# Analyse Algoritmen

**Basisstructuren:**

* **SequentieStructuur =>** programmadeel bestaande uit een reeks van 2 of meer opeenvolgende opdrachten.
* Geeft aan in welke volgorde de operaties gebeuren, opeenvolging van stappen, 1 voor 1 doorlopen.
* Opdrachten worden van **boven naar beneden** uitgevoerd.
* **SelectieStructuur** => voorwaardelijke opdrachten, enkel uitgevoerd onder bepaalde conditie.
* Bestaat dus uit een Selectievoorwaarde gecombineerd met 2 of meerdere verschillende componenten, waarvan er slechts 1 wordt uitgevoerd.
* Kan ook een samengestelde propositie zijn als voorwaarde
* Selectiestructuur kan ook genest zijn (ALS in een ALS)
* **ALS…DAN/ANDERS…DAN**
* **IteratieStructuur =**> zolang aan een iteratievoorwaarde voldaan wordt, wordt aangegeven onderdeel uitgevoerd.
* Bestaat uit iteratievoorwaarde en iteratiecomponent/onderdeel
* **ZOLANG…DOE/VOOR…TOT…DOE**

**Methodes onderdelen:**

1. **Methodeheader**: Legt naam vast van methode, toont input-parameters, output-parameters
2. **Specificaties:**

* Precondities: Beginvoorwaarden om methode te kunnen uitvoeren.
* Postcondities: Verduidelijking welk resultaat methode zal leveren.
* Import/gebruikt: De methodes die zullen worden aangeroepen om te gebruiken.
* Body: tussen “BEGIN” en “EINDE”

# Uitvoeringstijd algoritme

**Uitvoeringstijd algoritme:**

* **Efficiëntie** van algoritme.
* Aantal instructies die moeten worden uitgevoerd
* Aantal instructies => afhankelijk van de grootte van de inputvariabele.

T wordt voorgesteld in functie van de inputgrootte (n).

Uitvoeringstijd in verschillende functies:

* **Constante functie:** aantal instructies staat vast, onafhankelijk van inputwaar.
* **Lineaire uitvoeringstijd:** afhankelijk van inputgrootte van n (bvb T(n) = 5n +4
* **Kwadratische uitvoeringstijd:** dubbele lus (bvb: T(n) = 5n^2+4n+4

Voor kleine inputwaarden is de keuze van algoritme adhv uitvoeringstijd niet belangrijk, aangezien deze niet zo veel zal verschillen.

Bij grote inputwaarden zal dit wel belangrijk zijn dus we zullen een **asymptotische analyse** moeten maken (n tot oneindig bekijken).

Enkel focus op de sterkst groeiende term in uitdrukking, andere termen mogen worden verwaarloosd.

* O-notatie: bvb kwadratisch T(n) = O(n^2)

Meest efficiënte: log2n, dan 2n, dan n2 dan 2^n

# Recursie

Een recursieve functie is een functie die zichzelf aanroept

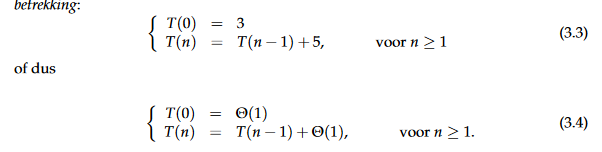
Bestaat uit 2 delen:

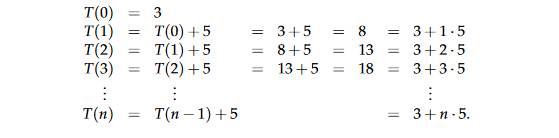
* **Een basisgeval:** beschrijft hoe de functie berekend wordt voor één of meerdere eindegevallen.
* **Een recursief gedeelte:** beschrijft hoe de functie moet berekend worden voor alle andere inputwaarden, alle inputwaarden != het basisgeval.

Uitvoeringstijd van een recursieve methode => berekend adhv **recurrente betrekking**.

Voor het bepalen van de O-notatie van de uitvoeringstijd moet de recurrente betrekking herleid worden naar een gesloten formule.

