Datastructuren en algoritmen

dr. ir. Annemie Vorstermans

Academiejaar 2016-2017

Inhoudsopgave

Hoofdstuk 1

Elementaire datastructuren

1.1 Inleiding

In dit hoofdstuk wordt dieper ingegaan op de begrippen array en linked list. Als men aan een programmeur vraagt: *Wat is een array?*, dan zal hij veelal antwoorden dat dit een opeenvolgende set van geheugenlocaties is. Dit is echter hoe men vaak een array implementeert. De compilers zullen er inderdaad voor zorgen dat de verschillende rij-elementen op opeenvolgende locaties in het geheugen staan omdat dit sneller is (als de rekeneenheid een rij-element nodig heeft, heeft hij heel vaak daarna het volgende element nodig). In het kader van deze cursus wordt ook op een hoger, abstracter niveau naar de datastructuren *array* en *linked list* gekeken. De abstracte datastructuren List, Set en Map worden besproken. Ook de veelgebruikte stapel (stack) en wachtlijn (queue) komen aan bod.

1.2 Abstracte array, gelinkte lijst

1.2.1 Array

Een **array** is een set van paren: **index** en **waarde**. Er is een correspondentie of mapping tussen de index en de waarde. Een array-datastructuur moet minstens over de volgende 3 functionaliteiten beschikken.

1. **CREATE** voor het aanmaken van een nieuwe, lege rij
2. **RETRIEVE(rij,index)** die de waarde met de juiste index teruggeeft, of een fout
3. **STORE(rij,index,waarde)**

Voor een gemakkelijk gebruik van array's is het ook handig als men de lengte van de rij kan opvragen en in mindere mate het aantal elementen effectief aanwezig.

Een array is een veelgebruikte datastructuur. Het nadeel van deze structuur wordt duidelijk als men elementen moeten toevoegen en verwijderen. Dit veroorzaakt vele manipulaties en is dus duur.

1.2.2 Gelinkte lijst

Het alternatief is een gelinkte lijst. Hier wordt bij elk element ook de locatie van het volgende element in de lijst opgeslagen. De opeenvolgende elementen kunnen zich gelijk waar in het geheugen bevinden.

Men moet steeds weten waar het eerste element zich bevindt, en wat het laatste element is (geen volgend element).

Het toevoegen van elementen gebeurt meestal vooraan of achteraan, maar indien de lijst gesorteerd moet zijn dan kan dit ook ergens middenin. Het opvragen van 1 element uit de lijst kan soms lang duren omdat de lijst sequentiëel doorlopen moet worden.

1.3 Array-, gelinkte lijst-implementatie

In de meeste programmeertalen wordt een array als een opeenvolging van geheugenplaatsen geïmplementeerd. De index is dan de offset ten opzichte van het adres van de eerste geheugenplaats. Het is echter ook mogelijk om dat met een gelinkte lijst te implementeren. Elk element moet dan de index en de waarde bevatten.

Een nadeel van de array-implementatie is dat men op voorhand een maximale grootte moet vastleggen. In de nieuwere programmeertalen kan men wel een nieuwe grotere array maken maar dit houdt weer veel manipulaties in.

Meestal gaat men een gelinkte lijst implementeren aan de hand van pointers (of referenties). Elke knoop bevat een waarde en een pointer (referentie) naar de volgende knoop.

Het is echter ook mogelijk om een array-implementatie te maken. Elk element van de array bevat dan de waarde en de index van de plaats van de volgende waarde.

Hierna volgt mogelijke Java-code voor een enkelvoudige gelinkte lijst. In de praktijk gaat men steeds de reeds voorziene taalmogelijkheden gebruiken (Java: LinkedList).

```
class Element<E> {
 private E data;
 private Element<E> volgend;

 public Element(E d,Element next){
   data=d;
   volgend=next;
 }
 public E getData(){
   return data;
 }
 public void setVolgend(Element e){
   volgend=e;
 }
 public Element getVolgend(){
   return volgend;
 }
}

class GelinkteLijst<E> {
 private Element<E> start;

 public GelinkteLijst(){
   start=null;
 }
```

```
 public String toString() {
   StringBuffer sb = new StringBuffer();
   Element hulp=start;
   while (hulp!=null) {
     sb.append(hulp.getData()+" ");
     hulp=hulp.getVolgend();
   }
   return sb.toString();
 }
 public void voegVooraanToe(E getal) {
   start=new Element<E>(getal,start);
 }
 public E haalVooraanAf() {
   if (start==null) throw
         new NullPointerException("gelinkte lijst is leeg");
   Element<E> hulp=start;
   start=start.getVolgend();
   return hulp.getData();
 }
 public void voegAchteraanToe(E getal) {
   if (start==null) start=new Element(getal,null);
   else {
     Element hulp=start;
     while (hulp.getVolgend()!=null){hulp=hulp.getVolgend();}
     hulp.setVolgend(new Element(getal,null));
   }
 }
 public E haalAchteraanAf() {
   if (start==null) throw
         new NullPointerException("gelinkte lijst is leeg");
   Element<E> hulp=start,vorig=null;
   while (hulp.getVolgend()!=null){vorig=hulp; hulp=hulp.getVolgend();}
   vorig.setVolgend(null);
   return hulp.getData();
 }
}
```

1.4 ADT's

Veelal gaat men ADT's (Abstract Data Types) definiëren: men beschrijft enkel de eigenschappen. In de huidige OO-talen wordt dit meestal voorgesteld door een Interface.

1.4.1 List, Set, Map

Een List is een lineaire geordende structuur waarbij duplicate elementen mogen voorkomen. De volgorde (plaats) van de elementen is belangrijk.

Bij een Set is de volgorde niet van belang. Een element behoort tot de set of niet. Net als bij een wiskundige verzameling mogen er geen duplicate elementen voorkomen.

Een Map is sleutel/waarde-tabel. Informatie wordt opgeslagen en opgevraagd aan de hand van een sleutel. Het wordt ook vaak een dictionary genoemd. Elke gebruikte sleutel moet ook uniek zijn.

In Java gaat men meestal de implementaties ArrayList of LinkedList, HashSet en HashMap gebruiken. Deze zijn niet thread-safe. Daarvoor zijn wrapper-implementaties en specifieke thread-safe implementaties (java.util.concurrency) voorzien. Hashing wordt later besproken.

1.4.2 Specifiekere ADT's en implementaties

Stack

De stack (stapel) is een veelgebruikte datastructuur met beperkte toegang. Men kan een element toevoegen (push) en verwijderen (pop) aan één kant van de stack. Je kan dit vergelijken met een stapel werk op je bureau: een nieuwe taak leg je steeds bovenaan op de stapel, en als je begint te werken neem je eerst wat er van boven ligt. Men noemt dit *LIFO (last in, first out)*. Een stack kan men via een array of een gelinkte lijst implementeren.

Sommige rekenmachines gebruiken een stapel om de bewerkingen uit te voeren: bv de onderstaande infix-notatie

```
2*(((4+8)*(3+7))+6)
```

kan geschreven worden in postfix-notatie als

```
2 4 8 + 3 7 + * 6 + *
```

Hierbij worden de getallen op de stapel geplaatst. Elke keer als er een bewerking (hier worden enkel de binaire operatoren + - * en / beschouwd) voorkomt, worden er 2 getallen van de stapel gehaald, de bewerking wordt uitgevoerd en het resultaat wordt terug op de stapel geplaatst. Er zijn geen haakjes nodig om de juiste volgorde van de bewerkingen vast te leggen.

Ook bij methode-oproepen wordt door de computer een stapel gebruikt. Bij te diepe recursie (steeds opnieuw oproepen van dezelfde methode) krijgt men dan soms de foutmelding *stack overflow*.

Queue

De queue (wachtlijn) is ook een veelgebruikte datastructuur met beperkte toegang. Men kan een element toevoegen aan het einde van de lijn, en een element verwijderen aan het begin van de lijn. Men kan dit dus vergelijken met een rij in de winkel: wie eerst binnenkomt, wordt eerst bediend. Men spreekt van *FIFO (first in first out)*.

Een queue kan men implementeren aan de hand van een array. Men moet dan natuurlijk voldoende plaats voorzien (aantal elementen dat terzelfdertijd in de wachtlijn kan zijn). Men heeft 2 indices nodig, één om het eerste en één om het laatste element aan te duiden. Men gebruikt *wraparound* als het einde van de array bereikt is en er vooraan nog plaats is.

Een veel natuurlijkere implementatie is deze met een gelinkte lijst. Men gebruikt dan enkel achteraan toevoegen en vooraan verwijderen. De lijst is steeds zo groot als nodig.

In de meeste huidige programmeertalen is een queue reeds voorzien (bv in Java: LinkedList of de thread safe java.util.concurrent.ArrayBlockingQueue en java.util.concurrent.LinkedBlockingQueue).

Een priorityqueue wijkt af van de fifo-eigenschap omdat het de elementen ordent volgens prioriteit (enkel fifo als gelijke prioriteit).

1.5 Voorbeelden datastructuur-keuzes

1.5.1 Polynoom

Neem als voorbeeld een polynoom (veelterm). De algemene vorm ervan is:

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0$$

Het ligt voor de hand om dit in een array op te slaan, met de waarde a_i op index i.

Voor de implementatie ervan is men afhankelijk van de gebruikte programmeertaal. Heel waarschijnlijk zal er een array-implementatie beschikbaar zijn. Voordat men elementen in de array kan plaatsen, moet deze eerst gecreëerd worden. Hierbij moet men een grootte opgeven dat het maximum aantal elementen dat de array kan bevatten bepaalt. Bij de oudere programmeertalen gebeurde dit statisch (de grootte wordt vastgelegd tijdens het programmeren), bij de nieuwere kan dit al meestal dynamisch (at run-time). Het is dan wel niet mogelijk om de array te vergroten. Men lost dit op door een nieuwe, grotere array te creëren en de waarden te kopiëren (enkel als dynamisch).

Als men de keuze van de datastructuur van het veeltermvoorbeeld bekijkt, dan kan men daar enige opmerking bij geven:

- hoe groot moet de array oorspronkelijk zijn?
  - te klein: veelvuldig nieuwe array creëren en kopiëren (+ vernietigen)
  - te groot: nutteloos verbruik geheugen
- wat als volgende polynoom: $x^{1000} + 1$?

Deze laatste polynoom noemt men schaars (sparse). Schaarsheid is een intuitief begrip en gebruikt men als het aantal niet-nul-elementen veel kleiner is dan het aantal nul-elementen. Men gaat dan meestal enkel de niet-nul-elementen opslaan.

```
class Term {
 double coeff;
 int exp;
...
};
class Polynoom {
 int aantalTermen
 vector<Term> termen;
...
};
```

1.5.2 Schaarse matrix

Een matrix is een wiskundig object dat vaak gebruikt wordt in de modellering van fysische problemen. Een matrix heeft 2 afmetingen. Men spreekt van een m*n-matrix met m rijen en n kolommen (dus m*n-elementen).

Voor de opslag ervan kan men 2-dimensionale array's gebruiken, bv `int[][] matrix`

$$\begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 & 0 \\ 0 & 11 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 31 \\ 91 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 26 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Als men deze voorbeeldmatrix bekijkt, dan ziet men veel nullen; slechts 9 van de 42 elementen zijn verschillend van 0. Dit is een schaarse matrix. Men spreekt van een schaarse matrix als er veel meer nullen dan niet-nul-elementen voorkomen (intuitief). Voor schaarse matrices is een alternatieve representatie voordelig, vooral voor grote matrices. Hierbij worden enkel de niet-nul-elementen opgeslagen in een lijst (of array) van 3-tuples: (rijnummer,kolomnummer,waarde). Het is gemakkelijker als de lijst geordend is volgens

- stijgend rijnummer
- stijgend kolomnummer per rij

Het is ook nodig om het aantal rijen en kolommen in de alternatieve voorstelling op te nemen. Voor het gemak kan men dan ook het aantal niet-nul-elementen opslaan.

De bovenstaande matrix wordt dan voorgesteld door:

$$\begin{pmatrix} 6 & 7 & 9 \\ 1 & 1 & 15 \\ 1 & 4 & 22 \\ 1 & 6 & -15 \\ 2 & 2 & 11 \\ 2 & 3 & 3 \\ 3 & 4 & -6 \\ 4 & 7 & 31 \\ 5 & 1 & 91 \\ 6 & 3 & 28 \end{pmatrix}$$

Voor het transponeren van dergelijke matrices kan men op 2 manieren te werk:

1. voor elke rij i uit de alternatieve voorstelling A, neem (i,j,waarde) en plaats (j,i,waarde) in B. Aangezien men een goede volgorde moet blijven behouden, betekent dit veel kopiëren binnen de array.
2. voor alle elementen in kolom j, plaats element (i,j,waarde) op plaats (j,i,waarde). Hierbij worden dus eerst alle elementen met origineel kolomnummer 1 behandeld, daarna deze met kolomnummer 2 enz.

1.5.3 Triangulaire matrices

Triangulaire matrices zijn vierkante matrices waarbij alle elementen onder of boven de diagonaal nul zijn. In een onderdiagonale matrix met n rijen is het maximum aantal niet-nul-elementen in rij i gelijk aan i. In het totaal zijn er dus maximum

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

niet-nul-termen. Voor grote n loont het de moeite om plaats te besparen.

a) Zoek een adresseringsformule voor de elementen a_{ij} als deze zijn opgeslagen in een array $B[1 \cdots \frac{n(n+1)}{2}]$.

b) Wat is de relatie tussen i en j voor de nul-elementen van A?

Hoofdstuk 2

Recursie en backtracking

2.1 Inleiding

Elke methode (subprogramma, functie, procedure) kan andere methoden oproepen, met als enige voorwaarde dat deze gekend moeten zijn door de oproepende methode. Daarenboven kan elke methode zichzelf oproepen. Deze techniek wordt recursie genoemd. Recursie is geen nieuw begrip. In de wiskunde wordt dit vaak toegepast. Hierbij wordt een probleem herleid tot een iets eenvoudiger probleem, totdat men een triviale oplossing heeft. Een heel eenvoudig voorbeeld is de faculteitsberekening. Men kan dit op twee manieren definiëren (iteratief of recursief):

$$\begin{array}{lcll} n! & = & n*(n-1)*(n-2)*\cdots*1 & \\ n! & = & 1 & \text{voor } n=0 \text{ en } n=1 \\ & & n*(n-1)! & \text{voor } n>1 \end{array}$$

Bemerk dat bij de recursieve definities het probleem steeds herleid wordt tot een eenvoudiger (n! wordt uitgedrukt in functie van (n-1)!). Dit kan op zijn beurt herleid worden tot een eenvoudiger probleem tot uiteindelijk een voor de hand liggende oplossing verkregen wordt (1!), van waaruit het geheel kan berekend worden. Dit idee blijft algemeen geldig, ook bij gebruik van recursieve methoden.

Als men bovenstaande recursieve definitie gebruikt voor het berekenen van 4!, dan volgt men onderstaand schema.

```
4! = 4 * 3!
         3! = 3 * 2!
                  2! = 2 * 1!
                           1! = 1
                  2! = 2 * 1 = 2
         3! = 3 * 2 = 6
4! = 4 * 6 = 24
```

2.2 Recursieve subprogramma's

Dit gebeurt op analoge manier bij een recursief programma : de computer houdt inwendig een stapel bij, waarop de tussenresultaten gestockeerd worden. Die kunnen dan één voor één van de stapel gehaald worden (in omgekeerde volgorde van het stapelen). Het bijhouden van informatie op de stapel en het terug afhalen van die informatie gebeurt automatisch: de programmeur hoeft hiervoor geen speciale instructies te schrijven.

Faculteit

Volgend programma bevat een recursieve functie die de faculteit van een getal berekent:

```
PROGRAM faculteitsberekening (input,output);

VAR n : Integer;

FUNCTION fac (m : Integer) : Integer;
  BEGIN
    IF m = 1
    THEN fac := 1
    ELSE fac := m * fac(m-1)
  END;

BEGIN
  REPEAT
    write('geef positief getal : ');
    readln(n)
  UNTIL n > 0;
  writeln(n:3, '! = ', fac(n):6)
END.
```

Algemene richtlijnen voor recursieve methoden

- In tenminste één triviaal geval moet de procedure of functie beëindigd worden zonder zichzelf op te roepen.
- Wanneer een recursieve procedure of functie opgeroepen wordt, dan mag dit slechts als men daardoor dichter naar het triviaal geval toegaat.

