FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Gegevensstructuren en Algoritmen

Rudy Stoop
aangevuld door Jan Cnops
24 september 2018

UNIVERSITEIT
GENT

Academiejaar 2018-2019

INHOUDSOPGAVE

DEEL 1

INLEIDING

HOOFDSTUK 1
ONDERWERP EN DOEL

The process of preparing programs for a digital computer
is especially attractive, not only because it can be
economically and scientifically rewarding,
but also because it can be an aesthetic experience
much like composing poetry or music.
(Donald E. Knuth, *The Art of Computer Programming*.)

Deze cursusnota's horen bij het eerste deel van het vak 'Algoritmen', dat zowel in het derde bachelorjaar als in de masteropleiding industrieel ingenieur informatica onderwezen wordt. Veel van dit materiaal is terug te vinden in goede (voornamelijk Engelstalige) leerboeken over algoritmen, zoals Sedgewick [16], Weiss [19], Cormen e.a. [4], Levitin [13], Manber [14], Knuth [11] of Goodrich en Tamassia [7]. Om het opzoeken van aanvullende informatie gemakkelijker te maken, worden meestal de originele Engelse termen gebruikt of tenminste vermeld.

Algoritmen zijn *methodes* om problemen op te lossen die geschikt zijn voor implementatie op een computer. De gegevens die ze daarbij gebruiken, kunnen op verschillende manieren georganiseerd worden. Hoe dát gebeurt is minstens even belangrijk als het algoritme zelf. Beide zijn dan ook nauw met elkaar verweven. Het hoofddoel van de meeste algoritmen is efficiëntie, gewoonlijk van uitvoeringstijd, soms ook van geheugengebruik (voor mobiele toepassingen bijvoorbeeld).

Algoritmen en gegevensstructuren spelen een belangrijke rol in alle domeinen waar computers gebruikt worden. Dat veel problemen efficiënt oplosbaar zijn is niet enkel te danken aan snelle computers, maar ook (en vaak voornamelijk) aan goede algoritmen. En hoe sneller computers worden, des te groter wordt het belang van efficiënte algoritmen. Snellere computers laten immers toe om steeds grotere problemen aan te pakken, en net daar maken goede algoritmen het verschil. Voor bepaalde problemen zijn de snelst mogelijke algoritmen reeds bekend, maar voor vele andere blijft men zoeken naar betere algoritmen.

Een hoogopgeleide informaticus beschikt dan ook over een stevige bagage aan algoritmische kennis en inzicht. Voor bepaalde toepassingen is het aanwenden van de juiste methode immers cruciaal. De bedoelingen van dit vak zijn dan ook:

- Een overzicht geven van beschikbare algoritmen en gegevensstructuren, en hun eigenschappen. Er bestaan goede implementaties van basisalgoritmen en gege-

vensstructuren, maar om ze oordeelkundig te gebruiken moet men ze eerst goed kennen.[1]

- Inzicht bieden in de implementatie van efficiënte algoritmen en gegevensstructuren. Standaardimplementaties zijn soms te algemeen om specifieke problemen efficiënt op te lossen, zodat het toch nodig kan zijn om zélf algoritmen (of gegevensstructuren) aan te passen of te implementeren.
- Tonen dat het gedrag van een algoritme kan voorspeld worden zonder het te implementeren, en leren hoe men dat doet.
- Oplossingstechnieken bijbrengen. Veel algoritmen zijn concrete toepassingen van algemene algoritmische methodes, die natuurlijk ook voor gelijkaardige en nieuwe problemen kunnen aangewend worden. Bovendien vormen algoritmen een schier onuitputtelijke bron van inspiratie door de talrijke technieken en oplossingsmethodes die ze gebruiken.

Een algoritme is een methode, geen volledig afgewerkt programma. Het staat ook los van de computer waarop het eventueel zal uitgevoerd worden: het zal meestal even geschikt zijn voor een groot aantal computers. Een programma is de *implementatie* in een bepaalde programmeertaal van een of meerdere algoritmen. Belangrijke implementatie-aspecten zoals gegevensabstractie, modulariteit, en het behandelen van fouten komen hier niet aan bod. Daarvoor dienen de uitgebreide oefeningensessies.

De efficiëntie van een algoritme bepalen is vaak moeilijk, en moet dikwijls beroep doen op gevorderde wiskundige technieken. Toch zal de nadruk vooral liggen op het praktisch belang van de methodes, en niet op uitgebreide performantieanalyses.

Algoritmen kunnen in gewoon Nederlands beschreven worden, of met een computerprogramma in een of andere (niet noodzakelijk geïmplementeerde) taal, of in pseudocode (de enige programmeertaal zonder syntactische fouten), of met een wiskundig formalisme. De enige vereiste is dat elk detail van de uit te voeren operaties ondubbelzinnig gespecificeerd is. De (sporadische) codefragmenten zijn hier geschreven in C++ –of C++-achtige pseudocode, omdat deze belangrijke en veel gebruikte taal reeds bekend is uit vorige jaren, en ook gebruikt wordt bij de oefeningen.

Numerieke algoritmen komen hier niet aan bod. Hoewel het onderscheid niet steeds zo duidelijk is, kan men toch stellen dat deze numerieke algoritmen hoofdzakelijk problemen uit de wiskundige analyse aanpakken. Hun hoofddoel is oplossingen *berekenen*, zodat er bijna uitsluitend met reële of complexe getallen gewerkt wordt, en er veel aandacht besteed wordt aan de nauwkeurigheid van die berekeningen. De gebruikte gegevensstructuren beperken zich gewoonlijk tot één- of tweedimensionale tabellen. In dit vak daarentegen worden voornamelijk gegevens behandeld (gerangschikt, opgezocht, onderverdeeld e.d.), die *meestal niet numeriek* zijn (strings, grafen, geometrische structuren e.d.), met behulp van soms zeer ingewikkelde gegevensstructuren. Bovendien is

[1] '*Know the cost (complexity, big O-measure) of every operation you use frequently.*' (B. Stroustrup, ontwerper van C++, bij het gebruik van de Standard Template Library, in [17].)

de controlestructuur van de algoritmen vaak complexer. Zowel de methodes als hun analyse steunen dan ook voornamelijk op discrete wiskunde.

HOOFDSTUK 2
EFFICIËNTIE VAN ALGORITMEN

π seconds is a nanocentury.
(Tom Duff, Bell Labs.)

2.1 ANALYSE VAN ALGORITMEN

In dit vak staat de uitvoeringstijd van de algoritmen centraal. Voor veel problemen bestaan er verschillende algoritmen, zodat men dikwijls een keuze zal moeten maken vooraleer te beginnen met de eigenlijke implementatie. Daarom zou het zeer nuttig zijn mochten we de performantie van verschillende algoritmen met elkaar kunnen vergelijken, *zonder ze te implementeren en te laten uitvoeren*.

Aangezien we de uitvoeringstijd niet kunnen meten zonder het geïmplementeerde algoritme uit te voeren, zullen we verplicht zijn om hem te bepalen via een *analyse* van het algoritme zelf. Een exacte tijdsduur voorspellen is echter een illusie, omdat deze niet enkel afhangt van het algoritme, maar ook van implementatiefactoren zoals de *programmeertaal* waarin het algoritme geschreven werd, de bekwaamheid van de *programmeur*, de efficiëntie van de code die de gebruikte *compiler* voor die taal produceert, en de *processorarchitectuur* van de computer (de structuur en de instructies van de processor). En zelfs al zouden we al deze factoren kunnen bepalen, dan nog zouden ze enkel van toepassing zijn voor één particuliere implementatie van het algoritme op één particuliere computer.

Omdat een goed algoritme meestal goed presteert in elke behoorlijke implementatie, is het zinvol om de performantie van algoritmen te bestuderen onafhankelijk van particuliere implementaties. Daartoe tracht men het *aantal primitieve operaties* te bepalen dat elke implementatie zou moeten uitvoeren op een typische sequentiële processor (het RAM-model), waarbij abstractie gemaakt wordt van de werkelijke uitvoeringstijd van deze operaties. Die tijden worden in de berekening gewoon vervangen door niet nader bepaalde constanten.

De uitvoeringstijd van een algoritme is uiteraard afhankelijk van het aantal verwerkte gegevens n, maar ook nog van andere factoren die specifiek zijn voor het probleem, zoals bijvoorbeeld de oorspronkelijke volgorde van die gegevens, of hun statistische verdeling.[1] Omdat het natuurlijk onbegonnen werk is om de uitvoeringstijd in functie

[1] Bij sommige problemen zijn er meerdere parameters, zoals we later zullen zien. Omdat ook de grootte van

van n te bepalen voor alle mogelijke collecties van n gegevens, beperkt men zich tot drie speciale gevallen, waarvan de details probleemafhankelijk zijn:

- Het beste geval ('best-case running time'). Zo zijn algoritmen om gegevens te rangschikken gewoonlijk het snelst wanneer die gegevens reeds in volgorde staan.
- Het slechtste geval ('worst-case running time'). Rangschikken van gegevens bijvoorbeeld is gewoonlijk het traagst wanneer ze in omgekeerde volgorde staan, maar uitzonderlijk ook wel wanneer ze reeds in volgorde staan.

  Het slechtste geval is meestal het belangrijkste, omdat het algoritme gegarandeerd nooit slechter zal performeren, omdat het slechtste geval bij bepaalde problemen in de praktijk vrij vaak voorkomt, en omdat het gemiddelde geval (zie hierna) dikwijls nagenoeg hetzelfde gedrag vertoont.
- Het gemiddeld geval ('average-case running time'). Deze tijd is dikwijls zeer moeilijk te bepalen (of men is er nog altijd niet volledig in geslaagd). Bovendien is er een probleem met de definitie van 'gemiddeld'. Om de analyse mogelijk te maken, gaat men gewoonlijk uit van veronderstellingen die niet altijd zeer realistisch zijn. Zo kan men bij rangschikken onderstellen dat elke permutatie van de invoergegevens even waarschijnlijk is, wat in de praktijk zelden het geval is.

  Soms is het gemiddelde geval zeer goed. Als de kans dat het mis gaat klein genoeg is neemt men soms het risico: men spreekt dan van een *Las Vegas*-algoritme.

Wanneer men de uitvoeringstijd op de hoger geschetste manier bepaalt in elk van deze drie gevallen, bekomt men telkens een resultaat dat enkel nog afhangt van n, het aantal verwerkte gegevens. Deze functie van n is meestal veel te ingewikkeld om bruikbaar te zijn, en daarom tracht men ze te vereenvoudigen. De termen van de functie bevatten typisch constante factoren die zelf opgebouwd zijn uit de vele constante tijden van de primitieve operaties. Aangezien we de exacte waarde van die tijden toch niet (wensen te) kennen, is het ook zinloos om de exacte waarde te berekenen van die factoren. Het is voldoende te weten dat het *constanten* zijn, en we stellen ze voor met een symbolische naam.

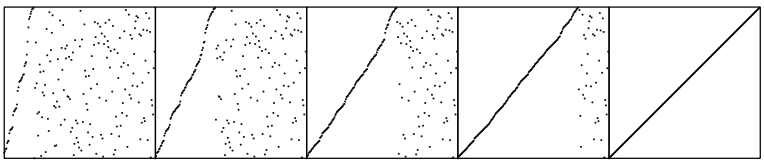
Laten we als voorbeeld eens de uitvoeringstijd bepalen van een eenvoudig algoritme: gehele getallen rangschikken met insertion sort[2]. Code 2.1 geeft een mogelijke implementatie. Elke toewijzing en elke afzonderlijke test vereisen slechts een constante tijd, die we als t_i aanduiden. Het aantal getallen (de grootte van de vector) noemen we n. De opdrachten in de `for` worden $n - 1$ keer uitgevoerd. Het aantal keer dat deze in de `while` uitgevoerd worden hangt af van de plaats van het volgende tussen

de gegevens een rol kan spelen gebruikt men soms als maat het *aantal bits* dat nodig is om die gegevens voor te stellen.

[2] Zie cursus Informatica II (en ook reeds Informatica I).

```
void insertion_sort (vector<T> & v) {
  // Stijgend rangschikken
  for (int i = 1 ; i < v.size() ; i++) {
     // De eerste i getallen staan reeds in volgorde
     T h = move(v[i]);
     int j = i-1;
     while (j >= 0 && h < v[j]) {
        v[j+1] = move(v[j]);
        j--;
     }
     v[j+1] = move(h);
  }
}
```

Pseudocode 2.1. Insertion sort.



Figuur 2.1. Insertion sort.

te voegen getal in de reeds gerangschikte deelvector. De primitieve controlesprongen die op machineniveau de `for`, `while` en `if` implementeren zijn hier niet zichtbaar, maar hun duur kan meegerekend worden in naburige operaties (als die even frequent uitgevoerd worden), aangezien de grootte van al die constante tijden toch niet gekend is.

- Het is duidelijk dat het beste geval zich voordoet wanneer de oorspronkelijke vector stijgend gerangschikt is, want dan worden de instructies in de `while` nooit uitgevoerd. We krijgen dan de aantallen vermeld in de derde kolom van tabel 2.1. In totaal geeft dat een tijd van

  $$(t_2 + t_3 + t_4 + t_5 + t_6 + t_7 + t_{10})n + (t_1 - t_3 - t_4 - t_5 - t_6 - t_7 - t_{10})$$

  en die heeft de gedaante $c_1 n + c_0$ van een lineaire veelterm in n.

- Het slechtste geval doet zich voor als de oorspronkelijke vector dalend gerangschikt is, want dan wordt elke `while` i keer herhaald. We krijgen dan de aantal-

Operatie	Tijd	Aantal (best)	Aantal (slechtst)
`i = 1`	t_1	1	1
`i < \nospell{v.size()}`	t_2	n	n
`i++`	t_3	$n-1$	$n-1$
`h = v[i]`	t_4	$n-1$	$n-1$
`j = i-1`	t_5	$n-1$	$n-1$
`j >= 0`	t_6	$n-1$	$(n+2)(n-1)/2$
`&& h < v[j]`	t_7	$n-1$	$n(n-1)/2$
`v[j+1] = v[j]`	t_8	0	$n(n-1)/2$
`j--`	t_9	0	$n(n-1)/2$
`v[j+1] = h`	t_{10}	$n-1$	$n-1$

Tabel 2.1. Aantal primitieve operaties voor het beste en slechtste geval.

len vermeld in kolom vier van tabel 2.1. In totaal geeft dat een tijd van

$$\left(\frac{t_6+t_7+t_8+t_9}{2}\right)n^2+\left((t_2+t_3+t_4+t_5+\frac{(t_6-t_7-t_8-t_9)}{2}+t_{10}\right)n \\ +(t_1-t_3-t_4-t_5-t_6-t_{10})$$

of $c_2'n^2+c_1'n+c_0'$, een veelterm van de tweede graad in n.

- Voor het gemiddelde geval onderstelt men gewoonlijk dat elke permutatie van de te rangschikken getallen even waarschijnlijk is. Een tussen te voegen getal kan dan met de zelfde waarschijnlijkheid op elke mogelijke positie terechtkomen. Gemiddeld moet dus de helft van de reeds gerangschikte deeltabel opgeschoven worden: de `while` wordt half zoveel keer uitgevoerd als bij het slechtste geval. Ook hier zullen we een veelterm van de tweede graad in n bekomen.

Met de nieuw ingevoerde constanten zijn deze functies reeds heel wat eenvoudiger, maar toch gaan we nog een stap verder. We zijn namelijk niet zozeer geïnteresseerd in de waarde van de uitvoeringstijd voor verschillende waarden van n, als in de *toename* van die tijd, wanneer n stijgt. Voor voldoend grote n wordt die toename voornamelijk bepaald door de belangrijkste term, en de constante factor van die term doet daarbij niet terzake, zodat ook die nog weggelaten wordt. (De waarde van de functie cn^2 wordt immers honderd maal groter als n tien maal groter wordt, voor om het even welke waarde van c.)

Toegepast op ons voorbeeld wordt de uitvoeringstijd voor het beste geval dus vereenvoudigd tot n, en die voor het slechtste en het gemiddelde geval tot n^2. Hoeveel tijd een particuliere implementatie van dit algoritme ook vraagt, die tijd zal zowel in het slechtste als het gemiddelde geval evenredig met n^2 stijgen, voor voldoend grote n. We zullen later zien dat het (gelukkig) beter kan.

Wanneer men twee algoritmen heeft voor hetzelfde probleem, dan wordt het algoritme waarvan de uitvoeringstijd voor het slechtst mogelijke geval het minst snel stijgt voor

grote n, doorgaans als efficiënter beschouwd.

2.2 ASYMPTOTISCHE BENADERING VAN FUNCTIES

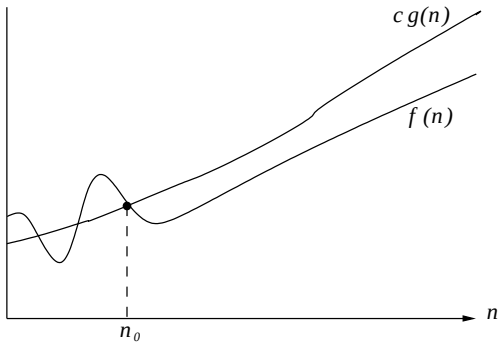
De vereenvoudigde functie voor de uitvoeringstijd van het algoritme hierboven is een voorbeeld van een *asymptotische benadering*: vanaf voldoend grote waarden voor n kan men de uitvoeringstijd benaderen met de functies cn^2 (slechtst) of cn (best), voor een niet nader bepaalde en niet terzake doende constante c. Voor bepaalde algoritmen is het moeilijk of zelfs (voorlopig) onmogelijk om op die manier te werk te gaan: een functie voor de uitvoeringstijd opstellen, en die daarna vereenvoudigen. In dat geval tracht men een functie van n te vinden, die de uitvoeringstijd zo goed mogelijk *begrenst*, wanneer n voldoende groot wordt. Soms kan men die grens later scherper stellen.

In beide gevallen echter hebben we te maken met een begrenzende functie. Ook voor de ondergrens van de uitvoeringstijd kan men op analoge manier een asymptotische benadering bepalen. Deze benaderingen worden dan, bij gebrek aan beter, gebruikt om algoritmen onderling te vergelijken.

Deze nogal vage omschrijving van 'begrenzing' voor grote n wordt nauwkeurig gedefinieerd met het volgende algemeen gebruikte notatiesysteem:

- Om aan te duiden dat een functie $f(n)$ niet sneller groeit dan een andere functie $g(n)$, zodat deze laatste dus een *bovengrens* vormt, gebruikt men de O-notatie:

  > Men zegt dat een functie $f(n) = O(g(n))$, als er positieve constanten c en n_0 bestaan, zodanig dat $0 \leq f(n) \leq cg(n)$ voor alle $n \geq n_0$ (figuur 2.2).