Recurrente betrekking:



Hoe kan je een gesloten formule opstellen? **Waarden uitschrijven en patroon herkennen**

**Wiskundige inductie:**

**Bewijs:** T(n) = 3 + 5n, voor n>=0

**Gegeven:**

T(0) = 3

T(n) = T(n-1)+5, voor n>= 1

**Basisstap (n = 0):** Voor n = 0 gegeven T(0) = 3 => komt overeen met t.b. formule: T(0) = 3 +5\*0=3

**Inductiestap:**

We veronderstellen dat T(m) = 3+5m voor alle m<= n => bewijzen dat formule ook geldig is voor n+1. Om dit te bewijzen gaan we aantonen dat T(n+1) = 3+5(n+1)

T(n+1) = T(n) +5 (gegeven) (T(n-1)+5 => T((n+1)-1)+5 => T(n) +5)

=(3+5n)+5 (veronderstelde)

= 3+(5n+5)

=3+5(n+1) = **Bewijst het gestelde.**

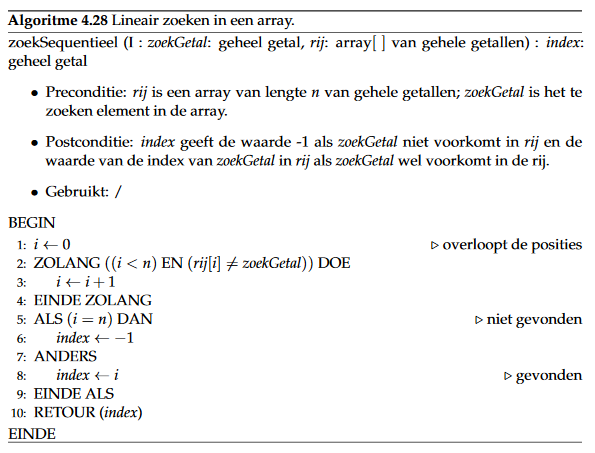
# Zoek- en sorteeralgoritmes

## Zoekalgoritmes

### Sequentieel/lineair zoeken

Elk element van de rij overlopen.

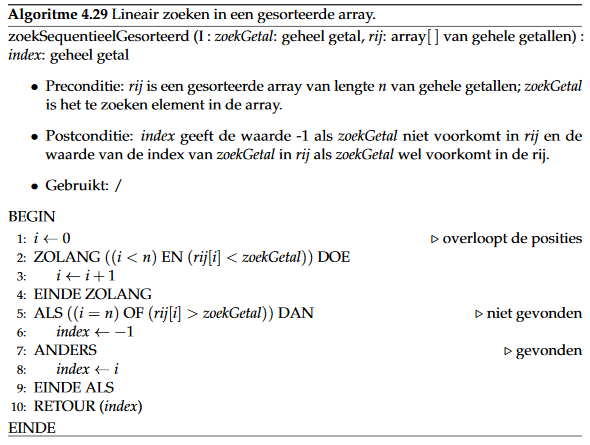
**Ongesorteerde rij:**



Hier is de uitvoeringstijd:

* In het beste geval: T(n) = O(1)
* In het slechtste geval: T(n) = O(n)
* In het gemiddelde geval: T(n) = O(n)

**Lineair zoeken is EFFICÏENTER als de rij gesorteerd is:**



Hier is de uitvoeringstijd weeral:

* In het beste geval: T(n) = O(1)
* In het slechtste geval: T(n) = O(n)
* In het gemiddelde geval: T(n) = O(n)

### Binair zoeken in een array

Bij binair zoeken gaat men beginnen met zoeken in het midden van de rij, kleiner dan gezochte getal? => verder gezocht in rechterhelft v/d rij. Anders wordt er in de linkerhelft verder gezocht. Dit werkt enkel in een **GESORTEERDE RIJ**

