# Gelinkte lijsten

## Classic Computer Science Algorithms

**Stijn Lievens  Pieter-Jan Maenhaut  Koen Mertens  Lieven Smits**

**AJ 2021–2022**

HO GENT

# Gelinkte lijsten

HO
GENT

# Inhoud



HO GENT

# Inleiding

- Array = eenvoudige datastructuur
- Maar wat met elementen tussenvoegen? Of verwijderen in het midden?
- Bewerkingen hebben lineaire tijdscomplexiteit
- Oplossing? $\Longrightarrow$ Gelinkte lijsten
- Toevoegen of verwijderen in constante tijd
- Opzoeken van een element blijft echter een lineare tijdscomplexiteit vertonen
- Nog een voordeel: aantal elementen toevoegen kan onbeperkt, geen limieten op de grootte zoals bij array

HO GENT

# Definitie

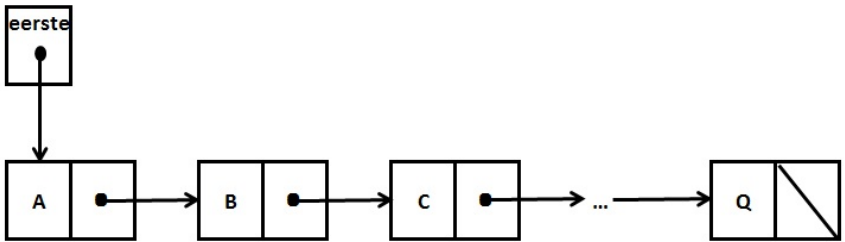
Een gelinkte lijst bestaat uit een aantal *knopen* die via een *kettingstructuur* aan elkaar geschakeld zijn. Een knoop bestaat uit twee velden:

- een data-veld *data*
- een veld *volgende*

De laatste knoop bevat een wijzer *null*: dit wordt grafisch voorgesteld door een schuine streep.
Voor de eerste knoop moet een referentie *eerste* bijgehouden worden.
In een lege lijst is de referentie *eerste* gelijk aan null.

HO GENT

# Een enkelvoudig geschakelde lijst




HO GENT

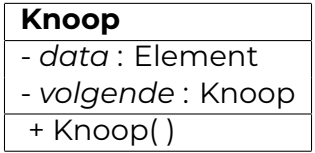
# Basisbewerkingen

De belangrijkste basisbewerkingen voor een enkelvoudig geschakelde lijst zijn:

- *zoek*( ): zoekt de positie van de knoop met als data-veld het argument;
- *verwijder*( ): verwijdert de knoop die volgt na de opgegeven knoop en geeft de waarde van het data-veld van de verwijderde knoop weer;
- *voegToe*( ): voegt een knoop toe na een opgegeven knoop, het data-veld krijgt de waarde van het tweede argument.

HO GENT

# De klasse Knoop in UML




De gedefinieerde Knoop is een datastructuur die een element van een niet nader gedefinieerde klasse Element bevat.

HO GENT

# Constructor voor een Knoop

De constructor Knoop maakt een nieuw object van de klasse Knoop aan.

**Invoer** /
**Uitvoer** er werd een nieuwe knoop aangemaakt

```
1: function KNOOP
2:     data ← null
3:     volgende ← null
4: end function
```

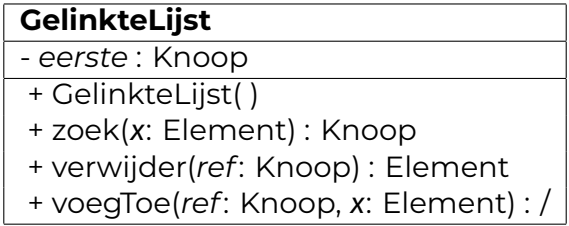
HO GENT

# Implementatie Gelinkte Lijst

De klasse Knoop wordt als inwendige klasse (inner class) van de klasse GelinkteLijst geïmplementeerd. Dit betekent dat methodes van de klasse GelinkteLijst toegang hebben tot de velden van Knoop. We schrijven de klasse GelinkteLijst uit in UML-notatie. Alle methodes worden in pseudocode beschreven.

HO GENT

# De klasse GelinkteLijst in UML




HO GENT

# Algoritme voor de Constructor

De constructor GelinkteLijst maakt een nieuw object van de klasse GelinkteLijst aan.