Figuur 2.2. Asymptotische bovengrens: $f(n) = O(g(n))$.

Bemerk dat de constante c niet nader gespecificeerd wordt: het is voldoende dat ze bestaat. Hetzelfde geldt voor n_0.

Ter illustratie tonen we aan dat als het aantal operaties een veelterm in n is

$$f(n) = c_k n^k + c_{k-1} n^{k-1} + \cdots + c_1 n + c_0$$

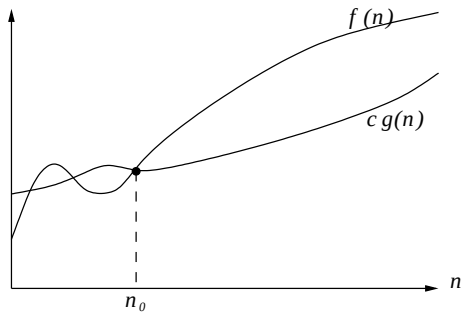
alle termen behalve de eerste verwaarloosd mogen worden, of anders gezegd, dat $f(n) = O(n^k)$. Immers,

$$\begin{aligned} f(n) &\leq |c_k| n^k + |c_{k-1}| n^{k-1} + \cdots + |c_1| n + |c_0| \\ &\leq (|c_k| + |c_{k-1}|/n + \cdots + |c_1|/n^{k-1} + |c_0|/n^k) n^k \\ &\leq (|c_k| + |c_{k-1}| + \cdots + |c_1| + |c_0|) n^k \quad \text{voor } n \geq 1 \end{aligned}$$

Met $n_0 = 1$ en de constante $c = |c_k| + |c_{k-1}| + \cdots + |c_1| + |c_0|$ is de eigenschap bewezen. Bemerk dat dit niet de enig mogelijke constante is. Voor elke $c > |c_k|$ kan immers een gepaste n_0 gevonden worden. Voor de O-notatie is dit echter van geen belang.

- Men kan een functie $f(n)$ ook karakteriseren met een *ondergrens*, als men bijvoorbeeld weet dat ze minstens even snel groeit als een andere functie $g(n)$. In dit geval gebruikt men de Ω-notatie ('Omega'):

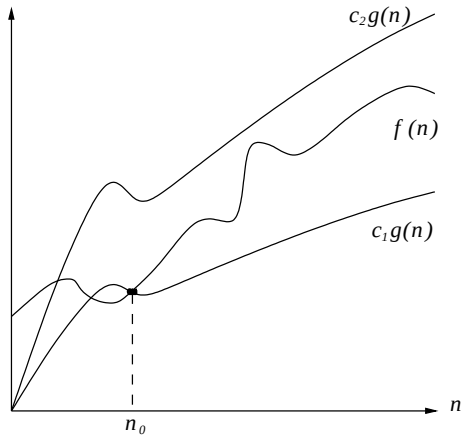
  Men zegt dat een functie $f(n) = \Omega(g(n))$ is, als er positieve constanten c en n_0 bestaan, zodanig dat $0 \leq cg(n) \leq f(n)$ voor alle $n \geq n_0$ (figuur 2.3).



Figuur 2.3. Asymptotische ondergrens: $f(n) = \Omega(g(n))$.

- Tenslotte bestaat er nog een Θ-notatie ('Theta'), wanneer *dezelfde* functie $g(n)$ zowel een boven- als een ondergrens vormt voor $f(n)$:

  Een functie $f(n) = \Theta(g(n))$, als er positieve constanten c_1, c_2, en n_0 bestaan, zodat $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ voor alle $n \geq n_0$ (figuur 2.4).



Figuur 2.4. Asymptotische boven- en ondergrens: $f(n) = \Theta(g(n))$.

Uit slordigheid wordt soms de O-notatie gebruikt waar een Ω- of Θ-notatie vereist zijn. Vaak voorkomende performanties zijn ($\lg x$ is een verkorte notatie voor $\log_2 x$)

$$O(1), O(\lg\lg n), O(\lg n), O(\sqrt{n}), O(n), O(n\lg n), O(n^2), O(n^3), O(2^n), O(n!)$$

Het is belangrijk bij de oplossing van een probleem om een idee te hebben van de efficiëntie van een voorgestelde oplossing. Nu kunnen we ons afvragen of er een theoretische manier bestaat om, gegeven een probleem, te bepalen wat de asymptotische efficiëntie is van het best mogelijk algoritme. In het algemeen blijkt dat (tot nu toe?) niet mogelijk. Zo is er de bekende klasse van NP-problemen (NP staat voor **N**ondeterministic **P**olynomial). Voor geen enkel van deze problemen is er een algoritme *bekend* dat efficiëntie $O(n^\alpha)$ heeft, voor welke α dan ook, maar niemand weet of er zo'n algoritme kan bestaan. In sommige gevallen is het echter mogelijk om toch zekere grenzen te trekken.

Stelling 1 *Een algoritme voor een probleem dat een oplossing zoekt die kan uitgedrukt worden als de keuze tussen $f(n)$ mogelijkheden is $\Omega(\lg f(n))$.*

Het invullen van een getal met maximale grootte $f(n)$ komt overeen met het invullen van $\lceil \lg f(n) \rceil$ bits, wat op zich $\Theta(\lg f(n))$ is. ∎

Zo is er bijvoorbeeld het sorteerprobleem. Gegeven een tabel van n *verschillende* elementen. Hoe efficiënt kan sorteren zijn? Het antwoord is dat er $n!$ verschillende permutaties zijn van n elementen. Een tabel sorteren komt overeen met het vinden van de juiste permutatie om alles in volgorde te zetten. Dus is sorteren $\Omega(\lg n!)$. De formule

van Stirling geeft

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right), \tag{2.1}$$

waarin e de basis is voor de natuurlijke logaritmen, $e = \lim_{n\to\inf}(1 + 1/n)^n$.

Wat sorteren in het algemene geval $\Omega(n \lg n)$ maakt.

2.3 AFSCHATTINGEN VAN RECURSIEBETREKKINGEN

Soms is het mogelijk een functie f af te schatten door een recursieve betrekking, waarbij we een bovengrens voor $f(n)$ kunnen bepalen aan de hand van vorige waarden van f. Dergelijke afschattingen zijn versies van het zgn. *master theorem*, zie bijvoorbeeld [4].

In deze paragraaf geven we daarvan een paar belangrijke voorbeelden.

2.3.1 Machten van n

(1)
$$f(n) \leq c + f(n-1) \implies f \text{ is } O(n). \tag{2.2}$$
Dit is zondermeer duidelijk.

(2) Meer algemeen geldt
$$f(n) \leq cn^\alpha + f(n-1) \implies f \text{ is } O(n^{\alpha+1}), \alpha > -1. \tag{2.3}$$
Ook dit is vrij snel duidelijk. We krijgen

$$\begin{aligned} f(n) &\leq cn^\alpha + f(n-1) \\ &\leq c(n^\alpha + (n-1)^\alpha)) + f(n-2) \\ &\leq \ldots \\ &\leq c(n^\alpha + (n-1)^\alpha) + \ldots 1^\alpha) + f(0) \end{aligned}$$

De gemakkelijkste manier om de term tussen haakjes af te schatten is door hem om te zetten naar een integraal. Voor $\alpha \leq 0$ geldt $x^\alpha \geq \lceil x \rceil^\alpha$ en dus

$$\begin{aligned} n^\alpha + (n-1)^\alpha + \ldots 1^\alpha &= \int_0^n \lceil x \rceil^\alpha \, dx \\ &\leq \int_0^n x^\alpha \, dx \\ &= \frac{1}{\alpha+1} n^{\alpha+1}. \end{aligned}$$

Voor $\alpha \geq 0$ gebruiken we de ongelijkheid $x^\alpha \geq \lfloor x \rfloor^\alpha$.

(3) Erg belangrijk is het limietgeval waar $\alpha = -1$. We krijgen de regel

$$f(n) \leq \frac{1}{n} + f(n-1) \implies f \text{ is } O(\lg n). \tag{2.4}$$

De redenering is dezelfde als hierboven, alleen is hier

$$\begin{aligned} \int_0^n \frac{1}{\lceil x \rceil}\, dx &\leq 1 + \int_1^n \frac{1}{x}\, dx \\ &= 1 + \log n \end{aligned}$$

2.3.2 logaritmische afschatting

Vooral bij verdeel-en-heerstechnieken, waarover later meer, krijgen we afschattingen die een relatie geven tussen $f(n)$ en $f(n/2)$ (of juister $f(\lceil n/2 \rceil)$, vermits f alleen voor gehele getallen gedefinieerd is). We laten echter meestal de afronding weg en stellen voor x reëel dat $f(x) = f(\lceil x \rceil)$. Zo krijgen we bijvoorbeeld

$$f(n) \leq f\left(\frac{n}{2}\right) + c \implies f \text{ is } O(\lg n). \tag{2.5}$$

Dit is te zien door $k = \lceil \lg n \rceil$ te nemen, zodat $n/2^k \leq 1$. Dit levert

$$f(n) \leq f\left(\frac{n}{2}\right) + c \leq f\left(\frac{n}{2^2}\right) + 2c \leq \ldots \leq f\left(\frac{n}{2^k}\right) + kc = f(1) + c\lceil \lg n \rceil.$$

2.3.3 Afschattingen met sommen

In een aantal belangrijke gevallen bevat de recursieve afschatting voor $f(n)$ niet één voorgaande term maar een som van voorgaande termen.

(1) Zij x gegeven dan is

$$(x-1)(1 + x + x^2 + \ldots x^p) = x^{p+1} - 1.$$

Als gevolg daarvan is

$$S(x,p) = (1 + x + x^2 + \ldots x^p) = \sum_{i=0}^{p} x^i = \frac{x^{p+1} - 1}{x - 1}.$$

Voor $x \geq 2$ en $p > 0$ geldt dus dat $x^p < S(n,p) < x^{p+1}$, terwijl voor $0 < x < 1$ geldt dat $S(x,p) < 1/(1-x)$. Hieruit kunnen we een eenvoudige afschatting bekomen voor $f(x,p) = \sum_{i=0}^{p} ix^i$ door te stellen dat

$$f(x,p) = \sum_{i=0}^{p} ix^i \leq \sum_{i=0}^{p} px^i \leq px^{p+1} \qquad x \geq 2, \tag{2.6}$$

terwijl ook hier $f(x,p)$ groter is dan de laatste term in de reeks, $f(x,p) > px^p$.

(2) Een andere belangrijke afschatting is die voor $g(x,p) = \sum_{i=0}^{p}(p+1-i)x^i$. Hier is het nuttig om uit te gaan van $(x-1)g(x,p)$. dit levert

$$\begin{aligned}(x-1)&\sum_{i=0}^{p}(p+1-i)x^i \\ &= -(p+1) + \left(\sum_{i=1}^{p}(p+2-i)x^i - (p+1-i)x^i\right) + x^{p+1} \\ &= -(p+1) - 1 + S(x,p) + x^{p+1},\end{aligned}$$

wat resulteert in

$$g(x,p) = \sum_{i=0}^{p}(p+1-i)x^i < 2x^{p+1} \qquad x \geq 2. \tag{2.7}$$

2.3.4 Bernoulliverdeling

De formule van Stirling is ook belangrijk in een andere context, omdat ze toelaat een redelijk eenvoudige afschatting te maken van de binomiaalgetallen:

$$\binom{n}{k} = \frac{n(n-1)\ldots(n-k+1)}{k!} < \frac{n^k}{k!} < \frac{n^k}{(k/e)^k} = \left(\frac{en}{k}\right)^k$$

Deze zijn belangrijk bij gerandomiseerde algoritmen waarbij een Bernoulliverdeling optreedt. Een Bernoulliverdeling wordt gedefinieerd door een randomgebeurtenis met twee mogelijke uitslagen, A en B, waarbij de kans op B gelijk is aan p, $(0 \leq p \leq 1)$. Als we een reeks van dergelijke gebeurtenissen hebben van lengte m, dan is de kans om exact k keer B te hebben gelijk aan

$$\binom{m}{k}(1-p)^{m-k}p^k < \left(\frac{pem}{k}\right)^k (1-p)^{m-k} < \left(\frac{pem}{k}\right)^k.$$

De kans $\mathrm{Bernoulli}(j,m,p)$ om *minstens* j keer B te hebben is dan, door de somregel 2.6 voor $j > pem$

$$\sum_{k=j}^{m}\binom{m}{k}(1-p)^{m-k}p^k < \sum_{k=j}^{m}\left(\frac{pem}{k}\right)^k < \sum_{k=j}^{m}\left(\frac{pem}{j}\right)^k = \left(\frac{pem}{j}\right)^j \sum_{k=j}^{m}\left(\frac{pem}{j}\right)^{k-j}.$$

De laatste som is kleiner dan $(1-(pem)/j)^{-1}$ en dus, voor $j > 2pem$, kleiner dan 2 zodat

$$\mathrm{Bernoulli}(j,m,p) < 2\left(\frac{pem}{j}\right)^j, \qquad j > 2pem \tag{2.8}$$

2.4 ANDERE CRITERIA

Na al de aandacht die we besteed hebben aan de definitie en de bepaling van een asymptotische benadering voor de uitvoeringstijd, is het toch nodig om op te merken dat dit niet noodzakelijk de enige, of in bepaalde gevallen zelfs niet de belangrijkste maatstaf is bij de beoordeling van een algoritme. Andere overwegingen kunnen dan een belangrijke rol spelen:

- Indien een programma slechts een paar keer gaat gebruikt worden, overweegt de moeite (en de kost) van implementeren en testen. Men kiest dan best een algoritme dat snel correct kan geïmplementeerd worden.
- De asymptotische benadering van de uitvoeringstijd is een *benadering*, en moet dus met de nodige voorzichtigheid gebruikt worden:
  - Constante factoren spelen geen rol in asymptotische benaderingen, maar natuurlijk wel in de praktijk. Twee algoritmen met hetzelfde asymptotische gedrag zijn niet noodzakelijk even snel.
  - Bovendien gelden deze benaderingen enkel voor voldoend grote n. Een algoritme met een minder goed asymptotisch gedrag kan voor kleine waarden van n toch beter zijn dan een potentieel sneller algoritme.
  - De asymptotische bovengrens van het slechtste geval kan soms te pessimistisch zijn (of dat geval te zeldzaam), en de gemaakte veronderstellingen bij de bepaling van de gemiddelde performantie zijn niet altijd zeer realistisch (noodgedwongen, om de analyse mogelijk te maken). Experimenten op reële gegevens worden dan noodzakelijk.
- Soms gebruiken snelle algoritmen teveel geheugen, zodat ze een uitwendig opslagmedium zouden moeten gebruiken, wat hun voordeel natuurlijk teniet doet. (Ook voor de geheugenvereisten van een algoritme gebruikt men de O-notatie en haar verwanten.)
- Het klassieke RAM-model dat we bij onze analyse gebruiken houdt geen rekening met de geheugenhiërarchie van de computer. Twee potentieel even snelle algoritmen kunnen dan ook een verschillende performantie vertonen, als de geheugentoegangen van het ene een grotere lokaliteit bezitten dan die van het andere, en dus efficiënter gebruik maken van de processorcaches.

  Er werden meer ingewikkelde modellen bedacht, maar hoe realistischer het model, des te lastiger de analyses. Zo zijn er 'cache-oblivious' algoritmen, die er met een eenvoudig model in slagen rekening te houden met de volledige geheugenhiërarchie, onafhankelijk van concrete geheugenkarakteristieken. (Voor meer informatie, zie Frigo e.a. [6].)
- Parallelle verwerking met meerdere processoren kan de werking versnellen. Dat beïnvloedt onze asymptotische afschattingen niet als het aantal processoren constant is maar wel als het samen groeit met n, wat bijvoorbeeld kan gebeuren in

een netwerk van n routers. De mogelijkheid van parallelle verwerking kan soms wel de keuze van algoritmes beïnvloeden.

2.5 OPTIMALISATIE

Vaak wordt de efficiëntie van een implementatie sterk beïnvloed door sommige zaken die op het eerste zicht onbelangrijk lijken, zoals de voorstelling van bepaalde data. Het is dan ook de zaak om, bij programmaontwerp, hierop steeds te letten. Nemen we een klein voorbeeld.

Bij een kwis mag elke ploeg een prijs kiezen, en dit in volgorde van de uitslag. Stel dat deze uitslag gegeven is in een tabel `uitslag` zodanig dat `uitslag`$[i]$ aangeeft wat de rangschikking is van ploeg i: `uitslag`$[i] = 0$ betekent dat ploeg i de winnaar is, en zo voorts. Bekijken we twee manieren om de ploegen in volgorde aan te spreken:

```
void naieveKeuze(){
    for (int i=0; i < n;i++){
        int j=0;
        while (uitslag[j] != i)
            j++;
        kies(j);
    };
};


void efficienteKeuze(){
    vector<int> invers(n);
    for (int i=0; i < n;i++)
        invers[uitslag[i]]=i;
    for (int j=0; j < n;j++)
        kies(invers[j]);
    };
};
```

Beide methodes hebben twee lussen, maar de naïeve heeft twee *vernestelde* lussen. Daarmee is ze $O(n^2)$ (als de `kies`-functie $O(1)$ is), terwijl de efficiënte methode $O(n)$ is. Op kleine voorbeeldjes zal het verschil niet merkbaar zijn (en in deze context waarschijnlijk nooit; kwissen hebben zelden heel veel deelnemers), maar zelfs als n maar een paar duizend bedraagt, of als het om een algoritme gaat dat zelf vaak wordt opgeroepen, kan zo'n extra factor n zwaar doorwegen.

Bijzondere aandacht moet dan ook altijd uitgaan naar het gebruik van de juiste gegevensstructuren: we zullen in het vervolg van de cursus zien dat een verkeerde keuze

het verschil kan uitmaken tussen een werkend en een niet-werkend programma. Bovendien maakt een juiste keuze vaak duidelijk hoe een gegevensstructuur zal gebruikt worden: op deze manier verbetert ze ook de leesbaarheid van code.

DEEL 2

RANGSCHIKKEN

INLEIDING

When in doubt, sort.
(Steven Skiena.)

Rangschikken van gegevens is een fundamentele operatie in de informatica. (En niet alleen daar.) Zelfs al is rangschikken niet het hoofddoel van een programma, toch moeten gegevens vaak gerangschikt worden. De oplossing van veel problemen wordt immers een stuk eenvoudiger (en sneller) wanneer de gegevens in volgorde staan. Het hoeft ons dan ook niet te verwonderen dat er talrijke algoritmen om te rangschikken bestaan. Nagenoeg elke mogelijke methode werd toegepast op rangschikken, zodat een studie van de verschillende algoritmen meteen een overzicht biedt van interessante technieken, die ook daarbuiten bruikbaar zijn.

