Informatica

Hoofdstuk 0 : Inleiding

1. De rol van algoritmen

Algoritme = reeks stappen die bepaalt hoe een taak wordt uitgevoerd

- ⇨ *Recepten voor koken; richtingsborgen voor weg;....*

Programma = een algoritme in een vorm waarmee een machine kan werken
Programmeren = maken, structureren en aanleren programma's
Software = programma's en de algoritmen die ze vertegenwoordigen
Hardware = apparatuur zelf

Kurt Gödel zijn onvolledigheidstheorema *die stelt dat in alle wiskundige theorieën van ons traditioneel wiskundig systeem er stellingen zijn waarvan correctheid of onjuistheid niet bewezen kan worden.*

2. De oorsprong van rekenmachines

- Telraam is een gegevensopslagsysteem waarbij het enkel in combinatie met een mens een complete rekenmachine wordt.
- Rekenmachines gebaseerd op werking van tandwielen

"Eerste programmeur ter wereld = Augusta Ada Byron en haar addendum"

-Ponskaarten waarbij informatie wordt voorgesteld als gaatjes
-Opkomst elektronica en gebruik van o.a. vacuümbuistechnologie
-Uitvinding transistor en geïntegreerde schakelingen

Verschilmachine van Babbage			
x	x^2	Eerste verschil	Tweede verschil
0	0		
1	1	1	2
2	4	3	2
3	9	5	2

3. Abstractie

Informatica = houdt zich bezig met algoritmen heeft erg breed studiegebied
Abstractie = vereenvoudigingstechniek waar men onderscheid tussen de externe eigenschappen v/e entiteit en de details v/d inwendige samenstelling ervan maakt
Men gebruikt abstractie om details te negeren en ze te beschouwen als enkelvoudige entiteit

Hoofdstuk 1 : Gegevensopslag

1. Bits en hoe ze worden opgeslagen

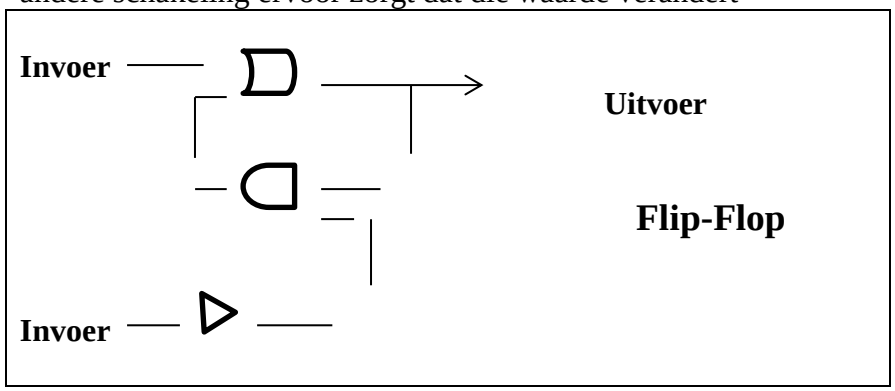
Bits = reeksen van nullen en eenen dat afbeeldingen, getallen, letters,...kunnen voorstellen

1.1 Boolean bewerkingen

Bewerkingen die rekenen met WAAR/NIET WAAR-waarden.
AND-bewerking = beiden moeten waar zijn
OR-bewerking = één moet maar waar zijn
XOR-bewerking = er mag maar één waar zijn
NOT-bewerking = 1 statement als invoeren en uitvoer is omgekeerde van invoer

1.2 Gates en flip-flops

Gate = apparaat dat uitvoer v/e Boleaan bewerking produceert oop basis van invoerwaarden
Flip-flop = schakeling die als uitvoer een waarde 0 of 1 die constant blijft totdat een puls v/e andere schakeling ervoor zorgt dat die waarde verandert



Betekenis van een flip-flop is dat ze uitermate geschikt is om een bit in een computer op te slaan.

1.3 Andere opslagtechnieken

Ringkerngeheugens = kleine ringen van magnetisch materiaal slaan bits op in computer
Condensator = twee evenwijdige metalen plaatjes met een smalle spleet ertussen en kan voorkomen in geladen en ongeladen toestand
Chip = combinatie van miljoenen condensatortjes
Read only memory (ROM)
Flasgeheugen waarbij silliciumdioxide dient als opslagmateriaal

Moderne technologieën kunnen bits opslaan in apparaten met een diameter < Angstroms

1.4 Hexadecimale notatie

Stream = lange reeksen bits

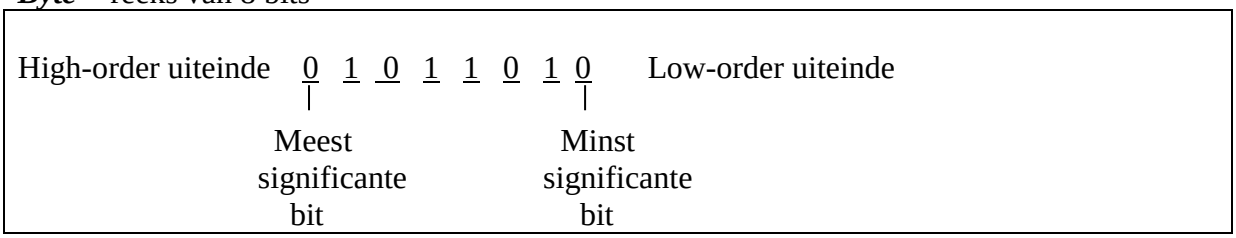
Bitpatroon	Hexadecimale representatie
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

2. Werkgeheugen

Werkgeheugen = alle schakelingen samen die elke 1 bit kunnen opslaan

2.1 Structuur v/h geheugen

Byte = reeks van 8 bits



Geheugenadres = unieke naam van elke cel in werkgeheugen ter onderscheid
Random acces geheugen (RAM) = cellen in geheugen kunnen willekeurig benaderd worden
Dynamic Random acces Memory (DRAM) = RAM met dynamische geheugentechnologie

2.2 De omvang v/h geheugen bepalen

Cellen van werkgeheugensystemen worden vaak uitgedrukt in machten van 2.
Kibibyte = 1024 bytes
Mebibytes = 1.048.576 bytes
Gibibytes = 1.073.741.824 bytes

3. Massageheugen

Massageheugensystemen = extra geheugenapparatuur omdat werkgeheugen te vluchtig is
Online = apparaat of informatie verbonden met de machine
Offline = menselijk ingrijpen is nodig om machine te activeren

Nadeel is het feit dat massageheugensystemen meestal meer tijd nodig hebben om gegevens op te slaan of op te halen.

3.1 Magnetische schijven

Magneetschijf = gegevens worden opgeslagen op een roterende magnetische schijf die voorzien is v/e coating
Track = cirkel van de schijf

"Bij een magneetschijf worden deze tracks bestreken door lees/schrijfkoppen"

Sectoren = elke track wordt verdeeld in cirkelsegmenten waarbij informatie als een ononderbroken reeks bits wordt opgeslagen

"Elke track bevat hetzelfde aantal sectoren en elke sector bevat hetzelfde aantal bits."
➔ **Tracks dichter bij kern hebben dus compactere opgeslagen bits!**

Formatteren = magnetisch aanbrengen van sporen en sectoren
Head crash = wanneer de lees/schrijfkop de schijf raakt (door een stofdeeltje)

"*Capaciteit v/e schijfgeheugensysteem is afhankelijk v/h gebruikte aantal schijven en de dichtheid waarmee de tracks en sectoren aangebracht zijn."*

Prestaties v/e schijfsysteem kunnen worden beoordeeld als volgt:

-Zoektijd (kop van ene track naar andere)	**-Rotatievertraging**
-Toegangstijd (som zoektijd en rotatievertraging)	**-Overdrachtssnelheid** (tijd waarin gegevens van of naar de schijf verplaatst kunnen worden)

"Harde-schijfsystemen leveren over het algemeen aanmerkelijk betere prestaties dan diskettesysstemen."
➔**Harde-schijfsystemen kunnen duizenden keren wentelen!**

3.2 Compact Discs

Compact disc = schijf met diameter van 12 cm en vervaardigd uit reflecterend materiaal met transparante beschermende coating

➔ **Informatie wordt opgeslagen door variaties in het reflecterend vlak aan te brengen**
➔ **Ze wordt opgehaald met een laserstraal die onregelmatigheden detecteert**
➔ **De informatie wordt opgeslagen op 1 spiraal die loopt van de kern tot de rand**
➔ **Er wordt meer informatie opgeslagen per omwenteling aan de buitenrand**
➔ **Cd-systemen presteren optimaal bij lange continue reeksen gegevens**

3.3 Magneetband

Informatie wordt opgeslagen in de magnetische coating v/e dunne plastic tape die om een spoel wordt gewonden.
Moderne tapesystemen delen de tape op in segmenten die dan nog eens verschillende tracks bevatten!

Nadeel is dat het verplaatsen tussen verschillende posities enorm tijdrovend is!
Voordeel is wel bij offline opslag door de grote capaciteit, betrouwbaarheid en lage kosten!

3.4 Bestanden opslaan en ophalen

Bestanden = grote eenheden informatie die tekst, foto's,... kunnen bevatten
Fysiek record = een gegevensblok dat voldoet aan de fysieke kenmerken v/e opslagapparaat
Logisch record = natuurlijke blokken binnen een bestand ter onderverdeling

- ⇨ Deze kan vervolgens uit '***velden'*** bestaan ook ter verdere onderverdeling
- ⇨ *Verschillende logische records kunnen zich bevinden op 1 fysiek record*
- ⇨ *Maar ook 1 logisch record kan meerdere fysieke records beslaan*

Buffer = stuk werkgeheugen waarin verschillende fysieke records worden gegroepeerd

Willekeurige toegang tot gegevens ➔ verschillend:

- **Bij werkgeheugen** = snelle toegang tot afzonderlijke bytes mogelijk door adresseringssysteem
- **Bij magnetische schijven** = alleen willekeurige toegang mogelijk op sectorniveau
- **Bij compact discs** = willekeurige toegang ook mogelijk op sectorniveau maar grotere vertragingen
- **Bij magneetband** = willekeurige toegang ongewenst!

4. Informatie representeren in de vorm van bitpatronen

4.1 Tekst

Tekst wordt voorgesteld door een code waarin alle verschillende symbolen een uniek bitpatroon hebben.

American Standard Code for Information Interchange (ASCII)

- ⇨ 8-bit-per-symboolindeling
- ⇨ 01001000 01100001 01101100 01101100 01101111 00101110
- ⇨ H a l l o .

Unicode

- ⇨ 16 bits voor elk symbool waardoor ruimte voor 65.536 verschillende patronen

International Standards Organisation

- ⇨ 32 bits voor elk symbool waardoor ruimte voor miljoenen patronen

Teksbestand = een bestand dat alleen een lange reeks in ASCII of Unicode gecodeerde symbolen bevat

- ⇨ *Onderscheid tussen teksbestanden gemaakt met behulp van editor, en bestanden die gemaakt zijn met tekstverwerkingsprogramma (bevat veel extra codes)*

4.2 Numerieke waarden (verder bij 1.6)

Getallen opslaan als gecodeerde tekens is inefficiënt en daarom gebruiken we de ***binaire notatie!***
Floating-pointnotatie gebruikt voor breuken!

4.3 Afbeeldingen

Twee technieken voor het coderen van afbeeldingen:

- ⇨ **Bitmaptechnieken en vectortechnieken**

Bitmaptechnieken beschouwen afbeeldingen als een verzameling puntjes, genaamd ***pixels***.

- ⇨ *Elke bit is ofwel 1 of 0, afhankelijk van of de bijbehorende pixel zwart of wit is*
- ⇨ *Bij kleurenafbeeldingen wordt elke pixel voorgesteld door een combinatie van bits*

Nadeel van bitmaptechnieken is dat een afbeelding niet eenvoudig vergroot of verkleind kan worden!

4.4 Geluid

Audio-informatie wordt gecodeerd door de amplitude v/d geluidsgolf met regelmatige tussenpozen te meten, waarna ze kunnen worden opgeslaan.

- ⇨ *Huidige muziekcd's hebben een samplefrequentie van 44.100 samples/seconde*

5. Het binaire systeem

5.1 Binaire notatie

1	**0**	**1**	**1**
1 x 2^3	**0 x 2^2**	**1 x 2^1**	**1 x 2^0**

= 11

Bepalen van de binaire represenatie van het getal 13:

13 | 2	6 | 2	3 | 2	1 | 2
-12 | 6	-6 | 3	-2 | 1	-0 | 0
1	0	1	1

13 = 1 0 1 1

5.2 Binair optellen

0	**1**	**0**	**1**
+0	**+0**	**+1**	**+1**
0	**1**	**1**	**10**

5.3 Binaire breuken

In de binaire notatie gebruiken we voor breuken een **radixpunt**!

Ook hier gelden de regels van binair optellen!

1	**0**	**1**	**.**	**1**	**0**	**1**
1 x 2^2	**0 x 2^1**	**1 x 2^0**		**1 x 2^{-1}**	**0 x 2^{-2}**	**1 x 2^{-3}**

= $5^{5/8}$

6. Integers opslaan

6.1 De 2-complementnotatie

In huidige apparatuur worden waarden meestal voorgesteld door een patroon van 32 bits.

Bitpatroon	Gerepresenteerde waarde	Bitpatroon	Gerepresenteerde waarde
011	3	0111	7
010	2	0110	6
001	1	0101	5
000	0	0100	4
111	-1	0011	3
110	-2	0010	2
101	-3	0001	1
100	-4	0000	0
		1111	-1
		1110	-2
		1101	-3
		1100	-4
		1011	-5
		1010	-6
		1001	-7
		1000	-8

Tekenbit = is het meest linker bit die aanduidt of het getal negatief of positief is

⇨ 1 staat voor negatief en 0 voor positief

Complement = men telt vanaf rechts en van het moment men een 1 tegenkomt neemt men telkens de tegengestelde waarbij men dan het complement vindt!

⇨ **1 0 1 0 = -6 ➔ 0 1 1 0 = +6**

6.2 Optellen in de 2-complementnotatie

3 **+ 2**	➔	**0011** **+ 0010** **0101**	➔	**5**
- 3 **+ - 2**	➔	**1101** **+1110** **1011**	➔	**-5**
7 **+ -2**	➔	**0111** **+ 1011** **0010**	➔	**2**

⇨ *Bij het optellen van 1000 en 1000 is de uitkomst 0000 en gaat 1 verloren*

6.3 Het probleem v/d overflow

Overflow = het probleem dat ontstaat wanneer het voor te stellen getal buiten het bereik getallen valt dat weergegeven kan worden

➔ *Bij 2-complementnotatie mogelijk wanneer men twee negatieve/positieve getallen optelt*

6.4 Excess-notatie

Bitpatroon	Gerepresenteerde waarde
1111	7
1110	6
1101	5
1100	4
1011	3
1010	2
1001	1
1000	0
0111	-1
0110	-2
0101	-3
0100	-4
0011	-5
0010	-6
0001	-7
0000	-8

Het verschil met een 2-complementsysteem is dat de tekenbits omgekeerd zijn.
Om de positieve getallen te tellen, tel je normaal en voor de negatieve getallen gebruik je gewoon de complementmethode

➔ **Excess-8-notatie!!**

7. Breuken opslaan

7.1 Floating-pointnotatie

Een tekenbit 1 betekent dat het getal negatief is en een 0 dat het getal positief is!
De 3 resterende bits aan de linkerkant v/h radixpunt noemen we het ***exponentveld*** en de 4 resterende bits aan de rechterkant v/h radixpunt noemen we het ***mantisseveld***.

....	....	.....	.....	•	....	.....	.....	.....	➔**Bitposities**
Tekenbit	**Exponent**				**Mantisse**				
0	1	1	0		1	0	1	1	

-De 0 vertegenwoordigt het teken van het getal, hier dus +
-Vervolgens pakt men de exponent en past men toe in het excess-notatiesysteem
-Volgens dit systeem bekomt men nu 2!
-Dit wil zeggen dat men bij .1011 het radixpunt 2 plaatsen naar rechts verschuift
-Indien het negatief was dan ging het links geweest zijn
Men bekomt zo 10.11 wat gelijk is aan $2^{3/4}$

➔**Excess-notatiesysteem die men gebruikt om radixpunt te plaatsen!**

Bitpatroon	Gerepresenteerde waarde
111	3
110	2
101	1
100	0
011	-1
010	-2
001	-3
000	-4

Om een getal te coderen gaat men zo te werk:
-Neem bv. het getal 3/8 dat binair .011 is
-Het is positief dus meest linkse bit zal 0 zijn
-Men neemt de meest linkse 1 van .011 en zet die direct na het radixpunt ➔ *.1100*
-Het radixpunt moet dus naar links verschoven worden met 1 en dus is exponent -1
-Omgerekend is die -1 = 011 en wordt ***uiteindelijke notatie 00111100***
➔ **Elke mantisse begint met een 1 hierdoor ➔ Genormaliseerde vorm!**

7.2 Afrondingsfouten

Bij sommige getallen kan men te maken hebben met een ***afrondingsfout***!
Stel bv $2\ ^{5}/_{8}$ dat door 10.101 wordt voorgesteld. In de mantisse zal die 10.101 worden overgenomen als .1010 wat uiteindelijk $2\ ^{1}/_{2}$ is!
Ook bij repeterende breuken is een afrondingsfout zeer frequent!

Een algemene regel voor het optellen is dat wanneer je verschillende getallen moet optellen, je eerst de kleinste optelt!
Bv. $\mathbf{2\ ^{1}/_{2} + {}^{1}/_{8} + {}^{1}/_{8}}$ ➔ Indien men eerst de $^{1}/_{8}$ er bij telt dan bekomt men bij afronding weer $2\ ^{1}/_{2}$! Indien men echter $2\ ^{1}/_{2} + {}^{2}/_{8}$ optelt dan bekomt men $2\ ^{3}/_{4}$ wat juist is

8. Gegevenscompressie

Gegevenscompressie = omvang van gegevens verkleinen
Huffman-codes = frequentie-afhankelijke codes die steunen op logaritme van David Huffman
Lempel-Ziv-Encoding = systemen hierop gebaseerd zijn meer algemene systemen
➔ Voorbeeld van **adaptive dictionary encoding**

9. Communicatiefouten

9.1 Pariteitsbits

Eenvoudige methode om fouten te detecteren is gebaseerd op het principe dat als elk bewerkt bitpatroon een oneven aantal enen heeft, een patroon met even aantal enen een fout moet bevatten
Hiervoor is een systeem nodig dat ervoor zorgt dat elk patroon een oneven aantal enen heeft!
➔ Hiervoor voegt men een extra bit toe langs links ; het ***pariteitsbit***

Oneven pariteit = in dit systeem wijst een patroon met een even aantal enen op een fout
Even pariteit = elk patroon heeft een even aantal enen en een patroon met oneven is dus fout

Checkbyte = verzameling pariteitsbits die lange bitpatronen vergezellen
➔ *Elke bit binnen de checkbyte is een pariteitsbit die hoort bij een bepaalde verzameling*

9.2 Foutencorrigerende codes

Pariteitsbits kan men niet gebruiken om codes te corrigeren!!
➔ **Een foutencorrigerende code!**

SYmbool	Code
A	000000
B	001111
C	010011
D	011100
E	100110
F	101001
G	110101
H	111010

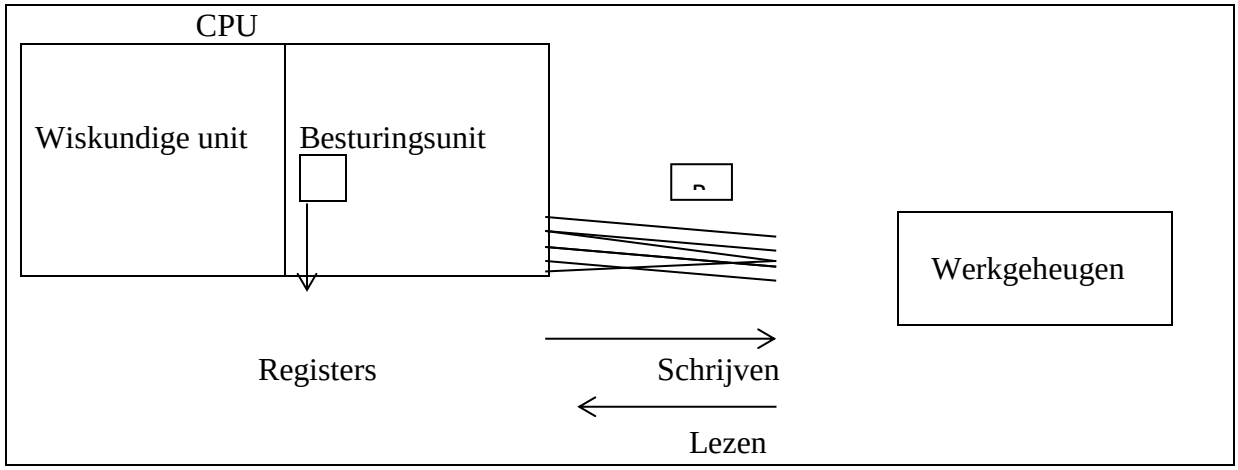
Eerst moet men de **Hamming-afstand** berekenen tussen 2 patronen!
➔ *De mate waarin de patronen verschillende bytes hebben!*
➔ *Tussen A en B is de afstand 4!*
➔ *Belangrijkste is dat er tenminste een afstand is van 3 of anders is er een fout!*
➔ ***VB veronderstel dat patroon 010100 ontvangen is! Deze verschilt 1 afstand met D en dus is de conclusie dat het oorspronkelijk een D moest zijn!***
➔ *Maximaal 2 fouten kunnen gevonden worden per patroon en 1 fouten gecorrigeerd!*

Hoofdstuk 2 : Gegevensbewerking

1. Computerarchitectuur

Central Processing Unit = geïsoleerde schakelingen die bewerkingen met gegevens uitvoeren

- *Gaat meestal schuil onder een koellichaam en ventilator*
- Cpu bestaat uit 2 delen :
  - ***Wiskundige/logica-unit*** *= bevat schakelingen voor gegevensbewerking*
  - ***Besturingsunit =*** *bevat schakelingen voor het coördineren van activiteiten*
- Voor tijdelijk opslaan van informatie bevat de CPU cellen (=***registers***) die vergelijkbaar zijn met die van het werkgeheugen
  - ***Algemene registers*** = tijdelijke opslagplaatsen voor gegevens die door de CPU bewerkt worden
  - ***Speciale registers*** = zie 2.3
- Een ***bus*** verbindt CPU met het werkgeheugen om bitpatronen te kunnen verplaatsen
  - *Controlebus*
  - *Adresbus*
  - *Databus*



Cachegeheugen = stuk uiterst snel geheugen in de CPU dat probeert kopieën v/h werkgeheugen op te slaan waardoor constante uitwisseling tussen registers en werkgeheugen wordt ontweken

***Een belangrijke doorbraak was de realisatie dat een programma gecodeerd en opgeslagen kon worden in het werkgeheugen!* = Stored-programconcept**

➔ *Vroeger waren programma's een onderdeel v/d besturingsunit*

2. Machinetaal

Machinetaal = verzameling instructies die gelezen kunnen worden door CPU's + coderingssyteem ➔ zo'n instructie heet een ***machine-instructie***

2.1 Instructierepertoire

Er zijn twee CPU-ontwerpfilosofieën:

- CPU moet een zo klein mogelijk aantal machine-instructies kunnen uitvoeren
  - ***Reduced Instruction Set Computer***
  - Ze zijn efficiënt en snel
- CPU moet groot aantal complexe instructies kunnen uitvoeren
  - ***Complex Instruction Set Computer***
  - Complexe CPU makkelijker programmeerbaar

Machine-instructies kunnen worden ingedeeld in 3 categorieën:

➔ **De gegevensopdrachtgroep; de wiskundige groep en de besturingsgroep**

2.2 Gegevensoverdracht

Gegevensoverdrachtgroep bestaat uit instructies waarmee gegevens v/d ene locatie naar een andere worden verplaatst (=***kopiëren*** is correcter)
LOAD-instructie = verzoek om een algemeen register te vullen met inhoud geheugencel
STORE-instructie = verzoek om inhoud register te verplaatsen naar een geheugencel
Input/Output-instructies = instructies die de opdrachten uitvoeren om te communiceren met externe apparaten zoals printers, toetsenborden,...