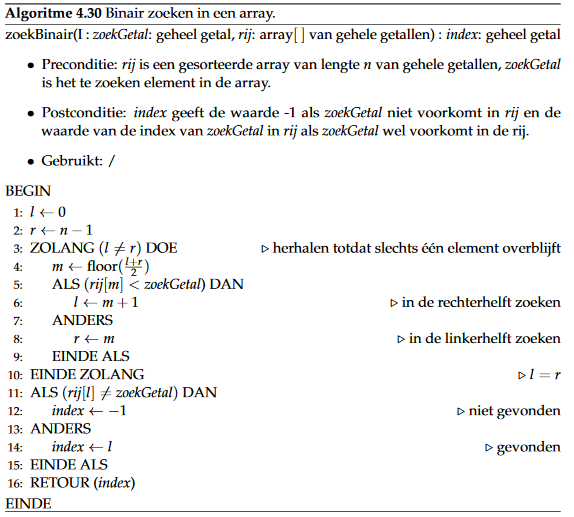
**Uitvoeringstijd/Tijdscomplexiteit:** O(log2n) (zeer goed, ook bij grote arrays)

**Twee mogelijke implementaties:**

* Iteratief
* Recursief

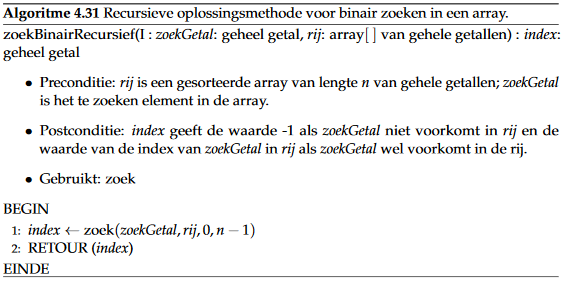
#### Binair Iteratief zoeken

Hier wordt er gebruik gemaakt van een lus om het te onderzoeken deel van de rij te blijven halveren totdat er slechts één element overblijft.

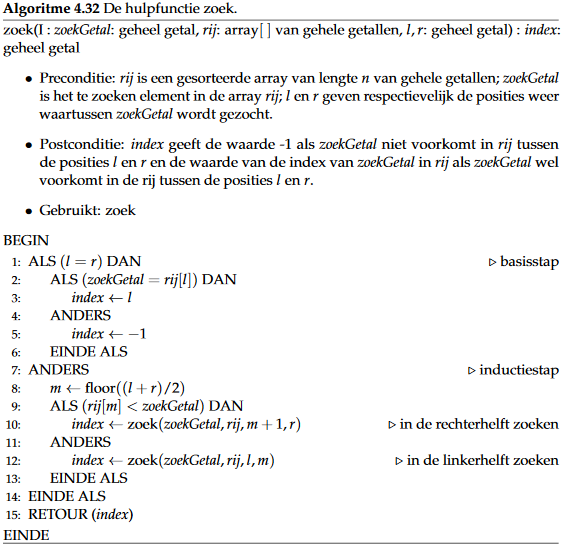


#### Binair Recursief zoeken

Bevat steeds een selectiestructuur, deze beschrijft wat er moet gebeuren in het basisgeval en wat er moet gebeuren in de recursieve stap.



Dit recursieve algoritme maakt gebruik van de hulpfunctie “zoek”:

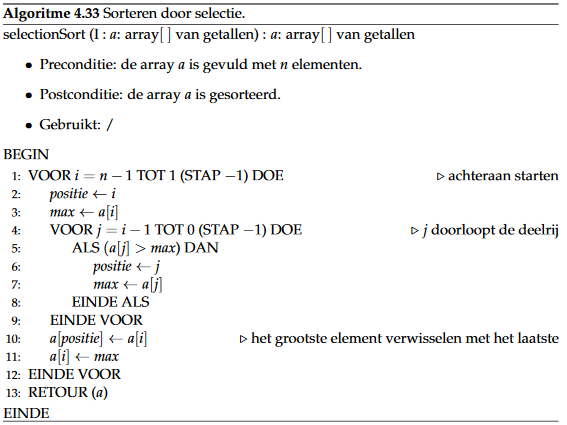


## Sorteeralgoritmen

Sorteren van gegevens heeft positieve invloed op verder verwerken ervan.