Geen enkele methode is optimaal voor elke toepassing.[3] Wélk algoritme het meest geschikt is hangt onder meer af van het *aantal* gegevens dat moet gerangschikt worden, van de *aard* van die gegevens, van de mate waarin de gegevens reeds *geordend* zijn, en van de *plaats* waar de gegevens tijdens het rangschikken opgeslagen zijn: in het inwendig of het uitwendig geheugen.

Gegevens worden gerangschikt volgens een bepaald kenmerk. In een telefoonboek staan abonnees gerangschikt op naam, maar voor een andere toepassing zou dat op telefoonnummer kunnen zijn. Dat kenmerk noemt men de *sleutel* van de gegevens. Vaak bestaan gegevens uit meer dan die sleutel alleen. Een sleutel heeft dan *bijbehorende informatie*, en het spreekt vanzelf dat de band tussen beide tijdens het rangschikken niet mag verbroken worden. Meestal worden de gegevens verplaatst tot ze in de gewenste volgorde staan. Het (eventueel meermaals) verplaatsen van grote gegevens is niet efficiënt, en in dat geval zal men eerder wijzers naar die gegevens verplaatsen (indirect rangschikken).

Naast de (tijds)efficiëntie van de algoritmen, zullen we ook nog aandacht besteden aan twee andere aspecten van rangschikken:

- *Geheugengebruik.* Zeker voor grote hoeveelheden gegevens is het van belang te weten of het algoritme hulpgeheugen voor (eventueel een deel van) die gegevens vereist. Als de grootte van dat hulpgeheugen onafhankelijk is van het aantal gegevens, dan spreekt men van *ter plaatse* rangschikken.

[3] Daarom gebruikt men in de praktijk vaak hybridische algoritmen, zoals bijvoorbeeld bij introsort (Musser [15]).

- *Stabiliteit.* Een algoritme rangschikt stabiel als het de oorspronkelijke volgorde van gegevens met gelijke sleutels niet wijzigt. Dit heeft natuurlijk enkel belang voor de bijbehorende informatie.

  Stel dat een lijst met namen van personen en hun leeftijd alfabetisch gerangschikt werd, met hun naam als sleutel. Als men daarna die lijst volgens leeftijd zou rangschikken, verwacht men dat de namen van personen met dezelfde leeftijd nog altijd in alfabetische volgorde staan. Dat is enkel zo als het gebruikte algoritme stabiel is.

Vooraleer we met de beschrijving van de verschillende methodes beginnen, maken we een aantal veronderstellingen:

- We gaan ervan uit dat alle te rangschikken gegevens in het *inwendig geheugen* opgeslagen zijn. Dit heet dan ook 'inwendig' rangschikken. Om zeer grote hoeveelheden gegevens te rangschikken moet men beroep doen op het uitwendig geheugen. De technieken van dit 'uitwendig' rangschikken zijn echter afhankelijk van de beschikbare hardware. Pas op het einde van dit deel komt dit probleem even aan bod.

- De gegevens zullen gewoonlijk ook in een (eendimensionale) *tabel* ondergebracht zijn. De meeste methodes om te rangschikken moeten immers snel gegevens op willekeurige plaatsen kunnen testen of verplaatsen, en daarvoor is een tabel ideaal.

- We veronderstellen dat de gegevens klein zijn, zodat we ze efficiënt kunnen verplaatsen. Om de beschrijving van de algoritmen eenvoudig te houden, laten we de bijbehorende informatie voorlopig buiten beschouwing (behalve bij het bespreken van de stabiliteit).

- Tenslotte spreken we af dat we steeds zullen rangschikken in *stijgende volgorde* (of correcter, in niet-dalende volgorde): het kleinste element komt bij de kleinste index.

- Asymptotische efficiëntie in de grootte n is een beetje dubbelzinnig bij dit probleem. Veel methodes zijn efficiënter (of soms zelfs: alleen maar bruikbaar) als er veel sleutels hetzelfde zijn. Maar om bij grote n veel verschillende sleutels te hebben moeten ze ook groot zijn. Nemen we bijvoorbeeld bitstrings. Er zijn 2^k bitstrings van lengte k. Om dus n even lange maar verschillende bitstrings te hebben moet $k \geq \lg n$. Hierdoor worden vergelijkingen trager (twee bitstrings van lengte k vergelijken vraagt k tijd). Als k dus meegroeit met n zoals $\lg n$ betekent, als we een sorteermethode hebben die $O(n \lg n)$ vergelijkingen maakt, dat deze $O(n \lg^2 n)$ tijd daarvoor nodig heeft.

- Het voorbeeld toont aan dat de `swap`operatie belangrijk is. Een string van lengte k verplaatsen is $O(k)$, twee strings van lengte k swappen is $O(1)$.

HOOFDSTUK 3
EENVOUDIGE METHODES

We beginnen met enkele eenvoudige methodes die echter verre van onbelangrijk zijn. Zo is insertion sort beter dan de asymptotisch meer efficiënte (en meestal meer ingewikkelde) methodes voor tabellen die nagenoeg gerangschikt zijn, en ook voor kleine tabellen. Bovendien wordt de methode gebruikt om efficiënte methodes nog sneller te maken. (Een eenvoudige vorm van een hybridisch algoritme.) Shellsort kan beschouwd worden als een uitbreiding van insertion sort, wat een snelle methode oplevert, ook voor grote tabellen. Selection sort is niet efficiënt en wordt dus nauwelijks gebruikt, maar hetzelfde principe gecombineerd met een efficiënte gegevensstructuur geeft heapsort, een van de snelste methodes.

3.1 RANGSCHIKKEN DOOR TUSSENVOEGEN

3.1.1 Insertion sort

Stel dat een aantal elementen in een tabel reeds in volgorde staat, en dat we een nieuw element aan de tabel willen toevoegen (gesteld dat er plaats voor is), zodanig dat ze opnieuw in volgorde staat. We moeten dan de plaats vinden waar het nieuw element moet komen, en alle grotere elementen één positie opschuiven. Het nieuw element wordt hierbij *tussengevoegd*, vandaar de naam van de methode.

De eenvoudigste manier om deze plaats te vinden is het nieuwe element te vergelijken met de opeenvolgende elementen in de tabel, beginnend bij het kleinste of het grootste (lineair zoeken). Daarna schuiven we de grotere elementen op, en zetten het element op zijn plaats. We hebben nu een gerangschikte tabel met één element meer. Dit proces kan natuurlijk herhaald worden.

Om een gegeven tabel te rangschikken zouden we dus een hulptabel kunnen gebruiken, waarin we het eerste element van onze tabel kopiëren, en dan alle volgende elementen aan de hulptabel toevoegen zoals hierboven. Deze hulptabel is echter overbodig, want we kunnen tussenvoegen in de oorspronkelijke tabel, omdat de gerangschikte en nog te rangschikken zones elkaar niet overlappen. We moeten er enkel voor zorgen dat er plaats is om op te schuiven, door het tussen te voegen element in een hulpvariabele op

te slaan. Zoeken en opschuiven kunnen samen gebeuren, als we van achter naar voor te werk gaan, zoals in code 2.1 op p. 7. (Bemerk dat we hier *opschuiven*, omdat het voor kleine elementen zoals getallen efficiënter is dan omwisselen. Voor sommige klassen is het gebruik van een swapoperatie beter).

De `while` vereist een dubbele voorwaarde, omdat het nieuw element kleiner kan zijn dan alle vorige. Men kan evenwel vooraf testen of dit het geval is, en dan alles opschuiven, zodat één voorwaarde volstaat.

Deze methode gebruikt enkel de oorspronkelijke tabel, en het aantal hulpvariabelen hangt niet af van n: ze rangschikt dus *ter plaatse*. Let ook op de vergelijking tussen het nieuw element en de vorige elementen: die zorgt ervoor dat de methode *stabiel* is.

De efficiëntie wordt duidelijk gedomineerd door de binnenste herhaling. De buitenste herhaling wordt steeds $n - 1$ maal uitgevoerd, zodat al haar operaties $\Theta(n)$ zijn. De binnenste herhaling bestaat voornamelijk uit sleutelvergelijkingen en schuifoperaties. Er zijn minstens zoveel sleutelvergelijkingen als schuifoperaties: elke schuifoperatie wordt voorafgegaan door een sleutelvergelijking, en per element is er hoogstens nog één extra sleutelvergelijking. Nu is het totaal aantal schuifoperaties gelijk aan het aantal *inversies* in de oorspronkelijke tabel. (Een paar elementen dat niet in de juiste volgorde staat vormt een inversie.) Want elke schuifoperatie verwijdert één inversie.

- Het *slechtste* geval doet zich voor wanneer elk paar elementen een inversie vormt. (De tabel staat dan in omgekeerde volgorde.) Er zijn $n(n-1)/2$ paren elementen, dus zijn er evenveel schuifoperaties en $O(n(n-1)/2+n)$ sleutelvergelijkingen. Aangezien de termen voor de andere operaties slechts $\Theta(n)$ zijn, is de totale uitvoeringstijd $\Theta(n^2)$.

- Voor het *gemiddeld* geval onderstellen we dat elke permutatie van de tabelelementen even waarschijnlijk is. Het gemiddeld aantal inversies is dan $n(n-1)/4$. Immers, elk paar elementen vormt een inversie hetzij in de tabel zelf, hetzij in de omgekeerde tabel, en er zijn $n(n-1)/2$ paren elementen, zodat het aantal inversies in beide tabellen samen steeds $n(n-1)/2$ bedraagt. Aangezien elke permutatie van de tabel even waarschijnlijk is (en dus ook die van de omgekeerde tabel), is het gemiddeld aantal inversies in beide tabellen gelijk, en dus $n(n-1)/4$. Met als gevolg dat er gemiddeld $\Theta(n^2)$ schuifoperaties en $O(n^2+n)$ sleutelvergelijkingen nodig zijn, zodat de totale gemiddelde uitvoeringstijd opnieuw $\Theta(n^2)$ is. Het gemiddelde geval vertoont dus hetzelfde gedrag als het slechtste.

- Het *beste* geval doet zich uiteraard voor wanneer er geen inversies zijn, omdat de tabel reeds in volgorde staat. De kwadratische termen zijn nu verdwenen, zodat de performantie slechts $\Theta(n)$ is.

  Belangrijker dan dit triviaal geval is dat dezelfde performantie gehaald wordt als de elementen *nagenoeg* in volgorde staan. De methode blijft immers $\Theta(n)$ zolang het aantal inversies $O(n)$ is. Dat is bijvoorbeeld het geval als het aantal

inversies per element constant is, of als er slechts een constant aantal elementen $O(n)$ inversies heeft. (Dit laatste doet zich bijvoorbeeld voor wanneer een gerangschikte tabel achteraan uitgebreid wordt met een beperkt aantal nieuwe elementen, in willekeurige volgorde.)

Op nagenoeg gerangschikte tabellen zal deze methode sneller blijken dan de meest efficiënte (algemene) methodes. Daarom wordt ze ook wel gebruikt in de slotfase van deze methodes: als de elementen reeds grotendeels gerangschikt staan, gebeurt de afwerking met insertion sort.

Samenvattend is de uitvoeringstijd van insertion sort dus $O(n^2)$, maar in het beste (en nagenoeg beste) geval wordt dat $\Theta(n)$. Het is dus een eenvoudig voorbeeld van een *adaptief algoritme*, dat zijn performantie aanpast aan de eigenschappen van de invoer.

Aangezien de deeltabel waarin tussengevoegd moet worden reeds in volgorde staat, zouden we binair zoeken kunnen gebruiken om sneller de juiste plaats voor het nieuwe element te vinden (zie bij 4.3). Enige voorzichtigheid is wel geboden als de tabel duplicaten bevat, indien we de methode stabiel willen houden. Het opschuiven moet nu achteraf gebeuren (en blijft natuurlijk even traag). Lineair zoeken is $O(n)$, binair zoeken slechts $O(\lg n)$. Het algemeen gedrag van de methode blijft echter kwadratisch.

3.1.2 Shellsort

Insertion sort is traag omdat het tussen te voegen element bij elke stap slechts over één positie opschuift. Opschuiven komt eigenlijk neer op omwisselen van twee naast elkaar gelegen elementen: beide verwijderen slechts één inversie. Aangezien een tabel met n elementen gemiddeld $n(n-1)/4$ inversies heeft, zal *elke* methode die rangschikt door naast elkaar gelegen elementen te verwisselen, een gemiddelde uitvoeringstijd van $\Omega(n^2)$ hebben. Om die grens te doorbreken zal men dus elementen moeten verwisselen die verder van elkaar liggen, zodat elke stap meerdere inversies wegneemt.[1]

Stel dat we een tabel t zo zouden ordenen dat, waar we ook beginnen, elementen die k posities van elkaar liggen een gerangschikte reeks vormen. Men kan een dergelijke tabel ook beschouwen als bestaande uit k gerangschikte reeksen die verweven zijn: de eerste reeks is dan $(t_1, t_{1+k}, t_{1+2k}, \ldots)$, de tweede $(t_2, t_{2+k}, t_{2+2k}, \ldots)$, en zo voorts. Een dergelijke tabel noemt men 'k-gerangschikt'.

Wanneer men een k-gerangschikte tabel h-gerangschikt maakt met een kleinere waarde h, dan blijft ze ook nog altijd k-gerangschikt. Dat kunnen we aantonen door gebruik te maken van de volgende eigenschap:

> Gegeven twee reeksen elementen x en y, in willekeurige volgorde, en niet noodzakelijk even lang. Elk van de laatste r elementen van x is minstens zo groot als ('domineert') het *overeenkomstig* element uit de eerste r elementen van y. Met $x = \{x_1, x_2, \ldots, x_m, x_{m+1}, \ldots, x_{m+r}\}$, en

[1] Selection sort verplaatst weliswaar elementen over grotere afstanden, maar gebruikt een inefficiënte selectiemethode.

> $y = \{y_1, y_2, \ldots, y_n, y_{n+1}, \ldots, y_{n+r}\}$, waarbij $m \geq 0$ en $n \geq 0$, betekent dit dat $x_{m+j} \geq y_j$, voor $1 \leq j \leq r$. Als men nu beide reeksen *apart* rangschikt, dan blijven die relaties geldig. (Met uiteraard de nieuwe waarden van deze elementen.)

Dat is eenvoudig om in te zien. Want na het rangschikken domineert x_{m+j} $m + j$ elementen van x. Alle elementen van x, op m na, domineren elk een ander element van y. Dus domineren minstens j van die $m+j$ elementen van x elk een ander element van y. Met als gevolg dat x_{m+j} tenminste j elementen van y domineert. En dus zeker de kleinste j elementen van y, wat de eerste j elementen zijn van de gerangschikte y. Kortom, $x_{m+j} \geq y_j$. Ditzelfde geldt voor alle mogelijke j ($1 \leq j \leq r$). Zodoende.

In een k-gerangschikte tabel geldt dat $t_{i+k} \geq t_i$, voor alle mogelijke i ($1 \leq i \leq n-k$). Blijft dit zo na h-rangschikken van de tabel? Om dit aan te tonen passen we de eigenschap toe op twee h-reeksen in de k-gerangschikte tabel. Reeks x is de h-reeks waartoe t_{i+k} behoort: $x = \{\ldots, t_{i+k-h}, t_{i+k}, t_{i+k+h}, \ldots\}$. Analoog is reeks y de h-reeks waartoe t_i behoort: $y = \{\ldots, t_{i-h}, t_i, t_{i+h}, \ldots\}$. (Bemerk dat t_{i+k} het laatste element kan zijn van x, of t_i het eerste element van y.) Nu bevat reeks x zeker het element dat k posities verder ligt in t dan het eerste element van y. Alle volgende elementen in x liggen dan ook k posities verder dan een overeenkomstig volgend element in y. Omdat de tabel k-gerangschikt is domineren al deze elementen van x hun overeenkomstig element in y. Bovendien zijn het de laatste elementen van x, en de eerste elementen van y. Alle voorwaarden voor de eigenschap zijn dus voldaan. Als we de tabel nu h-rangschikken, worden (onder meer) de h-reeksen x en y onafhankelijk gerangschikt. Toch blijven de laatste elementen van x (waaronder t_{i+k}) hun overeenkomstig element uit de eerste elementen van y (waaronder t_i) domineren. Dus geldt nog steeds dat $t_{i+k} \geq t_i$. Deze redenering geldt voor alle mogelijke i ($1 \leq i \leq n-k$): na het h-rangschikken, is de tabel dus nog steeds k-gerangschikt.

Shellsort (Shell, 1959) maakt de tabel k-gerangschikt met een bepaalde beginwaarde voor k, en herhaalt dit voor steeds kleinere k-waarden, waarbij de laatste waarde zeker één moet zijn om een volledig gerangschikte tabel te garanderen. De oorspronkelijke naam van deze methode was dan ook 'rangschikken met dalende incrementen' (*'diminishing increment sort'*). Op het eerste gezicht lijkt dit heel veel werk, maar aangezien de ordening bekomen met de grotere k-waarden niet verloren gaat bij de kleinere k-waarden (zie hierboven), hebben deze laatste steeds minder werk. Zelfs met slechts twee incrementen kan de efficiëntie reeds beter worden dan $O(n^2)$.

Aangezien de deelreeksen voor de grotere waarden van k kort zijn, en deze voor de kleinere k-waarden, hoewel langer, reeds beter geordend, is de beste methode om elke deelreeks te rangschikken insertion sort. In plaats van de deelreeksen voor een bepaalde k-waarde apart te rangschikken, kan dat beter samen gebeuren, zoals in pseudocode 3.1. We beginnen dus bij het tweede element van elke deelreeks, dan het derde, enz. Bemerk dat we voor k gelijk aan één insertion sort (moeten) terugvinden. En net als daar kan de dubbele voorwaarde vermeden worden door een voorafgaandelijke test.

Voor grote k worden de deelreeksen te kort om veel effect te hebben. Gewoonlijk

```
void Shellsort (vector<T> & v) {
  // Rangschikken met dalende incrementen
  int k = ...; // Initieel increment
  while (k >= 1) {
     for (int i = k ; i < v.size() ; i++) {
     // Vorige sleutels in deelreeks staan al in volgorde
        T h = move(v[i]);
        int j = i-k;
        while (j >= 0 && h < v[j]) {
           v[j+k] = move(v[j]);
           j -= k;
        }
        v[j+k] = move(h);
     }
     k = ...; // Volgend increment
  }
}
```

Pseudocode 3.1. Shellsort.

neemt men een begininterval kleiner dan $n/2$.