2.3 Wiskundige/logicagroep

Bestaat uit instructies die de besturingsunit vertellen dat deze een verzoek voor een activiteit binnen de wiskundige/logica-unit moet doen.
➔ Voorbeelden zijn AND, OR, XOR,..

Verzameling bewerkingen waarmee de inhoud van registers naar rechts of links worden verplaatst binnen het register ➔
- ***SHIFT-bewerkingen*** = bits gaan verloren aan het uiteinde v/h register
- ***ROTATE-bewerkingen*** =bits worden gebruikt om ontstane gat aan het andere eind te vullen

2.4 Besturingsgroep

Bestaat uit instructies voor de uitvoering v/e programma!
JUMP-instructies = worden gebruikt om de besturingsunit een andere instructie te laten uitvoeren dan de eerstvolgende in de lijst

- ***Voorwaardelijke sprong*** = *"Als 0 gevonden is, spring dan naar stap 5"*
- ***Onvoorwaardelijke sprong*** = *"Spring naar stap 5"*

Stap 1 ➔ Vul register met een waarde uit geheugen	LOAD
Stap 2 ➔ Vul ander register met waarde uit geheugen	LOAD
Stap 3 ➔ Spring, als de tweede waarde 0 is, naar stap 6	JUMP
Stap 4 ➔ Deel de inhoud van register 1 door de inhoud van register 2 en sla het resultaat op in een derde register	
Stap 5 ➔ Sla de inhoud v/h derde register op in het geheugen	STORE
Stap 6 ➔ Stop	

2.5 Illustratieve machinetaal

De gecodeerde versie v/e machine-instructie bestaat uit 2 delen:
- ***Op-codeveld*** = geeft aan welke elementaire bewerking nodig is
- ***Operandveld*** = geeft meer gedetailleerde informatie over de bewerking

Men werkt met een machine die 16 registers (van 0 to F) en 256 werkgeheugencellen (van 00 tot FF) heeft.

Op-code	**Operand**		
0 0 1 1	***0 1 0 1***	***1 0 1 0***	***0 1 1 1***
3	***5***	***A***	***7***

De op-code heeft slechts twaalf instructies gaande van 1 naar C
In dit voorbeeld betekent de 3 dat men moet opslaan in register 5 op het geheugencel met adres A7

156C = vul register 5 met bitpatroon uit geheugencel 6C ➔ LOAD
166D = Vul register 6 met bitpatroon uit geheugencel 6D ➔ LOAD
5056 = Tel registers 5 en 6 op en sla op in register 0 ➔ ADD
306E = sla inhoud op van register 0 in geheugencel 6E ➔STORE
C000 = HALT ➔HALT

3. Programma-uitvoering

Besturingsunit in de CPU bevat twee speciale registers:

- **De programmateller**
  - Bevat het adres v/d volgende instructie die moet worden uitgevoerd en houdt dus bij tot waar het programma uitgevoerd is
- **Het instructieregister**
  - Slaat de instructie op die wordt uitgevoerd

De machinecyclus:

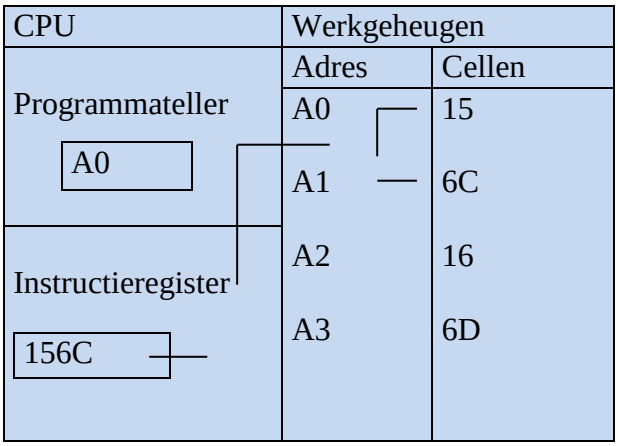
- ***Ophalen (fetch)***
  - *Besturingsunit vraagt aan werkgeheugen om instructie die opgeslagen is op het adres dat wordt aangereikt door de programmateller, waarna het wordt geplaatst in het instructieregister (dan verhogen programmateller met 2)*
- ***Decoderen (decode)***
  - *Besturingsunit decodeert de instructie*
- ***Uitvoeren (execute)***
  - *Besturingsunit voert uit door de schakelingen voor het uitvoeren v/d taak te activeren*

➔ Uitvoering van een JUMP-instructie is een bijzonder geval

➔ Indien B258 een JUMP-instructie is dan zal het verspringen als de inhoud van 2 gelijk is aan de inhoud v/e ander register dat gespecifieerd is geweest

3.1 Voorbeeld v/d uitvoering v/e programma

Werkgeheugen	
Adres	Cellen
A0	15
A1	6C
A2	16
A3	6D
A4	50
A5	56
A6	30
A7	6E
A8	C0
A9	00



Men plaatst het adres A0 v/d eerste instructie in de programmateller en start de machine.
De complete instructie bezet de geheugencellen met de adressen A0 en A1!
De besturingsunit verhoogt vervolgens de programmateller dan met 2 waardoor men begint met A2 en aan het einde v/d eerste fetch bevat de PT = A2 en het register 156C
Nadat de A8 en de A9 uitgevoerd zullen worden zal het register staan op C000 en PT = AA

3.2 Programma's versus gegevens

In het werkgeheugen kunnen vele programma's tegelijkertijd opgeslagen worden, zolang ze maar op verschillende locaties opgeslagen worden.
Een machine kan echter op zich geen onderscheid maken tussen gegevens en programma's.
Het concept om programma's en gegevens op dezelfde manier in het geheugen te coderen is echter een voordeel omdat een programma andere programma's (en zichzelf) kan bewerken op dezelfde manier alsof het met gegevens zou omgaan.

4. Wiskundige/logische instructies

4.1 Logische bewerkingen

Het resultaat v/e AND-bewerking is als volgt:

```
    10011010
AND 11001001
    10001000
```

Het resultaat v/e OR-bewerking of XOR-bewerking is als volgt:

```
    10011010          10011010
OR  11001001     XOR  11001001
    11011011          01010011
```

Een v/d belangrijkste toepassingen v/e AND-bewerking is de mogelijkheid om er nullen mee te plaatsen in een deel v/e bitpatroon zonder het andere deel aan te tasten

```
    00001111
AND 10101010
    00001010
```

➔ *Men weet dat meest significante bits van het resultaat zelfs zonder de tweede operand te kennen, alsook weten we dat we 4 minst significante bits kopieën zijn van de 2^{de} operand*
➔ ***Masking*** waarbij een operand (=***het masker***) bepaalt welk deel v/d andere operand invloed zal hebben op het resultaat ➔ ***handig bij bewerking van een bitmap!***

```
    11110000
OR  10101010
    11111010
```

➔ *De OR-bewerking kan gebruikt worden om een deel v/e reeks te dupliceren en enen in het andere deel te plaatsen*

```
    11111111
XOR 10101010
    01010101
```

➔ *De XOR-bewerking vormt het complement v/e bitreeks! (=indien enkel 1^{en} als operand)*

4.2 Rotatie- en shiftbewegingen

Met deze bewerkingen kunnen bits binnen een register verplaatst worden en worden vaak gebruikt om uitlijningsproblemen op te lossen.
Men deelt deze bewerkingen op in ***verplaatsingsrichting (naar links of rechts) en of het proces teruglopend is of niet.***

Als we de inhoud van de bits in een byte naar rechts of naar links verschuiven zal een bit verdrongen worden en zal er ook een gat ontstaan
➔ ***Rotatie (=teruglopende shift)*** = het verdrongen bit wordt in het gat geplaatst
➔ ***Logische shift*** = verdrongen bit verdwijnt en gat wordt gevuld met 0
➔ ***Arithmetic shift*** = de opening die ontstaat op de tekenbitpositie wordt gevuld met hetzelfde tekenbit

A	**5**	**0**	**1**
➔ ***Rotatie naar rechts***	➔ ***Bevat A3***		➔ ***1x naar rechts***

A3 = 10100011 ➔ 11010001 = D1

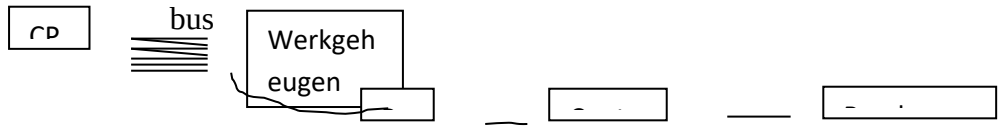
5. Communiceren met andere apparaten

Communicatie tussen een computer en andere apparaten wordt gewoonlijk afgehandeld door een apparaat (=***controller***) die zich ertussen bevindt.

- ⇨ Controllers zijn vaak zelf kleine computers
- ⇨ *Controller converteert berichten en gegevens zodanig dat ze compatibel zijn met de interne kenmerken v/d computer en die v/d randapparatuur waarop hij is aangesloten*
- ⇨ Controller is elektronisch verbonden met dezelfde bus die de CPU verbindt met het werkgeheugen en de CPU communiceert op dezelfde manier als met werkgeheugen
- ⇨ Er zijn extra op-codes voorzien in sommige ontwerpen om te communiceren met controllers en deze worden ***I/O-instructies*** genoemd
- ⇨ De set (I/O)-adressen van een controller wordt een ***poort*** genoemd

Een alternatief voor die extra op-codes is om dezelfde LOAD- en STORE-op-codes te gebruiken voor de communicatie met het werkgeheugen. In zo'n geval spreekt men van een ***memory-mapped I/O-systeem*** waarbij elke controller enkel reageert op een uniek set adressen, terwijl het werkgeheugen ontworpen moet zijn om verwijzingen naar deze locaties te negeren!

Memory-mapped I/O



Direct Memory Acces = mogelijkheid waarbij controller rechtstreek toegang heeft tot het werkgeheugen

- ⇨ *Dit komt de prestaties v/d computer ten goede uit omdat zo twee activiteiten tegelijkertijd kunnen uitgevoerd worden*
  - CPU schrijft programma en controller regelt overdracht
- ⇨ *Gebruik van DMA heeft ook een **negatief effect** en dat is dat de hoeveelheid informatie over de computerbus toeneemt*
  - ***CPU ⇔ Werkgeheugen***
  - ***CPU ⇔ Controller***
  - ***Controller ⇔ Werkgheugen***
- ⇨ Indien er een knelpunt ontstaat wanneer de CPU en de controllers beide toegang willen hebben tot de bus dan spreekt men de ***von Neumann-bottleneck***

Gegevensoverdracht tussen twee computercomponenten is zelden eenrichtingsverkeer

- ⇨ ***Handshaking*** = constante dialoog waarin computer en randapparatuur informatie uitwisselen over de status v/h apparaat om zo op elkaar te kunnen afstemmen
  - ***Statuswoord*** **=** de bits in dit woord stellen informatie over de condities v/h apparaat voor

Hoofdstuk 3 : Besturingssystemen

1. De ontwikkeling van besturingssystemen

Besturingssysteem = software die de werking van computer beheerst
Job/taak = de uitvoering van een programma

Batchverwerking = een van de eerste besturingssystemen waarbij jobs eerst worden verzameld en dan vervolgens worden uitgevoerd zonder verdere interactie met gebruiker

- ⇨ De jobs worden geladen in het massageheugen en wachten daar tot ze uitgevoerd worden in een ***takenwachtrij of job queue***
    - *Wachtrij is een manier van opslaan waarin objecten na elkaar worden opgeslagen en volgens FIFO-principe worden verwerkt*
        - In praktijk is er wel de mogelijkheid om een prioriteit aan jobs toe te kennen!

- ⇨ Bij deze batchverwerkingssystemen werd elke job vergezeld door een set instructies voor de voorbereiding v/d machine voor die bepaalde job!

- ⇨ *Belangrijkste nadeel van deze systemen is dat de gebruiker geen interactie met het programma meer heeft zodra het in de takenwachtrij geplaatst is*

Door dit nadeel werden nieuwe systemen ontworpen waarmee programma's uitgevoerd konden worden die een dialoog met de gebruiker onderhouden via werkstations

- ⇨ Deze mogelijkheid wordt ***interactieve verwerking*** genoemd

- ⇨ ***Real-timeverwerking*** = leveren van diensten waarbij handelingen v/d computer afgestemd zijn op de gebruiker
    - *Geeft problemen in een multi-useromgeving waarbij enkel 1 gebruiker bevredigend in real-time bediend zou kunnen worden*

Oplossing voor het probleem van real-timeverwerking was het proces ***timesharing*** dat verschillende jobs afwisselend na elkaar even uitvoerde

- ⇨ *Tijd wordt hierbij in intervallen verdeeld, waarna elke job telkens gedurende een interval wordt uitgevoerd*
    - Er ontstaat een illusie dat verschillende jobs tegelijkertijd worden uitgevoerd

- ⇨ ***Multitasking*** = timesharing maar bij systemen met één gebruiker

De ontwikkeling van ***multiprocessormachines*** maakte het zo dat verschillende taken aan verschillende processoren konden worden toegewezen in plaats van de tijd van 1 enkele processor te delenn!

- ⇨ *Besturingssysteem moet dan de activiteit v/d processoren coördineren!*
- ⇨ Hierbij treden problemen op zoals ***load balancing en scaling***

2. De architectuur van besturingssystemen

2.1 Meer over software

We zullen software in een machine verdelen in twee ruime categorieën:

- ⇨ **Toepassingssoftware**
    - *Programma's voor het uitvoeren van taken die samenhangen met het gebruik van de machine ➔ boekhoudsystemen, spellen,...*
- ⇨ **Systeemsoftware**
    - *Software voor het uitvoeren van taken die samenhangen met de werking van computerystemen in het algemeen*
    - Systeemsoftware levert de infrastructuur zodat de toepassingssoftware kan werken

Systeemsoftware kan ook weer onderverdeeld worden in 2 categorieën:

- ⇨ **Besturingssysteem**
- ⇨ **Utilitysoftware**
    - *Bestaat voor grootste deel uit programma's voor het uitvoeren van activiteiten die van vitaal belang zijn voor de werking van dat systeem, maar die niet in het besturingssysteem zitten* ➔ vergroot dus mogelijkheden
    - Schijf formatteren, bestand kopiëren,...
    - Door bepaalde activiteiten als utilitysoftware te implementeren, kan het ontwerp v/e besturingssysteem minder complex worden
    - Bovendien is het gemakkelijker om utilitysoftware aan te passen of uit te breiden naargelang de eisen die een bepaald systeem eist

2.2 Componenten van een besturingssysteem

Shell = deel v/h besturingssysteem dat communiceet met de gebruiker
Grafische gebruikersinterface = helpen shells hun taak uit te voeren door objecten op het beeldscherm voor te stellen als pictogrammen

- ⇨ *Shell is eigenlijk niet meer dan een interface tussen een gebruiker en het hart v/h besturingssysteem*
- ⇨ Een belangrijke component in de GUI-shells is de ***windowmanager*** *die vensters toewijst en bijhoudt welke toepassing aan welk venster gekoppeld is.*

Kernel = het inwendige van een besturingssysteem, dat de softwarecomponenten bevat die de basisfuncties van de computer uitvoeren

- ⇨ ***De filemanager*** is een van die componenten en coördineert het gebruik van het massageheugen v/d computer
    - Kwestie van te kunnen organiseren is het mogelijk bij de meeste filemanagers om bestanden te groeperen in ***directory's of mappen***
        - ***Pad =*** mappen binnen mappen
        - *Zo'n pad wordt vaak aangeduid door de namen v/d opeenvolgende mappen op te sommen, van elkaar gescheiden door schuine strepen*
    - Andere softwarecomponenten kunnen alleen toegang krijgen tot een bestand met behulp van de filemanager; indien de filemanager toegang verleent levert deze de noodzakelijke informatie om het bestand te kunnen vinden en bewerken
        - Deze informatie wordt opgeslagen in een stuk v/h werkgeheugen dat een ***bestandsbeschrijving (=file descriptor)*** heet
- ⇨ De kernel bestaat ook uit een verzameling ***apparaatstuurprogramma's (=device drivers)*** die met de controllers communiceren om bewerkingen uit te voeren op de randapparatuur die op de computer is aangesloten
    - *Elke apparaatstuurprogramma is speciaal ontworpen voor een bepaald apparaat*
- ⇨ ***De geheugenmanager*** is een ander component v/d kernel en deze coördineert het gebruik v/h werkgeheugen van de computer
    - *In een multitaskingomgeving worden veel programma's en gegevens gelijktijdig in het werkgeheugen geplaatst en moet de geheugenmanagers daarvoor afzonderlijke stukken geheugenruimte reserveren*

Indien de benodigde hoeveelheid werkgeheugen groter wordt dan de werkelijk beschikbare dan creeërt men de illusie van extra geheugenruimte door programma's en gegevens heen en weer te verschuiven tussen het werkgeheugen en het massageheugen.

*Men deelt de benodigde geheugenruimte dan in **pagina's** en slaat ze op in het massageheugen, als in het werkgeheugen andere pagina's nodig zijn dan verwisselt men ze met die in het massageheugen* ➔ ***virtueel geheugen***

- ⇨ De kernel bevat ook een ***scheduler*** (=welke activiteit moet worden uitgevoerd) en een ***dispatcher*** (=bepaalt de verdeling v/d tijdslots voor activiteiten)

2.3 Het besturingssysteem starten

Bootstrapping = procedure waarbij men het besturingssysteem zelf start

- ⇨ *Men kopieert het besturingssysteem vanuit het massageheugen naar het werkgeheugen*

Read-Only Memory (=ROM) = deel van het geheugen waarvan de inhoud permanent is

- ⇨ Deze zorgt ervoor dat de CPU zijn programmateller bij opstarten altijd de waarde v/h adres van een bepaalde cel heeft
- ⇨ ***Bootstrap*** *= programma dat in deel van het ROM-werkgeheugen zit en dat de CPU opdracht geeft om het besturingssysteem uit het massageheugen te halen en in het werkgeheugen te steken*

3. De activiteiten van de machine coördineren

3.1 Het procesconcept

Een programma is een verzameling aanwijzingen
De uitvoering van een programma is een dynamische activiteit waarvan de eigenschappen in de tijd veranderen = ***een proces***

- ⇨ Bij een proces is er ook een huidige status v/d activiteit (=***processtatus***) en deze heeft te maken met de huidige positie in het programma dat wordt uitgevoerd en de waarden in de andere CPU-registers (en de daarbij horende geheugencellen)
- ⇨ *Processtatus is een momentopname van de machine op een bepaald tijdstip*

3.2 Procesbeheer

De taken die samenhangen met de uitvoering van processen worden afgehandeld door de scheduler en de dispatcher!

- ⇨ **Scheduler**

Scheduler houdt bij welke processen er in het computersysteem aanwezig zijn, introduceert nieuwe processen bij deze verzameling en verwijdert de voltooide processen

De scheduler gebruikt een ***procestabel***, gesitueerd in het werkgeheugen, om alle processen in de gaten te kunnen houden.

Bij elke nieuwe taak voegt de scheduler een nieuw record toe aan de procestabel en die bevat dan informatie over het geheugengebied dat ter beschikking wordt gesteld, de prioriteit van het proces en of het proces gereed is of wacht.
➔ ***Proces is gereed*** als het verder kan worden uitgevoerd
➔ ***Proces wacht*** als het niet verder kan worden uitgevoerd omdat het wacht op een externe gebeurtenis

⇨ **Dispatcher**
De dispatcher zorgt dat de geplande processen ook daadwerkelijk uitgevoerd worden.