**Invoer** /

**Uitvoer** er werd een nieuwe (lege) gelinkte lijst aangemaakt

```
1: function GELINKTELIJST
2:     eerste ← null
3: end function
```

HO GENT

# Opzoeken van een element *x*

**Invoer** de gelinkte lijst werd aangemaakt, *x* is het te zoeken element
**Uitvoer** de referentie naar de eerste knoop met dataveld gelijk aan *x* werd geretourneerd, indien *x* niet voorkomt in de lijst werd de referentie null geretourneerd.

```
1: function ZOEK(x)
2:     ref ← eerste
3:     while ref ≠ null and ref.data ≠ x do
4:         ref ← ref.volgende
5:     return ref
6: end function
```

HO GENT

# Tijdscomplexiteit van ”zoek”

De uitvoeringstijd voor de methode *zoek* is afhankelijk van de positie van *x* in de lijst.

- Slechtste geval: elke knoop van de lijst overlopen en *x* niet gevonden $\Longrightarrow$ lineaire uitvoeringstijd.
- Beste geval: gezochte referentie onmiddellijk gevonden $\Longrightarrow$ constante uitvoeringstijd.
- Gemiddeld geval: de helft van de knopen overlopen $\Longrightarrow$ lineaire uitvoeringstijd.

HO GENT

# Vervolg tijdscomplexiteit

Zoeken van een element *x* in een gelinkte lijst is *niet* efficiënter dan in een *gewone* array.

Integendeel, in de praktijk is het volgende van referenties (*pointers*) trager dan het doorlopen van een array.

De overige basisfuncties voor een gelinkte lijst zullen wel allemaal uitvoerbaar zijn in een constante tijd, op voorwaarde dat de referentie reeds gekend is.

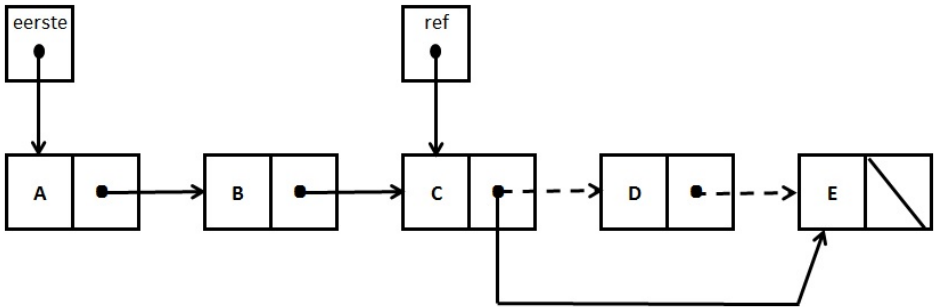
HO
GENT

# Verwijderen van een knoop

De functie VERWIJDER verwacht als inputwaarde een referentie *ref*. Deze referentie *ref* verwijst naar de knoop in de gelinkte lijst die de te verwijderen knoop voorafgaat en de waarde van het data-veld van de verwijderde knoop wordt geretourneerd.
Waarom *argument* ref nodig? En waarom ref naar *voorgaande* knoop?

HO GENT

# Verwijderen van een knoop




HO GENT

# Verwijderen van een knoop

**Invoer** de gelinkte lijst *l* bestaat, de knoop *ref* is niet de laatste knoop in de lijst.

**Uitvoer** de knoop die volgt na de knoop met referentie *ref* werd verwijderd uit de lijst, het data-veld van de verwijderde knoop werd geretourneerd.

```
1: function VERWIJDER(ref)
2:     x ← ref.volgende.data
3:     ref.volgende ← ref.volgende.volgende
4:     return x
5: end function
```

HO GENT

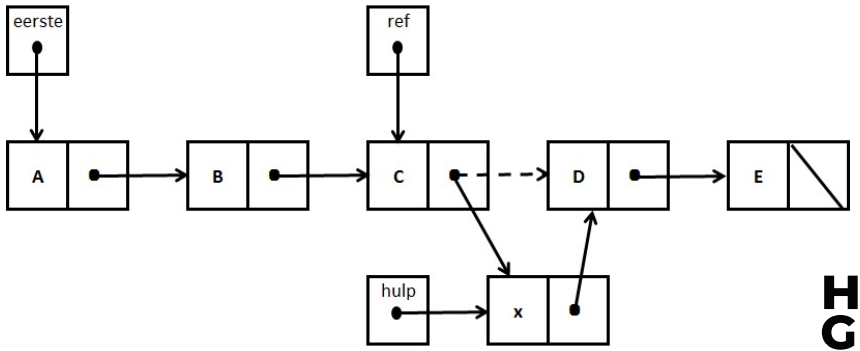
# Vooraan verwijderen

Wat als we nu de eerste knoop willen verwijderen? $\Longrightarrow$ is niet mogelijk, want je kan de voorgaande van de eerste niet meegeven als argument. Dit probleem wordt later aangepakt (zie ankercomponenten).

