Probleem Oplossend Denken I

1 Inleiding

Oefeningen

- De sequentie
- De Selectie
- De Iteratie

1.1 Basisstructuren

1.1.1 De sequentiestructuur

```
opdracht 1
opdracht 2
opdracht n
```

Voorbeeld: **Algoritme** Bereken BMI

```
VOERUIT(scherm, "geef lengte in meter: ")
VOERIN(klavier, lengte)
VOERUIT(scherm, "geef gewicht in kilo: ")
VOERIN(klavier, gewicht)
bodyMassIndex <- gewicht/(lengte . lengte)
RETOUR bodyMassIndex
```

1.1.2 De selectiestructuur

```
ALS voorwaarde DAN
    component1
ANDERS
    component2
EINDE ALS
```

De eenzijdige Selectie:

```
ALS voorwaarde DAN
```

```
    component1
EINDE ALS
```

Voorbeeld: **Algoritme** Evalueer BMI.

```
VOERUIT(scherm, "Geef BMI:")
VOERIN(klavier, bodyMassIndex)
ALS ((18,5 ≤ bodyMassIndex) EN (bodyMassIndex ≤ 25)) DAN
    VOERUIT(scherm, "Gezond")
ANDERS
    VOERUIT(scherm, "Risico")
EINDE ALS
```

De Geneste selectiestructuur:

```
VOERUIT(scherm, "Geef BMI:")
VOERIN(klavier, bodyMassIndex)
ALS bodyMassIndex < 18,5 DAN
    VOERUIT(scherm, "Risico voor ondergewicht")
ANDERS
    ALS bodyMassIndex > 25 DAN
        VOERUIT(scherm, "Risico voor obesitas")
    ANDERS
        VOERUIT(scherm, "Gezond")
    EINDE ALS
EINDE ALS
```

1.1.3 De iteratiestructuur

```
ZOLANG iteratievoorwaarde DOE
    iteratiecomponent
EINDE ZOLANG
```

> Er is geen do-while lus!

Voorbeeld: **Algoritme** Som van de eerste 10 strikt positieve gehele getallen.
Via `while` lus.

```
i <- 1
som <- 0
ZOLANG i ≤ 10 DOE
    som <- som + i
    i <- i + 1
```

```
EINDE ZOLANG
VOERUIT(scherm, "som = " som)
```

Alternatief (for-loop):

> i moet niet verhoogd worden in deze lus.

```
som <- 0
VOOR i = 1 TOT 10 DOE
    som <- som + i
EINDE VOOR
VOERUIT(scherm, "som = " som)
```

Met Stappen:

```
som <- 0
VOOR i = 1 TOT 10 STAP 2 DOE
    som <- som + i
EINDE VOOR
VOERUIT(scherm, "som = " som)
```

1.2 Gebruik van methodes

Sjabloon

```
naamAlgoritme (I: ...): ...
    * Preconditie: ...
    * Postconditie: ...
    * Gebruikt: ...
BEGIN
    1: ...
EINDE
```

1.3 Voorbeelden

1.3.1 Bepalen van het maximum van drie getallen

```
bepaalMaximum (I: a, b, c: gehele getallen): x: geheel getal
    * Preconditie: a, b en c zijn drie gehele getallen.
    * Postconditie: het maximum van drie getallen werd bepaald.
    * Gebruikt: /
BEGIN
    x <- a
```

```
    ALS (b > x) DAN
        x <- b
    EINDE ALS
    ALS (c > x) DAN
        x <- c
    EINDE ALS
    RETOUR X
EIND
```

1.3.2 Bepaal het aantal priemgetallen kleiner dan n.

```
telPriemgetallen (I: n: geheel getal) : aantal: geheel getal
    * Preconditie: n is een natuurlijk getal.
    * Postconditie: het aantal priemgetallen kleiner dan n werd
geretourneerd.
    * Gebruikt: /
BEGIN
    aantal <- 0
    p <- 2
    ZOLANG (p < n) DOE
        deler <- 2
        ZOLANG ((deler < p) EN (p MOD deler ≠ 0)) DOE
            deler <- deler + 1
        EINDE ZOLANG
        ALS (deler = p) DAN
            antal <- aantal + 1
        EINDE ALS
        p <- p + 1
    EINDE ZOLANG
    RETOUR aantal
EINDE
```

> Waarom tot vierkantswortel van n lopen:
>
> Stel $n = n_1 \times n_2$
>
> dan $n_1 \leq \sqrt{n}$ of $n_2 \leq \sqrt{n}$
>
> Bewijs
>
> Stel $n_1 > \sqrt{n}$ en $n_2 > \sqrt{n}$
>
> $n = n_1 \times n_2 > \sqrt{n} \times \sqrt{n} = n$
>
> Dus, n > n, kan niet = contradictie

1.3.3 Methode 2: De zeef van Eratosthenes

2 ***3*** 4 ***5*** 6 ***7*** 8 9 10 ***11*** 12 ***13*** 14 15 16 ***17*** 18 ***19*** 20 21 22 ***23*** 24 25 26 27 28 ***29***

```
telPriemgetallenEratosthenes (I: n: geheel getal) : aantal: geheel getal
    * Preconditie: n is een natuurlijk getal.
    * Postconditie: het aantal priemgetallen kleiner dan n werd
geretourneerd.
    * Gebruikt: /
BEGIN
    noteer de rij van natuurlijk getallen 2, 3, ..., n - 1
    p <- 2
    aantal <- 0
    ZOLANG (p < n) DOE
        schrap in de rij van getallen alle veelvouden van p
        aantal <- aantal + 1
        ALS alle elementen uit de rij zijn geschrapt DAN
            p <- n
        ANDERS
            p <- het eerste niet geschrapte element
        EINDE ALS
    EINDE ZOLANG
    RETOUR aantal
EINDE
```

2 De uitvoeringstijd van een algoritme

Oefeningen

> De tijd is rechtevenredig met het aantal instructies die uitgevoerd worden.
>
> We nemen aan dat alle basis instructies even lang duren, bijvoorbeeld: optelling, aftrekken, deling, vermenigvuldiging, ...

2.1 De uitvoeringstijd van een algoritme

> Het aantal instructies exact gaan tellen.

2.1.1 Voorbeeld 1

```
BEGIN
    kwadraat <- n . n
    RETOUR (kwadraat)
EINDE
```

	# instructies	# keer	totaal
kwadraat <- n x n	2	1	2
RETOUR(kwadraat)	1	1	1
			3

T(n) = 3

2.1.2 Voorbeeld 2

```
BEGIN
    som <- 0
    VOOR i = 1 TOT n DOE
        som <- som + i . i
    EINDE VOOR
    RETOUR (som)
EINDE
```

	# instructies	# keer	totaal
som <- 0	1	1	1
VOOR i = 1 TOT n DOE	2	n + 1	2n + 2
som <- som + i . i	3	n	3n
EINDE VOOR			
RETOUR (som)	1	1	1
			5n + 4

T(n) = 5n + 4

> Een `VOOR` lus heeft altijd 2 instructies.

2.1.3 Voorbeeld 3

```
BEGIN
    grootste <- 0
    VOOR i = 0 TOT n - 1 DOE
        ALS a[i] < grootste DAN
            grootste <- a[i]
        EINDE ALS
    EINDE VOOR
    RETOUR (grootste)
EINDE
```

	# instructies	# keer	totaal
grootste <- 0	1	1	1
VOOR i = 0 TOT n - 1 DOE	2	n + 1	2n + 2
ALS a[i] < grootste DAN	**1** c		
grootste <- a[i]	**1** c	n	cn
EINDE ALS			
EINDE VOOR			
RETOUR (grootste)	1	1	1

			(2 + c)n + 4

T(n) = (2 + c)n + 4

2.1.4 Voorbeeld 4

```
BEGIN
    som <- 0
    VOOR i = 0 TOT n DOE
        VOOR j = 1 TOT n DOE
            som <- som + i . j
        EINDE VOOR
    EINDE VOOR
    RETOUR (som)
EINDE
```

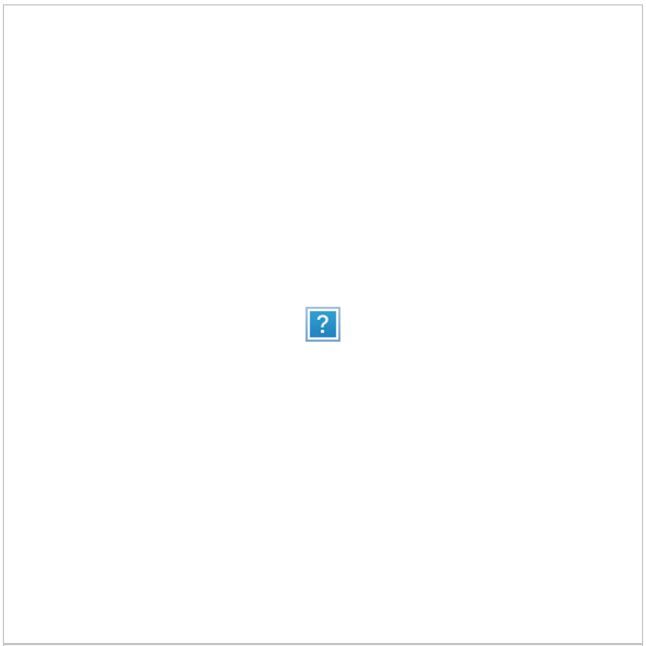
	# instructies	**# keer**	**totaal**
som <- 0	1	1	1
VOOR i = 0 TOT n DOE	2	n + 1	2n + 2
VOOR j = 1 TOT n DOE	2	(n + 1)n	$2n^2 + 2n$
som <- som + i . j	3	n^2	$3n^2$
EINDE VOOR			
EINDE VOOR			
RETOUR (grootste)	1	1	1
			$5n^2 + 4n + 4$

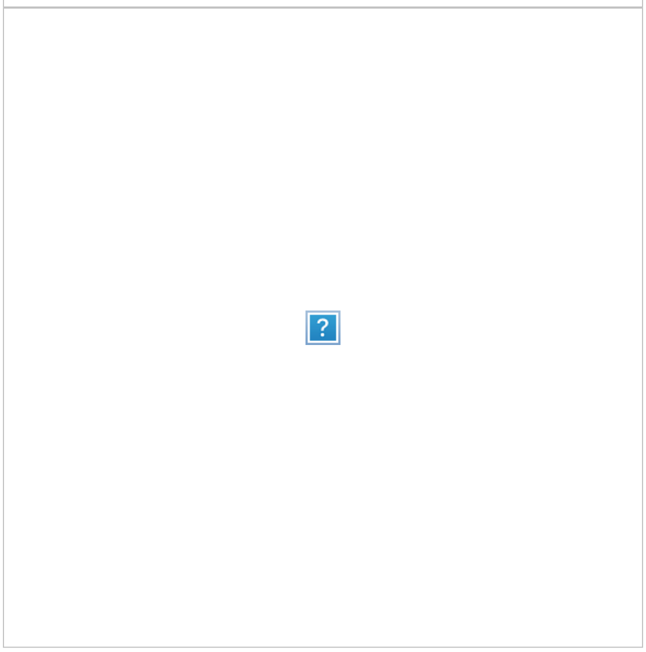
$T(n) = 5n^2 + 4n + 4$
$T(n) = \Theta(n^2)$

> **Examen:** zorg dat je er de Θ bij zet!