2.3 Voorbeelden

2.3.1 De torens van Hanoi

Op het binnenplein van een boeddistisch klooster in Hanoi staan er drie palen (A, B en C). In een ver verleden werden er op paal A 64 schijven gemonteerd, telkens met een kleinere diameter.

De monniken krijgen als opdracht deze 64 schijven te verplaatsen van paal A naar paal C, zodanig dat ze in dezelfde volgorde op de paal C terecht komen. Bij het verplaatsen, moeten de monniken de volgende drie regels in acht nemen :

1. er mag slechts 1 schijf per keer verplaatst worden van de ene paal naar een andere paal
2. er mag nooit een grotere schijf op een kleinere schijf geplaatst worden
3. alle beschikbare palen mogen voor tussenopslag gebruikt worden.

Die monniken zijn gelukkig nog bezig, want volgens de legende zou bij het beëindigen van hun taak het einde van de wereld intreden.

De oplossing voor dit probleem kan recessief geformuleerd worden:

- verplaats een toren met 63 schijven van A naar B;
- verplaats de 64-ste schijf van paal A naar paal C;
- verplaats de toren met 63 schijven van B naar C.

Volgens de opgelegde regels kan deze toren van 63 schijven niet zo maar verplaatst worden. Echter, een toren van 63 schijven verplaatsen van A naar B is gelijkaardig aan het verplaatsen van een toren van 64 schijven, maar is iets eenvoudiger:

- verplaats een toren met 62 schijven van A naar C;
- verplaats de 63-ste schijf van paal A naar paal B;
- verplaats de toren met 62 schijven van C naar B.

Zo kan men constateren dat het probleem verlegd is tot het verplaatsen van een toren van 62. Uiteindelijk komt men tot het probleem van het verleggen van 1 schijf. Dit is het triviaal geval en kan uitgevoerd worden. Dit start de uitvoering, zodat men tenslotte die toren van 63 kan verplaatsen. Zo gaat het proces verder tot de volledige uitvoering ervan.

Om het probleem van de torens van Hanoi algemeen op te lossen, is een procedure nodig die een toren van n schijven verplaatst van de 'van'-paal naar de 'doel'-paal, gebruik makende van een 'hulp'-paal als tussenopslagplaats. De procedure zal dus 4 waardeparameters hebben waarvan de waarden door het systeem (automatisch) bij elke oproep onthouden worden. Bij een recursieve oproep zal het systeem ook bijhouden waar het gekomen was in de vorige oproep van het programma. Dit gebeurt aan de hand van een terugkeeradres. Zo een terugkeeradres geeft aan waar er moet worden verder gegaan na het beëindigen van de recursieve oproep. Aangezien deze procedure twee recursieve oproepen heeft, bestaan er ook twee terugkeeradressen. Er is ook een terugkeeradres in het hoofdprogramma dat gebruikt wordt voor de eerste oproep van de procedure.

```
PROCEDURE verplaatstoren (n,van,hulp,doel : Integer);
BEGIN
  IF n=1 THEN
    Writeln('neem de schijf boven ',van,'en verplaats naar ',doel)
  ELSE
  BEGIN
    verplaatstoren(n-1,van,doel,hulp);
    Writeln('neem de schijf boven ',van,'en verplaats naar ',doel);
    verplaatstoren(n-1,hulp,van,doel)
  END;
END;
```

Voor het verplaatsen van een toren van 3 schijven zijn er (2 tot de macht 3) - 1 manipulaties nodig. Voor een toren van 64 elementen zijn er (2 tot de macht 64) - 1 manipulaties nodig. Gezien het verplaatsen van n enkele schijf meerdere dagen in beslag neemt zijn de monniken in Hanoi nog steeds bezig met het oplossen van het probleem ... Het geheugengebruik van de computer kan bij

recursieve procedures niet op voorhand door de compiler berekend worden. Het voorbeeld van de torens toont dat de stapel groeit tot een zeker maximum om dan weer af te nemen. Bij sommige problemen is het mogelijk dat er te weinig geheugenruimte beschikbaar is om de volledige stapel op te slaan. In dat geval treedt een uitvoeringsfout (stack overflow) op.

2.3.2 Fibonacci

$$f(n) = \begin{cases} 0 & \text{voor } n = 0 \\ 1 & \text{voor } n = 1 \\ f(n-2) + f(n-1) & \text{voor } n > 1 \end{cases}$$

2.4 Voor- en nadelen van recursie

Het gebruik van recursie heeft heel wat voordelen:

- voor sommige problemen is het een veel natuurlijkere oplossing
- het is meestal gemakkelijker om correctheid te bewijzen (via inductie, directe vertaling van wiskundige formule)
- het is gemakkelijker te analyseren

Natuurlijk zijn er ook nadelen:

- bij sommige, vooral oudere, compilers is dit traag
- slecht geschreven kan dit een explosie van oproepen veroorzaken (zie Fibonacci)
- men moet zo weinig mogelijk parameters doorgeven en zeker geen grote datstructuren want bij elke oproep moet daarvoor geheugen worden voorzien

2.5 Backtracking

Backtracking is een verzamelnaam voor algoritmen die stap voor stap een mogelijke oplossing opbouwen. Als duidelijk wordt dat de partiële kandidaatoplossing onmogelijk tot een goede oplossing kan leiden, doet men een stap terug (vandaar de naam backtracking) en probeert men een andere oplossing op te bouwen. Afhankelijk van wat er precies wordt gevraagd kan men alle mogelijke oplossingen genereren of stoppen bij de eerste of rekenen tot een bepaalde tijd is verstreken, ... Het is één van de meest gebruikte tools om allerlei problemen (puzzels) op te lossen. Om dit te implementeren gebruikt men meestal recursie.

Het klassieke voorbeeld is het 8 koninginnen-probleem. Hier moet men 8 koninginnen op een schaakbord (8*8) plaatsen zodat zij elkaar niet kunnen aanvallen (niet op dezelfde rij, kolom of diagonaal). Men gaat eerst een koningin plaatsen in de eerste rij en in de eerste kolom. Dan gaat men naar de 2de rij om er alle mogelijkheden uit te testen: kolom 1 kan niet dus moet men niet meer verder bekijken, kolom 2 kan ook niet, kolom 3 wel dus dan gaan we dieper naar de 3de koningin enz.

Daarna beginnen we terug met rij 1 en plaatsten de eerste koningin op kolom 2 en bekijken we de andere koninginnen op de andere rijen enz.

Het verschil met brute-force is dat bij deze laatste alle volledige kandidaten worden gegenereerd en beoordeeld. Backtracking kan (als goed toegepast) reeds vlug grote hoeveelheden kandidaten schrappen.

Hoofdstuk 3

Algoritmen en computerberekeningen

3.1 Inleiding

Bij het schrijven van een programma is niet alleen de correctheid van het programma van belang. Als men bijvoorbeeld jaren moet wachten op het antwoord, dan is het programma ongeschikt. In dit hoofdstuk wordt dieper ingegaan op de uitvoeringstijd en het vereiste geheugen van algoritmen. Ook worden de problemen aangekaart die het gevolg zijn van de beperkte voorstelling van reële getallen.

3.2 Algoritmen

Definitie: Een **algoritme** is een eindige verzameling van instructies dat een bepaalde taak uitvoert. Een algoritme moet voldoen aan volgende eisen:

1. input: 0 of meer grootheden worden extern toegevoerd (van de buitenwereld naar het algoritme)
2. output: minimum 1 grootheid wordt geproduceerd
3. bepaaldheid: elke instructie moet duidelijk en ondubbelzinnig zijn
4. eindigheid: voor alle mogelijke gevallen moet het algoritme stoppen na een eindig aantal stappen
5. effectief: elke instructie moet voldoende eenvoudig zijn, zodat het met potlood en papier uit te rekenen is.

Een algoritme kan op verschillende manieren beschreven worden. Men kan dit in een natuurlijke taal (Nederlands, Engels) beschrijven, maar men moet dan heel goed opletten met voorwaarde 3 (bepaaldheid). Veelal wordt gekozen om algoritmen te beschrijven in pseudo-code. Bij complexe algoritmen is het vaak duidelijker als men het algoritme grafisch voorstelt. Hiervoor gebruikte men vroeger flowcharts. Nu gebruikt men hier beter een activiteitendiagramma voor (moderne en standaard versie van flowchart).

Een algoritme beschrijft hoe de data van input naar output moet getransformeerd worden. Een programma bevat de data EN deze transformatie van data (dus algoritme+data).

Heel belangrijk bij de ontwikkeling van algoritmen is de correctheid ervan. Dit wil zeggen: worden er voor alle mogelijke inputs correcte outputs gegenereerd? De beste manier om de correctheid van een algoritme te bepalen is om dit formeel wiskundig te bewijzen. Een alternatief is het algoritme implementeren en het uit te testen voor een representatieve set van inputs.

Naast de correctheid van een algoritme is ook de efficiëntie belangrijk. Hoeveel geheugen en hoeveel uitvoeringstijd is er nodig om het algoritme op de data uit te voeren? Het heeft weinig zin om een algoritme te gebruiken dat correct is, maar dat 50 jaar duurt of TBytes intern geheugen nodig heeft.

Voor het ontwerpen van een algoritme gaat men eerst zorgen dat het eenvoudig te begrijpen is. Na implementatie en testen kan dan het gebruik van resources geoptimaliseerd worden of moet men een andere complexer algoritme ontwerpen. Het eerste kan dan gebruikt worden om de correctheid van het complexere algoritmen en hun implementaties te verifiëren.

3.3 Analyse van algoritmen

3.3.1 Inleiding

Om te beoordelen of een algoritme zal voldoen, moet de uitvoeringstijd en het benodigd geheugen worden bepaald. Een eerste manier is a posteriori: het algoritme wordt geïmplementeerd, het programma wordt uitgevoerd en de uitvoeringstijd en het gebruikte geheugen wordt gemeten. Aan deze methode zijn er een aantal grote nadelen verbonden. Vooreerst vraagt het een grote inspanning om een algoritme te implementeren. Ook is het misschien niet doenbaar binnen een redelijke tijd en met een beperkte geheugengrootte. Zelfs als het doenbaar is, dan zijn de metingen afhankelijk van de gebruikte hardware, compiler, programmeertaal, programmeervaardigheden, belasting van het systeem, ...

Een tweede en betere manier is a priori, waarbij een schatting zal gebeuren. Voor de nodige tijd gaat men daarbij het aantal basisbewerkingen voor een bepaalde inputgrootte bepalen. De basisbewerkingen zijn bv. een optelling, een toekenning, een inleesoperatie, ed. De inputgrootte stelt bijvoorbeeld het aantal namen dat ingelezen en gesorteerd moet worden, of het aantal cijfers van π dat bepaald moet worden. Tel het aantal basisbewerkingen in onderstaande codefragmenten.

```
int res = 0;
for (int i=0;i<=n;i++) res += i*i*i;
```

```
for (int i=0;i<n;i++)
  for (int j=0;j<n;j++) System.out.println(i*j);
```

3.3.2 Asymptotische analyse van uitvoeringstijd

Nu is men meestal niet geïnteresseerd in het juiste aantal bewerkingen. Stel bijvoorbeeld dat het aantal basisbewerkingen gelijk is aan

$$10n^3 + n^2 + 40n + 80$$

Als n=1000 en men enkel de hoogste term neemt ($10n^3$) dan is de fout maar 0,01%. Vandaar dat we steeds een asymptotische analyse maken (enkel grootste orde-term). Als n klein is, is er weinig verschil tussen de beste en slechtste algoritmen en neem je altijd het eenvoudigste.

De constante horende bij de hoogste term is enkel van belang als er algoritmen met dezelfde orde worden vergeleken. In het voorbeeld is het $O(n^3)$

Voor algoritmen kan men spreken van average (gemiddeld), worst case (slechtste) en best case (beste) gedrag. Voor sommige toepassingen (real time toepassingen) is het worst case gedrag enorm belangrijk, zoals bv voor een luchtverkeergeleiding waarbij het slechtste geval is dat alle vliegtuigen uit dezelfde richting komen. Voor andere toepassingen is vooral het typisch, dus average gedrag belangrijk.

$O(1)$	constant	1 of paar keer
$O(n)$	lineair	kleine hoeveelheid verwerking per inputelement
$O(n^2)$	kwadratisch	paren van data, dubbele lus
$O(n^3)$	kubisch	verwerken van triples
$O(\log n)$	logaritmisch	probleem opgesplitst in kleinere deelproblemen
$O(n \log n)$		probleem opsplitsen, oplossen en combineren
$O(2^n)$	exponentieel	weinig praktisch, "brute-force-oplossing

Een enkel lus is $O(n)$, een dubbele lus is $O(n^2)$. Als de buitenste lus een lusvariabele i heeft die van 1 t.e.m. n gaat, en de binnenste lus tot i gaat, dan is het aantal evenredig met $\frac{n(n+1)}{2}$. Als er als stap $i = i * 2$ wordt gebruikt dan is het $O(n \log n)$.

In onderstaande tabel vindt men een paar waarden voor de meeste courante ordes.

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	2147483648

Als het algoritme recursief is wordt het probleem steeds herleid tot een eenvoudiger maar analoog probleem. Voor de analyse van dergelijke problemen zijn onderstaande formules handig.

- steeds 1tje eenvoudiger ($5 \to 4 \to 3 \to 2 \to 1$):

$$C_n = C_{n-1} + n = C_{n-2} + n - 1 + n = \ = \frac{n(n+1)}{2} \Rightarrow O(\frac{n^2}{2})$$

- input halveren in 1 stap:

$$C_n = C_{\frac{n}{2}} + 1 \Rightarrow O(logn)$$

- input halveren maar elk item bekijken:

$$C_n = C_{\frac{n}{2}} + n \Rightarrow O(2n)$$

- lineair door input en splitsen in 2 halven (divide-and-conquer):

$$C_n = 2 * C_{\frac{n}{2}} + n \Rightarrow O(n \log n)$$

- splitsen in 2 halven met 1 stap:

$$C_n = 2 * C_{\frac{n}{2}} + 1 \Rightarrow O(2n)$$

3.3.3 Bepalen van nodige geheugen

Voor het bepalen van het benodigde geheugen worden de gebruikte datastructuren bekeken. Het aantal integers, doubles en karakters, ... wordt bepaald en dan omgerekend naar bytes. Vooral meerdimensionale arrays zullen in de praktijk voor problemen zorgen.

Heel vaak geldt het tijd-ruimte-tradeoff-principe. Een algoritme dat minder geheugen vraagt zal meer tijd in beslag nemen en een sneller algoritme zal meer geheugen nodig hebben. Enkele voorbeelden hiervan zijn

- informatie comprimeren zodat minder geheugen nodig, maar comprimeren en decomprimeren kost tijd
- tussenresultaten opslaan i.p.v steeds opnieuw te berekenen (sneller maar groter)

3.3.4 A posteriori meten

Men kan de tijd die een algoritme nodig heeft gaan meten door voor en na de huidige tijd op te vragen en dan het verschil te nemen.

Voor de meeste programmeertalen zijn profilers beschikbaar die deze metingen meer gedetailleerd kunnen uitvoeren en rapporten.

3.3.5 Oefeningen

1. analyse n!
2. analyse torens van Hanoi
3. analyse + verbeteren Fibonacci

3.4 Optimalisatie van code

Een mogelijkheid om een programma sneller te maken is door de code te optimaliseren. Dit is echter minder belangrijk dan de keuze van het beste algoritme (of de ontwikkeling van een nieuw). Toch kan optimalisatie van de code een belangrijke tijdswinst betekenen. Omdat de code later nog moet onderhouden worden, moet men bij de optimalisatie er wel voor zorgen dat de code niet onleesbaar wordt. Hoe er geoptimaliseerd moet worden is afhankelijk van de computer (hardware en besturingssysteem) waar het programma op moet draaien. Het is ook niet economisch om alles te optimaliseren. Via een profiler kan men te weten komen welke delen de meeste tijd opslorpen, zodat dan enkel deze delen worden aangepakt. Volgens het Pareto-principe gebruiken 20% van operaties 80% van resoures. In software is het nog extremer: 90% van uitvoeringtijd door uitvoeren van 10% van de code

Aangezien de huidige compilers vaak zelf al goed optimaliseren, moet men heel goed weten wat deze doen. Anders kan men de optimaliserende compilers tegenwerken.