HO GENT

# Toevoegen van een knoop

De methode VOEGTOE voegt een nieuwe knoop met data-veld *x* toe na de knoop *ref*.




HO GENT

# Toevoegen van een knoop

- Creëer een nieuwe knoop *hulp*;
- Stockeer de waarde ***x*** in het data-veld van de knoop *hulp*;
- Laat *hulp.volgende* verwijzen naar *ref.volgende*;
- Laat *ref.volgende* verwijzen naar *hulp*.

HO
GENT

# Toevoegen van een knoop

**Invoer** de gelinkte lijst bestaat, en *ref* is niet null.
**Uitvoer** na de knoop, waarnaar gerefereerd wordt door de referentie *ref*, werd een nieuwe knoop met data-veld *x* toegevoegd.

```
1: function VOEGTOE(ref, x)
2:     hulp ← nieuwe Knoop( )
3:     hulp.data ← x
4:     hulp.volgende ← ref.volgende
5:     ref.volgende ← hulp
6: end function
```

HO GENT

# Vooraan toevoegen

Wat als we een knoop helemaal vooraan willen toevoegen? $\Longrightarrow$ niet mogelijk, want je kan de voorgaande knoop niet meegeven als argument.

Wat met een lege lijst? $\Longrightarrow$ niet mogelijk

Ook hier kan een *ankercomponent* hulp bieden.

HO GENT

# Ankercomponenten

De methoden VERWIJDER en VOEGTOE uit de voorgaande paragraaf zijn enkel bruikbaar voor algemene gevallen.

- voegToe aan lege lijst? $\Longrightarrow$ implementatie niet bruikbaar
- voegToe vooraan in lijst?
- verwijder eerste knoop?
- Daarom ankercomponenten ingevoerd!

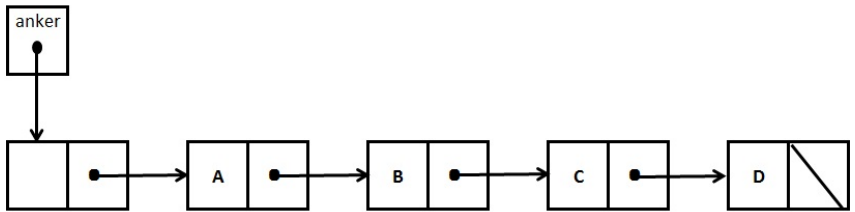
HO GENT

# Ankercomponenten

De ankercomponent is een extra knoop die helemaal vooraan aan de gelinkte lijst wordt toegevoegd. Het data-veld van deze ankercomponent is leeg. Het referentie-veld van de ankercomponent verwijst naar de eerste knoop van de gelinkte lijst. De ankercomponent maakt dus logisch gezien geen deel uit van de eigenlijke lijst maar dient enkel om de implementatie te vereenvoudigen.

HO GENT

# Gelinkte lijst met ankercomponent

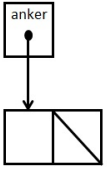



HO GENT

# Ankercomponenten

Nu heeft elke elke knoop dus een voorganger in de lijst. Problemen opgelost!

Een lege gelinkte lijst bestaat nu uit één enkele knoop, nl. de ankercomponent, en de referentie naar deze component. Hieronder een lege gelinkte lijst met ankercomponent.