2.2 Asymptotische analyse (Θ notatie)

> **EXAMEN:** bepaal theta notatie. (Big Θ Notation).





2.2.1 Voorbeeld 1

```
som <- 0
VOOR i = 1 TOT n DOE
    som <- som + i
EINDE VOOR
```

$T(n) = c_1 + c_2n$
$= \Theta(n)$

2.2.2 Voorbeeld 2

```
som <- 0
VOOR i = 1 TOT n DOE
    VOOR j = 1 TOT i DOE
        som <- som + j
    EINDE VOOR
EINDE VOOR
```

i	# keer lijn 4
1	1
2	2
3	3
n	n

1 + 2 + 3 + ... + n
= ((n + 1) n) / 2

> `((n + 1) n) / 2` is een geslote formule.

$T(n) = \Theta(n^2)$

2.2.3 Voorbeeld 3

```
som <- 0
VOOR i = 1 TOT n DOE
    VOOR j = 1 TOT n DOE
        som <- som + j
    EINDE VOOR
EINDE VOOR
```

$T(n) = \Theta(n^2)$

2.2.4 Voorbeeld 4

Stel: $n = 2^k$

```
som <- 0
i <- 1
ZOLANG i ≤ n DOE
```

```
    VOOR j = 1 TOT n DOE
        som <- som + j
    EINDE VOOR
    i <- i . 2
EINDE ZOLANG
```

bv.: $n = 8 = 2^3$

i	# keer lijn 6
1	8 (n keer)
2	8 (n keer)
4	8 (n keer)
8	8 (n keer)
16	/
	(k + 1)n

T(n) = ((lg(n)) + 1) n
T(n) = n . lg(n) + n
T(n) = Θ(ng lg(n))

> In Θ notatie zijn alle `log`, `lg`, `ln` gelijk. Ze verschillen van een factor die geen rol speelt bij deze notatie.

2.2.5 Voorbeeld 5

Stel: $n = 2^k$

```
som <- 0
i <- 1
ZOLANG i ≤ n DOE
    VOOR j = 1 TOT i DOE
        som <- som + j
    EINDE VOOR
    i <- i . 2
EINDE ZOLANG
```

i	# keer lijn 5
1	1

2	2
4	4
8	8
16	/
	(k + 1)n

$T(n) = 1 + 2 + 4 + 8 + ... + 2^k$
$a = 2$
$= 2^{k+1} - 1 / 2 - 1 = 2^{k+1} - 1 = 2 . 2^k - 1 = 2n - 1$
$T(n) = \Theta(n)$
Formule:
$S_k = 1 + a + a^2 + a^3 + ... + a^k$
$a . S_k = a + a^2 + a^3 + ... + a^k + a^{k+1}$

$S_k - a . S_k = 1 - a^{k+1}$
$(1 - a)S_k = 1 - a^{k+1} / 1 - a = a^{k+1} - 1 / a - 1 = S_k$

3 Recursie

Oefeningen

3.1 Berekenen van faculteiten

```
0! = 1
n! = n x (n - 1)!       als n ≥ 1
```

Voorbeeld:

```
4! = 4 x 3!
   = 4 x (3 x 2!)
   = 4 x (3 x (2 x 1!))
   = 4 x (3 x (2 x (1 x 0!)))
   = 4 x (3 x (2 x (1 x 1)))
   = 4 x (3 x (2 x 1))
   = 4 x (3 x 2)
   = 4 x 6
   = 24
```

3.1.1 Algoritme

```
berekenFaculteit(I: n: geheel getal): fac: geheel getal
    * preconditie: n is een natuurlijk getal
    * postcondotie: n! werd berekend
    * gebruikt: berekenFaculteit
BEGIN
    ALS n = 0 DAN
        faculteit <- 1
    ANDERS
        faculteit <- n . berekenFaculteit(n - 1)
    EINDE ALS
    RETOUR (faculteit)
EINDE
```

3.1.2 Complexiteitsanalyse

> Hoelang duurt dit?

$T(0) = \Theta(1)$
$T(n) = T(n - 1) + \Theta(1)$ als $n \geq 1$

Na vereenvoudiging:
T(0) = 1
T(n) = T(n - 1) + 1 als n ≥ 1
Uitwerking:
T(0) = 1
T(1) = T(0) + 1 = 1 + 1 = 2
T(2) = T(1) + 1 = 2 + 1 = 3
T(3) = T(2) + 1 = 3 + 1 = 4
T(4) = T(3) + 1 = 4 + 1 = 5
Gok:
T(n) = n + 1

Bewijs (Door inductie):

> Stel je hebt een oneindige rij van personen P_0, P_1, P_2, P_3, P_4, P_5, ...

1. De eerste persoon in de rij weet een geheim
2. Als een persoon een geheim weet dan vertelt die het door aan de volgende persoon in de rij.

Wie weet het geheim? Iedereen want het wordt doorgegeven.

```
Gegeven:        T(0) = 1
                T(n) = T(n - 1) + 1             als n ≥ 1
```

Te bewijzen: T(n) = n + 1
Bewijs:

> Kan op een examen komen!

```
1. Basisstap: verifieer dat het te bewijzen waar is voor n = 0
    Linker Lid:     T(0)                        = 1 (gegeven)
    Rechter Lid:    n + 1 = 0 + 1               = 1

2. Inductiestap:
    Veronderstel dat T(m) = m + 1               als m ≤ n
                       (Inductiehypothese)
    T(n + 1) = T(n) + 1                         (gegeven)
             = (n + 1) + 1
             = n + 2                            QED
```

> T(n) = Θ(n)

3.2 De torens van Hanoi

n	# Bewegingen
1	1
2	3
3	7
4	15
5	31

3.1.1 Oplossingsmethode

Recursiebetrekking:

```
T(1) = 1
T(n) = 2 T(n - 1) + 1        als n ≥ 2
```

Gok: (n) = 2^n-1

```
Gegeven:         T(1) = 1
                 T(n) = 2T(n - 1) + 1            als n ≥ 2
```

Te bewijzen: T(n) = 2^n-1 als ≥ 1
Bewijs:

1. Basisstap: verifieer dat het te bewijzen waar is voor n = 1
   Linker Lid: T(1) = 1 (gegeven)
   Rechter Lid: 2^n-1 = 2^1 - 1 = 2 - 1 = 1
2. Inductiestap: Veronderstel dat T(m) = 2^m - 1 als m ≤ n (Inductiehypothese)
   T(n + 1) = 2T(n) + 1 (gegeven)
   (inductiefase)
   = 2 . (2^n - 1) + 1
   = 2^{n+1} - 2 + 1
   = 2^{n+1} - 1

3.2.2 Algoritme

```
verplaatsToren(I: n, staaf1, staaf2, staaf3: geheel getal): /
```

```
    * Preconditie: het aantal schijven n (n e N0) en de drie staven staaf1,
staaf2, staaf3 zijn geïnitialiseerd
    * Postconditie: de n schijven werden verplaatsts van staaf1 naar staaf 3
met behulp van staaf2 voor tijdelijke opslag
    * Gebruikt: verplaatsToren
BEGIN
    ALS (n = 1) DAN
        VOERUIT(scherm, "Verplaats schijf van", staaf1, "naar", staaf3)
    ANDERS
        verplaatsToren(n - 1, staaf1, staaf3, staaf2)  // 1 3 2
        verplaatsToren(1, staaf1, staaf2, staaf3)      // 1 2 3
        verplaatsToren(n - 1, staaf2, staaf1, staaf3)  // 2 1 3
    EINDE ALS
EINDE
```

3.2.3 Complexiteitsanalyse

$T(n) = \Theta(2^n)$
\# Zetten = $2^{64}-1 = 1{,}84467441 \times 10^{19}$
1 Schijf per dag
\# jaar = $5{,}05 \times 10^{16}$ jaar
leeftijd aarde = $4{,}5 \times 10^{19}$ jaar
1 schijf per seconde
\# jaar = $5{,}85 \times 10^{11}$ jaar
leeftijd universum = $13{,}8 \times 10^{9}$ jaar

3.3 De rij van Fibonacci

$F_0 = 1$
$F_1 = 1$
$F_n = F_{n-1} + F_{n-2}$ als $n \geq 2$

3.3.1 Berekenen van Fibonacci-getallen met recursie

> Het volgende algoritme werkt maar is zeer traag.

```
berekenFibRec(I: n: geheel getal): getal: geheel getal
    * Preconditie: n is een natuurlijk getal.
    * Postconditie: het n-de Fibonacci-getal werd geretourneerd.
    * Gebruikt: berekenFibRec
BEGIN
    ALS (n = 0 of n = 1) DAN
        getal <- 1
    ANDERS
```

```
        getal <- berekenFibRec(n - 1) + berekenFibRec(n - 2)
    EINDE ALS
    RETOUR (getal)
EINDE
```

$T(n) = T(n - 1) + T(n - 2) + \Theta(1)$
$T(n) \geq (3/2)^{n-2}$ voor $n \geq 1$

3.3.1 Berekenen van Fibonacci-getallen met iteratie

> Het volgende algoritme is veel sneller.

```
berekenFibIter(I: n: geheel getal): getal: geheel getal
    * Preconditie: n is een natuurlijk getal.
    * Postconditie: het n-de Fibonacci-getal werd geretourneerd.
    * Gebruikt: /
BEGIN
    voorvorig <- 1
    vorig <- 1
    getal <- 1
    VOOR i = 2 TOT n
        getal <- voorvorig + vorig
        voorvorig <- vorig
        vorig <- getal
    EINDE VOOR
    RETOUR (getal)
EINDE
```

$T(n) = \Theta(n)$

4 Zoek- en sorteeralgoritmen

4.1 Zoekalgoritmen

4.1.1 Sequentieel of lineair zoeken in een array

```
zoekSequentieel(I: zoekGetal: geheel getal, rij: array[] van gehele
getallen): index: geheel getal
    * Preconditie: rij is een array van lengte n van gehele getallen;
zoekGetal is het te zoeken element in de array.
    * Postconditie: index geeft de waarde -1 als zoekGetal niet voorkomt in
rij en de waarde van de index van zoekGetal in rij als zoekGetal wel
voorkomt in de rij.
    * Gebruikt: /
BEGIN
    i <- 0
    // Volgorde is cruciaal (vals en iets anders is altijd vals bij een AND)
=> Short Circuit Evaluation
    ZOLANG (i < n) EN (rij[i] ≠ zoekGetal) DOE
        i <- i + 1
    EINDE ZOLANG

    ALS (i = n) DAN
        index <- -1
    ANDERS
        index <- i
    EINDE ALS

    RETOUR (index)
EINDE
```

