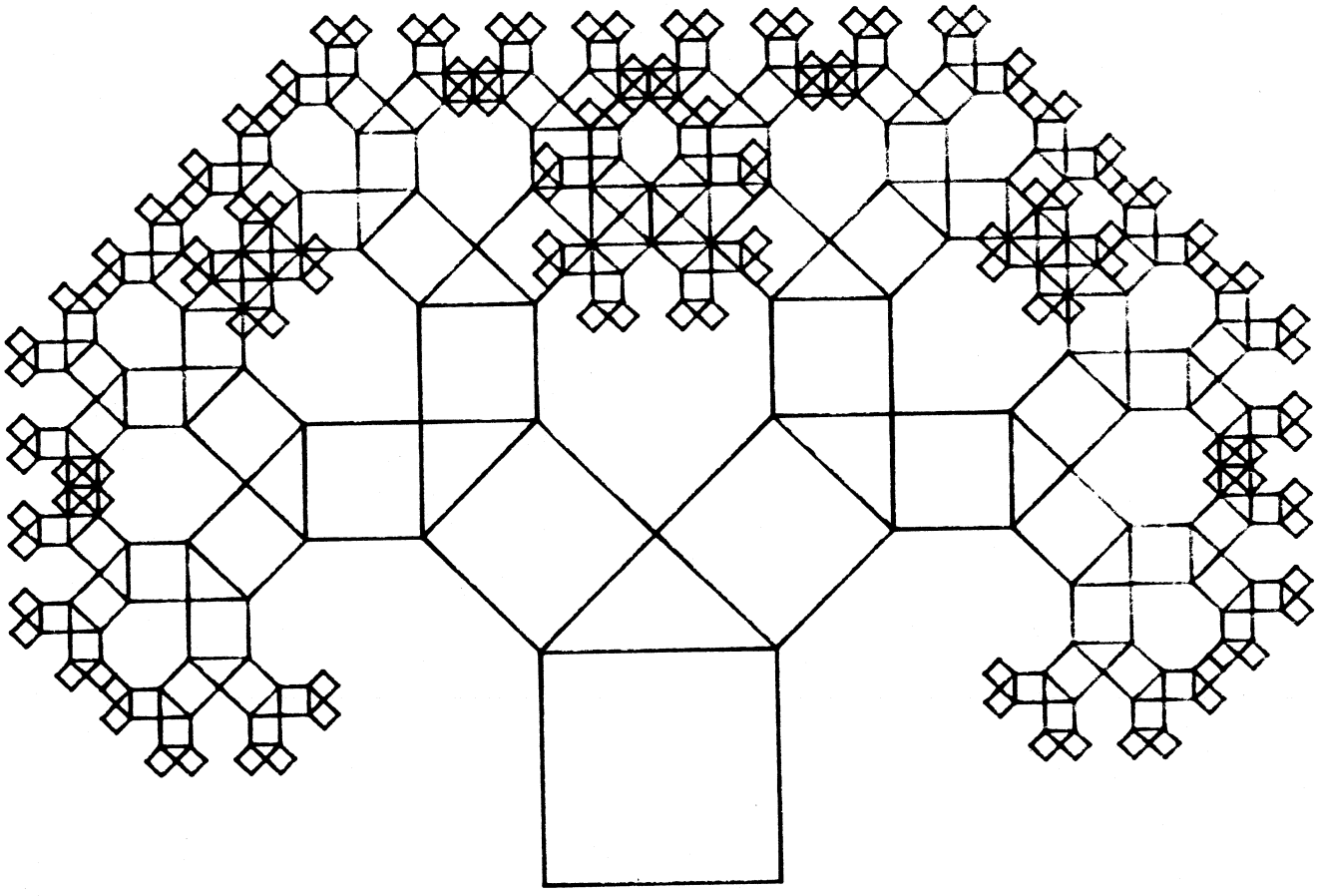
DATASTRUCTUREN

NAJAAR 1991

H.J. Hoogeboom



AFDELING WISKUNDE EN INFORMATICA
RIJKSUNIVERSITEIT LEIDEN

Mag. Uitgifte no. 61

VI. LIJSTEN

LITERATUUR

Er zijn veel geschikte engelstalige boeken over het onderwerp datastructuren, soms met nadruk op datastructuren en hun eigenschappen, soms meer nadruk op algoritmen. We noemen er een aantal van; de meeste zijn via de docent in te zien.

E. Horowitz, S. Sahni. *Fundamentals of Data Structures in Pascal.* Computer Science Press, 1989.

D.F. Stubbs, N.W. Webre. *Data Structures, with Abstract Data Types and Pascal.* Brooks/Cole Publishing Company, 1989 (tweede editie).

J.H. Kingston. *Algorithms and Data Structures.* Addison Wesley, 1990.

A.V. Aho, J.E. Hopcroft, J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1987 (herdruk met correcties).

K. Mehlhorn. *Data Structures and Algorithms.* (drie delen)
1. Sorting and Searching, 2: Graph Algorithms and NP-Completeness. Springer Verlag, 1984.
Tamelijk wiskundig van opzet, uitgebreide analyse van de efficiëntie.

D.E. Knuth. *The Art of Computer Programming.* (drie delen)
1: Fundamental Algorithms, 3: Sorting and Searching. Addison-Wesley.
Het klassieke naslagwerk; het wachten is op een nieuwe revisie.

Niet speciaal geschikt om datastructuren uit te leren, maar de moeite waard:

S. Baase. *Computer Algorithms, Introduction to Design and Analysis (2nd edition).* Addison-Wesley.
Gebruikt bij het college Analyse van Algoritmen. Geen nadruk op datastructuren. Beschrijft wel een aantal van de onderwerpen van dit college: oa. graaf-algoritmen, string matching, dynamisch programmeren.

G. Brassard, P. Bentley. *Algorithmics, Theory and Practice.* Prentice Hall, 1988. Eigenlijk gevorderd Algoritmiek boek. Breed scala van onderwerpen die vrij bondig aan de orde komen. Uitdagende opgaven.

R. Sedgewick. *Algorithms (2nd edition).* Addison-Wesley, 1988.
Nadruk op algoritmen, maar bevat daar dan ook veel van. Prima naslagwerk.

0. DATA ABSTRACTIE

0.0. Abstracte Data-Typen

Inleiding. Laten we eens kijken naar de stapel (of *stack*) van integers. Deze datastructuur kan als waarden alle verschillende (eindige) rijtjes van gehele getallen aannemen. Deze waarden zijn opgebouwd uit componenten, de afzonderlijke gehele getallen, volgens een speciale structuur, het rijtje, waarbij we dan ook nog één van de kanten van het rijtje een speciale status geven, nl. de 'bovenkant' van het rijtje.

Op de datastructuur staan we een aantal operaties en testen toe die de stapel kenmerken: de stapel kan leeggemaakt worden, we mogen testen of de stapel leeg is, we mogen het bovenste element bepalen en het bovenste element van de stapel verwijderen (mits de stapel niet leeg is) en tenslotte mogen we natuurlijk een element (hier een geheel getal) aan de bovenzijde van de stapel toevoegen. Behalve het bovenste element kunnen we géén elementen van de stapel direct bekijken of veranderen. Als we dát toelaten spreken we niet meer van een stapel, maar hebben we het over een andere datastructuur.

Merk op dat bovenstaande beschrijving niet op een of andere manier refereert aan de implementatie van de stapel in een programmeertaal. De stapel wordt beschreven zoals we deze voorstellen als we over het gedrag ervan nadenken; het is de beschrijving van een stapel als wiskundig model. Dit leidt tot het begrip abstract datatype.

Definitie. Een datatype bestaat uit een verzameling *waarden* (het domein) en een verzameling *operaties* en *functies*. We spreken van een gestructureerd datatype (of datastructuur) als binnen de waarden *componenten* zijn te onderscheiden die volgens een *interne structuur* zijn gerelateerd. Een niet-gestructureerd datatype heet atomair. ■

De gehele getallen, en ook de reële getallen, vormen een atomair datatype met als bekendste operaties optellen, vermenigvuldigen, negatieve bepalen (unaire min), gelijkheid testen, etc.

Als voorbeeld van interne structuren die we bij datatypen aan kunnen treffen, noemen we de verzameling (hier 'structuur' niet te letterlijk nemen, maar er zijn wél componenten in een verzameling te herkennen), de rij (dwz. lineaire ordening) en de boomstructuur.

Abstractienivo's van een datatype. Met een 'wiskundige' beschrijving van een datastructuur kunnen we wel het gedrag van de structuur leren begrijpen, maar we kunnen er (nog) niet direct mee programmeren. We moeten geschikte

types kiezen (in het geval van de stapel hebben we bijvoorbeeld de keuze tussen een array- of een pointer-representatie) en de operaties omzetten naar procedures.

Intikken van deze procedures heeft uiteindelijk niet veel zin als we geen manier hebben om het programma te 'draaien'. Voor de datastructuren zal plaats in het geheugen gevonden moeten worden, de procedures zullen in machinetaal omgezet moeten worden. Daar hebben we (gelukkig) gereedschap voor beschikbaar: de compiler. In dit college zullen we nauwelijks naar het machinenivo afdalen. Slechts in bijzondere gevallen loont het (in het kader van dit college) de moeite om met behulp van de handboeken elke bit eigenhandig een plaats te geven, nl. daar waar speciale eisen aan snelheid of geheugengebruik worden gesteld.

In de terminologie van Stubbs en Webre krijgen we de volgende nivo's:

abstract: wiskundig concept; beschrijving structuur, eigenschappen en gedrag
↓ waarden *representeren*, operaties *implementeren*
virtueel: hogere programmeertaal; types, procedures en functies
↓ compiler, interpreter
fysiek: machine nivo; bits, machine taal

Stapsgewijs programmeren. Eén van de belangrijke lessen van de vakken Programmeermethoden en Algoritmiek is dat programmeren niet gewoon maar knutselwerk is, maar een proces dat stapje voor stapje naar het einddoel - een programma - toewerkt. (Hoe groter het uiteindelijke programma, hoe meer mensen er aan een project meewerken, des te belangrijker wordt deze observatie. Er is dan ook een heel vak aan het projectmatig ontwikkelen van software gewijd: Software Engineering). We schetsen hier kort waar abstracte datastructuren bij het ontwerpen van programma's van pas komen. Dit doen we door drie fasen te onderscheiden in het omzetten van een probleem naar een programma. (Natuurlijk ..., in de praktijk zijn de grenzen niet zo zwart-wit te trekken, maar het geeft enig houvast.)

i. Informeel algoritme. Analyseer het probleem en probeer stapje voor stapje te komen tot een algoritme. Dit algoritme is nog informeel, evenals de modellen voor de gegevensopslag.
ii. Pseudo-taal programma. Analyseer het informele model voor de gegevensopslag om een indruk te krijgen van de soort gegevens en de meest belangrijke operaties op die gegevens die het algoritme uitvoert. Dit leidt dan tot een verstandige keuze van abstracte datatypen die bij het oplossen van het probleem van pas zullen komen. Kies namen voor datatypen en bijbehorende procedures. Zet het informele algoritme om tot een meer gestructureerd geheel

gebruikmakend van **if..then..else..**, **while..do..** en dergelijke controlestructuren.
iii. Pascal programma. Kies voor elk van de abstracte datatypen een representatie in Pascal, opgebouwd uit de in Pascal aanwezige datatypen en constructoren (dit zijn taalelementen waarmee eigen datatypen geconstrueerd kunnen worden, zoals ARRAY, SET en RECORD). Schrijf voor elk van de operaties van de datatypen een bijbehorende procedure of functie. Werk het pseudo-taal algoritme uit tot een compleet Pascal programma dat de gegevens manipuleert via de gekozen datastructuren.

Illustratie.

▷ Informeel model

een stapel met verschillende kleuren borden

▷ Informeel algoritme

zolang de stapel nog niet leeg is nemen we het bovenste bord, waarmee ...

▷ Abstracte Data Type

Atomair type *Bord*, kleuren Rood, Wit of Blauw
Structuur *Stapel-van-Borden*, met operaties *IsLeeg*, *Push*, *Pop*,

▷ Pseudo-taal programma

```
while not IsLeeg( Stapel )
do   Pop( Stapel, NieuwBord )
     case Kleur( NieuwBord ) of ...
od
```

▷ (Pascal-) datastructuren, *representatie* van waarden

```
TYPE BordTiep = (Rood, Wit, Blauw);
     Wijzer = ^Element;
     Element = RECORD
                    Bord : BordTiep;
                    Onder : Wijzer
               END;
     StapelVanBorden = Wijzer;
```

▷ (Pascal-) data structuren, *implementatie* van operaties

```
PROCEDURE IsLeeg( S : StapelVanBorden ): Boolean;
BEGIN
     IsLeeg := ( S = NIL )
END;    { ISLeeg }
```

Kenmerken van het werken met Abstracte Data Typen. Nogmaals: na het uitbannen van de **goto** en na de "ontdekking" van het stapsgewijze verfijnen van algoritmes (*top-down* aanpak) is het leren werken met (en denken in) ADT's een derde stap op weg naar *gestructureerd programmeren*. Veelgebruikte argumenten voor ADT's zijn in onderstaande lijst opgenomen. Hier en daar overlappen de argumenten elkaar.

• *overzichtelijkheid*: alle relevante declaraties en procedures zijn bij elkaar ondergebracht en dus snel te vinden; gebruik van de datastructuur gaat via

gestandaardiseerde procedure aanroepen en niet via soms lastig te herkennen foefjes (dit betekent niet dat alle lol bij het programmeren verdwenen is; slimmigheid kan elders ingezet worden).
• *scheiding implementatie en gebruik*: is de keuze van de datastructuur eenmaal vastgelegd, dan is gescheiden ontwikkeling van de implementatie van de datastructuur en van de programma's die deze moeten gaan gebruiken mogelijk.
• *modulariteit*: de datastructuur wordt in aparte eenheden geïmplementeerd, deze heten bij Turbo-Pascal *units* (zie verderop), en elders onder andere *package*, *module* of *class*. Voordelen van deze aanpak: de eenheden zijn *overdraagbaar*, dus in meerdere programma's te gebruiken; ze zijn veelal apart te compileren, dus tijdwinst; wijzigingen in de implementatie zijn eenvoudig aan te brengen, meer in het bijzonder:
• *onafhankelijk van de implementatie*: indien door systeemveranderingen of door strengere eisen wat betreft snelheid een andere implementatie gewenst is, kan dit lokaal gebeuren op de plaats waar de datastructuur geïmplementeerd is. Zolang het gedrag van de procedures (en de vorm waarin ze aangeroepen moeten worden) ongewijzigd blijft, hoeven er in de rest van het programma, daar waar de datastructuur gebruikt wordt, geen veranderingen aangebracht te worden.
• *afscherming van informatie*: bepaalde variabelen die alleen voor de implementatie van belang zijn, komen alleen lokaal voor. De gebruiker van het datatype kan ze niet gebruiken en dus ook niet *mis*bruiken. Dit leidt tot een beter gewaarborgde *correctheid* van het uiteindelijke programma.

Specificatie. Om de gescheiden ontwikkeling van implementatie van een datastructuur en van het programma dat deze structuur zal gebruiken mogelijk te maken, moeten er afspraken gemaakt worden over deze datastructuur. Zo'n specificatie zal twee aspecten vast moeten leggen. Allereerst de syntax van de operaties (de vorm; hoe worden ze aangeroepen, welke typen parameters zijn er), en daarnaast de semantiek (de betekenis; wat is de uitwerking van de operaties, zijn er zij-effecten?).

Voor grote projecten zijn er methoden ontwikkeld die deze aspecten precies vastleggen. We nemen twee voorbeelden over uit het boek van J.C. van Vliet, *Software Engineering*. (Zie ook het college van die naam waar de voors en tegens van de diverse methoden aan de orde komen.)

Bij *algebraische specificaties* wordt het gedarg van de operaties als vergelijkingen in elkaar uitgedrukt, we noemen deze vergelijkingen wel *axioma's* voor de datastructuur.

Er zijn ook zgn. *abstracte model specificaties*, die gebruik maken van abstracte (wiskundige) modellen. We nemen bv. aan dat de begrippen rij, lege rij, laatste element van een rij, etc., een duidelijk vastgelegde (wiskundige) betekenis hebben, en gebruiken deze begrippen in onze specificatie.

Voorbeeld. (1) Algebraïsche specificatie van een stapel.

```
type Intstack;
operators
     Create:                  ⟶ Intstack
     Push: Intstack * Int     ⟶ Intstack
     Pop: Intstack            ⟶ Intstack
     Top: Intstack            ⟶ Int
     Isempty: Intstack        ⟶ Boolean
axioms
     Isempty(Create) = true
     Isempty(Push(s,i)) = false
     Pop(Create) = Create
     Pop(Push(s,i)) = s
     Top(Create) = 0
     Top(Push(s,i)) = i
end Intstack;
```

(2) Abstracte specificatie van een stapel in de taal *Alphard*.

```
let stack = <... x  ...> where x  is int;
invariant 0 ≤ length(stack);
initially stack = nullseq;
function
     push (s: stack, x: int)
          pre 0 ≤ length(s)
          post s = s' ~ x,
     pop (s: stack)
          pre 0 < length(s)
          post s = leader(s'),
     top (s: stack) returns x: int
          pre 0 < length(s)
          post x = last(s'),
     isempty (s: stack) returns b: boolean
          post b = (s = nullseq)                    ■
```

Op dit college zijn we wat minder nauwkeurig. De werking van datastructuren wordt veelal informeel beschreven, terwijl de syntax van de operaties nog vrijgelaten wordt. Op enkele plaatsen zullen we als voorbeeld een specificatie tegenkomen afkomstig uit het boek van Kingston. Deze specificaties zijn geschreven in Modula-2. Deze taal is familie van de ons bekende taal Pascal, en daarom erg makkelijk te begrijpen. Zo'n 'definitie module' lijkt erg veel op de zgn. 'interface' van een unit in Turbo-Pascal (zie hierna). Het voordeel van de Modula-2 aanpak is dat sommige implementatie details die nodig zijn bij Turbo-Pascal hier weggelaten mogen worden.

Voorbeeld. Modula-2 *definition module* voor stapel; implementatie elders.

```
definition module Stack;
     type Stack;
     procedure Initialize(var S: Stack);
     procedure Empty(var S: Stack): boolean;
     procedure Push(x: Entry; var S: Stack);
     procedure Pop(var S: Stack): Entry;
end Stack.                                          ■
```

0.1. UNITS IN TURBO PASCAL

Vanaf versie 4.0 biedt Turbo-Pascal een nieuw hulpmiddel om modulair te kunnen programmeren: de unit. Een unit kan gezien worden als een bibliotheek die een aantal voorgedefinieerde types, constanten, functies en procedures bevat. Eénmaal geschreven kunnen de objecten van de unit in elk programma gebruikt worden door deze unit binnen het programma op te geven. Zolang er niets aan de unit verandert hoeft deze zelfs niet steeds opnieuw gecompileerd te worden. Een aantal van de door Borland bij Turbo-Pascal meegeleverde units zijn bij Programmeermethoden aan de orde geweest:

CRT - in- en uitvoer procedures
DOS - procedures verwant aan functies van het operating systeem
GRAPH - procedures voor grafiesch schermgebruik

We zullen het hier hebben over het schrijven, compileren en gebruiken van zelfgemaakte units. Dit verhaal is echter niet meer dan een eerste introductie. Voor de benodigde details moet de originele Borland handleiding of één van de vele Turbo-Pascal-leerboeken geraadpleegd worden.

Syntax. Een unit heeft de volgende vorm:

```
UNIT <unit naam>
INTERFACE
    <publieke declaraties>
IMPLEMENTATION
    <verborgen declaraties>
    <implementatie publieke declaraties>
BEGIN               {indien initialisatie volgt}
    <initialisatie code>
END.
```

Interface. In de *interface* worden de constanten, variabelen, types, functies en procedures opgesomd die in andere programma's gebruikt kunnen worden. Constanten en types worden gedeclareerd zoals gebruikelijk in een gewoon programma. Van functies en procedures wordt alleen de *header* gegeven; de precieze implementatie volgt verderop in de unit. In zekere zin vormt de interface een specificatie van de unit: de namen van de procedures/functies en de voor de aanroep benodigde syntax (aantallen parameters en hun types) worden aan de gebruiker bekend gemaakt. De compiler heeft aan deze informatie genoeg. Voor een menselijke gebruiker is het aan te bevelen om in kommentaar ook het doel van elk van de onderdelen aan te geven.

Implementatie. De *implementation* bevat de uitwerking van de eerder opgegeven functies en procedures. Daartoe moet er in ieder geval voor elk van de in de interface genoemde procedures/functies een *body* voorkomen, maar het is ook mogeijk om nieuwe types, constanten, variabelen, etc. te declareren die door deze procedures gebruikt kunnen worden. Deze nieuwe objecten blijven 'onzichtbaar' voor de gebruiker van de unit.

Initialisatie. Het kan voorkomen dat, vóór de unit gebruikt kan worden, er eerst een aantal handelingen verricht moeten worden. Zo zal soms het beeldscherm een bepaalde instelling krijgen, moeten er variabelen geïnitialiseerd worden of moet er geheugenruimte gereserveerd worden. Opdrachten hiervoor kunnen in het initialisatiegedeelte van de unit geplaatst worden.

Gebruik van een unit. Een te gebruiken unit wordt in een programma opgegeven mbv. USES..., net als we dat kennen van de standaard units. Alle in de interface van de betreffende unit gedeclareerde objecten mogen dan in het programma gebruikt worden. De alleen in de implementation voorkomende variabelen etc. blijven voor de gebruiker van de unit onzichtbaar. Bij gebruik hiervan zal de compiler een foutmelding geven.

Eénzelfde variabelenaam kan voorkomen in verschillende units. Om buiten de unit deze variabelen toch te kunnen onderscheiden kunnen we expliciet de unit-naam aan de naam van de variabele toevoegen: unit5.x stelt de variabele x uit unit5 voor. Overigens kunnen units zèlf ook weer units gebruiken.

Compilatie. Een gecompileerd programma levert, zoals bekend, een .EXE file op. Het compileren van een unit leidt daarentegen tot een .TPU file. Standaard wordt aangenomen dat een benodigde unit gecompileerd op schijf staat op het moment dat het aanroepende programma gecompileerd wordt. Bij testen van een programma kan dit veel tijd besparen: de unit behoeft niet steeds opnieuw gecompileerd te worden. Nadat een unit veranderd is, is het nodig om ook een programma dat deze unit gebruikt opnieuw te compileren. Het is natuurlijk niet altijd zo dat alle gebruikte units veranderd zijn. Er zijn dan ook meerdere manieren om programma's te compileren:

- compile: compileert gebruikmakend van reeds bestaande .TPU files van benodigde units (de standaard manier).
- make: compileert units alleen wanneer nodig, dwz. als de unit nog niet eerder gecompileerd was, of als de unit ná de vorige compilatie veranderd is.
- build: alle benodigde units worden opnieuw gecompileerd.

De compiler verwacht dat de Pascal tekst van de unit <unitnaam> te vinden is in de file <unitnaam>.PAS. Wanneer de naam van de file anders is dan de naam van de unit, moet deze in het aanroepende programma expliciet opgegeven worden (voor de liefhebbers: mbv. {$U ...}). Wanneer een bepaalde unit (via-via) tweemaal aangeroepen wordt, bv. eenmaal in het programma maar ook een keer in een unit die het programma gebruikt, wordt deze slechts éénmaal gecompileerd. Ook de initialisatiecode wordt slechts één keer uitgevoerd.

Omdat 'compileren' bij Turbo Pascal niet alleen compileren maar ook linken inhoudt is de situatie eigenlijk iets subtieler dan hierboven aangeduid. Stel unit1 gebruikt unit2 ; prog0 gebruikt unit1. Als van unit2 alleen de implementatie en *niet* de interface veranderd is, dan hoeft unit1 niet opnieuw gecompileerd te worden omdat bij compilatie van unit1 de juiste gegevens gebruikt zijn: de aantallen en types van parameters bv. zijn ongewijzigd gebleven. Pas bij compilatie van prog0 worden de gegenereerde codes gelinkt. Omdat prog0 anders de oude code gebruikt moet dit programma wél opnieuw gecompileerd worden.

Voorbeeld.

```
PROGRAM prog1
USES unit2
VAR x, factor: integer;
BEGIN
   writeln('programma 1');
   factor := -1;
   unit2.factor := -2;
   readln(x);
   x := flop(x) * factor;
   writeln(x);
END. {prog1}
```

```
UNIT unit2
INTERFACE
VAR factor: integer;
PROCEDURE flop(VAR a:integer);
IMPLEMENTATION
CONST tien = 10; {alleen lokaal}
PROCEDURE flop
BEGIN
   a := tien + a * factor
END; {flop}
BEGIN {initialisatie}
   writeln('unit2');
   factor := -1
END. {unit2}
```

Bij aanroep van het programma wordt eerst de initialisatie van unit2 uitgevoerd en dan de body van prog1.

Ten aanzien van de zichtbaarheid van gedeclareerde objecten geldt het volgende:

- de variabelen x en factor van prog1 zijn *niet* zichtbaar in unit2,
- de variabele factor en de procedure flop van unit1 zijn zichtbaar in prog1,
- de constante tien van unit2 is verborgen.

De zichtbare procedure flop kan in prog1 gebruikt worden onder de naam flop of voluit unit2.flop. De declaratie van factor in prog1 verdringt die van factor in unit2. Deze laatste kan in prog1 alleen gebruikt worden onder de volledige naam unit2.factor.

I. BOMEN

1. Binaire Bomen

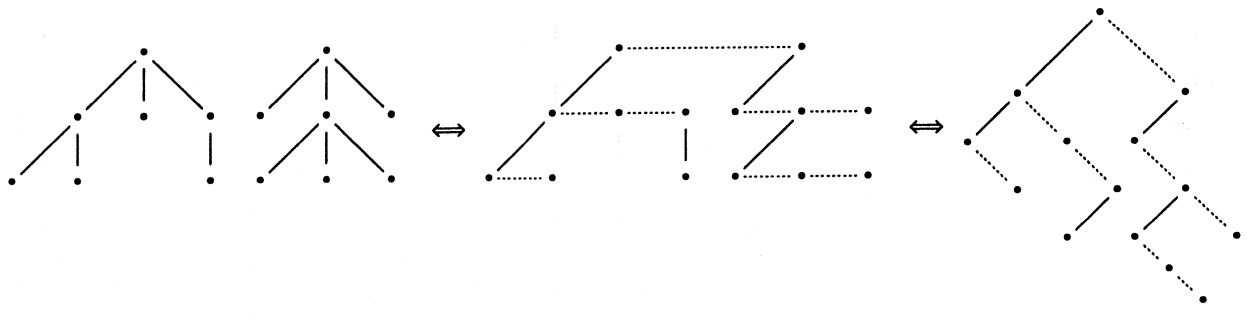
Bomen, en meer specifiek binaire bomen, vormen de meest gebruikte niet-lineaire structuur om gegevens te ordenen. Door met elke knoop van een boom een waarde te associeren, brengen we een hierarchische ordening aan tussen deze waardes. We komen bomen tegen in verschillende vormen. Soms zijn er beperkingen aan de structuur van de boom (bijvoorbeeld wat betreft de hoogtes van subbomen of het aantal kinderen van een knoop), soms zijn er eisen aan de relaties tussen de verschillende opgeslagen waardes gesteld (denk hierbij bijvoorbeeld aan zoekbomen of aan de *heap*). Een aantal van deze speciale soorten bomen komen in het volgende hoofdstuk aan de orde. Hier kijken we naar bomen in het algemeen.

We herhalen kort enige terminologie zoals geïntroduceerd bij het college Algoritmiek; zie verder ook het college Discrete Wiskunde.

Binaire bomen kunnen op recursieve wijze gedefinieerd worden. Een binaire boom is of leeg, of een knoop samen met twee binaire bomen. De knoop heet de wortel van de boom, de twee binaire bomen zijn de linker en rechter subboom van de boom. De wortels van de subbomen (als die niet leeg zijn) zijn de kinderen van de wortel van de boom. Een binaire boom heet vol als elke knoop van de boom of nul of twee kinderen heeft.

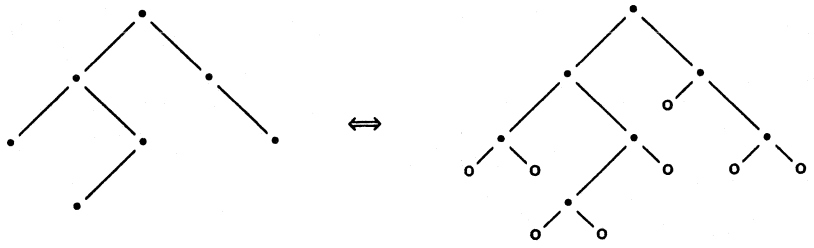
1.1. Algemene Wetenswaardigheden

Representatie van willekeurige bomen. Binaire bomen kunnen gebruikt worden om willekeurige (gerichte geordende) bomen en bossen te representeren. We behouden de knopen en gebruiken de linker tak om het eerste kind van elke knoop aan te geven, de rechter tak geeft telkens de rechter broer van de knoop in de oorspronkelijke boom. We vatten bij bossen de wortels van de bomen op als kinderen van dezelfde (niet bestaande) knoop.



Aantallen knopen en takken. Het aantal takken in een boom is één minder dan het aantal knooppunten. Dit valt met volledige inductie te bewijzen, zie bv. het college Discrete Wiskunde. Samen met het feit dat iedere knoop bereikbaar is vanuit elke andere knoop is dat zelfs een karakteriserende eigenschap van (ongerichte, dwz. zonder wortel) bomen.

Relatie met volle binaire bomen. Elke binaire boom kan veranderd worden in een *volle* binaire boom door elke knoop twee kinderen te geven. De knopen van de oorspronkelijke boom worden zo de interne knopen van de nieuwe boom, de toegevoegde knopen de bladeren. Op deze manier correspondeert een boom met n knopen met een volle binaire boom met 2n+1 knopen: n interne knopen en n+1 bladeren.



1.2. Het Tellen van Binaire Bomen

Een recurrente betrekking. We willen in deze paragraaf het aantal binaire bomen met een gegeven aantal knooppunten gaan bepalen. We gebruiken b_n als notatie voor het aantal binaire bomen met n knooppunten. De reeks van getallen b_n beginnend bij n = 0 heeft als beginwaarden 1, 1, 2 ,5, 14, 42. We kunnen de volgende waarden telkens uit de voorafgaande berekenen door gebruik te maken van de recursieve formule $b_{n+1} = \sum_{i=0}^{n} b_i b_{n-i}$. Immers, een binaire boom met n+1 knooppunten heeft twee subbomen die samen n knopen hebben. Wanneer de rechter en linker subboom respectievelijk i en n-i knopen hebben dan zijn er dus $b_i b_{n-i}$ manieren om deze te kiezen. Wat we zoeken is echter een 'gesloten' formule voor b_n, dwz. één die geen gebruik maakt van recursie.

Eerste methode. De meest gebruikte manier om het aantal binaire bomen met een gegeven aantal knopen te berekenen bevat helaas een aantal trucs en toverformules voor gevorderden. We geven de methode om volledig te zijn (en verklappen daarmee een opgave van het college Discrete Wiskunde).

Zij $g(x) = \sum_{n=0}^{\infty} b_n x^n$ het *genererende polynoom* van de getallen b_n. In het kwadraat van g is de coefficient van x^n gelijk aan $b_0 b_n + b_1 b_{n-1} + \ldots + b_n b_0$. Volgens de boven gegeven recurrentie is dit gelijk aan b_{n+1}, de coefficient van x^{n+1} in g zelf. We vinden daarmee dat $g(x) = x \cdot \{g(x)\}^2 + 1$. De oplossing

van deze kwadratische vergelijking in $g(x)$ levert ons $g(x) = \frac{1 \pm \sqrt{1-4x}}{2x}$.
Volgens het *Binomium van Newton* geldt $(1+z)^r = \sum_{n=0}^{\infty} \binom{r}{n} \cdot z^n$, waarbij de *gegeneralizeerde binomiaal coefficient* $\binom{r}{n}$ gelijk is aan $\frac{r \cdot (r-1) \cdot \ldots \cdot (r-n+1)}{1 \cdot 2 \cdot \ldots \cdot n}$.
Met deze formule schrijven we $\sqrt{1-4x} = (1-4x)$ in de formule voor $g(x)$ verder uit. We kiezen voor de negatieve wortel van de vergelijking omdat achteraf zal blijken dat dit juist een positieve oplossing voor de getallen b_n oplevert. Er geldt dus dat

$$g(x) = \frac{1}{2x}\left(1 - \sum_{n=0}^{\infty} \binom{n}{n} \cdot (-4x)^n\right) = \sum_{n=0}^{\infty} \binom{n+1}{n+1} \cdot (-1)^n 2^{2n+1} x^n.$$

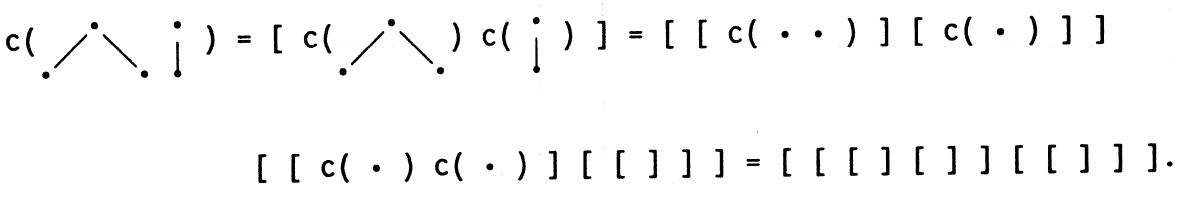
We rekenen de binomiaal coefficient uit: $\binom{n+1}{n+1} = \frac{\cdot(-1) \cdot \ldots \cdot (-n)}{1 \cdot 2 \cdot \ldots \cdot (n+1)} =$

$$(-)^{n+1} \cdot \frac{-1 \cdot 1 \cdot 3 \cdot \ldots \cdot (2n-1)}{(n+1)!} \cdot \frac{2 \cdot 4 \cdot \ldots \cdot 2n}{2^n \cdot n!} = \frac{(-1)^n}{2^{2n+1}} \cdot \frac{(2n)!}{(n+1)! \cdot n!}.$$

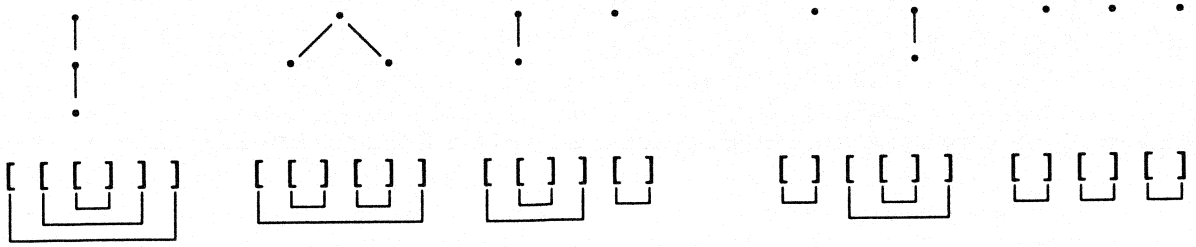
Hiermee zien we dat $b_n = \frac{(2n)!}{(n+1)! \cdot n!} = \frac{1}{n+1}\binom{2n}{n}$.

Tweede methode. Een meer elementaire methode is gebaseerd op de correspondentie tussen binaire bomen en rijtjes van netjes gebalanceerde haakjes [en]. Het is niet erg moeilijk om deze correspondentie rechtstreeks voor binaire bomen te geven, maar het is nog iets eenvoudiger om geordende bossen te tellen. We weten inmiddels dat er evenveel binaire bomen als geordende bossen zijn, dus dit levert dezelfde aantallen op.

We geven een recursieve definitie van de gebruikte codering. De codering $c(T)$ van de boom bestaande uit één enkele knoop is [ ]. De codering van een niet-lege boom T is het rijtje [$c(T_1)$ ... $c(T_n)$] , waarbij T_1, T_2, ..., T_n de subbomen van de wortel van T zijn. We coderen een bos door de coderingen van de afzonderlijke knopen achter elkaar te schrijven; de bomen uit een bos worden dus gezien als kinderen van één (niet-bestaande knoop). We geven een voorbeeld.



Deze codering vormt een één-één correspondentie tussen geordende bossen met n knopen en rijtjes van n [-en en evenveel]-en die *gebalanceerd* zijn, of anders gezegd, waar van voor naar achter gelezen nooit meer]-en voorkomen dan [-en. In zulke rijtjes hebben de [-en en]-en de vorm van goed-geneste haakjes paren.



In totaal zijn er dus $\binom{2n}{n}$ rijtjes met n [-en en evenzoveel]-en. We willen echter niet al deze rijtjes meetellen. In het soort rijtjes dat wij zoeken komt kan het aantal]-en nooit voorlopen op het aantal [-en. We zijn geinteresseerd in het aantal van die rijtjes. Het blijkt makkelijker te zijn om rijtjes te tellen die juist wèl op een punt meer]-en dan [-en hebben. Dit aantal trekken we dan later van het totaal aantal af.

Stelling. Het aantal rijtjes met n [-en en n]-en dat niet gebalanceerd is, is precies gelijk in het aantal rijtjes met n-1 [-en en n+1]-en (in een willekeurige volgorde).

Bewijs. We kunnen dit laten zien door een bijectieve afbeelding te geven die de twee soorten rijtjes in elkaar omzet, waarbij verschillende rijtjes op verschillende rijtjes terechtkomen.

Bekijk eerst een willekeurig rijtje met n-1 [-en n+1]-en, en zoek daarin de eerste sluithaak] die verkeerd staat; zo'n haak bestaat omdat er meer]-en zijn dan [-en. Voor deze sluithaak staan evenveel [-en als]-en, erna staat één] meer dan [. Wanneer we *na* deze foute [alle [-en en]-en omzetten in de andere soort krijgen we een rijtje met evenveel [-en als]-en, dat nog steeds niet gebalanceerd is.

Op dezelfde manier kunnen we ook het oorspronkelijke rijtje terug vinden; dit laat zien dat niet twee verschillende rijtjes op eenzelfde rijtje afgebeeld worden. ■

Illustratie. n = 3; 20 rijtjes totaal, waarvan 5 gebalanceerd. Er zijn 15 rijtjes met 2 [-en en 4]-en. Aangegeven met][is telkens de eerste niet-gebalanceerde] in de foute rijtjes.

[[[ ]]]
[[ ] [ ]]
[[ ]] [ ]
[[ ]]][↔ [[ ]]][ ]
[ ] [[ ]]
[ ] [ ] [ ]
[ ] [ ]][↔ [ ] [ ]][ ]
[ ]][[ ] ↔ [ ]][ ]] [
[ ]][ ] [↔ [ ]][ ] [ ]
[ ]][ ] [[↔ [ ]][ ]]

][[[[ ] ↔][ ]]] [[
][[[ ] [ ] ↔][ ]] [ ] [
][[[ ]] [↔][ ]] [[ ]
][[ ] [[ ] ↔][ ] [ ]] [
][[ ] [ ] [↔][ ]]] [ ]
][[ ]] [[↔][ ] [ ]]]
][ ] [[[ ] ↔][ ] [ ]] [
][ ] [[ ] [↔][ ] [ ] [ ]
][ ] [ ] [[↔][ ] [ ]]]
][ ]] [[[↔][[[ ]]]

■

We passen dit gegeven toe: Het aantal niet gebalanceerde rijtjes van n

[-en en n]-en is dus gelijk aan $\binom{2n}{n-1}$. Het aantal rijtjes van n [-en en n]-en dat we zoeken is gelijk aan het totaal aantal rijtjes $\binom{2n}{n}$ minus het aantal foute rijtjes $\binom{2n}{n-1}$, dus $b_n = \binom{2n}{n} - \binom{2n}{n-1} = \binom{2n}{n} \cdot (1-\frac{n}{n+1}) = \frac{1}{n+1} \cdot \binom{2n}{n}$. Deze getallen staan bekend als de *getallen van Catalan*.

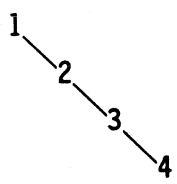
Volgens de *formule van Stirling* is n! ongeveer gelijk aan $\sqrt{2\pi n}\ (n/e)^n$. Deze benadering kan gebruikt worden om de grootte van b_n aan te geven. We vinden daarmee dat b_n ongeveer gelijk is aan $4^n/(n+1)\sqrt{\pi n}$.