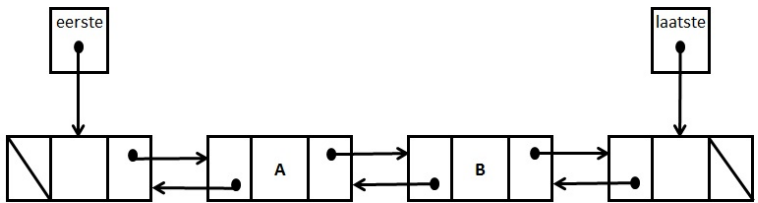



Het gebruik van een ankercomponent maakt controle op speciale situaties overbodig $\Longrightarrow$ vereenvoudigt de basisbewerkingen

HO GENT

# Dubbelgelinkte lijsten

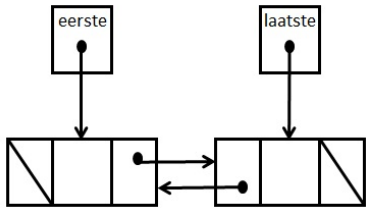
- Eerste knoop vlot bereikbaar
- Wat als je de laatste wil bereiken $\Longrightarrow$ Tijd!
- Oplossing? Een ref *vorige* en *volgende* ...
- Dan ook direct referentie laatste knoop bijhouden? Ja!
- Dit heet een *dubbelgelinkte lijst*




HO GENT

# Ankercomponenten

Een lege dubbelgelinkte lijst met twee ankercomponenten.




Testen of de lijst leeg is, kan als volgt:
als *eerste.volgende = laatste* of als *laatste.vorige = eerste*.

HO GENT

# Stapels

HO
GENT

# Stapel

De naam *stapel* of *stack* is gekozen naar analogie met een stapel boeken.
Indien we een boek van de stapel nodig hebben dan is het best bereikbare boek hetgeen bovenaan ligt. Een boek dat zich op een andere plaats bevindt, is slechts bereikbaar als we alle bovenliggende boeken verwijderd hebben.
Ditzelfde principe wordt toegepast voor het datatype stack:

- Een element dat we willen toevoegen aan een stapel, komt steeds bovenop de reeds bestaande stapel te liggen.
- Enkel het bovenste element van de stapel kan verwijderd worden. Dit element wordt de *top* van de stapel genoemd.

Een stapel is m.a.w. een *LIFO*- of *Last-In-First-Out*-structuur.

HO GENT

# Specificatie

De voorgaande beschrijving legt de structuur van een stapel vast. Voor de implementatie van deze structuur zullen we gebruik maken van een klasse Stack. In deze klasse worden een aantal basisbewerkingen gedefinieerd. Deze zijn:

- *Stack*( ): constructor, maakt een nieuwe stapel aan waarna de stapel bestaat als lege stapel;
- *empty*( ): controleert of een stapel al dan niet leeg is;
- *push*( ): voegt een nieuw element toe bovenaan een stapel, het toegevoegde element wordt de nieuwe top van de 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.

HO GENT

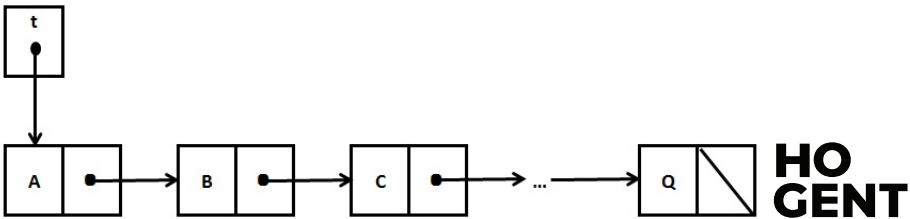
# Specificatie

Vertaling van deze basisbewerkingen nodig naar ondubbelzinnige algoritmen met efficiënte implementatie $\Longrightarrow$ uitvoeringstijd niet afhankelijk van de grootte van de stapel.

HO GENT

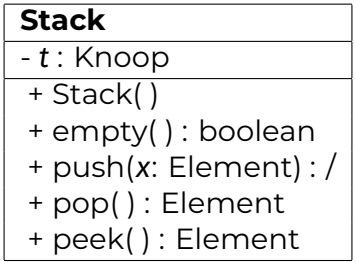
# Implementatie van een Stapel

We tonen nu hoe een stapel kan geïmplementeerd worden als een gelinkte lijst waarbij enkel de top bereikbaar is. Het gebruik van een ankercomponent is hier niet nodig. In het bijzonder wordt een stapel geïmplementeerd door een referentie $t$ naar de eerste knoop in de lijst, die de top van de stapel bevat. De top van de stapel bevat het element $A$. Het element $Q$ werd als eerste element op de stapel geplaatst.