3.5 Ontwerp van algoritmen

Het eerste algoritme waar men meestal aan denkt zijn exhaustieve zoekmethoden. Hierbij worden alle mogelijkheden uitgeprobeerd. Deze brute-force algoritmen zijn vaak eenvoudig maar zeer inefficiënt.

Beter is om te kiezen voor een verdeel en heers (divide and conquer) strategie. Hierbij wordt het probleem opgesplitst in 2 deelproblemen en doet men dit recursief verder (ev. terug samenbrengen).

Als men het probleem opsplitst in een aantal fasen en men per fase een (lokaal optimale) beslissing neemt, zonder naar de andere fasen te kijken, dan spreekt men van "greedy"(gretige) algoritmen. Men heeft dan niet noodzakelijk het optimale resultaat, maar een goede benadering is in vele situaties aanvaardbaar.

Bij het selecteren of ontwerpen van een algoritme kiest men best eerst voor het eenvoudigste. In sommige gevallen is dit voldoende. In andere gevallen is een meer gecompliceerd algoritme nodig. Het eenvoudige kan dan gebruikt worden om de correctheid van het ingewikkelde te beoordelen. Bij de keuze van een algoritme moet men steeds het volgende afwegen:

- de tijd nodig om het algoritme te implementeren
- de snelheidswinst door een complexer algoritme
- het aantal keer dat men het programma gaat gebruiken

3.6 Computerberekeningen

3.6.1 Inleiding

Er is een verschil tussen

- methoden uit de analyse, bv (met x in radialen)

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

- methoden uit de numerieke analyse (benaderingen)

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}$$

$$|fout| < \left|\frac{x^9}{9!}\right|$$

- computerberekeningen: de getalvoorstelling beïnvloedt de berekening van elke term en het geheel

3.6.2 Voorbeelden

Voorbeeld 1

Bepaal de wortels van onderstaande vierkantsvergelijking met een nauwkeurigheid van 4 decimalen. Herhaal dit met 8 en 2 decimalen.

$$x^2 + 0,4002x + 0,0008 = 0$$

Voorbeeld 2

Als volgend stelsel

$$\begin{aligned} 5x - 331y &= 3,5 \\ 6x - 397y &= 5,2 \end{aligned}$$

exact wordt opgelost, dan bekomt men

$$\begin{aligned} x &= 331,7 \\ y &= 5,0 \end{aligned}$$

Verandert men in de 2de vergelijking 5,2 door 5,1 (2% verandering) dan bekomt met (exact)

$$\begin{aligned} x &= 198,6 \\ y &= 4,5 \end{aligned}$$

y is hierbij 10% verandert.

Voorbeeld 3

Dit voorbeeld illustreert het gevolg van de genormaliseerde voorstelling van reële getallen in de computer. Met 4 beduidende cijfers:

$$(4000 + 0,2511) + 0,4231 = 4000$$

$$4000 + (0,2511 + 0,4231) = 4001$$

Voorbeeld 4

$$\begin{aligned} x_1 &= 0,5243.10^0 \\ x_2 &= 0,5262.10^0 \\ x_3 &= 0,5226.10^0 \\ x_4 &= 0,5278.10^0 \end{aligned}$$

Tel deze 4 getallen op met een nauwkeurigheid van 4 cijfers op verschillende manieren en vergelijk met de exacte oplossing.

Voorbeeld 5

$$\begin{aligned} x &= 0,5628 \pm 0,0001 \\ y &= 0,5631 \pm 0,0001 \\ x - y &= 0,0003 \pm 0,0002 \end{aligned}$$

De relatieve fout is hierbij opgelopen van 0,02% tot 67%.

Hoofdstuk 4

Sorteer- en zoekalgoritmen

4.1 Sorteren

Sorteren is een essentiële activiteit die heel veel gebruikt wordt binnen allerlei toepassingen. Er zijn tal van verschillende oplossingen voor dit probleem ontwikkeld. Op zich is sorteren een eenvoudige actie: een rij elementen kan geordend worden van klein naar groot of omgekeerd. Wat sorteren zo boeiend maakt zijn de verschillende uitgangsposities, zoals de omvang van de te sorteren rij, het toevoegen van elementen aan een gesorteerde rij, het type van de te sorteren elementen,... Dit resulteert in zeer uiteenlopende benaderingswijzen en verrassende resultaten.

Wanneer de omvang van de te sorteren elementen zo groot is dat deze niet meer in het intern geheugen kan geplaatst worden, zal men extern sorteren toepassen. Hieronder worden algoritmen verstaan die bij het sorteren zowel van het extern als het intern geheugen gebruik maken. Door de evolutie op niveau van hardware en software wordt deze techniek minder aangewend en zal deze in deze cursus niet besproken worden. Er worden dus enkel technieken behandeld waarbij het sorteren volledig in het intern geheugen gebeurt (intern sorteren), onafgezien van het feit of het resultaat al dan niet wordt weggeschreven op een extern medium.

De sorteeralgoritmen worden voor de eenvoud uiteengezet aan de hand van een rij getallen, maar zijn toepasbaar op iedere telbare en eindige rij elementen. Het vergt slechts een kleine aanpassing om de algoritmen aan te passen voor het sorteren van arrays van objecten.

Een belangrijk aspect bij het sorteren is het vergelijken van 2 elementen. Indien men werkt met rijen van getallen, dan kan men gebruik maken van de standaard vergelijkings-operatoren. Voor het vergelijken van objecten zal men zelf methoden moeten schrijven.

4.1.1 Bubblesort

Bubblesort is een eenvoudig algoritme dat gebruikt wordt om kleine rijen te sorteren. De tijd dat het algoritme nodig heeft is evenredig met het kwadraat van het aantal elementen. Vandaar dat deze methode niet geschikt is voor grote rijen.

Voor het sorteren van klein naar groot verloopt het algoritme als volgt:

- tijdens het doorlopen van de rij wordt elk element vergeleken met het volgende element in de rij. Staan deze elementen niet in volgorde (het eerste is groter dan het tweede als men sorteert van klein naar groot), dan worden ze verwisseld.
- wanneer de rij één keer doorlopen is, bevindt het grootste element zich achteraan in de rij

- vervolgens wordt het ongesorteerde deel (=vorige rij, zonder het laatste element) opnieuw doorlopen. Hierdoor komt het tweede grootste element op de juiste plaats te staan.
- dit wordt herhaald tot alle elementen in volgorde staan, en er dus geen omwisselingen meer nodig zijn. In het slechtste geval is dit N-1 keer, waarbij N het aantal elementen van de rij voorstelt.

In het volgende programmafragment vindt men de Java-code terug (getallen is een array van int's).

```
int hulp;
for(int i = getallen.length; i > 1; i--){
  for(int j = 0; j < i - 1; j++){
    if (getallen[j] > getallen[j+1]){
      hulp = getallen[j];
      getallen[j] = getallen[j+1];
      getallen[j+1] = hulp;
    }
  }
}
```

4.1.2 Selectionsort of Swapsort

Bij selectionsort doorloopt men de array op zoek naar het grootste element. Dit verwisselt men dan met het element op de laatste plaats. Net zoals bij bubblesort heeft men nu een rij met het grootste element achteraan, en waarvan de rest van de rij ongesorteerd is. Hetzelfde zal men nu herhalen op de rij telkens 1 kleiner.

```
int tmp;
int maxpos;
for (int laatste=getallen.length;laatste>1;laatste--){
  maxpos=0;
  for (int el=1;el<laatste;el++)
      if (getallen[el]>getallen[maxpos]) maxpos=el;
  tmp=getallen[maxpos];
  getallen[maxpos]=getallen[laatste-1];
  getallen[laatste-1]=tmp;
}
```

4.1.3 Cardsort of Insertion sort

Hierbij worden elementen gesorteerd zoals een kaarter dit doet. Een eerste kaart wordt in handen genomen. Voor een volgende kaart wordt de rij van het begin doorlopen en wordt de kaart op de juiste plaats tussengevoegd. Alle elementen groter dan het nieuwe worden dus een positie opgeschoven. Dit is een techniek die zeer veel verplaatsingen vergt en voor grotere reeksen resulteert in ontoelaatbare uitvoeringstijden. Wanneer men echter gebruik kan maken van dynamische gegevensstructuren (gelinkte lijsten) is deze techniek wel handig. Ook als de rij reeds bijna gesorteerd is, is dit bruikbaar.

Het algoritme is als volgt:

```
Program inlassen (input, output);
Const   n = 10;
Type    rij = Array [1..n] Of Integer;

Var waarde, pos, i : Integer;
    r : rij;
    ok : Boolean;

Begin
  For i := 1 To n Do
  Begin
    Write('Geef waarde ', i, ' : '); Readln(waarde);
    pos := i-1; ok := true;
    While ok and (pos >= 1) Do
    Begin
        If waarde >= r[pos]
        Then ok := false
        Else
        Begin
          r[pos + 1] := r[pos];
          pos := pos - 1
        End
    End;
  If not(ok)
  Then r[pos + 1] := waarde
  Else r[1] := waarde;
  {uitschrijven resultaat}
End.
```

4.1.4 Shellsort

Shellsort (genoemd naar de ontwerper van het algoritme) wordt vooral toegepast bij grotere reeksen. Hierbij wordt een element vergeleken met een element dat h plaatsen verder staat. Nadat de rij zo doorlopen is wordt een nieuwe h-waarde gekozen en wordt de werkgang hernomen. Dit wordt hernomen tot en met h = 1. Welke h-waarden men het best neemt, is moelijk te bepalen. Heel vaak wordt de formule $h_i = h_{i-1} * 3 + 1$ genomen en start men met de hoogst mogelijke $h < n$.

Heel vaak wordt ook onderstaande code gebruikt:

```
void shellsort (int[] a, int n)
{
    int i, j, k, h, v;
    int[] cols = {1391376, 463792, 198768, 86961, 33936, 13776, 4592,
                  1968, 861, 336, 112, 48, 21, 7, 3, 1}
    for (k=0; k<16; k++){
```

```
        h=cols[k];
        for (i=h; i<n; i++){
            v=a[i];
            j=i;
            while (j>=h && a[j-h]>v){
                a[j]=a[j-h];
                j=j-h;
            }
            a[j]=v;
        }
    }
```

4.1.5 Mergesort

Bij mergesort wordt de rij in 2 delen opgesplitst. Eerst wordt de linkerdeelrij gesorteerd (door opnieuw, recursief, mergesort te gebruiken) en daarna de rechterdeelrij (analoog). Als deze beiden gesorteerd zijn, is het geheel echter nog niet gesorteerd. Beide deelrijen moeten nog correct samengevoegd worden, vandaar de naam mergesort. Dit samenvoegen gebeurt door 2 lopers te plaatsen, 1 bij het begin van elke deelrij. Men vergelijkt de waarden waarnaar beide lopers verwijzen. De kleinste waarde plaatst men in een hulprij (waarvan de index ook steeds verhoogt wordt) en de loper die naar die waarde verwees wordt 1 plaats opgeschoven. Dit wordt herhaald tot 1 van beide deelrijen uitgeput is. De overige elementen van de andere deelrij worden dan naar de hulprij gekopiëerd.

```
private void merge(int[] getallen,int onder,int midden,int boven){
  int i=onder,j=midden+1,k=onder;
  int[] hulprij=new int[getallen.length];

  while ((i<=midden) && (j<=boven)){
    if (getallen[i]<getallen[j]) hulprij[k++]=getallen[i++];
    else hulprij[k++]=getallen[j++];
  }
  if (i>midden) for (int l=j;l<=boven;l++) hulprij[k++]=getallen[l];
  else for (int l=i;l<=midden;l++) hulprij[k++]=getallen[l];

  for (k=onder;k<=boven;k++) getallen[k]=hulprij[k];
}

private static void sort(int[] getallen,int onderindex,int bovenindex){
  if (onderindex<bovenindex){
    int midden=(onderindex+bovenindex)/2;
    sort(getallen,onderindex,midden);
    sort(getallen,midden+1,bovenindex);
    merge(getallen,onderindex,midden,bovenindex);
  }
}
```

```
public static void mergesort(int[] getallen,aantal){
  sort(getallen,0,aantal-1);
}
```

4.1.6 Quicksort

Bij mergesort wordt veel tijd verloren doordat beide deelrijen steeds samengevoegd moeten worden. Dit wordt bij quicksort vermeden. De rij wordt hierbij op zo'n manier gesplitst dat als beide deelrijen gesorteerd zijn, de volledige rij ook onmiddellijk gesorteerd is. Dit kan natuurlijk maar doordat voor de opsplitsing een aantal getallen in de rij verplaatst worden. De code van het algoritme vindt men hieronder. Enkel het gedeelte voor het sorteren is opgenomen.

```
private void sort(int onder,int boven){
  int i=onder,j=boven;
  int x=getallen[(onder+boven)/2]; //eigenlijk een willekeurig element

  do{
    while (getallen[i]<x) i++;
    while (getallen[j]>x) j--;
    if (i<=j){
      int tmp=getallen[i];
      getallen[i]=getallen[j];
      getallen[j]=tmp;
      i++;
      j--;
    }
  } while (i<=j);
  if (onder<j) sort(onder,j);
  if (i<boven) sort(i,boven);
}

public void quicksort(){
  sort(0,aantal-1);
}
```

4.2 Bibliotheken in C, C++, C# en Java

4.2.1 C

In C wordt voor het sorteren Quicksort gebruikt.

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <stdlib.h> /*qsort*/
```

```
#define MAX_AANTAL_LIJNEN 5000
#define MAX_LENGTE_LIJN 1000

int leesLijnen(char *lijnen[])
{int i=0; lijnen[0]=(char*)malloc(MAX_LENGTE_LIJN);
 while ((i<MAX_AANTAL_LIJNEN)&&
        (fgets(lijnen[i],MAX_LENGTE_LIJN,stdin)!=NULL))
 {i++;
  lijnen[i]=(char*)malloc(MAX_LENGTE_LIJN);
 }
 if (feof(stdin)) return i;
 return -1;
}

int vergelijk(const char **s1,const char **s2)
{return strcmp(*s1,*s2);
}

void sorteerLijnen(char *lijnen[],int aantal)
{int i;
 qsort((void *)lijnen,aantal,sizeof( char * ),vergelijk);
}

int main()
{char *lijnen[MAX_AANTAL_LIJNEN];
 int aantalLijnen,i;

 if ((aantalLijnen=leesLijnen(lijnen))>=0)
 {sorteerLijnen(lijnen,aantalLijnen);
  for (i=0;i<=aantalLijnen;i++) free(lijnen[i]);
  return 0;
 }
}
```

4.2.2 C++

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>

using namespace std;

class Persoon
```

```
{string naam;
 int leeftijd;

public:
        Persoon(string s,int l){naam=s; leeftijd=l;}
        string getNaam(){return naam;}
        int getLeeftijd(){return leeftijd;}
        void schrijf(){cout << naam << " " << leeftijd << endl;}
};

bool operator<(Persoon& p1,Persoon& p2)
{return p1.getLeeftijd()<p2.getLeeftijd();
}

bool isAlfabetisch(Persoon& p1,Persoon& p2)
{return p1.getNaam()<p2.getNaam();
}


int main()
{vector<Persoon> lijst;
 vector<Persoon>::iterator lijstIterator;

 Persoon p1("Koen",37),p2("Peter",31),p3("Annemie",38),p4("Mieke",28);
 lijst.push_back(p1);
 lijst.push_back(p2);
 lijst.push_back(p3);
 lijst.push_back(p4);

 cout << "origineel:" << endl;
 for (lijstIterator=lijst.begin();lijstIterator!=lijst.end();
      lijstIterator++)
   lijstIterator->schrijf();

 sort(lijst.begin(),lijst.end());
 cout << "gesorteerd:" << endl;
 for (lijstIterator=lijst.begin();lijstIterator!=lijst.end();
      lijstIterator++)
   lijstIterator->schrijf();

 sort(lijst.begin(),lijst.end(),isAlfabetisch);
 cout << "alfabetisch gesorteerd:" << endl;
 for (lijstIterator=lijst.begin();lijstIterator!=lijst.end();
      lijstIterator++)
   lijstIterator->schrijf();

  return 0;
```

```
}
```

4.2.3 C#

Ook hier wordt het QuickSort algoritme gebruikt. In het slechtste geval is dit $O(n^2)$ (n = aantal elementen), het gemiddelde gedrag is $O(n \log n)$

De implementatie voert een onstabiele sortering uit: als 2 elementen gelijk zijn, dan kunnen deze toch van plaats wisselen. Bij een stabiele sortering blijft de volgorde van gelijke elementen bewaard.