De efficiëntie van de methode hangt af van de gebruikte reeks k-waarden, maar de optimale reeks is nog steeds niet bekend: de volledige analyse blijft een uiterst moeilijk onopgelost probleem. Met de originele incrementen van Shell $\{\lfloor n/2 \rfloor, \ldots, k_i, k_{i-1} = \lfloor k_i/2 \rfloor, \ldots, k_0 = 1\}$, kan de methode in sommige gevallen nog steeds $\Theta(n^2)$ zijn. Twee opeenvolgende incrementen zijn dan ook best relatief priem, zodat opeenvolgende deelreeksen zo onafhankelijk mogelijk worden. Teveel incrementen gebruiken zou teveel werk betekenen. Gewoonlijk laat men ze ongeveer dalen zoals een meetkundige reeks, zodat er $O(\lg n)$ incrementen zijn.[2] Met een goede reeks blijkt de snelheidswinst zich te beperken tot ongeveer 25 procent.

Enkele van de (voorlopig) beste reeksen:

- De reeks van Sedgewick (1986):

$$k_i = \begin{cases} 9 \cdot 2^i - 9 \cdot 2^{i/2} + 1 & \text{voor } i \geq 0 \text{ en even} \\ 8 \cdot 2^i - 6 \cdot 2^{(i+1)/2} + 1 & \text{voor } i > 0 \text{ en oneven} \end{cases}$$

  Men kan aantonen dat de uitvoeringstijd voor deze reeks in het slechtste geval $O(n^{4/3})$ bedraagt, en er zijn sterke vermoedens (via zeer uitgebreide testen) dat

[2] Men kan trouwens aantonen dat als er een reeks incrementen bestaat die de *gemiddelde* performantie van Shellsort $O(n \lg n)$ maakt (het theoretisch minimum, zie later), dan moet ze $\Theta(\lg n)$ incrementen bevatten (Jiang e.a. [10]).

de gemiddelde uitvoeringstijd slechts $O(n^{7/6})$ is.[3]

- De reeks van Tokuda (1992) blijkt nog iets beter[4] :

$$k_i = \lceil (9(9/4)^i - 4)/5 \rceil \qquad \text{met } (9/4)k_i < n$$

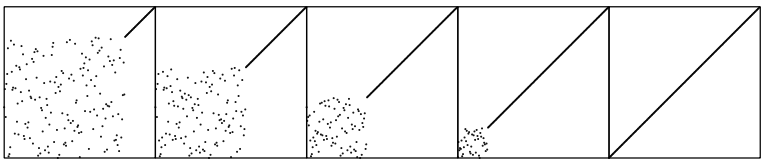
- Er werd ook systematisch gezocht naar reeksen die de *gemiddelde* performantie van Shellsort optimaliseren. Om het aantal mogelijkheden zoveel mogelijk te beperken, werden enkel de eerste (kleinste) incrementen onderzocht, aangezien die de grootste invloed hebben op de performantie van een reeks (Ciura [3]). De (voorlopig) beste van deze reeksen begint met de getallen: $1, 4, 10, 23, 57, 132, 301, 701, \ldots$.

Shellsort is dus aanzienlijk sneller dan insertion sort, en blijft zeer goed presteren voor zelfs vrij grote waarden van n. Bovendien is de methode eenvoudiger dan de (asymptotisch) snellere methodes. Voor vele gevallen is het dus de aangewezen methode: gebruik de gesofisticeerde methodes enkel als het echt nodig blijkt.

Shellsort rangschikt duidelijk *ter plaatse*. De methode is echter *niet stabiel*: het rangschikken van de verschillende deelreeksen kan de volgorde van gelijke elementen wijzigen.

3.2 RANGSCHIKKEN DOOR EENVOUDIG SELECTEREN

Een van de eenvoudigste methodes om te rangschikken werkt als volgt: zoek het groot-



Figuur 3.1. Selection sort.

ste element uit de tabel en verwissel het met het laatste element, zoek het grootste uit de overgebleven elementen en verwissel het met het voorlaatste element, enz. Telkens wordt het grootste element geselecteerd, vandaar de Engelse naam van de methode ('selection sort'). Aangezien de volgorde van de elementen willekeurig is, moeten we ze allemaal testen om telkens het maximum te vinden. Code 3.2 geeft een mogelijke implementatie.

[3] Men kan evenwel aantonen dat Shellsort in het slechtste geval $\Omega(n \lg^2 n/(\lg \lg n)^2)$ is, en dus steeds boven de theoretische ondergrens van $\Omega(n \lg n)$ blijft (zie later), voor om het even welke reeks incrementen. Voor (heel) grote n zal de methode dus trager worden dan de meest efficiënte methodes.

[4] De notatie $\lceil x \rceil$ (Engels *ceiling*) stelt het kleinste geheel getal voor $\geq x$. Analoog definieert men $\lfloor x \rfloor$ (*floor*) als het grootste geheel getal $\leq x$.

```
void selection_sort (vector<T> & v) {
   // Rangschikken door (eenvoudige) selectie
   for (int i = v.size()-1 ; i > 0 ; i--) {
      // De plaats van het maximum zoeken
      int imax = i;
      for (int j = 0 ; j < i ; j++)
         if (v[j] > v[imax]) imax = j;
      // Het maximum plaatsen via omwisselen
      swap(v[i],v[imax]);
   }
}
```

Pseudocode 3.2. Selection sort.

Bemerk dat deze methode over alle elementen tegelijk moet beschikken om haar werk te kunnen doen, en dat de definitieve gerangschikte reeks element na element gegenereerd wordt. Dat is net het tegenovergestelde van insertion sort, waar de oorspronkelijke elementen één voor één behandeld worden, terwijl we de definitieve positie van elk element pas kennen wanneer het rangschikken voltooid is.

Het aantal sleutelvergelijkingen is hier onafhankelijk van de oorspronkelijke volgorde van de elementen, en bedraagt

$$(n-1)+(n-2)+\cdots+2+1 = n(n-1)/2$$

Het aantal verwisselingen daarentegen is hoogstens $n-1$. De oorspronkelijke volgorde van de gegevens heeft enkel invloed op het aantal verwisselingen, dat $O(n)$ is, en op het aantal keer dat het maximum wordt aangepast, en dat is $O(n^2)$. Kortom, dit is een $\Theta(n^2)$ methode, die voor elke invoer trager uitvalt dan het gemiddeld geval van insertion sort: ze wordt dan ook nauwelijks gebruikt. Pas wanneer verwisselingen veel tijd kosten, zou deze methode overwogen kunnen worden. Bij insertion sort wordt er weliswaar opgeschoven en niet verwisseld, maar het aantal verplaatsingen is niettemin $O(n^2)$. Grote gegevens verplaatsen kan natuurlijk steeds efficiënt gemaakt worden door wijzers naar deze gegevens te verplaatsen. We vermelden deze methode dan ook enkel omdat hetzelfde principe maar met een efficiënter selectieproces een veel snellere methode oplevert (zie onder).

Het is duidelijk dat de methode *ter plaatse* rangschikt. Ze is echter *niet stabiel*. (Controleer.)

HOOFDSTUK 4
EFFICIËNTE METHODES

Voor grote tabellen worden $O(n^2)$-methodes zeer traag. Gelukkig heeft men verschillende methodes gevonden die substantieel sneller zijn. Het heeft echter enkel zin deze te gebruiken voor echt grote tabellen. Anders loont hun hogere complexiteit niet de moeite, en bovendien zijn eenvoudige methodes in dat geval vaak sneller.

Al deze methodes gebruiken een meer complexe gegevensorganisatie dan de vorige, of hebben een meer ingewikkelde controlestructuur.

4.1 RANGSCHIKKEN DOOR EFFICIËNT SELECTEREN

4.1.1 Heaps

Selection sort was traag omdat telkens het maximum moest gezocht worden in een ongeordende (deel)tabel. Door de elementen in de tabel een betere structuur op te leggen, wordt het misschien mogelijk om het maximum sneller te vinden. Het maximum van een geordende (deel)tabel is natuurlijk ogenblikkelijk te vinden, maar dan zou het rangschikprobleem reeds opgelost zijn. Daarom gebruikt men een structuur die de elementen enigszins ordent, net genoeg om snel het maximum te kunnen vinden. Bovendien zal blijken dat die structuur efficiënt kan opgesteld en aangepast worden.

Deze structuur heet een 'heap'.[1] Een heap is een *complete binaire boom*, waarvan de elementen voldoen aan de *heapvoorwaarde*. Aangezien bomen pas later (uitvoerig) aan bod komen, want het zijn zeer belangrijke gegevensstructuren, bespreken we hier enkel het speciaal geval van een complete binaire boom.

Een boom (zie paragraaf 8.7.1) is een tweedimensionale structuur van *knopen*, die gegevens bevatten. (Hier een sleutel met zijn bijbehorende informatie.) In een lineaire structuur zou elke knoop één voorloper en één opvolger hebben, behalve de eerste en de laatste. In een boom heeft een knoop nog steeds één voorloper (zijn *ouder*), maar

[1] Het heeft weinig zin om het Engelse woord 'heap' te vertalen, omdat het in deze context een heel specifieke gegevensstructuur aanduidt. Er bestaat nog een ander soort heap: bij de implementatie van sommige programmeertalen (zoals C++), noemt men het gedeelte van het computergeheugen waarin dynamisch gecreëerde gegevens ondergebracht worden, eveneens een heap.

kan meerdere opvolgers hebben (zijn *kinderen*). De eerste knoop heeft opnieuw geen voorloper, de 'laatste' kinderen hebben geen opvolgers. Knopen zonder opvolgers noemt men *bladeren*. Elke knoop van een binaire boom heeft hoogstens *twee* opvolgers. Om een boom visueel voor te stellen plaatst men de eerste knoop bovenaan (die heet vreemd genoeg de *wortel* van de boom), komen zijn kinderen op het niveau daaronder, zijn kleinkinderen (kinderen van zijn kinderen) op het volgende (lagere) niveau, enz. Niveau's worden geteld van boven naar beneden, te beginnen bij de wortel, op niveau nul. De *hoogte* van een boom is dan het nummer van het laagste niveau. Ouders en kinderen worden verbonden door (richtingloze) *takken*. Bemerk dat er vanuit de wortel precies één weg is naar elke knoop.

Een gewone binaire boom kan een zeer onregelmatige structuur hebben. Bij een complete binaire boom zijn alle niveau's volledig gevuld (alle knopen hebben twee kinderen), behalve eventueel het laatste. En als dat laatste niveau niet volledig gevuld is, liggen alle knopen ervan zoveel mogelijk links (zonder 'gaten'). Met als gevolg dat er hoogstens één knoop kan zijn met slechts één kind.

Door de regelmatige vorm van een heap bestaat er een eenvoudig verband tussen het aantal knopen n en zijn hoogte h. Op een volledig gevuld niveau i zijn er immers 2^i knopen. Een volledig opgevulde binaire boom van hoogte h heeft dan

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

knopen. Bij een heap is het laatste niveau niet noodzakelijk volledig gevuld. Daarom is

$$2^h - 1 < n \leq 2^{h+1} - 1$$

of ook

$$2^h \leq n < 2^{h+1}$$

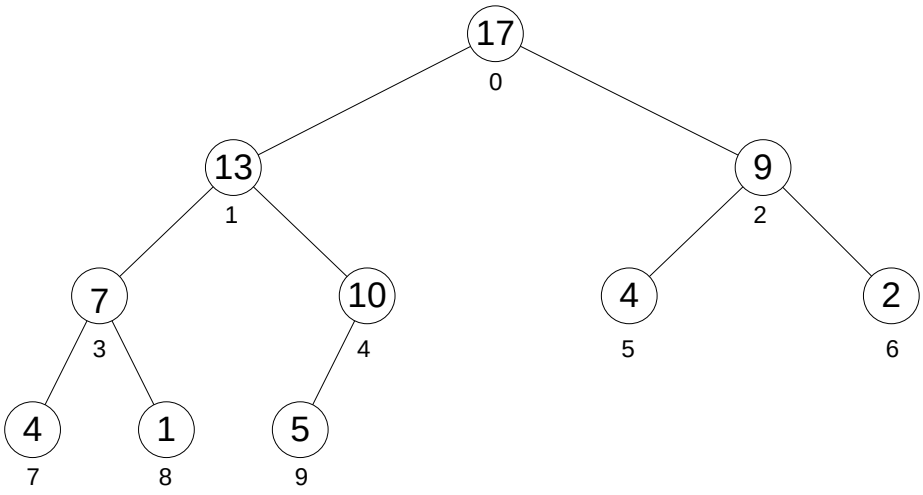
zodat tenslotte, na het nemen van de logaritme

$$h = \lfloor \lg n \rfloor$$

Het ligt voor de hand om een lineaire structuur van gegevens in een tabel op te slaan. Ook de knopen van een boom zou men in een tabel kunnen opslaan, met bij elke knoop de indices van zijn eventuele kinderen. Maar door de regelmatige vorm van een complete binaire boom hebben we deze indices niet nodig. Als we de tabelplaatsen nummeren vanaf nul, dan komt de wortel op plaats 0, zijn kinderen op plaatsen 1 en 2, de kinderen van 1 op plaatsen 3 en 4, die van 2 op 5 en 6, enz. Daardoor is het heel eenvoudig (en efficiënt) om de plaats van de ouder en de kinderen van knoop i ($0 \leq i < n$) te *berekenen*. Zijn ouder staat immers op plaats $\lfloor (i-1)/2 \rfloor$, zijn kinderen op de plaatsen $2i+1$ en $2i+2$.

Een complete binaire boom volstaat niet voor een heap: de sleutels van de gegevens opgeslagen in de knopen moeten ook nog voldoen aan de *heapvoorwaarde*. Bij een (stijgende) *heap* (of maxheap) moet de sleutel bij elke ouder tenminste zo groot zijn

als de sleutel(s) bij zijn kind(eren). (Bemerk dat de onderlinge volgorde van de sleutels bij zijn kinderen onbelangrijk is.) Met als gevolg dat de wortel van een stijgende heap steeds de grootste sleutel bevat. (Waar staat de kleinste?) Op analoge manier definieert men een dalende heap (of minheap). Bemerk dat elke knoop van een heap kan beschouwd worden als de wortel van een deelheap, die alle opvolgers van die knoop bevat. Figuur 4.1 toont een voorbeeld van een stijgende heap. De sleutels staan in de knopen, eronder staan de posities in de tabel.



Figuur 4.1. Stijgende heap.

4.1.2 Bewerkingen op heaps

Een heap is een belangrijke gegevensstructuur, ook (en zelfs vooral) buiten de context van rangschikken. De meeste bewerkingen op heaps zijn immers efficiënt, omdat hun tijdsduur begrensd wordt door de hoogte van de heap, die veel kleiner is dan het aantal elementen.

- *Een element toevoegen.* We beginnen met het toevoegen van een nieuw element g aan een bestaande heap met n knopen. Daartoe is een nieuwe knoop vereist, die enkel kan gecreëerd worden op het laagste niveau, bij tabelindex $n+1$. (Eventueel moet er zelfs een niveau bijkomen.) Als we g in die knoop zouden opslaan is de vorm van de heap in orde, maar niet noodzakelijk de heapvoorwaarde: g kan immers groter zijn dan het element bij zijn ouder. Daarom houden we de nieuwe knoop voorlopig vrij. Zolang de vrije knoop een ouder heeft en g groter is dan het ouderelement, schuiven we dat ouderelement naar beneden in de vrije knoop, zodat de ouder vrijkomt. Wanneer de vrije knoop geen ouder meer heeft (hij is

dan wortel), of als g niet groter is dan het ouderelement, slaan we tenslotte g op in de vrije knoop.

Net zoals bij insertion sort schuiven we kleine elementen op, maar soms is het beter om grote elementen om te wisselen. Eigenlijk ís dit ook dezelfde code als bij insertion sort, alleen gebeurt het tussenvoegen niet in een lineaire reeks elementen, maar op de weg in de heap tussen de nieuwe knoop en de wortel.

Let ook op de dubbele voorwaarde in de herhaling, die ervoor zorgt dat we niet verder dan de wortel komen. Eens te meer kan die vermeden worden door een voorafgaandelijke test op het wortelelement, gesteld dat de heap niet ledig is.

De langste weg die we moeten afleggen is niet groter dan de hoogte van de heap. Toevoegen is dus $O(\lg n)$.

- *Het wortelelement vervangen.* Deze operatie vervangt het wortelelement van een heap door een nieuw element g. Als g kleiner is dan het oorspronkelijke wortelelement, kan het de heapvoorwaarde verstoren. Daarom houden we de wortel voorlopig vrij. Zolang de vrije knoop kinderen heeft en g kleiner is dan een kindelement, schuiven we het grootste kindelement naar boven in de vrije knoop, zodat dat kind vrijkomt. Wanneer de vrije knoop geen kinderen meer heeft of g groter is dan de kindelementen, slaan we tenslotte g op in de vrije knoop.

  In vergelijking met de weg naar boven bij toevoegen, is de weg naar beneden ingewikkelder: elke knoop heeft slechts één ouder, maar wel meestal twee kinderen, zodat we bij iedere stap een splitsing tegenkomen. Per stap gebeuren er dus meer vergelijkingen, maar de langst mogelijke weg blijft gelijk aan de hoogte van de heap. Dus ook deze operatie is $O(\lg n)$.

- *Het wortelelement verwijderen.* Deze operatie is zeer verwant met de vorige. De heap wordt nu kleiner, en om een complete binaire boom te behouden moet de knoop met index n verdwijnen. Het element van die knoop kan voorlopig in de vacante plaats bij de wortel terecht. Dit komt neer op het vervangen van het wortelelement (in de kleiner geworden heap). Uiteraard is deze operatie dan ook $O(\lg n)$.

- *Het element van een willekeurige knoop vervangen.* Als het nieuwe element groter is dan het oude, kan het de heapvoorwaarde enkel verstoren op de weg naar de wortel. Dat wordt dan op analoge manier opgelost als bij toevoegen van een element, alleen beginnen we nu bij de betrokken knoop.

  Als het nieuwe element kleiner is dan het oude, kan het de heapvoorwaarde enkel verstoren in de deelheap waarvan die knoop wortel is. Dat wordt dan op analoge manier opgelost als bij het vervangen van het wortelelement van de volledige heap, maar nu enkel toegepast op de betrokken deelheap.

  In beide gevallen is de afgelegde weg (naar boven of naar beneden) niet langer dan de hoogte van de volledige heap. Het aantal bewerkingen is steeds evenredig met die weglengte, zodat deze operatie opnieuw $O(\lg n)$ is.

4.1.3 Constructie van een heap

- *Door toevoegen.* Een heap kan opgebouwd worden door de elementen een voor een toe te voegen aan een oorspronkelijk ledige heap. Dat wil niet zeggen dat de heaptabel oorspronkelijk ledig moet zijn: een ingevulde tabel kan ter plaatse getransformeerd worden tot een heap. Wanneer immers de eerste i tabelelementen reeds een heap vormen, dan kan tabelelement $i + 1$ hieraan toegevoegd worden als we het tijdelijk uit de tabel halen. (We hebben immers plaats nodig om eventueel op te schuiven - bemerk opnieuw de analogie met tussenvoegen bij insertion sort.)