HO GENT

# De klasse Stack in UML




De klasse Knoop wordt als inwendige klasse (inner class) van de klasse Stack geïmplementeerd. Dit betekent dat methodes van de klasse Stack toegang hebben tot de velden van Knoop. De implementatie van de klasse Knoop is zoals voorheen.

HO GENT

# De Stack constructor

Een lege stapel bevat nog geen elementen, m.a.w. de referentie ***t*** is *null*.

**Invoer** /

**Uitvoer** er werd een nieuwe stapel aangemaakt, deze stapel bestaat als lege stapel.

```
1: function STACK
2:     t ← null
3: end function
```

HO GENT

# De methode empty()

De methode EMPTY controleert of een stapel al dan niet leeg is. Het resultaat van de methode is een boolean.

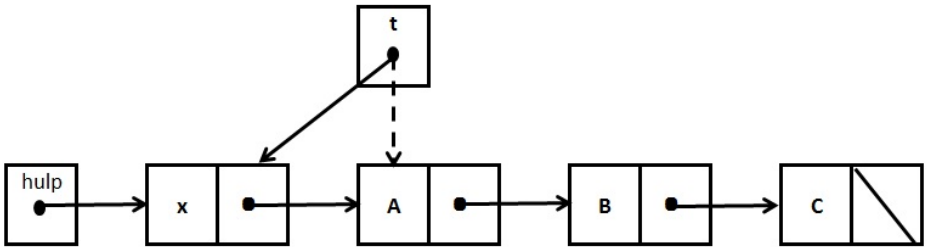
**Invoer** de stapel *s* bestaat

**Uitvoer** de waarde true of false werd afgeleverd, afhankelijk van het feit of de stapel *s* leeg is of niet.

```
1: function EMPTY
2:     return t = null                    # vergelijking, geen assignatie
3: end function
```

HO GENT

# Toevoegen van een element




HO
GENT

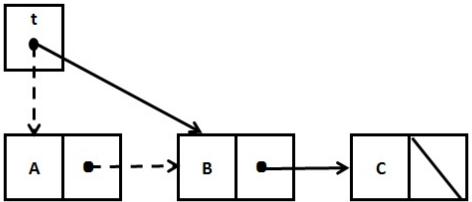
# Toevoegen van een element

**Invoer** de stapel *s* bestaat, *x* moet op de stapel worden geplaatst
**Uitvoer** het element *x* werd als top-element op de stapel *s* geplaatst.

```
1: function PUSH(x)
2:     hulp ← nieuwe Knoop( )
3:     hulp.data ← x
4:     hulp.volgende ← t
5:     t ← hulp
6: end function
```

HO GENT

# Verwijderen van een element




HO
GENT

# Verwijderen van een element

**Invoer** de stapel *s* bestaat en is niet leeg
**Uitvoer** de top werd verwijderd en de waarde van de top werd geretourneerd.

```
1: function POP
2:     x ← t.data
3:     t ← t.volgende
4:     return x
5: end function
```

HO GENT

# Retourneren van de waarde van de top van een stapel

Er wordt opnieuw verondersteld dat de stapel niet leeg is.

**Invoer** de stapel *s* bestaat en is niet leeg
**Uitvoer** de waarde van de top werd geretourneerd, *s* werd niet gewijzigd.

```
1: function PEEK
2:     return t.data
3: end function
```

HO GENT

# Controleren van haakjes

Elke programmeur kent het verschijnsel dat een programma niet werkt omdat ergens een haakje ontbreekt. Compilers controleren steeds of de haakjes allemaal correct zijn en genereren een foutboodschap wanneer dit niet het geval is.

We tonen nu aan hoe een stapel kan gebruikt worden om te controleren of de haakjes correct zijn. Het basisidee is dat men voor deze controle enkel het laatste openende haakje nodig heeft.

HO
GENT

# Controleren van haakjes

**Basisidee**: Voor de controle van de haakjes worden twee zaken onderzocht:

1. Er wordt gecontroleerd of elk openend haakje juist één overeenkomstig sluitend haakje heeft, bv. voor elk ‘(’-symbool moeten we verder in het programma één ‘)’-symbool terugvinden. dit geldt ook voor: [ en ], { en }.
2. De volgorderegels moeten in acht genomen worden. De haakjes mogen genest zijn, dit wil zeggen dat een paar haakjes zich tussen een ander paar mag bevinden, maar paren haakjes mogen elkaar niet overlappen.
   Bijvoorbeeld, ([ ]) is een geldige volgorde van symbolen, terwijl ([ )] niet geldig is.