Oefening a)

```
rij = [1, 2, 3, 4, 6]
zoekGetal = 6
```

i	rij[i]	iteratievoorwaarde
0	1	Waar
1	2	Waar
2	3	Waar
3	4	Waar

4	6	Vals

```
? i = n
<=> 4 = 5 -> Vals
    index <- i
    index <- 4
```

> Θ(n)

Oefening b)

```
rij = [6, 4, 3, 2, 1]
zoekGetal = 6
```

i	rij[i]	iteratievoorwaarde
0	6	Vals

```
? i = n
<=> 0 = 5 -> Vals
    index <- i
    index <- 0
```

> Als je getal voorraan staat, maakt het niet uit hoe lang de rij is, je uitvoeringsstij is constant

Oefening C)

```
rij = [1, 3, 6, 4, 2]
zoekGetal = 6
```

i	rij[i]	iteratievoorwaarde
0	1	Waar
1	3	Waar
2	6	Vals

```
? i = n
<=> 2 = 5 -> Vals
    index <- i
    index <- 2
```

Oefening D)

```
rij = [0, 2, 4, 6, 8]
zoekGetal = 5
```

i	rij[i]	iteratievoorwaarde
0	0	Waar
1	2	Waar
2	4	Waar
3	6	Waar
4	8	Waar
5		Vals

```
? i = n
<=> 5 = 5 -> Waar
    index <- -1
```

4.1.1.1 Gesorteerde rij

```
zoekSequentieelGesorteerd(I: zoekGetal: geheel getal, rij: array[] van
gehele getallen): index: geheel getal
    * Preconditie: rij is een gesorteere array van lengte n van gehele
getallen; zoekGetal is het te zoeken element in de array.
    * Postconditie: index geeft de waarde -1 als zoekGetal niet voorkomt in
rij en de waarde van de index van zoekGetal in rij als zoekGetal wel
voorkomt in de rij.
    * Gebruikt: /
BEGIN
    i <- 0
    ZOLANG (i < n) EN (rij[i] < zoekGetal) DOE
        i <- i + 1
    EINDE ZOLANG
```

```
    ALS (i = n) OF (rij[i] > zoekGetal) DAN
        index <- -1
    ANDERS
        index <- i
    EINDE ALS

    RETOUR (index)
EINDE
```

Oefening a)

```
rij = [1, 3, 5, 7, 9]
zoekGetal = 1
```

i	rij[i]	iteratievoorwaarde
0	1	Vals

```
? (i = n) OF (rij[i] > zoekGetal)
<=> (0 = 5) OF (1 > 1 ) -> Vals
    index <- i
    index <- 0
```

Oefening b)

```
rij = [1, 3, 5, 7, 9]
zoekGetal = 6
```

i	rij[i]	iteratievoorwaarde
0	1	Waar
1	3	Waar
2	5	Waar
3	7	Vals

```
? (i = n) OF (rij[i] > zoekGetal)
<=> (3 = 5) OF (7 > 6 ) -> Waar
    index <- -1
```

> In het beste geval: T(n) = Θ (1).
>
> In het slechtste geval: T(n) = Θ(n).
>
> In het gemiddeld geval: T(n) = Θ(n).

4.1.2 Binair zoeken in een array

```
zoekBinair(I: zoekGetal: geheel getal, rij: array[] van gehele getallen): 
index: geheel getal
    * Preconditie: rij is een gesorteere array van lengte n van gehele 
getallen; zoekGetal is het te zoeken element in de array.
    * Postconditie: index geeft de waarde -1 als zoekGetal niet voorkomt in 
rij en de waarde van de index van zoekGetal in rij als zoekGetal wel 
voorkomt in de rij.
    * Gebruikt: /
BEGIN
    l <- 0
    r <- n - 1

    ZOLANG (l ≠ r) DOE
        m <- floor((l + r) / 2)
        ALS rij[m] < zoekGetal  DAN
            l <- m + 1
        ANDERS
            r <- m
        EINDE ALS
    EINDE ZOLANG

    ALS rij[l] = zoekGetal DAN
        index <- l
    ANDERS
        index <- -1
    EINDE ALS
    RETOUR (index)
EINDE
```

> | *getal* | = afronden naar beneden = floor

```
l = 2, r = 6
m = floor((2 + 6) / 2)) = 4
```

```
l = 2, r = 3
m = floor(((2 + 3) / 2)) = 2
```

> l ≤ m < r

Oefening a)

```
rij = [1, 2, 3, 4, 6, 7, 8, 9, 10]
n = 9
zoekGetal = 3
```

l	r	m	rij[m]
0			
	8		
		4	6
	4		
		2	3
	2		
		1	2
2			

```
? rij[l] = zoekGetal
<=> 3 = 3 -> Waar
    index <- l
    index <- 2
```

Oefening b)

```
rij = [1, 2, 3, 4, 6, 7, 8, 9, 10]
n = 9
zoekGetal = 5
```

l	r	m	rij[m]
0			
	8		

		4	6
	4		
		2	3
3			
		3	4
4			

> l = r

```
? rij[l] = zoekGetal
<=> rij[4] = 5
    6 = 5 -> Vals
    index <- -1
```

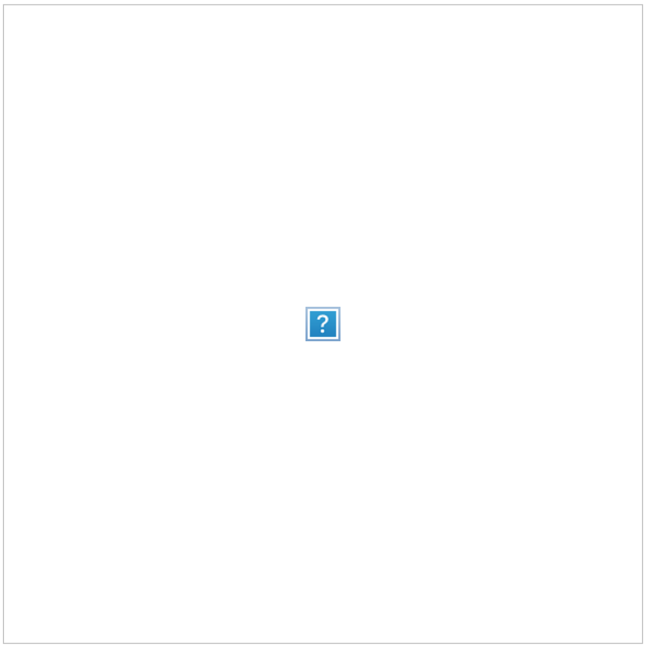
> **^ Op Examen!**

```
zoekRecursief(I: zoekGetal: geheel getal, rij: array[] van gehele getallen):
index: geheel getal
    * Preconditie: rij is een gesorteere array van lengte n van gehele
getallen; zoekGetal is het te zoeken element in de array.
    * Postconditie: index geeft de waarde -1 als zoekGetal niet voorkomt in
rij en de waarde van de index van zoekGetal in rij als zoekGetal wel
voorkomt in de rij.
    * Gebruikt: zoek
BEGIN
    index <- zoek(zoekGetal, rij, 0, n - 1)
    RETOUR (index)
EINDE

zoek(I: zoekGetal: geheel getal, rij: array[] van gehele getallen, l, r:
geheel getal): index: geheel getal
    * Preconditie: rij is een gesorteerde array van lengte n van gehele
getallen; zoekGetal is het te zoeken element in de array; l en r geven
respectievelijk de posities weer waartussen zoekGetal wordt gezocht.
    * Postconditie: index geeft de waarde -1 als zoekGetal niet voorkomt in
rij tussen l en r en de waarde van de index van zoekGetal in rij als
zoekGetal wel voorkomt in de rij tussen l en r
    * Gebruikt: zoek
BEGIN
    // Basiscase
```

```
    ALS (l = r) DAN
        ALS (rij[l] = zoekGetal) DAN
            index <- l
        ANDERS
            index <- -1
        EINDE ALS

    // Anders
    ANDERS
        m <- floor(((l + r) / 2))
        ALS (rij[m] < zoekGetal) DAN
            index <- zoek(zoekGetal, rij, m + 1, r)
        ANDERS
            index <- zoek(zoekGetal, rij, l, m)
        EINDE ALS
    EINDE ALS
    RETOUR (index)
EINDE
```

Oefening a)

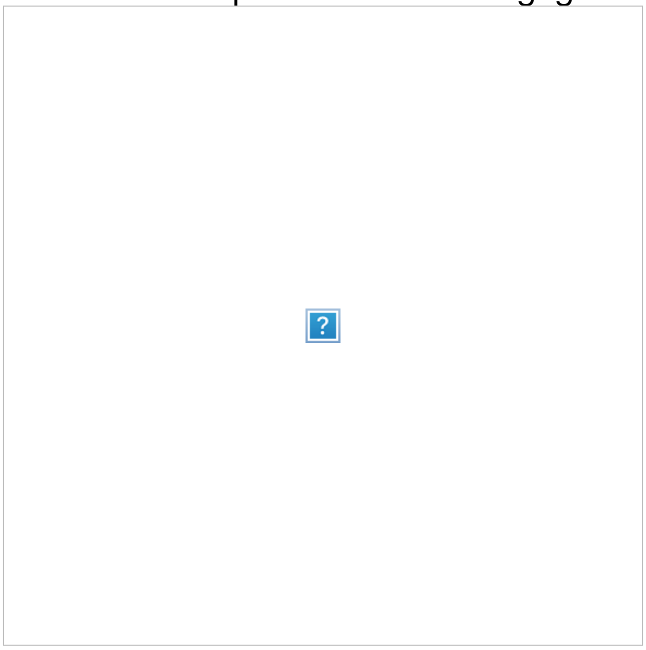
```
rij = [1, 2, 3, 4, 6, 7, 8, 9, 10]
n = 9
zoekGetal = 5
```