Het volgende voorbeeld vindt men terug in de documentatie van Visual .NET (C#).

```
// The following example shows how to sort the values in an array
// using the default comparer
// and a custom comparer that reverses the sort order.

using System;
using System.Collections;

public class SamplesArray  {

  public class myReverserClass : IComparer  {
    // Calls CaseInsensitiveComparer.Compare with parameters reversed.
    int IComparer.Compare( Object x, Object y )  {
        return( (new CaseInsensitiveComparer()).Compare( y, x ) );
    }
  }

  public static void Main()  {

    // Creates and initializes a new Array and a new custom comparer.
    String[] myArr = { "The", "QUICK", "BROWN", "FOX", "jumps", "over",
                       "the", "lazy", "dog" };
    IComparer myComparer = new myReverserClass();

    // Displays the values of the Array.
    Console.WriteLine("The Array initially contains the following values:");
    PrintIndexAndValues( myArr );

    // Sorts a section of the Array using the default comparer.
    Array.Sort( myArr, 1, 3 );
    Console.WriteLine("After sorting a section using the default comparer:" );
    PrintIndexAndValues( myArr );

    // Sorts a section of the Array using the reverse case-insensitive comparer.
    Array.Sort( myArr, 1, 3, myComparer );
    Console.WriteLine("After sorting a section using the
       reverse case-insensitive comparer:");
```

```
    PrintIndexAndValues( myArr );

    // Sorts the entire Array using the default comparer.
    Array.Sort( myArr );
    Console.WriteLine( "After sorting the entire Array using
the default comparer:" );
    PrintIndexAndValues( myArr );

    // Sorts the entire Array using the reverse case-insensitive comparer.
    Array.Sort( myArr, myComparer );
    Console.WriteLine( "After sorting the entire Array using
the reverse case-insensitive comparer:" );
    PrintIndexAndValues( myArr );
  }

  public static void PrintIndexAndValues( String[] myArr )  {
    for ( int i = 0; i < myArr.Length; i++ )  {
      Console.WriteLine( "   [{0}]: {1}", i, myArr[i] );
    }
    Console.WriteLine();
  }
}

/*
This code produces the following output.

The Array initially contains the following values:
   [0] : The
   [1] : QUICK
   [2] : BROWN
   [3] : FOX
   [4] : jumps
   [5] : over
   [6] : the
   [7] : lazy
   [8] : dog

After sorting a section using the default comparer:
   [0] : The
   [1] : BROWN
   [2] : FOX
   [3] : QUICK
   [4] : jumps
   [5] : over
   [6] : the
   [7] : lazy
   [8] : dog
```

```
After sorting a section using the reverse case-insensitive comparer:
   [0] : The
   [1] : QUICK
   [2] : FOX
   [3] : BROWN
   [4] : jumps
   [5] : over
   [6] : the
   [7] : lazy
   [8] : dog

After sorting the entire Array using the default comparer:
   [0] : BROWN
   [1] : dog
   [2] : FOX
   [3] : jumps
   [4] : lazy
   [5] : over
   [6] : QUICK
   [7] : the
   [8] : The

After sorting the entire Array using the reverse case-insensitive comparer:
   [0] : the
   [1] : The
   [2] : QUICK
   [3] : over
   [4] : lazy
   [5] : jumps
   [6] : FOX
   [7] : dog
   [8] : BROWN

*/
```

4.2.4 Java

De voorziene sorteermethoden in Java maken een onderscheid tussen het sorteren van primitieve datatypes (int, double, char, ...) en het sorteren van referentietypes:

- `public static void sort(int[] a)`: sorteert de gespecifiëerde array volgens stijgende numerische waarde. Het algoritme dat gebruikt wordt is een aangepaste quicksort (Jon L. Bentley and M. Douglas McIlroy's "Engineering a Sort Function", Software-Practice and Experience, Vol. 23(11) P. 1249-1265 (November 1993)). Dit algortime gedraagt zich beter ($O(n \log n)$) op vele data die bij andere quicksort implementaties een kwadratische performantie ($O(n^2)$) leveren.

- `public static void sort(Object[] a)`: sorteert de gespecifiëerde array volgens stijgende orde, volgende de natuurlijke ordening van de elementen. Alle elementen in de array moeten de Comparable interface implementeren en moeten mutueel vergelijkbaar zijn. Dit laatste wil zeggen dat e1.compareTo(e2) steeds mogelijk moet zijn (geen ClassCastException opwerpen) voor alle elementen e1 en e2 in de rij. Het is een stabiele sortering (gelijkewaardige elementen blijven hun volgorde bewaren). Het gebruikte algoritme is een aangepaste mergesort (de merge wordt niet uitgevoerd als de hoogste waarde in de eerste lijst kleiner is dan dan de kleinste waarde in de tweede lijst). Dit algoritme garandeert een ($O(n \log n)$ performantie.

```
import java.util.Arrays;

Arrays.sort(rij);
```

Als men objecten wil vergelijken moet de betrokken klasse (of klassen binnen een overervingshiërarchie) de interface Comparable implementeren zoals in onderstaande codefragmenten (voor en vanaf Java 5).

```
public class X implements Comparable{
 ...
 public int compareTo(Object o){
  X tmp=(X)o;
  ...
  return verschil;
 }
}
```

```
public class X implements Comparable<X>{
 ...
 public int compareTo(X ander){
 //geen conversie meer naar type X
  ...
 }
}
```

4.3 Zoeken

4.3.1 Sequentieel

```
public static int sequentieelZoeken(int[] r,int getal)
 {int i=0;
  while ((i<r.length) && (r[i]!=getal)) i++;
  if (i==r.length) return -1;
  return i;
 }
```

4.3.2 Binair

```
public static int binairZoeken(int[] r,int getal)
 {int begin=0,eind=r.length,midden;
  int ret = -1;
  while (begin<eind)
  {midden=(begin+eind)/2;
   if (getal==r[midden]) ret = midden;
   if (getal<r[midden]) eind=midden-1;
   else begin=midden+1;
  }
  return ret;
 }
```

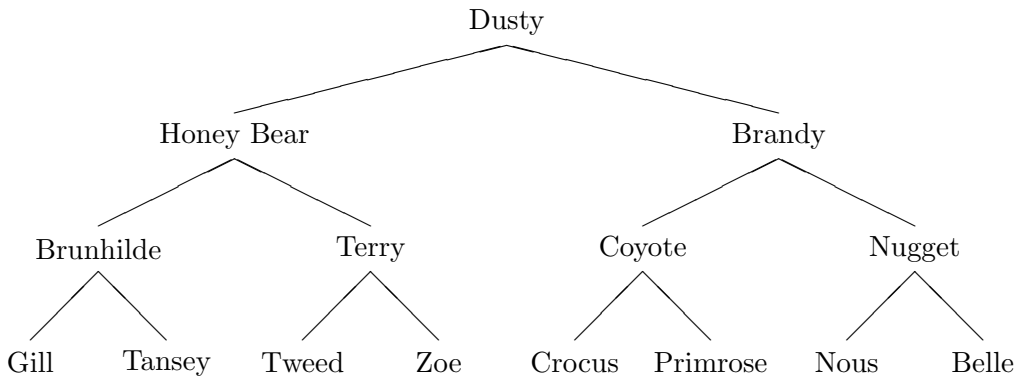
De methode binarySearch is voorzien in Java.

Hoofdstuk 5

Bomen

5.1 Inleiding

Dit hoofdstuk behandelt een zeer belangrijke datastructuur namelijk de **boom** (*tree*). Intuitief kan men aanvoelen dat de gegevens zo georganiseerd zijn dat de informatie gerelateerd is via takken. Twee voorbeelden uit de genealogie illustreren het begrip boom. Figuur 5.1 toont een stamboom. Figuur 5.2 toont de afkomst van de moderne Europese talen.



Figuur 5.1: Een stamboom

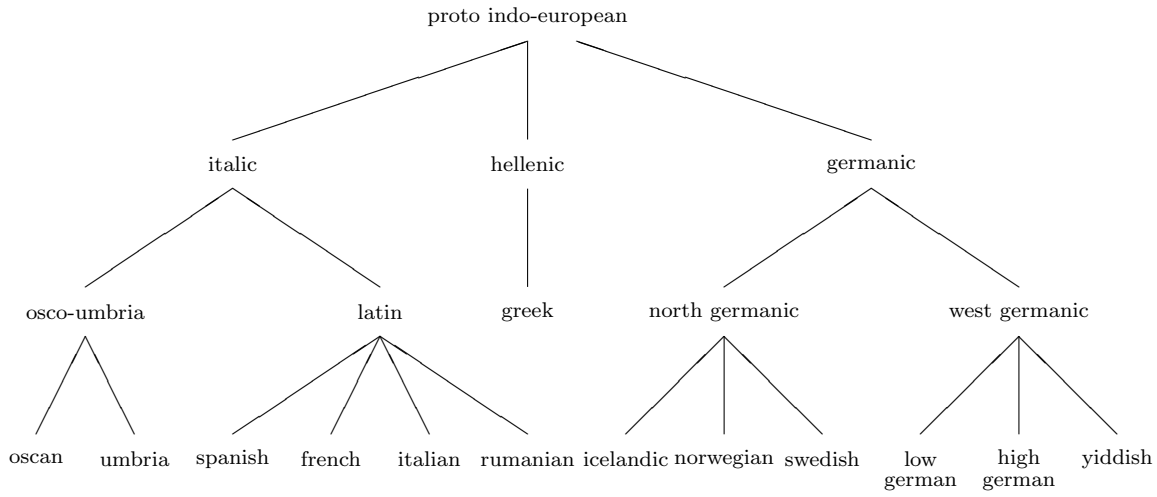
In de volgende sectie gaan we het begrip boom formeel definiëren.

5.2 Definities

definitie: Een boom is een eindige verzameling van één of meerdere knopen zodat

1. er een speciale knoop is, de wortel (*root*) genaamd en
2. de overblijvende knopen verdeeld worden in n, $n \geq 0$, disjuncte (niet-overlappende) verzamelingen $T_1 \ldots T_n$, waarbij elk van deze verzamelingen een boom is. $T_1 \ldots T_n$ noemt men de deelbomen (*subtrees*) van de wortel.

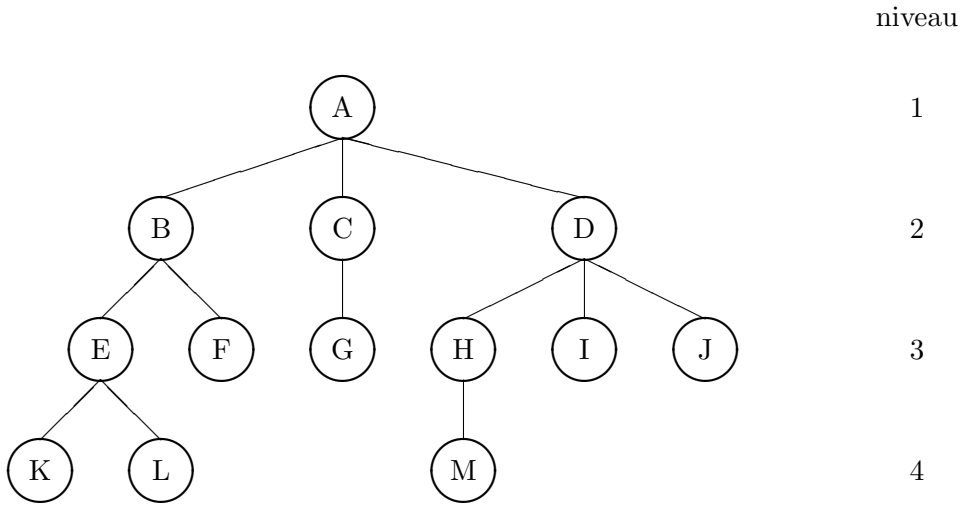
De bovenstaande definitie is een recursieve definitie. Toegepast op het voorbeeld uit figuur 5.2 is proto-indo-european de wortel van deze boom en zijn er drie deelbomen met wortels italic, hellenic en germanic.



Figuur 5.2: De afkomst van de moderne Europese talen

De voorwaarde dat de deelbomen $T_1 \ldots T_n$ niet mogen overlappen zorgt ervoor dat er geen verbindingen zijn tussen de deelbomen (geen kruisbestuiving). Als gevolg is elke knoop in een boom de wortel van een deelboom. In het voorbeeld is west germanic de wortel van een deelboom met drie deelbomen en yiddish de wortel van een boom zonder deelbomen.

Een knoop (*node*) is het geheel van informatie en takken naar andere informatie.



Figuur 5.3: Een voorbeeldboom

De boom uit figuur 5.3 heeft 13 knopen. Voor de eenvoud is wordt de informatie gewoon door een letter voorgesteld. De wortel A wordt gewoonlijk bovenaan getekend.

Het aantal deelbomen van een knoop wordt de graad (*degree*) genoemd. De graad van knoop A is 3, van C 1 en van F 0. Knopen met graad 0 worden ook bladeren (*leaf*) of terminale knopen genoemd. $\{K, L, F, G, M, I, J\}$ is de verzameling van bladerknopen. De andere knopen worden als niet-terminalen gerefereerd.

De wortels van de deelbomen van een knoop X zijn de kinderen (*children*) van X. X is de ouder (*parent*) van zijn kinderen. De kinderen van bijvoorbeeld knoop D zijn de knopen H, I en J. Knoop A is de ouder van knoop D. Kinderen van dezelfde ouder zijn broers (*siblings*). Zo zijn in het voorbeeld de knopen H, I en J broers.

De graad van een boom is de maximale graad van de knopen in de boom. De boom uit figuur 5.3 heeft graad 3.

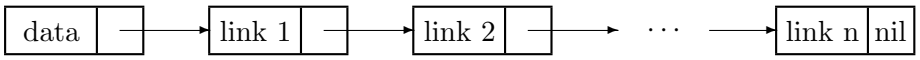
De voorouders (*ancestors*) van een knoop zijn alle knopen die op het pad liggen van de wortel van de boom naar die knoop. De voorouders van knoop M zijn A, D en H.

Het niveau (*level*) van een knoop wordt gedefiniëerd door eerst de wortel van de boom niveau 1 te geven en dan te definiëren dat als een knoop zich op niveau I bevindt, zijn kinderen zich op niveau I+1 bevinden.

De hoogte (*height*) of diepte (*depth*) van een boom is het maximum niveau van een knoop in die boom.

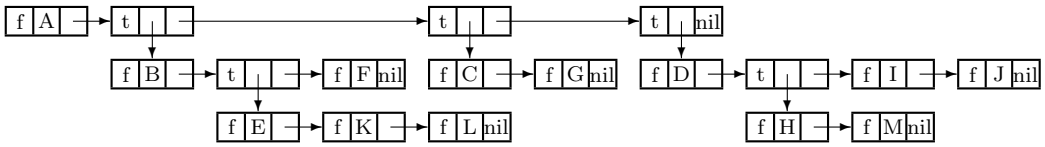
5.3 Opslag

Hoe kunnen we deze bomen in het geheugen van een computer opslaan? Voor elke knoop moeten we ruimte voorzien om de informatie behorende bij die knoop op te slaan. Ook moeten we verbindingen voorzien naar de kinderen van die knoop. Aangezien elke knoop een variabel aantal deelbomen kan hebben gebruiken we daarvoor gelinkte lijsten. Per knoop hebben we dus een lijst met de gegevens en de wijzers (*links*) naar de wortels van de deelbomen. Dit wordt voorgesteld in figuur 5.4.