We willen met behulp van een stapel een algoritme ontwerpen dat de vereffening van symbolen verifieert.

HO GENT

# Controleren van haakjes

Dit kan als volgt:

- Maak een lege stapel aan.
- Doorloop alle symbolen één voor één. Indien het symbool een haakje is, moet er een opdracht uitgevoerd worden:
  - als het een open-symbool is, plaats het dan op de stapel (*push*-bewerking);
  - als het een sluit-symbool is onderscheiden we twee mogelijke situaties:
    - de stapel is leeg: er wordt een foutmelding gegenereerd aangezien er geen corresponderend open-symbool aan is vooraf gegaan;
    - de stapel is niet leeg: het top-element wordt er afgehaald (*pop*-bewerking). Dit symbool wordt vergeleken met het net ingelezen symbool, als beide symbolen niet corresponderen wordt een fout gemeld.
- Alle karakters ingelezen en stapel niet leeg? Foutmelding: er zijn nog open-symbolen zijn waarvoor geen sluit-symbool is gevonden.

HO GENT

# Controleren van haakjes: algoritme

De pseudocode om te controleren of het om een openend haakje gaat en om na te gaan of haakjes van dezelfde soort zijn wordt niet getoond omdat deze niet tot de essentie van het algoritme behoren.

HO GENT

# Controleren van haakjes: algoritme

**Invoer** *uitdrukking* is een array van Strings met de tokens van een uitdrukking waarin eventueel haakjes voorkomen.

**Uitvoer** indien alle open haakjes correct worden afgesloten werd er geen foutmelding gegenereerd. In het andere geval wordt er een boodschap getoond op het scherm

HO GENT

# Pseudocode controle haakjes

```
1:  function CONTROLEERHAAKJES(uitdrukking)
2:      s ← STACK()
3:      for i = 0 ... uitdrukking.lengte – 1 do
4:          symbool ← uitdrukking[i]
5:          if symbool is openend haakje then
6:              s.PUSH(symbool)
7:          else
8:              if symbool is sluitend haakje then
9:                  if s.EMPTY() then
10:                     PRINT("Te veel sluit symbolen")
11:                 else
12:                     voorgaand ← s.POP()
13:                     if symbool en voorgaand niet corresponderend then
14:                         PRINT( "Fout symbool:", symbool)
15:     if not s.EMPTY() then
16:         PRINT("Te veel open symbolen." )
17: end function
```

HO GENT

# Infix en postfix

- In de wiskunde worden in een rekenkundige uitdrukking de binaire operatoren zoals + en × normaalgezien tussen de operanden geschreven:

  1 + 2 × 3.

- Hierbij staan de operatoren + en × *tussen* de operanden 1, 2 en 3. Dit heet *infix-notatie*.
- Vertaling door compiler naar machine-code niet rechtstreeks mogelijk
- Oplossing: *postfix-notatie*
- Andere naam: *RPN* (Reverse Polish Notation).
- vb. Sommige HP rekenmachines operatoren *na* hun operanden:

  3 4 × .

HO GENT

# Infix en postfix

De binaire vermenigvuldigingsoperator staat na zijn operanden en werkt in op de twee argumenten ervoor nl. 3 en 4. De betekenis van deze postfix-uitdrukking is dus $3 \times 4$ en de waarde ervan is 12.

Nog een voorbeeld:

$$1\ 2\ 3\ \times\ +$$

Komt overeen met

$$1\ 6\ +$$

wat evalueert naar 7.

HO GENT

# Infix: Prioriteiten

Verder moet bij het verwerken van rekenkundige uitdrukkingen eveneens rekening gehouden worden met de prioriteitsregels. Zo moet de vermenigvuldiging uitgevoerd worden vóór de optelling, tenzij haakjes anders aangeven. Ook dit moet correct verwerkt worden door de compiler. De waarde van

$$1 + 2 \times 3$$

is m.a.w. 7 en niet 9. Om de waarde 9 te bekomen schrijft men

$$(1 + 2) \times 3.$$

Hierbij werden haakjes gebruikt die aangeven dat de optelling eerst moet worden uitgevoerd.