4.2 Soorteeralgoritmen

4.2.1 Sorteren door selectie

In de te sorteren array a gaan we op zoek naar het grootste element. Indien dit element niet achteraan staat in de rij, moet dit element verwisseld worden met het element op de laatste plaats. Het grootste element staat nu achteraan in de rij; dat is de juiste plaats voor dit element. De (n – 1) overige elementen van de array moeten nog gesorteerd worden.
Voor de deelrij a[0], ..., a[n – 2] gaan we op analoge manier tewerk. Het grootste element in de rij wordt bepaald en achteraan geplaatst, dus op de (n – 2)-de positie.
Deze werkwijze wordt herhaald op steeds kortere deelrijen. De laatste keer zal de deelrij nog bestaan uit twee elementen. De implementatie wordt gegeven in Algoritme 4.33.



```
selectionSort(I: a: array[] van getallen): a: array[] van getallen
    * Preconditie: de array a is gevuld met n elementen
    * Postconditie: de array a is gesorteerd.
    * Gebruikt: /
BEGIN
    VOOR i = n - 1 TOT 1 (STAP - 1) DOE             // Achteraan starten
        positie <- i
        max <- a[i]
        VOOR j = i - 1 TOT 0 (STAP - 1) DOE         // j doorloopt de
deelrij
            ALS(a[j] > max) DAN
                positie <- j
                max <- a[j]
            EINDE ALS
        EINDE VOOR
        a[positie] <- a[i]                          // het grooste element
verwisselen met het laatste
        a[i] <- max
    EINDE VOOR
    RETOUR (a)
EINDE
```

4.2.1.1 Complexiteitsanalyse

- **T(n) = Θ(n²)**

4.2.2 Sorteren door tussenvoegen (Insertion sort of Card sort)

Sorteren door tussenvoegen of card sort kan het best vergeleken worden met het op volgorde steken van kaarten. We beginnen met de tweede kaart. We kijken of deze voor de eerste moet komen of niet. Vervolgens nemen we de volgende kaart en deze plaatsen we dan direct op de juiste positie ten opzichte van de vorige kaarten. Zo doen we verder tot alle kaarten op de juiste plaats zitten.
In het algoritme is dit: indien de eerste k elementen reeds gesorteerd zijn dan gaan we kijken naar het (k + 1)-ste element. Dit element wordt op de juiste plaats tussenge- voegd. Indien nodig moeten de reeds gesorteerde grotere elementen allen één positie doorschuiven.

```
cardSort(I: a: array[] van getallen): a: array[] van getallen
    * Preconditie: de array a is gevuld met n elementen.
    * Postconditie: de array a is gesorteerd.
    * Gebruikt: /
BEGIN
    VOOR i = 1 TOT n - 1 DOE
        x <- a[i]                               // x bevat het in te voegen
element
        j <- i                                  // j zoekt de juiste positie
voor x
        ZOLANG j > 0 EN x < a[j - 1] DOE        // de grotere elementen
doorschuiven
            a[j] <- a[j - 1]                    // schuif a[j - 1] eentje op
            j <- j - 1
        EINDE ZOLANG
        a[j] <- x                               // x wordt op de juiste
positie tussengevoegd
    EINDE VOOR
    RETOUR (a)
EINDE
```

4.2.2.1 Complexiteitsanalyse

- **T(n) = Θ(n²)** het slechtste geval
- **T(n) = Θ(n)** het beste geval
- **T(n) = Θ(n²)** het gemiddelde geval

4.2.3 Mergesort

> Heeft dubbel zoveel geheugen nodig omdat hij een hulp array aanmaakt dat even groot is.

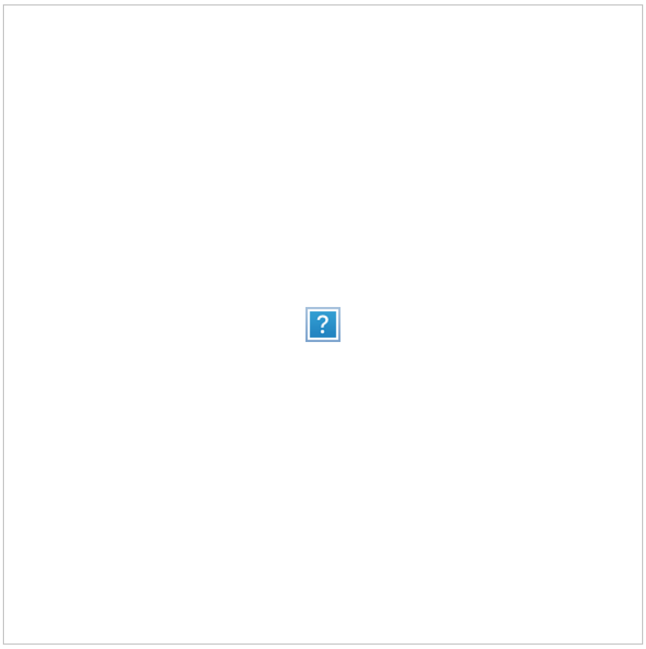
```
rij = [44, 55, 12, 42, 94, 18, 6, 67]
```

Rij in 2 splitsen

```
// Opsplitsen in 2 delen
rij = [44, 55, 12, 42 | 94, 18, 6, 67]

// Sorteren (beide rijen recursief sorteren)
rij = [12, 42, 44, 55 | 6, 18, 67, 94]

// Volledige rij gesorteerd (samengevoegd)
rij = [6, 12, 18, 42, 44, 55, 67, 94]
```

```
mergeSort (I : a: array[] van getallen) : a: array[] van getallen
    * Preconditie: de array a is gevuld met n elementen.
    * Postconditie: de array a werd gesorteerd.
    * Gebruikt: mergeSorteer.
BEGIN
    a <- mergeSorteer(a, 0, n - 1)
    RETOUR (a)
EINDE

mergeSorteer (I : a: array van getallen; begin, einde: geheel getal) : a:
array van
getallen
    * Preconditie: de array a is gevuld met n elementen.
    * Postconditie: de elementen met index begin tot en met index einde
werden gesorteerd.
    * Gebruikt: mergeSorteer, merge.
BEGIN
    ALS (begin < einde) DAN
        midden <- floor((begin + einde)/2)      // begin ≤ m < einde
        a <- mergeSorteer(a, begin, midden)     // Eerste helft sorteren
        a <- mergeSorteer(a, midden + 1, einde) // Tweede helft sorteren
        a <- merge(a, begin, midden, einde)     // Helften samenvoegen
    EINDE ALS
    RETOUR (a)
EINDE

merge(I: a: array[] van getallen; begin, midden, einde: geheel getal): a:
array[] van getallen
    * Preconditie: de array a is gevuld met n elementen; de elementen van de
deelrij gaande van de begin-positie tot en met de midden positie zijn
gesorteerd; de elementen van de deelrij gaande van de (midden+1)-positie tot
en met de eind-positie zijn gesorteerd.
    * Postconditie: de elementen met index begin tot en met index einde
werden gesorteerd
    * Gebruikt: /
```

```
BEGIN
    i <- begin              // De teller i doorloopt de linkse deelrij
    j <- midden + 1         // De teller j doorloopt de rechtse deelrij
    hulp <- nieuwe array[n] // De hulp array
    k <- i                  // De teller k doorloopt de hulparray hulp

    ZOLANG ((i ≤ midden) EN (j ≤ einde)) DOE
        ALS a[i] ≤ a[j] DAN
            hulp[k] <- a[i]
            i <- i + 1
        ANDERS
            hulp[k] <- a[j]
            j <- j + 1
        EINDE ALS
        k <- k + 1
    EINDE ZOLANG

    ZOLANG (i ≤ midden) DOE
        hulp[k] <- a[i]
        i <- i + 1
        k <- k + 1
    EINDE ZOLANG

    ZOLANG (j ≤ einde) DOE
        hulp[k] <- a[j]
        j <- j + 1
        k <- k + 1
    EINDE ZOLANG

    VOOR k = begin TOT einde DOE
        a[k] <- hulp[k]
    EINDE VOOR

    RETOUR (a)
EINDE
```



> Mege T(n) = Θ(n)
>
> T(1) = 1
>
> T(n) = T(n / 2) = 2T(n/2) + n // als n > 1
>
> T(n) = Θ(n x lg(n))
>
> VB.: 1000 = (1000 x 10) = 10.000

```
// Grootste element achteraan plaatsen
VOOR i = n - 1 TOT 1 (STAP - 1) DOE
    positie <- i
    max <- a[i]

    VOOR j = i - 1 TOT 0 (STAP - 1) DOE
        ALS (a[j] > max) DAN
            positie <- j
            max <- a[j]
        EINDE ALS
    EINDE VOOR
    a[positie] <- a[i]
    a[i] <- max
EINDE VOOR

// Kleinste element vooraan plaatsen
VOOR i = 0 TOT n - 1 DOE
    positie <- i
    min <- a[i]

    VOOR j = i TOT n - 1 DOE
        ALS (a[j] < min) DAN
```

```
            positie <- j
            min <- a[j]
        EINDE ALS
    EINDE VOOR
    a[positie] <- a[i]
    a[i] <- min
EINDE VOOR
```

4.2.4 Quicksort

```
rij = [12, 42, 67, 55, 06, 18, 44, 84]
spil = 55 // Random element van de rij
```

> Alle elementen kleiner dan de spil, voor de spil zetten. Alle elementen groter dan de spil, achter de spil zetten. Sorteer dan recursief de rij voor de spil en de rij na de spil

```
rij = [12, 42, 6, 18, 44, 55, 67, 94] // De spil staat nu op de correcte plaats
// recursief hetzelfde doen met de rij voor de spil en de rij na de spil

// Gesorteerde rij:
rij = [6, 12, 18, 42, 44, 55, 67, 94]
```

Tijdcomplexiteit in het slechste geval

> T(1) = 1
>
> T(n) = T(n - 1) + n // als n > 1 als grootste element als spil gebruikt wordt
>
> T(n) = n(n^2)

Tijdcomplexiteit in het beste geval

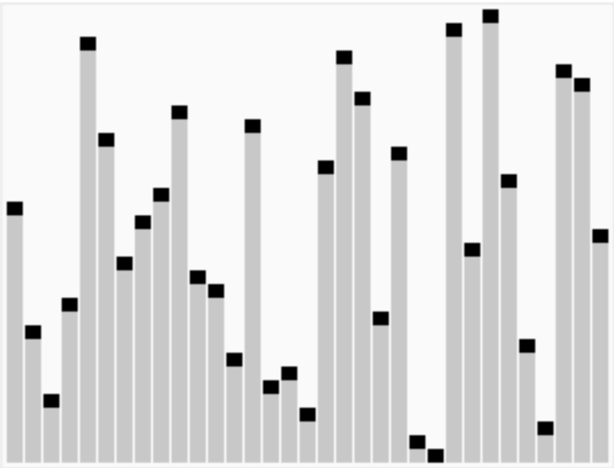
> T(1) = 1
>
> T(n) = 2 T(n / 2) + n // als n > 1 als element de mediaan is
>
> T(n) = Θ(n x lg(n))

Mediaan-van-drie

Het eerste, middelste en laatste element kiezen als mogelijke spillen.