  Deze heapconstructie vereist dus $n - 1$ keer toevoegen (het eerste element stond meteen goed), en de performantie van toevoegen wordt begrensd door de hoogte van de (groeiende) heap. Bovendien is het mogelijk dat elk toegevoegd element helemaal tot boven in de (voorlopige) heap moet stijgen. (Bedenk eens een voorbeeld.) De sommatie van al die bovengrenzen is dus geen overschatting.

  Stel dat we ook het laatste niveau volledig opvullen, wat zeker een bovengrens oplevert, dan geeft dat in totaal een uitvoeringstijd van

  $$T(n) \leq 2 \cdot 1 + 4 \cdot 2 + 8 \cdot 3 + \cdots + 2^{h-2}(h-2) + 2^{h-1}(h-1) + 2^h h$$

  vergelijken we dit met de afschatting (2.6) waarin we $x = 2$ en $p = h$ stellen dan krijgen we $T(n) \leq 2^{h+1}h$. Nu is $h \leq \ln n$ zodat $2^{h+1} \leq 2n$ en dus geeft dit

  $$T(n) \leq 2n \lg n,$$

  zodat het opbouwen van een heap in het slechtste geval $O(n \lg n)$ wordt.

- *Door samenvoegen van deelheaps.* Een veel efficiëntere methode bouwt de heap van onder naar boven op door deelheaps samen te voegen (Floyd, 1964). Samenvoegen van deelheaps betekent een knoop waarvan beide deelbomen reeds deelheaps zijn, tot wortel te maken van een grotere deelheap die deze deelbomen bevat. Daarbij kan enkel het wortelelement de heapvoorwaarde verstoren. Belangrijk: de heapvoorwaarde moet *enkel* gelden in de nieuwe deelheap. Conflicten hogerop worden hier dus (tijdelijk) genegeerd.

  Deze constructie vereist een reeds ingevulde tabel (of minstens een ingevulde tweede tabelhelft). Stel eens dat alle knopen op een bepaald heapniveau reeds wortel zijn van een deelheap. Eventueel herstel van de heapvoorwaarde gebeurde daarbij enkel naar beneden, zodat hogere niveau's nog steeds de originele tabelelementen bevatten. Elke knoop van het niveau daarboven is dan ouder van twee deelheaps, zodat ze kunnen samengevoegd worden met die knoop als wortel. Wanneer alle knopen van dat niveau wortel van deelheaps geworden zijn, doen we hetzelfde met het niveau daarboven, tot uiteindelijk de volledige heap in orde is.

  Maar hoe komen we onderaan aan de eerste deelheaps? Eenvoudig: één enkele knoop is steeds een heap. Alle knopen op het onderste niveau en die zonder kinderen op het voorlaatste niveau zijn dus al deelheaps. Daarom beginnen we

op het voorlaatste niveau, bij de knopen met kinderen. De volgorde waarin de knopen op een bepaald niveau behandeld worden is onbelangrijk (de deelheaps overlappen niet), maar het is eenvoudigst om dat van rechts naar links te doen, zodat de eerst behandelde knoop tabelindex $n/2$ heeft.

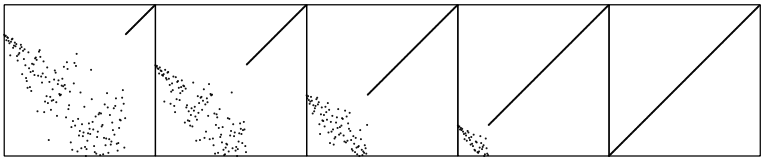
Op het eerste gezicht heeft ook deze constructie een performantie van $O(n \lg n)$, in het slechtste geval. Immers, het werk om twee deelheaps samen te voegen wordt begrensd door hun hoogte, en dat gebeurt $n/2$ maal, met deelheaps van hoogte $O(\lg n)$. Bovendien is het mogelijk dat elke toevoegoperatie de maximale hoeveelheid werk moet verrichten. (Bedenk eens een voorbeeld.) Om een correcte bovengrens te bekomen, moeten we dus opnieuw al die maxima sommeren. De meeste deelheaps zijn echter veel kleiner, en als we wat zorgvuldiger te werk gaan bij de berekening, blijkt de performantie van deze versie $O(n)$.

Het aantal knopen op niveau k van een complete binaire boom is 2^k (behalve eventueel op het laagste niveau), en als we onderstellen dat het laagste niveau volledig opgevuld is (wat zeker een bovengrens is), dan krijgen we, met veronachtzaming van de constante c:

$$T(n) \leq 2^{h-1} \cdot 1 + 2^{h-2} \cdot 2 + \cdots + 4(h-2) + 2(h-1) + 1 \cdot h$$

Gebruiken we (2.7) van pagina 14 dan zien we dat $T(n) \leq 2.2^{h+1}$ waaruit duidelijk blijkt dat $T(n) = O(n)$.

4.1.4 Rangschikken met een heap



Figuur 4.2. heapsort

Bij rangschikken door (eenvoudig) selecteren moesten we alle k resterende elementen overlopen om het maximum te vinden. De tabel was immers ongeordend. Als we die tabel echter eerst in een heap transformeren, vinden we het grootste element in $O(1)$ in plaats van $O(k)$. De rest van de methode blijft ongewijzigd: het grootste element wordt op zijn definitieve plaats gezet door om te wisselen. Door een efficiënte gegevensstructuur te gebruiken wordt een trage methode dus aanzienlijk sneller: ze heet nu ‘heapsort’ (Williams, 1964).

De te rangschikken tabel wordt dus eerst ter plaatse getransformeerd tot een stijgende heap. Door telkens het grootste element uit de heap te verwijderen, bekomen we de elementen in dalende volgorde. Bovendien kunnen we ze achteraan in dezelfde tabel als de heap opslaan, omdat de heap telkens kleiner wordt. Verwijderen van de wortel gebeurt natuurlijk met de gelijknamige heapoperatie.

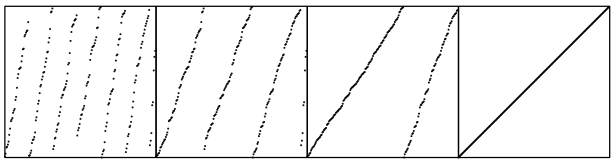
De efficiënte heapconstructie is $O(n)$. Het rangschikken verwijdert $n-1$ keer het wortelelement, en een analoge berekening als bij de eerste heapconstructie leert dat dit in het slechtste geval $O(n \lg n)$ is. (Bedenk eens een voorbeeld waarbij het herstel van de heap telkens een maximale hoeveelheid werk vereist.) De totale uitvoeringstijd is dus ook $O(n \lg n)$: voor grote n is heapsort sneller dan de vorige methodes.

Men kan aantonen dat dit resultaat ook geldt in het beste en het gemiddelde geval: heapsort is dus $\Theta(n \lg n)$. Uit experimenten blijkt bovendien dat dit gedrag zeer consistent is: het gemiddelde geval is nauwelijks sneller dan het slechtste.

Het is duidelijk dat heapsort *ter plaatse* rangschikt. De methode is echter *niet stabiel*, omdat de verplaatsing van gelijke elementen tijdens het opbouwen en herschikken van de heap hun oorspronkelijke volgorde kan wijzigen.

4.2 RANGSCHIKKEN DOOR SAMENVOEGEN

4.2.1 Principe



Figuur 4.3. Mergesort

Een belangrijke methode om problemen efficiënt op te lossen heet 'verdeel-en-heers' ('divide-and-conquer').[2] Ze verdeelt het probleem in een aantal *onafhankelijke* deelproblemen die dus afzonderlijk kunnen opgelost worden. Bovendien past ze op elk deelprobleem dezelfde methode toe. Deze onderverdeling kan natuurlijk niet blijven doorgaan: uiteindelijk is het deelprobleem zo klein dat het eenvoudig op te lossen valt.

Tot zover het 'verdelen'. Het 'heersen' wordt pas mogelijk als we de oplossingen van de deelproblemen kunnen *combineren* tot een oplossing voor het geheel. Dat gebeurt dan in omgekeerde richting (van onder naar boven, 'bottom-up'): de oplossingen voor de triviale problemen worden gecombineerd tot oplossingen voor de problemen die één niveau hoger liggen, die dan weer gebruikt worden voor het daarboven gelegen niveau, enz.

[2] De methode is al zeer oud, en ze wordt niet alleen in technische omgevingen met succes toegepast. Haar oorspronkelijke naam was trouwens 'divide et impera'.

Kunnen we deze methode gebruiken bij het rangschikken van een tabel? Stel dat we de tabel in twee gelijke stukken verdelen, en elk deel apart rangschikken. De deelproblemen zijn dan alvast onafhankelijk. Maar hoe rangschikken we daarmee de volledige tabel? Door de twee gerangschikte deeltabellen *samen te voegen* tot één gerangschikte tabel. In het Engels is samenvoegen 'to merge' zodat deze methode 'mergesort' heet.[3] Om elke deeltabel te rangschikken gebruiken we natuurlijk dezelfde methode. Wanneer stopt het opsplitsen? Bij een deeltabel met lengte één, want die is uiteraard gerangschikt.

Samenvoegen van (stijgend) gerangschikte tabellen kan zeer efficiënt gebeuren door telkens de kleinste elementen van beide deeltabellen met elkaar te vergelijken en het minimum weg te nemen. Zo bijvoorbeeld voor de deeltabellen

$$\left\{ \begin{array}{llll} 431 & 491 & 1027 & \ldots \\ 23 & 491 & 579 & 782 \quad \ldots \end{array} \right.$$

krijgen we

$$23 \left\{ \begin{array}{lll} 431 & 491 & 1027 \quad \ldots \\ 491 & 579 & 782 \quad \ldots \end{array} \right.$$

en vervolgens

$$23 \quad 431 \left\{ \begin{array}{lll} 491 & 1027 & \ldots \\ 491 & 579 & 782 \quad \ldots \end{array} \right.$$

Enzovoort. Als het einde van een van de deeltabellen bereikt wordt, mag men de rest van de andere deeltabel zonder meer kopiëren. Elke test verplaatst één element: het aantal operaties is dus evenredig met de gezamenlijke lengte van beide deeltabellen.

Omdat het resultaat in de plaats moet komen van de deeltabellen, zullen we een van de twee vooraf moeten kopiëren naar een hulptabel. (Het is mogelijk om de hulptabel te vermijden, maar die oplossingen zijn ingewikkeld en van weinig praktisch nut.) Enkel de kleinste deeltabel moet gekopieerd worden.

Bij het samenvoegen kan men ervoor zorgen dat gelijke elementen in dezelfde volgorde blijven staan, eerst deze uit de linkse deeltabel, gevolgd door die uit de rechtse deeltabel: dit samenvoegen is dus *stabiel*.

We beschikken nu over alle elementen om het volledige algoritme te beschrijven. Omdat de deeltabellen met dezelfde methode gerangschikt worden, ligt een recursieve implementatie voor de hand, zoals in de niet-optimale code 4.1.

Ook al wordt er op elk recursieniveau samengevoegd, toch is elke samenvoegoperatie afgewerkt vooraleer er een volgende begint. Er is dus nooit méér dan één hulptabel tegelijk vereist: alle samenvoegoperaties kunnen dezelfde hulptabel gebruiken. Dat is efficiënter dan telkens een lokale hulptabel te creëren en te verwijderen.

[3] Dit is een van de oudste algoritmen om te rangschikken, voor het eerst geïmplementeerd door von Neumann (1945).

```
void mergesort (vector<T> & v,
                int ℓ, int r,
                vector<T> & hulp) {
  // Rangschikt de deelvector v[ℓ..r-1] door samenvoegen
  // Gebruikt daarbij de hulpvector hulp
  if (ℓ < r-1) {
     int m = ℓ+(r-ℓ)/2; //Veiliger dan (ℓ+r)/2
     mergesort (v, ℓ, m, hulp);
     mergesort (v, m, r, hulp);
     merge(v, ℓ, m, r, hulp); //Samenvoegen met hulpvector
  }
}

void mergesort (vector<T> & v) {
  // Rangschikt vector v in stijgende volgorde
  // Gebruikt een hulpvector van halve grootte
  vector<T> h(v.size()/2);
  mergesort (v, 0, v.size(), h);
}
```

Pseudocode 4.1. Recursieve mergesort.

Een kleine opmerking over het aangeven van grenzen is hier op zijn plaats. Men kan een range in een tabel aanduiden door de index van het eerste en van het laatste element, maar maar het is beter om de index *na* het laatste element te nemen, omdat dit overeenkomt met de afspraak die gemaakt wordt bij iteratoren (`end()` wijst niet naar het laatste element, maar wijst een plaats verder), bij het gebruik van de grootte van een vector (`vector::size()` geeft de index voorbij het laatste element), en zo voorts.

4.2.2 Performantie

Hoe efficiënt is mergesort? Laten we voor de eenvoud eens onderstellen dat n een macht van twee is: $n = 2^k$. Aangezien de tabel dan telkens perfect in twee gesplitst wordt, krijgen we

$$T(n) = 2T(n/2) + cn$$

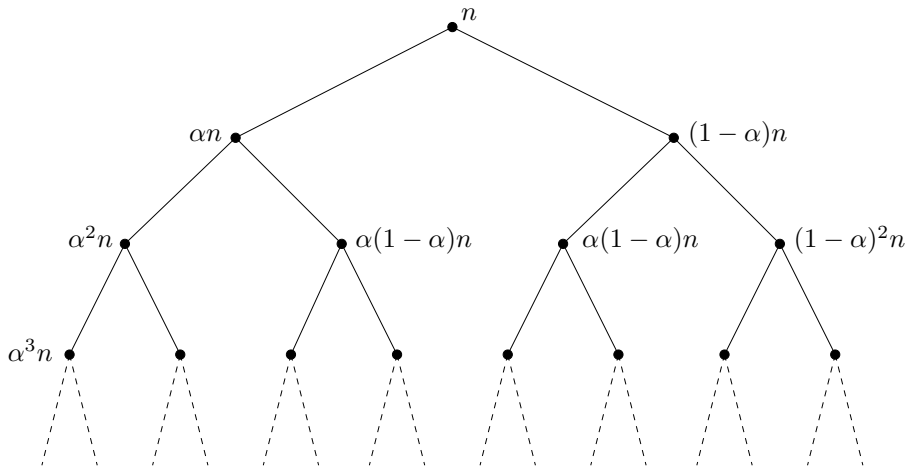
waarbij de term cn het aantal operaties voor het samenvoegen voorstelt (c is een of andere constante). Samenvoegen van twee gerangschikte deeltabellen met lengten n_1 en n_2 is immers $\Theta(n_1 + n_2)$. Om deze recursiebetrekking op te lossen delen we eerst beide leden door n

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + c.$$

Als we $f(n) = T(n)/n$ nemen dan kunnen we (2.5) op pagina 13 toepassen, zodat we krijgen dat $f(n)$ van de orde $O(\lg n)$ is en bijgevolg $T(n)$ $O(n \lg n)$.

Deze efficiëntie wordt gehaald omdat de tabel telkens in twee gelijke delen wordt verdeeld, en omdat het samenvoegen van die delen in lineaire tijd gebeurt. Het is zelfs niet nodig dat de twee delen (nagenoeg) even groot zijn: een *constante grootteverhouding* tussen de delen volstaat. Onderstel bijvoorbeeld dat de grootte van het grootste deel steeds een constante fractie α van het geheel is $(0.5 \leq \alpha < 1)$[4]. Na k verdelingen is de grootte van het grootste deel nog slechts $n\alpha^k$, en we stoppen ten laatste wanneer ze gelijk wordt aan één. Dus is $n\alpha^k \sim 1$, of $k = \lg n / |\lg \alpha|$, zodat $k = O(\lg n)$.

Het resultaat van alle verdelingen kan voorgesteld worden met een binaire boom (zie figuur 4.4). Bij de wortel staat de volledige tabel, bij de kinderen van een knoop de



Figuur 4.4. Partitieboom.

deeltabellen waarin de (deel)tabel van hun ouder werd onderverdeeld. Het aantal tabelelementen op elk niveau van de boom is $O(n)$. Deeltabellen die klein genoeg geworden zijn (ten laatste één element) worden immers niet verder opgesplitst: op de onderste niveau's van de boom wordt het aantal elementen kleiner dan n. Het werk om twee deeltabellen samen te voegen is evenredig met hun gezamenlijke lengte, zodat samenvoegen van alle paren deeltabellen op hetzelfde niveau van de boom $O(n)$ is. De hoogte van de boom is $O(\lg n)$, zodat samenvoegen van alle paren deeltabellen, en dus ook de performantie van mergesort, inderdaad $O(n \lg n)$ is.

Mergesort is dus (asymptotisch) even efficiënt als heapsort, en blijkt zelfs iets sneller met een goede implementatie. Bovendien speelt de oorspronkelijke volgorde van de gegevens bij beide methodes nauwelijks een rol: ook hun beste geval is $\Omega(n \lg n)$. Beide zijn dus $\Theta(n \lg n)$.

[4] Bij mergesort is dit niet direct zinvol; bij andere verdeel-en-heersmethoden kan dergelijke ongelijke verdeling wel voorkomen.

4.2.3 Implementaties

Het oproepen van de recursieve procedures zou eventueel voor enige vertraging kunnen zorgen, zodat er technieken bestaan om de recursie te verwijderen. Met een goede compiler en eventuele processorondersteuning is het verschil miniem, zoniet onbestaande. Bovendien is het resultaat van die transformaties meestal vrij onoverzichtelijk (dus riskant, en alleen te gebruiken indien echt nodig).

Een belangrijke verbetering kan ingevoerd worden bij de stopvoorwaarde. Als de deeltabellen vrij klein worden, is het inderdaad niet meer aangewezen om het volledige recursieve mechanisme te blijven gebruiken. Men kan ze beter rangschikken met insertion sort, dat immers efficiënter is voor korte tabellen dan krachtiger methodes. (Er is nóg een reden om insertion sort te gebruiken, zie later.)

Hoewel het samenvoegen zelf efficiënt is, zorgt het telkens kopiëren van een halve tabel voor veel extra werk. Als er echter plaats is voor een hulptabel van dezelfde grootte als de oorspronkelijke tabel, dan kunnen we *afwisselend* samenvoegen tussen beide tabellen: het nutteloos kopiëren vervalt.