Bij het college Algoritmiek is al aangegeven wat het nut van deze berekening is. Om van een representatie of notatie voor (binaire) bomen te kunnen nagaan of deze niet te veel ruimte inneemt, moeten we de lengte van de representatie vergelijken met het totaal aantal mogelijke bomen. Omdat het aantal binaire bomen met n knopen grofweg $4^n = 2^{2n}$ is, hebben we tenminste $\lg 4^n = 2n$ bits nodig om elke boom met n knopen een aparte representatie te kunnen geven. Gebruikt een representatie aanzienlijk meer bits, dan springen we slordig met de beschikbare geheugenruimte om. Let op dat deze analyse alleen geldt voor de structuur van de bomen. De extra ruimte nodig om de opgeslagen gegevens weer te geven hangt af van de soort van die gegevens.

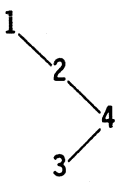
Merk op dat we nu niet alleen weten hoeveel binaire bomen er zijn. We kunnen de correspondentie tussen binaire bomen en (geordende) bossen of volle binaire bomen gebruiken om ook voor deze andere twee klassen bomen de aantallen af te leiden.

1.3. De Gemiddelde Padlengte van Binaire Bomen

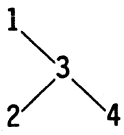
Mogelijke binaire bomen. Wanneer we een binaire zoekboom opstellen voor n sleutels $K_1 < \ldots < K_n$, dan hangt de vorm van de boom af van de volgorde waarin we de sleutels aan de boom toevoegen (aangenomen dat we niet tijdens het opstellen de vorm aanpassen). Voor vier sleutels zijn er bijvoorbeeld 4! = 24 mogelijke volgordes om de sleutels te rangschikken. De bomen die ontstaan door de sleutels in de aangegeven volgorde in te voeren zijn hieronder geschetst. Soms leiden verschillende volgordes tot dezelfde boom; er zijn immers slechts 14 binaire bomen met 4 knopen.



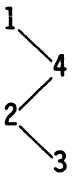
1234



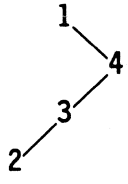
1243



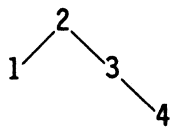
1324, 1342



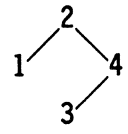
1423



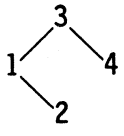
1432



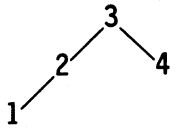
2134, 2314, 2341



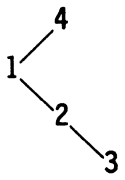
2143, 2413, 2431



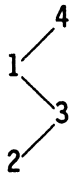
3124, 3142, 3412



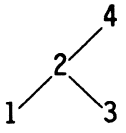
3214, 3241, 3421



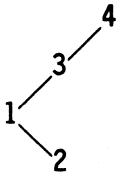
4123



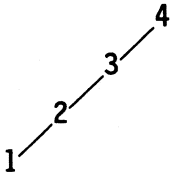
4132



4213, 4231



4312



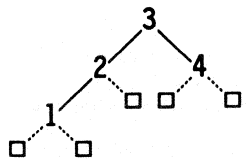
4321

Interne en externe padlengte. We herhalen een aantal wetenswaardigheden van het college Algoritmiek. Een binaire zoekboom wordt vaak (in gedachten) uitgebreid tot een volle binaire boom door bladeren toe te voegen; de boom die zo ontstaat heet dan een uitgebreide binaire zoekboom. De interne padlengte I van zo'n boom is de som over alle interne knopen van de boom van het nivo van de knoop min één. De externe padlengte E ontstaat op dezelfde manier door alle (toegevoegde) bladeren te bekijken. Tussen deze twee waarden geldt de relatie $E = I + 2n$, waarbij n het aantal interne knopen van de uitgebreide zoekboom is (dwz. het oorspronkelijke aantal knopen). Deze formule kan eenvoudig met inductie bewezen worden.

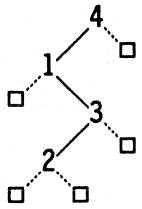
De waarden I en E zijn een indicatie voor de 'efficientie' van de boom. Onder de aanname dat alle sleutels in de boom even vaak gezocht worden, is het aantal benodigde vergelijkingen om een sleutel te vinden gemiddeld (I+n)/n.

Voor zoekacties naar sleutels die niet in de boom staan (eventueel voorafgaand aan toevoegingen in de boom) moeten we kijken naar de bladeren; het gemiddelde aantal vergelijkingen om te constateren dat een sleutel niet in de boom staat is E/(n+1).

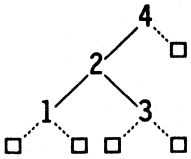
Voor een aantal representatieve bomen uit het voorafgaande voorbeeld levert dit de volgende getallen voor de beide padlengtes.



I = 0+1+1+2 = 4
E = 2+2+2+3+3 = 12



I = 0+1+2+3 = 6
E = 1+2+3+4+4 = 14



I = 0+1+2+2 = 5
E = 1+3+3+3+3 = 13

De gemiddelde interne padlengte. De interne padlengte van een binaire boom is dus een maat voor het te verwachten aantal zoekstappen wanneer die boom in gebruik is als zoekboom. Zoals reeds bij Algoritmiek besproken is lopen de interne padlengtes van de bomen met een vast aantal knopen (zeg n) nogal uiteen. Zo zijn er de ontaarde 'lineaire' bomen waarvan de interne padlengte gelijk is aan 0+1+2+...+(n-1) = (n-1)·n. Anderzijds zijn er de complete bomen (en bomen die daar op lijken) met interne padlengte 0+1+1+2+2+2+2+...+⌊*lg* n⌋, dit is ongeveer n·*lg* n. We zien dat voor n = 4 acht permutaties van de getallen 1,2,...,n lineare bomen geven en twaalf permutaties 'redelijk' complete bomen. Een voor de hand liggende vraag is nu wat de gemiddelde padlengte I_n is als functie van n, wanneer we aannemen dat alle permutaties van de sleutels bij het opstellen van de boom even waarschijnlijk zijn. We komen zo te weten wat de efficientie is van een 'gemiddelde' boom. Voor n = 4 kunnen we dat alvast uit de plaatjes halen: $I_4 = [\ 8\cdot(0+1+2+3) + 4\cdot(0+1+2+2) + 12\cdot(0+1+1+2)\ ] / 24 = 4^5/_6$.

Het rekenwerk. We mogen aannemen dat de sleutels die we gebruiken om de boom op te stellen de getallen 1,...,n zijn. De vorm van de boom wordt door de permutatie van deze getallen bepaald. Het eerste getal van de permutatie komt in de wortel te staan; van alle getallen die volgen komen de kleinere links van de wortel, de grotere rechts. Op deze manier valt de oorspronkelijke permutatie uiteen in twee kleinere permutaties: voor elk van de subbomen één. Bv.: 6 9 3 1 7 4 2 5 8 geeft wortel 6 en permutaties 3 1 4 2 5 en 9 7 8.

Om E_n (en daaruit later I_n) te berekenen zullen we eerst kijken naar bomen met een vast gekozen waarde k in de wortel. Er zijn (n-1)! permutaties die met k beginnen. Deze permutaties leveren elk een permutatie van de k-1 elementen kleiner dan k en een van de n-k elementen groter dan k, die de

linker en de rechter subboom zullen bepalen. Uitgaande van alle (n-1)! permutaties die met k beginnen, vinden we alle mogelijke 'subpermutaties' even vaak, nl. $\binom{n-1}{k-1}$ maal. De gemiddelde externe padlengtes voor de subbomen zijn dus E_{k-1} en E_{n-k}, omdat we gewoon over subpermutaties kunnen middelen. De subbomen komen één knoop lager te hangen waardoor elk van de n+1 bladeren één nivo zakt. Wanneer we ons beperken tot bomen met k in de wortel komt de gemiddelde externe padlengte dus op $E_{k-1} + E_{n-k} + n+1$.

Daarmee weten we ook E_n uit te drukken: $E_n = \frac{1}{n} \sum_{k=1}^{n} (E_{k-1} + E_{n-k} + n+1)$. Hieruit volgt dat

$$n \cdot E_n = n \cdot (n+1) + 2 \cdot \sum_{k=1}^{n-1} E_k .$$

Evenzo $$(n+1) \cdot E_{n+1} = (n+1) \cdot (n+2) + 2 \cdot \sum_{k=1}^{n} E_k .$$

Door de bovenste van deze formules van de onderste af te trekken, vinden we de recurrente betrekking

$$(n+1) \cdot E_{n+1} - n \cdot E_n = 2 \cdot (n+1) + 2 \cdot E_n$$

Dus $$E_{n+1} = 2 + \frac{n+2}{n+1} \cdot E_n .$$

Verder is $$E_1 = 2.$$

Het is leerzaam om hiervan een aantal waarden te berekenen:

$E_2 = 2 + {}^3/_2 \cdot E_1 = 2 + {}^3/_2 \cdot 2 = 5$

$E_3 = 2 + {}^4/_3 \cdot E_2 = 2 + {}^4/_3 \cdot 2 + {}^4/_2 \cdot 2 = 8\ {}^2/_3$

$E_4 = 2 + {}^5/_4 \cdot E_2 = 2 + {}^5/_4 \cdot 2 + {}^5/_3 \cdot 2 + {}^5/_2 \cdot 2 = 12\ {}^5/_6$

Een aardige veronderstelling is dat $E_n = 2 \cdot (n+1) \cdot H_n - 2 \cdot n$, waarbij $H_n = \sum_{k=1}^{n} {}^1/_k$ (hieruit volgt dat $H_n \leq 1 + \mathit{lg}\, n$). Dan geldt dat

$$\boxed{I_n = E_n - 2 \cdot n = 2 \cdot (n+1) \cdot H_n - 4 \cdot n}$$

Opgave. Laat zien dat bovenstaande veronderstelling inderdaad aan de recursieve vergelijking voldoet. ■

Conclusie. Met deze formule kan dan uiteindelijk de efficientie van de gemiddelde zoekboom worden berekend. Bij 'succesvol' zoeken in een boom met n knopen hebben we gemiddeld $2 \cdot (1 + {}^1/_n) \cdot H_n - 3$ vergelijkingen nodig. Grofweg zijn dit $2 \cdot \mathit{ln}\, n \approx 1{,}39 \cdot \mathit{lg}\, n$ vergelijkingen. Dit is lang niet slecht vergeleken met de optimale situatie: voor een complete boom is het aantal stappen gemiddeld ongeveer $\mathit{lg}\, n$ stappen. Aan de andere kant kunnen we ongelukkigerwijs een boom treffen waarin het aantal vergelijkingen gemiddeld (n+1) is.

2. Representaties van Bomen

De twee Pascal constructies die het meest geschikt zijn om gegevens mee te structureren zijn zoals bekend het array en de pointer. In het algemeen kan met pointers flexibeler omgegaan worden dan met arrays. Soms echter kan informatie met arrays compacter opgeslagen worden, als er een logische manier is om de gegevens over het array te verdelen. Dit is bv. het geval als er een eenvoudige manier bestaat om de plek van de geplaatste informatie te kunnen berekenen. Dit scheelt dan het expliciete opslaan van een geheugenadres.

De meeste methoden om bomen te representeren kennen we al van het college Algoritmiek. We beginnen bij binaire bomen.

Pointers en binaire bomen. Een veel gebruikte methode om binaire bomen te representeren maakt gebruik van pointers. Elke knoop krijgt twee pointers die wijzen naar de wortel van de twee subbomen van die knoop. NIL geeft de lege subboom aan.

```
TYPE Wijzer = ↑Knoop;
     Knoop = RECORD
               Info : NaarEigenKeus;
               Links, Rechts : Wijzer
             END;
```

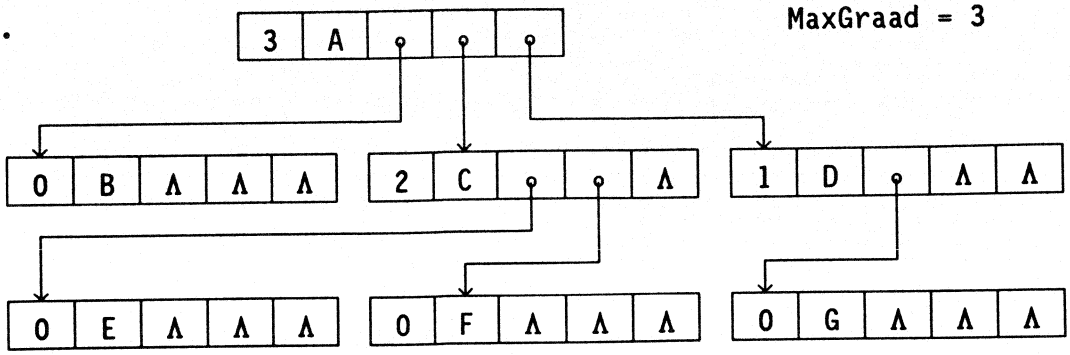
Arrays en complete binaire bomen. Wanneer we werken met *complete* binaire bomen ligt een array implementatie voor de hand. We slaan de knopen in nivo-orde (zie volgende paragraaf) in een array A[1..MaxKnoop] op. Familie-relaties kunnen simpel berekend worden uit de array indices. De kinderen van A[i] zijn A[2i] en A[2i+1], omgekeerd is de vader van A[i] het array element A[i *div* 2]. Deze methode geldt analoog voor k-aire bomen (waar elke knoop graad k heeft). Een belangrijke toepassing van deze representatie is de sorteermethode Heapsort, zie hiervoor het college Algoritmiek.

Geordende gerichte bomen. De methoden om binaire bomen te representeren kunnen aangepast worden voor willekeurige bomen.

Allereerst worden pointers gebruikt om willekeurige (geordende, gerichte) bomen te representeren. Er zijn twee variaties. We kunnen bijvoorbeeld de graad van de gebruikte knopen begrenzen door de constante MaxGraad. We krijgen dan:

```
TYPE Wijzer = ↑Knoop;
     Knoop = RECORD
               Graad : 0..MaxGraad;
               Info  : NaarEigenKeus;
               Kind  : ARRAY[1..MaxGraad] OF Wijzer
             END;
```

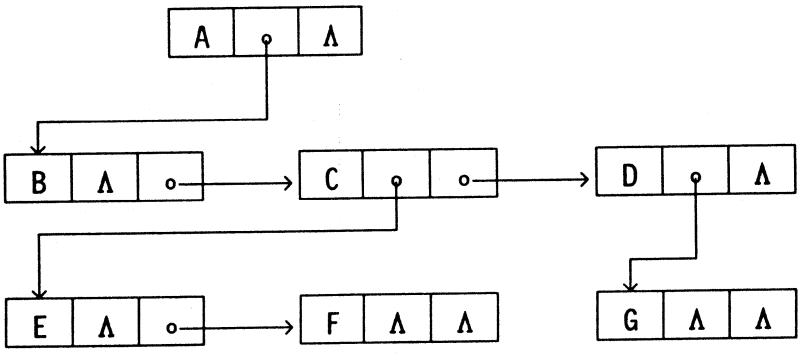
Vb.



We kunnen ook de correspondentie tussen willekeurige (gerichte geordende) bomen en binaire bomen gebruiken. Per knoop wijst nu één pointer naar het oudste kind, een andere pointer naar de eerst jongere broer van de knoop. De declaraties in Pascal zijn als de eerste hierboven gegeven, we zouden alleen de velden Links en Rechts kunnen herdopen in EersteKind en RechterBroer.

```
TYPE Wijzer = ↑Knoop;
     Knoop = RECORD
               Info         : NaarEigenKeus;
               EersteKind,
               RechterBroer : Wijzer
             END;
```

Vb.

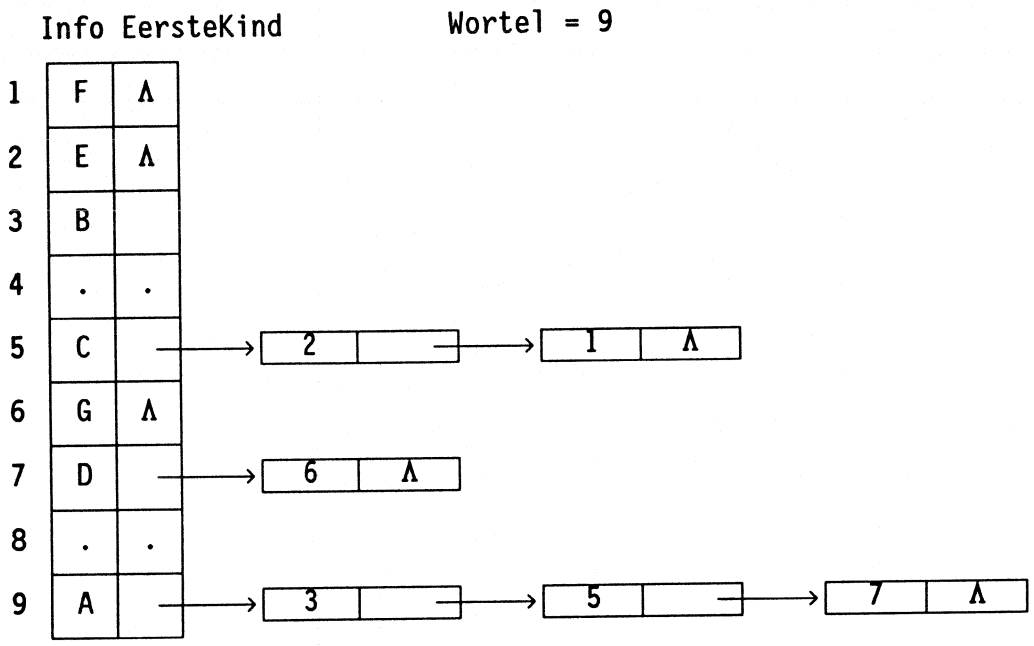


Wanneer er een redelijke bovengrens aan het aantal in de boom op te nemen knopen bestaat, kunnen we de knopen in een array opslaan. Vaak zijn dit variaties op de pointer methoden waarbij de pointer (een geheugen adres) vervangen is door een array index. Dit betekent dat we eigenlijk zèlf het geheugenbeheer nadoen.

Lijsten van kinderen. Elke knoop krijgt een plek in een array van lijsten. De lijst bevat indices van de kinderen van die knoop. Naar keuze kunnen we de kinderen al of niet geordend in de lijst opslaan. Deze methode lijkt erg op de *adjacency-lists* representatie voor grafen. Let op dat de wortel van de gerepresenteerde boom nog opgeslagen moet worden.

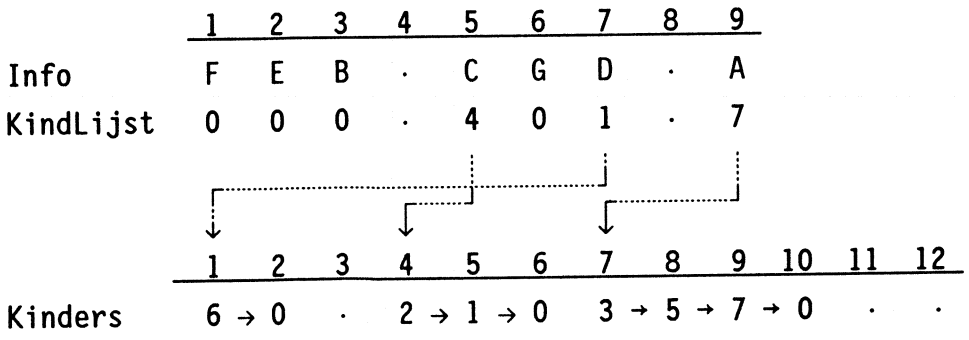
```
TYPE KindWijzer = ↑Kind;
     Kind = RECORD
               Index : 1..MaxKnoopAantal;
               Broer : Kindwijzer;
            END;
     KnoopArray = ARRAY[1..MaxKnoopAantal] OF
                     RECORD
                        EersteKind : KindWijzer;
                        Info       : NaarEigenKeus
                     END;
```

Vb.



Bij een erge afkeer van pointers kunnen zelfs de lijsten van kinderen in een array worden opgeslagen. In onderstaand voorbeeld gebeurd dit door de lijsten aaneengesloten stukken van het array te laten gebruiken. Lijsten scheiden we dan bv. door de index 0. Deze variant is dan helaas minder flexibel.

Vb.

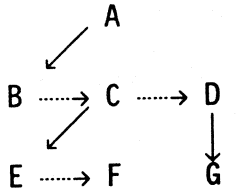


Ook de standaard pointer-representaties voor binaire bomen en voor willekeurige bomen kunnen geheel met arrays worden nagedaan. Als voorbeeld geven we de variant voor de representatie van binaire bomen. Iets dergelijks kan gedaan worden voor de EersteKind-RechterBroer representatie.

```
TYPE Knoop = RECORD
               Info          : NaarEigenKeus;
               Links, Rechts : 0..MaxKnoopAantal
             END;
     Boom = ARRAY[1..MaxKnoopAantal] OF Knoop;
```

Merk op dat het nulde array element niet bestaat; een verwijzing naar 0 wordt gebruikt om een NIL-pointer na te doen. In plaats van één array van records kunnen ook drie arrays Info, Links en Rechts gebruikt worden. De gegeven oplossing is echter eleganter omdat de samenhang van de gegevens duidelijker is.

Vb. EersteKind-RechterBroer representatie mbv. een array.



	1	2	3	4	5	6	7	8	9
EersteKind	0	0	0	·	2	0	6	·	3
RechterBroer	0	1	5	·	7	0	0	·	0
Info	F	E	B	·	C	G	D	·	A

Verwijzingen naar ouders. In de voorgaande representaties zijn de paden van de wortel richting bladeren eenvoudig te volgen. In sommige toepassingen is het juist handig om de takken van een boom als het ware onderste-boven op te slaan om zo, vanuit een gegeven knoop, de wortel te kunnen vinden. Dit kunnen we doen door voor elke knoop een verwijzing naar de ouder op te slaan in een array.

We moeten nog afspreken wat de 'ouder' van de wortel van een boom is. Een mogelijke afspraak is een variant op NIL te gebruiken (dwz. een verwijzing naar een speciale niet-bestaande knoop) of anders de wortel zelf. Een belangrijk nadeel van de representatie is dat de meeste andere familie-relaties moeilijk, dan wel helemaal niet te berekenen zijn: broers en kinderen kunnen alleen gevonden worden door het array langs te lopen. Ook geeft deze representatie geen ordening aan tussen broers. Indien deze relaties belangrijk zijn voor een bepaalde toepassing moeten ze dus toegevoegd worden.

Disjuncte verzamelingen. Eén van de opgaven bij dit hoofdstuk behandelt een belangrijk gebruik van deze laatste representatie: de abstracte datastructuur *union-find*. Deze datastructuur wordt gebruikt voor het bijhouden van een collectie van disjuncte verzamelingen over een bepaald domein. Elementen uit het domein komen in precies één van de verzamelingen uit de collectie voor. Als. bv. a,b,...,f namen van objecten zijn, dan is {{a,e},{b,c,f},{d}} een disjunctie verzameling-structuur. Een belangrijke toepassing hiervan vinden we in Hoofdstuk IV, over grafen, bij het bepalen van een minimale opspannende boom van een graaf.

De datastructuur heeft twee kenmerkende operaties:

Find: levert bij een gegeven element de (naam van de) verzameling op waar het element inzit.

Union: levert de vereniging van twee gegeven disjuncte verzamelingen.

Naast dit essentiële tweetal hebben we bv. ook een operatie nodig die verzamelingen initialiseert door het element x om te zetten in de verzameling {x}. Hieronder volgt een voorbeeld van de specificatie van deze datastructuur in Modula-2:

```
definition module DisSets;                          (* boek van Kingston *)
    type Entry;          (*an entry*)
    type SetType;        (*a set of entries*)
    type DisjointSets;   (*a set of sets of entries*)
    procedure New(): Entry;
    procedure Initialize(var D: DisjointSets);
    procedure MakeSet(x: Entry; var D: DisjointSets);
    procedure Find(x: Entry; var D: DisjointSets): SetType;
    procedure Union(v, w: SetType; var D: DisjointSets);
end DisSets.
```

3. Boomwandelingen

Voor (binaire) bomen is er een viertal standaard wandelingen; WLR (of *pre-orde*), LWR (of *symmetrisch*), LRW (of *post-orde*) en de *nivo-orde* wandeling. Voor de eerste drie van deze wandelingen is de meest voor de hand liggende definitie recursief. Deze definitie laat zich dan ook op eenvoudige wijze in een recursief algoritme omzetten.

We zullen nu een aantal boomwandel algoritmes bestuderen die niet expliciet van recursie gebruik maken.

3.1. Met Behulp van een Stapel

Bij recursie wordt impliciet een stapel gebruikt. Voor de pre-orde wandeling zullen we de recursie wegwerken en expliciet gebruik maken van een stapel.

In het algoritme gebruiken we een 'abstracte' stapel S, dwz. we interesseren ons niet voor de preciese representatie (bv. met een array of met pointers). We nemen daarbij aan dat een aantal functies en procedures op S gedefinieerd zijn: Initialiseer(S) initialiseert S als lege stapel; IsLeeg(S) test of de stapel leeg is; S ⇐ Elt voegt Elt toe aan de stapel; Elt ⇐ S

verwijdert (indien S niet leeg is) het bovenste stapelelement en geeft de waarde daarvan aan Elt, de operatie is ongedefinieerd wanneer S leeg is; Top(S) levert de waarde van het bovenste stapelelement zonder dit te verwijderen, de waarde is weer ongedefinieerd als de stapel leeg is.

Algoritme : pre-orde binaire boomwandeling mbv. stapel

```
Initialiseer(S);
S ⇐ Wortel;
while not IsLeeg(S)
do   Deze ⇐ S;
     while Deze ≠ NIL
     do   bezoek(Deze);          {bezoek is elders gedeclareerd}
          S ⇐ Deze↑.Rechts;
          Deze := Deze↑.Links
     od
od
```

3.2. LINKOMKERING

Het is mogelijk om zonder extra datastructuur een boomwandeling te maken door tijdens de wandeling de pointers van de boom te verhangen. Bij het college Algoritmiek werd één zo'n methode behandeld, Lindstrom Scanning. Een nadeel van deze methode is misschien dat we niet goed kunnen zien of we een knoop voor de eerste, tweede of derde maal bezoeken. Hier presenteren we een algoritme dat dit nadeel niet heeft. Wèl dient er in de boom bij elke knoop plaats te zijn om een extra bit aan te brengen. Overigens is een algemeen nadeel van algoritmes die de boomstruktuur veranderen door takken te verplaatsen dat het niet mogelijk is twee of meer wandelingen tegelijk uit te voeren (tenzij misschien met bijzondere voorzieningen).

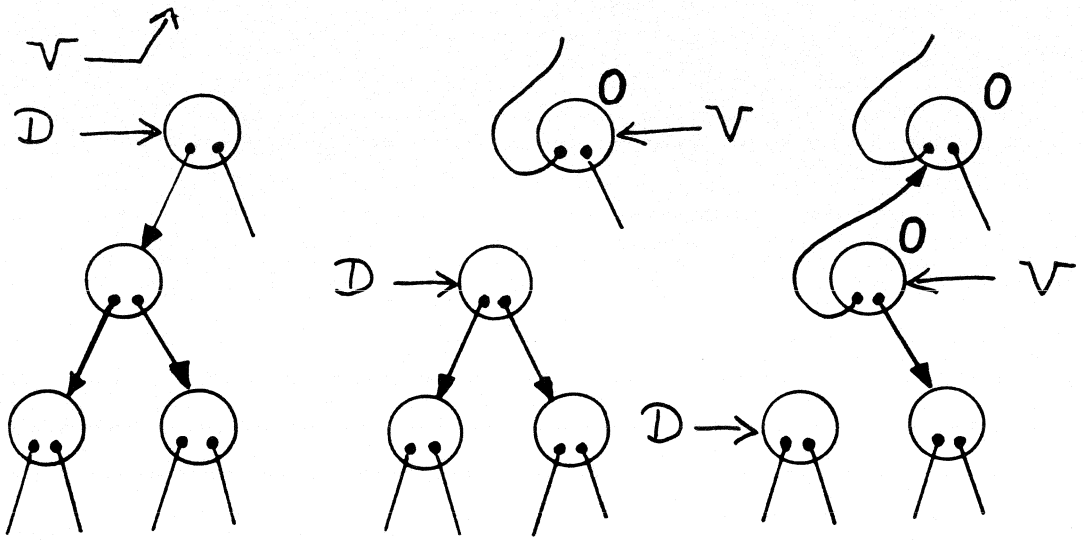
Bij onderstaand algoritme wordt er voor gezorgd dat tijdens de wandeling een pad vanaf de (ouder van de) bezochte knoop naar de wortel bijgehouden wordt. Dit pad wordt gevormd door de takken niet naar beneden (naar een kind) te laten wijzen, maar omhoog (naar de ouder). Bezoeken we de linker subboom van een knoop, dan wijst de linker kind-pointer naar de vader van de knoop, evenzo voor de rechter subboom. Met behulp van deze informatie kunnen we dus tijdens de wandeling terug klimmen. Omdat we moeten kunnen zien of we op de terugweg de linker dan wel rechter pointer moeten volgen voorzien we elke knoop van de boom van een Tag-veld. Deze heeft waarde 1 wanneer de rechter pointer naar de vader wijst en de waarde 0 in andere gevallen (bij een gewone knoop of wanneer de linker pointer omhoog wijst).

Omdat het algoritme elke knoop drie maal bezoekt en deze bezoeken (in tegenstelling tot Lindstrom's algoritme) kan onderscheiden, kunnen we het gebruiken om naar believen een pre-orde, symmetrische of post-orde wandeling uit te voeren. Daartoe moeten dan in het algoritme passende opdrachten toegevoegd worden.

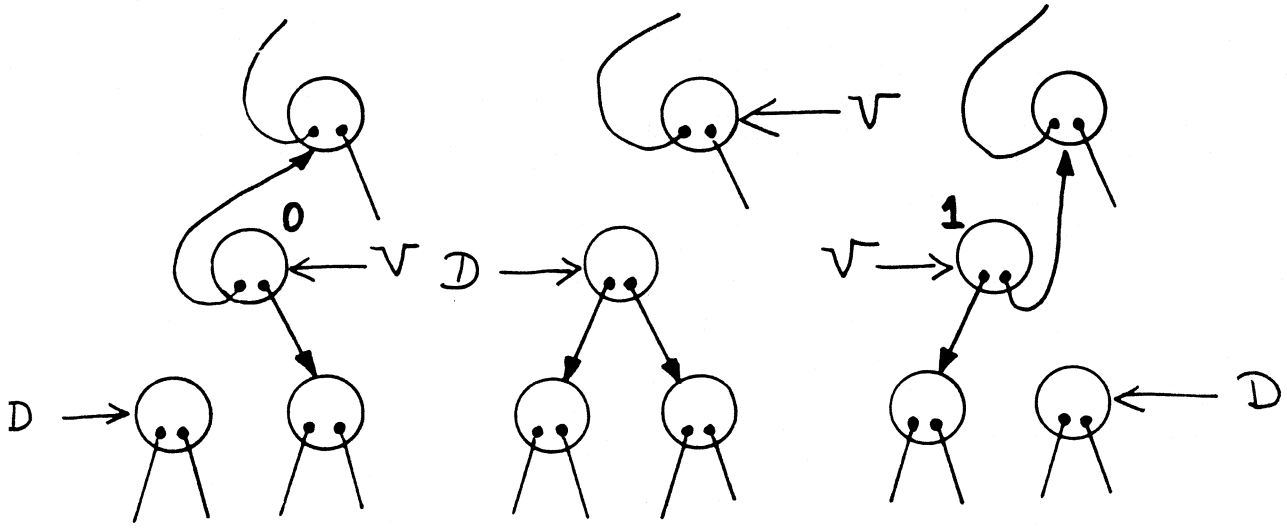
```
                    { De notatie a ← b ← c ← d betekent     }
                    {    a := b; b := c; c := d             }
```

Algoritme : Boomwandeling met linkomkering

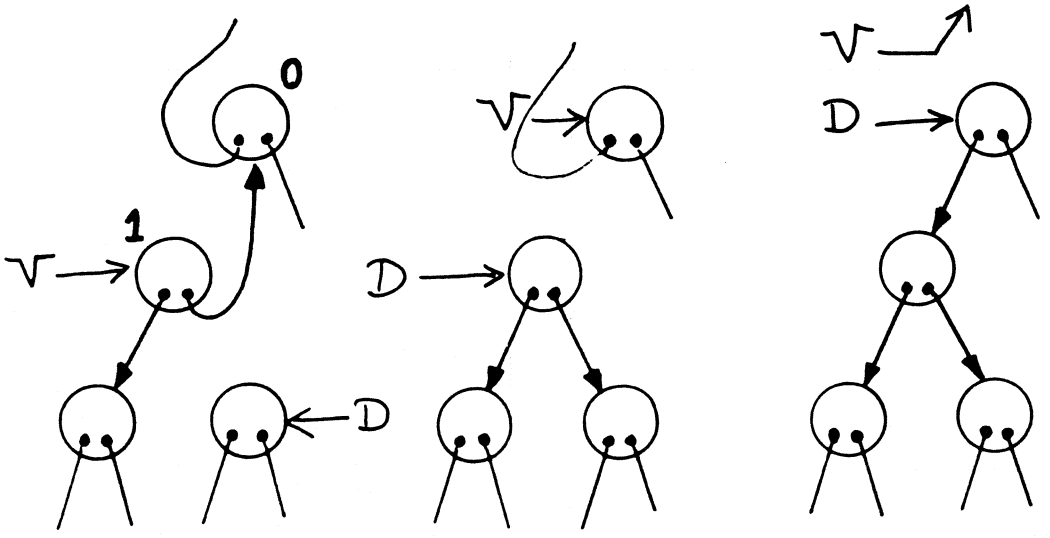
```
if Wortel ≠ NIL
then Vader := NIL; Deze := Wortel;
     Bezoek := 1; Klaar := false;
     repeat case Bezoek of
          1:   Volgend := Deze↑.Links;                    { pre-orde }
               if Volgend = NIL
               then Bezoek := 2
               else Deze↑.Links ← Vader ← Deze ← Volgend
               fi;
          2:   Volgend := Deze↑.Rechts;                   { symmetrisch }
               if Volgend = NIL
               then Bezoek := 3
               else Deze↑.tag := 1;
                    Deze↑.Rechts ← Vader ← Deze ← Volgend;
                    Bezoek := 1;
               fi;
          3:   if Vader ≠ NIL                             { post-orde }
               then if Vader↑.tag = 0
                    then Hulp ← Vader↑.Links ← Deze ← Vader ← Hulp;
                         Bezoek := 2
                    else Vader↑.tag := 0;
                         Hulp ← Vader↑.Rechts ← Deze ← Vader ← Hulp
                    fi
               else Klaar := true;
               fi
        endcase
     until Klaar;
fi;
```



Bezoek = 1 Bezoek = 1 Bezoek = 1



Bezoek = 3 Vader↑.Tag = 0 Bezoek = 2 Bezoek = 1



Bezoek = 3 Vader↑.Tag = 1 Bezoek = 3 Bezoek = 2

II. SPECIALE BOMEN EN TOEPASSINGEN

1. Bedrade Binaire Bomen

In de standaard pointer representatie van binaire bomen wordt veel ruimte ingenomen door NIL-pointers. Meer dan de helft van de pointers is een NIL-pointer, bij n knopen zijn er immers 2n pointers, waarvan er slechts n-1 als tak hoeven te fungeren. Om deze ruimte nuttig te gebruiken vervangen we elke rechter NIL-pointer door een pointer naar de symmetrische (LWR) opvolger van de knoop, linker NIL-pointers door een pointer naar de symmetrische voorganger. Deze nieuwe pointers worden *draden* genoemd. Om ze te onderscheiden van de oorspronkelijke takken voegen we extra velden toe aan de knopen van de boom; deze velden worden *tag's* genoemd. We maken een afspraak voor de twee knopen in de boom die geen voorganger of opvolger in de LWR ordening hebben. Deze knopen krijgen een linker tag resp. rechter tag die gelijk aan Draad is, maar de corresponderende pointer blijft NIL.

```
TYPE Wijzer = ↑Knoop;
     Knoop = RECORD
                Info : NaarEigenKeus;
                Links, Rechts : Wijzer;
                LTag, RTag : (Tak, Draad)
             END;
```

Het voordeel dat we voor deze ruimte investering terug krijgen is dat het nu mogelijk is om zonder extra hulpmiddelen een symmetrische boomwandeling te maken. De wandeling maakt alleen gebruik van rechter draden, de linker draden kunnen eventueel gebruikt worden om - op dezelfde manier - voorgangers te vinden. Merk op dat het algoritme elke tak en elke draad één keer volgt tijdens de wandeling.

Algoritme : Symmetrische wandeling in een bedrade boom (LWR)

```
function LWR_Opvolger(N : Wijzer) : Wijzer;
    Hulp := N↑.Rechts;
    if N↑.RTag = Tak
    then while Hulp↑.LTag = Tak
         do   Hulp := Hulp↑.Links
         od
    fi;
    LWR_Opvolger := Hulp;
endfunction;
```

```
Deze := Wortel;                    { we veronderstellen Wortel ≠ NIL }
while Deze↑.Links ≠ NIL
do   Deze := Deze↑.Links                    {zoek eerste knoop}
od;
repeat
     bezoek(Deze);
     Deze := LWR_Opvolger(Deze)
until Deze = NIL;
```

Wanneer we gewone pointer implementatie van binaire bomen gebruiken kunnen we zonder al te veel moeite subbomen vervangen. Bij bedrade bomen moeten we er echter aan denken dat we ook een aantal draden moeten verhangen en niet alleen één enkele tak. Daarbij is het belangrijk dat we ons afvragen of er geen draden naar de oude subboom 'los' blijven hangen. Hieronder staat als voorbeeld het vervangen van subbomen bij bedrade bomen uitgewerkt.

Algoritme : vervangen rechter subboom door bedrade boom

```
                        { Rechter subboom van Deze wordt vervangen door }
                        { bedrade boom met wortel Wortel↑ .              }
Opvolg := Deze;
while Opvolg↑.RTag = Tak
do   Opvolg := Opvolg↑.Rechts
od;
Deze↑.Rechts := Wortel;
Laatst := Wortel;                { zoek laatste LWR-knoop in nieuwe boom }
while Laatst↑.Rechts ≠ NIL and Laatst↑.RTag = Tak
do   Laatst := Laatst↑.Rechts
od;
Laatst↑.Rechts := Opvolg↑.Rechts;
Eerst := Wortel;                 { zoek eerste LWR-knoop in nieuwe boom }
while Eerst↑.Links ≠ NIL and Eerst↑.LTag = Tak
do   Eerst := Eerst↑.Links
od;
Eerst↑.Links := Deze;
```

2. Optimale Binaire Zoekbomen

Een binaire boom heet een binaire zoekboom als elke knoop een sleutel bevat die groter is dan alle sleutels opgeborgen in de linker subboom en kleiner dan alle sleutels in de rechter subboom. Een direct gevolg van deze definitie is dat een symmetrische wandeling in een binaire zoekboom de sleutels van klein naar groot afloopt.

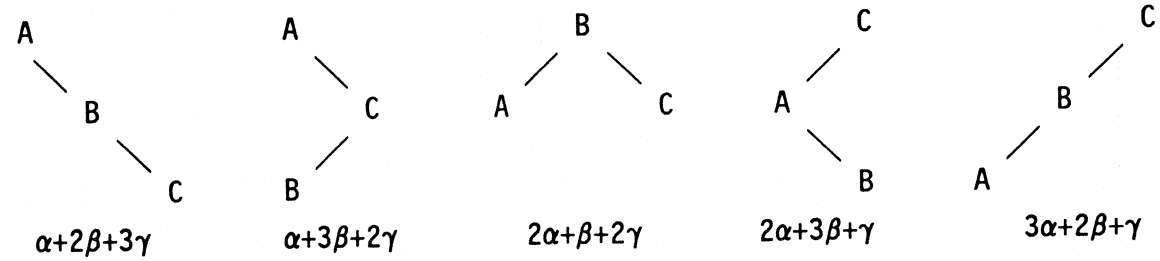
In zo'n zoekboom kunnen we op eenvoudige wijze de opgeslagen sleutels vinden. Bij iedere knoop is het duidelijk of we links dan wel rechts verder moeten zoeken.