Partionering

```
rij = [57, 12, 42, 67, 55, 6, 18, 44, 94]
spil = 55
// Verwissel spil met de laatste

rij = [57, 12, 42, 67, 94, 6, 18, 44, 55]

// Grote elementen moeten rechts, kleine elementen moeten links.
// Probleem 57 is groot en 44 is klein: omwisselen

rij = [44, 12, 42, 67, 94, 6, 18, 57, 55]

// 44 staat nu goed
// 12 staat nu goed
// 42 staat nu goed

// 67 staat verkeerd; we kijken naar de rechterkant
// 57 staat nu goed
// 18 staat verkeerd; we verwisselen 18 met 67

rij = [44, 12, 42, 18, 94, 6, 67, 57, 55]

// 18 staat nu goed
// 94 staat verkeerd; we kijken naar de rechterkant
// 67 staat nu goed
// 6 staat verkeerd; we verwisselen 6 met 94

rij = [44, 12, 42, 18, 6, 94, 67, 57, 55]

// Links (94) en rechts (6) zijn gewisseld; Links duid nu de eerste grote
aan.

// We verwisselen nu links met het laatste element

rij = [44, 12, 42, 18, 6, 55, 67, 57, 94]
```



Mediaan-van-drie Partionering

```
rij = [57 12 42 67 55 06 18 44 94]
mogelijkeSpillen = [57, 55, 94] // eerste, middelste, laatste

// De 3 spillen ordenen
rij = [55, 12, 42, 67, 57, 6, 18, 44, 94]

// We gaan wisselen met de voorlaatste omdat de laatste een mogelijke spil
was
rij = [55, 12, 42, 67, 44, 6, 18, 57, 94]

// We kijken pas vanaf de eerste omdat de eerste ook een mogelijke spil was
rij = [55, 12, 42, 67, 44, 6, 18, 57, 94]
links = 67
rechts = 6

// Links en rechts zijn al gewisseld
rij = [55, 12, 42, 18, 44, 6, 57, 67, 94]
```

> Quicksort is zeer **snel** als de rijen lang zijn; Voor kleine rijen zeer **traag**.

```
quickSort(I: a: array[] van getallen) : a: array[] van getallen
    * Preconditie: de array a is gevuld met n elementen.
    * Postconditie: de array a werd gesorteerd.
    * Gebruikt: quickSorteer
BEGIN
    a <- quickSorteer(a, 0, n - 1)
    RETOUR (a)
EINDE
```

```
quickSorteer(I: a: array[] van getallen; begin, midden, einde: geheel
getal): a: array[] van getallen
    * Preconditie: de array a is gevuld met n elementen
    * Postconditie: in de array a werden alle elementen tussen de positie
begin en de positie einde gesorteerd
    * Gebruikt: quickSorteer, cardSortBis
BEGIN
    cutoff <- 5
    ALS (begin + cutoff > einde) DAN
        a <- cardSortBis(a, begin, einde)
    ANDERS
        // Spil Bepalen
        midden <- floor((begin + einde) / 2)

        ALS (a[midden] < a[begin]) DAN
            verwissel a[midden] en a[begin]
```

```
        EINDE ALS

        ALS (a[einde] < a[begin]) DAN
            verwissel a[einde] en a[begin]
        EINDE ALS

        ALS (a[einde] < a[midden]) DAN
            verwissel a[einde] en a[midden]
        EINDE ALS
        spil <- a[midden]

        // Partitioneren
        verwissel spil met a[einde - 1]

        links <- begin + 1
        rechts <- einde - 2

        ZOLANG (links ≤ rechts) DOE
            ZOLANG (a[links] < spil) DOE
                links <- links + 1
            EINDE ZOLANG

            ZOLANG (a[rehts] > spil) DOE
                rechts <- rechts - 1
            EINDE ZOLANG

            ALS (links ≤ rechts) DAN
                verwissel a[links] en a[rechts]
                links <- links + 1
                rechts <- rechts - 1
            EINDE ALS
        EINDE ZOLANG
        verwissel a[links] met spil
        spilindex <- links
        a <- quickSorteer(a, begin, spilindex - 1)
        a <- quickSorteer(a, spilindex + 1, einde)
    EINDE ALS
EINDE
```

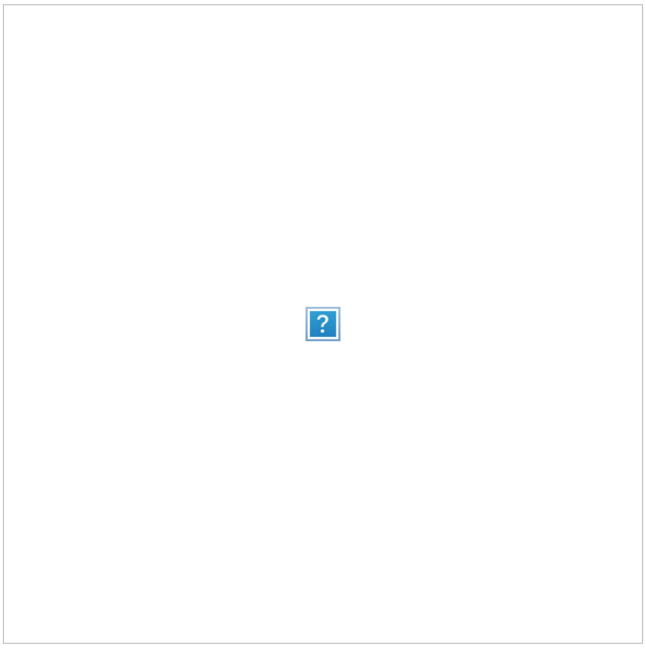
> **Examen**: van cardSort naar CardSortBis: welke lijnen moet je aanpassen en wat moet je aanpassen:

```
cardSortBis (I : a: array[] van getallen; begin, einde: geheel getal) : a:
array[] van getallen
    * Preconditie: de array a is gevuld met n elementen.
    * Postconditie: in de array a werden de elementen met index begin tot en
met index einde gesorteerd.
    * Gebruikt: /
BEGIN
    VOOR i = begin + 1 TOT einde DOE
```

```
            x <- a[i]
            j <- i
            ZOLANG j > begin EN x < a[j - 1] DOE
                a[j] <- a[j - 1]
                j <- j - 1
            EINDE ZOLANG
            a[j] <- x
        EINDE VOOR
        RETOUR (a)
    EINDE
```

Voorbeeld oefening:

14 40 31 28 03 **15** 17 51 77 04 07 **63**
14 40 31 28 03 07 17 51 77 04 **15** 63
14 40 07 28 03 31 17 51 77 40 **15** 63
14 40 07 03 28 31 17 51 77 40 **15** 63
14 40 07 03 28 31 17 51 77 40 **15** 63
14 40 07 03 **15** 31 17 51 77 40 28 63
`spilIndex = 4`
"cardSort": 03 04 07 14 **15** 31 17 51 77 40 28 63 omdat 0+5 > 4 // 0 = Begin; 5 = Cutoff; Einde = 3
`spilIndex = 5`
14 40 07 03 |15| **31** 17 51 **77** 40 28 **63**
14 40 07 03 15 **31** 17 51 **63** 40 28 **77**
14 40 07 03 15 31 17 51 28 40 **63** 77
14 40 07 03 15 31 17 51 28 40 **63** 77
`spilIndex = 10`
"cardSort": 14 40 07 03 15 31 17 28 31 40 51 63 77



5 Stapels (Stacks)

Oefeningen
Stapels:

> Een datastructuur waarbij je enkel bovenaan kan toevoegen en ook enkel het bovenste element bekijken en verwijderen: `LIFO` **(last in first out)**

5.1 Specificatie

- Een element dat we willen toevoegen aan een stapel, komt steeds bovenop de reeds bestaande stapel te liggen.
- Een element verwijderen is enkel mogelijk indien dit het bovenste element is, enkel het bovenste element van de stapel is bereikbaar. Dit element wordt de top van de stapel genoemd.

5.2 Implementatie met behulp van arrays

```
Stack(): constructor
empty(): controleert of een stapel al dan niet leeg is
push(): voegt een nieuw element toe bovenaan een stapel
pop(): verwijdert het bovenste element van een stapel en retourneert het verwijderde element
peek(): geeft het bovenste element van de stapel terug, zonder het te verwijderen
```

5.2.1 Voorbeeld

5.2.2 Algoritmen

Stack
- data: array[] van Element - t : geheel getal
+ Stack(n: geheel getal) + empty(): boolean + push(x: Element): / + pop() : Element + peek(): Element

> `t` is de top index, dus om het laatste element te zien moet je `data[t]` gebruiken.

5.2.2.1 Implementatie van Stack()

```
Stack(I: n: geheel getal): /
    * Preconditie: n is een natuurlijk getal
    * Postconditie: de array data van lengte n werd gealloceerd en t werd
geïnitialiseerd
    * Gebruikt: /
BEGIN
    data <- nieuwe array[n]
    t <- - 1 // Geen element op de stapel
EINDE
```

5.2.2.2 Implementatie van empty()

```
empty(I: /): vlag: Boolean
    * Preconditie: de stapel s bestaat
    * Postconditie: de waarde true of false werd afgeleverd, afhankelijk van
het feit of de stapel s leeg is of niet
    * Gebruikt: /
BEGIN
    RETOUR (t = - 1) // in java zou het ( t == - 1) zijn.
EINDE
```

5.2.2.3 Implementatie van push(x)

```
push(I: x: Element): /
    * Preconditie: de stapel s bestaat en is nog niet vol
    * Postconditie: het element x werd als top-element op de stapel s
geplaatst.
    * Gebriukt: /
BEGIN
    t <- t + 1
    data[t] <- x
EINDE
```

5.2.2.4 Implementatie van pop()

```
pop(I: /): x: Element
    * Preconditie: de stapel s bestaat en is niet leeg
    * Postconditie: de top van de stapel s werd verwijderd en geretourneerd
    * Gebruikt: /
BEGIN
    x <- data[t]
    t <- t - 1
    RETOUR(x)
EINDE
```

5.2.2.5 Implementatie van peek()

```
ppeekop(I: /): x: Element
    * Preconditie: de stapel s bestaat en is niet leeg
    * Postconditie: de waarde van de top werd toegekend aan x en
geretourneerd, de stapel s werd niet gewijzigd
    * Gebruikt: /
BEGIN
    RETOUR data[t]
EINDE
```

> Alle methodes zijn T(n) = Θ(1)

5.3 Voorbeeld: de methode one

> Bevat de stack 1 element of niet

```
one(I: s: Stack): vlag: boolean
    * Preconditie: een stapel s wordt meegegeven
    * Postconditie: indien juist één element tot s behoort, werd true
geretourneerd anders false; de stapel s werd niet gewijzigd.
    * Gebruikt: empty, push, pop
BEGIN
    ALS (s.empty()) DAN
        vlag <- false
    ANDERS
        x <- s.pop()
        vlag <- s.empty()
        s.push(x)
    EINDE ALS
    RETOUR vlag
EINDE
```