We kunnen eventueel een niet-recursieve versie afleiden uit de recursieve, maar het is eenvoudiger om *rechtstreeks* een niet-recursieve oplossing op te stellen. Wat gebeurt er immers bij de recursieve versie? De tabel wordt in steeds kleinere deeltabellen opgesplitst (de verwerking is dus *top-down*, en pas als ze klein genoeg zijn begint het eigenlijke werk, het samenvoegen van telkens groter wordende deeltabellen. We kunnen echter ook meteen onderaan beginnen (dit is dan *bottom-up* verwerking), en de tabel van begin tot einde overlopen, waarbij we alle paren opeenvolgende elementen samenvoegen tot deeltabellen met lengte twee, waarna we herbeginnen en al die deeltabellen twee aan twee samenvoegen tot deeltabellen met lengte vier, enz. De deeltabellen van deze versie zijn niet noodzakelijk dezelfde als die van de vorige, en ook het samenvoegen gebeurt in een andere volgorde, maar het resultaat is hetzelfde.

Deze niet-recursieve versie heeft dezelfde performantie als de recursieve. De tabel wordt immers $\lceil \lg n \rceil$ maal overlopen, waarbij het samenvoegen van alle deeltabellen samen telkens $\Theta(n)$ is. In totaal is dat dus weer $\Theta(n \lg n)$.

Ook hier kunnen we dezelfde verbeteringen gebruiken als hiervoor: afwisselend samenvoegen tussen twee even grote tabellen, en beginnen met kleine deeltabellen die we vooraf rangschikken met insertion sort.

Een belangrijke eigenschap van mergesort (beide versies) is dat de elementen in de tabel steeds in *sequentiële volgorde* behandeld worden. Dat is gunstig voor de processorcaches. Mergesort is ook de aangewezen methode als de elementen niet in tabellen, maar in gelinkte lijsten of, en belangrijker, op sequentiële bestanden opgeslagen zijn: het is de standaardtechniek voor uitwendig rangschikken. (Zie hoofdstuk 7.)

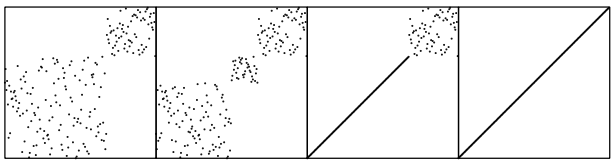
Bovendien zijn beide versies *stabiel*. Immers, het opsplitsen verplaatst geen elementen, dat gebeurt enkel tijdens het samenvoegen, en deze operatie is stabiel. (Ook daarom is het belangrijk om kleine deeltabellen te rangschikken met insertion sort.) Andere

methodes stabiel maken vereist extra plaats en tijd, zodat hun voordelen ten opzichte van mergesort nagenoeg wegvallen.

Het voornaamste nadeel van de methode is dat rangschikken *niet ter plaatse* gebeurt: de grootte van de hulptabel is $O(n)$. (De recursieve versie vereist ook nog een stapel met grootte $\Theta(\lg n)$.) Dat is geen bezwaar als men over voldoende geheugen beschikt.

4.3 RANGSCHIKKEN DOOR ONDERVERDELEN

4.3.1 Principe



Figuur 4.5. Quicksort

Dit is eveneens een soort verdeel-en-heersmethode, al kunnen de deelproblemen niet vooraf bepaald worden. Hierdoor wordt ook de logaritmische afschatting die we vroeger gemaakt hebben voor verdeel-en-heers onbruikbaar. Quicksort wordt ook wel 'partition and exchange sort' genoemd, maar ze bleek zo snel dat haar uitvinder (Hoare, 1962) het meteen over 'quicksort' had. Ook hier wordt de tabel opgesplitst in twee deeltabellen, die onafhankelijk gerangschikt worden met dezelfde methode. Bij mergesort is echter het onderverdelen van de tabel eenvoudig, en vereist het samenvoegen van de twee gerangschikte deeltabellen het meeste werk. Hier is de situatie net omgekeerd: de onderverdeling ('partitie') vraagt nu het meeste werk, het samenvoegen is echter ogenblikkelijk.

Principieel werkt het onderverdelen als volgt: kies een willekeurig element uit de tabel als spilelement (Engels 'pivot') en verdeel ze in twee, zodat het linkse gedeelte enkel elementen bevat die kleiner dan of gelijk zijn aan dat spilelement, en het rechtse enkel elementen die groter of gelijk zijn. In tegenstelling tot mergesort weten we dus niet op voorhand hoe groot beide deeltabellen zullen zijn.

Dat onderverdelen kan zeer snel gebeuren: rechts beginnend, zoeken we een element dat kleiner is dan (of gelijk aan) het spilelement, en van links komend zoeken we een element dat groter is (of gelijk). Dan verwisselen we die twee elementen van plaats, zodat ze nu aan de juiste kant van de tabel staan. Daarna zoeken we zowel rechts als links verder naar twee elementen die aan de verkeerde kant staan, en we verwisselen ze.

Dit gaat door tot beide zoekoperaties elkaar 'ontmoeten': de tabel is dan onderverdeeld in twee deeltabellen. Als we die elk apart kunnen rangschikken, is meteen de volledige tabel gerangschikt. Voor beide delen gebruiken we natuurlijk dezelfde methode, zodat een recursieve oplossing weer voor de hand ligt.

Wanneer we het linkse tabelelement als spilelement kiezen, ziet de oplossing er bijvoorbeeld uit zoals in code 4.2. De binnenste herhalingen van dit algoritme zijn zeer

```
void quicksort (vector<T> & v, int ℓ, int r) {
  // Rangschikt de deelvector v[ℓ..r-1]
  if (ℓ < r-1) {
     // Partitie met v[ℓ] als spilelement
     T pivot = v[ℓ];//Geen T&, geen move!
     int i = ℓ, j = r-1;
     while (v[j] > pivot)
        j--;
     while (i < j) {
        swap(v[i],v[j]);
        i++;
        while (v[i] < pivot)
           i++;
        j--;
        while (v[j] > pivot)
           j--;
     }
     // Recursief rangschikken van beide delen
     quicksort (v, ℓ, j+1);
     quicksort (v, j+1, r);
  }
}

void quicksort (vector<T> & v) {
  quicksort (v, 0, v.size());
}
```

Pseudocode 4.2. Quicksort met spilelement links.

eenvoudig: één sleutelvergelijking en één indexaanpassing. Daardoor is quicksort zo snel.

Ondanks de eenvoud van de code zijn er toch een paar belangrijke details die misschien niet meteen opvallen. Zo gebruiken we geen dubbele voorwaarde en ook geen stopelementen in de binnenste herhalingen. (Ga na of dat nodig is.) Controleer ook of de index j wel degelijk de tabel correct opsplitst. (Is i hier ook niet goed?) Bovendien is er nooit een deel ledig, zodat de recursie niet eindeloos doorgaat. Zou er iets veranderen

mochten we het rechtse tabelelement als spilelement kiezen?

Het omwisselen gebeurt ook voor elementen die *gelijk* zijn aan het spilelement, wat op het eerste gezicht niet nodig is, maar dat ervoor zorgt dat de tabel beter verdeeld wordt als er veel duplicaten aanwezig zijn. Want stel eens dat alle elementen in de op te splitsen tabel gelijk zijn. Als elementen gelijk aan het spilelement niet omgewisseld worden, dan is de opsplitsing zeer onevenwichtig. Doet men dat wel, dan zijn beide stukken even groot. En dat is ideaal bij verdeel-en-heersmethodes. Het lijkt misschien vreemd om zich te bekommeren om een tabel met gelijke elementen. Als een grote tabel echter veel duplicaten bevat, zullen die door de partities vroeg of laat in dezelfde deeltabel terechtkomen. (Voor dergelijke gevallen bestaat er zelfs een partitiemethode die de tabel in drie delen onderverdeelt, waarbij het middenste deel alle elementen bevat die gelijk zijn aan het spilelement. Die staan dan meteen op de juiste plaats, en moeten verder niet meer behandeld worden. Voor meer details, zie Bentley en McIlroy [1].)

4.3.2 Performantie

Hoe efficiënt is quicksort? Het werk om een (deel)tabel te partitioneren is evenredig met haar grootte. De totale uitvoeringstijd hangt voornamelijk af van het resultaat van de partities:

- In het *beste* geval zijn beide delen telkens even groot, zodat we dezelfde betrekking krijgen als bij mergesort (gesteld dat n een macht is van twee, anders geldt dit bij benadering):
$$T(n) = cn + 2T(n/2)$$
Met als resultaat dat $T(n) = O(n \lg n)$. De verborgen constante is gewoonlijk kleiner dan die van de andere $O(n \lg n)$ methodes, zodat een goede implementatie van quicksort het snelst bekende algoritme is om willekeurige sleutels te rangschikken. (Studies bevestigen dat quicksort optimaal is.)

  Gelijke deeltabellen, en dit bij elke partitie, is wel veel gevraagd. Ook een constante grootteverhouding tussen de deeltabellen volstaat (zie hoger), maar dat is evenmin gegarandeerd.

- In het *slechtste* geval echter bestaat een van de twee delen uit slechts één element, en als dat op elk niveau gebeurt blijkt de methode niet sneller dan insertion sort! Er geldt nu immers:
$$T(n) = cn + T(1) + T(n-1)$$
En dus ook
$$\begin{aligned} T(n-1) &= c(n-1) + T(1) + T(n-2) \\ T(n-2) &= c(n-2) + T(1) + T(n-3) \\ &\vdots \\ T(2) &= c(2) + T(1) + T(1) \end{aligned}$$

Eens te meer tellen we al die gelijkheden op, vallen de meeste termen weg, en komt er:

$$T(n) = c(n + (n-1) + \cdots + 3 + 2) + nT(1)$$

Tussen de haakjes staat de som van een rekenkundige reeks, en dus een term evenredig met n^2. De laatste term kunnen we verwaarlozen, omdat $T(1)$ constant is. In elk geval is quicksort nu $O(n^2)$.

Het slechtste geval doet zich voor wanneer de pivot stelselmatig het kleinste of grootste element van de deeltabel is. Dat gebeurt bijvoorbeeld als we de pivot steeds op een vaste plaats kiezen (zoals hierboven), en als de tabel reeds gerangschikt is. En juist voor dit geval is insertion sort $O(n)$.

Maar ook in minder extreme gevallen kan de performantie ongunstig uitvallen. Het volstaat dat de grootte van een van de deeltabellen een kleine constante bedraagt. De pivot op een vaste plaats kiezen is dus gevaarlijk.

- Het *gemiddeld* geval doet zich voor wanneer de pivot met dezelfde waarschijnlijkheid elk element kan zijn. Met als gevolg dat elke grootte van de deeltabellen even waarschijnlijk is.

  De gemiddelde uitvoeringstijd wordt dan[5]

  $$T(n) = cn + \frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i))$$

  Aangezien de meeste termen tweemaal in die som voorkomen, krijgen we

  $$T(n) = cn + \frac{2}{n}\sum_{i=0}^{n-1} T(i)$$

  Die som maakt het moeilijk. Gelukkig bevat ze alle vorige waarden van T ('full history recurrence'), zodat we ze kunnen wegwerken door aftrekken van dezelfde formule voor $T(n-1)$, na vermenigvuldigen met de n uit de noemer:

  $$nT(n) - (n-1)T(n-1) = c(2n-1) + 2T(n-1)$$

  Uitgewerkt wordt dat, na weglating van de constante term $-c$:

  $$nT(n) = (n+1)T(n-1) + 2cn$$

  Om dit te herleiden naar een van de standaardvormen moeten we eerst beide leden nog delen door $n(n+1)$:

  $$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

  Stellen we $f(n) = T(n)/(n+1)$ dan geeft regel (2.4) op pagina 13 dat $f(n)$ zelf $O(\lg n)$ is en $T(n)$ dus $O(n \lg n)$.

  Wanneer is elke grootte van een deeltabel even waarschijnlijk, of de pivot met dezelfde waarschijnlijkheid elk element?

[5] We veronderstellen hier een versie van quicksort die de pivot op zijn juiste plaats met index i zet. Voor een versie die dat niet doet moeten we $T(i-1)$ vervangen door $T(i)$.

  - Als elke permutatie van de invoertabel even waarschijnlijk is, komt elk element met dezelfde waarschijnlijkheid op elke plaats voor. Dan kunnen we gerust de pivot op een vaste plaats kiezen. Deze veronderstelling is echter niet zeer realistisch, maar we kunnen er zelf voor zorgen, door de invoertabel vooraf random te permuteren. Dat gebeurt met een randomgenerator, en vereist $\Theta(n)$.

    Dit moet niet alleen gelden voor de invoertabel, maar ook voor elke volgende deeltabel. De partitiemethode moet dus deze eigenschap intact houden.
  - Een nog eenvoudiger manier kiest de pivot op een random plaats, ook met een randomgenerator. Dan moet er niets meer ondersteld worden over de invoervolgorde. Gemiddeld is de afstand tussen het midden van de tabel en de doelplaats van een random pivot een kwart van de tabellengte.

  In beide gevallen kan men de bovenstaande versie van quicksort (pivot links) nog steeds gebruiken, als men eerst de pivot en het linkse element omwisselt.

- Een random element als pivot kiezen zorgt er zelfs voor dat de performantie van quicksort $O(n \lg n)$ is met een *zeer grote waarschijnlijkheid*, onafhankelijk van de invoervolgorde.

  Om dat aan te tonen maken we weer gebruik van een partitieboom (zie hoger). Op elk niveau van de boom is het aantal elementen $O(n)$, en het werk om alle deeltabellen op elk niveau te partitioneren is dus ook $O(n)$. Nu zal blijken dat de hoogte van die boom met zeer grote waarschijnlijkheid $O(\log n)$ is.

  De hoogte van de boom wordt bepaald door de langste reeks opeenvolgende partities. Gemiddeld is de afstand tussen het midden van de tabel en de doelplaats van een random pivot een kwart van de tabellengte. Dat levert een ruwe schatting voor verwachte boomhoogte van ongeveer $2{,}5 \lg n$, want $\lg(3/4) = -0{,}4\ldots$. Dit verklaart waarom quicksort normaal sneller is dan mergesort: elementen worden wel vaker vergeleken, maar minder verplaatst (een ronde van mergesort verplaatst alle sleutels, maar een ronde van quicksort maakt hoogstens half zoveel swaps als er sleutels zijn, en meestal maar ongeveer de helft daarvan). Bovendien geraken niet alle sleutels even diep in de partitieboom.

  We zullen trachten aan te tonen dat de waarschijnlijkheid dat een tabelelement betrokken is bij veel meer dan $2{,}5 \lg n$ opeenvolgende partities zeer klein is.

  Hierbij maken we gebruik van de theorie van Bernoulliverdelingen, en meer bepaald van (2.8) op pagina 14. Hiervoor moeten we echter onze randomkeuze uitdrukken zodat we twee mogelijke uitkomsten hebben. We nemen een cut-off $c > 0{,}5$, Gebeurtenis A van de Bernoulliverdeling laten we overeenkomen met een pivot waarbij de verhouding in grootte tussen het grootste partitiedeel en het geheel kleiner dan of gelijk aan c is (een gunstige opdeling), gebeurtenis B als we een ongunstige verhouding hebben die groter dan c is. Duidelijk is dat een sleutel in een tabel met grootte n hoogstens kan deelnemen aan $g = \log_{1/c} n$ gunstige opdelingen: om dieper in de boom te geraken moeten er genoeg ongunstige opdelingen zijn. Nu is de kans op een ongunstige opdeling $p = 2 - 2c$. De

kans voor een element om tot op niveau m te geraken is dus kleiner dan

$$\text{Bernoulli}(m-g, m, 2-2c) < 2\left(\frac{(2-2c)em}{m-g}\right)^{m-g}.$$

Deze afschatting is zeker zinloos als $e(2-2c) > 1$. Nemen we echter c voldoende groot, bijvoorbeeld $c = 2^{-1/8}$, dan krijgen we een reële afschatting. Met deze waarde is $g = 8\lg n$. Voor $m = 11g$ vereenvoudigt de uitdrukking tot $((2{,}2 - 2{,}2c)e)^{10g}$. Vermits $(2{,}2 - 2{,}2c)e = 0{,}494822 \sim 1/2$ kunnen we dit benaderen door $(1/2)^{80\lg n} = n^{-80}$. De kans dat een element $88\lg n$ keer aangesproken wordt, 35 keer de verwachte gemiddelde waarde van $2{,}5\lg n$, is dus uiterst miniem.

Wanneer quicksort een randomgenerator gebruikt spreekt men van 'randomized quicksort'. Randomized quicksort behoort tot een interessante en nog steeds groeiende klasse van 'randomized' algoritmen. Vaak zijn ze competitief met de beste deterministische algoritmen, en gewoonlijk heel wat eenvoudiger. Al deze algoritmen maken gebruik van een randomgenerator om een goede gemiddelde performantie te halen, *onafhankelijk* van de waarschijnlijkheidsverdeling van hun invoer. Hun gedrag hangt enkel af van de eigenschappen van de randomgenerator. (Waarvan de kwaliteit vaak te wensen overlaat. Uittesten is de boodschap.)

- *Mediaan-van-drie.* Het ideale spilelement zou natuurlijk de mediaan zijn, het middelste element van de gerangschikte (deel)tabel. Aangezien we dat uiteraard niet kennen, trachten we het statistisch te benaderen via een steekproef: we nemen willekeurig enkele elementen uit de tabel en bepalen hun mediaan. Hoe groter de steekproef, des te nauwkeuriger de schatting, maar ook des te meer werk. Experimenten wezen uit dat drie elementen volstaan, en dat men ze zelfs niet willekeurig hoeft te kiezen. De 'mediaan-van-drie'-methode gebruikt gewoonlijk het linkse, het middelste, en het rechtse element. Opdat het spilelement het kleinste of het grootste element van de (deel)tabel zou zijn, moeten er nu twee van die drie elementen gelijk zijn aan dat minimum of maximum, en om $O(n^2)$-gedrag te vertonen moet dat bovendien bij veel partities gebeuren. Die kans is dus heel klein.

  De twee andere elementen (naast de mediaan) kunnen ook nog als stopelementen fungeren, en daarom rangschikt men deze drie elementen ter plaatse. Bovendien houden we hetspilelement apart door het om te wisselen met het voorlaatste element (we weten het dan staan, zodat we het achteraf op zijn definitieve plaats kunnen zetten). Daarna verdeelt men de rest van de deeltabel. Vóór de recursieve oproepen wordt het spilelement nog op zijn definitieve plaats gezet, via omwisselen. Deze plaats valt dus buiten de volgende partities.

  De gemiddelde performantie van deze methode bepalen is ingewikkelder. Ze blijkt ongeveer 5% beter dan die met een random pivotkeuze. (Een schatting van de mediaan kan immers beter uitvallen dan een willekeurige keuze.)

- De performantie van quicksort kan verder verbeterd worden door net zoals bij

mergesort de recursie te stoppen als de deeltabel klein genoeg geworden is, en ze te rangschikken met insertion sort.