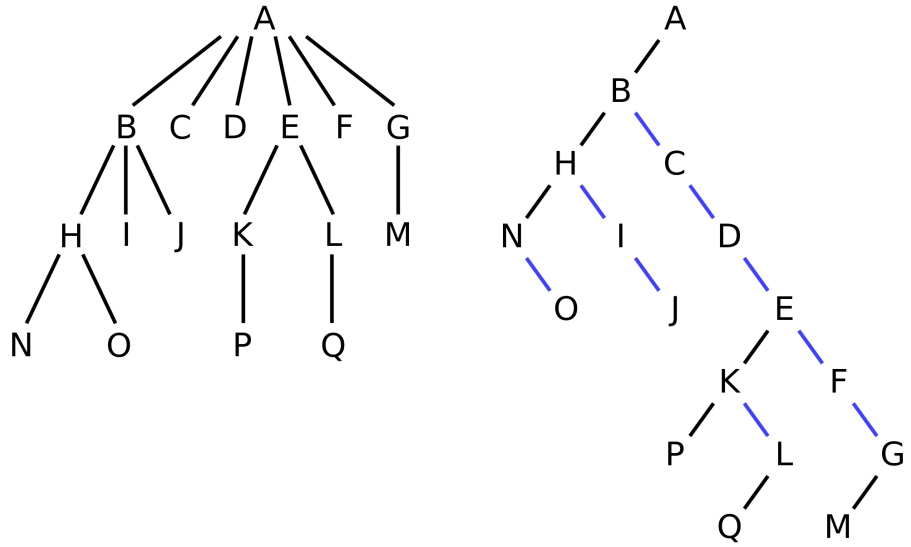
Figuur 5.4: Een gelinkte lijst per knoop

Voor de boom uit figuur 5.3 kan men de voorstelling uit figuur 5.5 gebruiken.



Figuur 5.5: Een voorstelling van een mogelijke opslag van de voorbeeldboom met variabele records/-structs

Een andere voorstelling is de *first child next sibling* voorstelling. Hierbij kan elke knoop een verwijzing bevatten naar zijn eerste kind en een verwijzing bevatten naar zijn volgende broer/zus. Een multi-way boom kan zo als een binaire boom (max 2 kinderen, zie verder) worden voorgesteld. Men noemt dit ook left child right sibling (LCRS). Dit wordt in figuur 5.6 geïllustreerd.



Figuur 5.6: first child next sibling

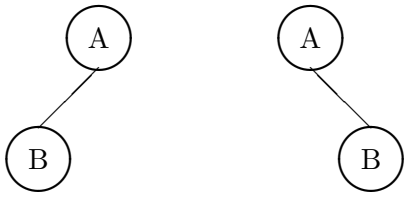
5.4 Binaire bomen

Een binaire boom is het belangrijkste en meestvoorkomende type van boomstructuur. Bij een binaire boom heeft elke knoop maximaal twee takken, d.w.z. dat er geen knopen zijn met een graad groter dan twee. Men kan hier ook spreken van een linkse en een rechtse deelboom waardoor er een ordening van de gegevens mogelijk is. Omdat een binaire boom ook geen enkele knoop kan hebben is het eigenlijk geen echte boom.

5.4.1 Definitie en eigenschappen

definitie: Een binaire boom is een eindige verzameling van knopen die ofwel leeg is ofwel bestaat uit een wortel en twee disjuncte (gescheiden) binaire bomen die de linkse deelboom en de rechtse deelboom worden genoemd.

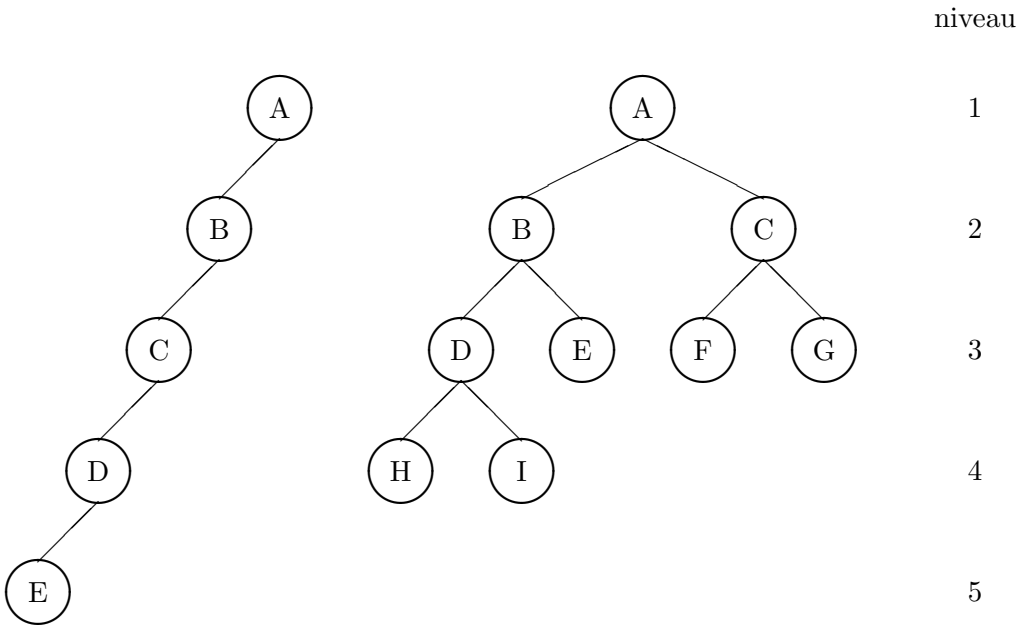
opmerking: De twee binaire bomen uit figuur 5.7 zijn verschillend. De linkse binaire boom heeft een lege rechtse deelboom terwijl de rechtse binaire boom een lege linkerdeelboom heeft. Als ze als algemene bomen bekeken worden (en niet als binaire bomen) dan zijn beide bomen gelijk.



Figuur 5.7: Twee verschillende binaire bomen

Figuur 5.8 toont enkele speciale binaire bomen.

De linkse is een scheve (*skewed*) binaire boom, meer specifiek een linkse scheve binaire boom (analoog bestaat er ook een rechtse scheve binaire boom). De rechtse boom uit de figuur is een



Figuur 5.8: Twee speciale binaire bomen

complete binaire boom (zie verder).

lemma 1:

- Het maximum aantal knopen op niveau i van een binaire boom is 2^{i-1}, als $i \geq 1$
- Het maximum aantal knopen in een binaire boom van diepte k is $2^k - 1$, als $k \geq 1$

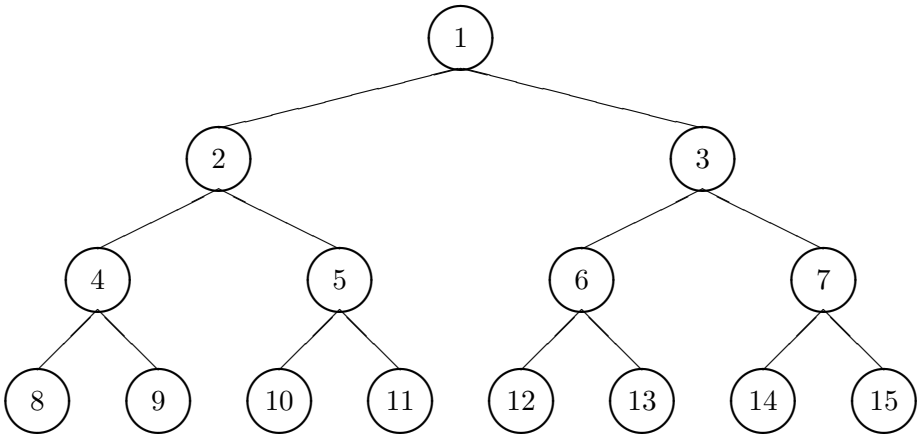
bewijs:

lemma 2: Voor elke niet-lege binaire boom, als n_0 het aantal terminale knopen is en als n_2 het aantal knopen met graad twee is dan is $n_0 = n_2 + 1$

bewijs:

5.4.2 Voorstelling van binaire bomen

Voor we de voorstelling van binaire bomen bespreken is het nodig om het begrip volle binaire boom te introduceren. Een volle (*full*) binaire boom van diepte k is een binaire boom van diepte k met $2^k - 1$ knopen. Dit is het maximum aantal knopen dat een binaire boom van die diepte kan hebben. Figuur 5.9 toont een volle binaire boom van diepte 4.



Figuur 5.9: Een volle binaire boom

Een sequentiële voorstelling van volle binaire bomen bekomt men door de knopen sequentiëel te nummeren startend met de knoop op niveau 1, dan deze op niveau 2 en zo verder. De knopen op één niveau worden van links naar rechts genummerd.

Aan de hand van deze definitie kunnen we een definitite geven van een **complete** binaire boom: een binaire boom met n knopen met diepte k is compleet als zijn knopen corresponderen met de knopen die van 1 tot n genummerd zijn in de volle binaire boom met diepte k.

De knopen kunnen nu opgeslagen worden in een ééndimensionale rij "tree"waarbij de gegevens horende bij knoop i worden opgeslagen in "tree[i]". Voor een complete binaire boom met n knopen ($diepte = log_2 n + 1$) kan de ouder, het linker- en het rechterkind van knoop i als volgt worden gevonden:

1. ouder(i) is op plaats i/2 als $i \neq 1$; als i=1 dan betreft het de wortel en deze heeft geen ouder.
2. linkerlid(i) is op 2i als $2i \leq n$. Anders heeft i geen linkerkind.
3. rechterkind(i) is op 2i+1 als $2i + 1 \leq n$. Anders heeft i geen rechterkind.

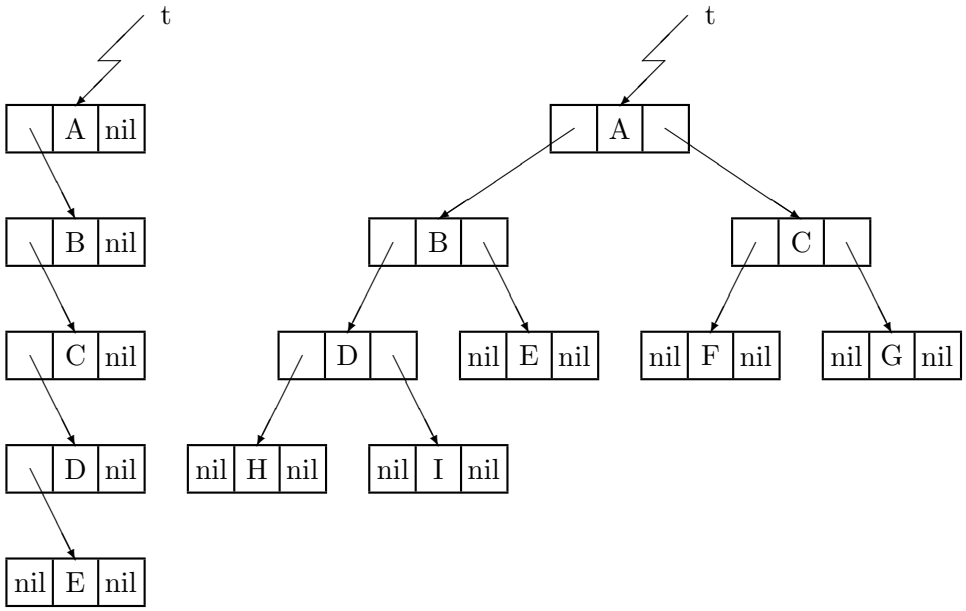
Deze voorstelling kan voor alle binaire bomen gebruikt worden, alhoewel er in de meeste gevallen veel geheugen ongebruikt zal zijn.

Een meer gebruikte voorstelling maakt gebruik van wijzers (pointers of referenties). Elke knoop heeft 3 velden: een wijzer naar het linkerkind, een veld voor de gegevens van de knoop en een wijzer naar het rechterkind. In Java wordt dit als volgt gedefiniëerd:

```
class Knoop{
  Knoop links;
  Data data;
```

```
  Knoop rechts;
}
```

Voor de bomen uit figuur 5.8 zal dit eruitzien zoals in figuur 5.10.



Figuur 5.10: Voorstellingen van binaire bomen met behulp van wijzers.

Met deze voorstelling is het wel moeilijk om de ouder van een knoop te bepalen. Als het in een toepassing nodig is om de ouder van een willekeurige knoop te kunnen bepalen dan kan men een vierde veld, nl. een wijzer naar de ouder, aan de recordstructuur toevoegen.

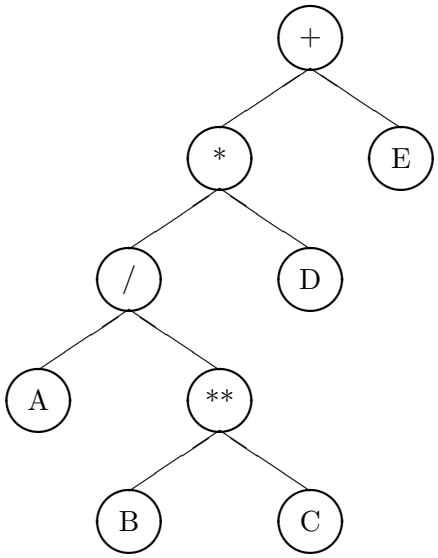
Een andere pointer-voorstelling is de *first child next sibling* voorstelling. Hierbij kan elke knoop een verwijzing bevatten naar zijn eerste kind en een verwijzing bevatten naar zijn volgende broer/zus. Deze voorstelling kan ook bij niet-binaire bomen gebruikt worden.

5.4.3 Doorlopen van binaire bomen

Er zijn verschillende operaties die we op binaire bomen willen uitvoeren. Eén hiervan is het doorlopen van de boom waarbij elke knoop éénmaal bezocht wordt. Bij het doorlopen van de boom willen we alle knopen op dezelfde manier behandelen. Op een knoop kan men naar links gaan (L), de gegevens uitschrijven (D) of naar rechts gaan (R). Er zijn zes mogelijke combinaties: LDR, LRD, DLR, DRL, RDL en RLD. Als we de conventie volgen om eerst naar links te gaan alvorens naar rechts te gaan, dan blijven er nog drie combinaties over: LDR, LRD en DLR. Deze worden respectievelijk inorder, postorder en preorder doorlopen genoemd omdat het overeenkomt met het genereren van de infix, postfix en prefix vormen van een uitdrukking. Als voorbeeld bekijken we de boom in figuur 5.11.

Bij het inorder doorlopen gaat men steeds naar links bewegen tot men niet verder kan. Dan wordt de knoop bezocht en dan beweegt men 1 knoop naar rechts. Dit wordt herhaald tot alle knopen bezocht zijn. Dit kan via de volgende recursieve procedure:

```
static void inorder(Knoop k) {
  if (k!=null) {
```



Figuur 5.11: Een uitdrukking voorgesteld door een binaire boom.

```
    inorder(k.links);
    System.out.print(k.data);
    inorder(k.rechts);
  }
}
```

Voor de boom uit figuur 5.11 zullen de elementen in de volgende volgorde worden uitgeschreven: A / B ** C * D + E.

Bij preorder doorlopen wordt eerst de knoop bezocht en dan pas naar links gegaan. Dit wordt herhaald tot men niet meer verder kan. Ga dan naar rechts en begin opnieuw.

```
static void preorder(Knoop k) {
  if (k!=null) {
    System.out.print(k.data);
    preorder(k.links);
    preorder(k.rechts);
  }
}
```

De binaire boom wordt dan uitgeschreven als: + * / A ** B C D E.

Voor postorder wordt de procedure:

```
static void postorder(Knoop k) {
  if (k!=null) {
    postorder(k.links);
    postorder(k.rechts);
    System.out.print(k.data);
  }
}
```

De elementen worden nu uitgeschreven als: A B C ** / D * E +.