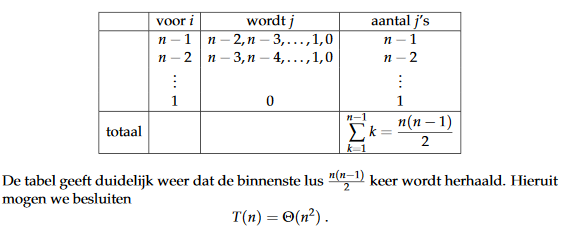
### Sorteren door selectie

Zoeken naar grootste element, indien het niet achteraan staat wordt het verwisseld met het element op de laatste plaats.



**Complexiteitsanalyse:**

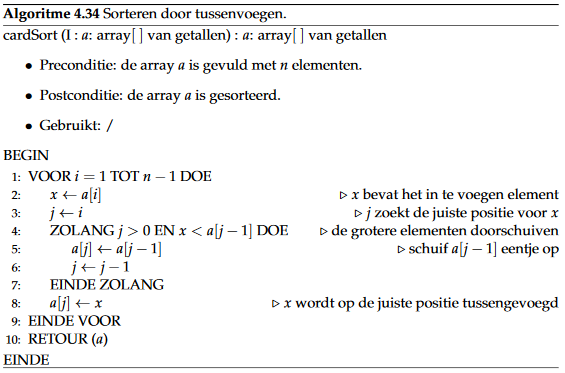
Uitvoeringstijd:



De formule van de binnenste lus is een vaste formule die je gewoon vanbuiten moet kennen.

### Sorteren door tussenvoegen, CardSort

Kan het best vergeleken worden met het op volgorde steken van kaarten. Vandaar de naam CardSort. We beginnen met het twee element in de rij en kijken of die voor de eerste moet komen of niet. Zo schuiven we op in de rij.

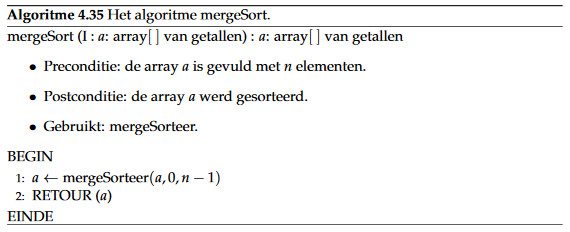


**Complexiteitsanalyse:**

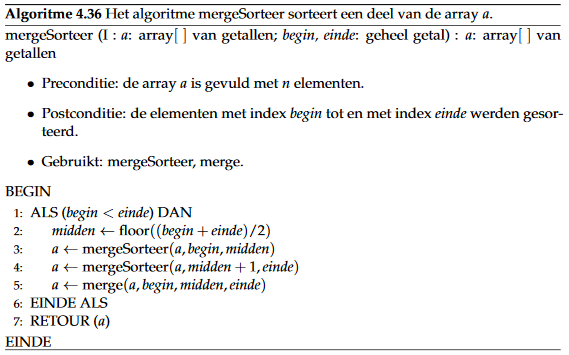
* Beste geval(de rij is reeds gesorteerd): T(n) = O(n)
* Slechtste geval (de rij staat in omgekeerde volgorde gesorteerd bij aanvang): T(n) = O(n^2)
* Gemiddelde geval(willekeurige ordening): T(n) = O(n^2)

### Mergesort

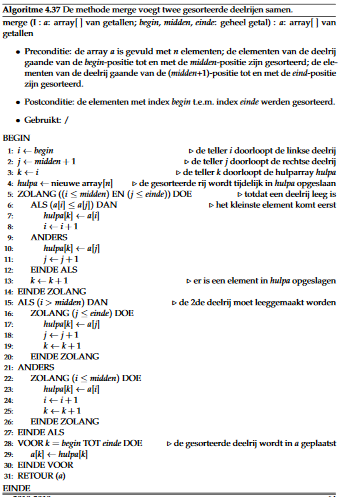
Recursieve methode, de rij wordt in 2 gesplitst, beide in dezelfde lengte (indien mogelijk). Beide worden recursief gesorteerd. Recursie stopt wanneer een deelrij van lengte 1 bereikt is. De 2 gesorteerde deelrijen worden samengevoegd, gemerged, tot één gesorteerde rij.