We onderscheiden twee soorten gebruik van binaire zoekbomen. *Dynamisch* gebruik waarbij het aantal opgeslagen sleutels en de vorm van de boom tijdens het programma frequent veranderen en *statisch* gebruik waarbij de boom, na deze éénmalig te hebben opgesteld, niet meer verandert. We richten onze aandacht hier vooral op het laatste geval.

Dynamische zoekbomen. Bij dynamisch gebruik van een zoekboom verandert de vorm van de boom geregeld. Een optimale zoektijd hebben we in complete bomen; deze is logaritmisch in het aantal knopen van de boom. Het is zaak om de vorm van de boom niet al te asymmetrisch te laten worden, daar anders de zoektijd lineair kan worden. Een manier om dat te doen is gebruik te maken van bomen waarvan de hoogte van de subbomen 'gebalanceerd' is. Een speciaal geval zijn de AVL-bomen (zie algoritmiek). Deze hebben een logaritmische zoektijd, terwijl tevens niet al te veel werk nodig is om de boom in goede vorm te houden bij het toevoegen van nieuwe sleutels.

Statische zoekbomen. Wanneer we de sleutels weten die in de boom moeten worden opgeborgen en ook nog enige informatie hebben over de frequenties waarmee deze sleutels in de boom moeten worden opgezocht, dan zal het de moeite lonen om een zo gunstig mogelijke boom te bepalen.

Voorbeeld. Gezocht worden sleutels A, B en C met frequenties α,β en γ ($\alpha+\beta+\gamma = 1$). Wanneer een sleutel zich op nivo k van de zoekboom bevindt kost het k vergelijkingen om de sleutel te vinden. Voor de vijf mogelijke zoekbomen voor drie sleutels zijn de respectievelijke aantallen verwachte vergelijkingen bij zoekacties als volgt:

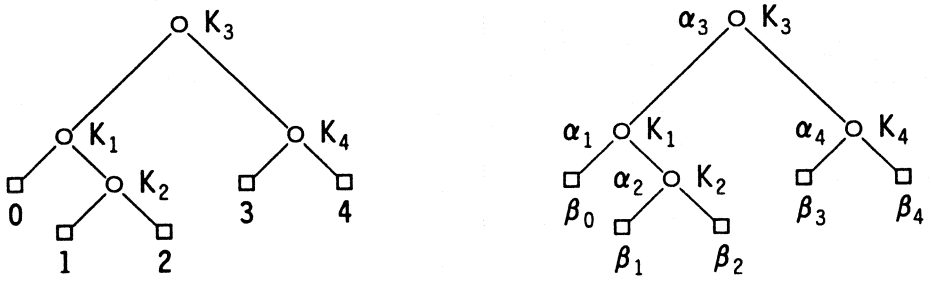


$\alpha+2\beta+3\gamma$ $\alpha+3\beta+2\gamma$ $2\alpha+\beta+2\gamma$ $2\alpha+3\beta+\gamma$ $3\alpha+2\beta+\gamma$

Wanneer nu $\alpha = 0.3$, $\beta = 0.2$ en $\gamma = 0.5$, dan zijn deze waarden gelijk aan 2.2, 1.9, 1.8, 1.7 en 1.8. De vierde boom verdient daarmee de voorkeur. ■

Voor een klein aantal sleutels kan dit uitzoeken van de beste boom nog met de hand gebeuren. We rekenen eenvoudig alle bomen na. Gezien het grote aantal mogelijke binaire bomen wordt dat al gauw ondoenlijk. We zoeken dus naar een redelijk algoritme.

We zullen eerst definieren hoe we de diverse zoekbomen tegen elkaar af zullen wegen. Zoals we in het voorbeeld gezien hebben gebruiken we het te verwachten aantal vergelijkingen als maat. Omdat we niet alleen zoeken naar elementen die in de boom voorkomen is het handig om de boom te voorzien van bladeren die plaatsen representeren van alle mogelijke niet-aanwezige sleutels. Zo'n boom heet dan wel een uitgebreide binaire zoekboom.



Veronderstel dat n sleutels $K_1 < \ldots < K_n$ gegeven zijn, met hun verwachte zoekfrequenties $\alpha_1,\ldots,\alpha_n$. Tevens weten we de frequenties $\beta_0, \beta_1,\ldots,\beta_n$ voor sleutels die niet in de boom staan; de kans dat een sleutel K met $K_i < K < K_{i+1}$ wordt aangeboden is gelijk aan β_i. Hierbij is dan natuurlijk β_0 de kans dat we een sleutel kleiner dan K_1 moeten zoeken, en β_n de kans op een sleutel groter dan K_n. De frequenties α_i heten de interne frequenties, terwijl de β_i de externe frequenties genoemd worden.

Voor een binaire zoekboom T voor de sleutels is het *verwachte aantal vergelijkingen* gelijk aan:

$$\Sigma_{1\le k\le n}\ \alpha_k\cdot(\text{nivo knoop } {}^{\circ}K_k) + \Sigma_{0\le k\le n}\ \beta_k\cdot((\text{nivo blad } {}^{\square}k) - 1)$$

Deze grootheid noemen we ook wel de *gewogen padlengte* van T, maar die naam is niet helemaal nauwkeurig; we zouden in dat geval ook bij de gewone knopen één van het nivo af moeten halen - de lengte van het pad van de wortel naar een knoop is immers één minder dan het nivo van een knoop. Een boom waarvan deze waarde minimaal is (bij gegeven α_i en β_i) heet een optimale zoekboom.

Om nu een manier te vinden om optimale zoekbomen te bepalen kijken we niet alleen naar de gehele boom, maar ook naar subbomen van die boom. Laat dus T' een subboom van T zijn met bladeren $\square i$ tot en met $\square j$. Voor T' definieren we de gewogen padlengte als:

$E(T') = \sum_{i<k\leq j} \alpha_k \cdot (\text{nivo } \circ K_k \text{ in } T') + \sum_{i\leq k\leq j} \beta_k \cdot ((\text{nivo } \square k \text{ in } T') - 1)$.

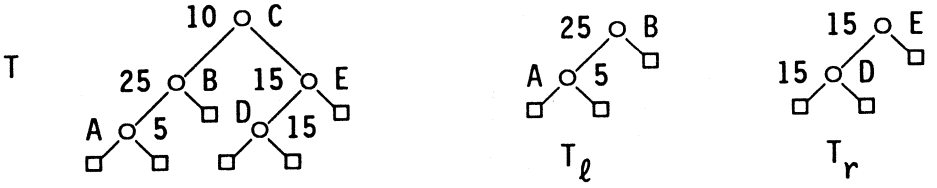
We zullen ook het <u>totale gewicht</u> van T', $F(T') = \alpha_{i+1}+\ldots+\alpha_j+\beta_i+\beta_{i+1}+\ldots+\beta_j$, gebruiken. Alweer is de term gewogen padlengte niet helemaal gelukkig gekozen.

Omdat het totale gewicht van T' niet gelijk aan 1 hoeft te zijn, is het verwachte aantal vergelijkingen voor de subboom niet gelijk aan $E(T')$, maar aan $E(T')/F(T')$.

Laat T een zoekboom zijn met subbomen T_ℓ en T_r, neem aan dat K_w de sleutel is die zich in de wortel van T bevindt. Het nivo van een knoop in de subbomen K_i is één minder dan het nivo van die zelfde knoop in de gehele boom T. Dit heeft tot gevolg dat elk gewicht in $E(T_\ell)$ en $E(T_r)$ één maal minder meegeteld wordt dan in $E(T)$, in het bijzonder geldt dit voor het gewicht α_w van de wortel dat niet meetelt bij de subbomen. We kunnen het verwachte aantal vergelijkingen van de boom dus uitdrukken in de verwachte aantallen voor de subbomen:

$$E(T) = E(T_\ell) + E(T_r) + F(T),$$
$$F(T) = F(T_\ell) + F(T_r) + \alpha_w \text{ - waarbij } K_w \text{ in de wortel van } T \text{ staat.}$$

Voorbeeld. Neem aan dat de sleutels A, B, C, D, E gezocht worden met respectievelijke frequenties 5%, 25%, 10%, 15% en 15%. De externe frequenties bedragen alle 5%. Beschouw de volgende zoekboom T.



Er geldt (in honderdsten uitgedrukt): $\alpha_w = 10$

$F(T_\ell) = 45, \quad F(T_r) = 45, \quad F(T) = 100 = F(T_\ell) + F(T_r) + \alpha_w$;

$E(T_\ell) = 2\cdot5 + 1\cdot25 + 2\cdot5 + 2\cdot5 + 1\cdot5 = 60,$

$E(T_r) = 2\cdot15 + 1\cdot15 + 2\cdot5 + 2\cdot5 + 1\cdot5 = 70,$

$E(T) = 3\cdot5 + 2\cdot25 + 1\cdot10 + 3\cdot15 + 2\cdot15$
$+ 3\cdot5 + 3\cdot5 + 2\cdot5 + 3\cdot5 + 3\cdot5 + 2\cdot5 = 230 = E(T_\ell) + E(T_r) + F(T).$ ■

Uit het voorafgaande volgt onmiddelijk dat de twee subbomen van een optimale zoekboom zelf ook optimaal moeten zijn (voor de sleutels in die bomen). Als immers één der subbomen niet optimaal is dan kunnen we ook de gehele boom verbeteren door de subboom te vervangen. Deze observaties leiden tot een recursieve methode om het verwachte aantal vergelijkingen van de optimale zoekboom te berekenen. We voeren enige notatie in.

$T_{i,j}$ *optimale zoekboom* voor $K_i, \ldots, K_j$

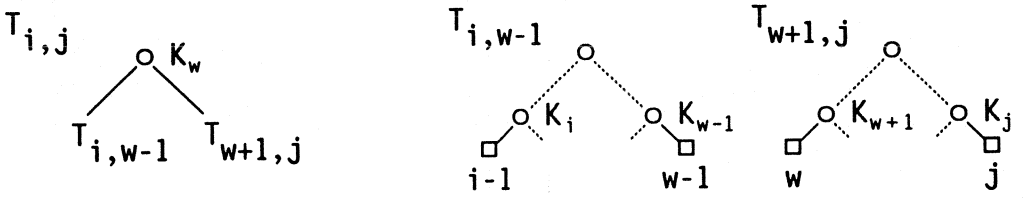
bijpassende gewichten $\alpha_i, \ldots, \alpha_j$ (succes)

$\beta_{i-1}, \beta_i, \ldots, \beta_j$ (niet aangetroffen)

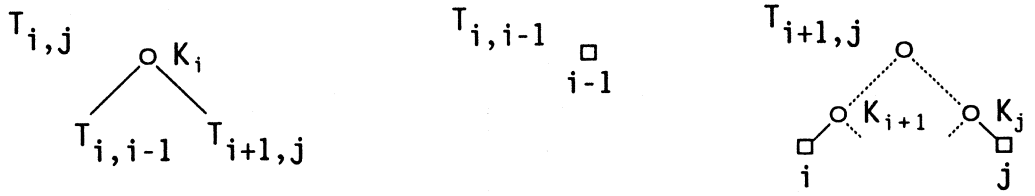
$F_{i,j} = F(T_{i,j})$ *totaal gewicht van* $T_{i,j}$: $\beta_{i-1}+\alpha_i+\beta_i \ldots \alpha_j+\beta_j$.

$E_{i,j} = E(T_{i,j})$ *gewogen padlengte voor* $T_{i,j}$.

Om $E_{i,j}$ (en $T_{i,j}$) te berekenen moeten we allereerst de sleutel K_w bepalen die zich in de wortel van de boom bevindt. We moeten díe w kiezen waarvoor de som van de gewogen padlengtes $E_{i,w-1} + E_{w+1,j}$ minimaal is - dit zijn de waarden die horen bij de subbomen van $T_{i,j}$.



We moeten ook rekening houden met de mogelijkheid dat de eerste of laatste sleutel (dus K_i of K_j) in de wortel geplaatst wordt (dit gebeurt bijvoorbeeld wanneer we $T_{i,i}$ proberen te bepalen, of wanneer die sleutel een erg groot relatief gewicht heeft). We krijgen dan te maken met een bijzonder geval van bovenstaand plaatje:



Om een eenvoudig stel recursieve vergelijkingen op te kunnen stellen is het dus zinvol om ook voor elke i de optimale zoekboom $T_{i,i-1}$ te definieren; zoals hierboven geillustreerd bestaat deze boom uit het enkele blad □ i-1.

Met deze afspraken kunnen we de optimale gewogen padlengte berekenen met behulp van de volgende vergelijkingen.

$$\begin{cases} F_{i,i-1} = \beta_{i-1}, & 1 \le i \le n \\ F_{i,j} = \beta_{i-1}+\alpha_i+\beta_i+\ldots+\alpha_j+\beta_j & 1 \le i \le j \le n \end{cases}$$

$$\begin{cases} E_{i,i-1} = 0, & 1 \le i \le n \\ E_{i,j} = F_{i,j} + \min\limits_{i \le w \le j} (E_{i,w-1} + E_{w+1,j}) & 1 \le i \le j \le n \end{cases}$$

Het aantal aanroepen van een dergelijke recursieve procedure om $E_{1,n}$ te berekenen is helaas exponentieel in n. Dit is geen echte verbetering ten opzichte van het narekenen van alle mogelijke zoekbomen. De oorzaak is natuurlijk dat we een aantal waarden steeds weer opnieuw moeten berekenen.

Stelling. Het aantal aanroepen om $E_{1,n}$ met een recursieve procedure te berekenenen volgens de hierboven gegeven vergelijkingen is gelijk aan 3^n.

Bewijs. We bewijzen de stelling met inductie. Onze bewering is dat het berekenen van $E_{i,j}$ in totaal 3^{j-i+1} aanroepen kost. Laat $a_{i,j}$ het benodigde aantal zijn.

Basis. Omdat $E_{i,i-1} = 0$, kost het berekenen van deze waarde slechts één aanroep; $a_{i,i-1} = 1$.

Inductiestap. Wanneer $E_{i,j}$ aangeroepen wordt, dan wordt vervolgens $E_{i,w-1} + E_{w+1,j}$ voor $i \le w \le j$ berekend. Er geldt dus

$$a_{i,j} = 1 + \sum_{w=i}^{j} a_{i,w-1}+a_{w+1,j} = 1 + \sum_{w=i}^{j} (3^{w-i}+3^{j-w}) = 1 + 2\cdot\sum_{k=0}^{j-i} 3^k$$

$$= 1 + 2\cdot\frac{3^{j-i+1}-1}{3-1} = 3^{j-i+1}. \quad \blacksquare$$

Dynamisch programmeren. Bovenstaande recursieve methode om de optimale zoekboom te berekenen is typisch *top-down*: het probleem wordt telkens teruggebracht tot kleinere deelproblemen. Omdat deze deelproblemen elkaar overlappen worden een aantal berekeningen steeds opnieuw uitgevoerd. Bij dynamisch programmeren draaien we de aanpak om. We beschouwen eerst de kleinste problemen, lossen deze op en bewaren de oplossing in een tabel. We gebruiken de gevonden oplossingen in stapsgewijs grotere problemen totdat we het oorspronkelijke probleem bereikt hebben; De methode werkt dus *bottom-up*.

Wanneer we met deze methode de optimale binaire zoekboom berekenen, bepalen we eerst de waarden $E_{i,i}$, dan de waarden $E_{i,i+1}$, enzovoorts.

Algoritme : bepalen optimale binaire zoekboom

```
for i := 1 to n+1                                   { initialisatie }
do   E[i,i-1] := 0;
     F[i,i-1] := β_{i-1}
od;
for k := 0 to n-1
do   for i := 1 to n-k
     do   j := i+k;
          F[i,j] := F[i,j-1] + β_j + α_j;
          kies w met i≤w≤j zó dat E[i,w-1] + E[w+1,j] minimaal;   {*}
          Wortel[i,j] := w;
          E[i,j] := F[i,j] + E[i,w-1] + E[w+1,j];
     od
od;
```

Efficientie. Wanneer we in het algoritme w willen bepalen, moeten we w de waarden i, i+1,..., j laten doorlopen, waarbij j = i+k; dit zijn dus k+1

waarden. Deze binnenste loop is genest binnen twee andere loops, die van i en van k. Het valt aan te tonen dat het totaal aantal keren dat we E[i,w-1] + E[w+1,j] uit de tabel berekenen evenredig is met n^3, een aanzienlijke verbetering ten opzichte van de recursieve methode. Er is echter nog een versnelling mogelijk. Het is tamelijk onlogisch dat wanneer w de wortel is van $T_{i,j-1}$, de wortel van $T_{i,j}$ plotseling kleiner dan w is. Intuitief gezien proberen we namelijk de boom zoveel mogelijk gebalanceerd te houden en toevoegen van een knoop aan de rechterzijde betekent dan niet een verschuiving naar links van de wortel. Met enig doorzettingsvermogen kan dit feit ook bewezen worden. Dit impliceert dat het aantal waarden dat we telkens voor w invullen kleiner genomen kan worden. Verander in bovenstaand algoritme daarom de met * gemerkte regel in:

kies w *met* Wortel[i,j-1] ≤ w ≤ Wortel[i+1,j]
zó dat E[i,w-1] + E[w+1,j] *minimaal*;

Het blijkt dat het algoritme met deze verbetering nog maar kwadratisch veel stappen nodig heeft.

Voorbeeld. We plaatsen de sleutels A, B, C, D en E in een binaire zoekboom, met aangeboden zoekfrequenties (de α_i waarden) 5%, 25%, 10%, 15% en 15%. De frequenties β_i voor sleutels niet in de tabel zijn telkens 5%. We geven de tabellen voor F, E en Wortel zoals zij door het algoritme berekend worden. Voor Wortel geven we alle mogelijke waarden; soms zijn er meerdere optimale zoekbomen. Hebben we bij het 'verbeterde' algoritme in dit soort gevallen eenmaal een wortel uitgekozen dan zal dit ook effect hebben voor de latere gevonden wortels.

F	i=1	2	3	4	5	6
k=-1	5	5	5	5	5	5
0	15	35	20	25	25	
1	45	50	40	45		
2	60	70	60			
3	80	90				
4	100			$F_{i,j}$	j=i+k	

E	i=1	2	3	4	5	6
k=-1	0	0	0	0	0	0
0	15	35	20	25	25	
1	60	70	60	70		
2	95	130	105			
3	155	185				
4	220			$E_{i,j}$	j=i+k	

Wortel	i=1	2	3	4	5
k=0	1	2	3	4	5
1	2	2	4	4,5	
2	2	2,3	4		
3	2	4			
4	2,4		$Wortel_{i,j}$	j=i+k	

We geven enige berekeningen voor de minimale waarden $E_{i,w-1} + E_{w+1,j}$. We hebben geen rekening gehouden met de gesuggereerde versnelling van het algoritme. Bij de berekening van het laatste element uit de tabel $E_{1,5}$ (i=1, k=4) hadden we ons bijvoorbeeld kunnen beperken tot $w \in \{2,3,4\}$. Wanneer het algoritme met de hand uitgevoerd wordt valt al snel op welke waarden bij elkaar opgeteld dienen te worden voor de berekening van een nieuwe waarde. Vergeet niet $F_{i,j}$ bij het bepaalde minimum op te tellen.

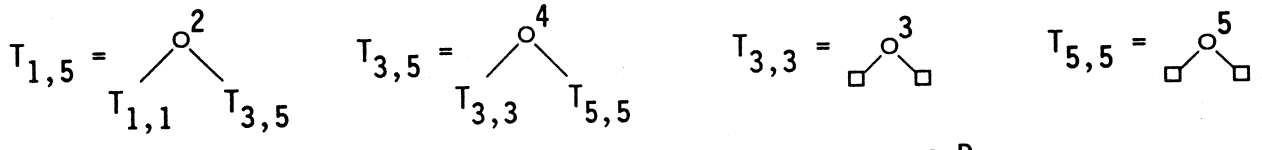
k	i	j	w	$E_{i,w-1}+E_{w+1,j}$
1	1	2	1	0 + 35
			2	**15 + 0** ←
2	1	3	1	0 + 70
			2	**15 + 20** ←
			3	60 + 0

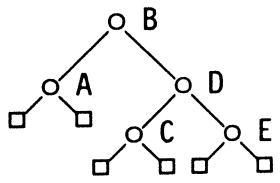
k	i	j	w	$E_{i,w-1}+E_{w+1,j}$
4	1	5	1	0 + 185
			2	**15 + 105** ←
			3	60 + 70
			4	**95 + 25** ←
			5	155 + 0

getallen nodig voor $E_{1,5}$:

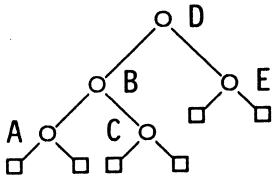
0	0	0	0	0	0
15	35	20	25	25	
60	70	60	70		
95	130	105			
155	185				
...					

Met behulp van het array Wortel kunnen de optimale zoekbomen op efficiente wijze recursief bepaald worden. We halen uit de tabel dat $T_{1,5}$ 2 (of 4) als wortel heeft (i=1, j=5 dus k=4). De subbomen $T_{1,1}$ en $T_{3,5}$ hebben de wortel 1 (vanzelfsprekend) en 4 (dit zoeken we weer op in de tabel Wortel, i=3 en k=2). De subbomen $T_{3,3}$ en $T_{5,5}$ zijn dan duidelijk, zij hebben wortel 3 en 5 (en geen andere knopen). Ook de andere optimale zoekboom (die met wortel 4) kan op deze wijze bepaald worden.





de gezochte boom



alternatief (wortel 4)

■

3. De Heap en de Priority Queue

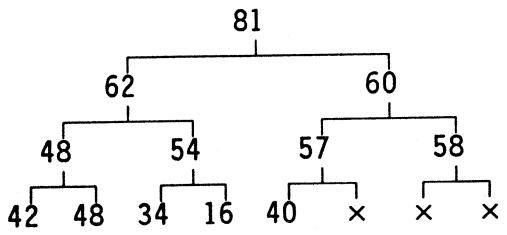
Een heap is - zoals we ons nog herinneren van het college Algoritmiek - een complete binaire boom waarvan de knopen voldoen aan de *heap-eigenschap*. Dit betekent dat elke knoop van de boom een sleutel bevat die groter is dan (of gelijk is aan) de sleutels van zijn kinderen (voorzover deze bestaan). Omdat een heap een complete binaire boom is, is het gebruikelijk om de array representatie voor de boom te kiezen (zie eerder).

Wanneer we de waarde van een knoop veranderen zal de resulterende boom in het algemeen geen heap meer zijn. Het blijkt redelijk eenvoudig om de heap-eigenschap weer te herstellen. Is de nieuwe waarde van de knoop groter dan de waarde van zijn vader dan verwisselen we deze twee waarden en herhalen het proces hogerop in de boom totdat de heap-eigenschap is hersteld. De waarde *borrelt* in de boom *omhoog*. Omgekeerd, wanneer de waarde kleiner is dan de waarde van tenminste één van de kinderen, dan verwisselen we de waarde met de grootste van de kinderen en herhalen dit in de richting van de bladeren totdat weer een heap is ontstaan. Deze operaties zijn vrij efficient, het verwisselen gebeurt op slechts één pad van de wortel naar de bladeren en kost dus maximaal een tijd die logaritmisch is in de grootte van de boom (dat wil zeggen $O(lg\ n)$, als de boom n sleutels bevat).

We nemen aan dat de benodigde type-declaraties en procedures reeds gegeven zijn; vergelijk het geleerde bij het college Algoritmiek.

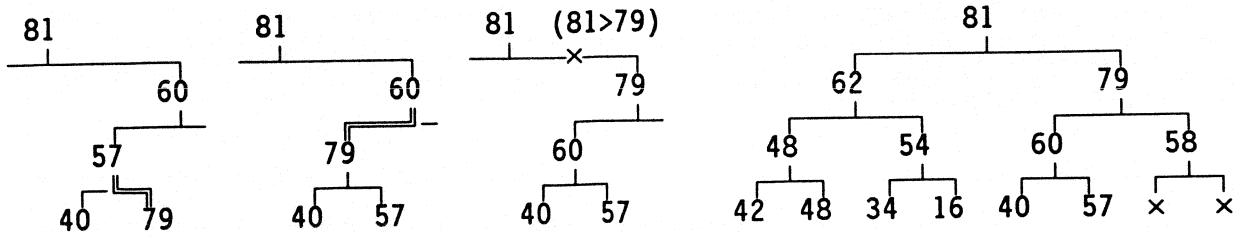
```
TYPE Heap = ARRAY[1..Max] OF Integer;
BorrelOmhoog(VAR H: Heap; Grootte, Plek: Integer);
                    { Zolang de waarde H[Plek] groter is dan de waarde  }
                    { van de ouder verwisselen we deze waarden.         }
ZinkNeer(VAR H: Heap; Grootte, Plek: Integer);
                    { Zolang de waarde H[Plek] kleiner is dan de waarde }
                    { van een kind verwisselen we deze waarde met die   }
                    { van grootste kind. H[Grootte] is laatste element. }
```

Voorbeeld. Het toevoegen aan en het verwijderen uit een heap.

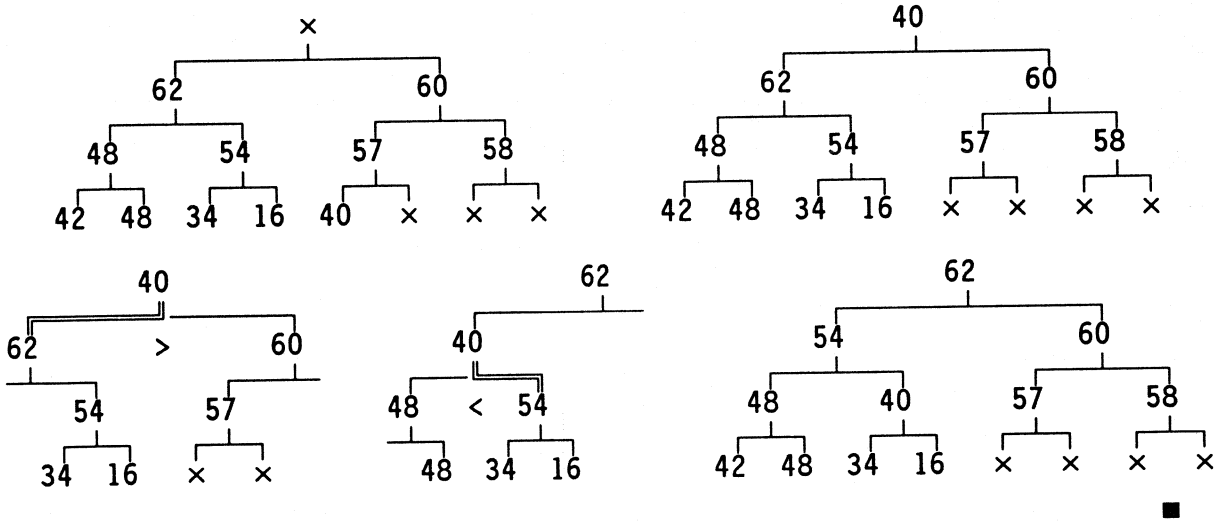


In het array: 81 62 60 48 54 57 58 42 48 34 16 40

— In de gegeven heap voegen we 79 toe. Plaats daartoe eerst 79 in de eerste vrije positie. Borrel daarna 79 omhoog (totdat een plaats is bereikt waar de ouder groter is dan 79).



— In de oorspronkeljke heap verwijderen we het grootste element (dit staat in de wortel). Om weer de eerste plek van de heap te vullen plaatsen we het laatste element in de wortel. Om de heap-eigenschap weer te herstellen zinken we het dan weer neer.



■

We kunnen met de gegeven procedures een programma schrijven om van een willekeurig gevuld array een heap te maken. Dit kan bijvoorbeeld door steeds één element meer bij de heap te betrekken en deze dan omhoog te borrelen (vergelijk met toevoegen in bovenstaand voorbeeld).

```
PROCEDURE MaakHeapLangzaam(VAR H: Heap; Grootte: Integer);
VAR I: Integer;
BEGIN
     FOR I := 2 TO Grootte DO BorrelOmhoog(H, I, I);
END;
```

Op deze manier duurt het proces $O(n \cdot lg\ n)$ stappen, waarbij n het aantal array elementen is. Deze waarde krijgen we door ons te realiseren dat in het slechtste geval steeds ieder element helemaal naar boven borrelt. Dit kost dan $\lfloor lg\ 2 \rfloor + \lfloor lg\ 3 \rfloor + \ldots + \lfloor lg\ n \rfloor$ verwisselingen, wat op $O(n \cdot lg\ n)$ uitkomt.

Door juist aan de andere kant te beginnen krijgen we een snellere methode (zie Algoritmiek). Bij nauwkeurige analyse blijkt het in lineaire tijd te werken.

```
PROCEDURE MaakHeap(VAR H: Heap; Grootte: Integer);
VAR I: Integer;
BEGIN
     FOR I := Grootte DIV 2 DOWNTO 1 DO ZinkNeer(H, Grootte, I);
END;
```

Priority Queue. Een belangrijke toepassing van de heap-structuur, naast de sorteermethode heapsort, is als representatie van de priority queue. Als abstracte datastructuur gezien is een priority queue een verzameling S (van in ons geval gehele getallen) waarop we de volgende operaties toelaten: test of $S = \emptyset$, toevoegen van een element aan S, en het vinden en verwijderen van het *grootste* element. Een priority queue kan vergeleken worden met de stack en de queue met het verschil dat we daar geïnteresseerd zijn in het *laatst* resp. het *eerst toegevoegde* element. Bij een priority queue wordt wel gesproken van een grootste-in-eerste-uit mechanisme. De procedures BorrelOmhoog en ZinkNeer kunnen gebruikt worden om de toevoeg en verwijder instructies te implementeren. We laten de preciese uitwerking aan de lezer over.

Efficientie; andere methoden. Met de boven voorgestelde procedures kosten de verwijder en toevoeg instructies een aantal verwisselingen dat logaritmisch is in het aantal opgeslagen getallen. Ook een reeds bekende datastructuur als de lijst kan gebruikt worden om een priority queue te representeren. We kunnen de lijst naar keuze ongesorteerd dan wel gesorteerd kiezen, en met pointers dan wel in een array geïmplementeerd. Dit heeft zijn invloed op de snelheid van de operaties.

	gereed maken	(plek vinden en) toevoegen	grootste (vinden en) verwijderen
Ongeordende lijst in array	$O(1)$	$O(1)$	$O(n)+O(1)$ $O(1)$
Geordende lijst in array	$O(n \cdot lg\ n)$	$O(n)+O(1)$ $O(lg\ n)+O(n)$	$O(1)$
Heap	$O(n)$	$O(lg\ n)$	$O(lg\ n)$

Bovenstaande lijst dekt bij lange na niet alle mogelijke representaties van de priority queue. Vanwege de belangrijkheid van de datastructuur, bv. bij simulatie van wachtrijen met processen, is er nogal veel onderzoek naar de representatie gedaan. Dit heeft geleid tot een groot aantal, naar smaak creative of exotische, op bomen gebaseerde representaties van de priority queue. We noemen *leftist trees*, *binomial queues* en *Fibonacci heaps*. De meeste

van genoemde representaties zijn ook efficiciënt in het samenvoegen van twee heaps, iets dat met een heap wat minder makkelijk kan. Tot slot dan maar weer een voorbeeld van een - Modula 2 - specificatie.

```
definition module PriQueue;                    (* boek van Kingston *)

    type Entry;

    procedure New(key: KeyType; value: ValueType): Entry;
    procedure KeyOf(x: Entry): KeyType;
    procedure ValueOf(x: Entry): ValueType;
    procedure Update(x: Entry; value: ValueType);

    type PriorityQueue;

    procedure Initialize(var Q: PriorityQueue);
    procedure Empty(Q: PriorityQueue): boolean;
    procedure Insert(x: Entry; var Q: PriorityQueue);
    procedure FindMin(var Q: PriorityQueue): Entry;
    procedure DeleteMin(var Q: PriorityQueue): Entry;

end PriQueue.
```

In enkele gevallen is men ook geïnteresseerd in operaties als de volgende, die elementen uit de priority queue kunnen verwijderen, twee prority queues kunnen verenigen tot één nieuwe, en de waarde van een element uit de queue kunnen veranderen.

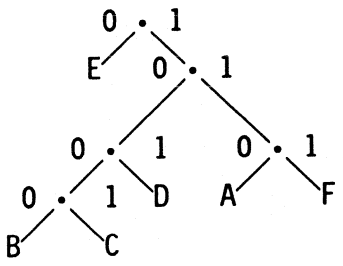
```
    procedure Delete(x: Entry; var Q: PriorityQueue);
    procedure Meld(P, Q: PriorityQueue): PriorityQueue;
    procedure DecreaseKey (x: Entry; key: KeyType; var Q: PriorityQueue);
```

4. Huffman Codes

Veronderstel dat we zes verschillende letters hebben. We kunnen deze letters coderen door voor elk drie bits te gebruiken; een binair gecodeerde boodschap is dan drie maal zo lang als de originele boodschap. Kan dit beter? Er blijkt inderdaad soms een betere codering gevonden te kunnen worden als we tenminste iets weten over de frequentieverdeling van de zes verschillende letters.

Voorbeeld. Zes letters, hun frequenties en codering.

A	18%	110
B	10%	1000
C	5%	1001
D	10%	101
E	40%	0
F	17%	111



Gemiddeld (in een tekst waarin de letters in de aangegeven frequenties voorkomen) gebruikt deze codering 2,35 bits per letter. Dit scheelt één vijfde ten opzichte van de voor de hand liggende drie bits codering.
De code

001000101111010101000011101101011001

dient als volgt ontleed te worden

0·0·1000·101·111·0·101·0·1000·0·111·0·110·101·1001

en wordt dan gedecodeerd tot

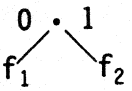
E E B D F E D E B E F E A D C. ■

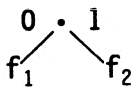
Merk op dat de codering uit bovenstaand voorbeeld aan een belangrijke voorwaarde voldoet. Er is geen codering van een letter die de codering van een andere letter als prefix heeft. Deze *prefix-eigenschap* zorgt er voor dat decodering van de boodschap ondubbelzinnig kan gebeuren; er zijn geen twee teksten met dezelfde codering. Een code die aan de prefix-eigenschap voldoet heet een Huffman code. Vanwege die eigenschap kunnen we een Huffman code representeren als binaire boom, waarbij de bladeren de codeletters dragen. Linker en rechter takken corresponderen met de codeletters 0 en 1 respectievelijk. De efficientie van de code (sommeer voor elke letter de frequentie maal de lengte van de codering) is dan precies de gewogen padlengte van de boom (waarbij dan alleen aan de bladeren gewichten zijn toegekend). Er is een eenvoudig (recursief) algoritme om een optimale Huffman code te vinden.

Algoritme : het bepalen van een optimale Huffman code

Gegeven de gewichten $f_1 \leq f_2 \leq \ldots \leq f_n$, *met* $n \geq 2$.

if $n = 2$

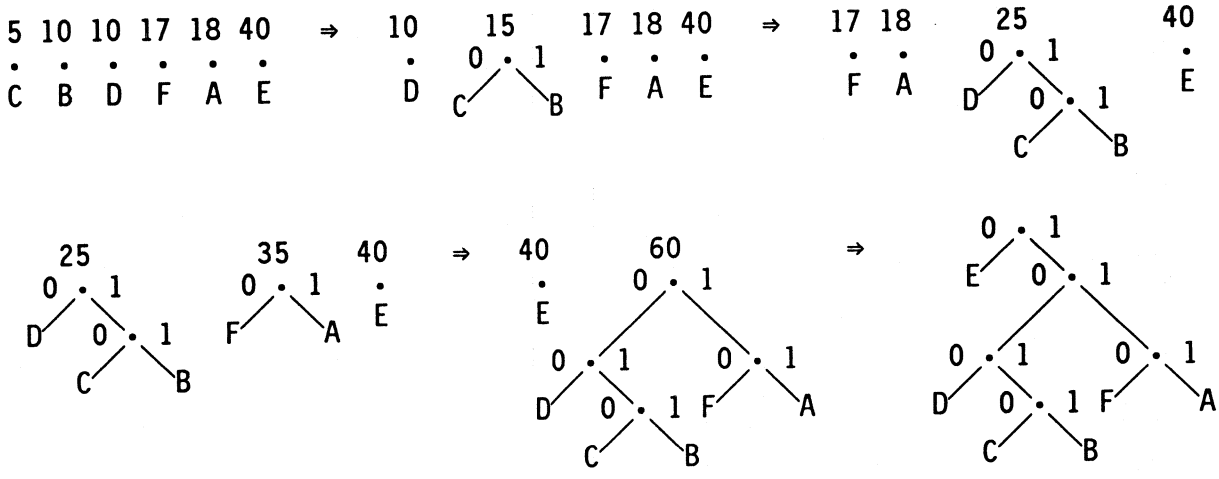
then *Een optimale boom is* 

else *Bepaal een optimale boom voor de gewichten* $f_1+f_2, f_3, \ldots, f_n$
(na deze op grootte gesorteerd te hebben);
Vervang in de verkregen boom het blad f_1+f_2 *door* 

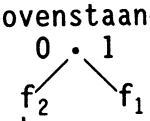
fi;

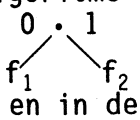
Het bovenstaande algoritme is recursief opgeschreven. We kunnen het ook zonder recursie formuleren: Vat elk gewicht op als een boom met één knoop. Herhaal de volgende stappen totdat er nog één boom over is. Neem de twee kleinste gewichten en voeg deze samen tot een boom; geef deze boom als gewicht de som van de oorspronkelijke gewichten.

Voorbeeld. We berekenen een Huffman code voor de gegeven frequenties.



■

Op deze manier wordt een andere codering dan de eerder gegeven code verkregen. Het is echter in te zien dat het verwisselen van de twee kleinste gewichten in het bovenstaande algoritme niet erg essentieel is; dat wil zeggen dat we net zo goed 

als 

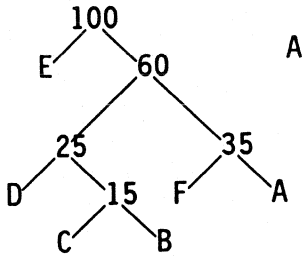
in de boom kunnen hangen. In de boom verwisselen dan links en rechts en in de code 0 en 1 van plaats.

Correctheid. We zullen laten zien dat de boom die we met het algoritme vinden inderdaad optimaal is. De bewering is: er bestaat geen boom met dezelfde gewichten in de bladeren en een kleinere gewogen padlengte. We maken daartoe twee observaties.

Eén. We mogen aannemen dat de optimale boom een knoop heeft met als kinderen de twee bladeren met de kleinste gewichten. De bladeren met de kleinste gewichten bevinden zich op het hoogste nivo van de boom (dwz. het

verst van de wortel) omdat we anders door verwisseling met een verder gelegen blad een kleinere gewogen padlengte kunnen krijgen. Kijk nu naar alle bladeren die op maximale afstand van de wortel liggen. Door verwisseling van plaats kunnen we ervoor zorgen dat de twee kleinste gewichten kinderen van dezelfde knoop zijn. Dit verandert de waarde van de gewogen padlengte niet.

Twee. Geef elke interne knoop van een Huffman boom een waarde die gelijk is aan de som van alle gewichten van bladeren in de subboom met die knoop als wortel. Tellen we alle waarden op dan vinden we precies de gewogen padlengte daar elk blad op alle lagere nivo's precies één keer meetelt.

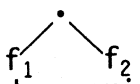
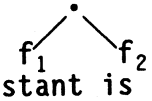


A 18 B 10 C 5 D 10 E 40 F 17

gewogen padlengte

$3 \cdot 18+4 \cdot 10+4 \cdot 5+3 \cdot 10+1 \cdot 40+3 \cdot 17$

$= 100+60+25+35+15 = 235$

Ten gevolge hiervan kunnen we de knoop  door een blad met gewicht f_1+f_2 veranderen, zonder dat de waarden van de overige interne knopen veranderen. Deze operatie vermindert de totale gewogen padlengte dus met precies f_1+f_2. Elke boom voor de gewichten f_1+f_2, $f_3,\ldots,$ f_n levert op die manier een boom voor de gewichten f_1, $f_2,\ldots,$ f_n met een knoop  (en andersom). Omdat het verschil tussen de gewogen padlengtes constant is correspondeert een optimale boom voor f_1+f_2, $f_3,\ldots,$ f_n met een optimale boom voor f_1, $f_2,\ldots,$ f_n via de constructie van het algoritme.

