bredde * bredde;
    brett = new int[bredde][bredde];
    flytt(1, 0, 0);
    System.out.printf("Ingen løsninger etter %,d trekk!\n",
                      antForsøk);
  }

  static void tegnBrett () {
    for (int rx = bredde-1;  rx >= 0;  rx--) {
      for (int kx = 0;  kx < bredde;  kx++) {
        System.out.printf("%4d",brett[kx][rx]);
      }
      System.out.println();
    }
  }
```

```
  static void flytt (int trekk, int r, int k) {
    ++antForsøk;

    // Er det lov å flytte hit?
    if (r < 0 || r >= bredde) return;
    if (k < 0 || k >= bredde) return;
    if (brett[r][k] > 0) return;

    // Det er lov, så vi prøver å flytte hit.
    brett[r][k] = trekk;

    // Har vi funnet en løsning?
    if (trekk == bredde2) {
      System.out.printf("Løsning etter %,d trekk\n",
                        antForsøk);
      tegnBrett();  System.exit(1);
    }

    // Prøv å flytte videre:
    flytt(trekk+1, r+2, k+1);  flytt(trekk+1, r+2, k-1);
    flytt(trekk+1, r+1, k+2);  flytt(trekk+1, r+1, k-2);
    flytt(trekk+1, r-1, k+2);  flytt(trekk+1, r-1, k-2);
    flytt(trekk+1, r-2, k+1);  flytt(trekk+1, r-2, k-1);

    // Det gikk ikke; da trekker vi oss tilbake.
    brett[r][k] = 0;
  }
}
```

Metoden **`flytt (int trekk, int r, int k)`** gjør jobben. Parametrene er:

- **`trekk`** er trekknummeret.
- **`r`** og **`k`** er *rad* og *kolonne* vi prøver å flytte til.

1. Vi øker antall forsøk.
2. Vi sjekker om det er lov å flytte til denne posisjonen.
3. Vi noterer trekket i rutenettet.
4. Vi sjekker om vi nå er ferdig.
5. Vi prøver de 8 mulige(?) trekkene videre.
6. Hvis ingen løsning, nuller vi ut forsøket og returnerer.

Da kan vi finne løsningene:

```
$ javac KnightsTour.java
$ java KnightsTour 1
Løsning etter 1 trekk
   1
$ java KnightsTour 2
Ingen løsninger etter 9 trekk!
$ java KnightsTour 3
Ingen løsninger etter 121 trekk!
$ java KnightsTour 4
Ingen løsninger etter 17 785 trekk!
$ java KnightsTour 5
Løsning etter 381 509 trekk
  21  16  11   4  23
  10   5  22  17  12
  15  20   7  24   3
   6   9   2  13  18
   1  14  19   8  25
```