Maakt gebruik van mergeSorteer:



Deze maakt dan op zijn beurt gebruik van merge:



**Complexiteitsanalyse:**

Recurrente betrekking:

T(1) = O(1)

T(n) = 2T((n/2)+f(n) (n>1)

Met f(n) de uitvoeringstijd van algoritme merge.

Lineaire uitvoeringstijd => T(n) = O(nlogn)

**Mergesort is efficiënter op vlak van snelheid dan sorteren door selectie en als sorteren door tussenvoegen MAAR is veel minder efficiënt op vlak van benodigde geheugenruimte!**

### QuickSort

Recursieve methode voor het sorteren van een rij.  
Basisgeval als te sorteren rij slechts 1 element bevat, in dit geval eindigt de recursie.

Van alle besproken algoritmen is quicksort het snelst voor grote inputwaarden n.

#### Bepalen van de spil

Bij quickSort wordt het probleem opgesplitst in kleinere delen, hiervoor gebruiken we een spil, dit is een willekeurig element uit de rij.

Het kiezen van **de spil** heeft invloed op de uitvoeringstijd van het algoritme:

* **Beste geval:** T(n) = O(nlgn)
* **Gemiddelde geval:** T(n) = O(n^2)
* **Slechtste geval:** T(n) = O(nlgn)

**Spil:**

* *Actieve*keuze = mediaan van drie, mediaan bepalen is tijdsintensief => **slechts een schatting = Sampling**. Sample = eerste, middelste en laatste element van de rij. Spil = mediaan van deze drie.
* Slechts mogelijke situatie uitgesloten (spil kan onmogelijk grootste of kleinste element van de rij zijn.)
* *Passieve* keuze = middelste element van rij als spil, indien rij gesorteerd is, is dit de beste keuze.

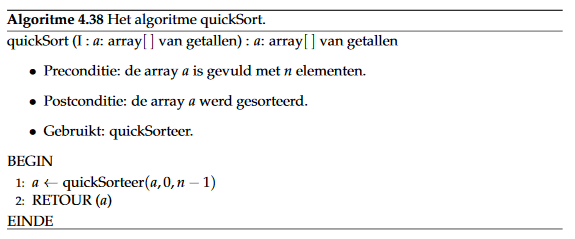
#### Het partitioneren

1. Spil s wordt achteraan geplaatst door ze te verwisselen met laatste element
2. Alle elementen kleiner dan s => naar links. Groter dan s => naar rechts. Alle elementen kleiner dan s worden gezocht van rechts naar links, via positie recht.  
   Indien er links een te groot element wordt gevonden en rechts een te klein element, worden beide van plaats verwisseld. Zolang beide posities, links en rechts, elkaar niet kruisen wordt dit proces herhaald.
3. Het grootste element wordt verwisseld met spil s.

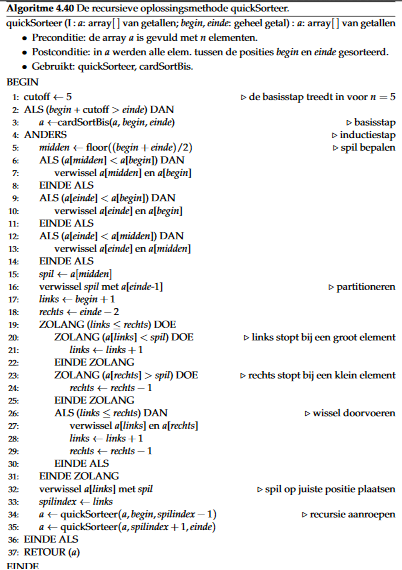
#### Verfijningen

* Beide lussen, links en rechts, zullen stoppen als een element gelijk aan de spil gevonden wordt. Zo worden alle elementen gelijk aan de spil over beide helften verdeeld zodat het aantal elementen van de deelrijen niet te sterk verschillend is.
* **Partitioneren met de mediaan van drie**, zoals eerder vermeld is dit de meest optimale keuze. Indien deze methode gebruikt wordt voor de spil kan het opsplitsen in deelrijen geoptimaliseerd worden.  
  De spil is sowieso groter dan het eerste element en kleiner dan het laatste element van de rij. Volgende verfijningen ontstaan hieruit:
* Spil kan verwisseld worden met voorlaatste element ipv laatste.
* “Links” kan starten bij het tweede element van de deelrij ipv het eerste.
* “Rechts” kan starten net voor de spil.