III. TABELLEN

0. SPECIFICATIE EN IMPLEMENTATIE

Specificatie. Met een tabel bedoelen we een gegevensbestand waarvan alle opgeslagen elementen (*records* of *entries*) 'direct' toegankelijk zijn. We nemen aan dat elk opgeslagen element van de vorm (K,I) is, waarbij K een sleutel is die kenmerkend is voor het element en I de daarbij behorende opgeslagen informatie. Een typische vraag is nu om, voor een gegeven sleutel K, de bij K opgeslagen informatie I op te zoeken, dan wel aan te passen. Voorbeeld van een tabel is bijvoorbeeld het telefoonboek (met persoonsnamen als sleutels en adressen en telefoonnummers als informatie). Zulke grote verzamelingen gegevens worden bestudeerd bij het college Bestandsorganisatie. Tabellen (soms kleiner en met een kortere levensduur) zijn als datastructuur ook breed toepasbaar in programma's.

Samenvattend, een tabel is opgebouwd uit elementen, en elementen hebben twee componenten, een sleutel en een bijbehorende waarde (de informatie). We willen graag elementen aan de tabel kunnen toevoegen en eruit kunnen verwijderen, en we willen bij een sleutel het bijbehorende element kunnen vinden (als dat bestaat). We krijgen dan zoiets:

```
definition module SymTab;          (* boek Kingston *)

    type Entry;

    var NilEntry: Entry;
    procedure New(key: KeyType; value: ValueType): Entry;
    procedure KeyOf(x: Entry): KeyType;
    procedure ValueOf(x: Entry): ValueType;
    procedure Update(x: Entry; value: ValueType);

    type SymbolTable;

    procedure Initialize(var S: SymbolTable);
    procedure Insert(x: Entry; var S: SymbolTable);
    procedure Delete(x: Entry; var S: SymbolTable);
    procedure Retrieve (key: KeyType; var S: SymbolTable): Entry;

end SymTab.
```

Merk op dat bij deze keuze het kennelijk niet mogelijk is om de waarde van een element in de tabel rechtstreeks te veranderen, de procedure Update werkt immers op losse elementen in de tabel. Soms is er een natuurlijke ordening van de sleutels (alfabetisch, van klein naar groot). Het kan dan handig zijn de tabelelementen in deze volgorde af te lopen. Hiervoor zijn dan extra procedures nodig.

```
procedure RetrieveMin(var S: SymbolTable): Entry;
procedure RetrieveNext(x: Entry; var S: SymbolTable): Entry;
```

Representatie. Er zijn vele manieren om tabellen te representeren. Allereerst zijn er de implementaties als (lineaire) lijst - al dan niet geordend, met pointers of in een array. Dan zijn er nog de op boom-structuren gebaseerde representaties. Een aantal zijn bij Algoritmiek aan de orde geweest. We noemen de (ongebalanceerde) binaire zoekbomen, gebalanceerde boomstructuren zoals AVL bomen en B-bomen (in vele varianten). Dan zijn er ook bomen die niet strikt gebalanceerd worden gehouden, maar waar elke zoekactie de boom iets beter balanceert. Een voorbeeld hiervan zijn de *splay-trees*, deze maken gebruik van rotaties (net als AVL-bomen). In dit hoofdstuk concentreren we ons vooral op lineaire representaties van tabellen en dan nog vooral op zgn. hash-tabellen.

Het file-type van Pascal is niet zo erg geschikt om een tabel te modelleren omdat het alleen sequentieel, dwz. van voor naar achter, toegankelijk is. We gebruiken daarom hoofdzakelijk het array als representatie. Hierbij maken we voorlopig de volgende afspraken. Om de discussie te vereenvoudigen leggen we de sleutels vast op het type integer. Afhankelijk van de toepassing kiezen we de kleinste array-index gelijk aan 0 of 1.

```
TYPE Tabel = ARRAY[Eerste..Laatste] OF
               RECORD
                    Sleutel : Integer;
                    Info : NaarSmaak
               END;
```

1. Ongeordende Tabellen

Bij een ongeordende tabel, dwz. de sleutels staan op willekeurige plaatsen in de tabel, is de meest eenvoudige zoekmethode - het bekijken van alle elementen - ook de enige goede.

Algoritme : zoeken in een ongeordende tabel

```
                    { Zoeken in een tabel T[1..N] naar sleutel K. }
Bezig := true;
Plek := 1;
while Bezig and Plek ≤ N
do   if T[Plek].Sleutel = K
     then Bezig := false
     else Plek := Plek+1
     fi
od;
```

Dit algoritme kan nog versneld worden. Allereerst kunnen we beginnend bij T[N] achterstevoren zoeken. Dit heeft als voordeel dat de test Plek ≤ N vervangen kan worden door de test Plek ≠ 0. Deze laatste test omvat slechts één variabele en zal daarom op de meeste computers sneller uitgevoerd worden. Er is nog een andere mogelijkheid. De test op Plek ≤ N wordt overbodig wanneer we zeker weten dat de sleutel in de tabel aangetroffen kan worden. Hier kunnen we voor zorgen door een extra tabelelement T[N+1] te creëren en hier K in te plaatsen.

Verbeterd algoritme : zoeken in een ongeordende tabel

```
                    { Zoeken in een tabel T[1..N] naar sleutel K, }
                    { beschikbaar is ook het element T[N+1].       }
T[N+1].Sleutel := K;
Plek := 1;
while T[Plek].Sleutel ≠ K
do   Plek := Plek+1
od;                 { Wanneer Plek = N+1 dan is K niet aanwezig.    }
```

Verwachte zoektijd. Het gemiddelde aantal zoekstappen is afhankelijk van de frequenties waarmee de sleutels gezocht worden. Zoeken we T[i] met de frequentie p_i, dan is het gemiddelde aantal zoekstappen gelijk aan $(p_1+2\cdot p_2+\ldots+N\cdot p_N)$. Stellen we alle frequenties op 1/N dan is deze waarde gelijk aan $(1+2+\ldots+N)/N = (N+1)$.

Het loont in ieder geval om de sleutels die het meest gezocht worden vooraan in de tabel te plaatsen. Omdat in de praktijk niet altijd van te voren bekend is welke sleutels het meest populair zijn maakt men wel gebruik van zelf-organiserende zoekmethoden. Hierbij wordt een sleutel, nadat hij gevonden is, bijvoorbeeld verwisseld met zijn voorganger. Op de lange duur heeft dit het effect dat de meest gezochte sleutels vooraan in de tabel te vinden zijn.

2. Geordende Tabellen

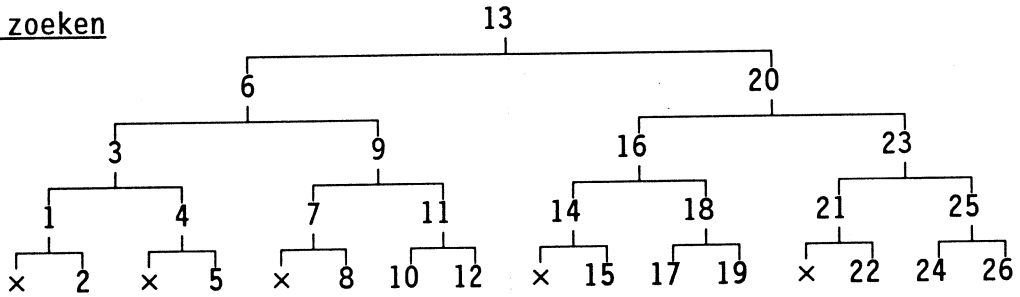
We zullen nu gaan kijken naar tabellen waarin de elementen volgens oplopende sleutel zijn opgeslagen. Dit stelt ons in staat om een stuk sneller te zoeken. Allereerst wordt het middelste element met de gezochte sleutel vergeleken. Is dit element de gezochte sleutel dan zijn we klaar. Anders hoeven we nog maar in de helft van de tabel te zoeken; in de eerste helft indien de sleutel kleiner is dan het middelste element, in de tweede helft in het andere geval. We zoeken verder door in de volgende stappen het interval waarin het getal kan liggen steeds te halveren. We krijgen zo een zoekmethode die in logaritmische tijd werkt.

Algoritme : binair zoeken in een geordende tabel

```
                { Gezocht wordt naar sleutel K in een tabel T[1..N]. }
Links := 0; Rechts := N+1;
Bezig := true; Gevonden := false;
Plek := Rechts div 2;
while Bezig
do   if K = T[Plek].Sleutel
     then Gevonden := true; Bezig := false          { Gevonden ! }
     else if K < T[Plek].Sleutel
          then Rechts := Plek
          else Links := Plek
          fi;
          Plek := (Links+Rechts) div 2;
          if Plek = Links then Bezig := false fi
     fi
od;
```

Een fraaie manier om in te zien hoe het algoritme werkt is om een binaire boom te tekenen die aangeeft in welke volgorde het algoritme de gezochte sleutel vergelijkt met elementen in de tabel. Voor een tabelgrootte N = 26 ziet deze boom er als volgt uit. Het algoritme begint bij tabelelement 13 en vervolgt dan bij 6 als K < T[13].Sleutel en vervolgt bij 20 als K > T[13].Sleutel.

Binair zoeken

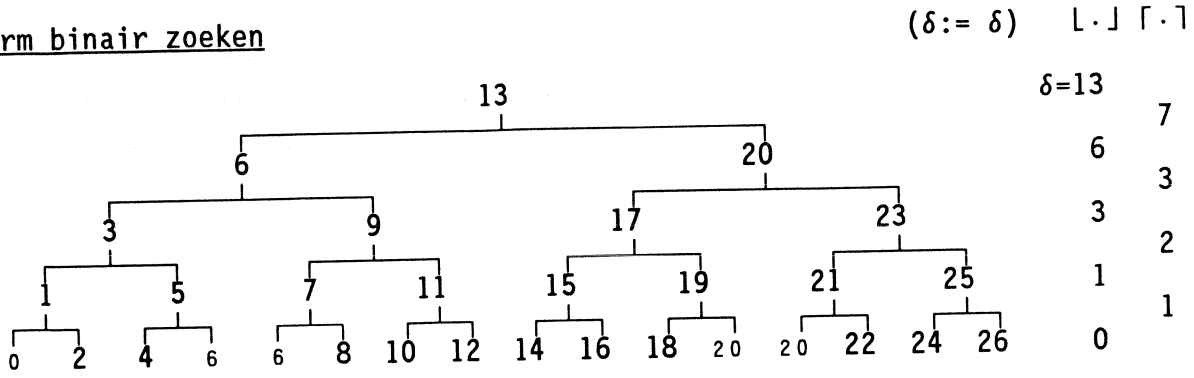


Uniform binair zoeken. In bovenstaand algoritme houden we de linker- en rechtergrens van het zoekinterval bij. We kunnen ook de verschuiving δ van de grenzen bijhouden en na iedere vergelijking δ (ongeveer) halveren. Deze

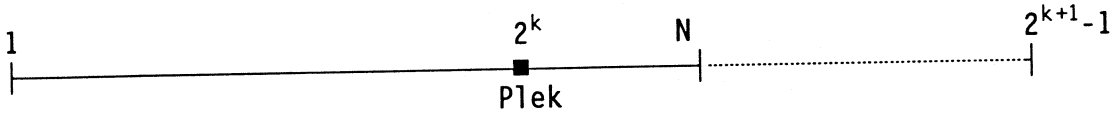
/ariant wordt wel *uniform binair zoeken* genoemd. In de boom die de /ergelijkingen aangeeft is de reden hiervan te zien; alle waarden op één nivo /erschillen steeds dezelfde constante van hun ouders. Merk op dat op het laatste nivo herhalingen voorkomen. Het algoritme maakt dan overbodige /ergelijkingen.

Het algoritme wordt hier niet expliciet uitgeschreven. Probeer dit zelf.

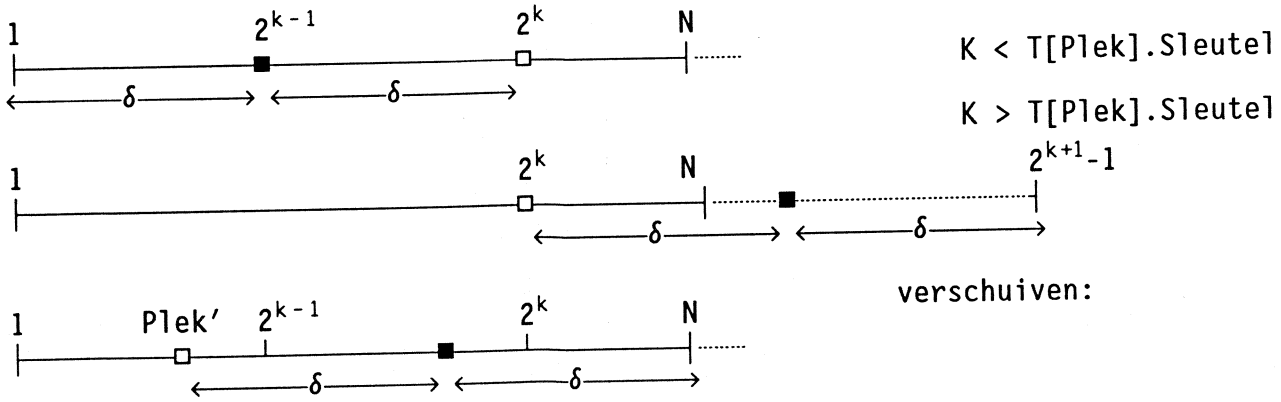
Uniform binair zoeken



Bijna uniform binair zoeken. Er is een andere mogelijkheid om te komen tot (bijna) uniform zoeken. Het gewone algoritme is uniform bij een tabelgrootte 2^n-1. Met een truukje kan er voor gezorgd worden dat de tabelgrootte na de eerste stap van deze vorm is. Eerst bepalen we een tabelgrootte die wél van die vorm is, en tenminste zo groot als de gegeven tabelgrootte N. We doen net of we in die tabel zoeken, en testen het element halverwege. In de onderstaande figuur geldt Plek = $2^k \le N < 2^{k+1}$.

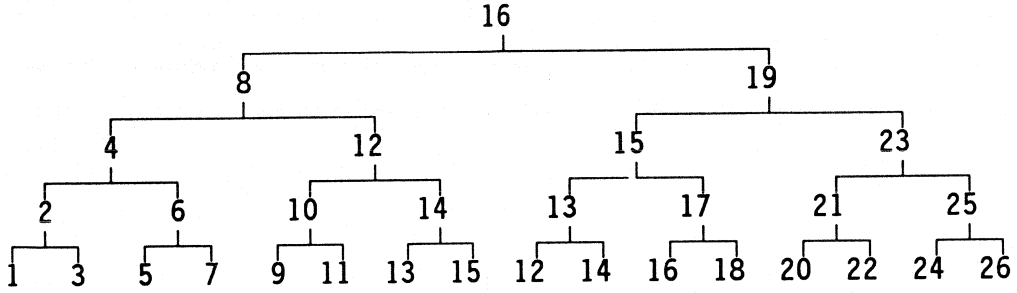


Blijkt dat de gezochte sleutel links moet liggen dan zoeken we volgens de uniforme methode verder. Moet de gezochte sleutel rechts liggen dan verschuiven we het 'midden' van de denkbeeldige grotere tabel naar links (Plek' = N+1-Plek) en zoeken we verder in het nieuwe rechter segment.



Omdat de rechter helft de linker overlapt zijn er sleutels (nl. die tussen Plek' en Plek) waar gezocht wordt maar waar de gezochte sleutel K niet kan liggen; we weten immers al dat K > T[Plek].Sleutel. Dit leidt tot een 'vergelijkingsboom' waarvan in de rechter subboom sommige takken niet gekozen kunnen worden. Vooral wanneer N een macht van twee is levert dit een asymmetrische situatie op.

Bijna uniform binair zoeken



Fibonacci zoekmethode. Alle bovenstaande berekeningen maken gebruik van de operaties vermenigvuldigen en delen. Op sommige computers schijnen dit tijdrovende operaties te zijn. (Men zou denken dat vermenigvuldigen met en delen door een factor 2 simpel te implementeren is als het verschuiven van een bitstring, maar dit terzijde.)

De Fibonacci zoekmethode maakt gebruik van een verdeling van de tabel in ongelijke delen volgens opeenvolgende Fibonacci getallen. De Fibonacci getallen hebben een lange geschiedenis; we vinden ze bij diverse telproblemen terug (zie bijvoorbeeld de colleges Algoritmiek en Discrete Wiskunde). De getallen worden berekend door te beginnen met twee enen en telkens de laatste twee getallen bij elkaar op te tellen. Wanneer we ook een 'nulde' Fibonacci getal toelaten krijgen we de vergelijkingen:

$F_0 = 0,\ F_1 = 1,\ F_i = F_{i-1} + F_{i-2}\ (i \geq 2).$

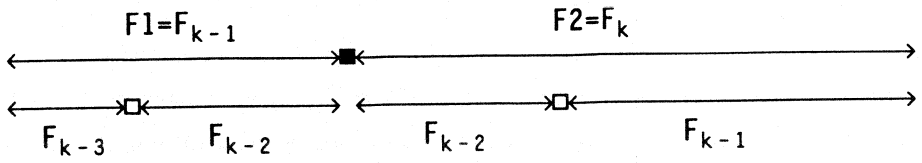
De rij begint dus met 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

De eigenschap van de Fibonacci getallen die hier belangrijk is, is dat telkens het voorgaande element van de rij berekend kan worden uit de laatste twee elementen. Er geldt immers dat $F_{i-2} = F_i - F_{i-1}$. Ga na dat zoiets niet lukt met twee-machten (waarbij alleen optellen en aftrekken gebruikt mag worden).

Het algoritme houdt twee waarden F1 en F2 bij; dit zijn steeds twee opeenvolgende Fibonacci getallen die de verdeling aangeven van het interval waarin we nog moeten zoeken. Vóór de waarde die we gaan vergelijken (Plek) bevinden zich nog F1-1 elementen, erna nog F2-1. Behalve deze asymmetrische verdeling gaat het zoekproces precies als bij (bijna) binair zoeken. Er wordt steeds bepaald of we in het linker dan wel het rechter interval verder moeten. Zoeken we rechts verder dan moeten de waarden F1 en F2 elk één stap in de

Fibonacci rij terug gezet worden. Zoeken we daarentegen links verder dan moeten we ze twee stappen opschuiven.

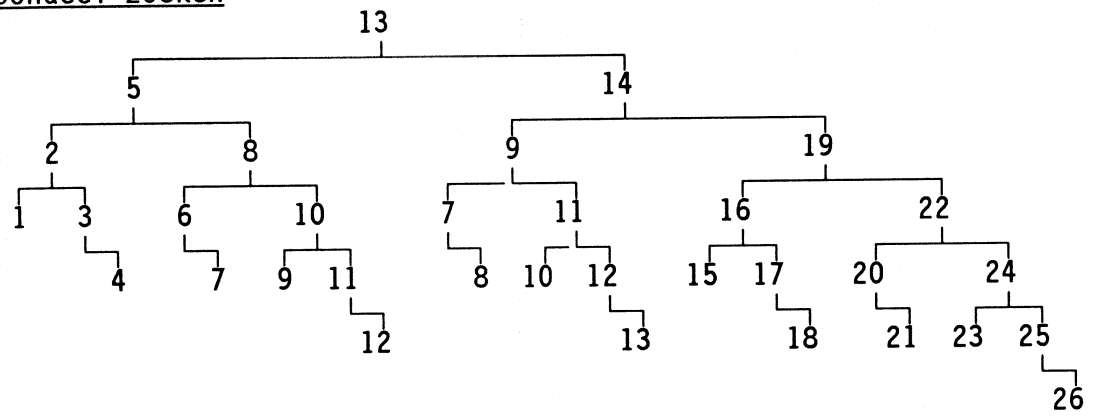
In een plaatje ziet het er als volgt uit. Hierbij stellen de getallen het verschil tussen de buitengrenzen van de intervallen aan. Het aantal elementen in de intervallen is telkens één minder.



Om het algoritme geschikt te maken voor tabellen van willekeurige grootte wordt weer net gedaan of we een grotere tabel beschouwen, en schuiven we na de eerste vergelijking eventueel het 'midden' op.

Ook bij dit algoritme kan een boom van vergelijkingen getekend worden. Het is niet zo verwonderlijk dat dit een Fibonacciboom is.

Fibonacci zoeken



3. HASH TABELLEN

We beginnen met het uit de weg ruimen van een misverstand. Het engelse werkwoord *to hash* betekent zo iets als fijnhakken, het zelfstandig naamwoord *hash* kan vertaald worden met hachee.

Wanneer we gegevens in een tabel op moeten slaan zouden we het liefst aan de hand van de sleutel direct het adres berekenen. Dit beperkt het aantal zoekstappen immers tot één. Het is verleidelijk om dan voor elke mogelijke sleutelwaarde een plaats in de tabel te reserveren. Bij het opslaan van slechts een klein percentage van alle mogelijke sleutels betekent dit een enorme ruimteverspilling. Hashen biedt hiervoor een redelijke tussenoplossing, dat wil zeggen een klein aantal zoekstappen en niet al te veel verspilling van ruimte.

We gaan uit van een beschikbare tabel met M elementen genummerd van 0 tot M-1. Voor elke sleutel wordt een plek in de tabel berekend met een adresfunctie (*hash function*) h. Het adres van de sleutel K is dan h(K). Er zullen in het algemeen verschillende sleutels zijn die hetzelfde adres toebedeeld krijgen; we noemen zulke sleutels synoniemen. Indien synoniemen in dezelfde tabel geplaatst moeten worden ontstaat een botsing (*collision*). Een hashmethode bestaat (*1*) uit het kiezen van een geschikte adresfunctie die zoveel mogelijk synoniemen probeert te vermijden en (*2*) uit het bepalen van een strategie die in werking treedt bij botsingen.

3.1. PERFECT HASHEN

Als de sleutels die in een hash-tabel opgezocht moeten worden van te voren vast liggen dan is met genoeg geduld vaak wel een adresfunctie te vinden waarbij er geen synoniemen zijn (tussen de gegeven sleutels). Zo'n functie heet een perfecte adresfunctie. Omdat er bij zo'n functie geen synoniemen zijn kan elk opgeslagen element in één stap gevonden worden. Merk op dat we hierbij uitgaan van een statisch gebruik van de hash-tabel, tijdens het gebruik worden er geen elementen toegevoegd of verwijderd.

Voorbeeld. Bij het analyseren van een engelstalige tekst spelen de meest voorkomende woorden een bijzondere rol. Elk in de tekst aangetroffen woord wordt opgezocht in de lijst van meest voorkomende woorden. Om het zoeken te versnellen kan deze lijst bijvoorbeeld opgeslagen worden als binaire zoekboom of als trie (zie college Algoritmiek). Een andere mogelijkheid is om de woorden in een hash-tabel op te slaan. Als perfecte adresfunctie kan dan in dit geval een functie genomen worden van de vorm

adres = lengte + waarde eerste letter + waarde laatste letter, waarbij de waarde van de letters opgezocht kan worden in een tabel (met indices 'A' tot en met 'Z').

Met enig rekenwerk kunnen de volgende waarden gevonden worden:

A : 3 B : 15 D : 7 F : 15 H : 10 M : 12 N : 13
O : 7 R : 12 S : 6 U : 15 W : 14 Y : 9
(overige letters : 0)

Daarmee is het adres van 'have' bijvoorbeeld 4 + 10 + 0 = 14, en dat van 'a' 1 + 3 + 3 = 7. De meest voorkomende woorden zijn, in de volgorde van hun adressen:
I, it, the, that, at, are, a, is, to, this, as, he, and, have, in, not, be, but, his, had, or, on, was, of, her, by, you, with, which, for, from. ■

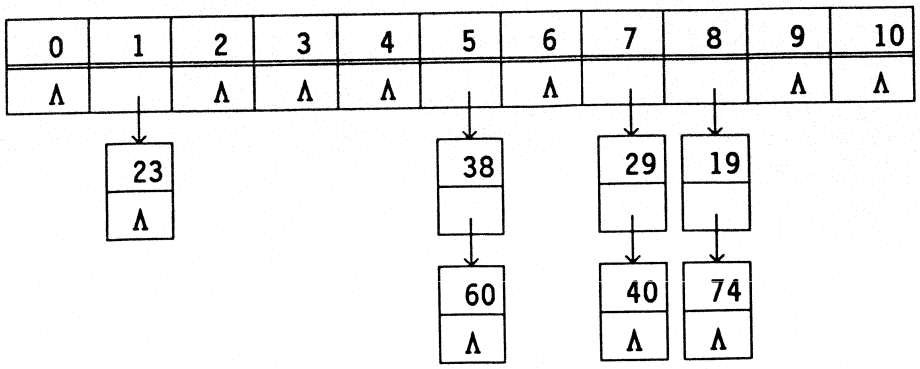
3.2. Hashen met Verbonden Lijsten

Botsingen kunnen opgelost worden door te werken met lijsten. Bij de meest directe methode is elk tabelelement het begin van een verbonden lijst die de synoniemen bevat die op dat adres geplaatst worden. Het zoeken naar elementen gaat sneller wanneer we de lijsten in oplopende volgorde houden. Het is dan wel niet mogelijk om binair zoeken toe te passen (de elementen zijn niet rechtstreeks toegankelijk) maar we kunnen ophouden indien we een element aantreffen dat groter is dan het gezochte.

Geschikte declaraties zijn nu bijvoorbeeld:

```
TYPE EltWijzer = ↑Element;
     Element = RECORD
                    Sleutel : Integer;
                    Volger : EltWijzer
               END;
     Tabel = ARRAY[0..M-1] OF EltWijzer;
```

Voorbeeld. Beschouw de tabel T[0..10] en de adres-functie h(K) = K *mod* 11. Na het plaatsen van de elementen 60, 29, 74, 38, 19, 23 en 40 ziet de tabel er als volgt uit (wanneer we de lijsten in oplopende volgorde ordenen).



■

Een andere manier om met lijsten te werken is door in de tabel zèlf lijsten aan te leggen. We werken dan niet expliciet met pointers, maar geven de adressen van de opvolgers. Telkens als een tabelelement reeds bezet blijkt te zijn kiezen we, volgens een bepaalde strategie, een lege plaats in de tabel. Die plaats wordt in de lijst gehangen en krijgt de toe te voegen sleutel als inhoud.

Nadeel van deze methode is dat de lijsten door elkaar heen kunnen gaan lopen. Dit vergroot in het algemeen de gemiddelde zoektijd.

Declaraties kunnen er nu als volgt uit zien. Er wordt in het algemeen een speciale waarde gekozen om een leeg element aan te geven (bv. Sleutel = 0, of MAXINT) en ook een speciale waarde die het einde van de lijsten aangeeft (bv. Volger = -1, of M).

```
TYPE Element = RECORD
                    Sleutel : Integer;
                    Volger : Integer;
               END;
Tabel = ARRAY[0..M-1] OF Element;
```

Voorbeeld. Gegeven is de Tabel T[0..10] (van bovenstaand type) en een adres-functie h(K) = K *mod* 11. We spreken af dat lege plaatsen de constante *Leeg* als sleutel toebedeeld krijgen, waarbij bv. *Leeg* = 0. Het einde van de lijsten geven we aan met Volger gelijk aan -1. Na het plaatsen van 60, 29 en 74 ziet de tabel er als volgt uit:

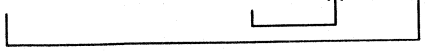
0	1	2	3	4	5	6	7	8	9	10
					60		29	74		
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

(*Leeg* weggelaten)

Bij het plaatsen van 38 blijkt adres 5 reeds bezet te zijn. We kiezen er hier voor om het hoogste ongebruikte adres als uitwijkplaats te nemen. 38 komt dus op adres 10, bij adres 5 komt een verwijzing naar 10 te staan. Wanneer we

nu 19 in de tabel plaatsen komt deze sleutel op 9 terecht met een verwijzing bij het oorspronkelijke (reeds bezette) adres.

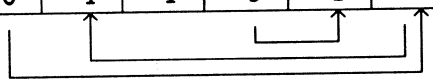
0	1	2	3	4	5	6	7	8	9	10
					60		29	74	19	38
-1	-1	-1	-1	-1	10	-1	-1	9	-1	-1



Sleutels met adres 10 komen in dezelfde lijst te hangen als de sleutels die op adres 5 terecht horen. Dit betekent dat we bij het zoeken naar deze sleutels beide soorten tegenkomen, terwijl bij de vorige hash-methode de lijsten gescheiden bleven.

Sleutel 32, met adres 10, komt terecht op adres 6. Bij het zoeken naar 16 (een element dat niet in de tabel voorkomt) moeten we drie adressen bezoeken, hoewel slechts twee synoniemen van 16 geplaatst zijn. We zoeken namelijk de lijst beginnend bij adres 5 af. We vinden zo de adressen 5, 10 en 6. Evenzo duurt het twee stappen voordat we 32 in de tabel aangetroffen hebben omdat we zowel adres 10 als adres 6 moeten proberen.

0	1	2	3	4	5	6	7	8	9	10
					60	32	29	74	19	38
-1	-1	-1	-1	-1	10	-1	-1	9	-1	6



■

3.3. Hashen met Open Adressering

Wanneer een sleutel niet op zijn eigen plek in de tabel terecht kan omdat daar al een synoniem staat worden andere adressen in de tabel geprobeerd. De volgorde waarin de andere adressen in de tabel bezocht worden kan afhangen van de sleutel. We zullen verschillende manieren bespreken waarop deze volgorde gekozen kan worden. Laten we aannemen dat, afhankelijk van de sleutel K, zo achtereenvolgens de adressen h(K,0) = h(K), h(K,1),..., h(K,M-1) geprobeerd worden totdat een lege plek aangetroffen wordt. Omdat het nutteloos is om een adres twee maal te proberen en omdat elk adres eventueel bekeken dient te kunnen worden, eisen we dat de reeks h(K,0), h(K,1),..., h(K,M-1) een permutatie vormt van de adressen 0,..., M-1. (M is de tabelgrootte.)

Het *zoeken* naar een sleutel volgt dezelfde adressen als het plaatsen van een sleutel. Eerst wordt dus gekeken of de sleutel zich op adres h(K) bevindt. Is de sleutel daar niet aanwezig dan worden de adressen h(K,1),..., h(K,M-1) nagegaan. Als tenminste één plaats in de tabel leeg is, mogen we het zoeken

staken wanneer we een lege plaats aantreffen. Immers, wanneer de sleutel in de tabel had gestaan, dan was hij geplaatst op het eerste lege adres in de reeks h(K,0), h(K,1),..., h(K,M-1).

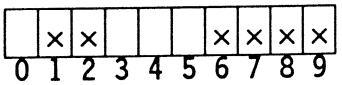
Hierbij zijn we er stilzwijgend van uitgegaan dat eenmaal geplaatste sleutels niet verwijderd worden. Het verwijderen van sleutels uit hash-tabellen kan niet zonder speciale voorzorgsmaatregelen gebeuren. Er wordt nog op ingegaan in een latere paragraaf.

We bespreken een aantal meer concrete methoden.

1. Lineair hashen. Wanneer een plek in de tabel bezet is wordt telkens het voorgaande adres in de tabel bekeken totdat een lege plek ontdekt wordt. In bovenstaande notatie geldt dus dat h(K,i) = h(K)-i *mod* M.

Voorbeeld. Beschouw de tabel T[0..10] en de adres-functie h(K) = K *mod* 11. De sleutels 60, 29 en 74 worden geplaatst op de respectievelijke adressen 5, 7 en 8. Indien vervolgens 38 in de tabel geplaatst moet worden blijkt dat het berekende adres h(38) = 5 reeds bezet is. Daarom plaatsen we 38 op adres 4, de eerst voorgaande onbezette plek. Willen we nu 19 plaatsen dan moet dit gebeuren op adres 6. Het oorspronkelijk berekende adres h(19) = 8, evenals het daaraan voorafgaande adres 7, is reeds bezet. Sleutel 23 kan gewoon geplaatst worden op het vrije adres 1. ■

Een belangrijk nadeel van lineair hashen ontstaat door een effect dat primair clusteren wordt genoemd. Dit houdt in dat de sleutels in het algemeen niet verspreid door de tabel liggen, maar in blokken van opeenvolgende adressen (*clusters*) te vinden zullen zijn. We bekijken een denkbeeldige tabel:



× = bezet

Wanneer we een willekeurige sleutel in deze tabel plaatsen zal deze in 50% van de gevallen terecht komen op adres 5. Immers sleutels die op een van de adressen 5,6,...,9 thuis horen, komen alle terecht op adres 5. Op dezelfde manier komt 30% van willekeurig gekozen sleutels terecht op adres 0, terwijl slechts 10% op adressen 3 of 4 geplaatst wordt.

Op deze manier zullen er bij lineair hashen clusters ontstaan, waarvan de grotere clusters bovendien een grotere groeisnelheid hebben dan de kleinere. Deze clustervorming heeft een negatieve invloed op de zoeksnelheid.

2. Pseudo random hashen, kwadratisch hashen. Om de effecten van primair clusteren te vermijden dienen we te zorgen dat niet elke sleutel steeds op de direct voorgaande adressen geprobeerd wordt. Een manier die in de literatuur is voorgesteld heet pseudo random hashen. Er wordt daartoe een 'random'

permutatie $r_0=0, r_1, r_2, \ldots, r_{M-1}$ van de adressen gegenereerd. Het zoeken in de tabel gebeurt dan op de adressen $h(K,i) = h(K)-r_i$. Natuurlijk kan deze rij random getallen niet steeds opnieuw bepaald worden. We moeten immers bij het zoeken naar een sleutel hetzelfde zoekpad volgen als waarmee die sleutel geplaatst is.

Een andere methode is kwadradisch hashen. Hierbij gebruikt men de getallen i^2 modulo de tabelgrootte, zogenaamde kwadraatresten. De adressen die we proberen zijn achtereenvolgens $h(K)$, $h(K)\pm1$, $h(K)\pm4$, $h(K)\pm9, \ldots$,dat wil zeggen de adressen $h(K)$, $h(K)\pm i^2$ (modulo de tabelgrootte M), $1 \leq i \leq (M-1)$. Zoals we eerder geëist hebben dienen deze adressen een permutatie te vormen van de tabeladressen. Nadere wiskundige beschouwingen leren ons dat dit het geval is wanneer M een priemgetal is, dat bovendien bij deling door vier een rest 3 heeft. De getallen 127, 251, 503 en 1019 voldoen hier bijvoorbeeld aan.

Ook bij deze beide methodes wordt het aantal zoekstappen negatief beïnvloed door een vorm van clustering, secundaire clustering: synoniemen volgen hierbij steeds hetzelfde zoekpad.

3. Dubbele hashfunctie. We proberen secundaire clustering zoveel mogelijk te vermijden. Hiervoor wordt een tweede hash-functie ingevoerd, de zogenaamde stap-functie (*probe function*) p. Bij het zoeken en plaatsen van een sleutel K stappen we nu p(K) adressen naar voren. Dit aantal is dus afhankelijk van de sleutel en wordt berekend met behulp van de tweede hash-functie. We willen daarbij graag dat p(K) *niet* direct uit h(K) berekend kan worden, om secundaire clustering te vermijden. Zoveel mogelijk dienen synoniemen onafhankelijke zoekpaden te volgen, dat wil zeggen dat de sleutels die op hetzelfde adres terechtkomen met grote waarschijnlijkheid in de volgende zoekstap niet weer op een zelfde adres geprobeerd worden. Zijn de twee hash-functies onafhankelijk, dan spreken we van dubbel hashen.

Bij het bepalen van de tweede hash-functie dienen we er voor te zorgen dat de bezochte adressen $h(K,i) = h(K)-i\cdot p(K)$ (modulo de tabelgrootte M), $0 \leq i \leq M-1$ allemaal verschillend zijn. Dit is het geval als (en slechts als) p(K) en M geen factoren gemeenschappelijk hebben: $(p(K),M) = 1$; in dat geval heet de stap-functie toelaatbaar. We dienen dus voorzichtig te zijn bij het opstellen van een tweede hash-functie.

Voorbeeld. We starten met de reeds bekende tabel T[0..10] en adres-functie $k(K) = K \ mod \ 11$. We gebruiken een stap-functie $p(K) = (K \ mod \ 4)+1$. Daar 11 een priemgetal is zijn de waarden die p levert steeds relatief priem met de tabelgrootte. De stap-functie voldoet daarmee aan de gestelde eis. Verder kan p(K) niet uit het adres h(K) van de sleutel berekend worden; de synoniemen 0, 11, 22 en 33 hebben bijvoorbeeld de

respectievelijke stapgroottes 1, 4, 3 en 2. Er is daarom sprake van hashen met een dubbele hash-functie *zonder* secundaire clustering.

De elementen 60, 29 en 74 worden als voorheen (bij lineair hashen) in de tabel geplaatst, hun adressen 5, 7 en 8 zijn immers vrij. Bij het plaatsen van 38 blijkt adres 5 bezet te zijn. Omdat p(38) = 3 proberen we nu adres 5-3 = 2. Dit adres is vrij, zodat 38 hier opgeslagen kan worden.

De sleutel 19 is een synoniem van het reeds geplaatste getal 74. We proberen 19 in tweede instantie te plaatsen op het adres h(19)-p(19) = 8-4 = 4. Dit lukt omdat dit adres nog vrij is.

Sleutel 23 kan terecht op zijn adres 1. Tenslotte plaatsen we 40 (adres 7 en stapgrootte 1) in de tabel. Dit lukt bij de tweede poging. Het tabelelement 7 is bezet, 7-1 = 6 is vrij. ■

De volgende algoritmes kunnen gebruikt worden bij dubbele hash-functies, maar ook bij lineair hashen - neem dan p(K) = 1.

Algoritmes : zoeken en toevoegen in een hash-tabel

```
{ Zoeken en toevoegen van sleutel K in tabel T[0..M-1],      }
{ mbv. hash-functies h en p.                                 }
{ Er zijn N<M plaatsen bezet; Sleutel = Leeg op lege plek.   }
```

Zoeken :

```
Adres := h(K); Bezig := true; Gevonden := false;
while Bezig
do   if T[Adres].Sleutel = K
     then Bezig := false;                                  { Gevonden }
          Gevonden := true
     else if T[Adres].Sleutel = Leeg
          then Bezig := false                              {*}
          else Adres := Adres - p(K);
          if Adres < 0 then Adres := Adres + M fi
     fi
od;
```

Toevoegen : (ipv. regel {*} in bovenstaand algoritme)

```
          then if N = M - 1
               then overflow
               else T[Adres].Sleutel := K;
                    N := N + 1
               fi;
               Bezig := false;
```

3.4. Open Adressering (vervolg): De geordende hashtabel

Veronderstel dat de sleutels aan de tabel in aflopende volgorde worden toegevoegd. Dit betekent dat bij het plaatsen van de sleutel K alleen sleutels groter dan K in de tabel staan. Dus, wanneer K na i stappen op plaats h(K,i) geplaatst wordt, dan bevatten de adressen h(K,0), h(K,1),..., h(K,i-1) sleutels die groter zijn dan K. We noemen dit een geordende hash-tabel.

Voorbeeld. De sleutels 60, 29, 74, 38, 19, 23 en 40 worden weer in een hash-tabel T[0..10] geplaatst met hash-functies h(K) = K *mod* 11 en p(K) = (K *mod* 4) + 1, ditmaal echter op volgorde van grootte.

We plaatsen 74, 60 en 40 op hun respectievelijke adressen 8, 5 en 7. Omdat h(38) = 5 reeds bezet is wordt 38 geplaatst op 5-p(38) = 5-3 = 2. Bij het plaatsen van 29 zijn drie stappen nodig: tabelelementen h(29) = 7, en 7-p(29) = 7-2 = 5 zijn al gevuld. De sleutel 29 kan echter nog terecht op 5-2 = 3. De laatste sleutel 19 komt niet op het als eerst geprobeerde adres 8, maar op adres 4 terecht.

Let op dat de sleutels niet op alle zoekpaden in volgorde liggen! Zo komen we, op zoek naar 29, eerst 40, dan 60 en tenslotte 29 tegen. Wél geldt dat alle sleutels die we tegenkomen (40 en 60) vóór de sleutel die we zoeken (hier 29) groter dan deze sleutel zijn. ■

Bij het zoeken naar een sleutel K in de tabel kunnen we hier ook gebruik van maken. We hoeven slechts door te zoeken zolang we sleutels tegenkomen die groter dan K zijn. Treffen we een sleutel kleiner dan K aan dan kunnen we het zoeken staken. K staat niet in de tabel (ook niet verderop langs het zoekpad).