5.4 Toepassing 1: controle van haakjes

```
controleerHaakjes(I: uitdrukking: array[] van Strings): /
    * Preconditie: uitdrukking is een uitdrukking waarin eventueel haakjes
voorkomen
    * Postconditie: indien alle open haakjes correct worden afgesloten werd
er geen foutmelding gegenereerd
    * Gebruikt: Stack, empty, push, pop, lengte, getal
BEGIN
    s <- nieuwe Stack(uitdrukking.lengte)
    VOOR i = 0 TOT uitdrukking.lengte - 1 DOE
        symbool <- uitdrukking[i]
        ALS (symbool e {  ),],}  }) DAN
            s.push(symbool)
        ANDERS
            ALS s.empty() DAN
                ALS (symbool e {  ),],}  }) DAN
                    VOERUIT(scherm, "Te veel sluit symbolen")
                ANDERS
                    voorgaand <- s.pop()
                    ALS(symbool != voorgaand) DAN
                        VOERUIT(scherm, "Fout symbool")
                    EINDE ALS
                EINDE ALS
            EINDE ALS
        EINDE ALS
    EINDE VOOR
    ALS (NIET s.empty()) DAN
        VOERUIT(scherm, "Te veel open symbolen")
    EINDE ALS
EINDE
```

5.5 Toepassing 2: het berekenen van postfix-uitdrukkingen

5.5.1 Het bepalen van de waarde van een postfix-uitdrukking

Infix:

> De operator staat tussen de operandum (parameters)

```
3 + 4 = 7
3 + 4 x 5 = 23
(3 + 4) x 5 = 35
```

Postfix:

> De operatoren staan na de operandum (parameters)

```
3 4 +
3 4 5 x +
3 4 + 5 x
```

Uitrekenen:

5.5.1.1 Voorbeeld Rekenmachine 1

`3 4 5 x +`

```
// De stapel
 5
 4    20
 3    3   23
---  --- ---
```

1. Kom je een getal tegen -> op de stapel plaatsen
2. Kom je een teken tegen -> laatste 2 waarden van de stapel uitwerken en terug op de stapel plaatsen
3. Herhalen tot het einde

5.5.1.2 Voorbeeld Rekenmachine 2

`3 4 + 5 x`

```
// De stapel

 4    5
 3    7    35
---  ---  ---
```

5.5.2 Van infix naar postfix

Eigenschappen:

1. Volgorde van getallen is altijd gelijk
2. Volgorde van getallen is niet altijd gelijk, maar kan wel
3. Er zijn nooit haakjes

Uitwerking:

1. Lees de invoertekst van links naar rechts
2. Een operand wordt naar de uitvoertekst geschreven
3. Een operator of een haakje wordt op de stapel bewaard als:
    - de stapel leeg is
    - de gelezen operator een hogere prioriteit heeft dan de operator die bovenaan de stapel ligt
    - het gelezen haakje een openingshaakje is

    Als de gelezen operator gelijke of lagere prioriteit heeft dan de operator aan de top van de stapel, dan worden alle operatoren van de stapel met gelijke of hogere prioriteit van de stapel gehaald en worden deze toegevoegd aan de uitvoertekst. Dit totdat een operator met lagere prioriteit bereikt wordt of totdat de stapel leeg is. Vervolgens wordt de ingelezen operator op de stapel geplaatst.
    Als een sluitingshaakje wordt ingelezen dan worden alle operatoren van de stapel gehaald en toegevoegd aan de uitvoertekst totdat een openhaakje wordt bereikt. Het haakje wordt eveneens van de stapel gehaald maar niet aan de uitvoertekst toegevoegd.
4. Als het einde van de invoertekst bereikt is, worden alle operatoren van de stapel gehaald en aan de uitvoertekst toegevoegd todat de stapel leeg is

Voorbeeld 1:
`(3 + 4) x 5`

```
// De Stapel

 +
 (  (     x
-- -- -- -- --



// De uitvoertekst
3 4 + 5 x
```

Voorbeeld 2:
`a + (b + c) x d x (e + f x g)`

```
// De Stapel

            x
            +   +
 +          (   (   (
 (   (   x  x   x   x   x
 +   +   +  +   +   +   +   +
--- --- --- --- --- --- --- --- ---



// De uitvoertekst
a b c + d x e f g x + x +
```

6 Wachtrijen (Queues)

Oefeningen

6.1 Specificatie

> Wachtrijen (Queues), dit is een **FIFO structuur (First In First Out)**

- `Queue()` constructor
- `empty()` controleert of een wachtrij al dan niet leeg is
- `enqueue()` voegt een gegeven element toe aan de start van een wachtrij
- `dequeue()` verwijdert het element aan de kop in een wachtrij en retourneert het verijwderde element
- `front()` retourneert het voorste element, m.a.w. de kop, van een wachtrij zonder het te verwijderen

6.2 Implementatie met behulp van arrays

6.2.1 Voorbeelden

6.2.1.1 Voorbeeld 1

```
|   |   |   |   |   |     // k = s = -1
| a |   |   |   |   |     // k = s = 0
| a | b |   |   |   |     // k = 0   s = 1
| a | b | c |   |   |     // k = 0   s = 2
|   | b | c |   |   |     // k = 1   s = 2
|   |   | c |   |   |     // k = 2   s = 2
|   |   | c | d |   |     // k = 2   s = 3
|   |   | c | d | e |     // k = 2   s = 4
| f |   | c | d | e |     // k = 2   s = 0
```

> Een wachtrij is een **circulaire** array, je kan dus alle plaatsen gebruiken.

6.2.2 Algoritmen

Queue
- data: array[] van Element - k : geheel getal - s : geheel getal

+ Queue(n: geheel getal)
+ empty(): boolean
+ enqueue(x: Element): /
+ dequeue() : Element
+ front(): Element

6.2.2.1 Algoritme voor de constructor

```
Queue(I: n: geheel getal): /
    * Preconditie: n is een natuurlijk getal
    * Postconditie: de array data van lengte n werd gealloceerd en t werd
geïnitialiseerd
    * Gebruikt: /
BEGIN
    data <- nieuwe array[n]
    k <- - 1 // Geen element op de stapel
    s <- - 1
EINDE
```

6.2.2.2 Algoritme ter controle of een wachtrij leeg is

```
empty(I: /): vlag: Boolean
    * Preconditie: de wachtrij q bestaat
    * Postconditie: de waarde true of false werd afgeleverd, afhankelijk van
het feit of de wachtrij q leeg is of niet
    * Gebruikt: /
BEGIN
    RETOUR (k = -1)
EINDE
```

6.2.2.3 Algoritme voor het toevoegen van een element aan een wachtrij

```
enqueue(I: x: Element): /
    * Preconditie: de wachtrij q bestaat en is nog niet vol
    * Postconditie: het element x werd aan de staart van de wachtrij q
toegevoegd
    * Gebruikt: empty, length
BEGIN
    ALS empty() DAN
        k <- 0
    EINDE ALS
```

```
    n <- data.length
    s <- (s + 1) MOD n

    data[s] <- x
EINDE
```

6.2.2.4 Algoritme voor het verwijderen van een element van een wachtrij

```
dequeue(I: /): x: Element
    * Preconditie: de wachtrij q bestaat en is neit leeg
    * Postconditie: de kop van de wachtrij q werd verwijderd en
geretourneerd
    * Gebruikt: length
BEGIN
    x <- data[k]

    ALS (k = s) DAN
        k <- -1
        s <- -1
    ANDERS
        n <- data.length
        k <- (k + 1) MOD n
    EINDE ALS

    RETOUR (x)
EINDE
```

6.2.2.5 Algoritme vooor het weergeven van de kop van een wachtrij

```
front(I: /): x: Element
    * Preconditie: de wachtrij q bestaat en is neit leeg
    * Postconditie: de kop van de wachtrij q werd geretourneerd, de wachtrij
q werd neit gewijzigd.
    * Gebruikt: /
BEGIN
    RETOUR data[k]
EINDE
```

6.2.3 Complexiteit

> Alle methodes zijn $T(n) = \Theta(1)$

6.3 Voorbeeld: methode one

```
one(I: q: Queue): vlag: boolean
    * Preconditie: een wachtrij q wordt meegegeven; q bevat maximaal 50
elementen
    * Postconditie: indien q juist één element heeft dan werd true
weergegeven anders false, de wachtrij q werd neit gewijzigd.
    * Gebruikt: Queue, empty, dequeue, enqueue
BEGIN
    ALS (q.empty()) DAN
        vlag <- false
    ANDERS
        x <- q.dequeue()
        vlag <- q.empty()

        qHulp <- nieuwe Queue(50)

        ZOLANG NIET q.empty() DOE
            qHulp.enqueue(q.dequeue())
        EINDE ZOLANG

        q.enqueue(x)

        ZOLANG NIET qHulp.empty() DOE
            q.enqueue(qHulp.dequeue())
        EINDE ZOLANG
    EINDE ALS

    RETOUR vlag
EINDE
```

7 Lijsten

Oefeningen

7.1 Specificatie

> In lijsten kan je data opvragen en verwijderen of vervangen in willekeurige plaatsen, of waar je maar wilt.