5.4.4 Toevoegen en verwijderen van een knoop bij een binaire zoekboom

Een veel gebruikte vorm van een binaire boom is een binaire zoekboom. Hier bevatten alle knopen links van elke knoop waarden die kleiner zijn dan de waarde van de knoop en analoog bevatten alle knopen rechts van elke knoop waarden die groter zijn dan de waarde van de knoop.

Voor het toevoegen van een knoop begint men bij de wortel. Is de waarde kleiner dan de waarde in de knoop dan gaat men naar links, is de waarde groter dan gaat men naar rechts. Is deze knoop null dan voegt men een knoop met de waarde daar toe.

Voor het verwijderen van een knoop zijn er 3 mogelijkheden. Een terminale knoop kan men gewoon verwijderen (en link bij ouder op nul zetten). Bij knoop met 1 kind kan men de knoop vervangen door zijn kind. Bij knopen met 2 kinderen moet men de knoop vervangen door zijn inorder voorganger of opvolger en deze voorganger of opvolger (een terminale knoop) verwijderen.

5.4.5 Toevoegen en verwijderen van een knoop bij een heap

Bij een heap is de waarde van elke knoop kleiner dan de waarden van zijn kinderen (min heap) of groter (max heap). Deze structuur laat toe om snel steeds de kleinste (of grootste) waarde af te halen. Waarden worden steeds op het onderste niveau toegevoegd (complete boom). Is bij een min heap deze waarde kleiner dan zijn ouder, dan worden deze 2 verwisseld en men herhaalt dit tot men zeker is dat de kleinste waarde in de wortel zit. Bij het verwijderen neemt men de waarde uit de wortel en men vervangt deze door zijn kleinste kind totdat men een terminale knoop heeft bereikt.

5.5 Gebalanceerde binaire bomen

5.5.1 Inleiding

Bouw een binaire zoekboom op met volgende waarden: 9, 17, 4, 20, 5, 1, 7 en 11. De resulterende boom heeft diepte 4 wat betekent dat er max 4 ($\log n$) testen moeten gebeuren om een element in de boom te vinden.

Doorloop de binaire boom inorder en sla deze gegevens op om daarna weer een binaire boom op te bouwen. De resulterende boom is een rechtse scheve met diepte 8 (= aantal elementen in de boom) waardoor het zoeken terug linair is. Bij de opbouw moet men corrigeren om toch $O(\log n)$ (gebalanceerde boom) te verkrijgen. Er zijn meerdere mogelijkheden:

- AVL
- red-black
- ...

In het vervolg wordt enkel AVL besproken.

5.5.2 Definities en stellingen

Een binaire boom is gebalanceerd als voor elke knoop de dieptes van de deelbomen hoogstens 1 verschillen. Men spreekt ook van een AVL-boom (Adelson-Velskii en Landis). Elke deelboom van een AVL-boom is zelf ook een AVL-boom.

stelling: de diepte van een AVL-boom met n knopen is $O(\log n)$
bewijs:

5.5.3 Inserteren van een knoop

Bij het toevoegen van een knoop kunnen de dieptes van een aantal deelbomen veranderen, nl deze met wortels gelegen op het pad van de nieuwe knoop naar de wortel van de gehele boom. De boom kan ongebalanceerd zijn en moet terug in balans worden gebracht.

werkwijze: Start bij de nieuwe knoop en volg het pad tot de wortel van de gehele boom. De eerste knoop op dit pad waarvan de grootouder ongebalanceerd is noemt men **x**. **z** is de ongebalanceerde grootouder van x en **y** is de ouder van x en het kind van z.

Aangezien z ongebalanceerd is geworden is de diepte van de deelboom met wortel y 2 groter dan de diepte van de deelboom met als wortel de broer van y.

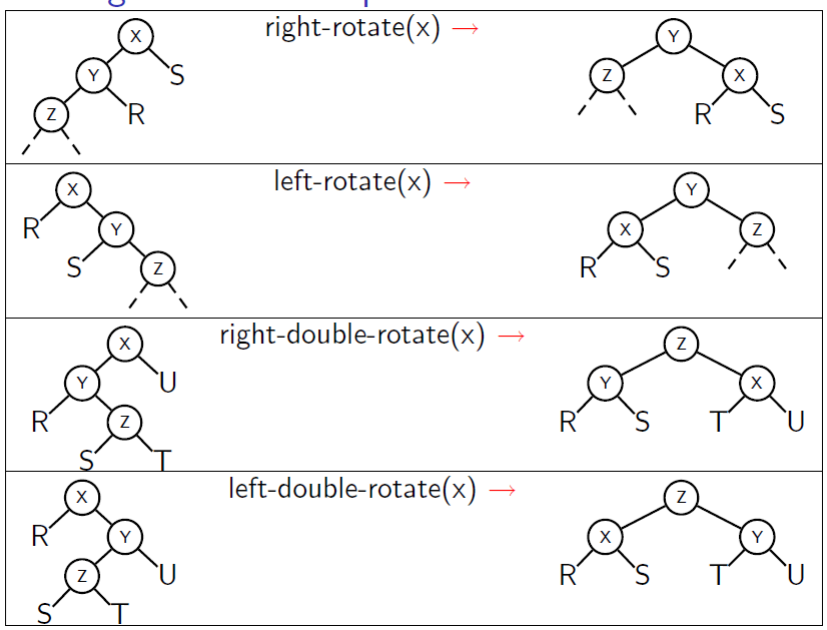
Er zijn 4 mogelijke gevallen die echter allemaal op dezelfde manier kunnen behandeld worden. Hiervoor hernoemen we de knopen x, y en z naar a, b en c maar zodat a, b en c elkaar opvolgen als de boom inorder wordt doorlopen. Vervang dan z door b. a en c worden de kinderen van b. De overige kinderen van x, y en z worden de kinderen van a en c met behoud van de inorder-relaties.

Als $b = y$ dan spreekt men van een enkele rotatie. Als $b = x$ dan spreekt men van een dubbele rotatie.

Door deze rotaties wordt de grotere boom naar omhoog geduwd en de kleinere boom naar omlaag.

5.5.4 Verwijderen van een knoop

Bij het verwijderen van een knoop (terminaal anders eerst verwisseling) wordt er vertrokken van de ouder (w) van de weggelaten knoop. z is de eerste ongebalanceerde knoop op weg naar de wortel van de gehele boom. y is het kind van z met de grootste diepte (geen voorouder van w). x is het kind van y met de grootste diepte (mogelijk niet uniek).



Figuur 5.12: De 4 mogelijke rotaties

Daarna gebeurt hetzelfde als bij het toevoegen van een knoop.

5.5.5 Oefening

5.6 Toepassingen

5.6.1 Beslissingsbomen

Een vaak gebruikte toepassing van bomen zijn de beslissingsbomen (decision trees). Als voorbeeld bekijken we het *acht geldstukken* probleem. Gegeven zijn de stukken a,b,c,d,e,f,g en h. Eén ervan is vals en heeft een verschillend gewicht. We willen bepalen welk geldstuk de valse is door gebruik te maken van een balans en dit met zo weinig mogelijk vergelijkingen. Ook willen we weten of het valse muntstuk zwaarder of lichter is dan de anderen.

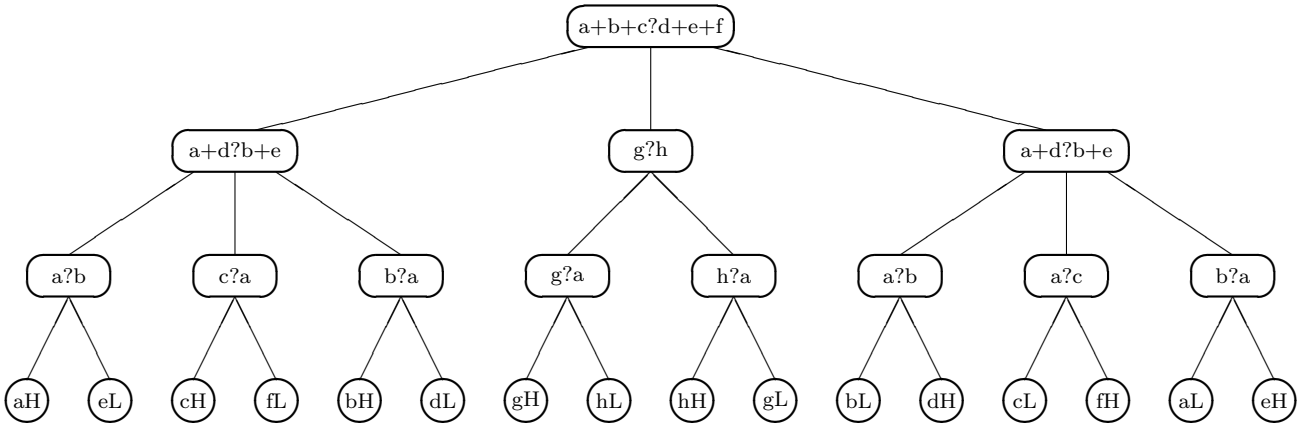
Een ander voorbeeld is de omzetting van Morse-code naar het gewone alfabet. Op niveau 1 wordt geselecteerd op het eerste teken, op niveau n over het n-de teken (punt, streep of einde/spatie).

5.6.2 XML-bomen

In programma's worden de data uit xml-documenten veelal voorgesteld als een boom. De wortel van de boom noemt men de *document root*. Transformaties zetten de ene xml-boom om in een andere.

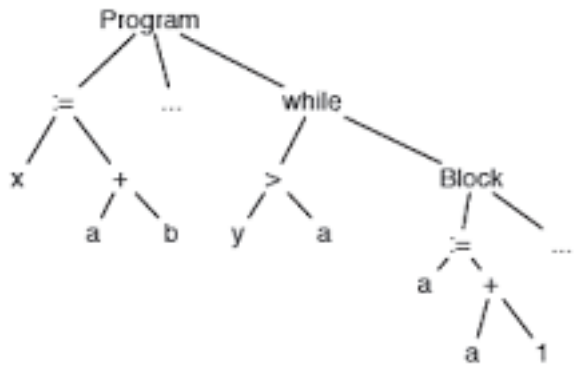
5.6.3 Compilers

Compilers werken bijna allemaal met een boomvoorstelling van het programma. Men noemt dit de *abstract syntax tree* (figuur 5.14).



Figuur 5.13: Een voorbeeld van een beslissingsboom.

```
x := a + b;
y := a * b;
while (y > a) {
   a := a + 1;
   x := a + b
}
```



Figuur 5.14: Een mogelijk voorstelling van een abstract syntax tree.

Hoofdstuk 6

Hashing

6.1 Direct Address Table

Als men een reeks van n elementen wil opslaan en opvragen, zijn er verschillende mogelijklheden.

Als de sleutels van deze elementen unieke integers zijn in het interval $[0, m[$ met $m > n$, dan kan men deze elementen opslaan in een *direct address table*, met $T[i]$ ofwel leeg ofwel 1 element van de reeks bevattend.

De toegang tot een direct address table heeft $O(1)$: gegeven een sleutel k, ga naar $T[k]$ (die het element of null bevat).

Het gebruik van een direct address table heeft 2 belangrijke beperkingen:

- de sleutels moeten uniek zijn
- het bereik van de sleutels is beperkt (geen willekeurige integers)

De eerste beperking kan men opvangen door per sleutelwaarde een lijst van elementen te voorzien. De tijd nodig om 1 element te vinden die bij een sleutel past blijft $O(1)$, maar als men een specifiek element voor die sleutel wil en het maximum aantal elementen per sleutel is maxAantalDuplicaten dan wordt het $O(maxAantalDuplicaten)$. Is maxAantalDuplicaten klein dan blijft de performantie goed. Wordt deze groter, dan moet er naar een andere structuur (bv een boom) worden overgestapt.

Het bereik van de sleutels kan echter ook te groot zijn. Het is onwaarschijnlijk dat men een tabel zal maken voor willekeurige 32-bit integer-sleutels.

6.2 Hashtabel

Direct addressing tables kunnen echter eenvoudig veralgemeend worden door de waarde van de sleutel te vervangen door een functiewaarde van deze sleutel te gebruiken als index. De functie $h(k)$ moet elke waarde van k mappen naar een waarde in een interval $[0, m[$. Een element met sleutel k wordt dan opgeslagen in $T[h(k)]$ i.p.v. in $T[k]$.

In het ideale geval zal de hashingfunctie $h(k)$ elke sleutel vertalen naar een verschillend adres. Als men alle sleutels op voorhand kent, dan kan men daaruit een ideale hashingfunctie afleiden. In de andere gevallen is het mogelijk dat 2 sleutels op 1 adres gemapped worden. Dit is een botsing (collision of overflow) en moet worden opgelost (collision-resolution).

Hashtabellen bieden een efficient gebruik van geheugen en snelheid. Zijn er geen geheugenproblemen dan is de sleutel het adres. Zijn er geen tijdsproblemen dan kan men sequentiëel zoeken.

Hashtabellen hebben een ladingsfactor en een sleuteldichtheid:

$$ladingsfactor = \frac{\text{aantal gebruikte geheugenplaatsen}}{\text{aantal mogelijke geheugenplaatsen}}$$

$$sleuteldichtheid = \frac{\text{aantal gebruikte sleutels}}{\text{aantal mogelijke sleutels}}$$

Een ladingsfactor van ongeveer 70% biedt vaak een goed evenwicht tussen meer geheugen en minder botsingen.

6.2.1 Hashingfuncties

Hashingfuncties moeten voldoen aan volgende eisen:

- eenvoudige, snelle berekeningen
- zo weinig mogelijk botsingen
- uniform: alle mogelijke adressen zijn even waarschijnlijk

Als de sleutel geen getal is, moet het worden omgezet naar een getal. Voor strings kan dit bijvoorbeeld gebeuren door elk karakter om te zetten naar zijn Ascii- of Unicode-waarde (of andere).

Modulo

$$h(k) = k \bmod M$$

Het is belangrijk dat M een priemgetal is (waarom?).

Neem als voorbeeld een tabel met grootte $M = 101$. Voor het omzetten van een string wordt een 5-bit code gebruikt waarbij A=1, B=2, ...

```
AKEY = 00001 01011 00101 11001
     => 44217 mod 101 = 80 (positie in tabel)
BARH => 80
```

Als de strings wat groter zijn zou dit zeer grote getallen opleveren. De methode van Horner en de eigenschappen van modulo zorgen beperken de grootte van de getallen tijdens de berekening van het adres. Dit wordt geïllustreerd aan de hand van de sleutel VERYLONGKEY.

$$22 * 32^{10} + 5 * 32^9 + 18 * 32^8 + 25 * 32^7 + 12 * 32^6 + 15 * 32^5 + 14 * 32^4 + 7 * 32^3 + 11 * 32^2 + 5 * 32^1 + 25$$

via methode van Horner:

$$(\cdots(((22 * 32 + 5) * 32 + 25) * 32 + 15) * 32 + \cdots) * 32 + 25$$

door eigenschap van mod (met b de basis van de code van elk karakter, hier 32)

$$h = (vorige - h * b) + waarde) \bmod M$$

Mid-Square

Bereken het kwadraat van de sleutel en neem de middenste bits eruit. Deze zijn afhankelijk van alle karakters van de string. Met r bits zijn er 2^r mogelijke adressen

Folding

Hierbij wordt de sleutel opgesplitst in verschillende delen met dezelfde lengte (laatste deel kan korter zijn). Daarna gaat men deze delen optellen.

```
123 203 241 112 20
shift folding      folding at boundaries
   123                   123
   203                   302
   241                   241
   112                   211
    20                    20
 + ---                 + ---
   699                   897
```

Multiplication

Hierbij wordt de sleutel k vermenigvuldigd met een constante A waarbij $0 < A < 1$. Daarna wordt het gedeelte na de komma geëxtraheerd en vermenigvuldigd met M, de grootte van de tabel om zo een waarde te bereiken die ligt in $[0, M[$. De hashingfunctie is dus als volgt:

$$h(k) = \lfloor M * (k * A - \lfloor k * A \rfloor) \rfloor$$

De grootte van de tabel is hierbij niet kritisch. Meestal wordt een macht van 2 genomen voor snellere berekeningen (shift ipv vermenigvuldiging). Een goede waarde voor A is:

$$A = \frac{\sqrt{5} - 1}{2} = 0.6180339887$$

6.2.2 Oplossen van botsingen

Het oplossen van botsingen of collision resolution of overflow handling is nodig als meerdere sleutels naar een zelfde adres worden omgerekend. De strategieën kunnen in 2 groepen worden onderverdeeld. Bij gesloten adressering worden op plaats $m = h(k)$ meerdere geheugenplaatsen voorzien. Bij open adressering gaat men op zoek naar een nog onbezette plaats in de tabel.