Bij een timesharingsysteem gebeurt dit door de tijd in korte segmenten, tijdslots, time slices of quantums op te splitsen en vervolgens de CPU steeds gedurende een tijdslot een van de processen uit te laten voeren.
➔ De omschakeling tussen processen heet ***procesomschakeling (=proces switch)***

Telkens wanneer een proces in zijn tijdslot begint, start de dispatcher een timer die het volle quantum klokt. Aan het eind v/h quantum geneert de timer een signaal (=***interrupt***). CPU maakt dan de huidige machinecyclus af, slaat de positie in het huidige proces op en voert vervolgens de ***interrupt handler*** (=onderdeel dispatcher) uit waarna een nieuw proces begint.

4. Omgaan met concurrentie tussen processen

4.1 Semaforen

Men maakt gebruik van een vlag om aan te duiden of een randapparaat bezet is of niet.
Een 'clear'-vlag (= waarde 0) betekent dat het beschikbaar is
Een 'set'-vlag (= waarde 1) betekent dat het bezet is

Telkens een proces de randapparatuur niet meer nodig heeft wordt ze gekoppeld aan een wachtend proces of wordt vlag op 'clear' gezet

Er is echter ook een nadeel bij dit systeem en dat is het feit dat voor het testen en instellen van de vlag er verschillende machine-instructies nodig kunnen zijn!

- ⇨ Men kan dit probleem oplossen door te eisen dat het testen en eventueel veranderen van instelling v/d vlag zonder onderbreking voltooid moet worden
  - ***Interrupt-disable-instructie*** = toekomstige interrupts worden onderdrukt
  - ***Interrupt-enable-instructie*** = CPU zal opnieuw gaan reageren op interruptsignalen
- ⇨ Een andere oplossing is een ***test-and-set-instructie*** waarbij men de vlag test en instelt met 1 machine-instructie

Een correct geïmplementeerde vlag heet een semafoor!
➔ ***Kritiek gebied*** = bepaalde reeksen instructies die door maar één proces gelijktijdig mogen uitgevoerd worden
➔ ***Wederzijdse uitsluiting*** = eis dat slechts één proces een kritiek gebied mag uitvoeren

- ⇨ *Wordt meestal gerealiseerd door het kritieke gebied te bewaken met een semafoor die 'clear' of 'set' zegt*

4.2 Deadlock

Deadlock = impasse die ontstaat wanneer twee of meer processen elkaar ophouden omdat ze allemaal wachten op een bron die toegewezen is aan een ander proces

Proces heeft toegang tot printer maar wacht op tapestreamer ➔ *ander proces heeft toegang tot tapestreamer maar wacht op printer*

Systemen waarin de processen nieuwe processen mogen maken (=***vorken***) om subtaken uit te voeren kunnen ook leiden tot een deadlock

Een deadlock kan alleen maar ontstaan wanneer de volgende 3 voorwaarden worden voldaan:

1. ***Er zijn meerdere partijen die niet-deelbare bronnen willen gebruiken***
2. ***De bronnen worden niet in één keer volledig benut; men zal later nog meer nodig hebben***
3. ***Een eenmaal toegewezen bron kan niet zonder meer afgepakt worden***

Het is mogelijk om een impasse op te heffen door slechts 1 van de 3 voorwaarden aan te pakken.

- Technieken die de derde voorwaarde aanpakken worden meestal ***deadlockdetectie- en herstelmechanismen*** genoemd
- Technieken die de eerste twee voorwaarden aanpakken worden ***deadlockvermijdingsmechanismen*** genoemd.

Men gebruikt ***spooling*** om de eerste voorwaarde aan te pakken en een niet-deelbare bron deelbaar te maken

- ⇨ *Men koppelt een proces aan een apparaatstuurprogramma dat de informatie opslaat in het massageheugen waarbij alle processen denken toegang te hebben tot de apparatuur en telkens ze dan vrijkomt stuurt men het volgende proces uit het massageheugen*

5. Beveiliging

Bij beveiliging moet je ook denken aan het voorkomen van onvolkomenheden in het ontwerp!

Bij beveiliging in de zin v/h voorkomen van kwadaardig gedrag, moet een besturingssysteem niet alleen afrekenen met externe bedreigingen, maar ook met interne
➔ *Proces dat probeert toegang te krijgen tot cellen die niet zijn toegewezen aan dat proces*

Gebruikelijkste methode om te verdedigen tegen externe gebruikers is om een wachtwoord in te stellen bij het ***inloggen***.

- ⇨ Om diefstal van wachtwoorden te voorkomen kan een besturingssysteem zo ontworpen worden dat het een stroomfoute wachtwoorden meldt
- ⇨ Men kan ook de inbreker de illusie geven dat dze toegang heeft gekregen om vervolgens verkeerde informatie te genereren en zijn identiteit te noteren
- ⇨ CPU's hebben meestal speciale registers waarin het besturingssysteem de begrenzingen van het aan elk proces toegewezen geheugengebied kan opslaan

Het probleem bij die laatste beveiling is dat men mogelijk die begrenzingen zelf zou kunnen aanpassen. Om dit te voorkomen zijn bepaalde instructies in een machinetaal ***privileged instructions*** en kan de CPU zowel in ***privileged- als non-privileged modus*** werken.

In het begin werkt de CPU in privileged modus wat betekent dat alle instructies uitgevoerd kunnen worden; een speciale instructie kan dat echter omvormen.
In een non-privileged modus weigert de CPU nog privileged instructies uit te voeren

➔ *Er bestaan in de praktijk wel een aantal* ***privilegeniveaus***

Pogingen om een privileged instructie toch uit te voeren veroorzaken een interrupt waarna de CPU weer overschakelt naar privileged modus maar de besturing komt in handen van een interrupt handler binnen het besturingssysteem

Hoofdstuk 5 : Algoritmen

1. Het concept algoritme

1.1 Een informele beschouwing

Machinecylus v/e CPU is niets meer dan volgende eenvoudige algoritme :

Voer, zolang de Halt-instructie niet uitgevoerd is, de volgende stappen uit :
a. Haal een instructie op (fetch)
b. Decodeer de instructie
c. Voer de instructie uit

Algoritmen kunnen echter ook voor andere dan technische activiteiten worden gebruikt!

1.2 De formele definitie v/e algoritme

Een algoritme is een geordende reeks ondubbelzinnige, uitvoerbare stappen die een proces beschrijven.

*De reeks stappen in een algoritme moeten dus **geordend zijn**, wat niet betekent dat de stappen in een vooraf vastgelegde volgorde uitgevoerd moeten worden!*

- ⇨ ***Parallelle algoritmen*** = bevatten meerdere reeksen stappen die elke ontworpen zijn om door verschillende processoren in een multiprocessormachine uitgevoerd te worden (waardoor men niet zoiets heeft als één reeks stappen die achtereenvolgens worden uitgevoerd)

Een algoritme moet ook uit stappen bestaan die uitgevoerd kunnen worden!
➔ Een stap heet ***effectief*** als die doenbaar is

Stappen in een algoritme moeten ook ***ondubbelzinnig*** zijn, *wat wil zeggen dat de informatie in de status v/h proces voldoende moet zijn om de handelingen voor elke stappen volledig en eenduidig te kunnen uitvoeren.*

In een algoritme moet een proces gedefinieerd zijn waarmee het algoritme beëindigd wordt!
➔ ***De uitvoering v/h algoritme moet door zichzelf stopgezet worden!***

1.3 De abstracte aard van algoritmen

Het is belangrijk om het onderscheid tussen een algoritme en de representatie ervan te benadrukken!

- ⇨ ***Algoritme is abstract en representatie is fysiek!***
- ⇨ De dubbelzinnigheid zit dus in de representatie maar niet het algoritme zelf!

Een programma is een representatie van een algoritme
Een proces is de activiteit waarbij een programma wordt uitgevoerd

- ⇨ *Proces is dus activiteit waarbij een algoritme wordt uitgevoerd*

2. De represenatie van een algoritme

2.1 Primitieven

Voor het representeren v/e algoritme is een vorm van taal noodzakelijk, maar hierbij ontstaan communicatieproblemen wanneer de gebruikte taal niet nauwkeurig gedefinieerd is (ook wanneer de informatie niet voldoende gedetailleerd is).

Men omzeilt deze problemen door in de informatica goed gedefinieerde bouwstenen (=***primitieven***) te gebruiken!
Elke primitief bestaat uit twee delen : de syntax (=symbool) en de semantiek (=uitleg)

Een programmeertaal ontstaat dan wanneer men een reeks primitieven combineert met een verzameling regels voor de manier waarop primitieven kunnen gecombineerd worden om meer complexe ideeën te representeren.

Als een algoritme met de detaillering v/e primitief word voorgesteld dan ontstaat met zekerheid een programma dat door de machine uitgevoerd kan worden, maar deze manier is echter omslachtig!

2.2 Pseudocode

Een pseudocode is in wezen een notatiesysteem om ideeën informeel aan te duiden tijdens de ontwikkeling v/e algoritme

- ⇨ *Men maakt zo'n code door de regels v/e taal waarin de uiteindelijke versie v/h algoritme gerepresenteerd zal worden, wat vrijer toe te passen*

Men gebruikt de volgende vorm om namen en waarden te koppelen

Naam ← Expressie

Waarbij 'Expressie' slaat op de ***waarde*** en 'Naam' op de ***zinnige naam*** waaraan de waarde wordt gekoppeld.

*Een ander structuur is de structuur die bepaalt wanneer men moet kiezen tussen twee mogelijke activiteit, afhankelijk van een voorwaarde (=**WAAR of NIET WAAR**)*

als (voorwaarde) **dan** (activiteit)
anders (activiteit)

Voor de gevallen waar geen 'anders' activiteit nodig gebruiken we dan deze syntax :

als (voorwaarde) **dan** (activiteit)

Vaak moeten statements herhaald worden zolang een bepaalde voorwaarde WAAR is :

terwijl (voorwaarde) **doe** (activiteit)

Als de voorwaarde dan NIET WAAR wordt dan voert men de volgende instructie uit!

Inspringen maakt een programma beter leesbaar!

```
als  (het niet regent)
  dan (als (temperatuur = hoog)
       dan (ga zwemmen)
       anders (ga golfen)
       )
anders (kijk televisie)
```

We willen onze pseudocode ook gebruiken om activiteiten te beschrijven die in andere toepassingen gebruikt kunnen worden!
➔ Men geeft aan zo'n stukje programma de naam ***procedure***

Men zal elk afzonderlijk stukje pseudocode beginnen met een statement v/d vorm:

procedure naam

'naam' is hierbij de naam v/h betreffende stukje code! Zie hier een vb :

```
procedure Begroeting
Teller ← 3 ;
terwijl (tTeller > 0) doe
    (druk het bericht 'Hallo' af
     en Teller ← Teller – 1)
```

Als de taak die door een procedure uitgevoerd wordt ook op een andere plaats in de pseudocode moet worden uitgevoerd, zullen we daar enkel de naam v/d procedure gebruiken:

```
als (...) dan (voer de procedure VerwerkLening uit)
        anders (voer de procedure AanvraagAfwijzen uit)
```

De verwijzing binnen de representatie v/d procedure moet zo algemeen mogelijk zijn

procedure Sorteer (Lijst)

Als later de diensten van Sorteer nodig zijn, dan kunnen we aangeven door welke lijst 'Lijst' in de procedure Sorteer vervangen moet worden

We kunnen afspreken om na een haakje sluiten, een kort ***commentaar*** op te nemen dat toelicht welke statement daarmee afgesloten wordt

```
terwijl (....) doe
    ( als (....)
        dan ( .
              .
              .
            ) einde als
      ) einde terwijl
```

3. Algoritmen ontdekken

De ontwikkeling v/e programma bestaat uit twee activiteiten : *het opsporen v/h onderliggende algoritme, waarna dat algoritme als een programma voorgesteld moet worden*

3.1 De kunst v/h oplossen van problemen

G. Polya definieerde de volgende basisprincipes voor oplossen van problemen :
- ***Fase 1*** : Begrijp het probleem
- ***Fase 2*** : Ontwikkel een plan om het probleem op te lossen
- ***Fase 3*** : Voer het plan uit
- ***Fase 4*** : Controleer de oplossing op nauwkeurigheid en nut voor andere toepassingen

Indien we dit betrekken op het ontwikkelen van programma's worden deze principes:
- ***Fase 1*** : Begrijp het probleem
- ***Fase 2*** : Probeer te bedenken hoe een algoritme het probleem kan oplossen
- ***Fase 3*** : Formuleer het algoritme en representeer het in programmavorm
- ***Fase 4*** : Controleer nauwkeurigheid en toepasbaarheid voor andere problemen

Je lost geen problemen op door te volgen. Je kunt ze alleen oplossen door initiatief te nemen en slagvaardig te werk gaan. De fases van Polya zullen waarschijnlijk pas achter duidelijk te onderscheiden zijn!

De fases van Polya hebben niet noodzakelijkerwijs de genoemde volgorde!

Om een volledig begrip te eisen en pas daarna oplossingen aan te dragen, is enigszins onrealistisch!

Henri Poincaré bestudeerde het proces dat wanneer men vruchteloos bezig is aan een probleem en daarna een andere taak uitvoert, plotseling de oplossing vindt!
➔ ***Incubatieperiode*** = periode tussen bewust werken en de plotseling inspiratie

3.2 Een voet tussen de deur

Er zijn talloze methodieken om succesvol problemen op te lossen. Deze methoden gebruiken allemaal een techniek genaamd '***een voet tussen de deur'***

A voorspelt dat B zal winnen
B voorspelt dat D als laatste eindigt
C voorspelt dat A als derde eindigt
D voorspelt dat de voorspelling van A uitkomt
➔ Slecht 1 voorspelling bleek WAAR te zijn
➔ A en D zijn gelijkwaardig en dit is de 'voet tussen de deur'

Er is natuurlijk een verschillen tussen weten dat je een voet tussen de deur moet krijgen, en weten hoe je dat moet doen ➔ dit vereist creativiteit!

Er zijn echter een paar algemene methodieken voor de minder creatieven :
- ⇨ 1 daarvan is om ***het probleem te benaderen vanuit de oplossing.***
- ⇨ Men kan zoeken naar een vergelijkbaar probleem dat ofwel gemakkelijker op te lossen is, ofwel al opgelost is
- ⇨ **Stapsgewijze verfijning**
    - Doel is om één taak te doorgronden door het eerst in verschillende kleinere problemen op te splitsen
    - ***Dit is top-downmethodiek*** omdat de problemen eerst algemeen, maar bij elke opsplitsing specifieker worden
        - Bottom-upmethodiek is complementair
    - Deze methodiek laat de mogelijkheid toe om het programmeren in ***teamverband*** te doen
    - *Bij de stapsgewijze verfijning gaat het niet zozeer om het ontdekken van nieuwe algoritmen, als wel om het aanbrengen van structuur*

Pascal-casing = KostenPerProduct
Camel-casing = kostenPerProduct

4. Iteratieve structuren

Iteratieve structuren = structuren waarin een verzameling instructies steeds opnieuw herhaald wordt

4.1 Het sequential search-algoritme

```
procedure Zoeken (Lijst, DoelWaarde)
als (Lijst leeg)
    dan
      (Declareer een zoekfout)
    anders
      (Selecteer het eerste item in de lijst en ken dat toe aan TestItem;
       terwijl (DoelWaarde > TestItem en er nog andere items zijn)
          doe (Selecteer het volgende item in de Lijst en noem dit TestItem);
       als (DoelWaarde = TestItem)
          dan (Declareer dat zoeken gelukt is.)
          anders (Declareer dat zoeken mislukt is.)
      ) einde als
```

Dit algoritme wordt het ***sequential search-algoritme*** genoemd omdat het de verschillende items in dezelfde volgorde als waarin ze in de lijst voorkomen controleert.

4.2 Lusbesturing

Lus = iteratieve structuur die steeds een instructie herhaalt
Body v/d lus = verzameling instructies die steeds opnieuw wordt uitgevoerd

```
terwijl (voorwaarde) doe (body)
```

Besturing v/e lus bestaat uit 3 componenten:

- **Initialiseren** : Definieer een initiële toestand die gewijzigd wordt tot de eindigingsvoorwaarde bereikt is

- **Testen** : Vergelijk de huidige toestand met de eindigingsvoorwaarde en stop de herhaling als deze gelijk zijn

- **Modificeren** : Wijzig de toestand zodanig dat deze dichter bij de eindigingsvoorwaarde komt

De testactiviteit zorgt ervoor dat de lus eindigt zodra de test aan een bepaalde voorwaarde voldoet. Deze voorwaarde wordt de ***eindigingsvoorwaarde*** genoemd.
➔ *De eindigingsvoorwaarde is de negatieve tegenhanger v/d voorwaarde in de terwijl-structuur*

(doelwaarde ≤ testitem) of (er zijn geen te testen items meer)

De andere twee activiteiten in de lusbesturing zorgen ervoor dat de eindigingsvoorwaarde uiteindelijk WAAR zal worden.
De initialisatiestap zorgt voor een startvoorwaarde.
De modificatiestap verschuift die voorwaarde in de richting van de eindigingsvoorwaarde.
De modificatie- en initialisatiestap moeten leiden tot de eindigingsvoorwaarde of anders hebben we te maken met een oneindige lus

Er zijn twee veelgebruikte lusstructuren:

terwijl (voorwaarde) **doe** (activiteit)

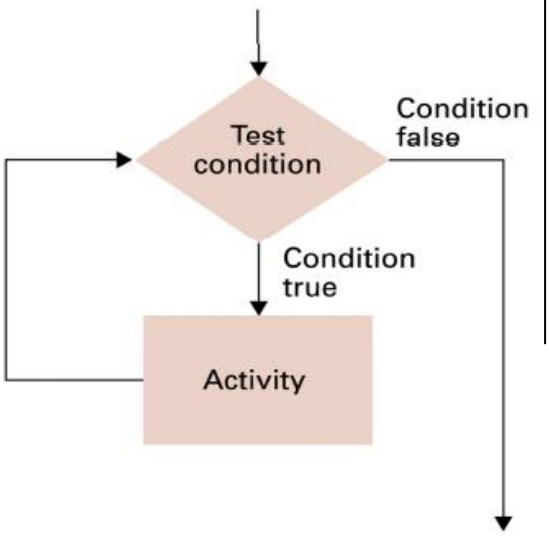
of

herhaal (activiteit) **tot** (voorwaarde)

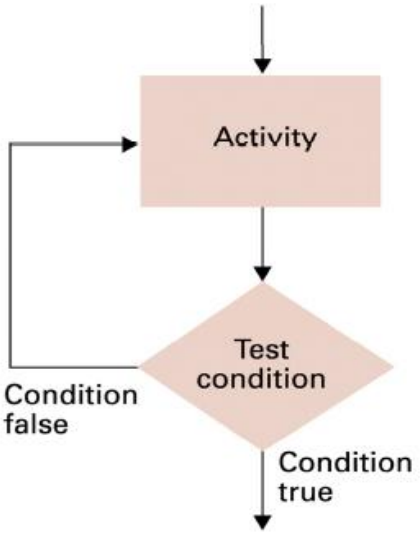
De test voor het eindigen van de terwijl-structuur wordt uitgevoerd voordat de body van de lus wordt uitgevoerd

De test voor het eindigen van de herhaal-structuur wordt uitgevoerd nadat de body v/d lus wordt uitgevoerd

terwijl-lustructuur (=pretest-lus)



herhaal-lusstructuur (=posttest-lus)

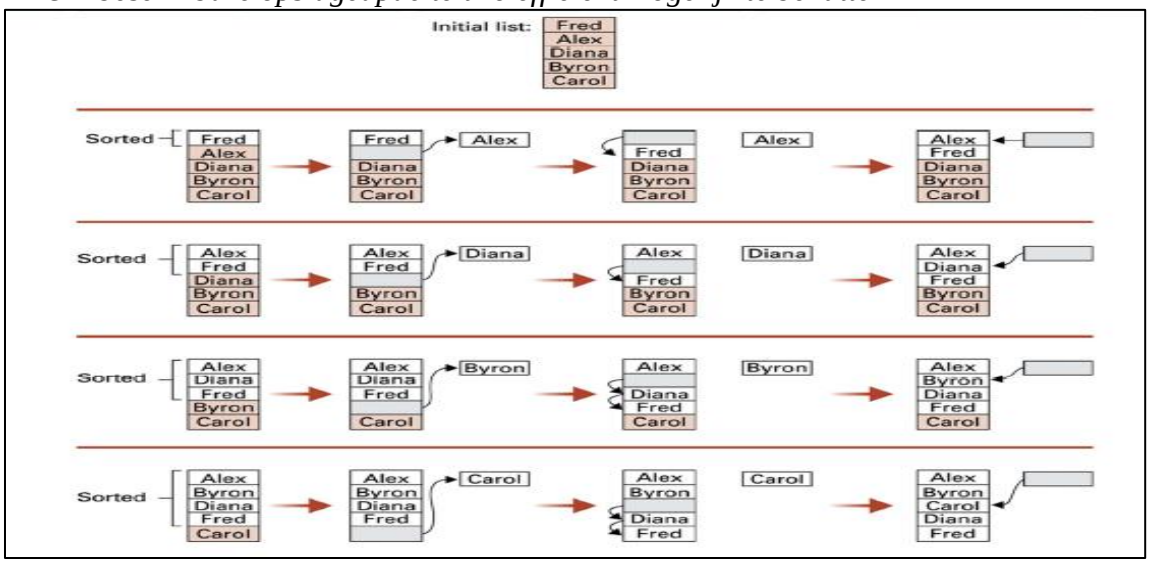


Dit zijn ***stroomschema's*** waarbij een *ruit een beslissing aangeeft en een rechthoek een bepaalde statement aangeeft*

4.3 Het insertion sort-algoritme

Stel we willen een lijst sorteren, maar we willen de lijst sorteren door de items erin door elkaar te verschuiven, echter zonder daarvoor de lijst naar een andere locatie te verplaatsen.