```
$ java KnightsTour 6
Løsning etter 815 823 trekk
  26  19  22   5  10   7
  23  32  25   8  21   4
  18  27  20  11   6   9
  33  24  31  14   3  12
  28  17   2  35  30  15
   1  34  29  16  13  36
$ java KnightsTour 7
Løsning etter 695 660 226 trekk
  49  40  29  38  27   8  23
  30  37  26  21  24   5  10
  41  48  39  28   9  22   7
  36  31  20  25   6  11   4
  47  42  45  32   3  14  17
  44  35   2  19  16  33  12
   1  46  43  34  13  18  15
```

Hanois tårn

Den oppdiktete bakgrunnen for spillet fra 1883 er:

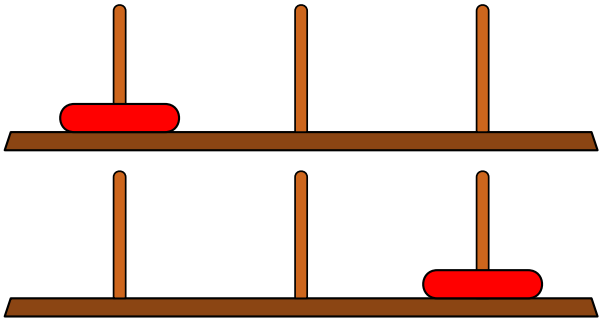
I et tempel i Hanoi står tre stolper, og på den venstre ligger 60 ringer av ulik størrelse, sortert med den minste øverste.

Oppgaven er å flytte ringene til den høyre staken, men

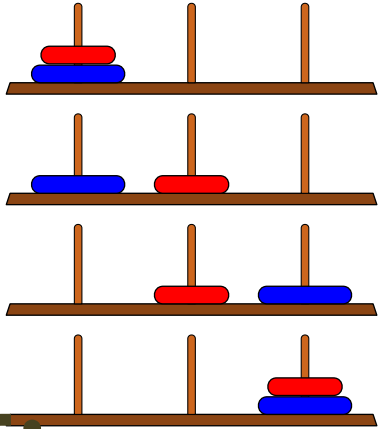
- Ringene må flyttes én og én.
- En ring må aldri legges oppå en mindre ring.
- Den midtre stolpen kan benyttes underveis.



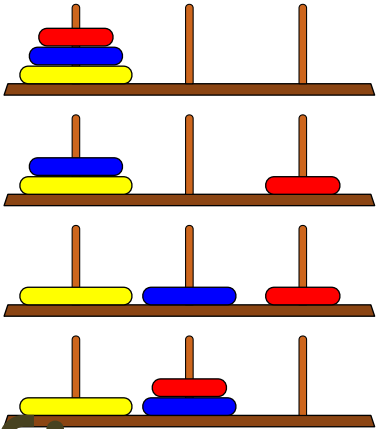
Hanois tårn med 1 ring

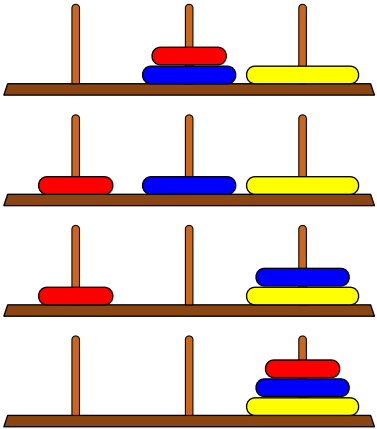


Hanois tårn med 2 ringer



Hanois tårn med 3 ringer





Programmere Hanois tårn

- Vi kaller de tre stolpene **A**, **B** og **C**.
- Vi skal flytte alle ringene fra **A** til **C**.
- For å kunne flytte den største ringen, må vi først flytte alle de andre ringene til hjelpestolpen **B**.
- Så kan vi flytte den største ringen fra **A** til **C**.
- Etterpå må vi flytte alle ringene fra hjelpestolpen **B** til **C** (oppå den største ringen).

Og da er jobben gjort.

Hanoi.java

```
class Hanoi {
    static void flytt (int n, char fra, char via, char til) {
        if (n == 1) {
            System.out.println("Flytt " + fra + " til " + til);
        } else {
            flytt(n-1, fra, til, via);
            flytt(1, fra, via, til);
            flytt(n-1, via, fra, til);
        }
    }

    public static void main (String[] args) {
        int antall = Integer.parseInt(args[0]);
        flytt(antall, 'A', 'B', 'C');
    }
}
```

```
$ java Hanoi 1
Flytt A til C

$ java Hanoi 2
Flytt A til B
Flytt A til C
Flytt B til C

$ java Hanoi 3
Flytt A til C
Flytt A til B
Flytt C til B
Flytt A til C
Flytt B til A
Flytt B til C
Flytt A til C
```