Gesloten adressering

Hierbij voorziet men meerdere records op plaats m. Deze worden meestal sequentieel doorlopen maar een nieuwe hashing (met collision resolution) is ook mogelijk. Het kan een vast aantal zijn. Is dit aantal te klein dan zijn er nog botsingen mogelijk, is dit te groot dan kan men evengoed geen hashing gebruiken. Logischer is het om een gelinkte lijst te gebruiken per plaats m. Als er teveel botsingen zijn heeft men grote gelinkte lijsten en is een binaire boom beter.

Open adressering

Meestal kiest men echter voor open adressering:

- lineair probing: als op plaats $h(k)$ reeds een ander gegeven zit, zoek dan sequentieel naar de eerstvolgende lege plaats. Een nadeel is de vorming van clusters die meestal zorgen voor een stijgend aantal botsingen.
- rehashing (double hashing): gebruik een tweede hashingfunctie als er botsing is en herhaal dit tot een lege plaats wordt gevonden. De stapgrootte is hierbij afhankelijk van de sleutel zodat er minder clusters worden gevormd. Eventueel kan men nog een 3de, 4de, ... hashingfunctie gebruiken
- overflow area: de tabel wordt opgedeeld in 2 delen: een primair gebied waarnaar de sleutels worden gemapped en een overflow gebied met de elementen die botsingen hadden. Dit overflow-gebied kan sequentieel worden opgevuld en doorzocht of men kan er ook een tweede hashing gebruiken.

6.3 Cryptografische hashfuncties

Een cryptografische hashfunctie is een hashfunctie die *onmogelijk* te inverteren is. Dit wil zeggen dat het *onmogelijk* is om uit het resultaat (vaak digest genoemd) het origineel (meestal message genoemd) te achterhalen.

Deze worden bijvoorbeeld gebruikt bij digitale handtekeningen, bij het opslaan van paswoorden, bij het controleren of berichten zijn gewijzigd, ...

De 2 meest gebruikte cryptografische hashingfuncties zijn MD5 en SHA-1. Voor beiden zijn er echter al problemen opgedoken. Na een competitie is een nieuwe standaard voorgesteld: SHA-3

Hoofdstuk 7

Grafen

7.1 Inleiding

Vele problemen kunnen geformuleerd worden als bestaande uit objecten en verbindingen tussen die objecten, bv.

- vliegtuigroutekaart van Europa: wat is de snelste, goedkoopste manier om van stad A naar stad B te vliegen. (objecten: steden, verbindingen: vliegroutes)
- schema van een elektrisch circuit (objecten: weerstanden, spanningsbronnen, ..., verbindingen: draden)
- jobscheduling (objecten: taken, verbindingen: afhankelijkheden van de verschillende taken)

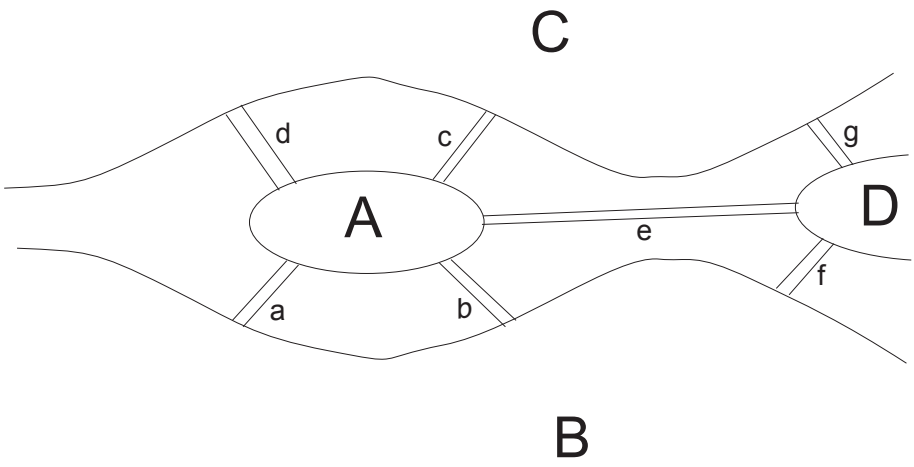
Het gebruik van grafen duikt voor het eerst op in 1736, wanneer Euler het gebruikt voor het oplossen van het Koenigsbergbruggenprobleem. Een rivier (de Pregal) stroomt langs een eiland (Kneiphof) en splitst dan in twee (zie figuur 7.1). Er zijn dus vier landgebieden: A, B, C en D. De gebieden zijn verbonden via zeven bruggen: a, b, c, d, e, f en g. Men vroeg zich toen al lang af of het mogelijk was om startend vanop een gebied alle bruggen één en slechts éénmaal te gebruiken en terug op de startpositie uit te komen.

Euler heeft met behulp van graaftheorie kunnen aantonen dat dit niet mogelijk is. Figuur 7.2 bevat de graaf die bij dit probleem hoort. De knooppunten zijn de gebieden. Men noemt dit de **vertices**. De verbindingen zijn de bruggen. Dit worden **edges** genoemd. Elke vertex heeft een **graad** die het aantal edges aan die vertex aanduidt. Euler heeft bewezen dat er een wandeling is die op een vertex start en die elke edge éénmaal doorloopt en eindigt aan de start als en slechts als de graad van elke vertex even is. Men noemt dit de Eulerian walk.

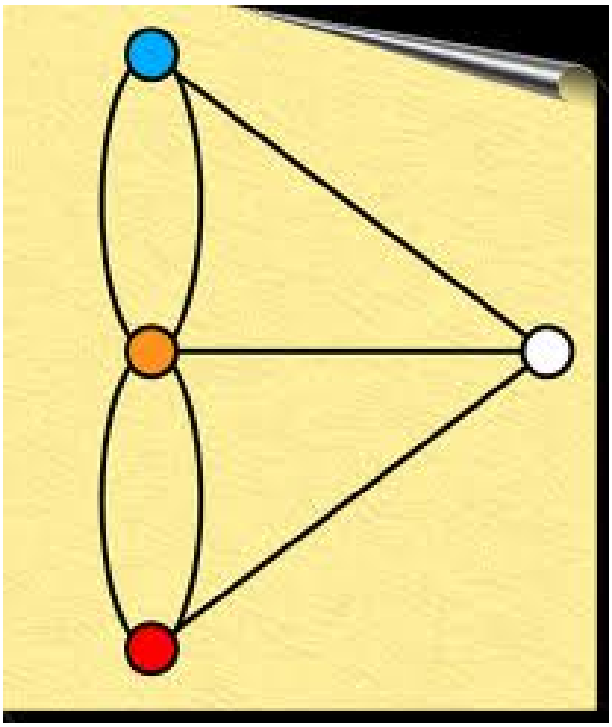
7.2 Definities en terminologie

Een graaf is een verzameling van vertices en edges. Vertices zijn enkelvoudige objecten die een naam en andere eigenschappen kunnen hebben. Edges zijn verbindingen tussen twee vertices. Grafen worden visueel voorgesteld door cirkeltjes voor de vertices en lijnen voor de edges. Figuur 7.3 bevat twee voorstellingen van dezelfde graaf. Deze graaf wordt dus voorgesteld door de verzameling van vertices

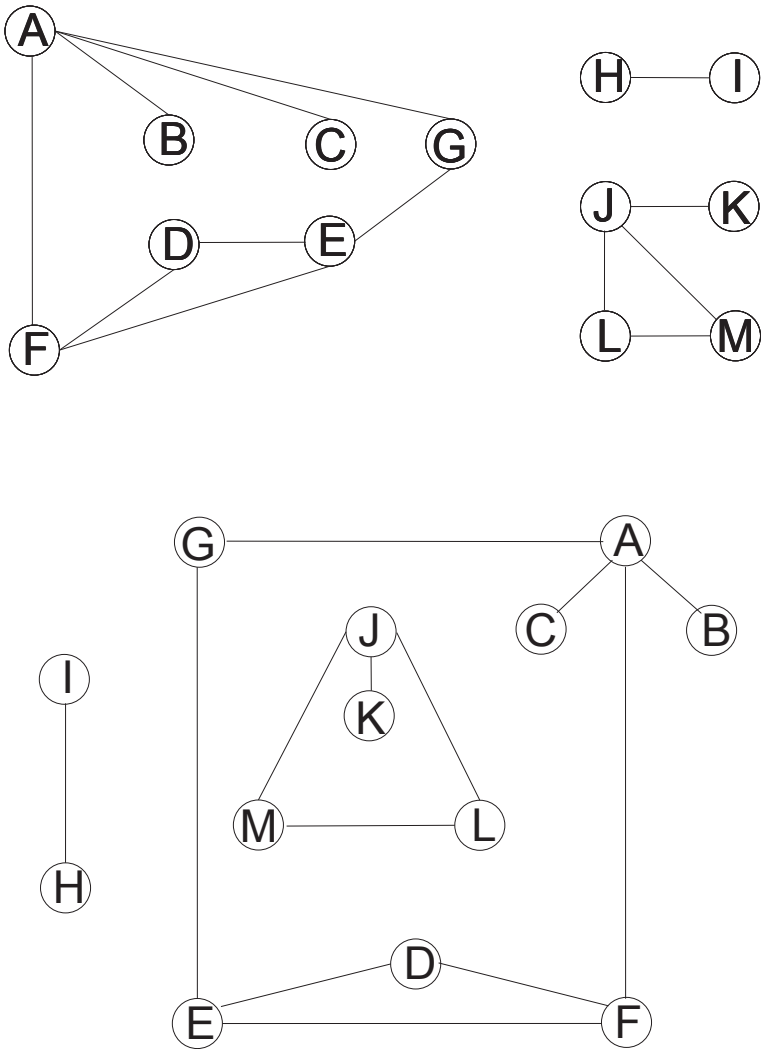
$$V = \{A, B, C, D, E, F, G, H, I, J, K, L, M\}$$



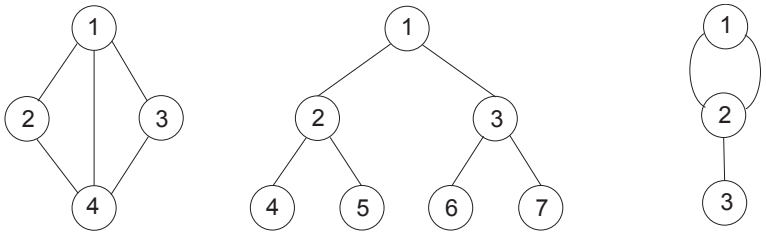
Figuur 7.1: Een visuele voorstelling van het Koenigsbergbruggenprobleem.



Figuur 7.2: De graaf horende bij het Koenigsbergbruggenprobleem.



Figuur 7.3: Twee voorstellingen van dezelfde graaf.



Figuur 7.4: Drie voorbeelden van grafen. De laatste moet een gerichte graaf zijn met een pijl van 1 naar 2, van 2 naar 1 en van 2 naar 3.

en de verzameling van edges

$$E = \{AG, AB, AC, LM, JM, JL, JK, ED, FD, HI, FE, AF, GE\}$$

Bij ongerichte grafen zijn de paren van vertices ongeordend. Bij gerichte grafen zijn dat gerichte paren. Dit wordt voorgesteld door

$$< v1, v2 >$$

met v1 de staart en v2 het hoofd. Door die richting is het paar $v1 - v2$, verschillend van het paar $v2 - v1$:

$$< v1, v2 > \neq < v2, v1 >$$

Figuur 7.4 bevat de voorstelling van drie grafen: G1, G2, G3. De verzamelingen van de vertices en edges van deze grafen zijn als volgt:

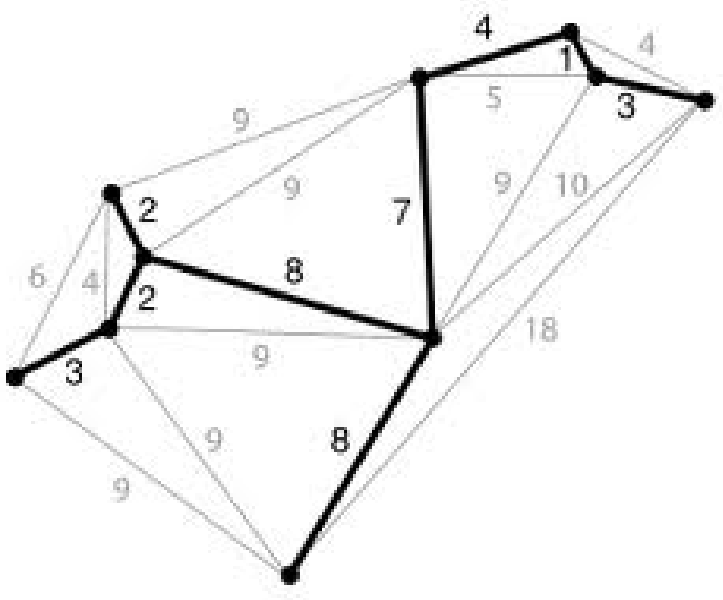
$$\begin{aligned} V(G1) &= \{1, 2, 3, 4\} \\ E(G1) &= \{(1, 2), (1, 3), (1, 4), (2, 4), (3, 4)\} \\ V(G2) &= \{1, 2, 3, 4, 5, 6, 7\} \\ E(G2) &= \{(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7)\} \\ V(G3) &= \{1, 2, 3\} \\ E(G3) &= \{< 1, 2 >, < 2, 1 >, < 2, 3 >\} \end{aligned}$$

Een **pad** van vertex x naar vertex y is een lijst van vertices zodat de opeenvolgende vertices in die lijst verbonden zijn door edges in de graaf. Zo is bijvoorbeeld in figuur 7.3 BAFEG een pad van B naar G.

Een graaf is **geconnecteerd** als er een pad is van elke vertex naar elke andere vertex in de graaf. Intuitief kan men dit als volgt bekijken. Als de vertices fysische objecten zijn en de edges draden die deze objecten met elkaar verbinden, dan zal een geconnecteerde graaf volledig omhoog komen als je een willekeurige vertex hoog genoeg tilt. Bij ongeconnecteerde grafen blijven er een of meerdere objecten liggen. Een graaf die niet geconnecteerd is bestaat wel uit geconnecteerde componenten. Zo is de graaf uit figuur 7.3 ongeconnecteerd en bestaat zij uit drie geconnecteerde componenten.

Een **enkelvoudig pad** is een pad waarbij er geen enkele vertex wordt herhaald. Zo is BAFEGAC geen enkelvoudig pad omdat A er twee maal in voorkomt. Een **lus (cycle)** is een enkelvoudig pad waarbij de eerste en de laatste vertex dezelfde zijn, bijvoorbeeld: AFEGA is een lus.

Een graaf zonder lussen is een boom. Een **spanning tree** van een graaf is een subgraaf met alle vertices, maar met net genoeg edges om een boom te vormen (zie figuur 7.5). Als we aan een boom een edge toevoegen wordt er een lus gevormd.



Figuur 7.5: Een graaf met zijn spanning tree.

Met V het aantal vertices en E het aantal edges, dan geldt steeds volgende ongelijkheid :

$$0 \leq E \leq \frac{1}{2}V(V-1)$$

Grafen waarbij alle mogelijke edges aanwezig zijn, noemt men complete grafen. Als er maar enkele edges ontbreken, spreken we van dichte (dense) grafen. Grafen met relatief weinig edges zijn schaars.