➔ *Om beschikbare opslagcapaciteit zo efficiënt mogelijk te benutten*



Als je goed kijkt vertegenwoordigt elke rij in de afbeelding hetzelfde algemene proces :

- ⇨ *Pak de eerste kaart v/h ongesorteerde deel v/d lijst op, verschuif alle kaarten met een grotere naam een positie omlaag en leg de opgepakte kaart op de opengevallen plaats*
- ⇨ Die opgepakte kaart zulen we ***de spil*** noemen
- ⇨ ***Insertion sort-algoritme*** = het programma sorteert dus een lijst door steeds een item uit de lijst te halen en het op de juist positie te leggen

```
procedure Sorteer (Lijst)
N ← 2;
terwijl (de waarde van N niet groter is dan de lengte v/d lijst) doe
    (Selecteer het N-de item in de lijst als spil;
     Verplaats de spil naar een tijdelijke locatie, zodat een opening in de lijst ontstaat;
       terwijl (er een naam boven de opening is en die naam groter is dan de spil) doe
           (verplaats de naam boven de opening naar de opening waardoor boven die
             naam een nieuwe opening ontstaat)
       Verplaats de spil naar de opening in de lijst;
       N ← N + 1
     )
```

5. Recursieve structuren

Zijn ook structuren waarmee herhaling van activiteiten geïmplementeerd kan worden, maar bij een recursie is het mogelijk om een reeks instructies *als een subtaak van zichzelf uit te voeren.*

5.1 Het binary search-algoritme

Binary search-algoritme pas een verdeel-en-heerstactiek toe bij het zoekproces.

Oorspronkelijke lijst	**Eerste sublijst**	**Tweede sublijst**	**Derde sublijst**
Alex			
Bob			
Carin			
Diana			
Hans			
Irene			
Jan	**Jan**	**Jan**	Jan
Louis	**Louis**	Louis	
Michiel	Michiel		
Olivier	Olivier		

Men verdeelt de gesorteerde lijst in twee en pakt het 'midden'; als dit niet de waarde is die men zoekt (=hier Jan) dan verdeelt een helft nogmaals in twee (=hij is gesorteerd dus bepalen van de juist helft is logisch) en dan kijkt men of het nu de juiste waarde is. ***Men blijft halveren tot de waarde is gevonden!***

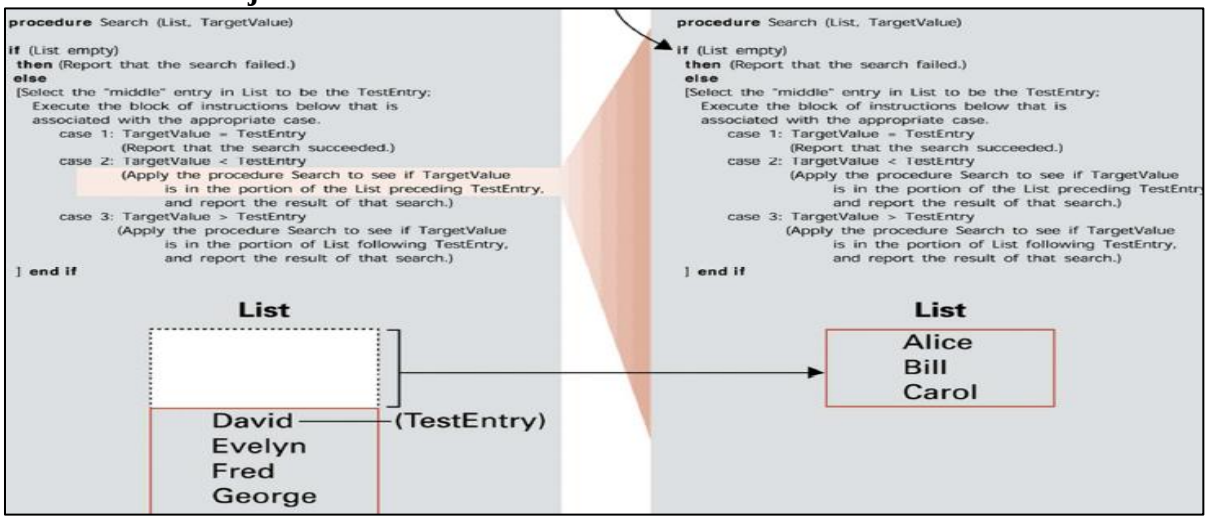
➔ *Splitsing bij gelijk aantal waarden pakt men de eerste waarde v/d tweede helft*

De strategie is dus om de lijst in steeds kleinere segmenten op te splitsen tot de gezochte waarde gevonden is of de zoekopdracht vernauwd is tot een leeg segment (=mislukt)

In de pseudocode moet men een zoekopdrachtverwerken die het doorzoeken van een tweede deel mogelijk maakt!!

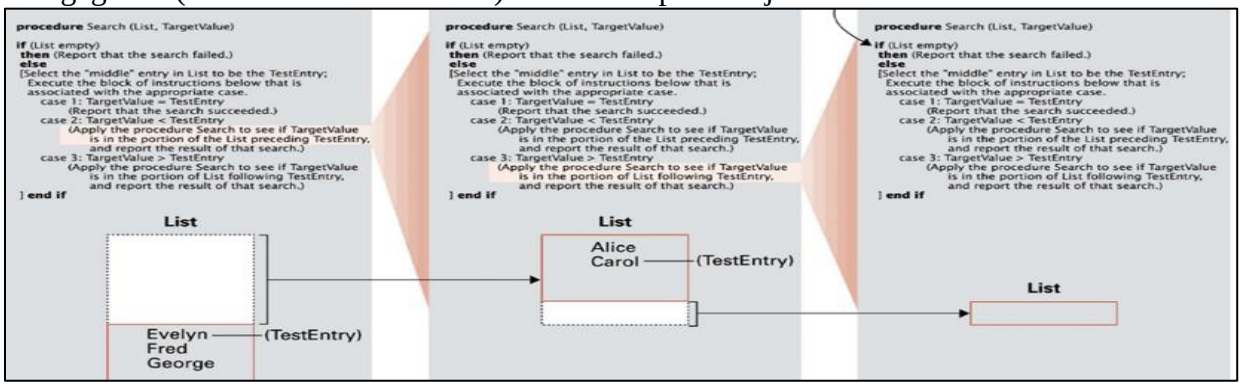
procedure Zoeken (Lijst, DoelWaarde)
als (Lijst is leeg)
dan
(Declareer dat zoeken mislukt is.)
anders
[Selecteer het 'middelste' item in de lijst als TestItem;
Voer het onderstaande blok instructies uit voor het betreffende geval.
geval 1 : DoelWaarde = TestItem
(Declareer dat zoeken gelukt is.)
geval 2 : DoelWaarde < TestItem
(Pas de procedure Zoeken toe om te controleren of de DoelWaarde in het deel van de lijst voor TestItem voorkomt en declareer het resultaat van die zoekopdracht.)
geval 3 : DoelWaarde > TestItem
(Pas de procedure Zoeken toe om te controleren of de DoelWaarde in het deel v/d lijst na TestItem voorkomt en declareer het resultaat van die zoekopdracht.)
] **einde als**

Voorbeeld waarbij we Bill zoeken:



Er wordt een tweede versie v/d procedure zoeken gestart die ook Bill zoekt!!
Wanneer de tweede versie dan voltooid is, rapporteert die zijn bevindingen aan de eerste versie waarna deze verder wordt uitgevoerd (en de tweede versie wordt 'vergeten'. De tweede versie is dus ondergeschikt aan de oorspronkelijke!

In geval dat een waarde niet voorkomt in een lijst gebeurt dan dit en wordt de mislukking doorgegeven (van 3de naar 2de naar 1ste) aan de oorspronkelijke versie!



5.2 Recursieve besturing

Bij de zoekopdracht bij het sequential search-algoritme de herhaling in cirkels wordt uitgevoerd, wordt bij het binary search-algoritme elke fase v/d herhaling als subtaak v/d vorige fase uitgevoerd ➔ ***Recursie***

Bij procedures die werken met recursie wordt elke versie v/d procedure een ***aanroep*** genoemd en er is uiteindelijk altijd slechts één die daadwerkelijk verder wordt uitgevoerd.

Recursieve systemen zijn ook afhankelijk van ***tests*** voor een eindigingsvoorwaarde en men zal algemeen eerst testen voor men een nieuw aanroep begint!
Het proces wordt ook impliciet ***geïnitialiseerd*** door het definiëren v/e initiële lijst en een gezochte waarde.
En men ***modificeert*** ook de eigen taak door een andere routine te vragen en een kleinere lijst te doorzoeken

6. Efficiëntie en correctheid

6.1 De efficiëntie van algoritmen

Men maakt bij de analyse over de prestaties van algoritmen meestal e***en best-case analyse, een average-case analyse en een worst-case analyse***.

Insertion sort-algoritme	**Binary Search-algoritme**
- Bij een lijst met n items zal in het beste geval **n – 1** vergelijkingen nodig zijn - Bij een lijst met n items zal in het slechtste geval (=bij een lijst dat aflopend gesorteerd is) **n (n – 1) / 2** vergelijkingen nodig zijn ➔ *Werkt optimaal bij korte lijsten, maar tijd gaat radicaal omhoog naarmate lijst stijgt*	- Bij een lijst met n items zal het maximaal **lg n** items in het worst-case scenario testen ➔ *Tijdstoename wordt steeds kleiner naarmate de lengte v/d lijst toeneemt!!*
Sequential search-algoritme	
- Bij een lijst met n items zal het gemiddeld **n/2** items testen	

Het is gebruikelijk om algoritmen te classificeren op basis v/d vorm van zijn grafieken die gewoonlijk zijn gebaseerd op de worst-case analyse v/h algoritme.
➔ **Θ (n^2)** = slaat op algoritmen waarvan de grafiek parabool is (=insertion sort)
➔ **Θ (lg n)** = slaap op algoritmen waarvan grafiek de vorm v/e logaritmische expressie heeft

6.2 Softwareverificatie

Er dient een onderscheid gemaakt te worden tussen een programma dat verondersteld juist is en een programma dat werkelijk juist is.

- *Men begint het bewijs van correctheid door de* ***randwaarden*** *te controleren! (=”zijn het allemaal namen die in een lijst voorkomen?”)*
- *Bekijken hoe de gevolgen van de randvoorwaarden doorwerken in het programma*

- *We kunnen de correctheid v/e programma dan steeds bewijzen door* ***beweringen*** *te controleren die op verschillende punten in het programma gedaan kunnen worden*

➔ ***Het resultaat is een verzameling beweringen = gewenste uitvoer = programma correct***

Hoofdstuk 6 : Programmeertalen

1. Historisch perspectief

1.1 De eerste generaties

Debuggen = fouten die in machinetaal moeten worden opgespoord en gecorrigeerd

Onderzoekers vereenvoudigden het programmeerproces door notatiesystemen te ontwikkelen waarmee instructies ***mnemonisch***, in plaats van getallen, konden worden weergegeven.

Verplaats de inhoud van register 5 naar register 6 ➔ **4 0 5 6** In een mnemonisch systeem wordt dit ➔ **MOV R5 , R6** **LD = load ; ADDI = add ; ST = store ; HLT = halt** Prijs, Verzendkosten,… zijn ***identifiers*** die verwijzen naar de geheugencellen op de locaties 6C,6D,6E	**1 5 6 C** **1 6 6 D** **5 0 5 6** **3 0 6 E** **C 0 0 0** *Wordt dan* **LD R5 , Prijs** **LD R6 , Verzendkosten** **ADDI R0 , R5 R6** **ST R0 , TotaleKosten** **HLT**

Assemblers = zijn programma's die andere programma's die geschreven zijn in mnemonische vorm omzetten naar numerieke machinetaalinstructies die de machine kan verwerken
Assemblertaal = mnemonisch systeem waarin programma's kunnen worden voorgesteld

- ⇨ *Deze is niet erg geschikt voor het maken van complexe programma's*
- ⇨ *Programma's in assemblertaal zijn alleen bruikbaar op een bepaald type computer*
- ⇨ *Programmeur moet nog steeds denken in de kleine, opeenvolgende stappen v/d taal v/e bepaalde machine*

Eerste generatie programmeertalen = machinetaal
Tweede generatie programmeertalen = assembleertaal
De derde generatie programmeertalen werken met primitieven van een hoger niveau en de primitieven zijn ook ***machineonafhankelijk***!

- ⇨ Men streeft om primitieven v/e hoog niveau te vinden
- ⇨ ***Een translator*** kan een verzameling primitieven vertalen naar machinetaalprogramma's
- ⇨ ***Een interpreter*** voert een programma uit op basis v/d instructies op het hogere niveau

Natuurlijke talen onderscheiden zich van formele talen (=programmeertaal) doordat de laatste een nauwkeurig gedefinieerd grammatica hebben!

1.2 Na de machineafhankelijkheid

Bepaalde kenmerken v/d onderliggende machine spelen soms een hinderlijke rol waardoor het dan ook vaak noodzakelijk is om tenminste kleine aanpassingen in een programma uit te voeren voordat het ook op een andere machine uitgevoerd kan worden.
Er ontbreekt ook vaak eenduidigheid over wat de correcte definitie van een bepaalde taal is (=ANSI en ISO,...).
Het feit dat men geen machineonafhankelijkheid heeft bereikt is om twee redenen van weinig belang :

➔ **De gerealiseerde mate van onafhankelijkheid maakt overgang redelijk vloeiend**
➔ **Het streven was slechts een voedingsbodem voor meer uitdagende doelen**

1.3 Programmeerparadigma's

De term generaties bij classificatie van programmeertalen is gebaseerd op een ***lineaire schaal*** die aanduidt in welke mate een gebruiker v/d taal zich niet hoeft te bekommeren om computereigenschappen en zich alleen hoeft te contreren op het probleem zelf

Er bestaan ook nog een ***meerdimensionaal model*** die 4 routes kent die resulteerden in verschillende modellen :

- ⇨ **Het imperatieve paradigma**

  We benaderen het programmeerproces door een algoritme te vinden waarmee het op te lossen probleem kan opgelost worden en dat dat algoritme vervolgens als een reeks opdrachten moet worden uitgedrukt (=machinetaal, pseudocode)

- ⇨ **Het declaratieve paradigma**

  Een programmeur moet het probleem beschrijven. Hij moet eerst een algemeen logaritme waarmee het probleem opgelost kan worden ontdekken en implementeren. *Daarna giet men problemen in een vorm die compatibel is met dat algoritme en men past die dan toe*.

- ⇨ **Het functionele paradigma**

  Beschouwt het proces v/h ontwikkelen van programma's als het aan elkaar schakelen van 'zwarte dozen' (=functies) met ingangen en uitgangen.
  Men probeert dus om de elementaire, primitieve functies met elkaar te combineren tot een systeem dat de gewenste resultaten berekent. *Programmeerproces bestaat dus uit construeren van functies als samenstellingen van eenvoudiger functies.*

- ⇨ **Het objectgeoriënteerde paradigma**

  Wordt gebruikt bij ***objectgeoriënteerd programmeren (=OOP)*** waarbij eenheden gegevens beschouwd worden als actieve 'objecten', in plaats van passieve eenheden.
  - ➔ *Lijsten worden gebouwd als objecten die naast de lijst ook bewerkingsprocedures bevat*
  - ➔ ***Een lijst kan dan bv zichzelf sorteren!***
  - ➔ ***Vormt een natuurlijke omgeving voor de 'bouwsteenbenadering'!***

2. Traditionele programmeerconcepten

FORTRAN en C ➔ derdegeneratie imperatieve talen
C + + ➔ objectgeoriënteerde taal als uitbreiding van C
Java en C# ➔ objectgeorieënteerde talen, afgeleid van C + +
Ada ➔ oorspronkelijk imperatieve taal met veel objectgeoriënteerde kenmerken

In deze paragraaf benaderen we een programma vanuit het imperatieve paradigma!

Declaratieve statements = terminologie wordt gedefinieerd die verderop in het programma zal worden gebruikt
Imperatieve statements = stappen worden in de onderliggende algoritmen beschreven
Commentaren = verbeteren leesbaarheid v/e programma

2.1 Variabelen en gegevenstypen

Variabele = naam die men gebruikt ipv numerieke adressen om te verwijzen naar een locatie in het werkgeheugen.
➔ *Wanneer de waarde opgeslagen in de locatie wijzigt, wijzigt ook de waarde gekoppeld aan de naam*

Gegevenstype = beschrijft de manier waarop het gegeven gecodeerd is en ook welke bewerkingen kunnen uitgevoerd worden op dat gegeven

- ⇨ ***Gegevenstype integer*** = verwijst naar numerieke gegevens die bestaan uit gehele getallen, waarop wiskundige bewerkingen kunnen worden uitgevoerd

- ⇨ ***Gegevenstype real (=float)*** = verwijst naar numerieke gegevens die ook andere dan gehele getallen kunnen bevatten (meestal opgeslagen in floating-pointnotatie), waarop ook wiskundige bewerkingen kunnen worden uitgevoerd.

int MaxGewicht = 100 ;

MaxGewicht is een integer en krijgt een initiële waarde van 100

- ⇨ ***Gegevenstype Char*** = verwijst naar gegevens die bestaan uit symbolen, waarop bewerkingen zoals vergelijkingen worden uitgevoerd

- ⇨ ***Gegevenstype Boolean*** = verwijst naar gegevens die alleen de waarde WAAR of NIET WAAR kunnen hebben, waarop bewerkingen kunnen worden uitgevoerd waarmee gevraagd wordt of de huidige waarde W of NW is.

2.2 Gegevensstructuren

Variabele worden in een programma ook vaak gekoppeld aan een ***gegevensstructuur***
➔ *Deze beschrijf de vorm v/d gegevens* (=tekst is lange reeks tekens)

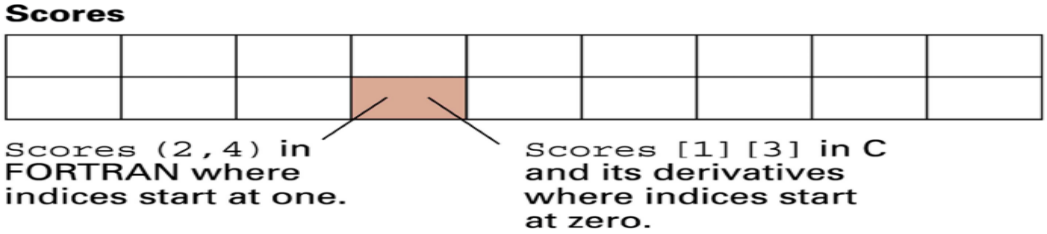
⇨ **Homogene array**

Dit is een blok getallen van hetzelfde gegevenstype!

Taal C ➔ int Scores [2] [9];

Taal FORTRAN ➔ INTEGER Scores (2,9)

Dit verwijst naar een tweedimensionele array met integers van 2 rijen en 9 kolommen



Men kan verwijzen naar de array door te verwijzen naar de naam ervan. Alsook kan men verwijzen naar een component v/d array door ***indices*** te vermelden.
➔ *In C starten indices bij 0*
➔ *In FORTRAN starten indices bij 1*

⇨ **Heterogene array**

Bevat gegevens van mogelijk verschillende typen. Dus zowel Char, integer of Bolean binnen dezelfde array.
Een component binnen een heterogeen array wordt meestal aangeduid met behulp v/d naam v/h array, gevolgd door een spatie en de naam v/d component.

2.3 Constanten en literals

Literal = onveranderlijke, vooraf gedefinieerde waarde die wordt gebruikt in een programma

- ⇨ Ze kunnen echter de betekenis van statements maskeren
- ⇨ Programma dat die gebruikt zal ook veel lastiger aan te passen zijn
  - Vooral als een literal ook nog voor iets anders gebruikt wordt (bv. 200)

Constante = leesbare naam die men koppelt aan bepaalde, onveranderlijke getallen

const int LuchthavenHoogte = 200;

EffectieveHoogte ← Hoogtemeter + 200 **→ 200 = literal**

wordt dan

EffectieveHoogte ← Hoogtemeter + LuchthavenHoogte

2.4 Toekenningsstatements

Met een ***toekenningsstatement*** wordt aan een variabele een waarde toegekend

- ⇨ *Opgeslagen in de geheugenlocatie die gekoppeld is aan de variabele*
- ⇨ Wordt meestal geschreven met de syntax :
  - Variabele, gevolgd door symbool toekenningsbewerking, gevolgd door een expressie voor de toe te kennen waarde
  - **Z = of := of ← X + Y**

Om tegenstrijdigheden bij algebraïsche expressies tegen te gaan heeft men ***rekenregels*** gedefinieerd waarin vastgelegd werd welke bewerkingen voorrang krijgen

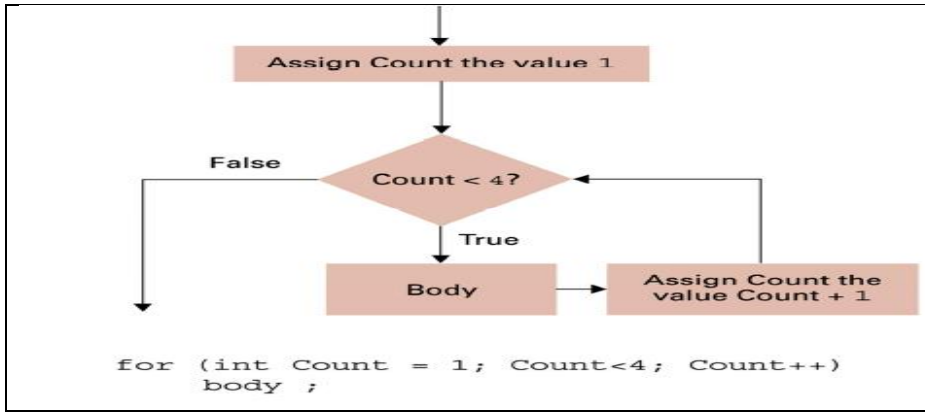
In meeste talen staat '+' voor optellen maar in Java staat dit voor samenvoegen
'abra' + 'cadabra'
Wordt dan abracadabra.
Het verschillende gebruik van een bewerkingsymbool wordt ***overloading*** *genoemd*

2.5 Besturingsopdrachten

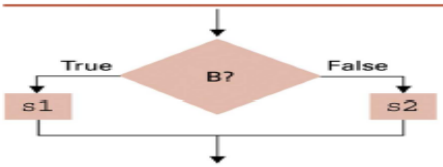
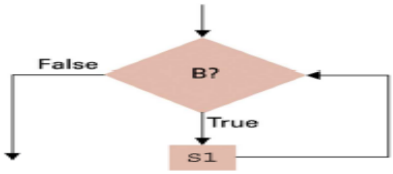
Besturingsopdracht = imperatief statement dat de uitvoeringsvolgorde v/e programma wijzigt

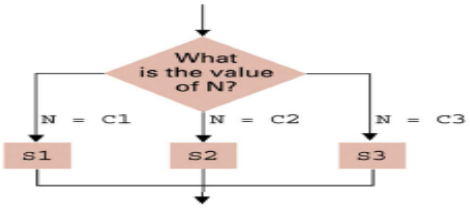
- ⇨ Belangrijkste boosdoener bij deze programmeeropdracht is de opdracht '***goto'*** dat complexe structuren veroorzaakt!