```
$ java Hanoi 4
Flytt A til B
Flytt A til C
Flytt B til C
Flytt A til B
Flytt C til A
Flytt C til B
Flytt A til B
Flytt A til C
Flytt B til C
Flytt B til A
Flytt C til A
Flytt B til C
Flytt A til B
Flytt A til C
Flytt B til C
```

Hvor mange trekk trenger man?

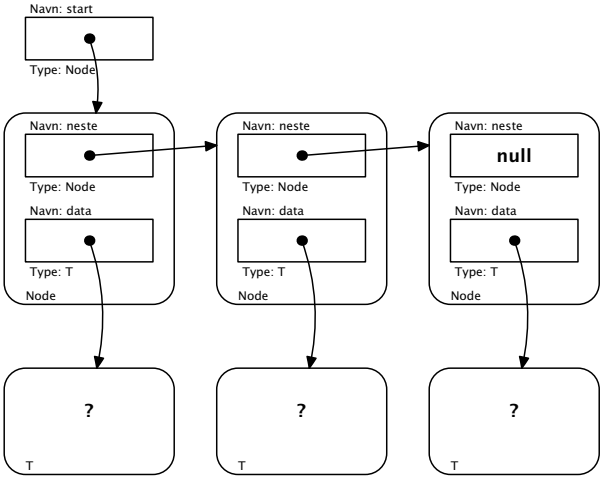
Leketøysvarianten har 8 ringer. Man må da foreta $2^8 - 1 = 255$ flyttinger.

Den originale «legenden» sier at det er 60 ringer. Da trengs $2^{60} - 1 \approx 10^{18}$ flyttinger.

Hvis vi antar at munkene kan flytte én ring hvert sekund, vil det ta omtrent 36 534 658 086 år før verden går under.

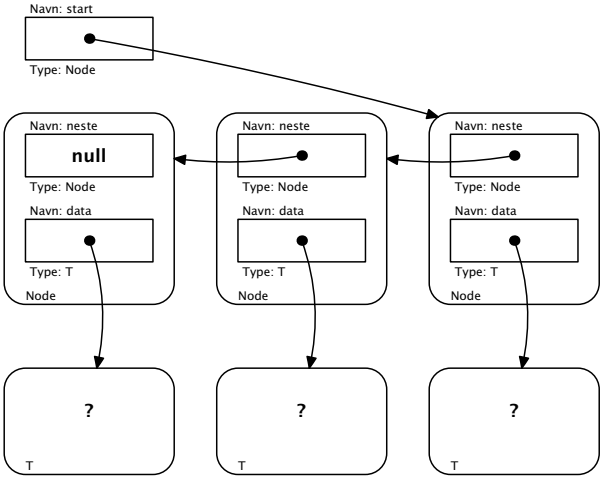
Enveislister

Dette er en vanlig enkelkjedet liste slik vi nå har sett så mange av.



Hvordan kan vi enklest snu en slik liste?

Oppgave: Skriv en metode **reverse** som snur listen til



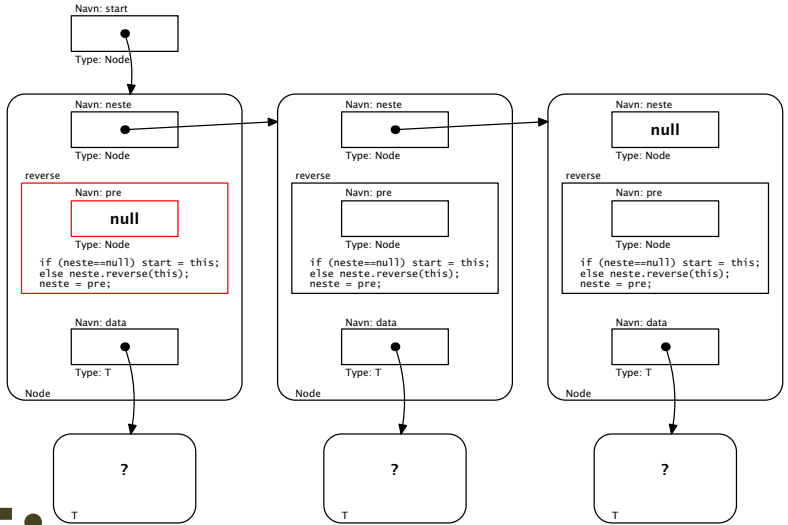
```
class Lenkeliste<T> implements Liste<T> {
    class Node {
        Node neste = null;
        T data;