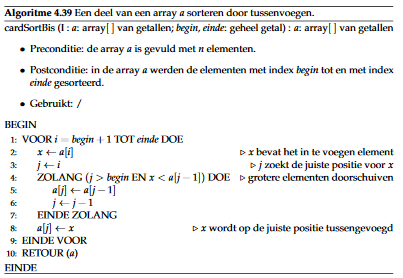
In het geval van kleine arrays of rijen, is het niet nuttig om quickSort te gebruiken daarom zullen we cardSortBis aanroepen, deze gaat sorteren met tussenvoegen.



Maakt gebruik van quickSorteer:



Maakt in het basisgeval gebruik van cardSortBis, dit gebeurd wanneer n < 5



DEEL 2

# Stapels

Dit is een datastructuur waarbij de toegang enkel beperkt is tot de uiteinden van de structuur.

* Een element dat we toevoegen aan de stapel komt bovenop de reeds bestaande stapel te liggen.
* Een element verwijderen is enkel mogelijk indien dit het bovenste element is. Dit noemt men *de top* van de stapel.
* Stapels passen LIFO toe, Last-In-First-Out.

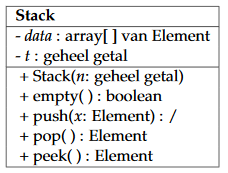
## Basisbewerkingen van de klasse **Stack**

* **Stack()**: Constructor, maakt een nieuwe stapel aan waarna er een lege stapel bestaat.
* **empty(**): Controleert of de stapel al dan niet leeg is.
* **push():** Voegt een nieuw element toe bovenaan de stapel, dit wordt de nieuwe top.
* **pop():** Verwijdert het bovenste element van de stapel en retourneert het verwijderde element.
* **peek() :** Retourneert het bovenste element zonder het te verwijderen.

Al de implementaties van deze basisfuncties zijn uitvoerbaar in constante tijd, de tijd is onafhankelijk van de lengte van de array.

T(n) = O(1)

## UML



## Toepassing 1: controle van Haakjes

### Methode

Voor de controle van haakjes worden twee zaken onderzocht:

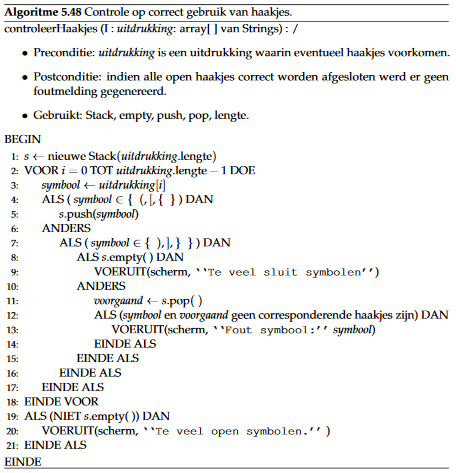
1. Controleren of elk openend haakje één overeenkomstig sluithaakje heeft. M.a.w. de symbolen moeten in paren voorkomen.
2. De volgorde regels moeten worden gerespecteerd. Paren haakjes mogen niet overlappen   
   vb: ([)] mag NIET.

Dit kan met een algoritme, met behulp van een stapel, geverifieerd worden als volgt:

* Maak een lege stapel aan
* Alle symbolen 1-voor-1 doorlopen. Indien het een haakje is, moet er een opdracht uitgevoerd worden:
* Open-symbool => op stapel plaatsen (push-bewerking)
* Sluit-symbool => 2 mogelijke situaties onderscheiden:

1. Stapel is leeg: foutmelding aangezien er geen corresponderend open-symbool vooraf gegaan is.
2. Stapel niet leeg: top-element wordt van stapel gehaald (pop-bewerking), dit symbool wordt vergeleken met ingelezen symbool, indien beide niet corresponderen een foutmelding.

* Als alle karakters ingelezen zijn en de stapel is niet leeg => foutmelding, dit betekend dat er nog open-symbolen zijn zonder sluit-symbool.