Gestructureerd programmeren = gebruik v/e beperkt aantal mogelijke besturingsopdrachten om duidelijke, goed leesbare programma's te ontwikkelen



De ***for-lusstructuur*** *verschilt v/d terwijl-opdracht omdat de initialisatiestap, de modificatiestap en de eindigingsstap in één statement geïntegreerd zijn*

Control structure	C, C++, C#, and Java	
	`if (B) S1` `else S2;`	**Als-dan-anders structuur**
	`while (B)` `S1;`	**Terwijl-structuur**

	`switch (N)`
	`{ case C1: S1; break;`
	`case C2: S2; break;`
	`case C3: S3; break;`
	`};`
		Case-structuur waarbij men de keuze heeft tussen talloze opties

2.6 Commentaren

Commentaren worden door een vertaalprogramma genegeerd en hebben geen enkele invloed op de uitvoering v/e programma!
Commentaren verhogen enkel de leesbaarheid v/e programma voor de programmeurs!

Er zijn 2 manieren om een commentaar op te nemen in een programma! :

- ⇨ **Men sluit het gehele commentaar in tussen markeringstekens**
  - *\/* dit is een commentaar*/*
- ⇨ **Men markeert enkel het begin v/h commentaar en de ganse regel geldt dan ervoor**
  - *// Dit is een commentaar*

3. Procedurele eenheden

Procedures kunnen worden gebruikt als basistechniek om te komen tot een modulaire representatie v/e programma ***in een imperatieve taal***.
In objectgeoriënteerde talen gebruiken programmeurs procedures om aan te geven hoe objecten moeten reageren op verschillende stimuli.

3.1 Procedures

Procedure = een verzameling instructies voor het uitvoeren v/e taak

- ⇨ *Op het moment dat de procedure uitgevoerd moet worden, wordt de besturing tijdelijk overgedragen aan die procedure (=JUMP), en wanneer die dan gedaan is dan keert de besturing terug naar originele programma-eenheid*

Aanroepen = proces waarbij besturing overgedragen wordt aan procedure

- ⇨ ***Aanroepende eenheid*** is hierbij de programma-eenheid zelf

Een procedure is in veel opzichten een programma in het klein! Er worden ook eerst variabelen gedeclareerd en imperatieve opdrachten gebruikt om de uit te voeren stappen te beschrijven als de procedure aangeroepen wordt!

- ⇨ ***Lokale variabele*** = een variabele die binnen procedure wordt gedeclareerd
- ⇨ ***Globale variabele*** = variabelen die niet beperkt zijn tot een specifiek onderdeel

Procedureheader = opdracht waarmee procedure begint en die naam bevat procedure

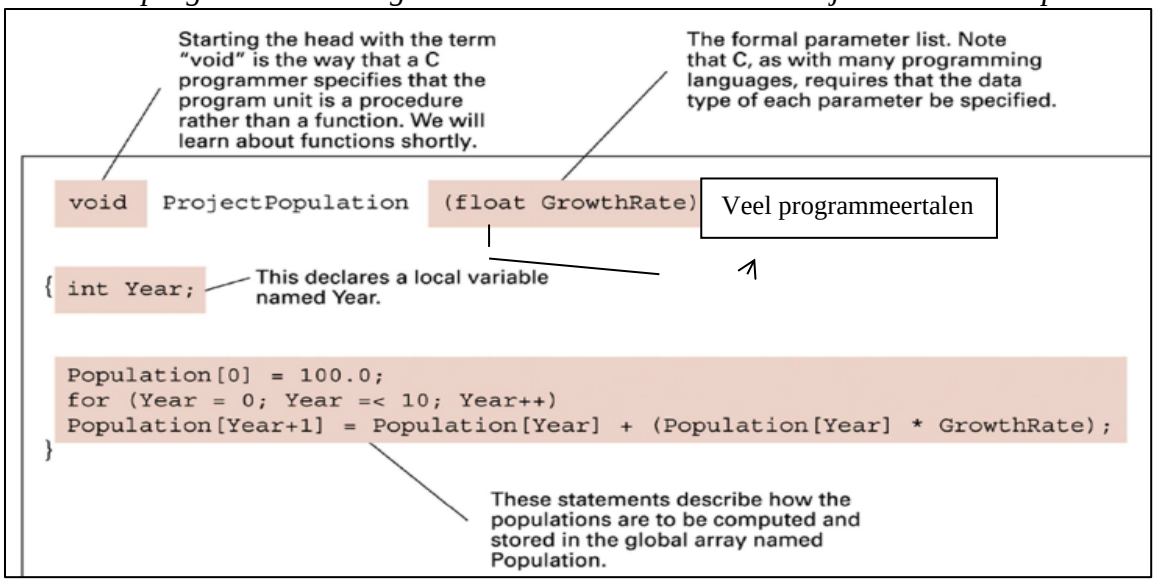
3.2 Parameters

procedure Sorteer (Lijst)

Algemene termen binnen de procedures worden ***parameters*** genoemd

- ⇨ ***Formele parameters*** = parameters die binnen de procedure worden gebruikt (=*Lijst*)
- ⇨ ***Actuele parameters*** = wanneer ze een waarde toegekend krijgen (=*Gastenlijst*)

De meeste programmeertalen eisen dat de formele parameters tussen haakjes in de header van de procedure worden geplaatst.
De meeste programmeertalen gebruiken ook de notatie met haakjes voor actuele parameters.

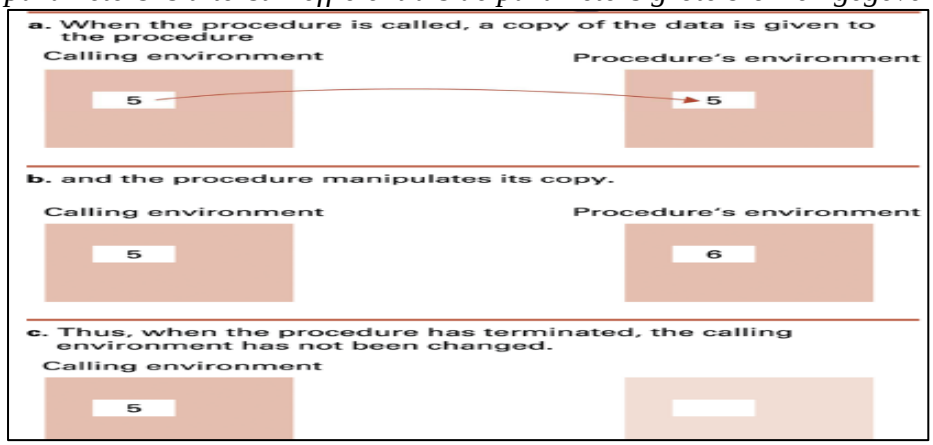


Als er meer dan één parameter is, worden actuele parameters gekoppeld aan de formele parameters in de volgorde zoals ze opgenomen zijn in de header v/d procedure!!

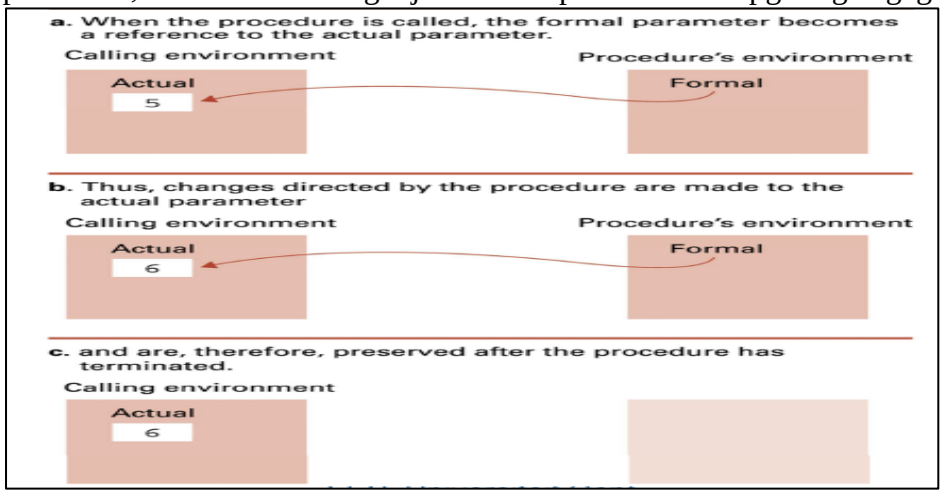
- ⇨ *procedure Overschrijving (Begunstigde, Bedrag)*
- ⇨ *Overschrijving ("Jan Van Dalen", 150)*

Het overdragen van gegevens tussen actuele en formele parameters gebeurt bij elke taal op een andere manier

- ⇨ **Als waarde overdragen (=passed by value)**
  Men maakt een kopie v/d gegevens die worden voorgesteld door de actuele parameters en dragen die vervolgens over aan de procedure. *De gegevens in de aanroepende programma-eenheid worden nooit gewijzigd, maar het als waarde overdragen van parameters is uiterst inefficiënt als de parameters grote blokken gegevens zijn.*



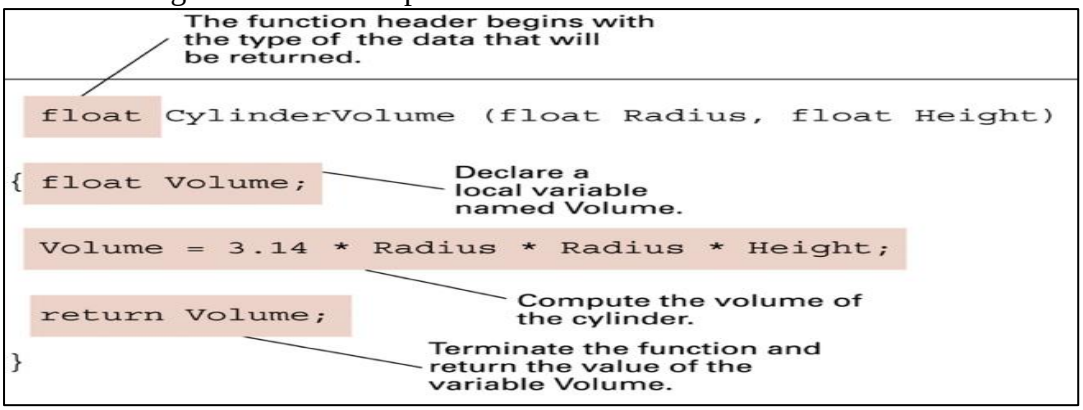
- **Als verwijzing overdragen (=passed by reference)**
  De formele parameter verwijst naar dezelfde werkgeheugenlocatie als de actuele parameter, waardoor het mogelijk is dat de procedure de opgeslagen gegevens wijzigt



3.3 Functies

Procedures worden meestal gebruikt om resultaat in de vorm v/e waarde te verkrijgen en niet zozeer om een bepaalde bewerking uit te voeren.
Functie *verwijst hier naar een programma-eenheid die vergelijkbaar is met een procedure, waarbij een waarde wordt berekend en geretourneerd aan de aanroepende programma-eenheid.*

- Kan opgeslagen worden ***in een variabele of worden gebruikt in berekening***
  - *StreefOmzetJan = GeschatteOmzet (januari) ;*
  - *Als (OmzetLaatsteJan < GeschatteOmzet (Januari))…. anders*
- Functies worden op ongeveer dezelfde manier gedefinieerd als procedures. Het verschil is dat een header v/e functie meestal begint met de specificatie v/h gegevenstype v/h getal dat geretourneerd moet worden, en de definitie v/d functie meestal eindigt met een returnopdracht

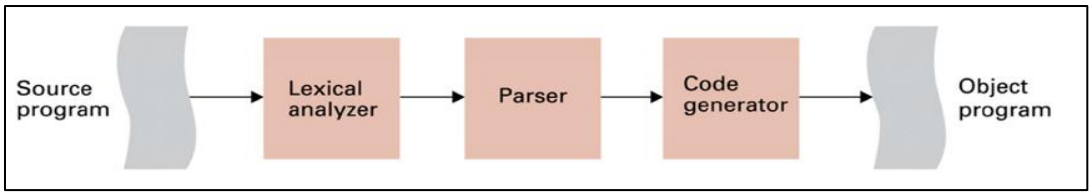


4. De implementatie van talen

4.1 Het vertaalproces

Vertalen = het proces waarmee een programma v/d ene taal wordt omgezet naar de andere
Bronprogramma (= source program) = het oorspronkelijke programma
Doelprogramma (= object program) = de vertaalde versie

Het vertaalproces bestaat uit 3 activiteiten : lexicale analyse *(=door lexicale analyzer),* **parsen** *(=door de parser)* **en genereren van code** *(=door de codegenerator)*

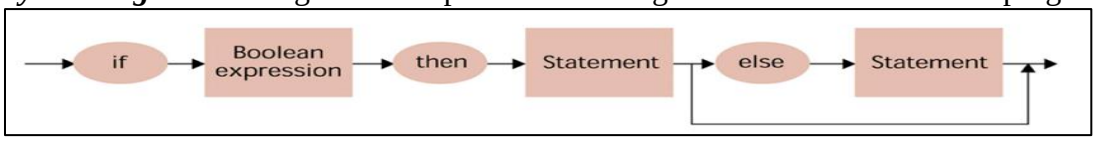


De lexicale analyse *is het proces dat reeksen symbolen in het bronprogramma herkent als behorende bij één entiteit.*
Als elke eenheid geclassificeerd is, genereert de lexicale analyzer een bitpatroon (=token) die de eenheid voorstelt en overhandigt aan de parser.

Parsen *is het herkenen v/d grammaticale structuur v/h programma en de rol van elke component, waarbij de parser de tokens samenstelt tot opdrachten!*

- ⇨ ***Fixed-formattalen*** = werden vroeger gebruikt om parsen te vereenvoudigen door opdrachten in een bepaalde volgorde te gebruiken
- ⇨ ***Free-formattalen*** = de plaats van opdrachten ten opzichte van elkaar is niet kritiek
  - *Free-formattalen gebruiken leestekens om het eind van een statement te markeren en sleutelwoorden om het begin van afzonderlijke zinnen te marken*
    - Sleutelwoorden zijn ***gereserveerde woorden*** die nergens anders mogen gebruikt worden

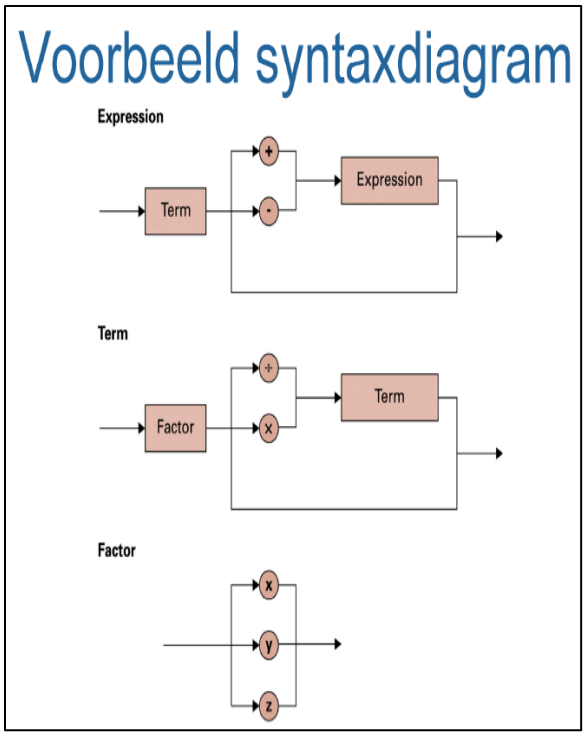
Syntaxdiagrammen = grafische representaties v/d grammaticale structuur v/e programma

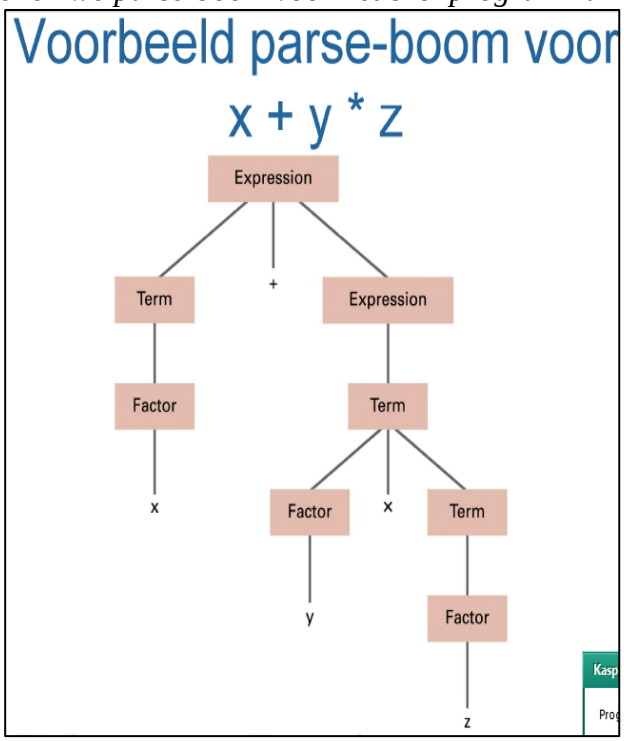


- ⇨ Ronde haakjes slaan op ***terminals***
- ⇨ Vierkante haakjes slaan op ***nonterminals*** (=behoeven nadere beschrijving)

Parse-boom = representatie v/d manier waarop de parser de grammaticale samenstelling v/h programma interpreteert

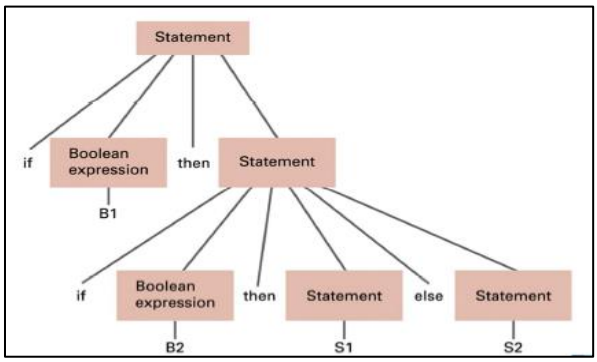
- ⇨ *Parsen is niet anders dan het samenstellen v/e parse-boom voor het bronprogramma*

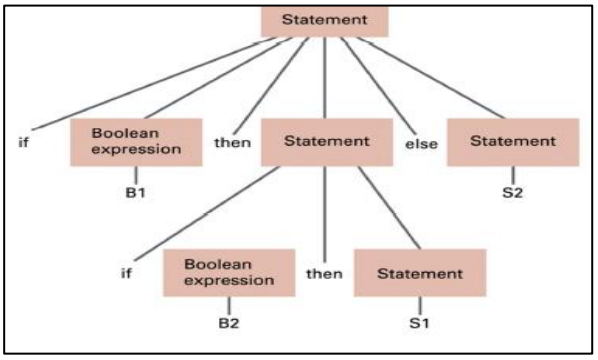




Er zijn echter onvolkomenheden mogelijk bij parse-boom. Stel de volgende opdracht :

Als B1 dan als B2 dan S1 anders S2





Beide interpretaties verschillen significant!!!
Syntaxdefinities van formele programmeertalen moeten twijfelgevallen voorkomen, zoals in dit geval haakjes enorm veel zouden helpen

Als een parser de declaratieve opdrachten in een programma analyseert, wordt de informatie in deze opdrachten in een ***symbooltabel*** opgeslagen. Die symbooltabel zal dus informatie bevatten over de variabelen die gedeclareerd zijn en welke gegevenstypen en gegevensstructuren bij die variabelen horen.
De parser gebruikt die informatie die voor het analyseren van imperatieve opdrachten!

z ← x + y ;

Is hier een imperatieve opdracht waarvan de betekenis van '+' belangrijk is om te achterhalen en hiervoor analyseert men de x en y.
Als x en y bijvoorbeeld niet gelijk zijn (=*zoals integer en real*) kan de parser de codegenerator vragen om een waarde te veranderen van type ➔ **dwang (=coercion)**

De meeste programmeurs keuren dwang af en het resultaat daarvan is dan dat de meeste moderne talen ***strongly typed*** zijn waardoor bewerkingen enkel worden uitgevoerd als de gegevenstypes niet strijdig zijn

Genereren van de code betekent dat de machinetaalinstructies worden samengesteld waarmee de door de parser herkende opdrachten geïmplementeerd worden.

x ← y + z ;
w ← x + z ;

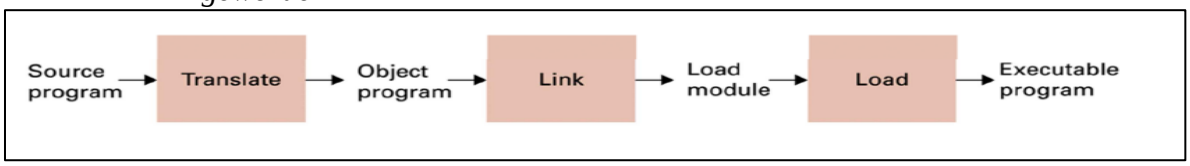
Men zou deze opdrachten als aparte opdrachten kunnen vertalen maar dat zou niet efficiënt zijn. De codegenerator zou dus het inzicht moeten hebben om al te zien dat de 'x' en 'z' al aanwezig zijn. Het implementeren van zulke inzichten heet het ***optimaliseren v/d code***.