Nog eenvoudiger zou zijn de recursie te stoppen bij kleine deeltabellen, en er verder niets meer mee doen. Op het einde houden dan we een tabel over die nagenoeg gerangschikt is: enkel binnen elk van de vele kleine deeltabellen moeten er eventueel nog elementen verplaatst worden. Insertion sort is de ideale methode voor een dergelijke tabel. Ze achteraf op de volledige tabel toepassen in plaats van op elke kleine deeltabel afzonderlijk zou in principe wat sneller kunnen zijn. Maar de geheugenhiërarchie van moderne processoren maakt lokaliteit van gegevens belangrijk, zodat dit zelfs nadelig zou uitvallen.

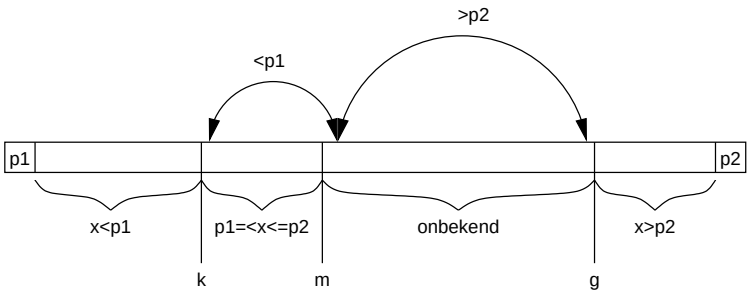
Voor het eventueel verwijderen van de recursie gelden dezelfde overwegingen als bij mergesort. Rechtstreeks een niet-recursieve methode opstellen zoals bij mergesort is echter onmogelijk, omdat het opsplitsen in deeltabellen afhangt van de *inhoud* van de tabel, terwijl ze bij mergesort eenvoudig in twee gedeeld werd.

Quicksort rangschikt ongeveer *ter plaatse*. De methode gebruikt immers een stapel met een gemiddelde grootte van $\Omega(\lg n)$, die echter in het slechtste geval $O(n)$ kan worden! De stapelgrootte kan beperkt worden door de staartrecursie (*'tail recursion'*) te verwijderen, en telkens de kleinste deeltabel recursief te rangschikken. Met als gevolg dat elke deeltabelgrootte op de stapel hoogstens de helft is van de vorige, zodat de stapelgrootte $O(\lg n)$ wordt.

Door de partitie is quicksort echter *niet stabiel*.

4.3.3 Implementatie

De sorteerfunctie van JDK is gebaseerd op een variant van quicksort zoals die beschreven is door Yaroslavskiy in 2009 (zie [21]), *dual pivot sort*. Opgemerkt mag worden dat een volledige theoretische analyse van dit algoritme pas gepubliceerd is in 2015, zie [20].



Figuur 4.6. quicksort met twee spilelementen

Bij dit algoritme wordt gebruik gemaakt van twee spilelementen die uit de volledige rij genomen worden. Hierbij is $p_1 \leq p_2$. De twee spilelementen worden aan het uiteinde

van de rij geplaatst en de rij wordt, met behulp van de drie indices k, m en g ingedeeld in vier stukken:

(1) Elementen kleiner dan p_1, met index $i < k$.
(2) Elementen tussen p_1 en p_2, met index $k \leq i < m$.
(3) Nog niet onderzochte elementen, met index $m \leq i < g$.
(4) Elementen groter dan p_2, met index $i > m$.

Nemen we nu het eerste nog niet onderzochte element, t_m. er zijn nu drie mogelijkheden:

(1) $t_m < p_1$. We verwisselen t_m van plaats met t_k en verhogen zowel k als m met 1.
(2) $t_m > p_2$. We verwisselen t_m van plaats met t_{g-1} en verminderen g met 1.
(3) $p_1 \leq t_m \leq p2$. We verhogen m met 1.

Dit eindigt uiteraard met $m = g$, waarna p_1 verwisseld wordt met t_{k-1} en p_2 met t_m.

Op het eerste gezicht lijkt dit geen verbetering op te leveren tegenover quicksort met 1 pivot: Daar moet elk element van de tabel met die ene pivot vergeleken worden, hier moeten we vaak een tweede vergelijking maken. Bovendien moeten we hier veel meer swaps maken. Toch blijkt het algoritme zo'n 10% sneller.

Twee opmerkingen nog:

(1) Voor tabelfragmenten met minder dan zeventien elementen gebruikt de implementatie insertion sort.
(2) De twee pivotelementen worden gekozen met een variant van mediaan-van-drie. Als ℓ en r de linker- en rechterkant van het te sorteren fragment aanduiden dan kiezen we de elementen $t_{\ell+(i+1)\lfloor(r-\ell-1)/6\rfloor}$, met $i = 0, 1, 2, 3, 4$ en sorteren deze. Als spilelementen nemen we dan het tweede en het vierde element.

4.4 ZOEKEN IN EEN GERANGSCHIKTE TABEL

Zoeken of een bepaalde waarde in een gewone tabel voorkomt is $O(n)$: we kunnen nooit zeker zijn dat een bepaalde niet in een tabel voorkomt tenzij we alle waardes nakijken. Een belangrijke reden om een tabel te sorteren is dat we in een gesorteerd tabel veel sneller kunnen zoeken. Sorteren vraagt meer tijd dan een opzoekoperatie voor één element, maar als er meerdere opzoekingen moeten gebeuren dan kan dit werk toch de moeite lonen. We bekijken even verschillende zoekmethodes.

- *Sequentieel zoeken.* Indien de gezochte sleutel zich *niet* in de tabel bevindt, kan sequentieel zoeken gemiddeld vroeger stoppen, namelijk als we een sleutel vinden die groter is dan de gezochte sleutel (gesteld dat we vooraan beginnen).

Voor sleutels die wél in de tabel voorkomen is deze methode gemiddeld niet beter. (Waarom?) Ze wordt dan ook bijna nooit gebruikt, omdat de twee volgende methodes zoveel beter zijn.

- *Binair zoeken.* Hier vergelijkt men de gezochte sleutel met het middelste element van de tabel, waarna men indien nodig op dezelfde manier verder zoekt in de linkse of rechtse halve deeltabel. Het aantal elementen wordt dus telkens gehalveerd, zodat zoeken in een tabel met n elementen maximaal $\lfloor \lg n \rfloor + 1$ vergelijkingen vereist. Voor duizend elementen zijn dat ongeveer tien vergelijkingen, voor een miljoen nog altijd maar twintig. Binair zoeken is dus $O(\lg n)$, zowel voor een aanwezige als een afwezige sleutel.

  Het principe van deze methode is eenvoudig, maar de implementatie moet zorgvuldig gebeuren. Code 4.3 toont een mogelijke oplossing, die gebruik maakt van linker- en rechterindices ℓ en r, om het gedeelte van de tabel af te bakenen waarin de gezochte sleutel zich zou moeten bevinden.

```
int binairZoeken (const T& s, const vector<T> & v) {
   // Onderstelt v niet ledig.
   // Resultaat als gevonden: index
   //           anders: niet bestaande index v.size()
   int ℓ = 0, r = v.size();
   // Invariant: v[ℓ-1] bestaat niet of is <= s <= v[r-1]
   while (ℓ < r-1) {
      int m = ℓ+(r-ℓ)/2; // ℓ <= m < r
      if (s < v[m])
         r = m;
      else
         ℓ = m;
   }
   return s == v[ℓ] ? ℓ : v.size();
}
```

Pseudocode 4.3. Binair zoeken.

Let op de laatste commentaar: als de gezochte sleutel s in de tabel zit, moet hij groter dan of gelijk zijn aan het element bij de linkerindex, en kleiner dan of gelijk aan dat bij de rechterindex. En deze eigenschap wordt niet gewijzigd door de herhaling (invariant). Indien ℓ gelijk wordt aan r, moet s bij ℓ te vinden zijn, of zit niet in de tabel. (Overtuig u dat de herhaling in alle gevallen stopt.)

Er zijn enige variaties op deze oplossing mogelijk. Zo kunnen we bijvoorbeeld de herhaling onderbreken wanneer het middelste element gelijk is aan de sleutel. Dit komt eigenlijk neer op testen met drie mogelijke resultaten in plaats van twee, zodat dit toch trager kan uitvallen. Ook het aantal en de plaats van de gelijkheidstekens bij het lokaliseren van de sleutel kunnen variëren.

Varianten van binair zoeken kunnen gebruikt worden om verwante problemen efficiënt op te lossen:

- *Zoeken in een cyclisch gerangschikte sequentie.* Een sequentie van de vorm $x_1, x_2, \ldots, x_n$, waarvan alle elementen verschillend zijn, heet cyclisch gerangschikt wanneer als voor een bepaalde i x_i het kleinste element is, en $x_i, x_{i+1}, \ldots, x_n, x_1, \ldots, x_{i-1}$ stijgend gerangschikt zijn. De index i is echter onbekend, en moet gezocht worden.

  Stel dat we twee elementen x_ℓ en x_r vergelijken ($\ell < r$). Als $x_\ell < x_r$ dan kan i niet voldoen aan $\ell < i \leq r$, want x_i is het kleinste element. Als echter $x_\ell > x_r$ dan moet i voldoen aan $\ell < i \leq r$. Door ℓ en r goed te kiezen, kan men telkens de helft van de elementen elimineren, zodat de performantie $O(\lg n)$ wordt.
- *Zoeken in een gerangschikte sequentie van onbekende lengte.* Onderstel dat we een element y moeten zoeken in een stijgend gerangschikte sequentie $x_1, x_2, \ldots$ van onbekende lengte. (Deze elementen worden dan ook niet opgeslagen, maar bijvoorbeeld berekend. Als een element berekenen veel werk vraagt, hebben we er alle belang bij om zo weinig mogelijk elementen te berekenen.) Eerst zoeken we een element dat niet kleiner is dan y. Daartoe vergelijken we eerst y met x_1. Als $y \leq x_1$, dan kan y, indien aanwezig, enkel gelijk zijn aan x_1. Zoniet vergelijken we y met x_2, en daarna indien nodig met x_4, en x_8, enz. In plaats van het bereik telkens te halveren, wordt het verdubbeld, zodat we heel snel voorbij y geraken. Om de index j te vinden waarvoor $y \leq x_j$, zijn er immers slechts $\Theta(\lg j)$ vergelijkingen nodig. Als $j > 1$, dan weten we dat $x_{j/2} < y \leq x_j$, en binair zoeken in dat interval kan y vinden met $O(\lg j)$ bijkomende vergelijkingen. De totale performantie is dus $\Theta(\lg j)$.

- *Interpolerend zoeken.* Binair zoeken maakt geen gebruik van de waarschijnlijkheidsverdeling van de sleutels. Interpolerend zoeken test telkens op de plaats waar de sleutel s verwacht wordt. Die is eenvoudig te bepalen bij uniform verdeelde sleutels.

  De vorige methode moet daartoe slechts lichtjes aangepast worden. In plaats van de index m te berekenen als $\ell + (r - \ell)/2$ vervangen we de factor $1/2$ door de verhouding $(s - v[\ell])/(v[r] - v[\ell])$, waarbij we er natuurlijk voor moeten zorgen dat het resultaat geheel is, en binnen het gedefinieerde deel van de tabel valt. (We veronderstellen dat het verschil van sleutels gedefinieerd werd, met als resultaat een getal.)

  Men kan aantonen dat er nu *gemiddeld* slechts $O(\lg \lg n)$ testen nodig zijn, zowel voor een aanwezige als een afwezige sleutel. (De analyse is ingewikkeld.[6]) Voor een tabel met een miljard sleutels is dat minder dan vijf. Dit is potentieel sneller dan binair zoeken, maar de methode is heel gevoelig voor een ongelijkmatige sleutelverdeling. Voorzichtigheid is dus geboden, want dit komt in de praktijk

[6] Men kan zelfs aantonen dat $\Omega(\lg \lg n)$ een ondergrens is voor de gemiddelde performantie van zoeken met om het even welke methode. In die zin is interpolerend zoeken optimaal.

vaak voor. Er zijn zelfs gevallen waarbij *elke* sleutel in de tabel getest wordt: in het slechtste geval is deze methode $O(n)$. Bij twijfel: gebruik binair zoeken.

De berekeningen in de herhaling zijn ook ingewikkelder dan bij binair zoeken, zodat de tabel al heel groot moet zijn om voordeel te halen uit deze methode. Het klein aantal testen is interessant wanneer sleutels vergelijken veel werk vraagt (sleutels zijn immers niet altijd getallen). Binair zoeken test aanvankelijk tabelindices die ver uit elkaar liggen, terwijl interpolerend zoeken in de te verwachten omgeving van de sleutel blijft. Dat is een voordeel bij virtueel geheugen (minder paginafouten), en processorcaches (zie bijvoorbeeld Graefe [8]). De methode wordt ook wel gebruikt bij zoeken in zeer grote tabellen die in het uitwendig geheugen opgeslagen zijn, waarbij het aantal leesoperaties zo klein mogelijk moet blijven. En wanneer het bereik relatief klein geworden is, gaat men over op binair zoeken.

HOOFDSTUK 5
SPECIALE METHODES

5.1 ONDERGRENS VOOR DE EFFICIËNTIE VAN RANGSCHIKKEN

De efficiënte methodes die we besproken hebben voeren in het slechtste geval $O(n \log n)$ operaties uit om n gegevens te rangschikken. (Enkel quicksort is $O(n^2)$.) Men kan zich afvragen of er wezenlijk snellere methodes bestaan, die misschien nog niet ontdekt zijn. Het antwoord daarop is negatief, zodat (voor grote n) deze algoritmen de snelste algemene methodes blijken te zijn.[1]

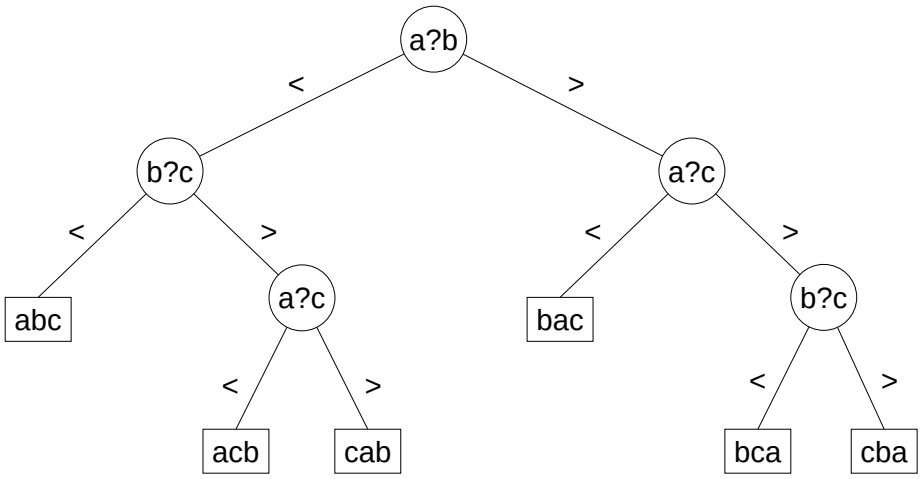
Om dit te kunnen aantonen mag men geen rekening houden met de verschillende werkwijzen van de algoritmen, maar moet men zich concentreren op een gemeenschappelijke eigenschap: ze bepalen de juiste volgorde van de gegevens door hun sleutels onderling te *vergelijken*. Alhoewel er nog andere factoren zijn die de performantie van een algoritme beïnvloeden, beschouwen we hier het aantal sleutelvergelijkingen als een maatstaf voor de snelheid.

Al deze vergelijkingen en hun mogelijke resultaten kunnen voorgesteld worden door een volle binaire boom. De inwendige knopen van die boom hebben steeds twee kinderen, de uitwendige knopen hebben er geen. Als elke inwendige knoop een sleutelvergelijking voorstelt, en elke uitwendige knoop een gerangschikt eindresultaat, dan wordt deze boom een beslissingsboom (decision tree). Voor het eenvoudige geval van insertion sort op drie elementen a, b, c ziet deze boom er uit zoals in figuur 5.1.

Rangschikken begint met de sleutelvergelijking bij de wortel. We dalen af naar het linkse kind als de eerste sleutel kleiner is dan de tweede, anders gaan we naar het rechtse kind. Bij elke inwendige knoop op deze dalende weg gebeurt de overeenkomstige sleutelvergelijking. Als we een blad bereiken, bevat dit het eindresultaat van al deze vergelijkingen: de gerangschikte volgorde.

Voor de eenvoud zullen we onderstellen dat alle sleutels verschillend zijn, echter zonder verlies van algemeenheid. (Als we n verschillende sleutels met een bepaalde snelheid kunnen rangschikken, kunnen we dat zeker minstens even snel als er duplicaten tussen zitten.) De resultaten van elke vergelijking zijn dus groter dan, of kleiner dan. (Als we ook gelijkheid zouden testen, heeft elke vergelijking drie resultaten, en wordt de boom driewegs. Het eindresultaat zal echter hetzelfde zijn.) Bemerk dat deze boom enkel

[1] Algoritmen die rekenen op bepaalde soorten invoer, of adaptieve algoritmen die zich aanpassen aan hun invoer kunnen het beter doen (zie bijvoorbeeld Estivill-Castro en Wood [5]).



Figuur 5.1. Beslissingsboom.

sleutelvergelijkingen bevat: de manier waarop de gegevens verplaatst worden is niet zichtbaar.

Als we geen redundante sleutelvergelijkingen uitvoeren, dan zijn er evenveel uitwendige knopen (bladeren) als mogelijke volgordes. Aangezien er $n!$ mogelijke permutaties zijn van n verschillende sleutels, heeft de boom $n!$ bladeren.

Voor elk mogelijk algoritme dat n verschillende sleutels rangschikt kan een dergelijke boom opgesteld worden, en al die bomen hebben $n!$ bladeren. Het aantal sleutelvergelijkingen dat een algoritme nodig heeft om een bepaalde permutatie te rangschikken, komt dus overeen met de lengte van de afgelegde weg in zijn beslissingsboom.

Aan de hand van deze bomen kunnen we nu aantonen dat rangschikken van n gegevens $\Omega(n \lg n)$ sleutelvergelijkingen vereist, in het slechtste en in het gemiddelde geval:

- *Het slechtste geval.* Het slechtste geval komt overeen met de langst mogelijke weg van de wortel naar een blad, en dat is bij definitie de hoogte van de boom. Om een ondergrens te bekomen hebben we dus de minimale hoogte nodig van een volle binaire boom met $n!$ bladeren.

  Nu heeft een volle binaire boom met hoogte h niet meer dan 2^h bladeren. Dat kan eenvoudig aangetoond worden via inductie op h. Voor $h = 0$ is de eigenschap zeker voldaan. Voor $h > 0$ onderstellen we dat de eigenschap geldt voor elke kleinere hoogte, en tonen aan dat ze ook geldt voor h. Een boom met hoogte h heeft twee deelbomen die niet hoger zijn dan $h - 1$, zodat ze elk maximaal 2^{h-1} bladeren bevatten, wat de eigenschap aantoont.