        Node (T x) { data = x; }

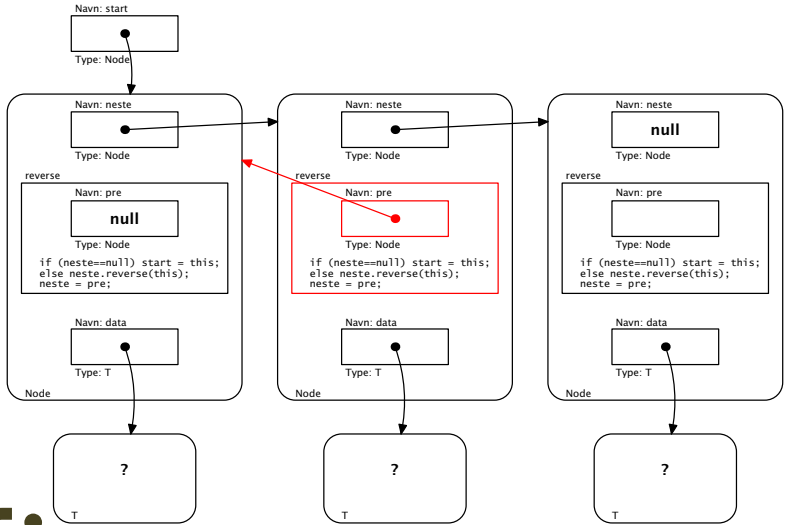
        void reverse (Node pre) {
            if (neste == null) start = this;
            else neste.reverse(this);
            neste = pre;
        }
    }
    private Node start = null;