Dit levert het volgende algoritme.

Algoritme : zoeken in geordende hash-tabel

```
                          { Aanname: tabel bevat tenminste één lege plaats }
Adres := h(K);
while K < T[Adres].Sleutel
do   Adres := Adres - p(K);
     if Adres < 0 then Adres := Adres + M fi
od;
Gevonden := (K = T[Adres].Sleutel);
```

We kunnen ons afvragen of we in de praktijk veel aan bovenstaand algoritme zullen hebben. Het zal in het algemeen weinig voorkomen dat we de sleutels in volgorde aangeleverd krijgen, behalve in de gevallen waar ze van te voren vaststaan. Gelukkig blijkt het mogelijk om, ook wanneer de sleutels

door elkaar aangeleverd worden, de tabel te vullen zodat bovenstaand algoritme werkt. Dit gaat wel ten koste van wat extra werk. We moeten namelijk de stap-grootte opnieuw berekenen voor een aantal sleutels in de tabel die verplaatst moeten worden. Meestal is dit de moeite waard in die gevallen waar we relatief veel zoeken naar elementen die niet in de tabel voorkomen (zonder dat we ze toe voegen).

Algoritme : sleutel toevoegen aan geordende hash-tabel

```
                                { Sleutel K in tabel T[0..M-1] plaatsen. }
                                { Tenminste twee plaatsen vrij.          }
Adres := h(K);
while T[Adres].Sleutel ≠ Leeg
do   if T[Adres].Sleutel < K
     then verwissel K en T[Adres].Sleutel      { Plaats K, ga verder   }
     fi;                                       { met T[Adres].Sleutel. }
     Adres := Adres-p(K);
     if Adres < 0 then Adres := Adres+M fi
od;
T[Adres].Sleutel := K;
```

Voorbeeld (vervolg). In de zojuist ontstane tabel plaatsen we de sleutel 85. Deze sleutel is groter dan de sleutel 74 die aanwezig is op adres h(85) = 8. We zetten 85 op de plaats van 74 en stappen verder met 74 en stapgrootte p(74) = 3. Op adres 8-3 = 5 treffen we 60 aan, die verdrongen wordt door 74. Deze sleutel op zijn beurt verdringt de sleutel 19 op adres 5-p(60) = 5-1 = 4. Deze laatste sleutel kan terecht op de vrije plek 4-p(19) = 4-4 = 0.

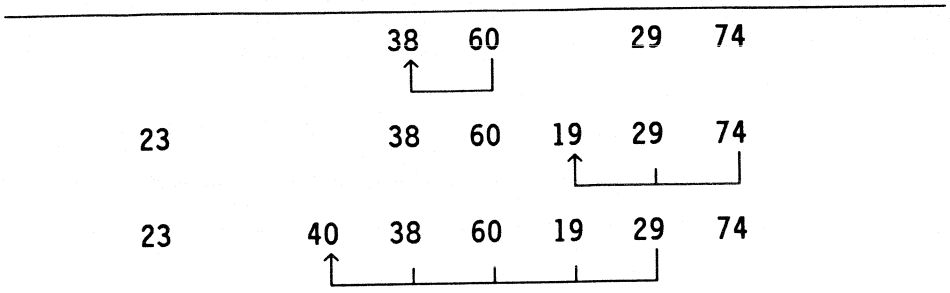
Het is niet altijd zo dat in élke stap van het plaatsingsalgoritme een sleutel door een andere verdrongen wordt. ■

Stelling. Het bovenstaande plaatsingsalgoritme levert dezelfde tabel op als wanneer de sleutels, op de gewone manier, in volgorde geplaatst worden.

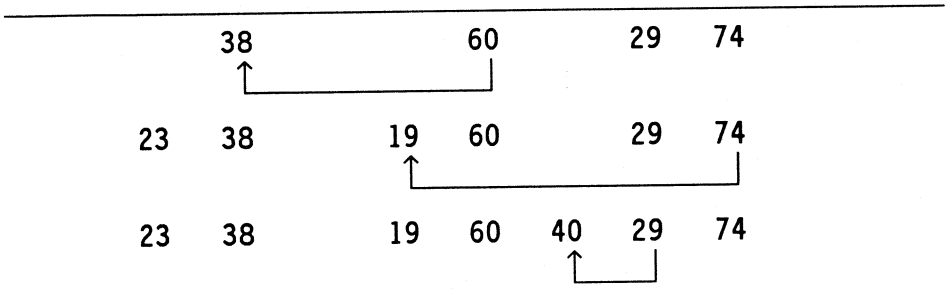
Bewijs. Neem aan dat K de grootste sleutel is die niet in beide tabellen op dezelfde plek terecht komt. De adressen waar deze sleutel geprobeerd wordt is voor beide methoden dezelfde rij h(K,0), h(K,1),..., waarbij h(K,i) = h(K)-i·p(K). Neem aan dat we volgens de geordende methode K op plek h(K,i) plaatsen en volgens de gewone methode op plek h(K,j). Bij beide methoden geldt dat de sleutels die eerder langs het zoekpad staan groter zijn dan de geplaatste sleutel. Bekijk het geval waarin j > i. De sleutel op plek h(K,i) is in de geordende tabel gelijk aan K, maar in de gewone tabel (waarin we de sleutels op volgorde plaatsen) groter dan K. Dit is in tegenspraak met

Sleutel	60	29	74	38	19	23	40	K
Adres	5	7	8	5	8	1	7	h(K) = K *mod* 11
Stapgrootte	1	2	3	3	4	4	1	p(K) = (K *mod* 4) + 1

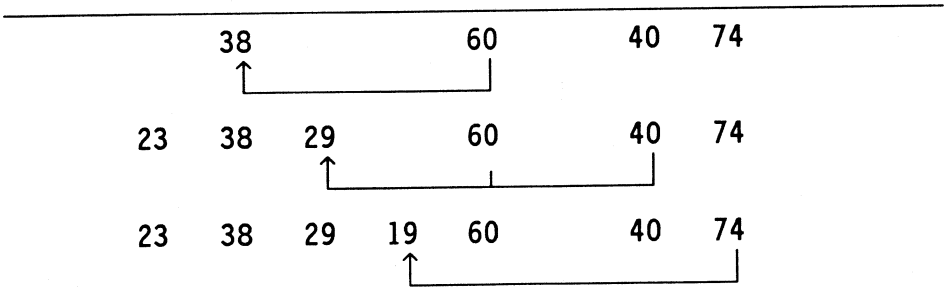
▷ lineair hashen



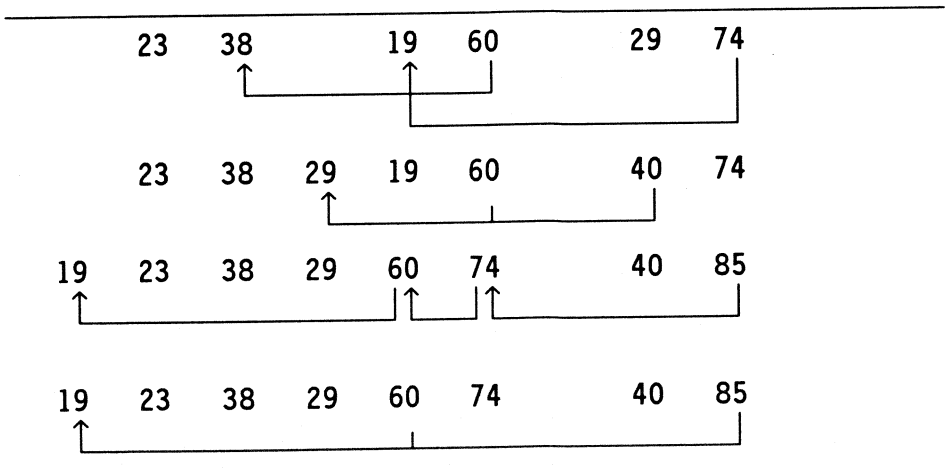
▷ dubbel hashen



▷ opvolgend hashen



▷ geordend hashen



de aanname, volgens welke die grotere sleutel in beide tabellen op dezelfde plek staat. Op dezelfde manier leidt $j < i$ tot tegenspraak. ■

Aantal bezochte adressen. We hebben nu kennis gemaakt met een aantal hash-methoden met open adressering. In het bijzonder maken we onderscheid tussen *lineair* hashen, hashen met *secundaire clustering* en tenslotte *dubbel hashen* (met onafhankelijke hash-functies). Bij deze drie methoden kunnen we bovendien kiezen of we de tabel al dan niet geordend opbouwen.

Voor elk van de methoden wordt onderzoek gedaan naar het gemiddelde aantal zoekstappen nodig bij het zoeken naar elementen in de tabel. Er wordt daarbij onderscheid gemaakt tussen het zoeken naar elementen die wél en die *niet* in de tabel voorkomen (vaak *succesvol* en *onsuccesvol* zoeken genoemd). De aantallen zoekstappen blijken (zowel bij theoretische analyse als bij praktische simulaties) niet zo zeer af te hangen van de tabelgrootte als wel van de mate waarin de tabel gevuld is.

Laat α de vullingsgraad van de tabel zijn; dat wil zeggen het aantal geplaatste sleutels gedeeld door de grootte van de tabel. In de notatie die we tot nu toe gebruikten geldt $\alpha = N/M$. Bij grote tabellen blijken de volgende formules een redelijke benadering te geven voor het te verwachten aantal zoekstappen in de diverse soorten hash-tabellen.

	succesvol zoeken	zonder succes
lineair	$(1 + \frac{1}{1-\alpha})$	$(1 + \frac{1}{(1-\alpha)^2})$
secundaire clustering	$1 + ln\frac{1}{1-\alpha} - \alpha$	$\frac{1}{1-\alpha} - ln(1-\alpha) - \alpha$
dubbel hashen	$\frac{1}{\alpha} \cdot ln\frac{1}{1-\alpha}$	$\frac{1}{1-\alpha}$

verwacht aantal zoekstappen bij *ongeordend* hashen

In al deze gevallen is er dezelfde relatie tussen de waarden voor zoeken met en zonder succes. We krijgen de waarde mét succes door de primitieve van de waarde zónder succes te nemen en deze door α te delen. Dit kan als volgt verklaard worden.

Laat (bij een vaste tabelgrootte M) S_N en Z_N de gemiddelde aantallen zoekstappen zijn bij het zoeken wanneer de tabel N sleutels bevat, respectievelijk naar een sleutel die wèl (succes) en een sleutel die niet in de tabel voorkomt (zonder succes). Bij het zoeken naar een sleutel die in de tabel staat moeten we hetzelfde aantal zoekstappen doen als bij het plaatsen van die sleutel nodig waren. Bij een tabel met N opgeslagen sleutels zijn deze sleutels geplaatst toen er nog slechts 0, 1,..., N-1 sleutels in de tabel stonden. Middelen we over alle mogelijke tabellen dan vinden we de formule $S_N = \frac{1}{N}(Z_0+Z_1+\ldots Z_{N-1})$.

Analyse van dubbel hashen. Verreweg het eenvoudigst te analyseren is dubbel hashen. We nemen aan dat met twee onafhankelijke hash-functies het plaatsen van een sleutel gebeurt alsof de adressen bepaald worden met een kansmechanisme. De resultaten die we met deze aanname verkrijgen blijken bij simulaties redelijk te kloppen met de echte waarden.

We berekenen Z_N. Stel P_r is de kans dat er precies r stappen nodig zijn om een sleutel die niet in de tabel staat te zoeken. Dan is $Z_N = \sum_{1 \le r \le N} r \cdot P_r$.

Precies r stappen nodig betekent dat wanneer we langs het zoekpad van die sleutel gaan, de eerste r-1 adressen bezet zijn en het r-de leeg. De kans hierop valt te vergelijken met het trekken zonder terugleggen uit een vaas met M ballen waarvan er N rood zijn en de overige zwart. De kans die we zoeken is gelijk aan de kans op het trekken van r-1 rode ballen gevolgd door een zwarte. Deze kans is $P_r = \binom{M-r}{N-r+1} \Big/ \binom{M}{N}$.

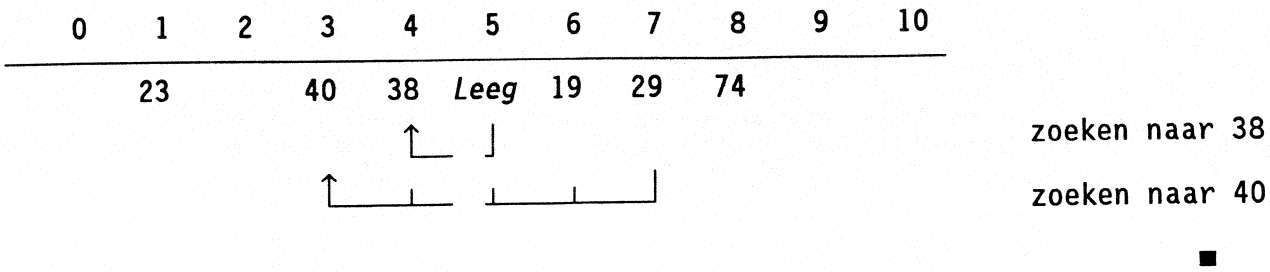
Rekenen we hiermee Z_N uit, dan blijkt daar $\frac{M+1}{M-N+1} = \frac{1+1/M}{1-\alpha+1/M} \approx \frac{1}{1-\alpha}$ uit te komen. Dus $S_N = \frac{1}{N}(Z_0+Z_1+\ldots Z_{N-1}) = \frac{M+1}{N}(\frac{1}{M+1}+\frac{1}{M}+\ldots+\frac{1}{M-N+2}) \approx \frac{M+1}{N} ln \frac{M+1}{M-N+2} \approx \frac{1}{\alpha} ln \frac{1}{1-\alpha}$.

Relatie met geordend hashen. Er bestaat een relatie tussen de verwachte aantallen zoekstappen voor geordend hashen en voor gewoon hashen waarbij de sleutels in dalende volgorde worden toegevoegd (dit noemden we eerder opvolgend plaatsen). Zonder uitleg vermelden we dat $S_N^\circ = S_N$ en $Z_N^\circ = S_{N+1}$, waarbij $^\circ$ op geordend betrekking heeft. Omdat voor grote tabellen $N/M \approx (N+1)/M$, zijn voor geordend hashen de aantallen voor succesvol (S°) en zonder succes (Z°) zoeken beide gelijk aan succesvol zoeken bij gewoon hashen.

3.5. Het verwijderen van Sleutels

In een hash-tabel met verbonden lijsten kunnen we zonder enige moeite een element verwijderen. In open hash-tabellen ligt dat niet zo simpel. Bij het zonder enige voorzorg weghalen van een sleutel kunnen andere sleutels 'kwijtraken' omdat we op hun zoekpad een leeg adres tegenkomen.

Voorbeeld. In een eerder opgestelde tabel (h(K) = K *mod* 11, lineair hashen) verwijderen we de sleutel 60. Hierdoor worden (met het gewone zoekalgoritme voor hash-tabellen) de sleutels 38 en 40 niet meer gevonden. Het zoeken stopt immers op het lege adres 5, waar eerder de sleutel 60 stond.



■

Bij het verwijderen van één sleutel zouden we eigenlijk van alle sleutels opnieuw het adres moeten berekenen; de tabel wordt opnieuw *gehasht*. Dit is natuurlijk relatief veel werk. Beter is het om de plaats van het verwijderde element op een speciale manier te markeren, bijvoorbeeld met een gereserveerde sleutel (zoals we ook de sleutel *Leeg* voor lege plaatsen gereserveerd hebben).

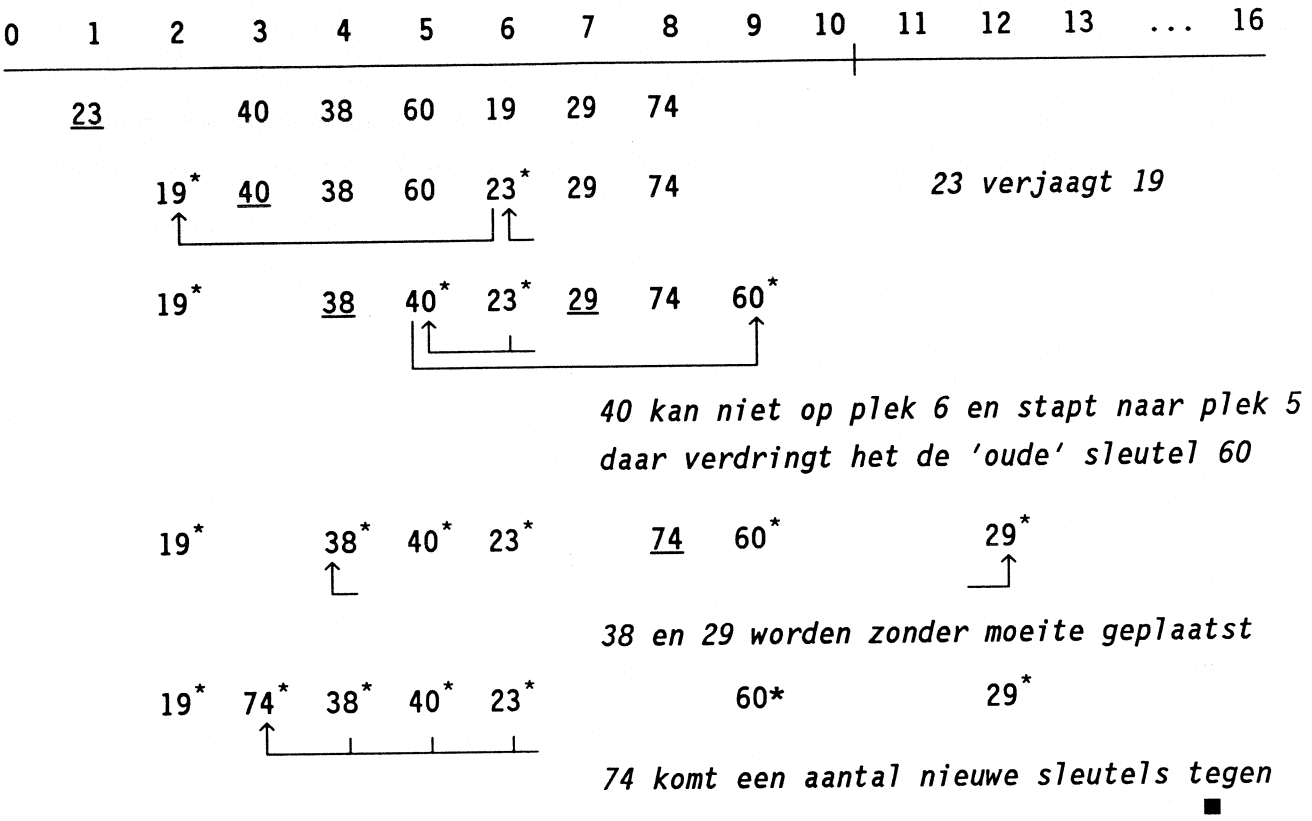
Dit betekent dat we een plaats van de tabel kwijtraken iedere keer wanneer we een element verwijderen. Hierdoor neemt de efficientie van het zoeken snel af. Om de tabel weer 'schoon' te krijgen, en daarmee de zoeksnelheid te verhogen, kan de tabel van tijd tot tijd opnieuw gehasht worden waarbij dan de plekken van de verwijderde elementen weer vrijgegeven worden.

Onderstaand algoritme voert dit opnieuw berekenen van de hashtabel uit. Het is ook toepasbaar indien een tabel vergroot moet worden omdat de vullingsgraad erg groot is. De nieuwe tabel kan dan de oude in de eerste elementen overlappen; we noemen dit ter plekke opnieuw hashen. Omdat de tabel groter wordt hebben we dan natuurlijk ook een andere hash-functie nodig. Dit betekent dat sleutels twee adressen hebben: die in de oorspronkelijke tabel (het oude adres) en een nieuw adres. Het algoritme heeft een aantal elementen gemeen met geordend hashen. Bij geordend hashen worden kleine sleutels door grotere verdrongen. Hier verdringen de 'nieuwe' sleutels (dwz. de sleutels met hun nieuwe adres) de 'oude' sleutels (dwz. de sleutels die nog op hun oude adres staan). Waarschuwing: *ter plekke* opnieuw hashen levert *niet* hetzelfde resultaat op als gewoon de tabel leeghalen en dan de sleutels weer één voor één toevoegen in een grotere tabel. Het voordeel van ter plekke hashen is natuurlijk dat we geen extra ruimte hoeven te reserveren om de sleutels tijdelijk op te bergen als we de tabel leeghalen. Ze blijven immers gewoon in de tabel staan. Wel nodig is voor elk tabelelement een bit dat aangeeft of de aanwezige sleutel 'oud' of 'nieuw' is.

Algoritme : opnieuw hashen (ter plekke)

```
                                { M1, M2 oude resp. nieuwe tabelgrootte,  }
                                { h(K), p(K) nieuwe hash-functies.        }
                                { Hulp van zelfde type als tabelelementen.}
for 0 ≤ Plek ≤ M2-1
do   Nieuw[Plek] := false;
     if T[Plek].Sleutel = Verwijderd then T[Plek].Sleutel := Leeg fi
od;
Plek := 0; Hulp.Sleutel := Leeg;
repeat
     if T[Plek].Sleutel ≠ Leeg and not Nieuw[Plek]
     then verwissel Hulp en T[Plek];
          repeat
               K := Hulp.Sleutel;
               Adres := h(K); Stap := p(K);
               while Nieuw[Adres]
               do   Adres := Adres-Stap;
                    if Adres < 0 then Adres := Adres+M2 fi
               od;
               verwissel Hulp en T[Adres];
               Nieuw[Adres] := true
          until Hulp.Sleutel = Leeg
     fi;
     Plek := Plek+1
until Plek = M1;
```

Voorbeeld. Een tabel met 11 elementen wordt uitgebreid tot 17 elementen. De nieuwe adres-functie wordt h(K) = K *mod* 17, de stap-functie p(K) = 1 (dus lineair hashen). We geven telkens de nieuw geplaatste sleutels met een sterretje aan. De sleutels die in de volgende stap(pen) aan de beurt zijn worden onderstreept.



■

3.6. De Keuze van een Hash-functie

Bij het kiezen van een adres-functie zijn er twee factoren belangrijk. De hash-functie mag niet te veel rekenwerk met zich mee brengen, terwijl de functie verder zoveel mogelijk botsingen moet vermijden. Dit leidt in het algemeen tot een compromis.

Verder veroorzaken regelmatigheden in de invoer vaak een vorm van clustering. Dit beïnvloedt de gemiddelde zoektijd dan nadelig. Houd hier zoveel mogelijk rekening mee.

omzetten van letters. Hash-functies worden vaak gedefinieerd voor getallen. Het is dus zaak sleutels die bestaan uit letters om te zetten in getallen. Dit kan meestal door de letters op te vatten als cijfers in een ander getalstelsel. Bij computergebruik ligt dan een 128- of 256-tallig stelsel voor de hand.

Een aantal veel voorkomende methoden om adres-functies te berekenen zijn:

▷ *truncation.* Neem de laatste bits van de sleutel. Dit is een speciaal geval van:

▷ *extraction.* Neem een aantal bits van de sleutel, van vooraf vastgestelde plekken. Goed opletten dat dit geen opeenhoping van sleutels oplevert: bits die min of meer afhankelijk van elkaar zijn dienen vermeden te worden.

▷ *folding.* Deel de sleutel in stukjes en tel deze op (modulo een geschikt getal).

▷ *mid-squaring.* Kwadrateer de sleutel en neem uit het resultaat de middelste bits. Dit geeft een redelijk onvoorspelbaar gedrag. Sleutels die dicht bij elkaar liggen komen ver uit elkaar in de tabel te liggen. Wanneer de aangeleverde sleutels met grote waarschijnlijkheid dicht bij elkaar liggen kan dit voordelig zijn. Het verhindert clustering in de tabel. Uitgekeken dient te worden wanneer de sleutels beginnen of eindigen met tamelijk veel nullen. Deze komen dan ook veel in het midden van het kwadraat voor.

▷ *(prime) division.* Het adres wordt gelijk aan de sleutelwaarde modulo de tabelgrootte (zoals in de voorbeelden). Deze methode blijkt gunstige resultaten op te leveren, vooral wanneer de tabelgrootte een priemgetal is. We zullen hier nog speciaal op ingaan.

De stap-functie. Bij de keuze van de tweede hash-functie p moet allereerst gelden dat de berekende stapgroottes relatief priem zijn met de tabelgrootte (om ervoor te zorgen dat de bezochte adressen een permutatie vormen van de tabel adressen). Als aan deze voorwaarde is voldaan noemen we de stapfunctie toelaatbaar. Het is daarom vaak handig om als tabelgrootte M een priemgetal te kiezen; p(K) mag dan alle waarden van 1 tot en met M-1 aannemen. Een andere mogelijkheid is om een tabelgrootte te nemen die een macht van twee is; elk oneven getal kan dan stapgrootte zijn.

Verder dienen we er op te letten dat de stap-functie zo weinig mogelijk afhankelijk is van de adres-functie. Dit heeft een gunstige invloed op het aantal zoekstappen. Ook wanneer volgens onze definitie sprake is van dubbel hashen (de stap-functie is niet uit de adres-functie te berekenen) hoeft de stap-functie nog niet optimaal te zijn. Neem bijvoorbeeld een tabel van 1024 elementen. Als we voor sleutel K adres K *mod* 1024 kiezen, en als stapgrootte 1, 3 of 5 (afhankelijk van de waarde K *mod* 3), dan zijn h(K) en p(K) formeel niet afhankelijk. Er worden echter per adres slechts drie mogelijke zoekpaden gebruikt in plaats van zo'n 500 mogelijke (nl. alle oneven stapgroottes).

Adressering modulo de tabelgrootte (*division*). Bij een tabelgrootte M is het adres van sleutel K gelijk aan K *mod* M. Dit blijkt een snel te berekenen adres-functie te zijn met goede resultaten wat zoektijden betreft.

We moeten echter uitkijken voor onverwachte effecten.

Wanneer M bijvoorbeeld even is worden alle even sleutels op even adressen afgebeeld. In sommige toepassingen levert dit nadelen op die soms ook voorkomen bij *extraction*. Als een bestand toevallig veel even sleutels bevat (de laatste bits van de sleutel geven een jaartal aan, we hashen een bepaalde jaargang) benutten we in eerste instantie slechts de helft van de beschikbare adressen.

Het beste resultaat geeft deze methode daarom wanneer de tabelgrootte een priemgetal is. Zelfs dan kunnen er bijzondere verschijnselen optreden.

We gaan uit van een sleutel die bestaat uit letters uit een alfabet met $256 = 2^8$ symbolen (deze voorstelling is niet zó onwaarschijnlijk). Een drieletterige sleutel $L_2L_1L_0$ wordt dan opgevat als het getal $\ell_2 \cdot 256^2 + \ell_1 \cdot 256 + \ell_0$ waarbij ℓ_i de orde van L_i is. Bij een tabelgrootte van 257 (priem!) wordt het adres gelijk aan $\ell_2 \cdot 256^2 + \ell_1 \cdot 256 + \ell_0$ *mod* $257 = \ell_2 - \ell_1 + \ell_0$; een wel erg simpele samenstelling van de oorspronkelijke sleutel (vergelijk *folding*). De algemene les die hieruit getrokken kan worden is om tabelgroottes van de vorm $r^k \pm a$, waar k en a kleine getallen zijn en r het aantal mogelijke letters, te vermijden.

Bij het hashen van variabelenamen (zoals door compilers wel gebeurt) komen vaak reeksen in de sleutels voor (VAR1, VAR2, VAR3). Een prettige eigenschap van *division* is dat deze sleutels gegarandeerd niet botsen (alhoewel ze helaas wel clusteren).

IV. GRAFEN

In dit hoofdstuk kijken we naar grafen. In het algemeen beschouwen we zowel gerichte als ongerichte grafen. We laten echter géén evenwijdige takken en geen lussen toe: we kijken alleen naar *normale* grafen.

1. Specificatie en Representatie

Specificatie. Een (gerichte) graaf bestaat uit twee soorten objecten: knopen en takken. Elke tak wordt bepaald door twee knopen: de begin- en eindknoop. In sommige toepassingen is het van belang om ook een waarde aan takken te kunnen toekennen. Een mogelijke specificatie is als volgt:

```
definition module Digraph;

    type Digraph;
    type Vertex;
    type Edge;

    var NilVertex: Vertex; NilEdge: Edge;

    procedure InitDigraph(var G: Digraph);
    procedure InsertVertex(val: VertexVal; var G: Digraph): Vertex;
    procedure InsertEdge(v,w: Vertex; var Digraph); Edge;
    procedure GetEdgeVal(e: Edge; var eval: EdgeVal);
    procedure SetEdgeVal(e: Edge; val: EdgeVal);

    procedure FirstVertex(var G: Digraph): Vertex;
    procedure NextVertex(v: Vertex; var G: Digraph): Vertex;
    procedure FirstEdge(v: Vertex; var G: Digraph): Edge;
    procedure NextEdge (v: Vertex; e: Edge; var G: Digraph): Edge;
    procedure Endpoint(e: Edge; var G: Digraph): Vertex;

end Digraph.
```

De specificatie gaat impliciet uit van een zekere ordening van knopen (en takken). De procedures FirstVertex en NextVertex leveren de eerste knoop van de graaf, respectievelijk de opvolger van een gegeven knoop in de graaf. Indien er geen opvolger is wordt een gereserveerde waarde NilVertex afgeleverd. Een soortgelijk verhaal geldt voor FirstEdge en Nextedge. Met behulp van deze operaties kunnen alle uitgaande takken bij een gegeven knoop gevonden worden, in een verder ongespecificeerde volgorde, op de volgende manier. (opmerking: dit is weer onze pseudo Pascal taal, voor Modula-2 **od** vervangen door **end**.)

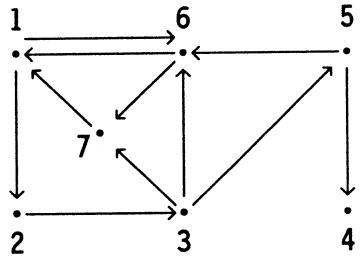
```
e := FirstEdge (v,G)
while e ≠ NilEdge
do w := Endpoint(e,G);
   e := NextEdge (v,e,G)
od
```

Het is nogal opvallend dat er geen procedure StartPoint is die het beginpunt van een tak geeft. Het lijkt erop dat de specificatie is geschreven met een bepaalde implementatie in gedachten.

Representatie. Er zijn twee gangbare methoden om grafen te representeren, de *adjacency matrix* en de *adjacency list*. In het nederlands zouden we deze representaties buur-matrix en buur-lijst kunnen noemen. Kenmerkend voor beide methoden is dat het aantal knopen vooraf vastgelegd wordt (of in ieder geval aan een maximum gebonden) doordat we de knopen in een array plaatsen. Binnen deze beperkingen is het aantal takken flexibel.

Bij het college Discrete Wiskunde is de *adjacency matrix* van een graaf behandeld onder de naam *structuurmatrix*; we hoeven over deze representatie dus niet veel meer te zeggen. De matrix is een twee dimensionaal array waarvan element (i,j) aangeeft of er een tak loopt van knoop i naar knoop j. Merk op dat de matrix altijd evenveel ruimte in beslag neemt, ongeacht het aantal takken van de graaf.

Voorbeeld. Een (gerichte) graaf en bijbehorende adjacency matrix (nullen weggelaten).



van

	1	2	3	4	5	6	7
1	.	1	.	.	.	1	.
2	.	.	1	.	.	.	.
3	.	.	.	.	1	1	1
4	.	.	.	.	.	.	.
5	.	.	.	1	.	1	.
6	1	.	.	.	.	.	1
7	1	.	.	.	.	.	.

naar

■

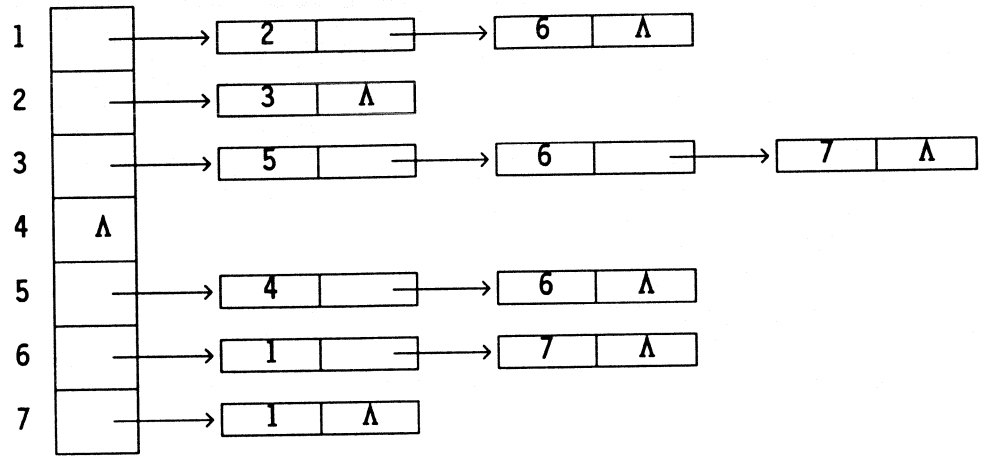
Adjacency List. Bij een adjacency list representatie construeren we bij iedere knoop x een lijst met die knopen naar welke een tak loopt vanuit x. We gebruiken een array met voor elke knoop x een pointer naar het begin van de lijst die bij deze knoop hoort. Dit array heet wel de listheader. We kunnen er eventueel ook andere informatie over de knoop in opslaan. Elke tak vanuit knoop x wordt nu in de lijst bij knoop x weergegeven door de bijbehorende eindknoop.

De ruimte van deze implementatie is dus afhankelijk van het aantal takken (en het aantal knopen) van de graaf. Vooral wanneer de graaf relatief weinig takken heeft is deze representatie voordeliger dan de adjacency matrix. Anderzijds kunnen we niet meer in één oogopslag controleren of er een tak loopt van knoop x naar knoop y. We moeten er nu een lijst voor doorlopen.

Geschikte declaraties in Pascal zijn bijvoorbeeld:

```
CONST MaxKnoop = {...};
TYPE KnoopTal = 1..MaxKnoop;
     TakWijzer = ↑Tak;
     Tak = RECORD
               EindKnoop : KnoopTal;
               Volger : TakWijzer;
           END;
     AdjList = ARRAY[KnoopTal] OF TakWijzer;
```

Voorbeeld. De Adjacency list van de graaf uit het bovenstaande voorbeeld.



■

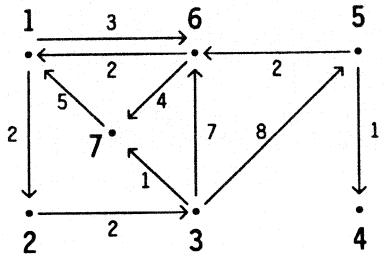
Grafen met gewichten. Vaak dragen takken nog informatie met zich mee. We zullen hier vooral denken aan een getal (het gewicht van de tak) dat bijvoorbeeld kosten of capaciteit kan voorstellen. We kunnen dit getal opnemen in de adjacency matrix en (met een aanpassing van de declaraties) in de adjacency list.

Bij de matrix representatie moeten we dan afspreken welke waarde we geven aan niet aanwezige takken. Wanneer de gewichten capaciteiten voorstellen zal deze waarde 0 zijn, terwijl we bij kosten de waarde ∞ (dus iets heel groots, bijvoorbeeld MAXINT) zullen kiezen omdat de kosten van het doorlopen van een niet bestaande tak heel hoog zullen zijn.

Bij de lijst representatie nemen we aan dat bij de declaratie van Tak is toegevoegd de component

```
Gewicht: Integer;
```

Voorbeeld. We kennen gewichten toe aan de takken van de hiervoor gegeven graaf. Deze gewichten stellen afstanden tussen knopen voor. Niet bestaande takken worden door de afstand ∞ gerepresenteerd.



	1	2	3	4	5	6	7
1	∞	2	∞	∞	∞	3	∞
2	∞	∞	2	∞	∞	∞	∞
3	∞	∞	∞	∞	8	7	1
4	∞	∞	∞	∞	∞	∞	∞
5	∞	∞	∞	1	∞	2	∞
6	2	∞	∞	∞	∞	∞	4
7	5	∞	∞	∞	∞	∞	∞

■

Ongerichte grafen. Tot nu toe namen we als voorbeeld steeds gerichte grafen. De representaties kunnen ook voor ongerichte grafen gebruikt worden door de ongerichte tak {x,y} op te vatten als twee gerichte takken (x,y) en (y,x), die we allebei in de representatie opnemen. Merk op dat de matrix bij een ongerichte graaf daarmee symmetrisch wordt.

2. GRAAFWANDELINGEN

Bij bomen kennen we een aantal systematische wandelingen door de boom. Twee van deze manieren om de knopen van een boom te bezoeken kunnen we wat algemener ook voor een graaf gebruiken. Deze manieren zijn de WLR-wandeling (pre-orde of *depth-first search*) en de nivo-orde wandeling (*breadth-first search*). Bij grafen ligt het belang van de wandelingen vooral in het feit dat ze als grondslag voor andere algoritmen gebruikt kunnen worden.

In de diepte zoeken. Net als bij bomen laat deze methode zich het beste recursief beschrijven. De wandeling begint in een knoop x van de graaf en volgt daarna één-voor-één de wandelingen vanuit de knopen die bereikbaar zijn met takken vanuit x, waarbij telkens alleen knopen beschouwd worden die nog niet eerder bezocht zijn.

Om deze wandeling te implementeren gebruiken we een globaal Boolean array Bezocht dat aangeeft welke knopen tijdens de wandeling bezocht zijn. Dit array moet geïnitialiseerd worden op **false**.

```
procedure DFS(x: KnoopTal);
    Bezocht[x] := true;
    Bezoek(x);
    for elke knoop w aangrenzend aan x
    do   if not Bezocht[w]
         then DFS(w)
         fi
    od
endprocedure;
```

In de breedte zoeken. Bij *breadth-first* wandelen wordt eerst een knoop x uit de graaf bezocht, vervolgens alle knopen die met een tak vanuit x bereikbaar zijn, hierna weer de knopen die aan deze laatste knopen grenzen (voorzover ze nog niet bezocht zijn), enzovoorts. Merk op dat de knopen bezocht worden in volgorde van afstand tot de eerste knoop van de wandeling.

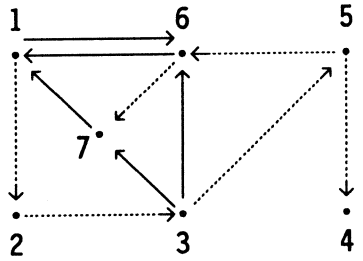
De meest voor de hand liggende datastructuur om bij deze wandeling te gebruiken is natuurlijk een rij (*queue*). We schrijven het algoritme in abstracte notatie (vergelijk de boomwandeling mbv. stapel).

Algoritme : graafwandeling mbv. rij (*breadth-first search*)

```
Initialiseer(Q);
Q ⇐ 1; Bezocht[1] := true;
while not IsLeeg(Q)
do   x ⇐ Q;
     Bezoek(x);
     for elke knoop w aangrenzend aan x
     do   if not Bezocht[w]
          then Q ⇐ w;
               Bezocht[w] := true
          fi
     od
od;
```

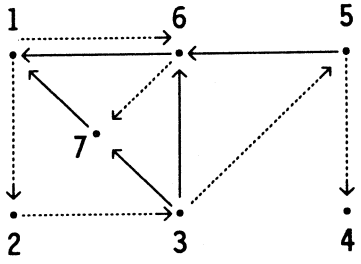
Voorbeeld. Wandelingen in een gerichte graaf. Beide graafwandelingen definiëren een opspannende boom voor de graaf door die takken te nemen waarlangs de wandeling verloopt. We geven deze opspannende bomen. Bij de wandelingen kiezen we de knopen zoveel mogelijk in oplopende nummering. Vanuit knoop 3 gaan we dus eerst naar knoop 5, dan naar 6 en uiteindelijk naar 7 (mits deze knopen nog niet eerder bezocht werden).