- `List()` Constructor
- `empty()` controleert of een lisjt al dan niet leeg is.
- `size()` geeft het aantal elementen van een lisjt terug
- `geefElem()` retourneert het element dat zich bevindt op de positie bepaald door het argument
- `geefPositie()` retourneert de positie van het eerste voorkomen van het meegeleverde argument
- `verwijderElem()` verwijdert het element dat zich op de positie bepaald door het argumenet bevindt en geeft dit element terug
- `invoegenNa()` het element, dat als tweede argument wordt meegeleverd, wordt ingevoegd in de lijst na de meegegeven positie.
- `invoegenVoor()` het element, dat als tweede argument wordt meegeleverd, wordt ingevoegd in de lijst voor de meegegeven positie.
- `vervang()` het element in de lijst op de positie bepaald door het eerste argument, wordt vervangen door het meegeleverde nieuwe lijstelement

7.2 Implementatie met behulp van arrays

List
- data: array[] van Element - aantal: geheel getal
+ List(n: geheel getal) + empty() : boolean + size() : geheel getal + geefElem(p: geheel getal) : Element + geefPositie(x: Element) : geheel getal + verwijderElem(p: geheel getal) : Element + invoegenNa(p: geheel getal, x: Element): / + invoegenVoor(p: geheel getal, x: Element): / + vervang(p: geheel getal, x: Element): /

7.2.2 Algoritmen

7.2.2.1 Algoritme voor de constructor

```
List(I: n: geheel getal): /
    * Preconditie: n is een natuurlijk getal.
    * Postconditie: de array data van lengte n werd gealloceerd, het
natuurlijk getal
aantal werd geïnitialiseerd.
    * Gebruikt: /
BEGIN
    data <- nieuwe array[n]
    aantal <- 0
EINDE
```

7.2.2.2 Algoritme ter controle of een lijst leeg is

```
empty(I: /): vlag: boolean
    * Preconditie: de lijst l bestaat
    * Postconditie: de waarde true of false werd afgeleverd, afhankelijk van
het feit of de lijst l leeg is of niet
    * Gebruikt: /
BEGIN
    ALS (aantal = 0) DAN
        vlag <- true
    ANDERS
        vlag <- false
    EINDE ALS
    RETOUR (vlag)
EINDE
```

7.2.2.3 Algoritme ter controle van de omvang van de lijst

```
size(I: /): grootte: geheel getal
    * Preconditie: de lijst l bestaat
    * Postconditie: het aantal elementen van de lijst l werd geretourneerd
    * Gebruikt: /
BEGIN
    grootte <- aantal
    RETOUR (grootte)
EINDE
```

7.2.2.4 Algoritme voor het opzoeken van een element

```
geefElem(I: p: geheel getal): x: Element
    * Preconditie: de lijst l bestaat; l bevat minstens p + 1 elementen.
    * Postconditie: het element op de p-de positie in de lijst werd
geretourneerd.
    * Gebruikt: /
BEGIN
    x <- data[p]
    RETOUR (x)
EINDE
```

7.2.2.5 Algoritme voor het opzoeken van een positie

```
geefPositie(I: x: Element): p: geheel getal
    * Preconditie: de lijst l bestaat
    * Postconditie: de positie van x werd geretourneerd
    * Gebruikt: /
BEGIN
    i <- 0
    ZOLANG i < aantal EN data[i] ≠ x DOE
        i <- i + 1
    EINDE ZOLANG

    ALS i = aantal DAN
        p <- -1
    ANDERS
        p <- i
    EINDE ALS

    RETOUR (p)
EINDE
```

7.2.2.6 Algoritme voor het verwijderen van een element

```
verwijderElem(I: p: geheel getal): x: Element)
    * Preconditie: de lijst l bestaat
    * Postconditie: het element op de p-de positie werd verwijderd uit de
lijst l en werd geretourneerd
    * Gebruikt: /
BEGIN
    x <- data[p]

    VOOR i = p TOT aantal - 2 DOE
        data[i] <- data[i + 1]
    EINDE VOOR
```

```
    aantal <- aantal - 1

    RETOUR (x)
EINDE
```

7.2.2.7 Algoritme voor het vervangen van een element

```
vervang(I: p: geheel getal, x: Element): /
    * Preconditie: de lisjt l bestaat.
    * Postconditie: het element op de p-de positie van de lijst l werd
vervangen door x
    * Gebruikt: /
BEGIN
    data[p] <- x
EINDE
```

7.2.3 Tijdscomplexiteit

- In het beste geval **T(n) = Θ(1)**
- In het gemiddelde geval **T(n) = Θ(n)**
- In het slechtste geval **T(n) = Θ(n)**

8 Gelikte lijsten

Oefeningen

8.1 Specificatie

```
class Knoop {
    private Object data;

    private Knoop volgende;
}

class Lijst {
    private Knoop eerste;
}
```

```
// Hoe het echt zou zijn
class Lijst {

    private Knoop eerste;

    class Knoop {
        private Object data;

        private Knoop volgende;
    }
}
```

8.2 Algoritmen

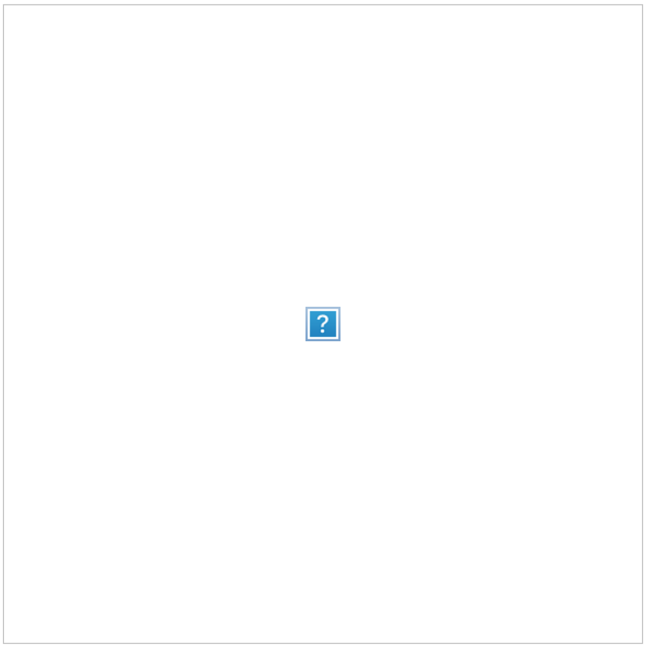
Knoop
- data : Element - volgende : Knoop
+ Knoop()

8.2.1 Een implementatie van een knoop

```
Knoop (I: /): /
    * Preconditie: /
```

```
    * Postconditie: er werd een nieuwe knoop aangemakat.
    * Gebruikt: /
BEGIN
    data <- null
    volgende <- null
EINDE
```

GelinkteLijst
- eerste : Knoop
+ GelinkteLijst() + zoek(x: Element) : Knoop + verwijder(ref : Knoop): Element + voegToe(ref: Knoop, x : Element): /



8.2.2 Implementatie van een gelinkte lijst

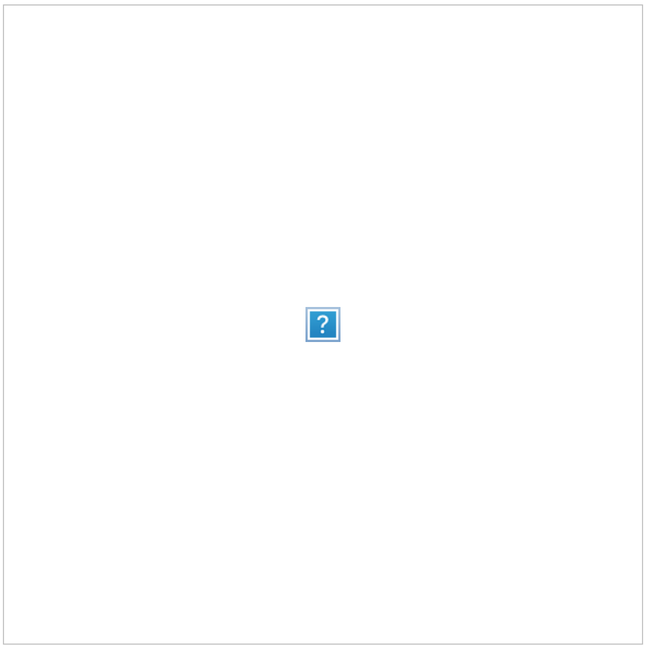
8.2.2.1 Algoritme voor de constructor

```
GelinkteLijst(I: /): /
    * Preconditie: /
    * Postconditie: er werd een nieuwe gelinkte lijst
    * Gebruikt: /
BEGIN
    eerste <- null
EINDE
```

8.2.2.2 Algoritme voor het opzoeken van een element x

```
zoek(I: x: Element): ref: Knoop
    * Preconditie: de gelinkte lijst l bestaat
    * Postconditie: de referentie naar de knoop met dataveld x werd
geretourneerd, indien x niet voorkomt in de lijst werd referentie null
geretourneerd.
    * Gebruikt: /
BEGIN
    ref <- eerste
    ZOLANG (ref ≠ null EN ref.data ≠ x) DOE
        ref <- ref.volgende // Equivalent van i++
    EINDE ZOLANG
    RETOUR (ref)
EINDE
```

8.2.2.3 Algoritme voor het verwijderen van een component



```
verwijder(I: ref: Knoop): x: Element
    * Preconditie: de gelinkte lijst l bestaat, ref is niet de laatste knoop
in de lijst
    * Postconditie: de knoop die volgt na de knoop met referentie ref werd
verwijderd uit de lijst, het data-veld van de verwijderde knoop werd
geretourneerd
BEGIN
    x <- ref.volgende.data
    ref.volgende <- ref.volgende.volgende
    RETOUR (x)
EINDE
```

8.2.2.4 Algoritme voor het toevoegen van een component

```
voegToe(I: ref: Knoop, x: Element): /
    * Preconditie: de gelinkte lijst l bestaat.
    * Postconditie: na de knoop, waarnaar gerefereerd wordt door de
referentie ref, werd een nieuwe knoop met data-veld x toegevoegd.
    * Gebruikt: /
BEGIN
    hulp <- nieuwe Knoop()
    hulp.data <- x
    hulp.volgende <- ref.volgende
    ref.volgende <- hulp
EINDE
```

8.3 Dubbelgelinkte lijsten

In een dubbelgelinkte lijst kan je zowel naar de volgende als naar de vorige knoop gaan. Zo kan je snel zoeken zonder altijd van in het begin te beginnen. Dubbelgelinkte lijsten maken gebruik van 2 anker componenten.
Dubbelgelinkte Lijsten

8.4 Toepassing

> Stack op basis van knopen

Stack
- t : Knoop

+ Stack( )
+ empty( ) : boolean
+ push(x: Element) : /
+ pop( ) : Element
+ peek( ) : Element

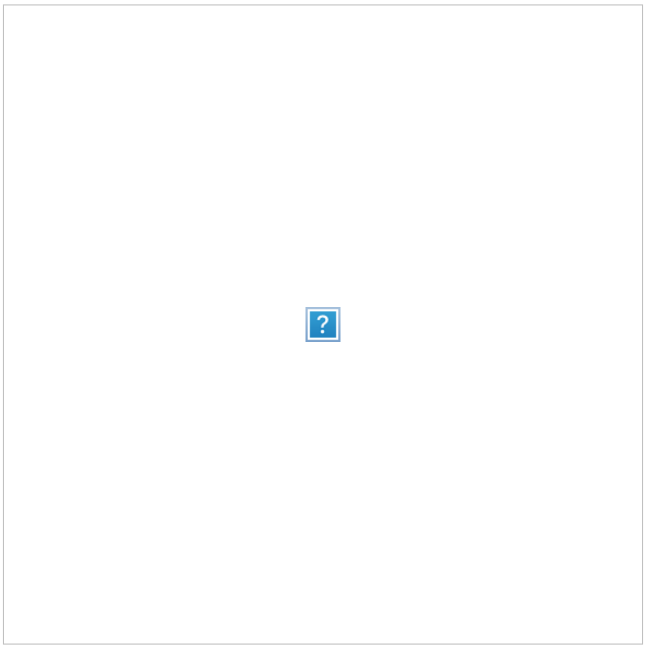
8.4.1 Algoritme voor de constructor

```
Stack(I: /): /
    * Preconditie: /
    * Postconditie: er werd een nieuwe stap aangemaakt, deze stap bestaat
als lege stapel
    * Gebruikt: /
BEGIN
    t <- null
EINDE
```