De lexicale analyse, het parsen en het genereren v/d code worden niet noodzakelijk in die volgorde uitgevoerd!!
Deze vertaling past mooi in het objectgeoriënteerde paradigma!! (elk object voert zichzelf uit)

4.2 Linken en loaden

Een doelprogramma is in feite een programma in machinetaal met verschillende losse eindjes die eerst nog aan andere doelprogramma's 'geknoopt' moeten worden voordat een daadwerkelijk uitvoerbaar programma ontstaat.

- ⇨ ***De linker*** koppelt doelprogramma's, routines van het besturingssysteem en andere utility software samen tot een volledig, uitvoerbaar programma ***(=een load module***) dat dan vervolgens in het massageheugensysteem wordt opgeslagen.
  - *Tegenwoordig is het linken veelal een onderdeel van het vertaalproces geworden*



- ⇨ ***De loader*** (een onderdeel van de scheduler) plaatst het programma in het werkgeheugen van de computer.

Zodra het linken en loaden voltooid is, kan het programma steeds opnieuw geladen en uitgevoerd worden zonder dat daarvoor de bronversie voor nodig is. *Wijzigingen moeten echter in het bronprogramma worden doorgevoerd!*

4.3 Softwareontwikkelingspakketten

De softwarecomponenten voor het softwareontwikkelingsproces worden meestal gegroepeerd tot een pakket dat als geïntegreerd softwareontwikkelingssysteem fungeert (=***toepassingssoftware***).
Deze pakketten laten de programmeur beschikken over een editor om programma's te schrijven, een vertaalprogramma om programma's om te zetten in machinetaal en verschillende debug-gereedschappen om fouten te vinden

5. Objectgeoriënteerde programmeren

Objectgeoriënteerde paradigma werkt met het ontwikkelen van actieve programma-eenheden die objecten genoemd worden.
Elk van deze objecten bevatten procedures die beschrijven hoe dat object moet reageren op verschillende stimuli.

5.1 Klassen en objecten

Stel een computerspel voor waarbij lasers meteorieten moeten neerknallen!

- ⇨ *In het objectgeoriënteerde paradigma wordt elke laser geïmplementeerd als een object die zelf de resterende vuurcapaciteit bijhoudt en de procedures voor richten en vuren bevat.*
- ⇨ Alle laserobjecten hebben dezelfde eigenschappen gemaakt op basis van hetzelfde sjabloon (=***klasse***)

klasse Laserklasse { **int** ResterendeCapaciteit = 100 ; **void** MikRechts ( ) {.....} **void** MikLinks ( ) {....} **void** Vuren ( ) {.....} }	Een variabele binnen een object wordt een ***instantievariabele*** genoemd (=ResterendeCapaciteit), en de procedures binnen een object ***methoden***.

We kunnen vervolgens drie variabelen declaren als van het 'type' LaserKlasse als volgt:

LaserKlasse Laser1 , Laser2 , Laser3 ;

- ⇨ ***In C + +*** zal deze opdracht drie objecten creëren v/h type LaserKlasse en die koppelen aan de variabelen Laser1, Laser2 en Laser3
- ⇨ ***In Java en C#*** zullen echter enkel de variabelen gecreëerd worden waaraan later waarden kunnen worden toegekend
  - *Voor creatie object 'type' LaserKlasse en koppeling aan Laser1 doet men* : **LaserKlasse Laser1 = new LaserKlasse ( ) ;**

Na creatie en koppeling gaat men verder met het schrijven van opdrachten waarmee de methoden binnen deze objecten geactiveerd kunnen worden:

Laser1.vuren ( ) ;

Laser2.mikLinks ( ) ;

Onderscheid tussen een klasse en een object:

- ⇨ Een klasse is een sjabloon waarmee objecten gecreëerd worden
- ⇨ Een object wordt vaak ***een instantie*** v/d klasse waarmee het werd gecreëerd genoemd

5.2 Constructors

Om objecten specifiek te initialiseren zijn er speciale methoden, ***constructors*** genoemd, binnen de klassen die automatisch worden uitgevoerde wanneer object v/d klasse wordt gecreëerd. Het is een methode met dezelfde naam als de klasse!

```
klasse Laserklasse
 { int  ResterendeCapaciteit  =  100 ;
   LaserKlasse (StartVermogen)
     {ResterendeCapaciteit = StartVermogen;
     }
   void  MikRechts (  )
   {.....}
   void  MikLinks  (   )  {....}
   void  Vuren   (  )
   {.....}
 }
```

In C + +

LaserKlasse Laser1 (50) , Laser2 (100) ;

In Java en C#

```
LaserKlasse Laser1 = new LaserKlasse(50)
LaserKlasse Laser2 = new LaserKlasse(100)
```

5.3 Extra functionaliteit

In objectgeoriënteerde talen is het mogelijk dat een klasse de eigenschappen v/e andere klasse overneemt door gebruik te maken v/e systeem dat ***overerving*** heet.

Class OplaadbareLaser extendes Laserklasse
{ . . .
}

Extends betekent hier dat deze klasse de functies v/d klasse LaserKlasse overerft, maar daarnaast ook de kenmerken bezit die tussen de accolades beschreven zijn.
Overerving maakt het creëren van een hele reeks objecten met vergelijkbare, maar toch verschillende kenmerken gemakkelijker.
➔ **OplaadbareLaser Laser3, Laser4;**

Polymorfisme = het gelijkvormig zijn van de interface van klassen en objecten, maar met verschillende implementaties.
De gelijkvormigheid betreft dan voornamelijk het gebruik en de naamgeving van operaties (of methodes).

- ⇨ *"Polymorfisme komt erop neer dat je een bepaald object (obj2) mag gebruiken op de plaats van een ander object (obj1), zolang obj2 een subtype is van obj1."*

klasse Laserklasse { private **int** ResterendeCapaciteit = 100 ; public LaserKlasse (StartVermogen) {ResterendeCapaciteit = StartVermogen; } public **void** MikRechts ( ) {.....} public **void** MikLinks ( ) {....} public **void** Vuren ( ) {.....} }	***Inkapseling*** = de mogelijkheid om de toegang tot de interne eigenschappen v/e object te beperken ⇨ Kenmerken zijn ingekapseld dan kan alleen object zelf ze benaderen ⇨ Ingekapselde kenmerken zijn ***private kenmerken***, de andere zijn ***public***

6. Concurrerende bewerkingen programmeren

Parallelle verwerking = het simultaan uitvoeren van verschillende versies van hetzelfde programma

- ⇨ *Werkelijke parallelle verwerking is enkel mogelijke met meerdere CPU's*
- ⇨ Met één CPU is het mogelijk een illusie van parallelle verwerking te creëren door de versies de tijd v/d CPU te laten delen
- ⇨ *In Ada noemt men een versie* ***een taak*** *en in Java* ***een thread***

De meest basale bewerking die in een programma voor parallelle verwerking uitgedrukt moet worden, is die waarmee nieuwe processen worden gecreëerd.

- ⇨ Bijna hetzelfde als een traditionele procedure *(*zie 6.3 waar controle wordt overgedragen) *maar bij parallelle verwerking zal de programma-eenheid wel verder worden uitgevoerd terwijl de gevraagd procedure wordt uitgevoerd*

Een complexer aspect van parallelle verwerking is de communicatie tussen verschillende processen

- ⇨ ***Wederzijdse exclusieve toegankelijkheid*** zorgt ervoor dat gegevens door maar één proces gelijktijdig benaderd kunnen worden
- ⇨ Een gegevensitem dat de mogelijkheid heeft om de toegang tot zichzelf te regelen, wordt ***een monitor*** genoemd

7. Declaratief programmeren

Formele logicale levert een algemeen algoritme voor het oplossen v/e probleem en dat kan worden gebruikt om een declaratief programmeersysteem op te bouwen.

7.1 Logische deductie

Resolutie = een deductief redenatieprincipe

- ⇨ Afspraken hierbij dat enkelvoudige opdrachten ***een letter*** zullen zijn en de negatie v/e statement zal '¬' zijn
- ⇨ '→' voor de betekenis van impliceert
  - o *'Kermit is een prins impliceert dat Miss Piggy een actrice is'*

Resolutieprincipe stelt dat :

P OR Q
R OR ¬Q
➔ P OR R

Twee statements lossen op tot het derde statement (=***de resolvent***)

Resolutie is alleen mogelijk bij paren opdrachten in ***clausevorm***. Dat zijn opdrachten waarvan de elementaire componenten verbonden zijn met behulpen v/d Boolean bewerking OR.

P → Q
=
Q OR ¬P

Een verzameling opdrachten wordt ***inconsistent*** genoemd als het onmogelijk is dat alle opdrachten gelijktijdig WAAR zijn. Statement v/d vorm P in combinatie met ¬P is inconsistent en geeft de ***lege clause***!

- ⇨ *Als herhaalde toepassing van resolutie leidt tot de lege clause dan is de oorspronkelijke verzameling opdrachten inconsistent*

P OR Q **R OR ¬Q** **¬R** **¬P**
➔ Is inconsistent

Unificatie = het proces waarbij waarden aan variabelen worden toegekend zodat resolutie mogelijk wordt

7.2 Prolog

Prolog (=Programming in Logic) is een declaratieve programmeertaal waarvan het onderliggende algoritme voor het oplossen v/e probleem gebaseerd is op herhaalde resolutie. Programma in Prolog bestaat uit een verzameling initiële opdrachten die het onderliggende algoritme gebruikt voor deductieve redeneringen

- ⇨ Componenten v/d opdrachten worden ***predikaten*** genoemd

Predikaat bestaat uit de naam ervan, gevolgd door een statement tussen haakjes met de argumenten v/h predikaat

ouder (bennie, michiel).

Hieruit leidt men af dat Bennie de vader is van Michiel. Prolog eist dat constanten beginnen met een kleine letter en variabelen met een hoofdletter.

Statements in een Prolog-programma zijn ofwel feiten ofwel regels, die echter beiden worden afgesloten met een punt. ***Een feit bestaat uit één predikaat.***
Een Prolog-regel is een 'impliceert'-statement waarbij ipv '→' een ':-' wordt gebruikt

oud (X) → wijs (X)
wordt
wijs (X) :- oud (X)

Een komma wordt gebruikt om de conjunctie AND voor te stellen.

Prolog-feiten worden meestal gebruikt om specifieke instanties v/e predikaat te benoemen, terwijl regels meestal worden gebruikt om algemene principes te beschrijven!

Hoofdstuk 8 : Gegevensabstracties

1. Basisprincipes van gegevensstructurering

1.1 Eenvoudige gegevensstructuren

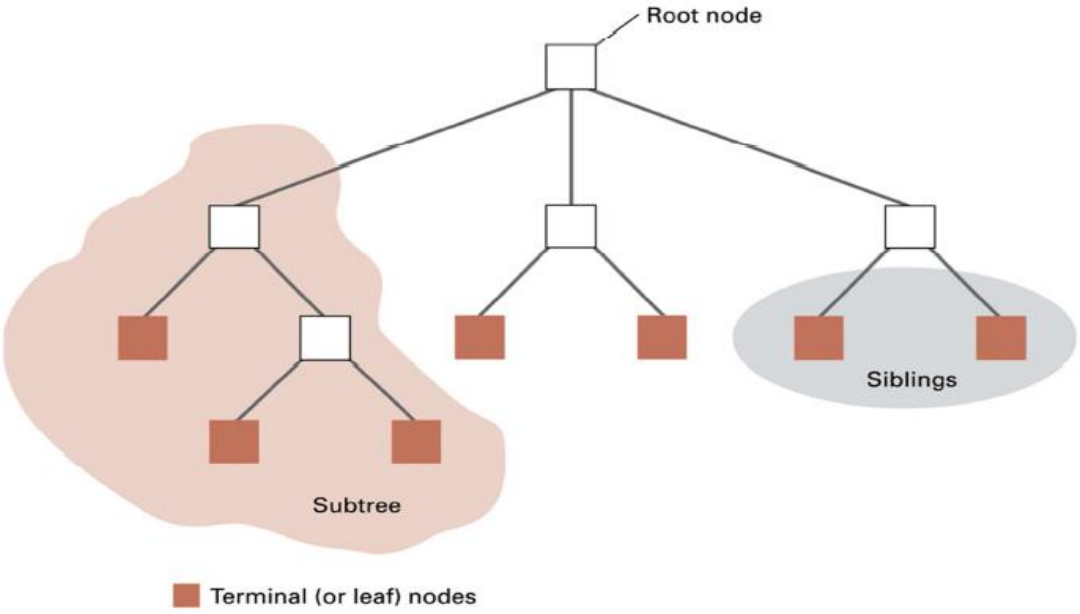
Homogeen array = 'rechthoekig' blok gegevens met items die allemaal van hetzelfde type zijn (=gegevensstructuur)

- ⇨ Tweedimensioneel homogeen array bestaat uit rijen en kolommen (=1ste getal is rijnummer en 2de getal kolomnummer)

Er zijn ook andere gegevensstructuren, namelijk lijsten en bomen

- ⇨ **Lijst** = een verzameling gegevens waarvan de items achter elkaar gerangschikt zijn
  - *Begin van de lijst is de **kop** en het einde is de **staart***
  - Er zijn 2 soorten lijsten, namelijk stacks en wachtrijen
    - ***Stack*** = lijst waarin de items alleen aan de kop toegevoegd of verwijderd worden
      - *Last-in, First-out systeem*
      - Kop v/e stack is de ***top*** en de staart heet de ***onderkant of base***
      - Verwijderen v/e item heet ***poppen*** en invoeren heet ***pushen***
    - ***Wachtrij*** = lijst waarin items aan de kop verwijderd worden, maar aan de staart worden toegevoegd
      - *First-in, First-out systeem*

- ⇨ **Boom** = verzameling gegevens die hiërarchisch gestructureerd is
  - *In dit organogram rapporteert niemand aan twee superieuren!*
  - ***Elke positie in een boom heet een node***
    - Node aan de top heet een ***root node***
    - Node aan het einde v/d structuur heet ***een eindnode***
    - ***Subbomen*** = boom die deel uitmaakt v/e grotere boom
    - *Nakomelingen in de eerste lijn heten **kinderen** en de voorouders in de eerste lijn **ouders***
    - ***Siblings*** zijn nodes met dezelfde ouders
    - ***Binaire boom*** = elke ouder heeft maximaal 2 kinderen
    - ***Diepte v/e boom*** = aantal nodes vanaf root naar eindnode



1.2 Nogmaals abstractie

Werkgeheugen v/e computer is niet georganiseerd in de vorm van stacks, wachtrijen en bomen, maar sequentieel, als een verzameling adresseerbare cellen.
Belangrijk is dat gebruikers (=mens, client, modulaire structuur) de gegevens als abstracte gereedschap kan gebruiken.

1.3 Statische versus dynamische structuren

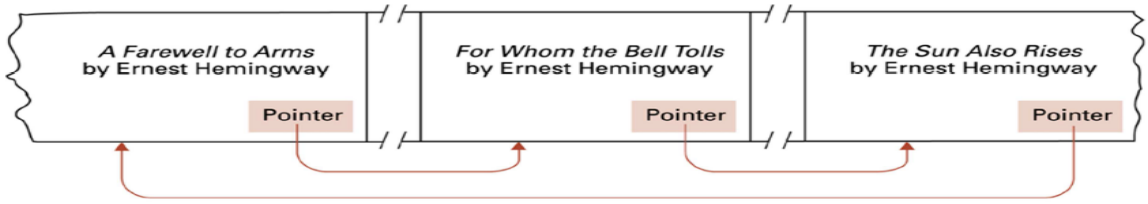
De structuur is dynamisch als ofwel de vorm of wel de structuur veranderlijk is.
Het omgekeerde principe telt voor statische structuren.
Statische structuren kunnen gemakkelijker beheerd worden over het algemeen.

1.4 Pointers

Pointer = opslaggebied dat het geheugenadres bevat waarin een stukje informatie is opgeslagen ➔ Verwijzen dus in zekere zin naar gegevens

- ⇨ *De programmateller wordt ook wel de **instructiepointer** genoemd omdat die wordt gebruikt om het adres v/d volgende uit te voeren instructie in op te slaan*

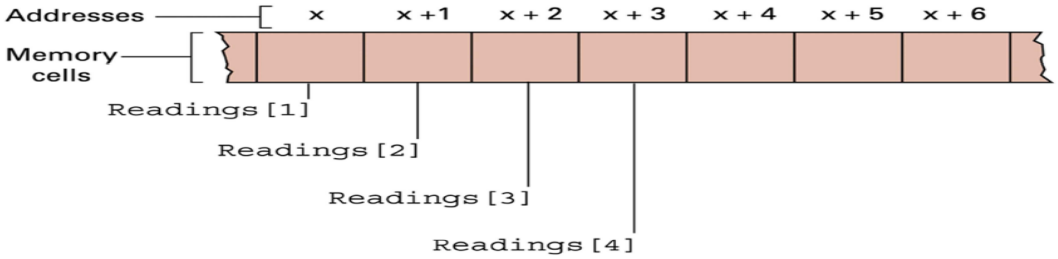
Stel dat we romans alfabetisch willen ordenen.
We reserveren een extra geheugencel binnen een blok cellen die een roman representeert en die cel gebruiken we als pointer naar een ander blok dat een boek v/d dezelfde auteur bevat



2. Gegevensstructuren implementeren

2.1 Homogene arrays opslaan

We slaan metingen op in 24 opeenvolgende geheugencellen
Als het adres van de eerste cel x is dan kunnen we de locatie v/e bepaalde meting berekenen door 1 v/h rangnummer v/d gewenste meting af te trekken en het resultaat op te tellen bij x



Wanneer de vertaler de volgende programmaregel tegenkomt

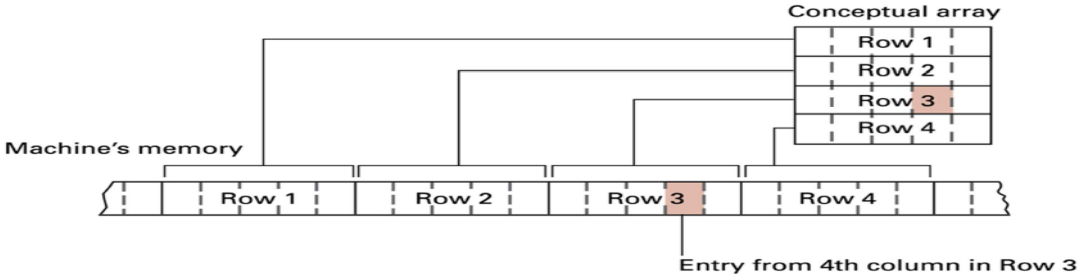
Int Metingen [24]

Die declareert dat de term metingen verwijst naar een eendimensionaal array integerwaarden, dan zal de vertaler ervoor zorgen dat 24 aaneengesloten geheugencellen gereserveerd worden

Metingen [4] ← 67;

Dit verzoek wil de waarde 67 opslaan in het 4de item v/d array

Stel dat we nu werken in een tweedimensioneel array



Om bv. tot het 4de item v/d 3de rij te komen moet men eerst 13 items passeren.
We kunnen hier een formule op stellen. k stelt het aantal kolommen voor, waardoor het adres v/e item in rij i en kolom j als volgt aanduiden :

***x* + (k x (i – 1)) + (j – 1)** ➔ **Adrespolynoom/adresveelterm**

➔ *x* + (5 x (3 -1)) + (4 – 1) = *x* + *13*
Indien men een programmaregel tegenkomt zoals :

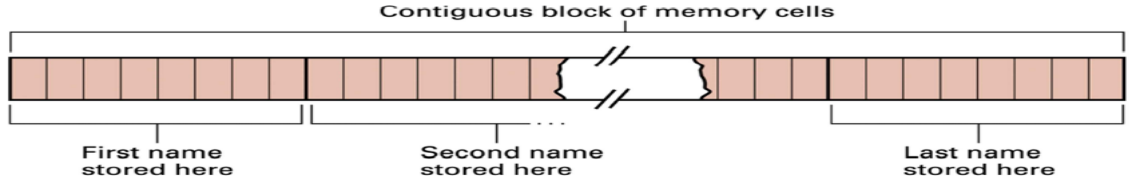
Int omzet [8 , 5];

Dan verwijst naar een tweedimensioneel array met 8 rijen en 5 kolommen en zal men daarvoor 40 aaneengesloten geheugencellen reserveren.
Iets opslaan in de 3de rij en 4de kolom gebeurt dan als volgt :

Omzet [3 , 4] ← 5;

2.2 Lijsten opslaan

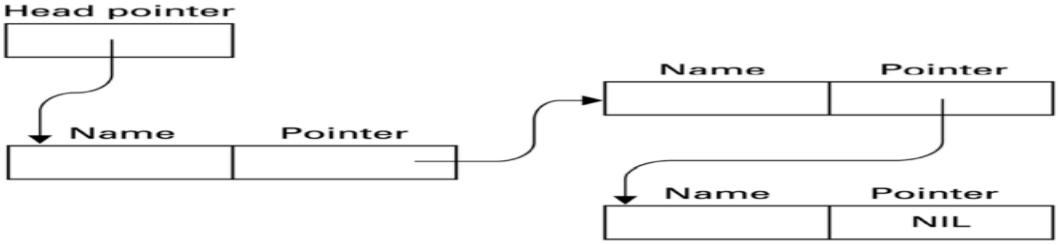
Een methode is om de hele lijst op te slaan in een aaneengesloten rij geheugencellen.
De hele lijst wordt dan opgeslagen in één groot geheugenblok waarbij opeenvolgende items opeenvolgende in de geheugencellen worden opgeslagen ➔ ***Aaneengesloten lijst***



- ⇨ We veronderstellen dat elke naam uit maximaal acht letters bestaat
- ⇨ Voor opslaan v/e lijst met 10 namen hebben we dan 80 geheugencellen nodig
- ⇨ **Deze vorm van werken vormt problemen bij dynamische lijsten!**

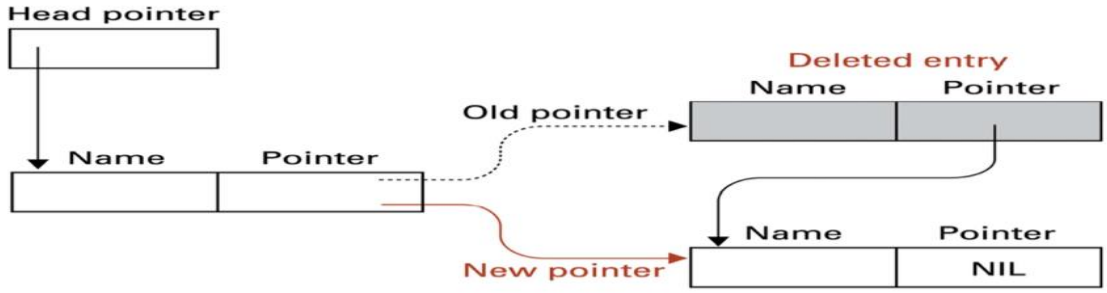
Voor dynamische lijsten kunnen we dan toestaan dat afzonderlijke items in een lijst in verschillende geheugengebieden mogen worden opgeslagen ipv in één groot blok.
We gebruiken het vorige voorbeeld opnieuw maar ditmaal gebruiken we 9 geheugencellen ipv 8 met dit extra geheugencel als pointer naar de volgende naam in de lijst.