    public void reverse () {
        if (start != null)
            start.reverse(null);
    }
```

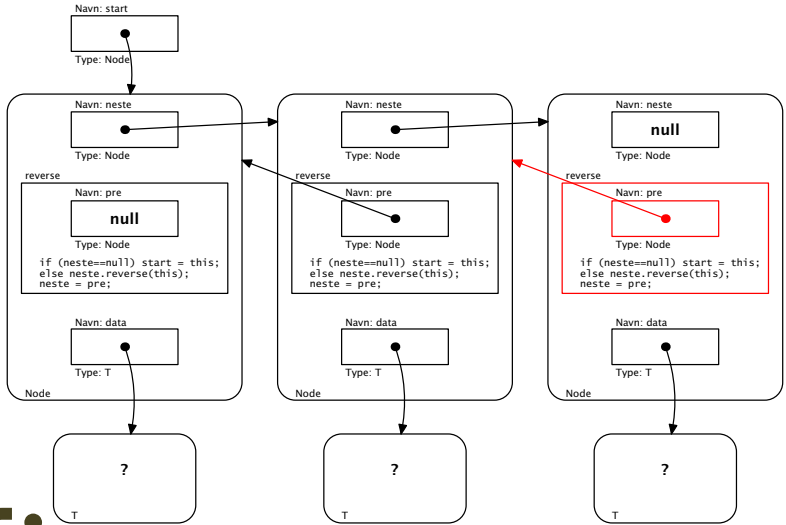
Metoden reverse



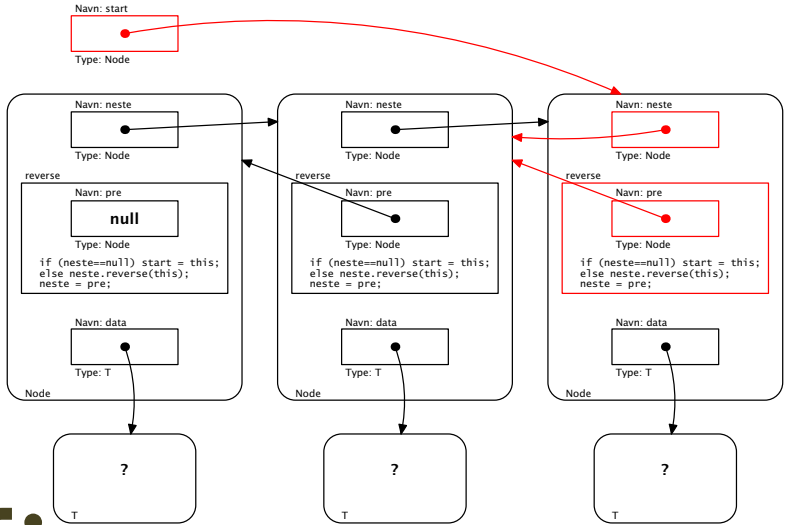
Metoden reverse



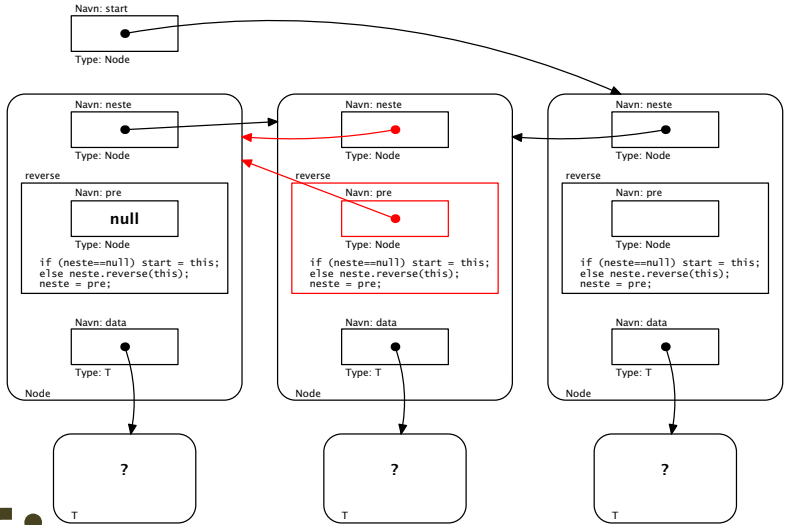
Metoden reverse



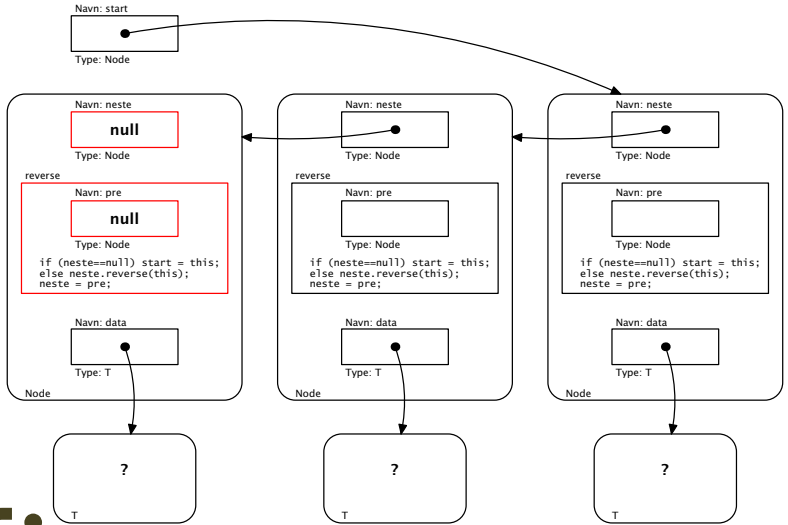
Metoden reverse



Metoden reverse



Metoden reverse



Metoden reverse