Men kan aan verbindingen ook gewichten toekennen. Deze gewichten stellen afstanden of kosten voor. Men spreekt dan van **gewogen** grafen. Bij **gerichte** grafen hebben de verbindingen een richting. Men maakt dan ook een onderscheid tussen de in-graad (in-degree) en de uit-graad (out-degree) van een knoop (vertex) v. De in-graad van knoop v is het aantal verbindingen die in v toekomen. De uit-graad is het aantal verbindingen die vertrekken uit v.

7.3 Computervoorstelling

Voor de voorstelling van grafen onderstellen we dat de vertices genummerd zijn van 1 tot V. Namen kunnen door middel van hashingtechnieken herleid worden tot een getal dat ligt tussen 1 en V.

7.3.1 Aanliggende matrix (Adjacency Matrix)

Hier wordt een graaf met V vertices voorgesteld door een V*V matrix met $a[x, y] = 1$ als er een edge is van vertex x naar vertex y en $a[x, y] = 0$ anders. Voor de diagonaal (verbinding van een knoop naar zichzelf) wordt soms 0 en soms 1 gebruikt.

Voor de graaf uit figuur 7.3 bekomt men aldus volgende matrix.

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	1	0	0	1	1	0	0	0	0	0	0
B	1	1	0	0	0	0	0	0	0	0	0	0	0
C	1	0	1	0	0	0	0	0	0	0	0	0	0
D	0	0	0	1	1	1	0	0	0	0	0	0	0
E	0	0	0	1	1	1	1	0	0	0	0	0	0
F	1	0	0	1	1	1	0	0	0	0	0	0	0
G	1	0	0	0	1	0	1	0	0	0	0	0	0
H	0	0	0	0	0	0	0	1	1	0	0	0	0
I	0	0	0	0	0	0	0	1	1	0	0	0	0
J	0	0	0	0	0	0	0	0	0	1	1	1	1
K	0	0	0	0	0	0	0	0	0	1	1	0	0
L	0	0	0	0	0	0	0	0	0	1	0	1	1
M	0	0	0	0	0	0	0	0	0	1	0	1	1

Bij ongerichte grafen is de matrix symmetrisch en is het dus eigenlijk voldoende om de helft van de matrix op te slaan.

7.3.2 Incidence matrix

Een incidence matrix is een E*V-matrix met op plaats $[edge, vertex]$ de edge's data. Bij ongewogen grafen betekent 1 geconnecteeerd en 0 niet-geconnecteerd. Elke edge verbindt twee vertices, dus elke rij bevat 2 niet-nul waarden. Soms neemt men ook de getransformeerde voorstelling met V-rijen en E-kolommen.

7.3.3 Aanliggende lijsten (Adjacency Lists)

Hier worden de n rijen van de adjacency matrix vervangen door n gelinkte lijsten, namelijk één gelinkte lijst per vertex.

```
type nextnode = ^node;
     node     = record
                  vertex : integer;
                  link   : nextnode;
                end;
var  headnodes : array[1..n] of nextnode;
```

Bij een ongerichte graaf bekomt men de graad van een vertex door het aantal knopen in de adjacency lijst van die vertex te tellen. Bij gerichte grafen is er een onderscheid tussen de in- en out-degree. Voor de uit-graad is het identisch als bij de ongerichte grafen. Voor de in-graad is het heel wat complexer, en moeten alle lijsten doorlopen worden. Als men dit vaak nodig heeft, stelt men de inverse aanliggende lijsten op. Per vertex bevatten die de vertices van waaruit de verbindingen vertrekken.

7.3.4 Incidence lijst

Bij een incidence list worden de edges opgeslagen in een array of andere lineaire datastructuur. Voor elke edge wordt het paar vertices (die edge verbindt) opgeslagen met eventueel het gewicht (en andere data).

7.4 Doorlopen van een graaf

Het doel is hier om, gegeven een ongerichte graaf G=(V,E) en een vertex v in V(G), alle vertices in G die bereikbaar zijn vanuit v te bezoeken (d. i. alle vertices die geconnecteerd zijn met v). Er zijn hiervoor twee mogelijkheden.

7.4.1 Depth First Search (diepte eerst)

Als eerste wordt de startvertex v bezocht. Vervolgens wordt een onbezochte vertex w, aanliggend aan v, geselecteerd en een depth first search wordt vanaf w opgestart.

Als er een vertex u wordt bereikt waarvan alle aanliggende vertices reeds bezocht zijn, dan trekken we ons terug naar de laatst bezochte vertex die nog een aanliggende vertex w heeft die nog niet bezocht is. We doen dan een depth first search vanaf die vertex w.

Het algoritme stopt als er geen enkele onbezochte vertex meer kan bereikt worden vanuit de bezochte vertices.

Het is het gemakkelijkst om de procedure recursief te schrijven. Er wordt gebruikt gemaakt van een globale rij van boolese waarden: $visited[1..n]$ die geïnitialiseerd zijn op false.

```
procedure dfs(v:integer);
var w : integer;
begin
  visited[v]:=true;
  for each vertex w adjacent to v do
    if not visited[w] then dfs(w);
end;
```

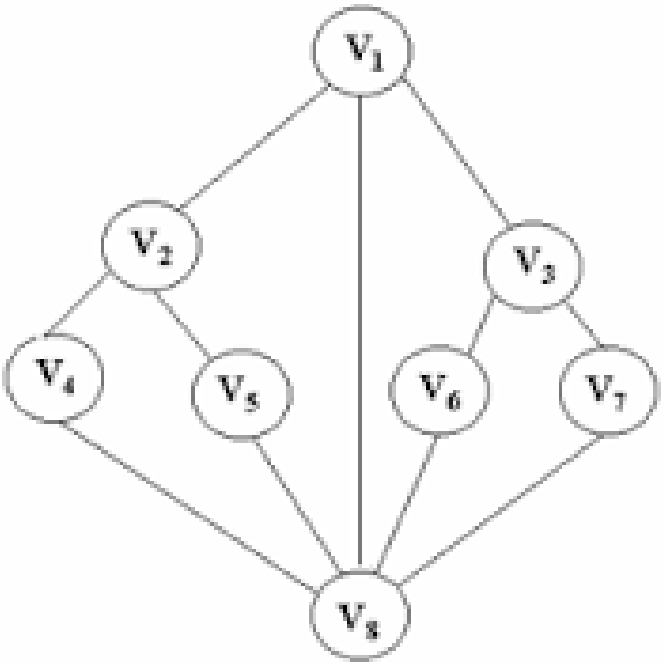
Bekijken we als voorbeeld de graaf uit figuur 7.6. Teken eerst de adjacency lists. We starten bij de vertex v1. Telkens we een knoop bezoeken, duiden we dit aan in de rij visited.

	v1	v2	v3	v4	v5	v6	v7	v8
visited:								

De volgorde waarin de vertices werden bezocht is dus als volgt: v1, v2, v4, v8, v5, v6, v3, v7.

7.4.2 Breadth First Search (breedte eerst)

Bij een breedte eerst doorlopen starten we opnieuw bij vertex v die we als bezocht markeren. We bezoeken daarna alle onbezochte aanliggende vertices van v. Daarna worden de onbezochte aanliggende vertices van die vertices bezocht, enz ....



Figuur 7.6: Een voorbeeldgraaf

```
procedure bfs(v:integer);
var w : integer;
    q : queue;
begin
  visited[v]:=true;
  initialisequeue(q);
  addqueue(q,v);
  while not emptyqueue(q) do
  begin
    deletequeue(v);
    for all vertices w adjacent to v do
      if not visited[w] then
      begin
        addqueue(q,w);
        visited[w]:=true;
      end;
  end;
end;
```

Bekijken we hetzelfde voorbeeld als hierboven (figuur 7.6).

	v1	v2	v3	v4	v5	v6	v7	v8
visited:								

De volgorde waarin de vertices werden bezocht is dus als volgt: v1, v2, v3, v4, v5, v6, v7, v8.

7.5 Kortste paden

Een graaf kan bijvoorbeeld een voorstelling zijn van de autowegstructuur van een staat met als vertices de steden en als edges de wegen die deze steden verbinden. Aan deze edges kunnen gewichten worden toegekend: de afstand tussen de verbonden steden of de gemiddelde tijd die nodig is om die afstand te overbruggen. Stel dat we van stad A naar stad B willen reizen. De eerste vraag die we ons kunnen stellen is: "Is er een pad van A naar B?". Als er meerdere paden zijn, welk pad is het kortst of het snelst?

De lengte van het pad wordt nu niet langer bepaald door het aantal edges, maar door de som van de gewichten van die edges. De startvertex noemt men de *source* of bron. De eindvertex noemt men de *destination* of bestemming.

7.5.1 Single Source All Destinations

Gegeven is een gerichte graaf $G = (V, E)$ met gewichten $w(e) \geq 0$ en bronknoop v_0. Bepaal de kortste paden van v_0 naar alle andere vertices.

Algemeen

Noem S de verzameling van vertices (inclusief de bronvertex) naar waar een kortste pad reeds is gevonden. Voor elke knoop w die niet tot die verzameling behoort berekenen we $dist[w]$. Dit is de lengte van het kortste pad startend vanuit v_0 naar w en gaande enkel via vertices die reeds in S zitten.

Is u de knoop, niet in S, waarvoor $dist[u]$ het kleinste is voor alle knopen niet in S, dan is dat pad naar u het volgende kortste pad dat gegenereerd wordt. u wordt dan aan S toegevoegd.

Algoritme van Dijkstra

Dit algoritme bepaalt de kost van de kortste paden van v_0 naar alle andere knopen in G. Het algoritme genereert die paden niet. Daarvoor is een extensie van het algoritme nodig.

De n vertices van de graaf G worden genummerd van 1 tot n. S is een rij van boolese waarden met

$$s[i] = \begin{array}{ll} false & \text{als } i \notin S \\ true & \text{als } i \in S \end{array}$$

De graaf wordt voorgesteld door een *cost adjacency matrix* met

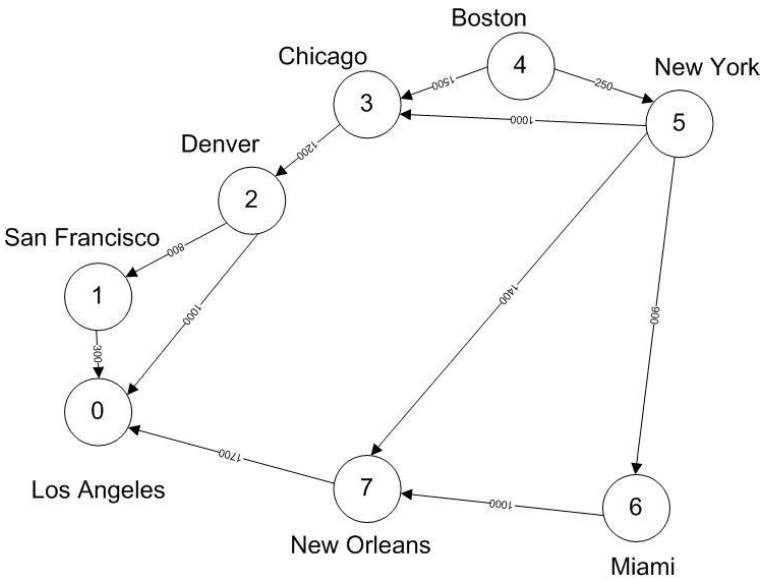
$$cost[i,j] = \begin{array}{l} \text{gewicht van edge } < i,j > \\ \infty \text{ als edge niet in E(G)} \\ 0 \text{ (willekeurig niet negatief) als } i = j \end{array}$$

```
const
 maxn = 10;
 maxwaarde = 9999;
type
 adjacencymatrix=array[1..maxn,1..maxn] of integer;
 distance = array[1..maxn] of integer;

procedure shortestpath(v:integer;
                       cost:adjacencymatrix;
                       var dist:distance;n:integer);
(*dist[j] bevat de lengte van het kortste pad
  van vertex v naar vertex j in een gerichte
  graaf met n vertices voorgesteld door
  de costadjacencymatrix. dist[v]=0
*)
var s:array[1..maxn] of boolean;
    i,u,w : integer;

  function choose(d:distance;n:integer):integer;
  (* bepalen van de knoop, niet in s,
     met de kleinste d
  *)
  var  minimum,w,i : integer;
  begin
    minimum:=maxwaarde; w:=0;
    for i:=1 to n do if not s[i] then
      if d[i]<minimum then
      begin
        minimum:=dist[i]; w:=i;
      end;
    choose:=w;
  end;

begin
(* initilisatie *)
  for i:=1 to n do
  begin
    s[i]:=false;
    dist[i]:=cost[v,i];
  end;
  s[v]:=true; dist[v]:=0;
(* algoritme *)
  for i:=1 to n-2 do
  begin
    u:=choose(dist,n);
    s[u]:=true;
    for w:=1 to n do if not s[w] then
```



Figuur 7.7: Voorbeeld voor algoritme van Dijkstra

```
      if dist[u]+cost[u,w]<dist[w]
      then dist[w]:=dist[u]+cost[u,w];
  end;
end;


1 9999 9999 9999 1500 0 250 9999 9999  6
1 9999 9999 9999 1250 0 250 1150 1650
2 9999 9999 9999 1250 0 250 1150 1650  7
2 9999 9999 9999 1250 0 250 1150 1650
3 9999 9999 9999 1250 0 250 1150 1650  4
3 9999 9999 2450 1250 0 250 1150 1650
4 9999 9999 2450 1250 0 250 1150 1650  8
4 3350 9999 2450 1250 0 250 1150 1650
5 3350 9999 2450 1250 0 250 1150 1650  3
5 3350 3250 2450 1250 0 250 1150 1650
6 3350 3250 2450 1250 0 250 1150 1650  2
6 3350 3250 2450 1250 0 250 1150 1650

KLAAR
5-1 :   3350
5-2 :   3250
5-3 :   2450
5-4 :   1250
5-5 :      0
5-6 :    250
5-7 :   1150
5-8 :   1650
```

7.5.2 Negatieve gewichten

Als de graaf negatieve gewichten bevat, dan kan men het algoritme van Dijkstra niet meer gebruiken. Het BellmanFord algoritme, single source all destinations, (soms ook BellmanFordMoore algoritme genoemd) en het FloydWarshall algoritme, all-pairs shortest paths, zijn 2 algoritmen die wel kunnen omgaan met negatieve gewichten op voorwaarde dat er geen negatieve cycles zijn (worden wel gedetecteerd).

7.6 Minimum spanning tree

Een spanning tree is een subgraaf van een geconnecteerde ongerichte graaf die alle knopen verbindt en een boom is. Voor elke graaf kunnen er meerdere spanning trees bestaan. Bij een gewogen graaf heeft elke verbinding een gewicht. Men kan dan ook een gewicht aan de spanning tree toekennen, namelijk de som van de gewichten van de verbindingen die deel uitmaken van de spanning tree. Een minimum spanning tree heeft dan een gewicht dat kleiner is dan of gelijk is aan het gewicht van alle andere spanning trees.

Voor een ongeconnecteerde graaf spreekt men dan van een minimum spanning forest die de unie is van de minimum spanning trees van de geconnecteerde subgrafen.

Voor het vinden van de minimum spanning tree van een graaf zijn er meerdere algoritmen voorgesteld. De meest gebruikte zijn deze van Prim en deze van Kruskal.

7.6.1 Prim

Bij het algoritme van Prim start men met 1 knoop (meestal arbitrair gekozen). De boom gaat telkens met 1 verbinding groeien: uit de verbindingen die de huidige boom verbinden met knopen die nog niet tot de boom behoren wordt de verbinding geselecteerd met het kleinste gewicht. Deze verbinding wordt dan aan de boom toegevoegd. Dit toevoegen van de verbinding met het kleinste gewicht wordt herhaald tot alle knopen van de graaf tot de boom behoren.

7.6.2 Kruskal

Bij het algoritme van Kruskal wordt er gestart met de volledige verzameling van knopen, maar zonder verbindingen (edges). Opnieuw wordt er nu telkens 1 nieuwe verbinding toegevoegd, namelijk de verbinding met het kleinste gewicht dat als het wordt toegevoegd geen lus veroorzaakt. Verbindingen die een lus zouden vormen worden uit de lijst van kandidaten verwijderd (of gekleurd). Dit wordt herhaald tot er geen verbindingen van de originele graaf meer zijn.