- ⇨ ***Een gekoppelde lijst***

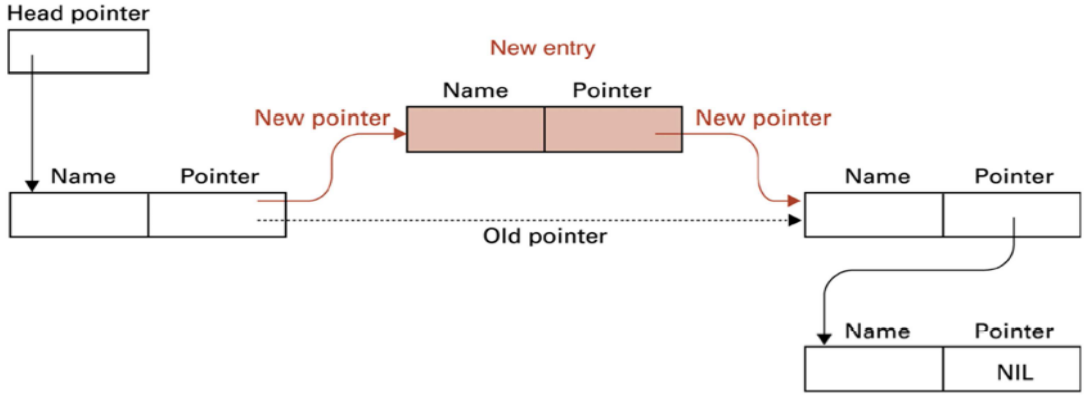


- ⇨ Om te onthouden waar de gekoppelde lijst begint, reserveren we nog een pointer waarin we het adres v/h eerste item opslaan ➔ ***Koppointer***
- ⇨ Om het einde v/e gekoppelde lijst te markeren gebruiken we een ***NIL-pointer***
    - Geeft aan dat er geen volgende items meer zijn in laatste item

Voordeel v/e gekoppelde lijst tegenover aaneengesloten lijst wordt duidelijk wanneer we proberen een item te verwijderen.
Bij een aaneengesloten lijst zou hierdoor een gat ontstaan die gevuld wordt door de andere items op te schuiven. Bij een gekoppelde lijst kunnen we een item verwijderen door een enkele pointer te wijzigen.

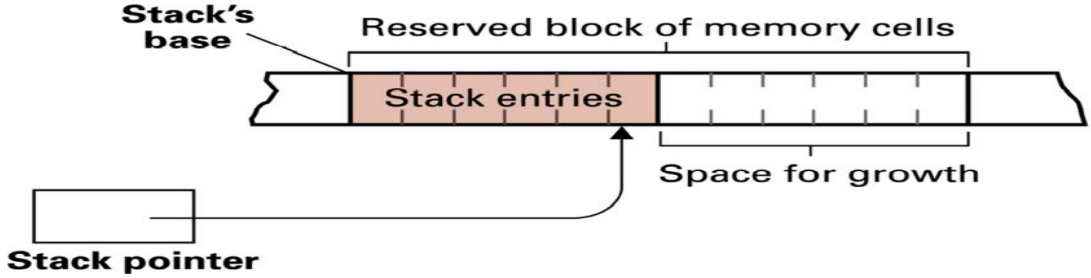


Het invoegen v/e nieuw item in een gekoppelde lijst is wat lastiger. We moeten eerst een ongebruikt blok geheugencellen vinden dat groot genoeg is voor een nieuw item en pointer. Bij opslaan moeten we dan zorgen dat de pointer verwijst naar het juiste volgende item.



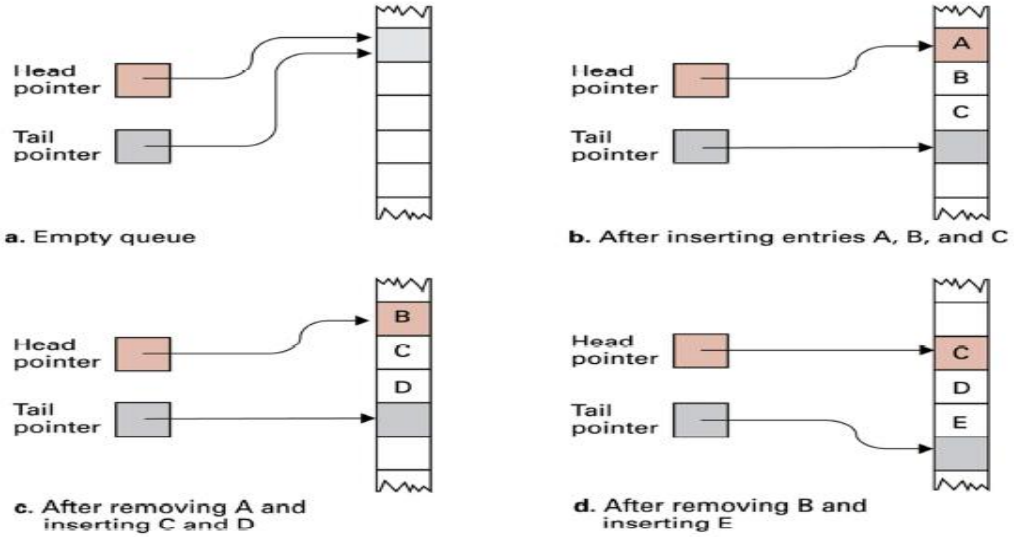
2.3 Stacks en wachtrijen opslaan

Voor het opslaan v/e stack wordt een aaneengesloten blok geheugencellen gereserveerd dat groot genoeg is om ook de groei v/d stack op te vangen.
Om bij te houden waar de top v/d stack zich bevindt, wordt een extra geheugencel, de ***stackpointer***, gebruikt.



- ⇨ Om een nieuw item in de stack te pushen, passen we de stackpointer eerst aan zodat deze verwijst naar de eerstvolgende lege plek aan de top v/d stack, en plaatsen we het nieuwe item op deze locatie

De traditionele implementatie v/e wachtrij is vergelijkbaar met die v/e stack.
Het verschil is dat bij een wachtrij we aan de beide kanten v/d structuur bewerkingen uitvoeren. Daarom moeten twee geheugencellen reserveren voor 2 pointers.
De koppointer houdt de locatie bij v/d kop v/d wachtrij
De staartpointer houdt de locatie bij v/d staart v/d wachtrij (=ongebruikte locatie)
Indien de wachtrij leeg is, verwijzen beide pointers naar dezelfde locatie

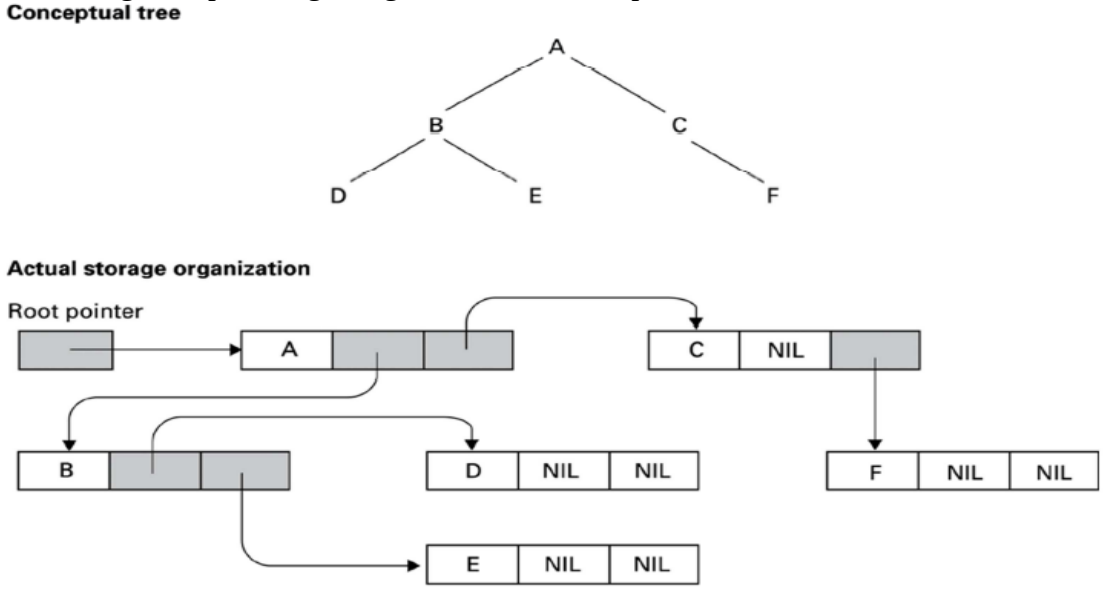


2.4 Binaire bomen opslaan

Binaire bomen worden meestal opgeslagen met behulp v/e gekoppelde structuur waarbij elk item (of node) v/e binaire boom uit drie componenten bestaat :

- ***Het gegeven***
- ***Een pointer naar het eerste kind v/d node (de linker pointer)***
- ***Een pointer naar het tweede kind v/d node (de rechter pointer)***

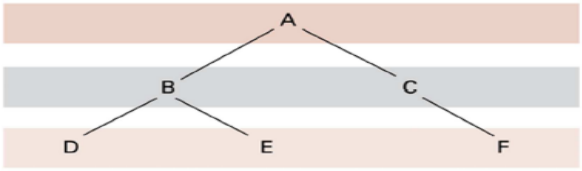
Elke pointer moet verwijzen naar een linker of rechter kind of de waarde NUL (=NIL) als er geen andere nodes meer zijn in die richting v/d boom.
Eindnode is zo herkenbaar omdat de beide pointer NIL-pointers zijn.
Er is ook nog een speciale geheugencel voor de rootpointer voor het adres v/d rootnode.



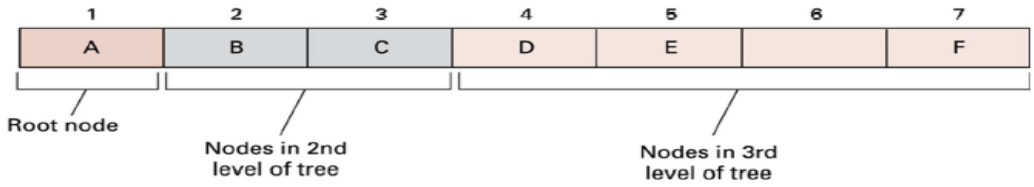
Men kan ook een binaire boom als een gekoppelde structuur opslaan op een andere manier. Men gebruikt een enkel aaneengesloten blok geheugencellen voor de complete boom. De rootnode wordt in de eerste cel v/h blok opgeslagen.

- Nodes hiërarchisch, per niveau sequentieel opslaan
  - Rootnode in 1ste cel van het blok
  - Voor een node in de *n*-de cel van het blok
    - Linker kind in 2*n* - de cel
    - Rechter kind in 2*n*+1 – ste cel
- Ongebruikte locaties worden gemarkeerd met een default bitpatroon

Conceptual tree



Actual storage organization



Dit systeem biedt een efficiënte methode om de ouder of sibling v/e willekeurige node te vinden.

2.5 Gegevensstructuren bewerken

Om de gebruiker in staat te stellen om de structuur als een abstract gereedschap te benaderen, moeten we de gebruiker afschermen v/d complexiteit v/h feitelijke opslagsysteem.

Voor bv. de naam H.M. Meer in te voegen bij klas die cursus fysica 208 volgt :

Invoegen ("Meer, H.M." , Fysica208)

In een ander voorbeeld proberen we een gekoppelde lijst met namen af te drukken. Deze procedure veronderstelt dat een koppointer verwijst naar het eerste item v/d lijst en dat elk item in de lijst bestaat uit twee delen : een naam en een pointer naar het volgende item.

Wanneer de procedure is ontwikkeld, kan men deze als abstract gereedschap gebruiken. Een gebruiker hoeft dus alleen volgende procedure aan te roepen om resultaat te krijgen :

LijstAfdrukken (Economie301)

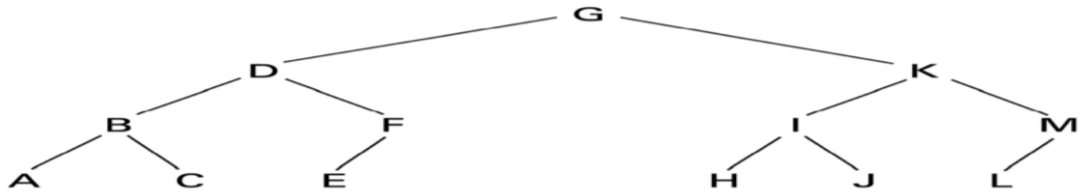
```
procedure PrintList (List)
CurrentPointer ← head pointer of List.
while (CurrentPointer is not NIL) do
   (Print the name in the entry pointed to by CurrentPointer;
    Observe the value in the pointer cell of the List entry
     pointed to by CurrentPointer, and reassign CurrentPointer
     to be that value.)
```

3. Een beknopt praktijkgeval

- We wensen een alfabetisch gesorteerde lijst met items bij te houden
- Volgende bewerkingen moeten mogelijk zijn
  - Zoeken naar de aanwezigheid van een item
  - Afdrukken van de lijst
  - Een nieuw item aan de lijst toevoegen
- Verdere specificaties
  - De lijst is dynamisch
  - Lijst moet binair doorzocht kunnen worden

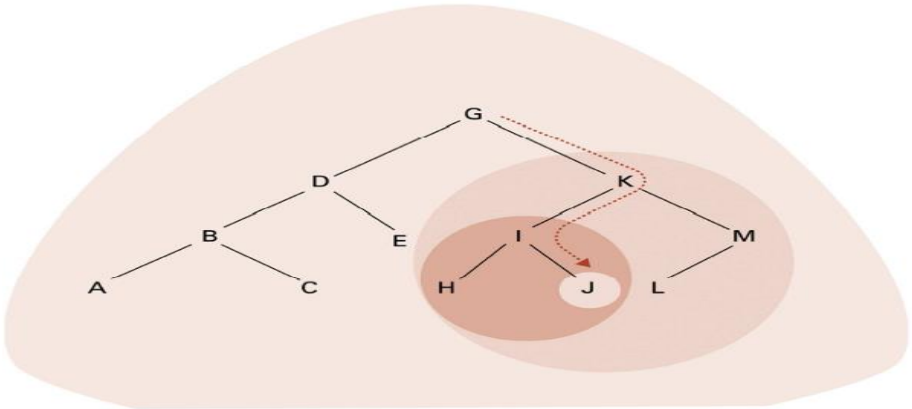
Als de lijst opgeslagen wordt volgens het gekoppelde lijstmodel, moeten we de lijst sequentieel doorzoeken, wat inefficiënt kan zijn bij lange lijsten.
We zullen daarom een implementatie zoeken waarin we het binary search algoritme kunnen gebruiken, waardoor het dus moet mogelijk zijn om het middelste deel te vinden.
We gebruiken daarom een boom waarbij het middelste item in de lijst de rootnode is, het middelste kind v/d eerste helft het linker kind en het middelste kind v/d andere helft het rechterkind van de root.

Voorbeeld: De letters A t.e.m. M alfabetisch gesorteerd in een binaire boom



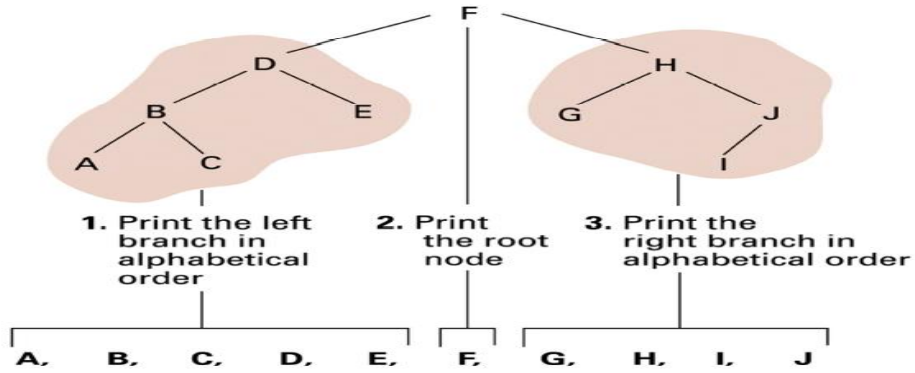
Om de opgeslagen lijst te doorzoeken, vergelijken we de doelwaarde met de rootnode. Als de twee gelijk zijn, is de zoekopdracht voltooid. Als ze niet gelijk zijn, gaan we verder met het linker of rechter kind v/d root, afhankelijk van of de doelwaarde kleiner, respectievelijk groter is dan de root.

```
procedure Search(Tree, TargetValue)

if (root pointer of Tree = NIL)
   then
      (declare the search a failure)
   else
      (execute the block of instructions below that is
       associated with the appropriate case)
       case 1: TargetValue = value of root node
               (Report that the search succeeded)
       case 2: TargetValue < value of root node
               (Apply the procedure Search to see if
                  TargetValue is in the subtree identified
                  by the root's left child pointer and
                  report the result of that search)
       case 3: TargetValue > value of root node
               (Apply the procedure Search to see if
                  TargetValue is in the subtree identified
                  by the root's right child pointer and
                  report the result of that search)
   ) end if
```



Om de lijst in alfabetische volgorde af te drukken, hoeven we alleen de linker subboom in alfabetische volgorde af te drukken, de rootnode af te drukken en ten slotte ook de rechter subboom in alfabetische volgorde af te drukken.

```
procedure PrintTree (Tree)

if (Tree is not empty)
    then (Apply the procedure PrintTree to the tree that
              appears as the left branch in Tree;
          Print the root node of Tree;
          Apply the procedure PrintTree to the tree that
              appears as the right branch in Tree)
```

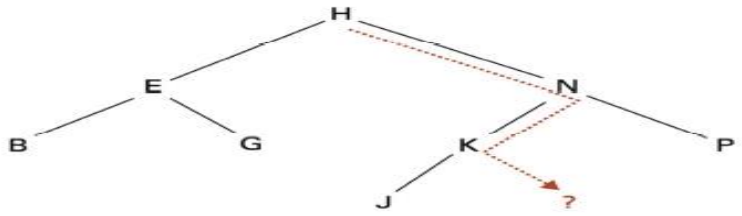


We illustreren nu hoe we een nieuwe waarde moeten invoeren

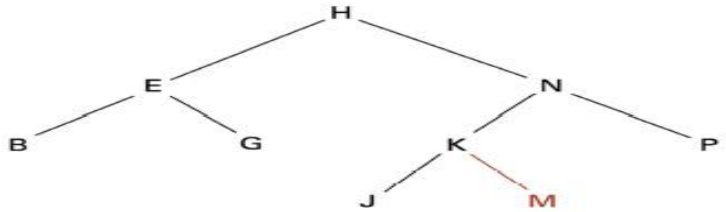
Procedure voor toevoegen

```
procedure Insert(Tree, NewValue)

if (root pointer of Tree = NIL)
  (set the root pointer to point to a new leaf
        containing NewValue)
  else (execute the block of instructions below that is
        associated with the appropriate case)
          case 1: NewValue = value of root node
                  (Do nothing)
          case 2: NewValue < value of root node
                  (if (left child pointer of root node = NIL)
                        then (set that pointer to point to a new
                                  leaf node containing NewValue)
                        else (apply the procedure Insert to insert
                                  NewValue into the subtree identified
                                  by the left child pointer)
           case 3: NewValue > value of root node
                  (if (right child pointer of root node = NIL)
                        then (set that pointer to point to a new
                                  leaf node containing NewValue)
                        else (apply the procedure Insert to insert
                                  NewValue into the subtree identified
                                  by the right child pointer)
        ) end if
```

a. Search for the new entry until its absence is detected



b. This is the position in which the new entry should be attached



4. Aangepaste gegevenstypen

4.1 Door gebruikers gedefinieerde gegevenstypen

In veel moderne programmeertalen is het mogelijk om extra gegevenstypen te definiëren, waarbij de primitieve gegevenstypen als bouwstenen fungeren.