(1) *Depth-first search*



1, 2, 3, 5, 4, 6, 7

(2) *Breadth-first search*

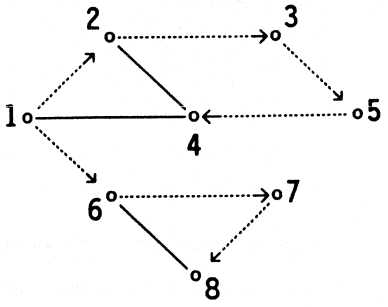


1, 2, 6, 3, 7, 5, 4

Tak ·····→ in bijbehorende opspannende boom. ■

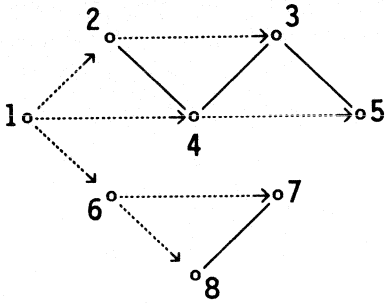
Voorbeeld. Wandelingen in een ongerichte graaf, met bijbehorende opspannende bomen.

(1) *Depth-first search*



1, 2, 3, 5, 4, 6, 7, 8

(2) *Breadth-first search*



1, 2, 4, 6, 3, 5, 7, 8

Tak ·····→ in bijbehorende opspannende boom (in wandelrichting). ■

3. Minimale Opspannende Boom

3.1. Het Algoritme

In deze paragraaf bekijken we een *ongerichte* graaf G = (V,E) met gewichten op de takken. We zijn geïnteresseerd in de constructie van een opspannende (=voortbrengende) boom waarvan de som van de gewichten van alle takken minimaal is.

Besproken wordt een algoritme dat afkomstig is van Prim. Het is een gretig algoritme; er wordt telkens een *lokale* optimale keuze gemaakt zonder direct rekening te houden met de uiteindelijke (globale) oplossing. Het zal duidelijk zijn dat deze methodiek lang niet voor alle soorten problemen een (optimale) oplossing zal vinden. Bij het college Discrete Wiskunde wordt de wiskundige theorie behandeld die ten grondslag ligt aan die problemen (zoals het construeren van een minimale opspannende boom in een graaf) waarvoor een 'gretige' aanpak de juiste oplossing levert.

Prim's algoritme begint in een willekeurige knoop van de graaf en laat van daaruit de opspannende boom 'groeien'. Dit gebeurt door telkens een tak te kiezen die aan één zijde aan de boom grenst en een minimaal gewicht heeft. Deze tak (en de bijbehorende eindknoop) wordt dan aan de boom toegevoegd. Deze stap wordt herhaald totdat alle knopen van de graaf tot de opspannende boom behoren.

Informeel ziet het algoritme er dus als volgt uit; (TV,TE) wordt uiteindelijk de minimale opspannende boom.

Algoritme : minimale opspannende boom volgens Prim

Kies een willekeurige knoop x;
TV := {x}; TE := ∅;
while TV ≠ V
do *Kies tak* {x,y} *met* x ∈ TV, y ∉ TV *en met minimaal gewicht*;
TV := TV ∪ {y};
TE := TE ∪ {{x,y}}
od;

Correctheid. We moeten nog laten zien dat dit 'gretige' algoritme een optimale oplossing levert. Dit kunnen we doen met volledige inductie naar het aantal knopen dat al aan de boom is toegevoegd, waarbij we gebruik maken van het volgende resultaat. (Teken een plaatje bij het bewijs !)

Stelling. G is een ongerichte graaf met gewichten. Laat de boom B = (TV,TE) een deelgraaf van G = (V,E) zijn (TV ⊂ V). Kies de tak {x,y} als in het algoritme. Als B uitgebreid kan worden tot een optimale opspannende boom T van G, dan kan dat ook voor B′ = (TV∪{y},TE∪{{x,y}}).

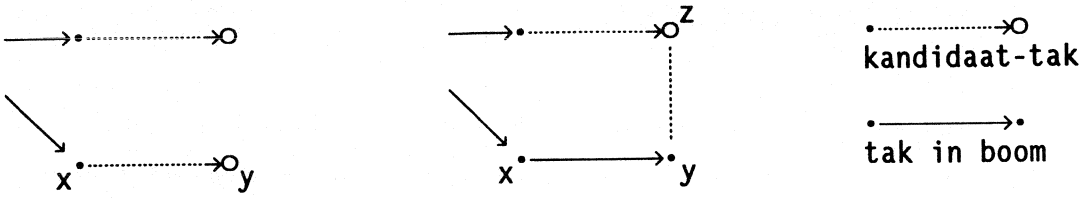
Bewijs. Als {x,y} zelf een tak in T is dan zijn we klaar. Neem dus aan dat {x,y} *niet* tot T behoort. Als {x,y} aan T wordt toegevoegd, dan ontstaat een kring. Deze kring bevat zowel knopen *in* als *buiten* TV omdat x ∈ TV en y ∉ TV. De tak {x,y} verlaat TV. Omdat de kring uiteindelijk weer in TV terugkeert moet langs de kring nog een tak t′ liggen met één knoop in TV en de andere erbuiten. Het gewicht van t′ is niet kleiner dan het gewicht van {x.y}, omdat deze laatste tak gekozen is vanwege zijn minimale gewicht. Vervangen we in T de tak t′ door {x,y} dan ontstaat weer een opspannende boom T′ van G, waarvan het totale gewicht niet groter is dan dat van T en dus ook optimaal is. B′ kan dus uitgebreid worden tot T′. ■

3.2. Een Implementatie

We zullen ons best doen om het boven geschetste algoritme in Pascal te implementeren. We gebruiken de adjacency list representatie.

Er dient onderscheid gemaakt te worden tussen knopen in en buiten de tot dusver geconstrueerde boom. We geven dit aan in een array die per knoop één van de waarden InBoom of BuitenBoom bevat. Herhaaldelijk moet een minimale tak gezocht worden tussen een knoop binnen en een knoop buiten de boom. We willen daarvoor niet iedere keer alle takken bekijken. Bij elke knoop buiten de boom behoren een aantal takken vanuit knopen in de boom. Van deze takken wordt er hoogstens één in de opspannende boom gekozen. Per knoop onthouden we daarom

slechts één *kandidaat*-tak, nl die met minimaal gewicht, en vergelijken alleen de gewichten van deze kandidaat-takken met elkaar. Telkens als een knoop aan de boom wordt toegevoegd moeten ook de kandidaat-takken van de knopen buiten de boom worden aangepast. We hoeven daarvoor echter alleen maar te kijken naar de uitgaande takken van de zojuist toegevoegde knoop.



Plaatje. Voeg {x,y} toe aan boom. Is {y,z} een betere kandidaat voor z ?

Het eerdere algoritme is dus iets precieser ingevuld. We hebben:

Begin met een knoop x;
Laat de boom bestaan uit de knoop x;
De kandidaat-tak voor elke andere knoop y *wordt de tak* {x,y};
while *Nog niet alle knopen gehad*
do *Kies knoop* y *buiten boom met minimale kandidaat-tak*;
Voeg y *toe aan knopen van boom*;
Pas kandidaat-takken aan;
od;

De kandidaat-tak voor elke knoop buiten de boom slaan we op in een array (of eigenlijk in twee arrays, één voor het gewicht en één voor de andere knoop - die in de boom - van de tak). Deze manier van representeren heeft een prettige bijkomstigheid. Na afloop van de berekeningen staat de opspannende boom in het array van kandidaat-takken. Voor elke knoop hebben we zo namelijk de inkomende tak in de opspannende boom vastgelegd. Uitzondering hierop is de knoop waar we het algoritme starten. Deze heeft geen inkomende tak in de opspannende boom. (Niet zo raar, we weten immers dat een boom één tak minder heeft dan knopen.)

Nog één kleinigheidje. Voor sommige knopen bestaat er eenvoudig geen inkomende tak vanuit de opspannende boom. We zetten het kandidaat gewicht voor die knopen op een groot getal (bv. MAXINT) dat we aangeven met ∞. We doen alsof dit het gewicht is van de (niet-bestaande) tak die loopt vanuit knoop 1.

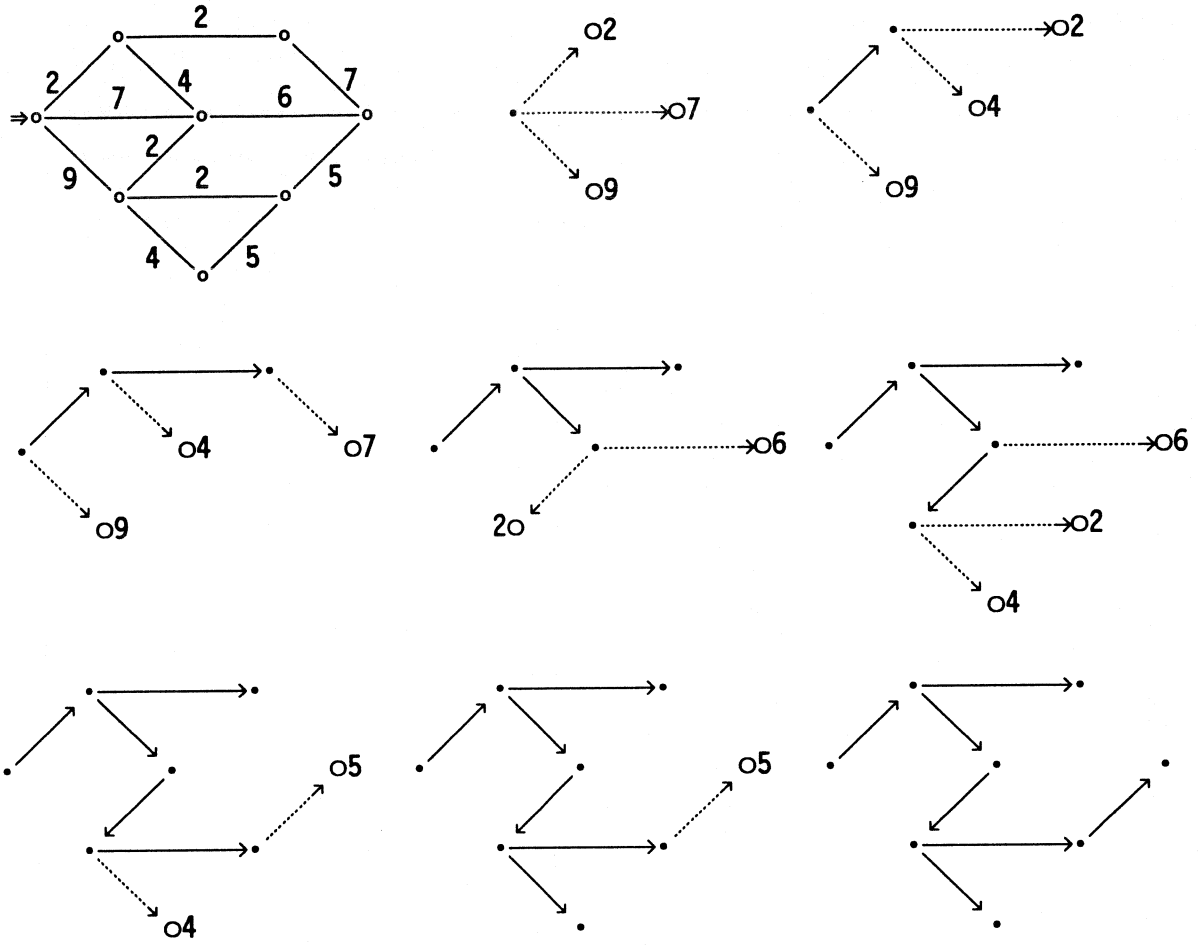
```
VAR  KandWaarde: ARRAY[KnoopTal] OF Integer;
     KandOuder: ARRAY[KnoopTal] OF KnoopTal;
     Graaf: AdjList;
     Status: ARRAY[KnoopTal] OF (InBoom, BuitenBoom);
     y: KnoopTal;
     AantalInBoom: Integer;


PROCEDURE PasKandidatenAan(y: KnoopTal);
VAR  Tak: TakWijzer;
     z: KnoopTal;
BEGIN
     Tak := Graaf[y];
     WHILE Tak ≠ NIL
     DO BEGIN
          z := Tak↑.EindKnoop;
          IF (Status[z] = BuitenBoom) AND (Tak↑.Gewicht < KandWaarde[z])
          THEN BEGIN
               KandWaarde[z] := Tak↑.Gewicht;
               KandOuder[z] := y
          END;
          Tak := Tak↑.Volger
     END
END;


FUNCTION KnoopMetMinimaleKandidaat: KnoopTal;
VAR  x, Kand: KnoopTal;
     MinGewicht: Integer;
BEGIN
     Kand := 1;
     MinGewicht := ∞;
     FOR x := 2 TO MaxKnoop DO
          IF (Status[x] = BuitenBoom) AND (KandWaarde[x] < MinGewicht)
          THEN BEGIN
               MinGewicht := KandWaarde[x];
               Kand := x
          END;
     KnoopMetMinimaleKandidaat := Kand
END;


BEGIN                                                    { Programma }
     FOR y := 2 TO MaxKnoop
     DO BEGIN
          Status[y] := BuitenBoom;
          KandWaarde[y] := ∞;
          KandOuder[y] := 1
     END;
     Status[1] := InBoom;
     AantalInBoom := 1;
     PasKandidatenAan(1);
     WHILE AantalInBoom ≤ MaxKnoop
     DO BEGIN
          y := KnoopMetMinimaleKandidaat;
       { IF y = 1 THEN Foutmelding; }
          Status[y] := InBoom;
          AantalInBoom := AantalInBoom+1;
          PasKandidatenAan(y);
     END
END;
```

Voorbeeld. De constructie van een minimale opspannende boom volgens Prim's algoritme. In de plaatjes geven we de kandidaat-tak voor elke knoop buiten de boom, voorzover die tak bestaat. Denkbeeldige takken met gewicht ∞ vanuit de beginknoop zijn dus niet weergegeven.



tak in opspannende boom

o2 Kandidaat-tak, met gewicht ■

4. Kortste Paden

Een in de praktijk veel voorkomend probleem is het bepalen van de (minimale) afstand tussen knopen uit de graaf, waarbij de lengte van een wandeling de som van de gewichten van de takken is. We kunnen dit probleem zowel voor ongerichte als voor gerichte grafen bekijken, maar we dienen er in het algemeen voor te zorgen dat de gewichten niet negatief zijn.

We willen hier bestuderen hoe we de afstand tussen twee gegeven knopen uit de graaf kunnen bepalen. Voor dit probleem is geen (essentieel) beter algoritme bekend dan het algoritme dat tegelijkertijd *alle* andere afstanden vanuit de beginknoop berekent. We zullen daarom dit laatste algoritme geven. Wanneer het algoritme de gezochte afstand berekend heeft, kunnen we het eventueel (voortijdig) stoppen.

4.1. Afstanden vanuit een Gegeven Knoop

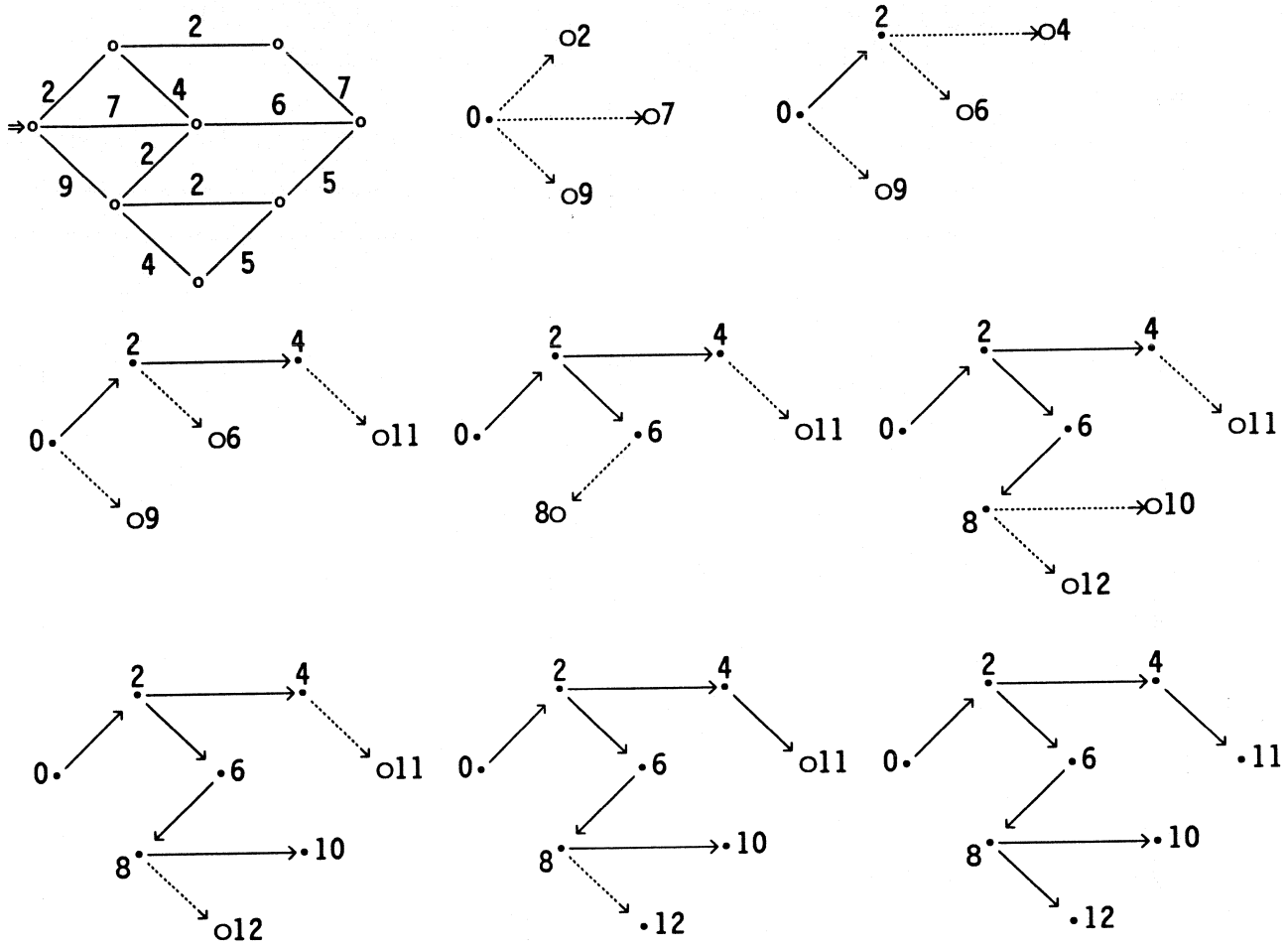
Stel dat we in een graaf vanuit een gegeven knoop (de *bron*) naar elk van andere knopen een kortste pad aan willen geven. Het is niet al te moeilijk om in te zien dat deze paden dan een boomstructuur zullen gaan vormen. Immers het kortste pad naar een knoop loopt via andere knopen die dichter bij de bron liggen. De verste knoop wordt dus door een tak verbonden met de kortste afstanden boom voor de overige knopen. Met inductie levert dit een boomstructuur.

Het levert ook een idee om deze boom te construeren. Net als bij het opspannende boom algoritme voegen we steeds takken aan een boom toe, nú echter steeds de tak naar de knoop die het dichtst bij de bron ligt. Dit is weer een gretig algoritme. Het staat bekend onder de naam van Dijkstra, die tegelijkertijd Prim's algoritme (her)ontdekte.

Algoritme : kortste paden volgens Dijkstra

```
for elke knoop x    do BoomAfstand[x] := ∞ od;
Kies een startknoop x;
TV := {x}; TE := ∅;
BoomAfstand[x] := 0;
while TV ≠ V
do   Kies tak {x,y) met x ∈ TV, y ∉ TV,
          waarbij Afstand = BoomAfstand[x]+Gewicht(x,y) minimaal is;
     TV := TV ∪ {y}; TE := TE ∪ {{x,y}};
     BoomAfstand[y] := Afstand
od;
```

Voorbeeld. De constructie van een boom van kortste afstanden volgens Dijkstra's algoritme. De getallen bij de knopen geven de 'boomafstand' voor de knoop weer. Knopen met boomafstand ∞ zijn weggelaten. Tevens is de tak getekend via welke de boomafstand minimaal is.

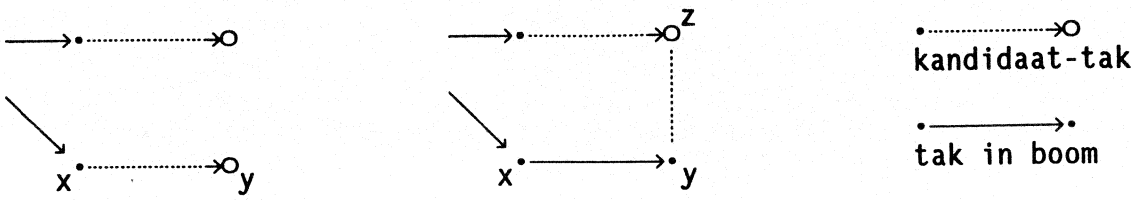


2•——→•4 tak in boom van kortste afstanden, met afstanden

•·····→o10 Kandidaat-tak, afstand via deze tak (tot nu toe kortste) ■

De implementatie die we voor Prim's algoritme gaven kan op eenvoudige wijze voor dit algoritme aangepast worden. De kandidaat-tak voor een gegeven knoop y is nu steeds de tak waarlangs de boomafstand tot y minimaal zou zijn. De berekening van Kandgewicht voor een knoop moet daartoe aangepast worden. De waarde die in Kandgewicht opgeslagen wordt is steeds de afstand tot de startknoop via de kandidaat-tak en de reeds geconstrueerde boom.

Neem nu aan dat we tak {x,y} aan de boom toevoegen, x binnen en y buiten de boom. Voor elke knoop z buiten de boom moeten we nagaan of de tak {y,z} (als die bestaat) een betere kandidaat is. De afstand via de tak {y,z} tot de startknoop is gelijk aan het gewicht van {y,z} plus de afstand (dwz. het kandgewicht) van y. (zie ook plaatje)



Plaatje dat al eerder geweest is.

Wanneer we bovenstaande opmerkingen verwerken, krijgen we het volgende stuk programma.

```
PROCEDURE PasKandidatenAan( y: KnoopTal);
VAR  Tak: TakWijzer;
     z: KnoopTal;
BEGIN
     Tak := Graaf[y];
     WHILE Tak ≠ NIL
     DO BEGIN
          z := Tak↑.EindKnoop;          (*  Tak is de tak {y,z}   *)
          IF Status[z] = BuitenBoom
          THEN BEGIN
               Afstand := KandWaarde[y] + Tak↑.Gewicht;
               IF   Afstand < KandWaarde[z]
               THEN BEGIN
                    KandWaarde[z] := Afstand;
                    KandOuder[z] := y;
               END
          END;
          Tak := Tak↑.Volger
     END
END {PasKandidatenAan};
```

Een beetje complexiteit. Hoewel we in dit college de algoritmes niet nauwgezet op hun efficiëntie analyseren, willen we voor bovenstaand algoritme toch enig inzicht krijgen in het benodigde aantal stappen bij een gegeven graaf. Veronderstel dat de gerepresenteerde graaf n knopen en e takken heeft. Elke tak wordt éénmaal gebruikt om te controleren of die tak een kandidaat-tak moet gaan vervangen. Dit kost dus $O(e)$ stappen bij de *adjacency-list* representatie. De hoofdlus van het programma wordt n-1 maal doorlopen, voor elk toe te voegen knooppunt éénmaal. Elke keer moet de knoop met minimale kandidaat bepaald worden. Omdat er n knopen zijn kost dit uiteindelijk totaal $O(n^2)$ stappen. Het helpt slechts een beetje om de knopen buiten de boom in een lijst te hangen. Het totaal aantal knopen wat dan wordt bekeken is $(n-1)+(n-2)+\ldots+1 = n\cdot(n-1)$. Dit is ook kwadratisch. Omdat (voor een samenhangende graaf $n <) e \leq n\cdot(n-1)$ is de totale complexiteit $O(e+n^2) \subseteq O(n^2)$.

Dit kan soms beter. Stop de kandidaat-takken in een heap. De heap bevat ten hoogste voor elke knoop één element, dus maximaal n elementen. Een heap

operatie kost daarom maximaal $lg(n)$ stappen. Er moet maximaal e maal een kandidaat-tak vervangen worden door een nieuwe, en er moet n maal de minimale kandidaat gekozen worden. Totaal zijn dit $e+n \leq 2 \cdot e$ heap operaties. De complexiteit is daarmee $O(e \cdot lg(n))$. Dit is beter dan $O(n^2)$ mits het aantal takken van de graaf relatief klein is, in ieder geval minder dan $\frac{n^2}{lg(n)}$. Wanneer het aantal takken ongeveer dit aantal is hangt de keuze voornamelijk van de precieze implementatie af.

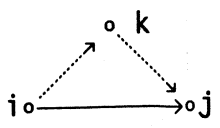
4.2. Alle Afstanden in de Graaf

Om alle afstanden in de graaf te bepalen kunnen we Dijkstra's algoritme natuurlijk voor elke knoop éénmaal uitvoeren. Volgens bovenstaande analyse kost dit in de orde van n^3 stappen. Er bestaat echter een algoritme met de zelfde complexiteit, dat veel eenvoudiger te implementeren is. Het staat bekend onder de naam van Floyd.

Het algoritme maakt weer gebruik van dynamisch programmeren, het berekenen en (tijdelijk) opslaan van deeloplossingen van het probleem (zie het hoofdstuk over speciale bomen: optimale binaire zoekboom).

Het algoritme berekent voor k = 0,1,...,n de matrices A^k, waarvan element $A^k[i,j]$ de minimale afstand tussen knoop i en knoop j geeft, langs een pad dat geen knopen met nummer hoger dan k mag passeren. In het bijzonder mogen de paden voor A^0 geen enkele knoop passeren. De kortste paden voor die matrix bestaan dus steeds uit een enkele tak: de afstand tussen i en j mag alleen langs tak (i,j) gemeten worden.

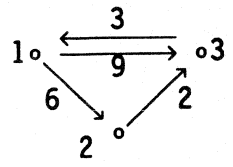
Wanneer het algoritme de matrix A^{k-1} berekend heeft, wordt de matrix A^k berekend. Het kortste pad van i naar j zonder de knopen k+1 en hoger aan te doen loopt al of niet via knoop k. De afstand via knoop k is gelijk aan de som van de al gevonden waarden $A^{k-1}[i,k]$ en $A^{k-1}[k,j]$, de afstand die niet via knoop k loopt hadden we al berekend als $A^{k-1}[i,j]$. Het matrixelement $A^k[i,j]$ is dus het minimum van deze waarden.



via knoop k

via knopen < k

Voorbeeld. We bekijken een klein graafje. In elke stap hoeven slechts twee nieuwe waarden berekend te worden. Immers, de diagonaal blijft nul en bij het berekenen van A^k blijven de k^e kolom en rij onveranderd.



$$\text{Kosten} = A^0 = \begin{bmatrix} 0 & 6 & 9 \\ \infty & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

$$A^1 = \begin{bmatrix} 0 & 6 & 9 \\ \infty & 0 & 2 \\ 3 & \underline{9} & 0 \end{bmatrix} \qquad A^2 = \begin{bmatrix} 0 & 6 & \underline{8} \\ \infty & 0 & 2 \\ 3 & 9 & 0 \end{bmatrix} \qquad A^3 = \begin{bmatrix} 0 & 6 & 8 \\ \underline{5} & 0 & 2 \\ 3 & 9 & 0 \end{bmatrix}$$

$A^1[2,3] = \mathit{min}\{\ A^0[2,3],\ A^0[2,1]+A^0[1,3]\ \} = \mathit{min}\{\ 2,\ \infty+9\ \} = 2$
$A^1[3,2] = \mathit{min}\{\ A^0[3,2],\ A^0[3,1]+A^0[1,2]\ \} = \mathit{min}\{\ \infty,\ 3+6\ \} = \underline{9}\ (\ <\infty\ )$
$A^2[1,3] = \mathit{min}\{\ A^1[1,3],\ A^1[1,2]+A^1[2,3]\ \} = \mathit{min}\{\ 9,\ 6+2\ \} = \underline{8}\ (\ <9\ )$
$A^2[3,1] = \mathit{min}\{\ A^1[3,1],\ A^1[3,2]+A^1[2,1]\ \} = \mathit{min}\{\ 3,\ 9+\infty\ \} = 3$
$A^3[1,2] = \mathit{min}\{\ A^2[1,2],\ A^2[1,3]+A^2[3,2]\ \} = \mathit{min}\{\ 6,\ 8+9\ \} = 6$
$A^3[2,1] = \mathit{min}\{\ A^2[2,1],\ A^2[2,3]+A^2[3,1]\ \} = \mathit{min}\{\ \infty,\ 2+3\ \} = \underline{5}\ (\ <\infty)$ ■

Bij het berekenen van de matrix A^k veranderen de waarden op de plekken [i,k] en [k,j] niet. We kunnen hiervan gebruik maken door niet n+1 verschillende matrices te declareren, maar het rekenwerk in de oorspronkelijke matrix uit te voeren. We herinneren ons nog dat we een ontbrekende tak het gewicht ∞ krijgt. Hierdoor zullen paden die niet bestaan ook dit gewicht krijgen. Bij een implementatie moeten we nog uitkijken. Optellen van een getal bij ∞ zou weer ∞ moeten opleveren. Het optellen van een getal bij MAXINT in Pascal heeft in het algemeen desastreuze gevolgen (afhankelijk van de gebruikte compiler).

Algoritme : alle afstanden in een graaf volgens Floyd

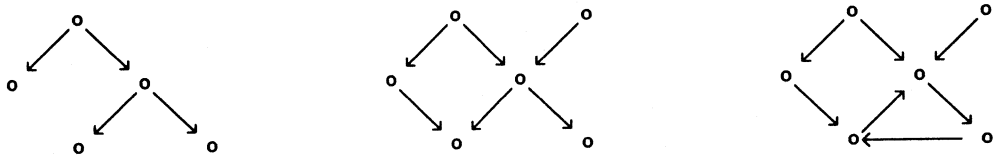
```
{ Kosten bevat de adjacency matrix (met gewichten) van de graaf.  }
{ Afst is de matrix waarin de minimale afstanden berekend worden. }
    Afst := Kosten;
    for k := 1 to AantalKnopen
    do   for i := 1 to AantalKnopen
         do   for j := 1 to AantalKnopen
              do   Afst[i,j] := min( Afst[i,j], Afst[i,k]+Afst[k,j] )
              od
         od
    od;
```

Het is mogelijk om tijdens het berekenen niet alleen de afstanden tussen elk tweetal, maar ook de kortste paden zèlf tussen elk tweetal op efficiënte wijze op te slaan. Verder bestaat er een variant van bovenstaand algoritme, dat voor een gegeven graaf een Boolese matrix T aflevert zodanig dat T[i,j] true is als er een pad in de graaf bestaat van i naar j. Zo'n matrix stelt zelf ook weer een graaf voor en heet de *transitieve afsluiting* van de oorspronkelijke graaf. Dit laatste algoritme is genoemd naar Warshall; het bestond al vóór het algoritme van Floyd.

De complexiteit van deze algoritmes is natuurlijk evenredig aan n^3, waarbij n het aantal knopen van de graaf is (in onze implementatie: MaxKnoop).

5. Topologisch Sorteren

Een gerichte acyclische graaf is (zoals de naam al zegt) een gerichte graaf zonder cykels. Acyclische grafen zijn daarmee algemener dan bomen, maar vormen bij lange na niet alle gerichte grafen. Ze zijn het onderwerp van deze paragraaf.

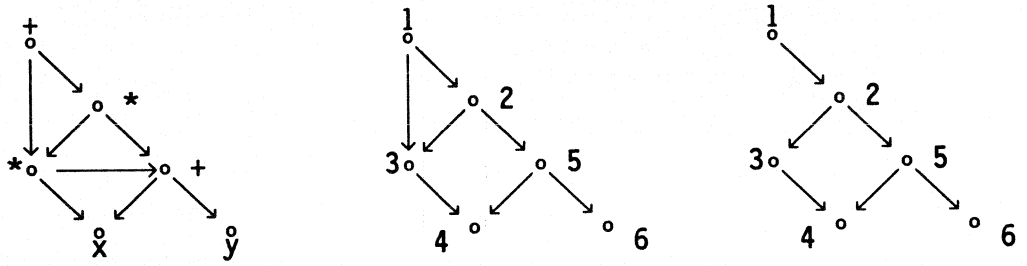


Plaatje. Drie gerichte grafen: boom, acyclisch, met cykel.

Er zijn een aantal belangrijke toepassingen van acyclische grafen. Zo worden ze bij formule manipulatie gebruikt als datastructuur om formules met gedeelde subformules weer te geven. Een andere toepassing is de weergave van taken en hun onderlinge afhankelijkheden. (zie onderstaand voorbeeld)

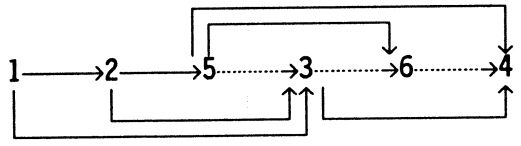
Van het college Logica herinneren we ons de partiele ordening misschien nog wel. Zo'n relatie R is irreflexief (nooit xRx), a-symmetrisch (als xRy dan niet yRx) en transitief (als xRy en yRz dan xRz). De representatie van een partiele ordening (op een eindige verzameling) vormt een acyclische graaf. Vaak worden bij de representatie transitieve takken weggelaten.

Voorbeeld. Linker graaf: representatie van de formule +(*(x,+(x,y)),*(*(x,+(x,y)),+(x,y))), ofwel in infix notatie (x*(x+y))+((x*(x+y))*(x+y)), waarbij twee gedeelde formules onderstreept zijn.



Middelste en rechter graaf: Onderlinge afhankelijkheid van hoofdstukken uit een leerboek. Hoofdstukken 3 en 5, en hoofdstukken 4 en 6 kunnen onafhankelijk van elkaar bestudeerd worden. ■

Wanneer een acyclische graaf gebruikt wordt om taken weer te geven met hun onderlinge afhankelijkheden, dan kunnen we uit de graafstructuur informatie halen over de volgorde waarin de taken uitgevoerd moeten worden. In het laatste voorbeeld kunnen we de hoofdstukken van het boek in de volgorde 1, 2, 5, 3, 6, 4 gelezen worden, zonder dat we bij het lezen van één van de hoofdstukken voorkennis missen. Hetzelfde plaatje kunnen we ook een andere betekenis geven. Elke knoop geeft een Turbo-Pascal module weer, de takken geven aan welke module door andere modules gebruikt wordt. Het gegeven rijtje geeft een mogelijke volgorde waarin de modules gecompileerd kunnen worden.



We willen nu een algoritme geven dat zo'n volgorde bij een acyclische graaf bepaalt. Wat sjieker gezegd: een algoritme dat een partiele ordening inbedt in een lineaire. Dat de gevraagde volgorde ook altijd bestaat leert ons de volgende stelling.

Stelling. Zij G = (V,E) een gerichte graaf zonder kringen. Er bestaat een ordening op de knopen van G, $V = \{v_1, v_2, \ldots, v_n\}$ zó dat als $(v_i, v_j) \in E$, dan $i < j$.

Bewijs. Als G geen kringen heeft, dan is er een knoop zonder inkomende takken. Stel immers dat G geen knopen zonder inkomende takken heeft. We kunnen dan, uitgaande van een knoop een willekeurig lange wandeling in de graaf vinden door telkens bij de voorste knoop een inkomende tak te kiezen en deze vóór de wandeling te plaatsen. Omdat V eindig is moeten we op een gegeven ogenblik een knoop voor de tweede maal tegenkomen. Daarmee hebben we dan ook een kring gevonden. Tegenspraak.

Hiermee wordt de stelling met inductie naar het aantal knopen bewezen.

$|V| = 1$. De ordening ligt nogal voor de hand.

Inductie stap. We weten dat G een knoop v_0 heeft zonder inkomende takken. Bekijk de graaf $G-\{v_0\}$ die uit G ontstaat door v_0 en bijbehorende takken te verwijderen. Deze graaf is weer gericht, acyclisch en heeft één knoop minder. Volgens de inductie aanname is er een ordening op de knopen van $G-\{v_0\}$. Door v_0 nu vóór deze ordening te plaatsen ontstaat een ordening voor G (ga na). ■

Bovenstaand bewijs is niet alleen interessant voor graaftheoretici, maar ook voor ons. Het levert namelijk een (informeel) algoritme om de knopen van een acyclische graaf te ordenen. Het vinden van zo'n ordening (als in de stelling) wordt <u>topologisch</u> <u>sorteren</u> genoemd.

```
while nog knopen in de graaf
do   Kies een knoop x zonder inkomende takken;
     Verwijder x met bijbehorende uitgaande takken;
     Voer x uit
od;
```

Implementatie. We maken een aantal opmerkingen bij een mogelijke implementatie van het algoritme wanneer de graaf als *adjacency lists* gerepresenteerd is. Het is natuurlijk niet erg handig om iedere stap van het algoritme voor elke knoop de inkomende takken te moeten tellen. Daarvoor moeten we namelijk alle lijsten afwerken. We nemen dus aan dat er een array beschikbaar is met daarin voor elke knoop het aantal inkomende takken. Om dit array op te stellen doorlopen we dus éénmaal de lijsten. Iedere maal als we één van de knopen verwijderen, doorlopen we de bijbehorende 'buur'-lijst en passen we de aantallen inkomende takken aan.

Nu zouden we telkens in het array op zoek moeten naar een knoop zonder inkomende takken. Dit zou in het slechtste geval kwadratisch veel tijd kunnen kosten. Beter is het om een lijst aan te leggen van knopen met nul inkomende takken die we elke stap aanpassen.

V. PATROONHERKENNING

In dit hoofdstuk behandelen we het zoeken van een gegeven woord in een vrij lange tekst. Het op te zoeken woord zullen we verder het *patroon* noemen. We nemen aan dat zowel het patroon als de tekst gegeven zijn als een array van letters (zoiets als het Turbo-Pascal String type). Wanneer de doorzochte tekst echt lang is zal dit geen realistische aanname zijn. De tekst zal dan in een file staan. In dat geval kunnen de algoritmes op eenvoudige wijze aangepast worden. Er is echter een belangrijk verschil tussen het array en een file als invoer. In een array is het makkelijk om een aantal letters terug te kijken, terwijl dit niet altijd kan in een file.

1. Een Standaard Algoritme

Wanneer we het zoeken naar het patroon op voor de hand liggende wijze op proberen te lossen leggen we het patroon aan het begin langs de tekst en vergelijken we ze letter voor letter. Wanneer verschil optreedt schuiven we het patroon één letter verder en beginnen we opnieuw. Zo gezegd zo gedaan.

Algoritme : simpel patroon herkennen

```
Plek := 0;
BeginPatr := 1; PatPlek := 1; TekPlek := 1;
while PatPlek ≤ PatroonLengte and TekPlek ≤ TekstLengte
do
     if Patr[PatPlek] = Tekst[TekPlek]
     then PatPlek := PatPlek+1;
          TekPlek := TekPlek+1
     else BeginPatr := BeginPatr+1;
          PatPlek := 1;
          TekPlek := BeginPatr
     fi
od;
if PatPlek > PatroonLengte
then Plek := BeginPatr
fi;
```

Voorbeeld. We zoeken Patr = 'ABCABABC' in een tekst 'ABABCABCABABCC..'.

```
tekst   ABA..    ABA..     ABABCABCA..   ABABCABCA..          ..CABCABABCC..
        ↑↑×  ⇒   ×     ⇒   ↑↑↑↑↑×   ⇒   ×          ⇒ ... ⇒   ↑↑↑↑↑↑↑↑
patr    ABC..    ABC..     ABCABABC      ABCABABC               ABCABABC  ■
```

Een nadeel van dit algoritme is overigens dat soms in de tekst teruggegaan moet worden. Wanneer de tekst uit een file gelezen wordt betekent dit dat we een buffer met de laatst gelezen letters aan moeten houden.

2. Het Algoritme van Knuth-Morris-Pratt

Laten we in bovenstaand voorbeeld de situatie na twee verschuivingen nog eens bekijken. De eerste 5 letters van het patroon kloppen met de tekst, de zesde niet. We weten daarom dat het laatst bekeken stuk van de tekst eindigt met ...ABCABC× (waarbij × een willekeurige letter is). Nu schuiven we het patroon op. Het is vrij zinloos om slechts één letter op te schuiven. We weten onmiddellijk dat de eerste letters van patroon en tekst niet overeenstemmen. We zien dat we in één keer drie letters kunnen doorschuiven. Het begin van het patroon klopt dan met het eind van de bekeken tekst. Dit is niet afhankelijk van de tekst die we bekijken, maar alleen van het patroon. Steeds wanneer er een fout ontstaat bij de zesde letter hebben we een situatie als boven en mag het patroon drie letters opgeschoven worden. Dit leidt tot een verbeterd zoekalgoritme. Eénmalig bepalen we hoever we het patroon door mogen schuiven bij een fout op een bepaalde plek. Deze informatie slaan we op en gebruiken we bij het zoeken.