HO GENT

# Waardebepaling van een rekenkundige uitdrukking

Standaard wordt er gewerkt volgens de volgende afspraken:

- de operatoren × en / hebben een hogere prioriteit dan + en −;
- prioriteiten kunnen door gebruik van haakjes (in de meeste programmeertalen zijn alleen ronde haakjes hiervoor toegelaten) worden aangepast;
- de prioriteit van × en / is gelijk. Als ze beide in een uitdrukking staan, worden ze uitgevoerd in de volgorde waarin je ze tegenkomt, dus van links naar rechts. Hetzelfde geldt voor + en −.

HO GENT

# Haakjes

Tabel: Infix-notatie versus postfix-notatie. Haakjes zijn overbodig in de postfix-notatie.

| Infix-notatie | Postfix-notatie |
|---|---|
| 2 × 3 | 2 3 × |
| 2 × 3 + 5 = (2 × 3) + 5 | 2 3 × 5 + |
| 2 × 3 + 5/7 | 2 3 × 5 7 / + |
| 2 × 3/5 × 7 | 2 3 × 5 / 7 × |
| 2 + 3 × 5 + 7 | 2 3 5 × + 7 + |
| (2 + 3) × (5 + 7) | 2 3 + 5 7 + × |
| (2 + (3 − 5) × 4)/((6 − 7) × 9) | 2 3 5 − 4 × + 6 7 − 9 × / |

HO GENT

# Prioriteitsregels

Voor de evaluatie van een postfix-uitdrukking hoeven eveneens geen prioriteitsregels meer in acht genomen te worden. De uitdrukking wordt steeds eenvoudigweg van links naar rechts doorlopen. Wanneer een binaire operator ontmoet wordt, wordt deze uitgevoerd op de beide voorgaande operanden. In de postfix-uitdrukking wordt vervolgens de operator samen met zijn operanden vervangen door het resultaat van de bewerking.

HO GENT

# Postfix: waardebepaling

Een machine kan de waarde van een postfix-uitdrukking gemakkelijk bepalen met behulp van een stapel:

- De uitdrukking wordt van links naar rechts doorlopen.
- Als een operand (i.e. een getal) wordt ontmoet dan wordt dit op de stapel geplaatst.
- Als een (binaire) operator wordt ontmoet dan worden de twee laatst gestapelde operanden van de stapel gehaald en de bewerking met die twee operanden wordt uitgevoerd. Het resultaat van de bewerking wordt op de stapel geplaatst.
- Deze stappen worden herhaald totdat de volledige uitdrukking is doorlopen.
- Als de volledige uitdrukking is doorlopen bestaat de stapel nog slechts uit één element, nl. de waarde van de postfix-uitdrukking.

HO GENT

# Uitvoeringstijd

Dit algoritme is uitvoerbaar in lineaire tijd, aangezien voor elk symbool uit de uitdrukking maximaal een constant aantal stapel-bewerkingen moet uitgevoerd worden, waarbij elke stapel-bewerking constante tijd vraagt. Dit wordt uitgewerkt in de oefeningen

Toch worden in de meeste programmeertalen rekenkundige uitdrukkingen in infix-notatie geschreven omdat dit voor de programmeur handiger is. Gelukkig bestaat er een eenvoudig algoritme om een infix-uitdrukking om te zetten naar een equivalente postfix-uitdrukking.

HO
GENT

# Van infix naar postfix

De conversie van een (gewone) infix-uitdrukking als 3 + 4 × 5 naar zijn overeenkomstige postfix-uitdrukking 3 4 5 × + kan eveneens met behulp van een stapel gebeuren.
Uit de voorbeelden van Tabel 1 leren we dat in een postfix-uitdrukking de operanden steeds in precies dezelfde volgorde staan als in de equivalente infix-uitdrukking. De volgorde van de operatoren kan echter wijzigen.