## Toepassing 2: Berekenen van postfix-uitdrukkingen

Infix => 2 X 3, 2 + 3,…

Postfix => 2 3 X, 2 3 +,…

### Waarde van de postfix-uitdrukking bepalen.

Waarde van postfix bepalen gaat als volgt:

* Uitdrukking wordt van links naar rechts doorlopen.
* Operand wordt gelezen => wordt op stapel geplaatst.
* Binaire operator wordt gelezen => twee laatst gestapelde operanden van stapel gehaald en de bewerking met die 2 operanden wordt uitgevoerd. Resultaat => op stapel geplaatst.
* Wordt herhaald totdat de volledige uitdrukking is doorlopen.
* Doorlopen = gedaan => 1 element op de stapel = waarde van de postfix-uitdrukking.

### Infix naar postfix

Conversie gaat als volgt:

1. Invoertekst van links naar rechts lezen.
2. Operand (getal) wordt gelezen => rechtstreeks naar uitvoertekst
3. Operator of haakje wordt gelezen => tijdelijk op stapel bewaard indien 1 van volgende situaties zich voordoet:

* Stapel is leeg
* Gelezen operator heeft hogere prioriteit dan operator bovenaan op stapel
* Gelezen haakje is openingshaakje, => laagste prioriteit, alles mag er bovenop gelegd worden.

Indien gelijke of lagere prioriteit dan top van de stapel => alle operatoren van de stapel met gelijke of hogere prioriteit van de stapel gehaald en worden deze toegevoegd aan de uitvoertekst. Dit gebeurd totdat de stapel leeg is. Daarna wordt de ingelezen operator op de stapel geplaatst.  
 Indien een sluitingshaakje wordt ingelezen => alle operatoren van de stapel gehaald en toegevoegd aan uitvoertekst totdat openhaakje in de stapel bereikt is. Haakje wordt van de stapel gehaald maar niet in uitvoertekst gezet.

1. Einde invoertekst bereikt => Alle operatoren worden van de stapel gehaald en aan de uitvoertekst toegevoegd totdat stapel leeg is.

# Wachtrijen (Queues)

* Element dat wordt toegevoegd aan een wachtrij wordt achteraan toegevoegd, dit noemen we *de staart* van de wachtrij
* Element van de wachtrij verwijderen is enkel vooraan mogelijk. We kunnen enkel het eerste element bereiken, dit noemen we *de kop*.

FIFO, First-In-First-Out-structuur.

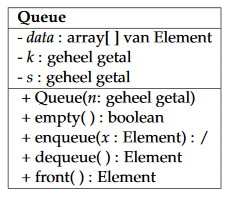
## Basisbewerkingen

* **Queue():** Constructor, maakt een nieuwe, voorlopig lege, wachtrij aan.
* **empty():** Controleert of een wachtrij al dan niet leeg is.
* **enqueue():** Voegt een gegeven element toe aan de staart van een wachtrij.
* **dequeue():** Verwijdert het element aan de kop en retourneert het verwijderde element.
* **front():** Retourneert de kop zonder het te verwijderen.
* **Uitvoeringstijd is constant** T(n) = O(1)

De elementen van een wachtrij worden in een array data opgeslagen.

De gehele getallen k en s houden de index van de kop en de staart bij => Bij een lege wachtrij zijn deze beide = -1.

## UML



# Lijsten

Een lijst, soms ook wel sequence genoemd, is een geordende opsomming van elementen.

In een lijst kan eenzelfde element meerdere keren voorkomen en alle elementen moeten bereikbaar zijn.

**Tijdscomplexiteit:** Constante tijd (behalve voor geefPositie en verwijderElem)

## Basisbewerkingen

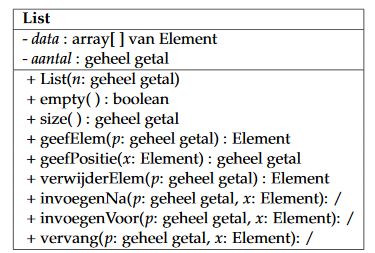
* **List():** Constructor, maakt een nieuwe, lege lijst aan.
* **empty():** Controleert of een lijst al dan niet leeg is.
* **size():** Geef het aantal elementen van een lijst terug.