Stel dat bij de k-de letter van het patroon de tekst niet overeenstemt met het patroon. De tekst eindigt dan op $P_1 \ldots P_{k-1}\times$. We zoeken een waarde r zodat we de r-de letter van het patroon kunnen gaan vergelijken met de letter van de tekst waar we gebleven waren. De voorgaande letters van het patroon moeten dan kloppen met die uit de tekst. We moeten dus eisen dat $P_1 \ldots P_{r-1} = P_{k-r+1} \ldots P_{k-1}$. Omdat we niet te ver mogen doorschuiven (we zouden dan een stuk tekst met het patroon erin kunnen overslaan) moeten we r zo groot mogelijk kiezen (de verschuiving is dan zo klein mogelijk). We komen daarmee op het volgende begrip. De <u>failure-link</u> behorend bij de k-de letter uit een patroon P is de maximale r (kleiner dan k) zodat $P_1 \ldots P_{r-1} = P_{k-r+1} \ldots P_{k-1}$. Voor k = 1 stellen we de failure-link gelijk aan 0.

Opmerking. Omdat de tekst niet overeenstemt met het patroon op plek k, zouden we nog kunnen eisen dat P_r ongelijk is aan P_k. Immers, nu komt het soms voor dat we twee maal achter elkaar dezelfde letter met de × uit de tekst vergelijken. We doen dit niet. Het algoritme om de failure-links te berekenen (dat we verderop geven) blijft daarmee wat doorzichtiger. ■

Voorbeeld. We bepalen de failure-links van het eerder gegeven patroon. Als voorbeeld doen we dit uitgebreid voor de link voor de achtste positie.

```
ABCABAB..    ABCABAB..    ABCABAB..    ABCABAB..    ABCABAB..
 ABCABAB      ABCABAB      ABCABAB      ABCABAB      ABCABAB
```

Het is duidelijk dat het langste beginstuk van het patroon dat overeenkomt met het laatste stuk vóór de achtste positie uit twee letters bestaat. De failure-link van 8 wijst daarmee naar 3: wanneer de achtste letter niet correspondeert met de letter uit de tekst, wordt het patroon opgeschoven zodat de derde letter van het patroon onder de huidige letter in de tekst komt te liggen.

Voor de verschillende posities krijgen we de volgende plaatjes.

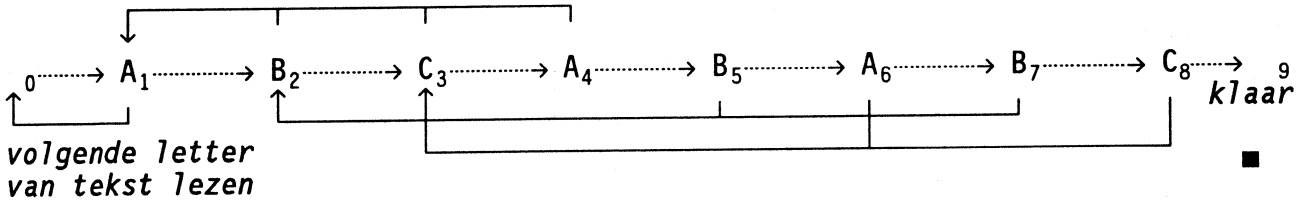
```
tekst    ..×..     ..A×..     ..AB×..     ..ABC×..      ..ABCA×..
vóór       A..       AB..       ABC..       ABCA..        ABCAB..
na          A..       AB..       ABC..       ABCA..        ABCAB..

tekst    ..ABCAB×..     ..ABCABA×..     ..ABCABAB×..
vóór       ABCABA..       ABCABAB..       ABCABABC
na            ABCABA..       ABCABAB..       ABCABABC
```

De failure-links worden hiermee:

k	1	2	3	4	5	6	7	8
Patr[k]	A	B	C	A	B	A	B	C
FLink[k]	0	1	1	1	2	3	2	3

In een plaatje weergegeven:



Wanneer de failure-links eenmaal berekend zijn kunnen we ze voor het zoekalgoritme gebruiken. We vergelijken steeds een letter van het patroon met een letter van de tekst. Zolang ze overeenkomen nemen we van beide telkens de volgende letter. Verschillen ze dan vergelijken we dezelfde letter van de tekst nogmaals met een letter van het patroon, nl. met de letter die door de failure-link aangegeven wordt. Heeft de failure-link de waarde 0 dan moeten we de volgende letter van de tekst met de eerste van het patroon vergelijken.

We nemen aan dat failure-links opgeslagen zijn in het array FLink en het patroon in het array Patr.

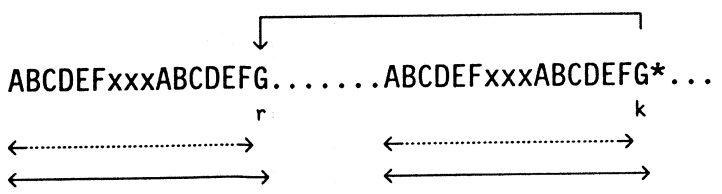
Algoritme : zoeken mbv. failure-links (Knuth-Morris-Pratt)

```
Plek := 0;
PatPlek := 1; TekPlek := 1;
while PatPlek ≤ PatroonLengte and TekPlek ≤ TekstLengte
do
     if Patr[PatPlek] = Tekst[TekPlek]
     then PatPlek := PatPlek+1;
          TekPlek := TekPlek+1
     else PatPlek := FLink[PatPlek];
          if PatPlek = 0          { Aan begin óók tekst doorschuiven. }
          then PatPlek := 1;
               TekPlek := TekPlek+1
          fi
     fi
od;
if PatPlek > PatroonLengte
then Plek := TekPlek-PatroonLengte; { Als Plek = 0 dan niet gevonden. }
fi;
```

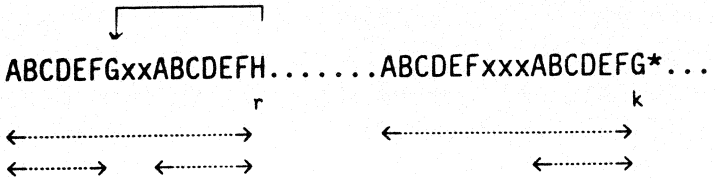
Voorbeeld. We zoeken met de KMP-methode naar het eerder gegeven patroon.

```
tekst   ABABC..        ABABCABCABABCC..        ABABCABCABABCC..
        ↑↑×      ⇒     ↑↑↑↑↑×            ⇒          ↑↑↑↑↑↑
patr    ABC..          ABCABABC..                   ABCABABC
            FLink[3] = 1          FLink[6] = 3
                                  (zelfde plek verder gaan met derde letter) ■
```

Het berekenen van de failure-links. Voor de failure-link r van de k-de letter moet gelden dat $P_1 \ldots P_{r-1} = P_{k-r+1} \ldots P_{k-1}$. Als bovendien $P_r = P_k$, dan hebben we $P_1 \ldots P_r = P_{k-r+1} \ldots P_k$. Deze laatste gelijkheid is nodig voor de failure-link van de (k+1)-ste letter. Zo blijkt dat, als $P_r = P_k$ dan FLink[k+1] = FLink[k]+1.



Wanneer P_r en P_k verschillend zijn blijven we de failure-links volgen totdat we bij een letter aangekomen zijn die wel gelijk is aan P_k. De failure-link bij k+1 is dan één groter dan de laatst gevonden waarde. Indien we geen letter vinden die gelijk is aan P_k dan wordt de failure-link bij k+1 gelijk aan 1.



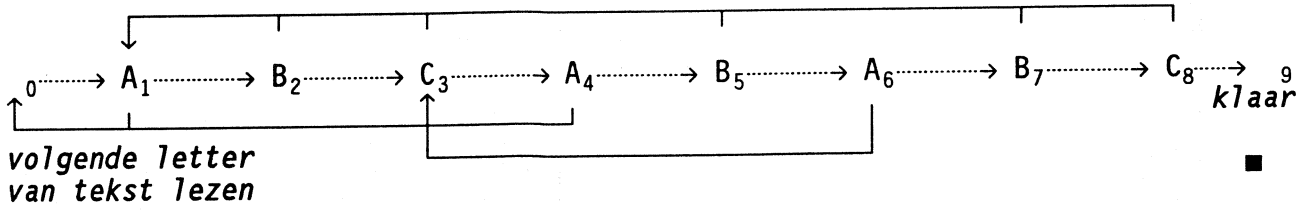
Algoritme : berekenen *failure-links* voor Knuth-Morris-Pratt

```
FLink[1] := 0;
for PatPlek := 2 to PatroonLengte
do   Fail := FLink[PatPlek-1];
     while Fail > 0 and Patr[Fail] ≠ Patr[PatPlek-1]
     do   Fail := FLink[Fail]                { luie evaluatie van and }
     od;
     FLink[PatPlek] := Fail+1
od;
```

Zoals boven gezegd, kan het voorkomen dat tweemaal achter elkaar dezelfde letter vergeleken wordt. We kunnen dit verhelpen door de zojuist opgestelde failure-links naar voren door te trekken. Loopt een failure-link van letter P_k naar letter P_r (FLink[k] = r) terwijl $P_k = P_r$ dan zal het algoritme tevergeefs P_r met de tekst vergelijken en vervolgens het patroon weer doorschuiven. We kunnen ook vooraf de failure-link van P_k gelijk maken aan de failure-link van P_r. Dit extra doorschuiven zit dan al in de failure-links verwerkt.

Voorbeeld. (vervolg) De failure-link van 4 wijst naar 1, terwijl de eerst en vierde letter van het patroon beide A zijn. Maak dus FLink[4] gelijk aan FLink[1] = 0. Gelijksoortige situaties doen zich voor bij FLink[5] en FLink[7] die beide naar 2 wijzen. De letters op 2, 5 en 7 zijn alle B. Maak daarom FLink[5] en FLink[7] gelijk aan FLink[2] = 1. Ook FLink[8] kan aangepast worden.

Na doorschuiven ziet het diagram er als volgt uit:



■

3. Patroon Herkennen met Eindige Automaten

Bron: Boek van Sedgewick, hoofdstuk 20 en 21. Dit onderwerp wordt ook vaak opgenomen in teksten over Compilerconstructie, bv. Aho, Sethi, Ullman: Compilers, Principles, Techniques, and Tools.

Aanwijzing: de structuur van het programma dat we gaan schrijven lijkt erg op dat van de code-generator van het college Algoritmiek. Kijk dat eventueel nog even na.

Inleiding. De zoekmethode die we hebben leren kennen is misschien wel efficiënt (hoewel het nog wel beter kan) maar weinig flexibel. We kunnen met het algoritme van KMP zoeken naar één enkel woord, met enige inspiratie ook nog wel naar een paar woorden tegelijk (zie opgaven), maar niet naar woord-patronen. Hoe zoek je bijvoorbeeld in een lijst naar alle woorden die beginnen met een B en twee A's bevatten?

In deze paragraaf geven we een mogelijke oplossing voor dit probleem. Er blijken heel wat kanten aan te zitten. Allereerst moeten we een notatie afspreken waarin we de woord-patronen aan de computer opgeven. Dan moet er een geschikte datastructuur gevonden worden om het patroon te representeren (en waarmee het zoeken vereenvoudigd wordt) en een methode om het opgegeven patroon te ontleden en naar de datastructuur om te zetten. Tenslotte moeten we uitwerken hoe we met behulp van de representatie van het patroon in een tekst gaan zoeken naar woorden die aan het patroon voldoen. Van dit laatste moeten we natuurlijk al enig idee hebben op het moment dat we de keuze voor de datastructuur maken.

Specificatie van de Woord-Patronen. De notatie waarin we woord-patronen zullen beschrijven is die van de reguliere expressies. Een voorbeeld van zo'n expressie is "**C(A+BB)*C**" (we moeten nog leren wat dit betekent). Expressies bevatten (hoofd-)letters en speciale hulpsymbolen: de haakjes, + en *. We kunnen dit qua vorm vergelijken met rekenkundige expressies, maar het gebruik en de betekenis van de + en * verschilt sterk. Als ingrediënten van onze reguliere expressies kiezen we de volgende operaties:

— Keuze, weergegeven door het symbool +. De expressie "**A+BB**" specificeert de string "**A**" of de string "**BB**". In de tekst mag elk van deze naar keuze voorkomen.

— Herhaling, weergegeven door * àchter de uitdrukking. De expressie "**B***" geeft een willekeurig aantal (nul, één of meer) B's achter elkaar aan. De expressie "**(A+BB)***" geeft aan dat we herhaald ofwel een A ofwel twee B's mogen schrijven. Hieraan voldoen bijvoorbeeld "**ABBAABB**" en "**AAA**", maar niet "**ABA**" en "**BBB**". Let op het gebruik van de haakjes: "**BBB**" voldoet wél aan de

expressie "**A+BB***" (één A of tenminste één B).

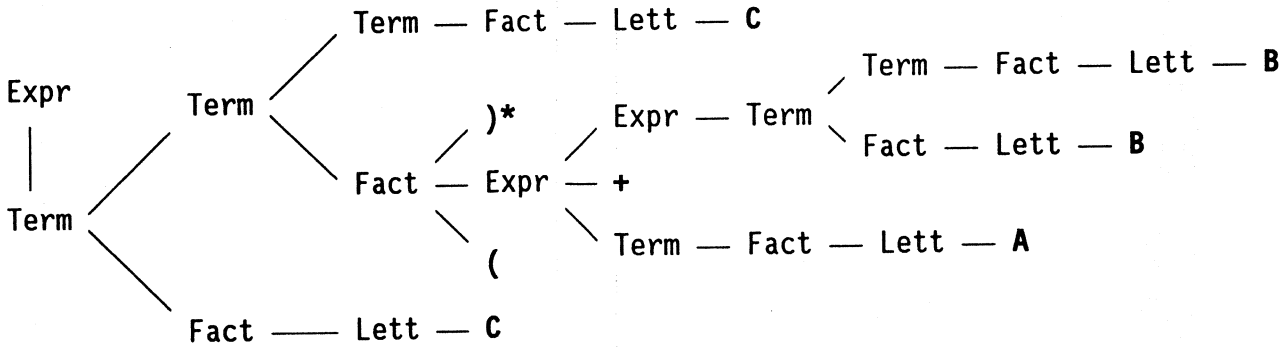
— Concatenatie, dwz. het achter elkaar schrijven van expressies. Zo geeft "**C(A+BB)*C**" aan dat we eerst een C willen zien, daarna herhaald een A of twee B's en tenslotte weer een C. Eigenlijk is de expressie "**BB**" zelf ook de concatenatie van twee losse B's.

Net als voor rekenkundige expressies gedaan werd bij algoritmiek, kan ook voor reguliere expressies een formele beschrijving gegeven worden met een grammatica in BNF:

```
Expr  ::=  Term  |  Term + Expr
Term  ::=  Fact  |  Fact Term
Fact  ::=  Lett  |  (Expr)  |  Lett*  |  (Expr)*
Lett  ::=  A  |  B  |  C  ...  Y  |  Z
```

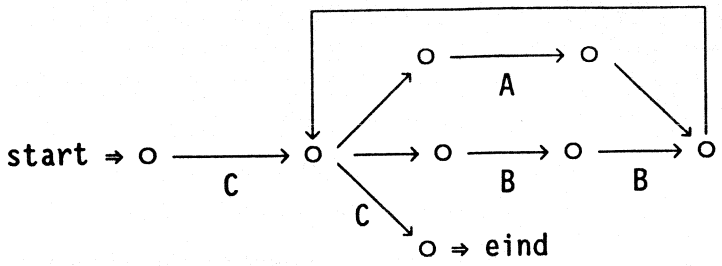
Dus, een expressie is de keuze tussen één of meer termen; een term is de concatenatie van één of meer factoren; een factor is een letter of een expressie tussen haakjes, waarbij beide gevolgd mogen worden door de herhalings-ster.

Met de gegeven regels kan elke reguliere expressie ontleed worden, maw. we kunnen er een *parse-tree* (ook wel *afleidingsboom*) voor opstellen. (In het plaatje hieronder is zo'n boom gegeven voor "**C(A+BB)*C**". Om ruimte uit te sparen staat de boom een beetje gekanteld: boven en onder corresponderen met rechts en links.) De methode die bij Algoritmiek is gebruikt om aan de hand van zulk soort regels een rekenkundige expressie om te zetten naar een boom-representatie zal ons ook hier goede diensten bewijzen.



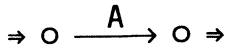
Representatie van het Patroon. Grafen met letters langs de takken kunnen goed gebruikt worden om de betekenis van de reguliere expressies weer te geven. Dit komt nog uitgebreid aan de orde bij het college Formele Talen en Automaten; we nemen wel alvast de naam van zulke grafen over en noemen ze eindige automaten. Bij de graaf (automaat) bekijken we wandelingen van een speciaal aangegeven startknoop naar een eindknoop en lezen we de letters langs de takken af. (Voor het gemak wordt op deze plek ook toegelaten dat sommige

takken geen letters dragen.) De woorden die op deze manier gevonden worden, moeten dan precies de woorden zijn die door de reguliere expressie beschreven worden.

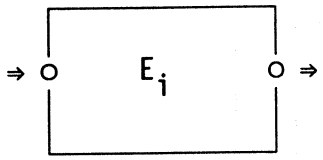


Op dit moment leggen we nog niet vast hoe we automaten precies in termen van enkelvoudige Pascal datastructuren zullen representeren. De automaten worden opgevat als een *abstracte* datastructuur, waarvoor eerst de noodzakelijke operaties bepaald worden.

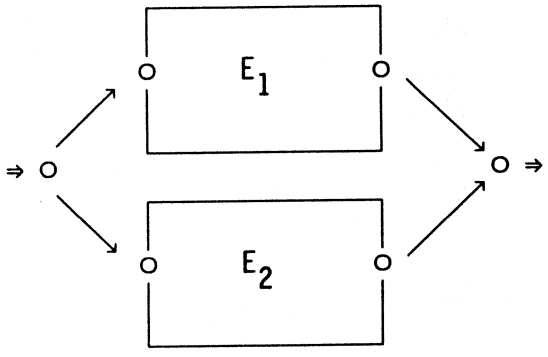
De gewenste operaties worden duidelijk door te onderzoeken hoe we bij elke expressie een automaat kunnen bepalen. Voor een simpele expressie als "**A**" is dat eenvoudig: de automaat bestaat uit twee knopen waartussen een tak met de letter A loopt. Stel nu dat voor de expressies E_1 en E_2 reeds automaten gevonden zijn. Daaruit kan een automaat voor de met keuze samengestelde expressie E_1+E_2 geconstrueerd worden door een nieuwe startknoop te nemen van waaruit takken gelegd worden naar de startknopen van de twee automaten; iets soortgelijks wordt voor de eindknopen gedaan. De automaat voor de concatenatie E_1E_2 ontstaat door de twee automaten achter elkaar te leggen met een verbindingstak. De automaat voor de herhaling $(E_1)^*$ tenslotte kan gemaakt worden door een tak zonder letter van de eindknoop naar de startknoop te leggen zodat we weer opnieuw kunnen beginnen. Er moet voor gezorgd worden dat de automaat ook nul maal doorlopen kan worden, dit kan gebeuren door te beginnen in de oorspronkelijke eindknoop.



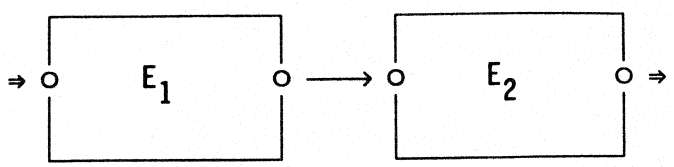
MaakTak('A')



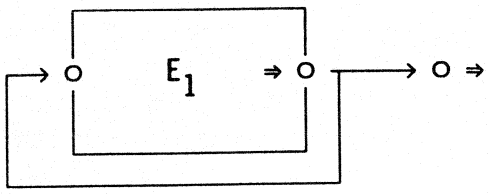
Aut*i* *i*=1,2



Plus(Aut1,Aut2)



Conc(Aut1,Aut2)



Ster(Aut1)

Voor onze abstracte datastructuur Automaat nemen we aan dat de volgende functies beschikbaar zijn die de geschetste constructies uitvoeren.

```
function MaakTak( K : char ): Automaat;
function Plus( Aut1, Aut2 : Automaat ): Automaat;
function Conc( Aut1, Aut2 : Automaat ): Automaat;
function Ster( Aut1 : Automaat ): Automaat;
```

Omzetten van Expressie naar Automaat. De omzetting van expressie naar automaat gebeurt met elkaar recursief aanroepende functies die nauwgezet de grammaticale beschrijving volgen. In tegenstelling tot de parser voor rekenkundige expressies (zoals bij Algoritmiek gegeven) wordt hier niet eerst een boomstructuur opgebouwd die daarna doorlopen wordt. De boomstructuur is alleen in gedachten aanwezig om de werking van de functies te kunnen beschrijven en wordt niet expliciet geconstrueerd.

Bij de hieronder gegeven functies geeft HuidigeLetter de gelezen letter van het ingevoerde woord-patroon aan. Er wordt niet nauwkeurig omschreven wat er moet gebeuren als we voorbij de laatste letter van het ingevoerde patroon lezen. Dit moet bij latere verfijningen van de functies gebeuren.

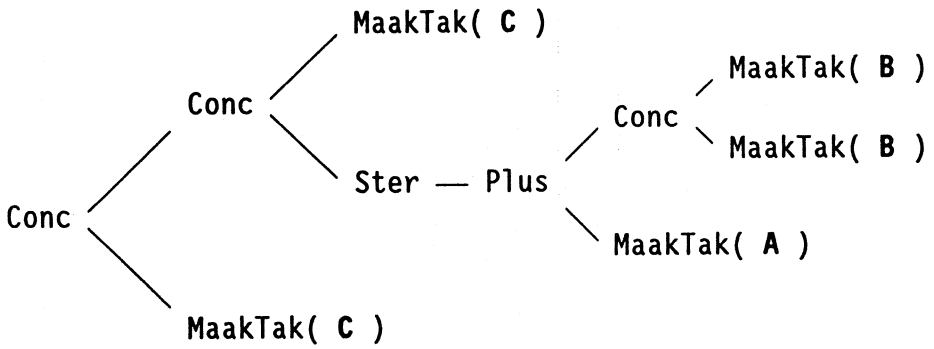
Let op dat in Pascal de functies Term en Fact al in Expr bekend moeten zijn, hetzij door ze lokaal in Expr te zetten (zoals bij de code-generator), hetzij door ze **FORWARD** te declareren. Bovendien hebben we het type Automaat nog niet precies vastgelegd; waarschijnlijk wordt dit geen enkelvoudig Pascal-type zodat de functies later omgezet moeten worden in procedures (met een **VAR** parameter voor de uitvoer).

```
function Expr : Automaat;
     Aut1 := Term;                          { Eerste term wordt het   }
     Expr := Aut1;                          { voorlopige resultaat.   }
     if HuidigeLetter = '+'                 { Als er een + volgt dan  }
     then Lees volgende letter;
          Expr := Plus( Aut1, Expr );       { recursieve aanroep Expr.}
     fi;
endfunction; { Expr }

function Term : Automaat;
     Aut1 := Fact;                          { Eerste factor wordt het }
     Term := Aut1;                          { voorlopige resultaat en }
     if HuidigeLetter ∈ ['(', 'A'..'Z']     { met eventuele volgende  }
     then Term := Conc( Aut1, Term );       { term geconcateneerd.    }
     fi;
endfunction; { Term }
```

```
function Fact : Automaat;
     if HuidigeLetter = '('                    { Bij een haakje          }
     then Lees volgende letter;
          Aut1 := Expr;                        { voor een expressie      }
          if HuidigeLetter = ')'
          then Lees volgende letter;           { hoort een sluithaakje.  }
          else Error
          fi
     else if HuidigeLetter ∈ ['A'..'Z']
     then Aut1 := MaakTak( HuidigeLetter );    { Een enkele letter.      }
          Lees volgende letter
     else Error
     fi;
     if HuidigeLetter ≠ '*'                    { Volgt een ster dan moet }
     then Fact := Aut1
     else Fact := Ster( Aut1 );                { de automaat aangepast.  }
          Lees volgende letter;
     fi;
endfunction; { Fact }
```

Voor ons voorbeeld "C(A+BB)*C" kunnen we de te construeren automaat weergeven mbv. een boomstructuur, de zgn. *semantische boom*, die afgeleid kan worden uit de eerder gegeven *parse-tree*. Dit is niet verwonderlijk: we hebben bij het opstellen van onze procedures nauwkeurig de BNF-grammatica gevolgd. Let eens op de nauwkeurige overeenkomst tussen de boomstructuur, de manier waarop de automaten worden geconstrueerd en de postfixnotatie "CABB•+*C••" van de uitdrukking "C(A+BB)*C" (waarin we • voor concatenatie gebruiken).



Implementatie van de Abstracte Datastructuur Automaat. Bij nadere inspectie van de boven geschetste grafen blijkt dat er slechts twee soorten knopen ontstaan: knopen met één uitgaande tak (van een letter voorzien door MaakTak, of zonder letter: de tak tussen de automaten bij Conc) en knopen die twee uitgaande takken hebben zonder letter (de eerste knoop van de grafen die ontstaan bij Plus en Ster). Een mogelijke implementatie is de volgende, die erg veel lijkt op de methode om binaire bomen in een array op te bergen. Voor elke knoop reserveren we drie velden: één om de eventuele letter in op te bergen, en twee voor de eventuele opvolgerknopen. We moeten nog afspreken wat de 'NIL-waarden' zijn, dwz. hoe we onderscheid maken tussen knopen met nul, één of twee uitgaande takken en hoe we takken aangeven die geen letter hebben. Wij kiezen hier voor de speciale waarden 0 en #0.

Zoals bij binaire bomen de boom vastgelegd wordt door het geven van de wortel als toegangsknoop, wordt een automaat vastgelegd door het geven van de start- en eindknoop.

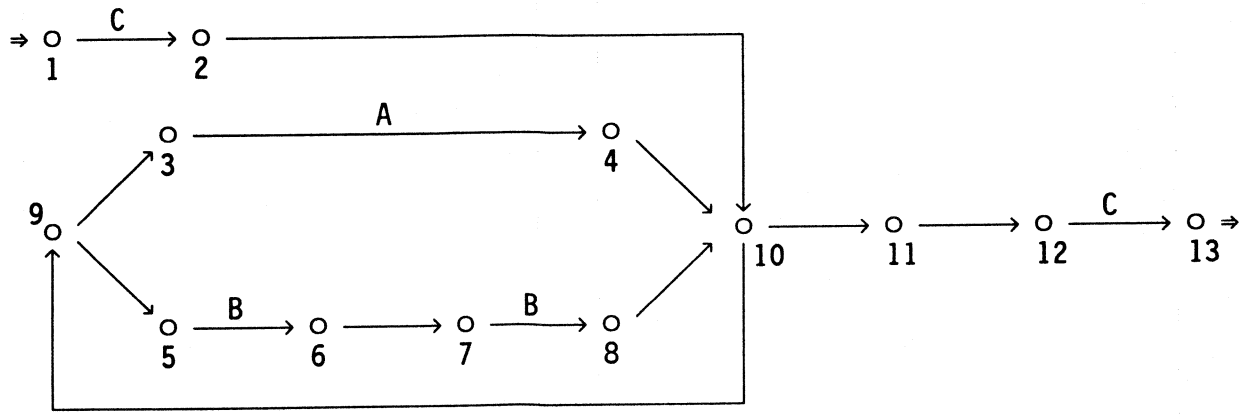
```
TYPE
     KnoopTal  = 0..255;
     KnoopTiep = RECORD
                    Kar   : CHAR;
                    Volg1,
                    Volg2 : KnoopTal
                 END;
     Automaat  = RECORD
                    Start,
                    Eind  : KnoopTal
                 END;
VAR
     Knoop : ARRAY[KnoopTal] OF KnoopTiep;
```

Het is nu een eenvoudige opgave om te abstracte functies MaakTak, Conc, Plus en Ster in Turbo-Pascal uit te werken tot procedures (in plaats van functies) in een Unit die de datastructuur Automaat implementeert.

Ook het ontledings-algoritme is elders uitgewerkt. Bij een invoer "C(A+BB)*C" geeft de Pascal versie van Expr de volgende automaat als uitvoer.

KnoopNr	⇒1	2	3	4	5	6	7	8	9	10	11	12	13⇒
Kar	C		A		B		B					C	
Volg1	2	10	4	10	6	7	8	10	3	9	12	13	-
Volg2	-	-	-	-	-	-	-	-	5	11	-	-	-

Dit is niet helemaal de automaat die we oorspronkelijk opgesteld hebben, maar één die dezelfde woorden specificeert. Het blijkt dat de gebruikte methode om termen achter elkaar te plakken nogal inefficiënt is wanneer een letter op een term volgt: de extra tak zonder letter is dan niet nodig. Deze methode is echter de eenvoudigste manier om twee gehaakte expressies achter elkaar te plakken, zoals in "(A+B)(AB+BA)".



Zoeken naar het opgegeven patroon. Wanneer het patroon omgezet is in een eindige automaat begint de volgende fase van het probleem: van een gegeven woord moet (met behulp van de automaat) nagegaan worden of dit woord aan het patroon voldoet. In termen van de automaat betekent dit dat er een pad in de automaat bestaat, van de startknoop naar de eindknoop, waarlangs de letters het gegeven woord vormen. (Dit probleem lijkt misschien op het eerste gezicht erg simpel. We moeten niet vergeten dat er ook takken zonder letter in de graaf voorkomen die het aantal mogelijke paden sterk vergroten.)

Een eindige automaat is een speciaal soort graaf. De eerder beschreven methode om door een graaf te wandelen kan aangepast worden om te bepalen of door de graaf een wandeling met een gegeven woord bestaat. We gaan hierbij *letter voor letter* te werk.

Te beginnen in de startknoop volgen we takken zonder letter op zoek naar takken met de eerste letter van het woord. De eindknopen van de gevonden takken (als die er zijn) vormen het startpunt van een zelfde zoekactie voor de tweede letter. Zo wordt vervolgd tot de laatste letter van het gegeven woord behandeld is. Dan dienen we nog te controleren of de eindknoop bereikbaar is via takken zonder letter.

Een Pascal versie van dit algoritme is elders in dit dictaat opgenomen. Er wordt daarin in principe gebruik gemaakt van *depth-first search* met behulp van een stapel.

Voorbeeld. Bij een patroon gegeven door de expressie "A*(AA+AB)*ABB*" wordt de volgende automaat geconstrueerd (start- en eindknoop zijn respectievelijk 2 en 21): — zie volgende bladzijde.

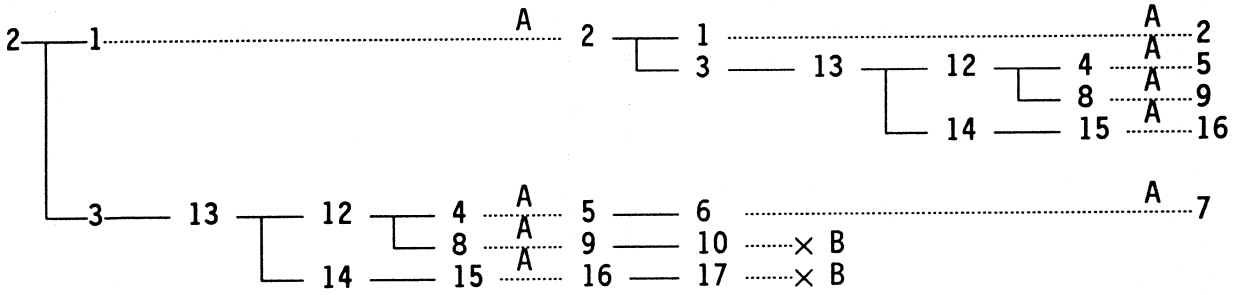
We testen het woord "AABABB", dat aan de expressie blijkt te voldoen.

Expressie "A*(AA+AB)*ABB*" , patroon AABABB

KnoopNr	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Kar	A	.	.	A	.	A	.	A	.	B	.	.	.	.	A	.	B	.	B	.	.
Volg1	2	1	13	5	6	7	13	9	10	11	13	4	12	15	16	17	18	20	20	19	-
Volg2	-	3	-	-	-	-	-	-	-	-	-	8	14	-	-	-	-	-	-	21	-

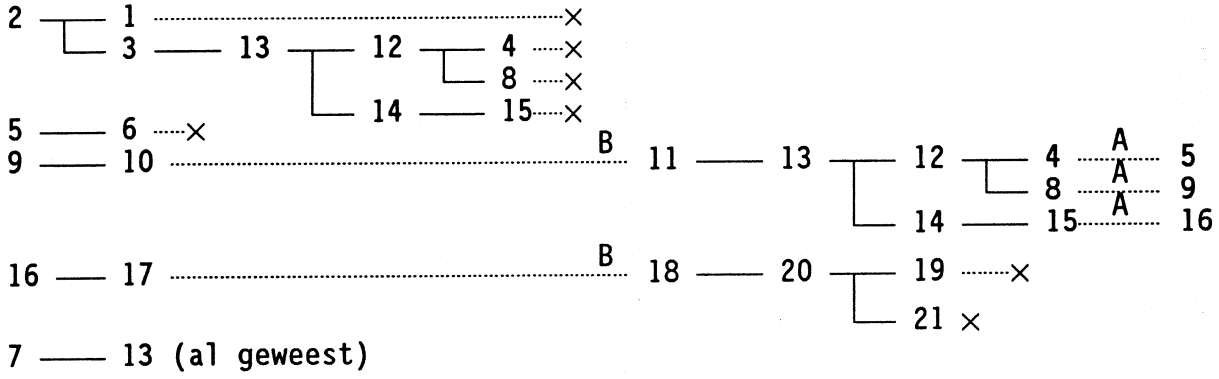
1 Letter A. Start in 2. Loop mbv. een *depth-first* wandeling.
Via kale takken bereikbaar 1, 3, 13, 12, 4, 8, 14, 15 (en eigenlijk 2 ook, want daar beginnen we). De onderstreepte knopen zijn beginknoop van een tak met letter A. Deze leiden naar respectievelijk 2, 5, 9 en 16.

2 Letter A. Start in 2, 5, 9, 16. Via kale takken bereikbaar (weer) 1, 3, 13, 12, 4, 8, 14, 15, (ook) 6, 10, 17.



3 Letter B. Start in 2, 5, 9, 16, 7. Weer via kale takken bereikbaar 1, 3, 13, 12, 4, 8, 14, 15, 6, 10, 17. De onderstreepte knopen corresponderen nu met B's. Takken leiden naar 11 en 18.

4 Letter A. Start in 11, 18. Via kale takken 13, 12, 4, 8, 14, 15, 20, 19, 21.



7 13 (al geweest)

5 Letter B. Start in 5, 9, 16. Via kale takken naar 6, 10, 17.

6 Letter B. Start in 11, 18. Net als in stap 4 via kale takken naar 13, 12, 4, 8, 14, 15, 20, 19, 21.

7 Laatste stap. Start in 20, bereikbaar zijn (behalve 20) 19, 21 (eindknoop).

VI. LIJSTEN

In dit hoofdstuk zullen we een zeer algemene datastructuur bekijken, de lijst. Hiermee bedoelen we dan een veel uitgebreider begrip dan de bekende *lineaire* lijsten zoals de rij (*queue*) en de stapel (*stack*). Er wordt in dit hoofdstuk veelvuldig teruggegrepen naar de kennis die we bij bomen en grafen opgedaan hebben. In zekere zin vormen lijsten een uitbreiding van bomen (vooral wat betreft de representatie) en van grafen (wat betreft structuur).

1. Definitie en Representatie

Wat informeel gezegd is een lijst $A = (x_1,\ldots x_n)$ een rij objecten, waarbij elk object x_i ofwel zelf weer een lijst is, of een atoom (dus 'ondeelbaar'). n is de lengte van de lijst A. De objecten heten de elementen van de lijst, niet-atomaire elementen heten ook wel deellijsten. We laten hierbij bovendien recursief gedefinieerde lijsten toe, dwz. lijsten die (via-via) zichzelf bevatten.

Voorbeeld. $A_1 = (a,b,c)$ is een (lineaire) lijst met drie atomaire elementen.

$A_2 = ((a,b),c,A_1,(a,b),())$ is een lijst met vijf elementen, waarvan alleen de tweede atomair is. De lijst (a,b) komt tweemaal als element van A_2 voor. Het laatste element van A_2 is een lijst zonder elementen. Dit is de lege lijst (), deze heeft lengte nul.

$A_3 = (a,A_4)$, $A_4 = ((a,b),A_3,a,A_3)$ definieert een 'echte' recursieve lijst, waarbij elk van de lijsten de ander bevat.

$A_5 = (a,A_5)$ is een lijst van lengte twee. De lijst kan ook gezien worden als de oneindige lijst (a,(a,(a,...))). ■

Op lijsten worden twee operaties - Head en Tail - toegelaten. Voor een lijst $A = (x_1,\ldots,x_n)$ levert Head(A) het eerste element x_1, en Tail(A) de lijst $(x_2,\ldots,x_n)$ die uit de overige elementen van A bestaat. Let dus op: Tail(A) is altijd een lijst, terwijl Head(A) ook een atoom kan zijn (en zelfs ongedefinieerd als A de lege lijst is).

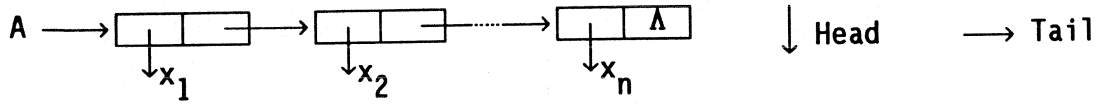
Vervolg voorbeeld. $\text{Head}(A_1) = a$ is atomair; $\text{Tail}(A_1) = (b,c)$ is een (lineaire) lijst met twee elementen.

$\text{Head}(A_2) = (a,b)$ is een lijst van lengte twee.

$\text{Head}(A_3) = a$, $\text{Tail}(A_3) = (A_4)$, $\text{Head}(A_4) = (a,b)$ en $\text{Tail}(A_4) = (A_3,a,A_3)$. Dus $\text{Head}(\text{Tail}(A_4)) = \text{Head}((A_3,a,A_3)) = A_3$.

$\text{Head}(A_5) = a$, terwijl $\text{Tail}(A_5)$ gelijk is aan de lijst (A_5). ■

Deze twee operaties zijn niet alleen geschikt om op een abstracte manier algoritmes op lijsten te beschrijven, ze vormen ook de basis van de meest gangbare representatie van lijsten. Een lijst A = $(x_1,x_2,\ldots,x_n)$ wordt weergegeven door een lineaire lijst die pointers bevat naar de elementen $x_1,\ldots,x_n$. Elk van die elementen is dan of zelf weer een lineaire lijst, of is een (atomaire) knoop met informatie. Wanneer we een pointer van de lijststructuur volgen, vinden we de lijst die overeenkomt met de Tail van de oorspronkelijke lijst. De pointers die opgeslagen zijn in de lijst vormen de adressen van de elementen van de lijst, steeds overeenkomend met een Head van een Tail-lijst.