HO GENT

# Van infix naar postfix

1. Lees de invoertekst van links naar rechts.
2. Operand? $\Longrightarrow$ schrijf deze rechtstreeks naar de uitvoer
3. Operator of haakje $\Longrightarrow$ tijdelijk op de stapel als:
   - de stapel leeg is;
   - de gelezen operator hogere prioriteit dan de operator bovenaan de stapel;
   - openend haakje (fungeert verder als operator met de laagste prioriteit)
4. Operator heeft gelijke of lagere prioriteit dan die op de top van de stapel
   - Haal alle operatoren van de stapel met gelijke of hogere prioriteit en voeg toe aan de uitvoer;
   - Hierna is de stapel leeg of is de top een operator met lagere prioriteit;
   - Vervolgens wordt de ingelezen operator op de stapel geplaatst;
5. Sluitend haakje wordt ingelezen
   - Haal alle operatoren van de stapel en voeg toe aan de uitvoer totdat een openend haakje wordt bereikt;
   - Het haakje wordt eveneens van de stapel gehaald maar niet aan de uitvoer toegevoegd;

HO GENT

# Van infix naar postfix: Tijdscomplexiteit

Deze methode voor de conversie van infix-notatie naar postfix heeft een uitvoeringstijd die lineair is in $n$, met $n$ de lengte van de uitdrukking.

HO GENT

# Oefeningen

1. Breid de klasse GelinkteLijst uit met een methode *size*. De methode heeft als resultaat het aantal elementen van een gelinkte lijst.
2. 2.1 Pas in de klasse GelinkteLijst de constructor GelinkteLijst aan zodat de methode een lege gelinkte lijst met ankercomponent aanmaakt.
   2.2 Breid deze klasse uit met een methode invert. Deze methode heeft als resultaat een gelinkte lijst $l_2$. De gelinkte lijst $l_2$ bevat dezelfde elementen als een bestaande gelinkte lijst $l_1$ maar de elementen komen voor in omgekeerde volgorde. De gelinkte lijst $l_1$ is na afloop ongewijzigd.

HO GENT

# Oefening 3

Pas de klasse Knoop aan zodat met de objecten van deze klasse een dubbelgelinkte lijst kan aangemaakt worden. Noem de nieuwe klasse KnoopDubbel.
Schrijf vervolgens voor dubbelgelinkte lijsten met twee ankercomponenten de basisfuncties van de klasse DubbelGelinkteLijst in pseudocode uit.

| **DubbelGelinkteLijst** |
| --- |
| - *eerste* : KnoopDubbel<br>- *laatste*: KnoopDubbel |
| + DubbelGelinkteLijst( )<br>+ verwijder(*ref*: KnoopDubbel) : Element<br>+ voegToeVoor(*ref*: KnoopDubbel, *x*: Element) : /<br>+ voegToeNa(*ref*: KnoopDubbel, *x*: Element) : /<br>+ zoek(*x*: Element) : *ref*: KnoopDubbel |

De klasse KnoopDubbel wordt als inwendige klasse (inner class) van de klasse DubbelGelinkteLijst geïmplementeerd.

HO GENT

# Oefening 4

Een wachtrij is een FIFO-datastructuur. Het element dat het eerst op de wachtrij werd geplaatst is ook het eerste dat wordt verwijderd. Een wachtrij heeft m.a.w. twee uiteinden: een *kop* en een *staart*. Elementen worden toegevoegd aan de kant van de staart en verwijderd aan de kant van de kop.

HO
GENT

# Oefening 4. Deel 1

Een wachtrij kan geïmplementeerd worden door twee referenties ***k*** en ***s***. De knoop ***k*** refereert naar de kop van de wachtrij, de knoop ***s*** refereert naar de staart van de wachtrij.
Herschrijf alle basisbewerkingen voor een wachtrij corresponderend met deze implementatie. Deze zijn

- *Queue*( ): constructor, maakt een nieuwe wachtrij aan waarna de wachtrij bestaat als lege wachtrij;
- *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 in een wachtrij en retourneert het verwijderde element;
- *front*( ): retourneert het voorste element, m.a.w. de kop, van een wachtrij, zonder het te verwijderen.

HO GENT

# Oefening 4. Deel 2

Breid deze klasse uit met een methode *invert*. Deze methode plaatst de elementen van een bestaande wachtrij in omgekeerde volgorde.
Je algoritme mag geen nieuwe knopen alloceren. Je code mag tevens het veld ‘data’ van geen enkele knoop wijzigen.

HO
GENT

# Oefening 5

Implementeer de methodes om

- een postfix uitdrukking te evalueren
- een infix uitdrukking om te zetten naar een postfix uitdrukking
- een infix uitdrukking te evalueren. Dit is dan een soort van eenvoudige rekenmachine.

HO GENT