## Extra bewerkingen met toegang tot de lijst

* **geefElem():** Retourneert het element dat zich bevindt op de positie bepaald door het argument.
* **geefPositie():** Retourneert de positie van het eerste voorkomen van het meegeleverd argument.
* **verwijderElem():** verwijdert het element dat zich op de positie van het meegeleverd argument bevindt en retourneert dit element.
* **invoegenNa():** Het element, dat als tweede argument wordt meegeleverd, wordt ingevoegd na de meegegeven positie.
* **invoegenVoor():** Het element, dat als tweede argument wordt meegeleverd, wordt ingevoegd 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.

## UML

* De elementen van een lijst worden in een array *data* opgeslagen.
* *Aantal* = hoeveel elementen er tot de *lijst* *l* behoren



# Gelinkte lijsten

Dit is een alternatieve implementatie van een lijst.

Het voordeel van een gelinkte lijst is dat het uitvoeren van de basisfuncties zoals toevoegen of verwijderen van elementen in een **constante tijd** gebeurt, op voorwaarde dat men weet waar het element moet worden toegevoegd of verwijderd worden. Het opzoeken van een element vraagt echter nog steeds lineaire tijd.

Een ander voordeel is dat deze geen beperking oplegt aan het aantal elementen dat aan de lijst kan worden toegevoegd. Een gelinkte lijst heeft een **dynamische structuur**.

## Structuur

Bestaat uit **knopen** die via een **kettingstructuur** aan elkaar gekoppeld zijn.

Knoop bestaat uit 2 velden:

* Data-veld **data** waarin de data wordt opgeslagen.
* Veld **volgende** dat verwijst naar volgende knoop in de lijst.

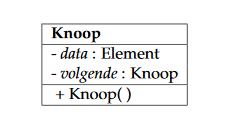
De laatste knoop => null, op figuren wordt deze aangeduid met een streep.

Eerste knoop => referentie *eerste*, in een lege lijst is deze = null.

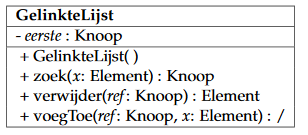
## Basisbewerkingen enkelvoudig geschakelde lijst

* **zoek():** Zoekt de positie van de knoop met als data-veld het argument.
* **verwijder():** Verwijdert de knoop die volgt na de opgegeven knoop en retourneert de waarde van het data-veld van de verwijderde knoop.
* **voegToe():** Voegt een knoop toe na een opgegeven knoop, het data-veld krijgt de waarde van het argument.

## UML Knoop



## UML GelinkteLijst



**Tijdscomplexiteit zoek():**

* Beste geval = constante uitvoeringstijd
* Gemiddelde geval = lineaire uitvoeringstijd
* Slechtste geval = lineaire uitvoeringstijd

## Ankercomponenten

Lost het probleem op van een element toe te voegen aan een lege lijst of dat van een element toe te voegen vooraan in de lijst, wat niet mogelijk is met de klassieke voegToe() methode.

Eveneens lost het ook het probleem op dat het niet mogelijk is om met de klassieke verwijder() methode het eerste element van een gelinkte lijst te verwijderen.

Het ankercomponent is een extra knoop die vooraan aan de gelinkte lijst wordt toegevoegd. Het data-veld hiervan is leeg en het referentie-veld verwijst naar de eerste knoop. Het ankercomponent maakt zelf GEEN deel uit van de lijst maar dient enkel om implementatie te vereenvoudigen.

## **Dubbel gelinkte lijsten**

In een enkelvoudig gelinkte lijst is de eerste knoop bereiken vrij simpel. De laatste daarin tegen kost tijd lineair in de lengte van de lijst.

* Oplossing Dubbel gelinkte lijst

### Structuur

We gaan 2 ipv 1 referentie bijhouden: *Volgende* en *Vorige*

Naast de referentie *eerste* die de eerste knoop aanduidt gaan we ook een referentie *laatste* in de lijst bijhouden.

### Methodes