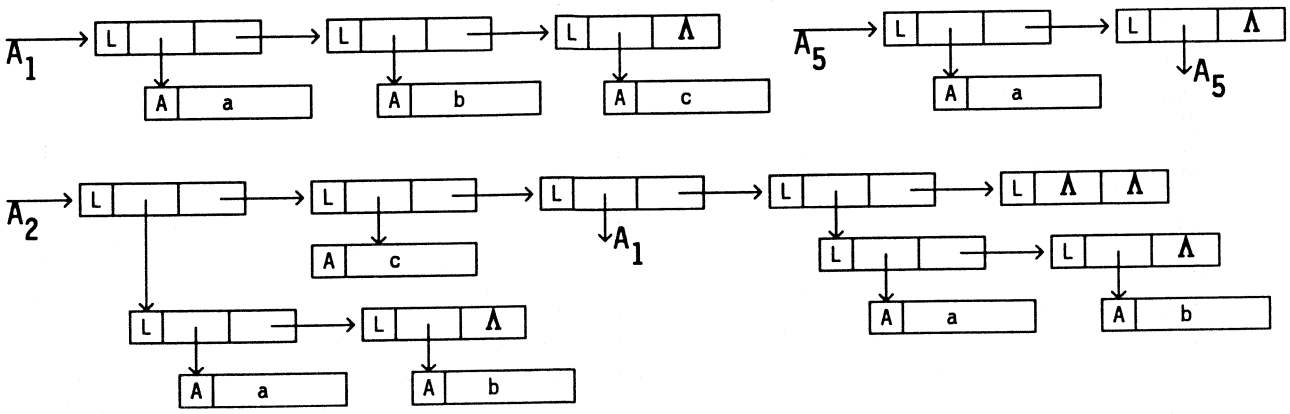


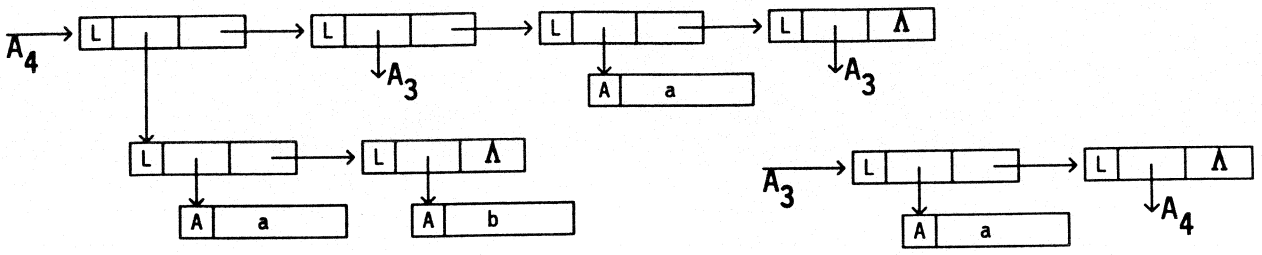
De representatie van lijsten heeft dan ook twee soorten elementaire bouwstenen: de gewone knopen, die twee pointers bevatten, en de atomaire, die informatie (van een van te voren vastgelegd type) bevatten.

```
TYPE KnoopType = (Atoom,Lijst);
     LijstWijzer = ↑LijstElt;
     LijstElt = RECORD CASE Tiep: KnoopType
                      OF   Atoom: (Info: AtoomType);     { zelf kiezen }
                           Lijst: (Head, Tail: LijstWijzer);
                END;
```

Opmerking. Om onopgehelderde redenen dient er in (Turbo) Pascal bij een *variant record* slechts één **END** gebruikt te worden, dat dan zowel voor de **RECORD** als voor de **CASE** dienst moet doen.

De lijsten uit het voorbeeld zien er in deze representatie dus als volgt uit. Merk op dat Λ (NIL) gebruikt wordt om een pointer naar de lege lijst weer te geven.





Het is tamelijk waarschijnlijk dat de Pascal compiler voor het type LijstElt de maximaal benodigde geheugenruimte reserveert. In het bovenstaande geval dus de ruimte benodigd voor een element van AtoomType of voor twee LijstWijzers. In het geval dat het AtoomType nogal veel ruimte inneemt (bijvoorbeeld een lange string) is dit niet erg efficient. Beter is dan de volgende variant van de representatie.

```
TYPE KnoopType = (Atoom,Lijst);
     AtoomWijzer = ↑AtoomType;
     LijstWijzer = ↑LijstElt;
     LijstElt = RECORD
                    Tail: LijstWijzer;
                    CASE Tiep: KnoopType
                    OF   Atoom: (AHead: AtoomWijzer);
                         Lijst: (LHead: LijstWijzer);
                END;
```

2. Lijstwandelingen

Anders dan bij grafen zijn de knopen van een lijst niet direct bereikbaar. Om de informatie uit de atomaire knopen te halen moeten we een wandeling uitvoeren. We kunnen dit op de klassieke manier met een stapel doen, maar we kunnen ook het linkomkeringsalgoritme voor bomen aanpassen voor lijsten. Net als bij grafen zorgen we dat we niet in rondjes blijven rondlopen door de bezochte knopen te markeren. We hebben dan een extra Boolese veld Bezocht in de declaratie van LijstElt nodig.

2.1. Met een Stapel

Wanneer we een *depth-first* wandeling uitvoeren volgen we eerst zo veel mogelijk de Head pointers. Onderwijl zetten we dan de Tail pointers op de stapel. Bij het eerste bezoek worden knopen gemarkeerd; ze worden niet nogmaals bezocht.

Algoritme : lijstwandeling mbv. stapel

```
                    { De functie Bezocht geeft de waarde van het Bezocht }
                    { veld, maar levert ook true bij een NIL pointer.     }
    MaakLeeg(S);
    S ⇐ BeginKnoop;
    while not Leeg(S)
    do   Deze ⇐ S;
         while not Bezocht(Deze)
         do   Deze↑.Bezocht := true;
              if Deze↑.Tiep = Atoom
              then Bezoek(Deze)
              else S ⇐ Deze↑.Tail;
                   Deze := Deze↑.Head
              fi
         od
    od;
```

2.2. LINKOMKERING

Ook het linkomkeringsalgoritme voor bomen is na enig nadenken aan te passen voor lijsten. Het voordeel van het algoritme is ook nu weer dat we geen extra datastructuur nodig hebben om de wandeling uit te voeren. Daartegen moeten we wel twee markeervelden in elke knoop tot onze beschikking hebben. Eén zoals hiervoor om de bezochte knopen aan te geven, het andere om het 'omgekeerde' pad naar de startknoop aan te geven. Voor de volledigheid volgt hier de type declaratie.

```
TYPE LijstElt = RECORD
                    Bezocht, Tag: Boolean;
                    CASE Tiep: KnoopType
                    OF   Atoom: (Info: AtoomType);
                         Lijst: (Head, Tail: LijstWijzer);
                END;
```

Het linkomkeringsalgoritme voor lijsten voert een wandeling uit zoals het algoritme voor bomen dat zou doen op de pre-orde (*depth-first*) opspannende boom. We volgen zoveel mogelijk de Head pointers (vroeger Links) totdat we een NIL pointer of een reeds bezochte knoop tegenkomen. We 'klimmen' dan weer omhoog in de opspannende boom via de knoop aangegeven door Vader. Net als bij het algoritme mbv. een stapel moeten we ook hier de atomaire knopen apart in

de gaten houden. Deze knopen hebben geen verdere pointers en vormen dus bladeren in de opspannende boom. Ze worden slechts éénmaal bezocht.

{← van links naar rechts uitvoeren}

Algoritme : Lijstwandeling met linkomkering (*Schorr-Waite-Deutsch*)

```
                    { De functie Bezocht geeft de waarde van het Bezocht }
                    { veld, maar levert ook true bij een NIL pointer.     }
    if StartKnoop ≠ NIL
    then Vader := NIL; Deze := StartKnoop;
         Bezoek := 1; Klaar := false;
         repeat case Bezoek of
              1:   Deze↑.Bezocht := true;                          { pre-orde }
                   if Deze↑.Tiep = Atoom
                   then BrengBezoekAan(Deze);
                        Bezoek := 3
                   else Volgend := Deze↑.Head;
                        if Bezocht(Volgend)
                        then Bezoek := 2
                        else Deze↑.Head ← Vader ← Deze ← Volgend
                        fi
                   fi;
              2:   Volgend := Deze↑.Tail;                        { symmetrisch }
                   if Bezocht(Volgend)
                   then Bezoek := 3
                   else Deze↑.Tag := 1;
                        Deze↑.Tail ← Vader ← Deze ← Volgend;
                        Bezoek := 1
                   fi;
              3:   if Vader ≠ NIL                                { post-orde }
                   then if Vader↑.Tag = 0
                        then Hulp ← Vader↑.Head ← Deze ← Vader ← Hulp;
                             Bezoek := 2
                        else Vader↑.Tag := 0;
                             Hulp ← Vader↑.Tail ← Deze ← Vader ← Hulp
                        fi
                   else Klaar := true;
                   fi
             endcase
         until Klaar;
    fi;
```

2.3. Varianten van de Lijstwandel-Algoritmes

```
                                        {← van links naar rechts uitvoeren}
Algoritme : Lijstwandeling met linkomkering en bit-stack (Wegbreit)
                    { De functie Bezocht geeft de waarde van het Bezocht }
                    { veld, maar levert ook true bij een NIL pointer.    }
                    { S is een stapel van bits, bevat benodigde Tag's.   }
    if StartKnoop ≠ NIL
    then Vader := NIL; Deze := StartKnoop;
         Bezoek := 1; Klaar := false;
         repeat case Bezoek of
              1:   Deze↑.Bezocht := true;                    { pre-orde }
                   if Deze↑.Tiep = Atoom
                   then BrengBezoekAan(Deze);
                        Bezoek := 3
                   else Volgend := Deze↑.Head;
                        if Bezocht(Volgend)
                        then Bezoek := 2
                        else S ⇐ 0;                                {#}
                             Deze↑.Head ← Vader ← Deze ← Volgend
                        fi
                   fi;
              2:   Volgend := Deze↑.Tail;                { symmetrisch }
                   if Bezocht(Volgend)
                   then Bezoek := 3
                   else S ⇐ 1;                                     {#}
                        Deze↑.Tail ← Vader ← Deze ← Volgend;
                        Bezoek := 1;
                   fi;
              3:   if Vader ≠ NIL                          { post-orde }
                   then Tag ⇐ S;                                   {#}
                        if Tag = 0
                        then Hulp ← Vader↑.Head ← Deze ← Vader ← Hulp;
                             Bezoek := 2
                        else Hulp ← Vader↑.Tail ← Deze ← Vader ← Hulp
                        fi
                   else Klaar := true
                   fi;
            endcase
         until Klaar;
    fi;
```

Bovenstaande variant gebruikt geen Tag bit voor de knopen van de lijst. Dit bit is alleen nodig voor knopen van de lijst die op het 'omgekeerde' pad van de knoop Vader naar de StartKnoop liggen. We slaan de waardes van deze Tag's niet bij de knoop op maar op een stapel. Voordeel boven het eerder gegeven algoritme dat van een stapel gebruik maakt is dat een stapel met bits minder plaats in neemt dan een stapel met pointers (adressen).

Bij de tot nu toe gegeven algoritmes hebben we steeds twee bits (Bezocht en Tag) per knoop nodig of slechts één bit (Bezocht) per knoop en een extra datastructuur (stapel of rij). Wanneer we wat meer tijd willen investeren in het controleren of een knoop al bezocht is, dan kunnen we dit tot één bit per knoop terugbrengen zonder dat we een stapel hoeven te gebruiken. We gebruiken dan de Bezocht velden van de knopen die liggen op het pad van Vader naar StartKnoop om aan te geven of dit pad via de Head of de Tail pointer loopt (dit gebeurde voorheen met het Tag veld).
Voor de betekenis van Bezocht spreken we het volgende af:

- **true:** Knoop is reeds bezocht,
  wanneer op pad naar StartKnoop dan pad via Head pointer.
- **false:** Knoop is nog niet bezocht,
  tenzij op pad naar StartKnoop dan bezocht en pad via Tail pointer.

De functie *Bezocht* moet dus aangepast worden en wordt gedefinieerd door *Bezocht*(X) ::= (X = NIL) **or** (X↑.Bezocht) **or** *OpPadNaarStartKnoop*(X), waarbij uitgegaan is van 'luie' evaluatie van **or**. De functie *OpPadNaarStartKnoop*(X) wordt uitgerekend door het volgende programmaatje.

```
Loper := Vader;
while (Loper ≠ NIL) and (Loper ≠ X)
do   if Loper↑.Bezocht
     then Loper := Loper↑.Head
     else Loper := Loper↑.Tail
     fi;
od;
OpPadNaarStartKnoop := (Loper = X);
```

In het algoritme voor lijstwandelen met link-omkering gebruiken we nu het Bezocht veld ipv. het Tag veld. De waarde van dit veld is achtereenvolgens:

- **false**, in het begin van het algoritme (niet bezocht, niet op pad),
- **true**, na het eerste bezoek (bezocht, pad via Head),
- **false**, na het tweede bezoek (bezocht!, pad via Tail), en
- **true**, uiteindelijk, na het derde bezoek (bezocht, niet op pad).

Pas, met deze aanwijzingen, het standaard linkomkerings-algoritme aan.

AANHANGSEL

A. Wiskunde

1. Notaties

Gebruikt worden de logaritmen $lg x$, $ln x$ en $log x$ voor de grondtallen 2, e en 10. Deze kunnen in elkaar omgerekend worden via de formule ${}^a log\ x = \frac{{}^b log\ x}{{}^b log\ a}$.

Laat $f : \mathbb{N} \to \mathbb{N}$ een functie zijn. Dan is $\mathcal{O}(f)$ de verzameling functies

$\mathcal{O}(f) = \{ g : \mathbb{N} \to \mathbb{N} \mid$ er zijn $c \geq 0$ en N zdd. $g(n) \leq c \cdot f(n)$ voor all $n \geq N \}$.

Het is gebruikelijk om bij het $\mathcal{O}$ symbool het gelijkheidssymbool te gebruiken in plaats van symbolen $\in$ en $\subseteq$. Dit maakt dat uitdrukkingen waarin de $\mathcal{O}$ notatie voorkomt alleen van links naar rechts gelezen mogen worden.
De operaties worden van functies uitgebreid tot verzamelingen functies. Zo bedoelt men voor verzamelingen A en B van functies met A + B de verzameling $\{ f+g \mid f \in A, g \in B \}$.

Kortom, voor de uitdrukking $3n + ln\ n = 3n + \mathcal{O}(lg\ n) = \mathcal{O}(n)$ gelieve men te lezen $3n + ln\ n \in \{ 3n \} + \mathcal{O}(lg\ n) \subseteq \mathcal{O}(n)$.

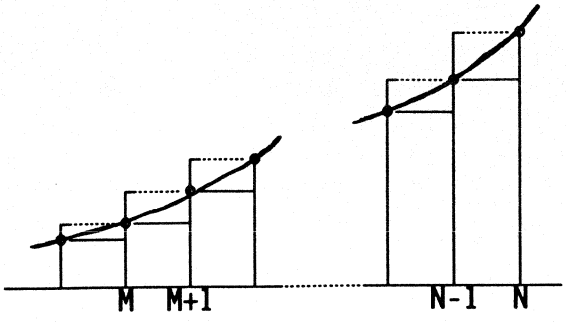
2. Afschatten Sommaties

Laat f een positieve, stijgende functie zijn. De integraal van f over een bepaald interval komt overeen met de oppervlakte onder de grafiek. Deze kan afgeschat worden mbv. rechthoeken. Zo geldt dan

$$\sum_{k=M}^{N-1} f(k) \leq \int_M^N f(x)\ dx \leq \sum_{k=M+1}^{N} f(k)$$

Omgekeerd geldt dus

$$f(M) + \int_M^N f(x)\ dx \leq \sum_{k=M}^{N} f(k) \leq \int_M^N f(x)\ dx + f(N)$$



Voorbeeld. Kies $f(x) = \ln x$. We weten $\int \ln x\, dx = x\cdot\ln x - x$. Volgens bovenstaande geldt (kijk uit voor $\ln 0$ bij de ondergrens):
$n\cdot\ln n - n + 1 = \int_1^n \ln x\, dx \leq \sum_{k=1}^n \ln k \leq \int_1^n \ln x\, dx + \ln n = (n+1)\cdot\ln n - n + 1$.
Hieruit volgt dat $\sum_{k=1}^n \ln k \in O(n\cdot\ln n)$. Het is opvallend dat deze afschatting ook met een veel grovere methode gevonden kan worden: $\sum_{k=1}^n \ln k \leq n\cdot\ln n$ omdat elk van de n termen ten hoogste $\ln n$ is.

Voor andere logaritmes gelden soortgelijke formules. De waarden voor de verschillende grondtallen verschillen immers slechts een constante.

3. Exacte Sommaties

i. $\sum_{k=0}^n k = (n+1)\cdot n$

ii. $\sum_{k=0}^n r^k = \frac{r^{n+1}-1}{r-1}$ $(r \neq 1)$

iii. $\sum_{k=0}^n k\cdot 2^k = (n-1)\cdot 2^{n+1} + 2$

Bewijs.

i. Inductie naar n. ii. Vermenigvuldig links en rechts met r-1.

iii. Bekijk $f(r) = \sum_{k=0}^n r^k$ als functie van r. Bereken van deze functie de afgeleide (naar r) op twee manieren, overeenkomend met de beide zijden van de gelijkheid in ii. Dan:

$f'(r) = \sum_{k=1}^n k\cdot r^{k-1} = \frac{d}{dr}\left(\frac{r^{n+1}-1}{r-1}\right) = \frac{(r-1)(n+1)r^n - r^{n+1} + 1}{(r-1)^2}$. Voor $r = 2$ geldt dus: $\sum_{k=0}^n k\cdot 2^k = 2\cdot\sum_{k=1}^n k\cdot 2^{k-1} = 2\cdot[\,(n-1)\cdot r^n + 1\,]$. ■

```
PROGRAM PATROON_HERKENNING (Input, Output);

{ DOEL     PATROON-HERKENNING MET EINDIGE AUTOMATEN volgens Sedgewick.     }
{          Nagegaan wordt of gegeven woord voldoet aan gegeven patroon.   }
{          Het patroon wordt gespecificeerd als zgn. reguliere expressie. }
{          De syntax van expressies wordt als volgt beschreven in BNF     }
{             Expr ::=  Term   |  Term + Expr                             }
{             Term ::=  Fact   |  Fact Term                               }
{             Fact ::=  Lett   |  ( Expr )  |  Lett *  |  ( Expr ) *      }
{             Lett ::=  A  |  B  ..  Y  |  Z                             }
{          Het patroon wordt na ontleden gerepresenteerd mbv eindige aut'n.}
{          Zie verder dictaat Datastructuren RULeiden.                    }
{ AUTEUR   HJH (vrij naar Sedgewick)                                      }
{ DATUM    13 feb 90 - inlezen & ontleden expressie                       }
{          20 apr 90 - simulatie automaat op woord                        }

  USES
    Dat_Aut;              { Implementatie van Datastructuur Eindige Automaat }

  CONST
    Letters = ['A'..'Z','a'..'z'];

  PROCEDURE Error( J : Byte ); FORWARD; { Eerst belangrijke zaken          }

  PROCEDURE Term (Pat : STRING; VAR PatPlek : Byte; VAR Aut : Automaat );
    FORWARD;
  PROCEDURE Fact (Pat : STRING; VAR PatPlek : Byte; VAR Aut : Automaat );
    FORWARD;


  PROCEDURE Expr (Pat : STRING; VAR PatPlek : Byte; VAR Aut : Automaat );

  { DOEL     Procedures Expr, Term en Fact ontleden door recursief       }
  {          elkaar aan te roepen het ingevoerde patroon, en zetten deze }
  {          om in een eindige automaat.                                 }
  { IN       Pat      - het ingevoerde patroon                           }
  { INUIT    PatPlek - positie in patroon onder behandeling              }
  { UIT      Aut   - de geconstrueerde automaat.                         }

    VAR
      Aut1,
      Aut2  : Automaat;
    BEGIN { Expr }
      Term( Pat, PatPlek, Aut1 );
      Aut := Aut1;
      IF ( PatPlek <= Length(Pat) ) AND ( Pat[PatPlek] = '+' )
      THEN BEGIN
        INC(PatPlek);
        Expr( Pat, PatPlek, Aut2 );
        Dat_Aut.Plus( Aut1, Aut2, Aut );
      END;
    END; { Expr }


  PROCEDURE Term;
    VAR
      Aut1,
      Aut2  : Automaat;
    BEGIN
      Fact( Pat, PatPlek, Aut1 );
      Aut := Aut1;
      IF ( PatPlek <= Length(Pat) )
         AND ( Pat[PatPlek] IN ['('] + Letters )
      THEN BEGIN
        Term(  Pat, PatPlek, Aut2 );
        Dat_Aut.Conc(Aut1, Aut2, Aut);
      END;
    END; { Term }


PROCEDURE Fact;
  VAR
    Aut1  : Automaat;
  BEGIN
    IF Pat[PatPlek] = '('
    THEN BEGIN
      INC(PatPlek);
      Expr(  Pat, PatPlek, Aut1 );
      IF ( PatPlek > Length(Pat) ) OR ( Pat[PatPlek] = ')' )
      THEN
        INC(PatPlek)
      ELSE
        Error(PatPlek);
    END
    ELSE
      IF ( PatPlek <= Length(Pat) ) AND ( Pat[PatPlek] IN Letters )
      THEN BEGIN
        Dat_Aut.MaakTak( Pat[PatPlek], Aut1 );
        INC(PatPlek);
      END
      ELSE
        Error(PatPlek);
    IF ( PatPlek > Length(Pat) ) OR ( Pat[PatPlek] <> '*' )
    THEN
      Aut := Aut1
    ELSE BEGIN
      Dat_Aut.Ster( Aut1, Aut );
      INC(PatPlek);
    END
  END; { Fact }


PROCEDURE Error( J : Byte );
{ DOEL     Melden van fout in patroon. Stopt daarna executie.
{ IN       J  -  plek in patroon waar fout wordt gevonden.
  BEGIN
    WriteLn('Fout op positie ', J:2); Halt(1)
  END;  { Error }

PROCEDURE LeesInvoer( VAR Pat, Woord : STRING );
{ DOEL     Leest expressie en woord in.
{ UIT      Pat     - Ingelezen Patroon
{          Woord   - Ingelezen Woord
  BEGIN
    Write('Het zoekpatroon (als reguliere expressie)    : ');
    ReadLn( Pat );
    Write('Het woord dat aan patroon getest moet worden : ');
    ReadLn( Woord );
  END;  { LeesInvoer }


VAR   { In gebruik in hoofdprogramma, maar niet globaal in proc's
  Pat,
  Woord     : STRING;
  PatPlek   : Byte;
  Auto      : Automaat;
  Resultaat : Boolean;

BEGIN { Hoofdprogramma }
  LeesInvoer( Pat, Woord );
  PatPlek := 1;
  Expr(  Pat, PatPlek, Auto );
  IF Dat_Aut.Simulatie( Woord, Auto )
  THEN   WriteLn('Succes !')
  ELSE   WriteLn('Past Niet');
END.
```

```
PROCEDURE Plus   ( Aut1, Aut2 : Automaat; VAR AutP : Automaat );
{ OPM  Er wordt een nieuwe begin- en nieuwe eindknoop genomen.          }
  VAR
    Samen,
    Splits : KnoopTal;
  BEGIN
    Splits := NieuweKnoop;
    Samen  := NieuweKnoop;
    ZetKnoop( Splits,    GeenLabel, Aut1.Start, Aut2.Start );
    ZetKnoop( Aut1.Eind, GeenLabel, Samen,      NulKnoop   );
    ZetKnoop( Aut2.Eind, GeenLabel, Samen,      NulKnoop   );
    ZetKnoop( Samen,     GeenLabel, NulKnoop,   NulKnoop   );
    AutP.Start := Splits;
    AutP.Eind  := Samen;
  END;  { Plus }



{ *** FUNCTIE OM AUTOMAAT OP WOORD TE SIMULEREN ************************** }

  FUNCTION Simulatie( Woord : STRING; Aut : Automaat ) : BOOLEAN;

  { HOE  Volg takken zonder letters mbv. een variant van DFS; als stapel   }
  {      dient de voorkant van een Deque. Er wordt een array gebruikt om   }
  {      de bezochte knopen te markeren. Wanneer we takken tegen komen met }
  {      een gezochte letter worden deze aan de achterzijde van de Deque   }
  {      geplaatst. De Deque doet zo dienst als dubbele stapel.            }
  {      Een speciaal symbool wordt aan het opgegeven woord toegevoegd om  }
  {      op eenvoudige wijze te kunnen zien of na afloop de eindknoop      }
  {      bereikbaar is.                                                    }

    TYPE
      MarkeerKnoopArray = ARRAY[KnoopTal] OF BOOLEAN;

    CONST
      Nul   : BYTE = 0;              { Scheidt voor- en achtekant Deque      }
      MerkTeken    = #255;           { Verlenging invoerwoord                }
    VAR
      TopStapel,                     { Voorste element Deque Stapels         }
      EindKnoopTak,
      I            : BYTE;
      Stapels      : DequeOfByte;

                   { In gebruik als dubbele stapel, gescheiden door Nul.     }
                   { De voor-stapel wordt gebruikt voor de wandeling         }
                   { op zoek naar takken met de juiste letter.               }
                   { De na-stapel bevat de knopen die eindpunt zijn van de   }
                   { gevonden takken. Hier gaat de wandeling bij de vol-     }
                   { gende letter weer van start.                            }
                   { Wanneer voor-stapel leeg is wisselen we de stapels      }
                   { door het symbool Nul van voor naar achter te plaatsen.  }

      BezochtVoor,
      BezochtNa    : MarkeerKnoopArray;

                   { Knopen markeren bij wandeling door automaat, om         }
                   { oneindig rondlopen te voorkomen.                        }
                   { In BezochtVoor alle knopen die bezocht zijn tijdens de  }
                   { wandeling op zoek naar de huidige letter; dit zijn de   }
                   { knopen die op de voor-stapel gestaan hebben.            }

    PROCEDURE VoegToeVoor( I : BYTE );
    { DOEL Voeg Knoop I toe aan voorzijde & markeer als bezocht.            }
      BEGIN
        Dat_Deq.VoegToe( Dat_Deq.Voorzijde, I, Stapels );
        BezochtVoor[ I ] := TRUE;
      END;  { VoegToeVoor }

    PROCEDURE VoegToeNa( J : BYTE );
    { DOEL Voeg Knoop J toe aan achterzijde & markeer als bezocht.          }
      BEGIN
        Dat_Deq.VoegToe( Dat_Deq.Achterzijde, J, Stapels );
        BezochtNa[ J ] := TRUE;
      END;  { VoegToeNa }

    PROCEDURE ZetFalse(VAR A: MarkeerKnoopArray);
    { DOEL Maak alle waarden van array A FALSE, dwz 'niet bezocht'.         }
      VAR
        I : KnoopTal;
      BEGIN
        FOR I := 0 TO 255 DO A[I] := FALSE;
      END;  { ZetFalse }


    BEGIN { Simulatie }
      Dat_Deq.MaakLeeg( Stapels );       { INITIALISATIE DATASTRUCTUREN     }
      ZetFalse( BezochtVoor );
      ZetFalse( BezochtNa );
      Woord := Woord+MerkTeken;          { Voeg speciale extra letter toe   }

      VoegToeNa( Aut.Start );            { Zet startknoop op na-stapel      }

      FOR I := 1 TO Length(Woord)        { LETTER VOOR LETTER ZOEKEN        }
      DO BEGIN {FOR I}
                                         { Voor-stapel wordt na-stapel      }
        Dat_Deq.VoegToe( Dat_Deq.AchterZijde, Nul, Stapels );
        BezochtVoor := BezochtNa;
        ZetFalse( BezochtNa );

        Dat_Deq.HaalWeg( Dat_Deq.Voorzijde, TopStapel, Stapels );
        WHILE TopStapel <> Nul
        DO BEGIN
          IF Knoop[TopStapel].Kar = GeenLabel
          THEN BEGIN                                  { - Tak zonder letter }
            EindKnoopTak := Knoop[TopStapel].Volg1; { - 1e uitgaande tak  }
            IF (EindKnoopTak <> NulKnoop) AND (NOT BezochtVoor[EindKnoopTak])
            THEN
              VoegToeVoor( EindKnoopTak );
            EindKnoopTak := Knoop[TopStapel].Volg2; { - 2e uitgaande tak  }
            IF (EindKnoopTak <> NulKnoop) AND (NOT BezochtVoor[EindKnoopTak])
            THEN
              VoegToeVoor( EindKnoopTak );
          END
          ELSE BEGIN                                  { - Tak met letter    }
            EindKnoopTak := Knoop[TopStapel].Volg1;
            IF (Knoop[TopStapel].Kar = Woord[I])      { ... juiste letter   }
               AND (NOT BezochtNa[EindKnoopTak])
            THEN
              VoegToeNa( EindKnoopTak );
          END{IF};
          DAT_Deq.HaalWeg( Dat_Deq.Voorzijde, TopStapel, Stapels );
        END{WHILE};

      END{FOR};

      Simulatie := BezochtVoor[ Aut.Eind ];
    END;  { Simulatie }

BEGIN
  LaatsteKnoopNr := 0;
END.
```

```
UNIT DAT_AUT;
{ DOEL    Implementatie van eindige automaten voor patroon-herkenning.     }
{         Knopen van eindige automaten worden opgeslagen in een globaal    }
{         array dat geen deel uitmaakt van de interface en dus alleen via  }
{         procedures ingevuld kan worden.  De automaten zelf worden gespe-  }
{         cificeerd door het geven van de indices van start- en eindknoop.  }
{         Voor documentatie zie dictaat Datastructuren RULeiden.            }
{ COMPILER   Turbo-Pascal 4.0 (en hoger)                                    }
{ DATUM   18 feb 90 - operaties; 20 apr 90 simulatie automaat         HJH   }

INTERFACE { ############################################################ }

  USES
    Dat_Deq;        { Implementatie van de datastructuur Deque.           }

  TYPE
    KnoopTal  = 0..255;
    Automaat  = RECORD
                  Start,
                  Eind  : KnoopTal
                END;

  PROCEDURE MaakTak( Let : CHAR           ; VAR AutM : Automaat );
  { DOEL    Maakt automaat bestaande uit enkele tak met letter Let.       }
  { IN      Let   - Letter van de tak                                     }
  { UIT     AutM  - De geconstrueerde automaat                            }

  PROCEDURE Conc   ( Aut1, Aut2 : Automaat; VAR AutC : Automaat );
  { DOEL    Concateneert twee automaten door ze te verbinden met een tak. }
  { IN      Aut1, Aut2  - De te concateneren automaten                    }
  { UIT     AutC  - De geconstrueerde automaat                            }

  PROCEDURE Plus   ( Aut1, Aut2 : Automaat; VAR AutP : Automaat );
  { DOEL    Zet twee gegeven automaten 'parallel' (keuze, plus)           }
  { IN      Aut1, Aut2  - De ingevoerde automaten                         }
  { UIT     AutP  - De geconstrueerde automaat                            }

  PROCEDURE Ster   ( Aut1       : Automaat; VAR AutS : Automaat );
  { DOEL    Voert ster-operatie uit op gegeven automaat (herhaling)       }
  { IN      Aut1  - De ingevoerde automaat                                }
  { UIT     AutS  - De geconstrueerde automaat                            }

  FUNCTION Simulatie( Woord : STRING; Aut : Automaat ) : BOOLEAN;
  { DOEL   Gaat na of een pad in de automaat Aut bestaat met tekst Woord  }
  { IN     Woord  - de tekst                                              }
  {        Aut    - de automaat                                           }
  { TERUG  TRUE desdals zo'n pad bestaat.                                 }


IMPLEMENTATION { ######################################################## }

{ BESCHRIJVING   Elke knoop van een eindige automaat heeft   of           }
{       - een uitgaande tak met label,              of                  }
{       - een uitgaande tak zonder label,           of                  }
{       - twee uitgaande takken zonder label.                           }
{       De eindknoop heeft geen uitgaande takken.                       }

  TYPE
    KnoopTiep = RECORD
                  Kar   : CHAR;
                  Volg1,
                  Volg2 : KnoopTal
                END;

  CONST
    GeenLabel = #0;               { Markering takken zonder label       }
    NulKnoop  : KnoopTal = 0;     { In gebruik bij knopen met <2 takken }

  VAR
    Knoop          : ARRAY[KnoopTal] OF KnoopTiep;    { array met knopen  }
    LaatsteKnoopNr : KnoopTal;     { Hoogste index in gebruik bij Knoop  }

  FUNCTION NieuweKnoop : KnoopTal;
  { DOEL      Levert laagste niet in gebruik zijnde index van Knoop       }
  {           Is een lokale functie, dus niet buiten unit bruikbaar.      }
  { GLOBAAL   LaatsteKnoopNr                                              }
  { PRE       LaatsteKnoopNr < 255.                                       }
  { POST      LaatsteKnoopNr een (1) opgehoogd                            }
  { TERUG     De nieuwe waarde van LaatsteKnoopNr                         }

    BEGIN
      INC(LaatsteKnoopNr);
      NieuweKnoop := LaatsteKnoopNr;
    END; { NieuweKnoop }

  PROCEDURE ZetKnoop( Nr      : KnoopTal;
                      InKar   : CHAR;
                      InVolg1,
                      InVolg2 : KnoopTal );
  { DOEL      Het invullen van een knoop van een automaat; lokaal in unit. }
  { GLOBAAL   Knoop    - Array waarin knopen opgeslagen worden            }
  { IN        Nr       - Plek in array Knoop waar knoop opgeslagen wordt  }
  {           InKar    - Letter van tak                                   }
  {           InVolg1, InVolg2  - Opvolger knopen van uitgaande takken    }
  { POST      Knoop[Nr] heeft de waarde (InKar, InVolg1, InVolg2)         }

    BEGIN
      WITH Knoop[Nr]
      DO BEGIN
        Kar   := InKar;
        Volg1 := InVolg1;
        Volg2 := InVolg2;
      END
    END;  { ZetKnoop }

{*** OPERATIES DIE AUTOMAAT OPBOUWEN ************************************ }

  PROCEDURE MaakTak( Let : CHAR           ; VAR AutM : Automaat );
  { OPM     Reserveert twee nieuwe elementen uit array Knoop.             }
    VAR
      NwStart,
      NwEind   : KnoopTal;
    BEGIN
      NwStart := NieuweKnoop;
      NwEind  := NieuweKnoop;
      ZetKnoop( NwStart, Let,       NwEind,   NulKnoop);
      ZetKnoop( NwEind,  GeenLabel, NulKnoop, NulKnoop);
      AutM.Start := NwStart;
      AutM.Eind  := NwEind;
    END;  { MaakTak }

  PROCEDURE Conc   ( Aut1, Aut2 : Automaat; VAR AutC : Automaat );
    BEGIN
      ZetKnoop( Aut1.Eind, GeenLabel, Aut2.Start, NulKnoop );
      AutC.Start := Aut1.Start;
      AutC.Eind  := Aut2.Eind;
    END;  { Conc }

  PROCEDURE Ster   ( Aut1       : Automaat; VAR AutS : Automaat );
  { OPM     Er is nieuwe eindknoop nodig evenals verplaatsing beginknoop. }
    VAR
      NwEind : KnoopTal;
    BEGIN
      NwEind := NieuweKnoop;
      ZetKnoop( Aut1.Eind, GeenLabel, Aut1.Start,  NwEind   );
      ZetKnoop( NwEind,    GeenLabel, NulKnoop,    NulKnoop );
      AutS.Start := Aut1.Eind;
      AutS.Eind  := NwEind
    END;  { Ster }
```

```
UNIT DAT_DEQ;

{ DOEL       Implementatie van een Double Ended Queue (Deque);              }
{            de elementen van de Deque zijn Bytes.                          }
{            Implementatie mbv. een array, dus op cyclische wijze.          }
{            Oefening in abstracte programmeerstijl.                        }
{ OPMERKING  De spec's bij de proc's/fun's verdienen geen schoonheidsprijs  }
{ COMPILER   Turbo-Pascal 4.0 (en evt. hoger)                               }
{ DATUM      20 apr 90                                                      }
{                                                                      HJH  }

INTERFACE { ############################################################ }

  CONST
    MinIndex = 0;
    MaxIndex = 255;

  TYPE
    IndexTiep = MinIndex..MaxIndex;
    ZijdeTiep = ( VoorZijde, AchterZijde );
    ElementTiep = BYTE;
    DequeOfByte = RECORD
                    Voor,
                    Achter : IndexTiep;
                    Rij    : ARRAY[IndexTiep] OF ElementTiep
                  END;

{ Om in andere programma's Deque's te kunnen declareren. We zouden echter  }
{ liever de velden "onzichtbaar" willen maken zodat alleen toegang tot de  }
{ datastructuur via de operaties mogelijk is.                              }


{ *** DE OPERATIES OP DE DATASTRUCTUUR ********************************** }

  FUNCTION  IsLeeg(   VAR Dq : DequeOfByte ): BOOLEAN;
  { DOEL    Geef aan of Deque Dq leeg is.                                  }
  { IN      Dq (DequeOfByte), VAR om kopieren te voorkomen.                }

  FUNCTION  IsVol (   VAR Dq : DequeOfByte ): BOOLEAN;
  { DOEL    Geef aan of Deque Dq lmaximale capaciteit gebruikt.            }
  { IN      Dq (DequeOfByte), VAR om kopieren te voorkomen.                }

  PROCEDURE MaakLeeg( VAR Dq : DequeOfByte );
  { DOEL    Maak Deque Dq leeg.                                            }
  { INUIT   Dq (DequeOfByte), VAR hier noodzakelijk.                       }

  PROCEDURE VoegToe(     Kant  : ZijdeTiep;
                         Waard : ElementTiep;
                     VAR Dq    : DequeOfByte );
  { DOEL    Voeg element Waard toe aan Dq aan de gespecificeerde Kant.     }
  { IN      Kant  - zijde waar waarde toegevoegd wordt                     }
  {         Waard - toe te voegen waarde                                   }
  { INUIT   Dq    - deque waarop operatie uitgevoerd wordt                 }
  { PRE     Dq is niet vol; capaciteit is implementatie afhankelijk!       }

  PROCEDURE HaalWeg(     Kant  : ZijdeTiep;
                     VAR Waard : ElementTiep;
                     VAR Dq    : DequeOfByte );
  { DOEL    Verwijder el't uit Dq aan juiste Kant; ken deze toe aan Waard. }
  { IN      Kant  - zijde waar waarde verwijderd wordt                     }
  { UIT     Waard - verwijderde waarde                                     }
  { INUIT   Dq    - deque waarop operatie uitgevoerd wordt                 }
  { PRE     Dq is niet leeg                                                }


MPLEMENTATION { ######################################################## }

  { HOE Inhoud van de Deque bestaat hier uit Bytes.                        }
  {     Implementatie mbv "cyclisch" array dat 256 elementen kan bevatten. }
  {     De door de Deque gebruikte elementen worden aangegeven door        }
  {         Voor    - index eerste element                               }
  {         Achter  - index laatste element plus een (modulo 256)        }
  {     De elementen in gebruik zijn                                       }
  {             Voor, Voor+1, ..              Achter-1   (Voor <= Achter)  }
  {     of      Voor, Voor+1, .. 255, 0, ..   Achter-1.  (Voor > Achter)   }
  {     Als Voor = Achter dan is de Deque leeg. Om verwarring met volle    }
  {     Deque te voorkomen moet er altijd tenminste 1 vakje leeg blijven.  }

  FUNCTION  IsLeeg;
  { PRE     Niet alle vakjes van Dq gebruikt !                             }
    BEGIN
      IsLeeg := (Dq.Voor = Dq.Achter)
    END; { IsLeeg }

  FUNCTION  IsVol;
    BEGIN
      IF Dq.Voor = MinIndex
      THEN  IsVol := (Dq.Achter = MaxIndex)
      ELSE  IsVol := (Dq.Voor = Dq.Achter+1)
    END;  { IsVol }

  PROCEDURE MaakLeeg;
    BEGIN
      Dq.Voor := MinIndex;
      Dq.Achter := MinIndex;
    END; { MaakLeeg }

  PROCEDURE VoegToe;
    BEGIN
      IF Kant = VoorZijde
      THEN BEGIN
        IF Dq.Voor = MinIndex
        THEN  Dq.Voor := MaxIndex
        ELSE  DEC(Dq.Voor);
        Dq.Rij[Dq.Voor] := Waard;
      END
      ELSE BEGIN
        Dq.Rij[Dq.Achter] := Waard;
        IF Dq.Achter = MaxIndex
        THEN  Dq.Achter := MinIndex
        ELSE  INC(Dq.Achter);
      END;
      IF IsLeeg(Dq) THEN Halt(1);   { vanwege VOLLE Deque ! }
    END;  { VoegToe }

  PROCEDURE HaalWeg;
    BEGIN
      IF IsLeeg(Dq) THEN Halt(1);   { vanwege lege Deque ! }
      IF Kant = VoorZijde
      THEN BEGIN
        Waard := Dq.Rij[Dq.Voor];
        IF Dq.Voor = MaxIndex
        THEN  Dq.Voor := MinIndex
        ELSE  INC(Dq.Voor);
      END
      ELSE BEGIN
        IF Dq.Achter = MinIndex
        THEN  Dq.Achter := MaxIndex
        ELSE  DEC(Dq.Achter);
        Waard := Dq.Rij[Dq.Achter];
      END{IF THEN ELSE};
    END;  { HaalWeg }

END.
```