➔ ***Door gebruikers gedefinieerde gegevenstypen***

Stel dat we een programma zouden willen ontwikkelen met talloze variabelen, die echter allemaal dezelfde heterogene structuur hebben die bestaat uit naam, leeftijd en een functiecode.
Men zal deze heterogene structuur als een nieuw gegevenstype definiëren en dit nieuwe type gebruiken alsof het een primitief gegevenstype was :

De variabelen medewerker1 en medewerker2 zullen dan beiden verwijzen naar een blok geheugencellen die de naam, leeftijd en functiecode code v/d medewerkers bevatten.
Naar deze afzonderlijke items verwijzen we dan door Medewerker.naam

- ⇨ De variabelen medewerker1,… zijn dan ***instanties*** v/h gedefinieerde gegevenstype

4.2 Abstracte gegevenstypen

Een compleet gegevenstype bestaat uit twee delen :

- Een vooraf gedefinieerd opslagsysteem
- Een verzameling vooraf gedefinieerde bewerkingen

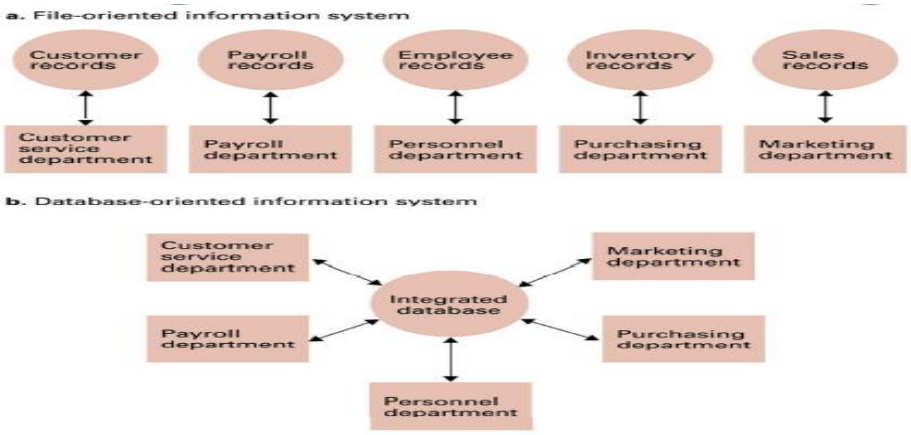
Door gebruikers gedefinieerde gegevenstypen kunnen alleen maar nieuwe opslagsystemen definiëren en bieden niet de mogelijkheid om ook bewerkingen op data met deze structuren te definiëren.
We kunnen dit probleem oplossen door de definitieopdracht v/h gegevenstype uit te breiden met procedures en gegevensomschrijvingen.
Abstracte gegevenstypen = *door gebruikers gedefinieerde gegevenstypen die definities van bewerkingen omvatten*

Hoofdstuk 9 : Databasesystemen

1. Basisprincipes van databases

Database = verzameling gegevens met verschillende dimensies en dus verschillende perspectieven om te bekijken
Plat bestand = traditioneel bestandssysteem dat een eendimensioneel opslagsysteem is

- ⇨ *Studentengegevens opgeslagen in een flat file op alfabetische volgorde, deze gegevens kunnen later enkel maar teruggevonden worden op basis v/d naam. Studentengegevens opgeslagen in een database, kunnen teruggevonden worden op basis van eender welk kenmerk van een student waarvoor gegevens bewaard worden (vb. naam, stamnummer, studierichting, studieresultaten, …).*



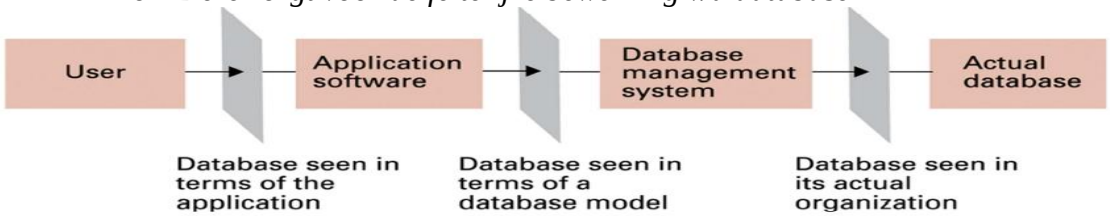
1.1 De rol van schema's

Vroeger werd veel informatie binnen een organisatie op allerlei verschillende plekken opgeslagen. Databasesystemen integreerden al deze informatie op 1 locatie.
Een nadeel hier is echter de mogelijkheid dat vertrouwelijke gegevens toegankelijk kunnen blijken te zijn voor onbevoegden.
Om verschillende gebruikers de mogelijkheid te bieden om de verschillende informatie binnen een database te gebruiken, werken databasesystemen vaak met schema's en subschema's.
Schema = omschrijving v/d hele databasestructuur die de databasesoftware gebruikt om de database te beheren
Subschema = omschrijving v/h deel v/d database dat een bepaalde gebruiker nodig heeft

1.2 Databasemanagementsystemen (DBMS)

Meestal is bij een databasetoepassing sprake van twee softwarelagen :

- ***Een applicatielaag***
  - *Deze bepaalt de externe kenmerken v/h complete systeem*
- ***Een databasemanagementlaag***
  - *Deze zorgt voor de feitelijke bewerking v/d database*



Gedistribueerde database = de database is verspreid over veel computers
Gegevensonafhankelijkheid = de structuur v/d database zelf kan gewijzigd worden, zonder dat daarvoor ook de toepassingssoftware gewijzigd moet worden

1.3 Databasemodellen

Een DBMS bevat routines die opdrachten in termen v/e conceptueel beeld v/d database vertalen naar bewerkingen die het feitelijke gegevensopslagsysteem nodig heeft.
Dit conceptuele beeld v/e database wordt een ***datamodel*** genoemd.

Bij het relationele databasemodel bestaat het conceptuele beeld v/d database uit een verzameling tabellen die bestaan uit rijen en kolommen.

- Het DBMS schermt de interne structuur van de database af voor de toepassingsprogramma's..
- ..door middel van een databasemodel

2. Het relationele model

Dit model stelt de gegevens voor in rechthoekige tabellen, genaamd ***relaties***.

Empl Id	Name	Address	SSN
25X15	Joe E. Baker	33 Nowhere St.	111223333
34Y70	Cheryl H. Clark	563 Downtown Ave.	999009999
23Y34	G. Jerry Smith	1555 Circle Dr.	111005555
.	.	.	.
.	.	.	.
.	.	.	.

- ⇨ Een rij in een relatie wordt ***een tupel*** genoemd
- ⇨ Kolommen in een relatie heten ***attributen***

2.1 Aspecten v/h relationele model

De eerste stap in het ontwerpen v/e relationele database is om de relaties waaruit de database bestaat vorm te geven.
Stel bv. dat we aan de vorige afbeelding wat extra informatie willen toevoegen. Men kan dit door extra kolommen toe te voegen.

Empl Id	Name	Address	SSN	Job Id	Job Title	Skill Code	Dept	Start Date	Term Date
25X15	Joe E. Baker	33 Nowhere St.	111223333	F5	Floor manager	FM3	Sales	9-1-2002	9-30-2003
25X15	Joe E. Baker	33 Nowhere St.	111223333	D7	Dept. head	K2	Sales	10-1-2003	*
34Y70	Cheryl H. Clark	563 Downtown Ave.	999009999	F5	Floor manager	FM3	Sales	10-1-2002	*
23Y34	G. Jerry Smith	1555 Circle Dr.	111005555	S25X	Secretary	T5	Personnel	3-1-1999	4-30-2001
23Y34	G. Jerry Smith	1555 Circle Dr.	111005555	S26Z	Secretary	T6	Accounting	5-1-2001	*
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.

- ⇨ Er ontstaan meerdere problemen zoals het ontbreken van efficiëntie door ***redundantie***.
  - *Als een medewerker promotie heeft gemaakt en daardoor meerdere functies heeft gehad moeten meerdere tupels in de nieuwe relatie dezelfde informatie bevatten*
- ⇨ Een ander probleem is wat er gebeurt als er informatie uit de database wordt verwijdert. Het weglaten v/e tupel kan het ongewenste neveneffect hebben dat ook andere gegevens verloren gaan

De oorzaken van deze problemen is dat meer dan één concept hebben gecombineerd in één enkele relatie. We zouden onze problemen kunnen oplossen door het systeem met 3 relaties opnieuw te ontwerpen. (=***decompositie zonder gegevensverlies of met***)

EMPLOYEE relation

Empl Id	Name	Address	SSN
25X15	Joe E. Baker	33 Nowhere St.	111223333
34Y70	Cheryl H. Clark	563 Downtown Ave.	999009999
23Y34	G. Jerry Smith	1555 Circle Dr.	111005555
.	.	.	.
.	.	.	.
.	.	.	.

JOB relation

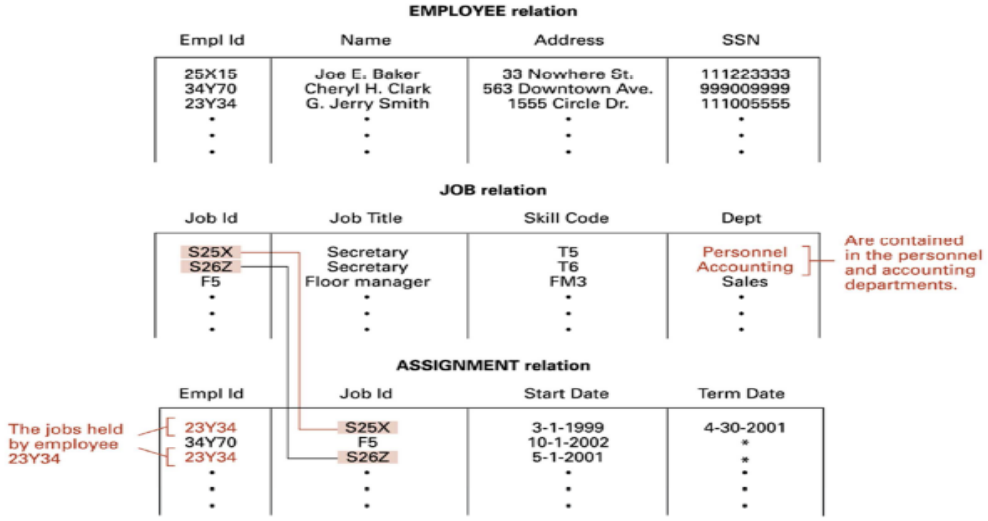
Job Id	Job Title	Skill Code	Dept
S25X	Secretary	T5	Personnel
S26Z	Secretary	T6	Accounting
F5	Floor manager	FM3	Sales
.	.	.	.
.	.	.	.
.	.	.	.

ASSIGNMENT relation

Empl Id	Job Id	Start Date	Term Date
23Y34	S25X	3-1-1999	4-30-2001
34Y70	F5	10-1-2002	*
23Y34	S26Z	5-1-2001	*
.	.	.	.
.	.	.	.
.	.	.	.

- ⇨ *De koppeling tussen de WERKNEMER en JOB relaties komt tot stand via de ASSIGNMENT relatie*
- ⇨ Het ontleden v/e relatie in kleinere relaties kan ***in sommige gevallen*** leiden tot verlies van informatie!

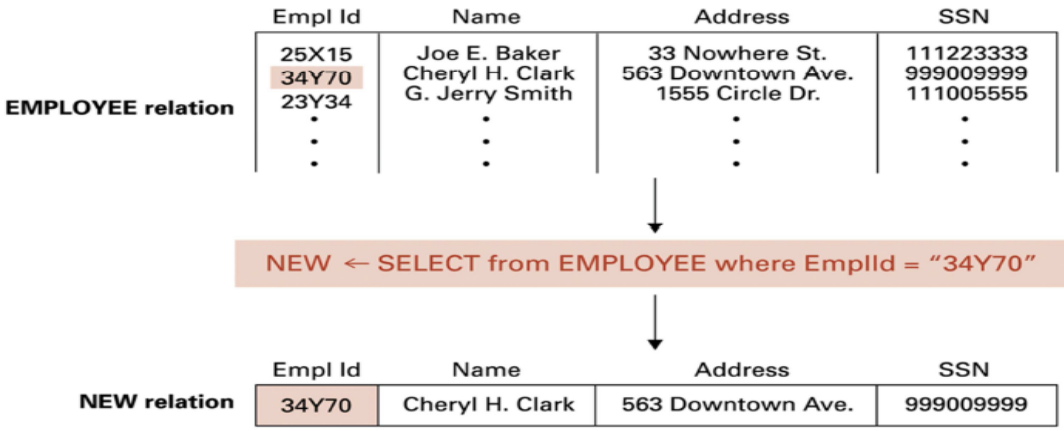
Voorbeeld: zoeken van afdelingen voor een medewerker



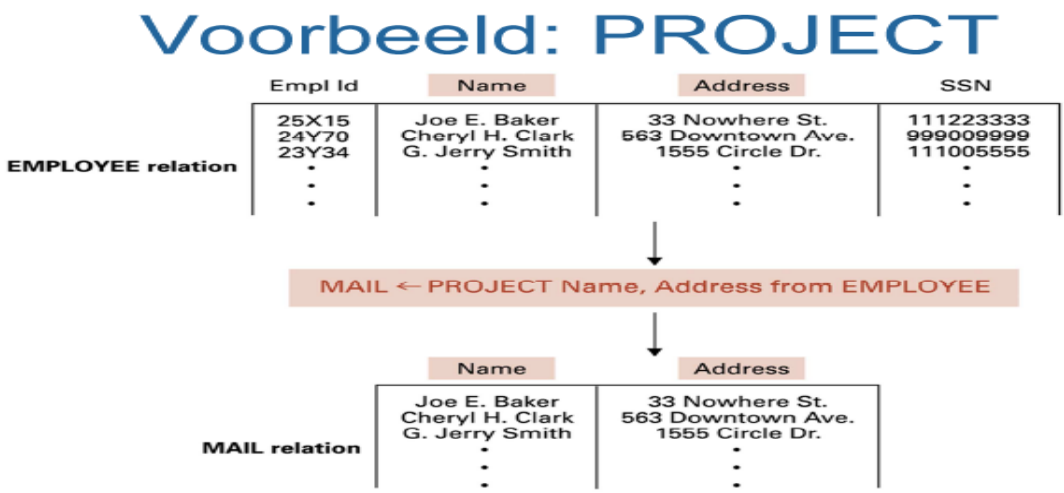
2.2 Relationele bewerkingen

- Hoe informatie uit een relationele database halen?
  - SELECT-bewerking
    - Tupels uit een relatie selecteren
  - PROJECT-bewerking
    - Relatie projecteren op één of meerdere kolommen
  - JOIN-bewerking
    - Twee relaties combineren tot één relatie
    - Veelal om er vervolgens SELECT en/of PROJECT op toe te passen

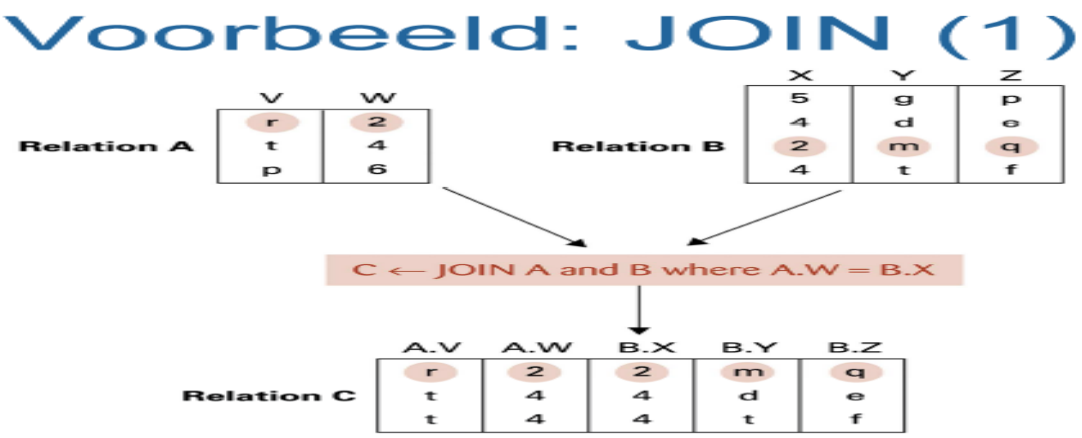
Voorbeeld: SELECT



De betekenis van deze opdracht is : maak een nieuwe relatie met de naam NIEUW, die de tupels bevat uit de relatie MEDEWERKER, waarvan het Mdwnr-attribuut gelijk is aan 34Y70



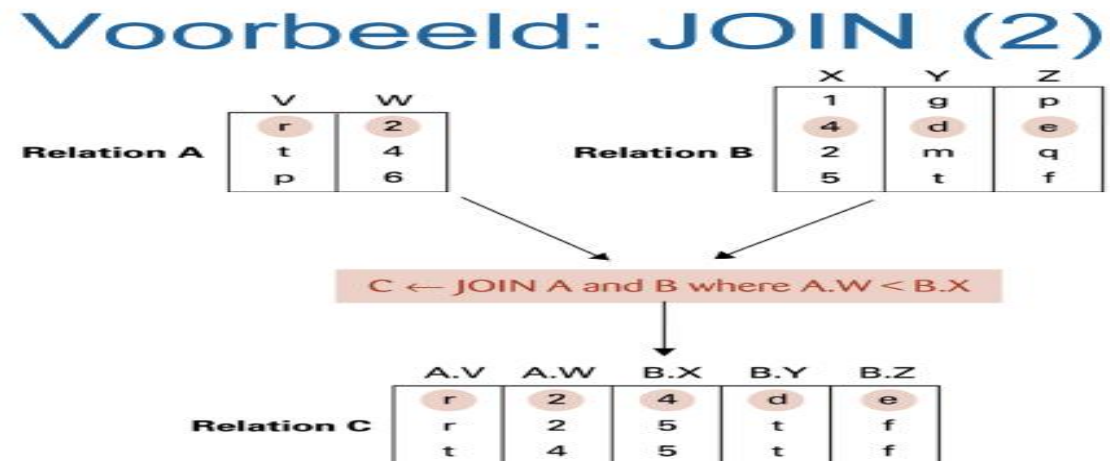
In tegenstelling tot de SELECT-bewerking, die rijen uit een relatie selecteert, selecteert de PROJECT-bewerking kolommen.
Dit betekent dat een nieuwe relatie (MAIL) gecreëerd wordt met voor elke tupel uit EMPLOYEE enkel de gegevens van de kolommen naam en adres.



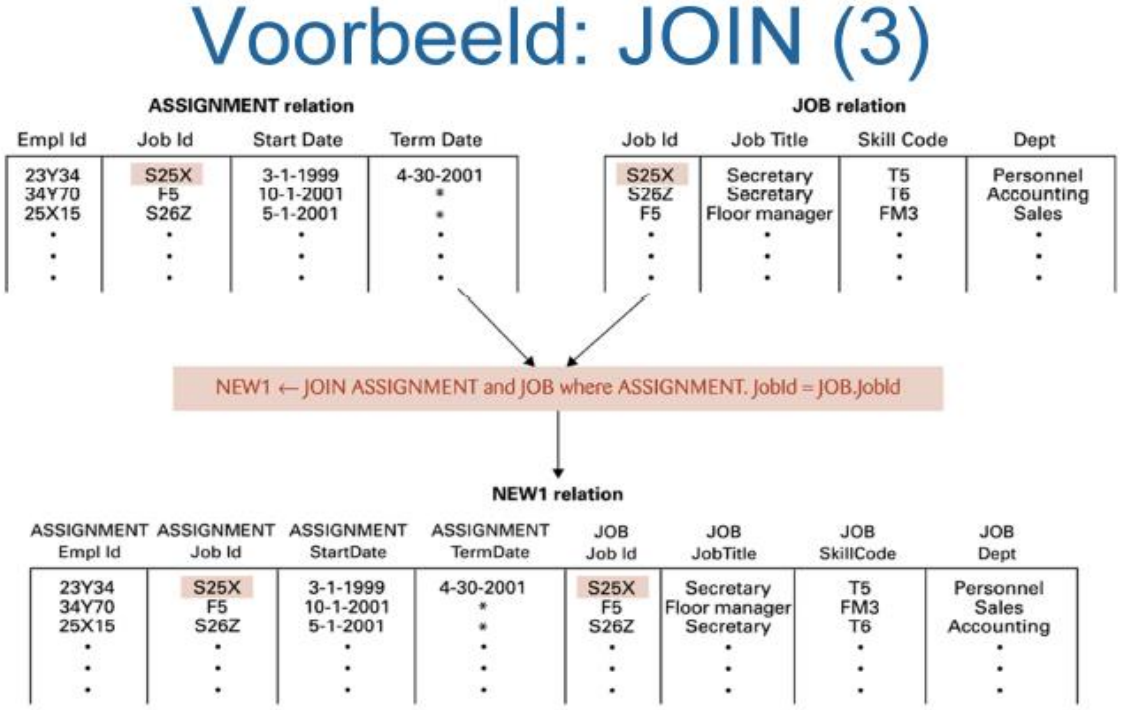
Met een JOIN-bewerking kunnen we twee verschillende relaties combineren tot één relatie

De tupels v/d nieuwe relatie worden gemaakt door de tupels v/d twee oorspronkelijke relaties ‘aan elkaar te plakken’.
Welke tupels worden samengevoegd tot tupels in de nieuwe relatie, wordt bepaald door conditie waaronder JOIN-bewerking (=hier A.W = B.X) gemaakt werd.

- ⇨ ***(p,6) en (5,g,p) voldoen hier niet aan de conditie!***



Proberen we nu met behulp v/e JOIN-bewerking om in de database een lijst te halen met personeelsnummers en de bijbehorende afdeling waar de werknemer werkt.
De informatie is over meer dan één relatie verspreid opgeslagen, waardoor we voor het ophalen zowel SELECT-bewerkingen als PROJECT-bewerkingen nodig zullen hebben.



Met de gevormde relatie kunnen we het probleem oplossen door eerst met een SELECT-bewerking de tupels te selecteren waarin LOOPBAAN.Uitdienst gelijk is aan * (=betekenis is ‘in dienst’)

NIEUW2 ← SELECT from NIEUW1 where LOOPBAAN.Uitdienst = “*”

Vervolgens voert men een PROJECT-bewerking uit op de attributen LOOPBAAN.Mdwnr en FUNCTIE.Afd

LIJST ← PROJECT LOOPBAAN.Mdwnr , FUNCTIE.Afd from NIEUW2