8.4.2 Algoritme ter controle of een stapel al dan niet leeg is

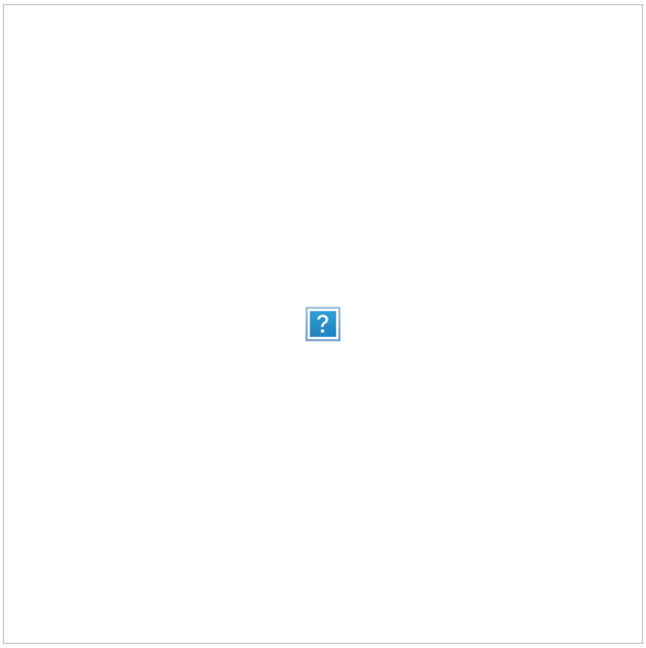
```
empty(I: /): vlag: boolean
    * Preconditie: de stapel s bestaat
    * Postconditie: de waarde true of false werd afgeleverd, afhankelijk van
het feit of de stapel s leeg is of niet.
    * Gebruikt: /
BEGIN
    RETOUR (t = null)
EINDE
```

8.4.3 Algoritme voor het toevoegen van een element aan een stapel



```
push(I: x: Element): /
    * Preconditie: de stapel s bestaat
    * Postconditie: het element x werd als top-element op de stapel s geduwd
    * Gebruikt: /
BEGIN
    hulp <- nieuwe Knoop()
    hulp.data <- x

    hulp.volgende <- t
    t <- hulp
EINDE
```

8.4.4 Algoritme voor het verwijderen van een element van een stapel



```
pop(I: /): x: Element
    * Preconditie: de stapel s bestaat
    * Postconditie: het element x werd geretourneerd
    * Gebruikt: /
BEGIN
    x <- t.data

    t <- t.volgende

    RETOUR (x)
EINDE
```

9 Hashtabellen

Oefeningen

9.1 Woordenboeken met tweeletterwoorden

```
'aa' -> Gezonde melk
...
'xx' -> popgroep
'zz' -> geluid dat men maakt als men slaapt
```

> # woorden = 26 * 26 = 676
>
> array van lengte 676 (indices 0 t.e.m. 675
>
> Afbeelding van 'woorden' naar 'plaats in de array'

```
a ~> 0
b ~> 1
...
z ~> 25

'aa' ~> 26 *  0 +  0 =   0
'bc' ~> 26 *  1 +  2 =  28
'zz' ~> 26 * 25 + 25 = 675

=> Berekenen van de hashCode
```

9.1 Specificatie

> Langste woord:
>
> Hottentottententententoonstelling: 33 letters

```
grootte array:  26^33
                27^33 (Als je lege spaties wilt)

-> array van grootte N (veel kleiner dan 26^33)
-> bereken nog steeds hashcode:
    ~> positie in de array = hashcode MOD N
```

```
| ------------ | ------------ | ------------ |
| Gezonde Melk |              |              |
| ------------ | ------------ | ------------ |
```

9.1.2 Implementatie

9.1.2.1 Algoritme voor de methode hashCode

```
w.hashCode(I: /): positie: geheel getal
    * Preconditie: de sleutel w bestaat
    * Postconditie: een getal tussen 0...675 werd berekend en geretourneerd
    * Gebruikt:
BEGIN
    positie <- 26 * w[0] + w[1]

    RETOUR (positie)
EINDE
```

9.1.2.2 Algoritme voor de methode voegToe

```
b.voegToe(I: w: Sleutel, definitie: String): /
    * Preconditie: de sleutel w wordt meegegeven, zijn definitie moet aan
het woordenboek b toegevoegd worden.
    * Postconditie: de gegevens betekenis van w werd aan het woordenboek b
toegevoegd op de juiste positie.
    * Gebruikt: hashCode.
BEGIN
    b[w.hashCode()] <- definitie
EINDE
```

9.1.2.3 Algoritme voor de methode zoekOp

```
b.zoekOp(I: w: Sleutel): definitie: String
    * Preconditie: de sleutel w wordt meegegeven.
    * Postconditie: de definitie van w werd geretourneerd.
    * Gebruikt: hashCode.
BEGIN
    definitie <- b[w.hashCode()]
    RETOUR(definitie)
EINDE
```

9.2 Specificatie

Voor de opbouw van het woordenboek met tweeletterwoorden was er nood aan een functie die alle sleutels afbeeldt op een positie (bucket) van een tabel. Aangezien het over een beperkt aantal sleutels ging, was het mogelijk om de tabel voor te stellen door een array.
Wanneer alle mogelijke woorden van willekeurige lengte moeten opgenomen worden in het woordenboek dan wordt het aantal benodigde sleutels aanzienlijk groter. Stel dat het woord 'hottentottententententoonstelling', dat 33 letters telt, het langste op te nemen woord is. Dan moeten er, om alle mogelijke woorden te kunnen opnemen, 2633 sleutels worden voorzien. Werken met een array van dergelijke lengte is niet meer efficiënt naar geheugengebruik toe.
Een mogelijke oplossing is om bij aanvang een getal N te selecteren dat het maximum posities van de te gebruiken tabel, waarin alle waarden worden opgeslagen, aangeeft. Dit betekent dat de hashtabel in een array van lengte N opgeslagen wordt.
Een woord wordt, analoog aan het voorbeeld van de tweeletterwoorden, omgezet naar een code via een methode hashCode. Er zullen veel meer hashcodes zijn dan enkel de waarden tussen 0 en N – 1. Standaard wordt het totaal aantal woorden op te nemen in het woordenboek voorgesteld door n. Dit aantal n komt overeen met het aantal benodigde hashcodes. Over het algemeen zal n > N.
Aangezien er slechts N posities voorzien zijn in de tabel moeten al deze hashcodes met een hashfunctie omgezet worden naar een correcte positie uit de tabel.
Een mogelijke hashfunctie h die de positie berekent van alle sleutels w die overeenkomen met alle op te nemen woorden in het woordenboek, zou kunnen zijn:
`h(w) = w.hashCode() (mod N).`
Het resultaat van deze hashfunctie is steeds een getal tussen 0 en N – 1. Wat overeenkomt met alle mogelijke posities in de tabel.
Door deze hashfunctie toe te passen op alle sleutels is het zeer plausibel dat de betekenis van een aantal sleutels op dezelfde positie moet opgeslagen worden. De opbouw van de hashtabel zal afhangen van de manier waarop wordt omgegaan met deze samenvallende posities.
Algemeen kunnen we stellen dat hashing kan opgesplitst worden in twee luiken:

1. De keuze van een hashfunctie h die alle mogelijke sleutels afbeeldt op een positie uit de hashtabel.
2. Een methode selecteren om de waarden die overlappen of botsen (collisions) te verwerken.

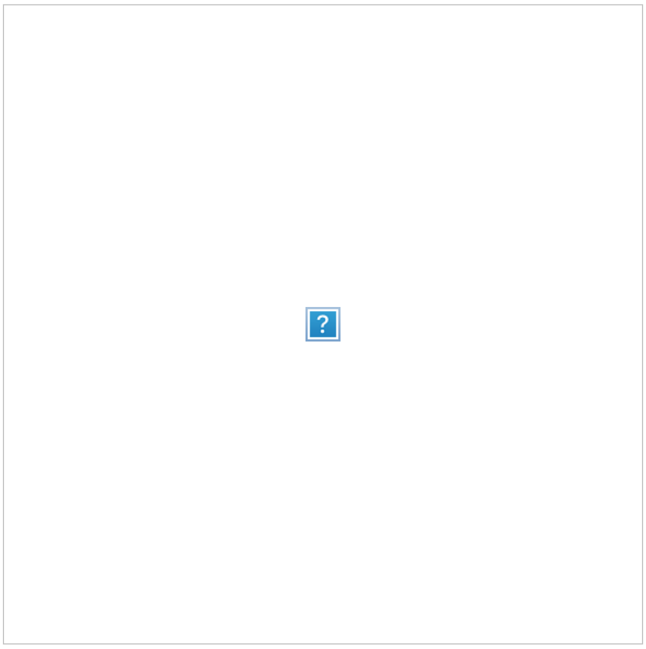
9.2.1 Verwerken van de overlappingen

9.2.1.1 Gesloten Hashing

```
   0      1      2      3      4      5      6      7      8      9
| ---- | ---- | ---- | ---- | ---- | ---- | ---- | ---- | ---- | ---- |
|  10  |  100 |  129 |      |      |  15  |  115 |      |      |  29  |
| ---- | ---- | ---- | ---- | ---- | ---- | ---- | ---- | ---- | ---- |

  10   hashCode(10)  = 10   10  MOD 10 = 0
```

```
15   hashCode(15)  = 15   15  MOD 10 = 5
29   hashCode(29)  = 29   29  MOD 10 = 9
100  hashCode(100) = 100  100 MOD 10 = 0
115  hashCode(115) = 115  115 MOD 10 = 5
129  hashCode(129) = 129  129 MOD 10 = 9

Positie 1 is al bezet, dus we kijken naar de eerst volgende lege plaats
```

9.2.1.2 Open Hashing



```
10   hashCode(10)  = 10   10  MOD 10 = 0
15   hashCode(15)  = 15   15  MOD 10 = 5
29   hashCode(29)  = 29   29  MOD 10 = 9
100  hashCode(100) = 100  100 MOD 10 = 0
115  hashCode(115) = 115  115 MOD 10 = 5
129  hashCode(129) = 129  129 MOD 10 = 9
```

> Je wilt liefst veel korte lijsten; Want 1 lange lijst is eigenlijk een gewone gelinkte lijst
>
> Bij **Modulo**, kan je beter voor een priemgetal kiezen.