  Voor onze beslissingsbomen geldt dan dat $2^{h-1} < n! \leq 2^h$, of $h = \lceil \lg n! \rceil$. Als we $n!$ benaderen met behulp van de formule van Stirling (zie hoger) en wat

vereenvoudigen, dan zien we dat $h > \lg \left(\frac{n}{e}\right)^n$, of ook $h > n \lg n - n \lg e$, zodat inderdaad blijkt dat $h = \Omega(n \lg n)$.

Men noemt dit resultaat wel eens de *informatietheoretische ondergrens* van rangschikken, omdat de informatietheorie leert dat wanneer men k verschillende gevallen van elkaar wil onderscheiden via ja/nee vragen (binaire testen), er steeds een geval bestaat waarvoor $\lceil \lg k \rceil$ vragen nodig zijn.

- *Het gemiddelde geval.* Het gemiddelde geval komt overeen met de gemiddelde weglengte van de wortel naar een blad, in een volle binaire boom met $n!$ bladeren. Om opnieuw een ondergrens te bekomen, hebben we nu de minimale gemiddelde weglengte nodig. Daarbij zullen we onderstellen dat elke permutatie van de (verschillende) sleutels even waarschijnlijk is. De verwachtingswaarde van die weglengte wordt dan het rekenkundig gemiddelde van alle mogelijke wegen. De som van alle wegen vanuit de wortel van een boom tot een blad heet de 'uitwendige weglengte'.

  Nu is de uitwendige weglengte in een volle binaire boom met b bladeren minstens $b \lg b$. Ook dat is eenvoudig aan te tonen via inductie op de hoogte h van de boom. Voor $h = 0$ is de eigenschap voldaan, want de wortel is de enige knoop, en een blad. Voor $h > 0$ onderstellen we dat de eigenschap geldt voor alle kleinere hoogten. De wortel heeft nu twee deelbomen, met respectievelijk b_1 en b_2 bladeren. Voor beide deelbomen geldt de eigenschap, want hun hoogte is kleiner dan h. De uitwendige weglengte van de volledige boom is dan minstens $b_1 + b_1 \lg b_1 + b_2 + b_2 \lg b_2$. Met $b = b_1 + b_2$ wordt de kleinst mogelijke waarde van deze uitdrukking bekomen voor $b_1 = b_2 = b/2$, en ze bedraagt inderdaad $b \lg b$.

  De gemiddelde weglengte naar een blad in elke beslissingsboom met $n!$ bladeren is dus $\Omega(\lg n!)$, en zoals bij het slechtste geval is dat ook $\Omega(n \lg n)$.

Rangschikken van n gegevens vereist dus $\Omega(n \lg n)$ sleutelvergelijkingen, zowel in het slechtste als het gemiddelde geval. Dat zijn dan meteen ondergrenzen voor de performantie van rangschikken, als we onderstellen dat een sleutelvergelijking $O(1)$ is. (Voor grote sleutels zoals lange strings is dat niet realistisch.) Deze ondergrenzen gelden niet noodzakelijk voor methodes die rangschikken zonder sleutels te vergelijken, door andere eigenschappen van de sleutels te gebruiken. Voor bepaalde sleutels bestaan er zeer efficiënte methodes, waarvan we enkele bespreken.

5.2 RANGSCHIKKEN DOOR TELLEN

Om gegevens met willekeurige sleutels te rangschikken zit er niets anders op dan deze te vergelijken. Als de sleutels echter getallen zijn (of aldus kunnen beschouwd worden), dan krijgen we meer mogelijkheden.

Rangschikken door tellen (*c*ounting sort, distribution counting, Seward, 1954) vereist dat de sleutels *gehele getallen* zijn, die bovendien tot een *beperkt interval* behoren. Voor elk getal kunnen we dan efficiënt berekenen hoeveel getallen eraan vooraf moeten gaan, zodat we het meteen op zijn juiste plaats kunnen zetten. (Daarom wordt de methode soms als een speciaal geval van bucket sort beschouwd, zie onder.) Aangezien het aantal getallen n gewoonlijk (veel) groter is dan het interval, zullen er nogal wat duplicaten zijn. Het is dus belangrijk om de methode stabiel te maken.

Men gebruikt uiteraard een frequentietabel om te *tellen* hoeveel duplicaten er van elk getal in het interval zijn. (Vandaar de naam van de methode.) Daarmee berekent men de hoogste index van de duplicaten van elk getal in de gerangschikte tabel, en die wordt gebruikt om elk getal meteen op zijn plaats te zetten in een tweede tabel.

Als er k verschillende sleutels zijn dan is de performantie $\Theta(n+k)$. We moeten immers tweemaal alle sleutels overlopen, en zowel de initialisatie van de frequentietabel als de berekening van de hoogste indices zijn $\Theta(k)$. Als $k = O(n)$, dan is dit algoritme dus $\Theta(n)$. De ondergrens voor (algemeen) rangschikken is hier immers niet van toepassing, omdat sleutels niet onderling vergeleken worden. (De bewerkingen op gehele sleutels uit een beperkt bereik mogen we $O(1)$ onderstellen.)

Dit algoritme is *stabiel*. Het rangschikt echter *niet ter plaatse*. Er zijn immers twee hulptabellen nodig: een (kleinere) frequentietabel en een nieuwe tabel die even groot is als de oorspronkelijke.

5.3 RADIX SORT

Deze methode werkt niet met de sleutels in hun geheel, maar deelt ze op in afzonderlijke elementen, die achtereenvolgens gebruikt worden. Sommige sleutels zijn van nature geschikt voor een dergelijke aanpak, zoals strings of vectoren die lexicografisch moeten gerangschikt worden, of ook sleutels die uit heterogene elementen bestaan (datums bijvoorbeeld). Maar eigenlijk kan elk binair patroon wel opgedeeld worden. Dat geldt dus ook voor gehele getallen, als ze ongeschikt zijn voor telsorteren. Stel bijvoorbeeld dat de bovengrens van hun interval niet zo klein is, maar bijvoorbeeld $k = O(n^2)$. Dan is de performantie van telsorteren meteen $O(n^2)$, nog afgezien van het feit dat de frequentietabel veel te groot wordt.

Radix sort beschouwt sleutels als getallen in een m-tallig stelsel, en rangschikt op de 'cijfers' van deze voorstelling. (We onderstellen hier homogene sleutelelementen.) Vandaar ook de naam van de methode: de 'radix' m is de basis van het gebruikte talstelsel. Voorzichtigheid is geboden wanneer men de performantie van radix sort vergelijkt met methodes die sleutels vergelijken. Een sleutelvergelijking werd steeds $O(1)$ ondersteld, onafhankelijk van de lengte van de sleutels. Omdat hier cijfers vergeleken worden, zal het totaal aantal cijfers in alle sleutels belangrijker zijn dan het aantal sleutels n.

Er zijn twee versies van radix sort: de ene behandelt de cijfers van zijn sleutels van links naar rechts (van meest significant naar minst significant), de andere doet dat in omgekeerde volgorde.

5.3.1 Van links naar rechts

5.3.1.1 Binaire quicksort

De officiële naam van deze methode is 'radix-exchange sort' (Hildebrandt en Isbitz, 1959). Het is een eenvoudige variant van quicksort, die zelfs iets vroeger ontdekt werd. Zoals bij quicksort wordt de tabel in twee gepartitioneerd, maar nu volgens de i-de bit van de sleutels. Op beide deeltabellen wordt de methode dan recursief toegepast, nu met bit $i + 1$. De kans op ontaarde partities is groot: het gebeurt vaak dat de i-de bit van veel (zoniet alle) sleutels hetzelfde is. Gelukkig wordt het aantal recursieve oproepen steeds beperkt door de grootste bitlengte van de sleutels. Voor deeltabellen die voldoende klein geworden zijn, kunnen we zoals gewoonlijk een andere methode gebruiken.

Van elke sleutel moeten maar zoveel bits getest worden als nodig om hem te onderscheiden van de andere. Dat is een voordeel als de bits random zijn, maar een nadeel bij sleutels met lange identieke prefixen. In het slechtste geval, als alle sleutels gelijk zijn, worden al hun bits getest. De performantie is dan lineair, niet in het aantal sleutels, maar in het totaal aantal sleutelbits. Omdat er vaak minder bits getest worden, wordt de performantie dan sublineair. Men kan aantonen dat er gemiddeld $n \lg n$ bits getest worden, bij sleutels met random bits. (De kans op een nagenoeg evenwichtige partitie is dan immers veel groter dan bij standaard quicksort.) Zowel binaire als standaard quicksort zijn dus efficiënt voor sleutels met random bits. (De eerste methode moet de bits uit de sleutel halen, de tweede moet sleutels vergelijken.) Standaard quicksort reageert echter beter op sleutels met bits die niet random zijn.

5.3.1.2 MSD radix sort

Deze *M(ost)S(ignificant)D(igit) radix sort* gebruikt hetzelfde principe als binaire quicksort, maar met een radix m in plaats van twee. Met als gevolg dat de tabel nu telkens in m delen gepartitioneerd wordt. En op elk van die delen past men de methode recursief toe, met het volgende cijfer.

Het is best mogelijk dat in de ene deelgroep minder cijfers moeten onderzocht worden om de sleutels te rangschikken dan in de andere. Deze methode is dan ook zeer geschikt voor sleutels van verschillende lengte.

Aangezien de cijfers als gehele getallen kunnen beschouwd worden, die behoren tot een beperkt interval (tussen 0 en $m - 1$), ligt het voor de hand om ze te rangschikken met telsorteren. Dat is niet alleen zeer snel, maar bovendien stabiel, zodat de volledige methode *stabiel* wordt. Ze rangschikt dan natuurlijk *niet ter plaatste*.

Telsorteren vereist een frequentietabel met grootte m, en een hulptabel voor de te rangschikken sleutels. Alle deelgroepen kunnen ondergebracht worden in één hulptabel met grootte n, waarvan elke recursieve oproep zijn eigen deel gebruikt. En dat gebeurt natuurlijk ook met de oorspronkelijke tabel, die afwisselend met de hulptabel gebruikt wordt, zoals bij mergesort. Elk recursieniveau vereist echter een eigen frequentietabel (nadien fungerend als partitietabel), die dus lokaal moet blijven.

Als de radix m voldoende groot gekozen wordt, kunnen de sleutels reeds nagenoeg gerangschikt zijn na partitie op het eerste cijfer. (Daarom wordt de methode soms als een speciaal geval van bucket sort beschouwd, zie onder.) De meeste deelgroepen zijn dan klein, en hun aantal is groot, zodat het belangrijk is dat hiervoor een aangepaste methode gebruikt wordt. (Die liefst stabiel is: insertion sort dus.)

Wanneer een deelgroep voldoende groot is om verder radix sort toe te passen, moet men eventueel de radix aanpassen om te vermijden dat hij groter is dan het aantal elementen in de groep. Zoniet zijn er veel ledige deelgroepen, waarvan men de tellers toch moet initialiseren, zodat veel tijd (en plaats) verspild wordt. Grote aantallen ledige deelgroepen komen trouwens vaak voor bij radix sort, bijvoorbeeld door een ongelijkmatige verdeling van bepaalde cijfers in de sleutels.

In plaats van de radix aan te passen om veel ledige deelgroepen te vermijden kan men ook de *bin-span heuristic* gebruiken. Daarbij houdt men tijdens de telfase de kleinste en de grootste waarde bij in elke deelgroep. De recursie op een deelgroep voorziet dan enkel partitiegroepen binnen dat bereik (met eventueel enkele speciale gevallen).

De diepte van de recursiestapel wordt ook hier beperkt door de grootste sleutellengte, zodat het niet nodig is om de grootste groepen laatst te behandelen, zoals bij quicksort.

Zoals bij binaire quicksort is het aantal cijfervergelijkingen in het slechtste geval lineair in het totaal aantal cijfers, en vaak, bijvoorbeeld voor sleutels met random cijfers, sublineair. Maar een deel van de performantie is evenredig met m, en heeft niets te maken met de sleutels. Immers, de methode maakt minstens eenmaal gebruik van telsorteren, en vereist dus minimaal $2(n + m)$ operaties. (Bij kleine deeltabellen kan m veel groter zijn dan n.) Wanneer de radix steeds kleiner blijft dan de grootte van de (deel)tabel, kan men aantonen dat de gemiddelde performantie $O(n \log_m n)$ bedraagt, voor sleutels met random cijfers. Voor grote m is $\log_m n$ klein, zodat de methode in de praktijk gemiddeld lineair is.

5.3.1.3 Ternaire radix quicksort

Quicksort met een ternaire partitiemethode verdeelt de tabel telkens in drie, met in het middendeel de sleutels die gelijk zijn aan de pivot. De recursie moet dan nog enkel toegepast worden op het linkse en rechtse deel. Ternaire radix quicksort gebruikt hetzelfde principe, maar verdeelt op grond van een cijfer (Bentley en Sedgewick [2]). De recursie moet nu toegepast worden op de drie delen, maar enkel voor het middendeel gebruikt men het volgende cijfer.

Men kan aantonen dat de methode gemiddeld $O(n \lg n)$ cijfers vergelijkt, bij sleutels met random cijfers. (Standaard quicksort doet gemiddeld evenveel vergelijkingen, maar van sleutels.)

Voor strings blijkt deze methode beter dan zowel standaard quicksort als MSD radix sort. Men kan ze beschouwen als een combinatie van beide. Een nadeel ten opzichte van MSD radix sort is dat de meerwegspartitie op grote deeltabellen ontbreekt. Die moet hier gesimuleerd worden via opeenvolgende ternaire partities, wat meer omwisselingen vergt. Een voordeel is dat er geen hulptabel en geen frequentietabel nodig zijn, en dat een groot aantal ledige partitiedelen bij kleinere deeltabellen vermeden wordt.

Een hybridische methode met een zeer goede performantie gebruikt MSD radix sort voor grote deeltabellen, en ternaire radix quicksort met een kleinere radix voor kleinere deeltabellen. De eerste zorgt ervoor dat er meerwegs opgedeeld wordt, de tweede vermijdt een groot aantal ledige partitiedelen.

5.3.2 Van rechts naar links

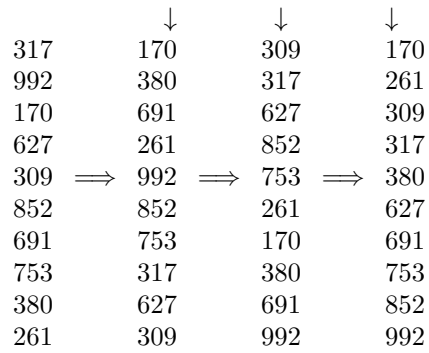
Deze versie gaat omgekeerd te werk, en begint bij het minst significante cijfer van de sleutels ('straight radix sort', 'L(east)S(ignificant)D(igit) radix sort', Seward, 1954). Ze is dus niet zeer geschikt voor sleutels met verschillende lengte, alhoewel dit enigszins kan verholpen worden (zie onder). De werkwijze is op het eerste gezicht wat vreemd, omdat ze tijd spendeert aan cijfers die geen invloed hebben op het resultaat. (De methode is al vrij oud: ze werd vroeger zelfs gebruikt om machinaal ponskaarten te rangschikken.)

Eerst rangschikt men alle sleutels volgens het laatste cijfer, waarbij sleutels met hetzelfde cijfer in dezelfde volgorde moeten blijven staan. Anders gezegd, het rangschikken op een cijfer moet *stabiel* zijn. (Bij de MSD radix sort is stabiliteit mooi meegenomen, maar niet vereist, zoals hier.) Daarna rangschikt men opnieuw alle sleutels volgens het voorlaatste cijfer, enz. Dit wordt herhaald tot alle cijfers behandeld zijn. Verrassend genoeg blijken de sleutels dan gerangschikt. Het voorbeeld in figuur 5.2 illustreert de werkwijze.

Dat de methode juist is kan men aantonen door inductie op het gerangschikte cijfer, te beginnen vanaf links: een sleutel komt vóór een andere, omdat zijn huidig cijfer kleiner is dan dat van de andere sleutel, of omdat beide cijfers gelijk zijn maar de sleutel vóór de andere geplaatst werd bij rangschikken op het vorige cijfer (en door de stabiliteit werd die volgorde niet verstoord). Op dat cijfer kan men dezelfde redenering herhalen, enz.

In principe voldoet elke methode om de sleutels volgens een bepaald cijfer te rangschikken, zolang ze maar stabiel is. Als de basis van het talstelsel m niet te groot is, ligt rangschikken door tellen voor de hand. Er is dan wel een frequentietabel met grootte m en een hulptabel met grootte n vereist.

Het is ook mogelijk om slechts de (laatste) helft van een LSD radix sort te doen. Een



Figuur 5.2. Radix sort van rechts naar links

tabel waarvan de sleutels gerangschikt werden op de hoogste helft van hun cijfers is immers nagenoeg gerangschikt, met enkel nog lokale inversies. Die worden nadien efficiënt verwijderd door insertion sort op de volledige tabel. (Voor sleutels met verschillende lengte wordt het aantal te rangschikken cijfers begrensd door de kortste sleutel. Hopelijk zijn de deelgroepen dan niet te groot, anders is deze methode onbruikbaar.)

Hoe snel is LSD radix sort? Als het (vast) aantal cijfers van de sleutels d is, wordt er d maal telsorteren toegepast, en is de performantie $O(dn + dm)$. Voor $m = O(n)$ is dat dus ook $O(n)$.

Hoe kiezen we m? Groot genoeg zodat d klein wordt, maar toch beperkt zodat de frequentietabel niet te groot uitvalt. Bovendien moeten de cijfers uit de m-tallige voorstelling gemakkelijk te bepalen zijn. Bytes of veelvouden ervan zijn dus vaak een goede keuze.

Stel bijvoorbeeld dat we een miljoen gehele getallen van 64-bit moeten rangschikken. Met $m = 2^{16}$ is $d = 4$, zodat de performantie van nagenoeg $O(4n)$ van de methode beter uitvalt dan de $O(20n)$ van de $O(n \log n)$ methodes. De hulptabellen zijn echter al vrij groot. De binnenste herhaling van het algoritme duurt wel langer dan bijvoorbeeld die van quicksort, zodat de snelheidswinst vaak minder spectaculair is dan verwacht. (De O-notatie houdt immers geen rekening met constante factoren.) Bovendien hebben quicksort en mergesort een betere lokaliteit van hun gegevens, met als gevolg minder cachefouten.

Omdat de afzonderlijke cijfers gerangschikt worden via tellen, werkt de methode dus *niet ter plaatse*, maar ze is wel *stabiel*.

5.4 BUCKET SORT

Deze methode onderstelt dat de sleutels reële getallen zijn, met een uniforme waar-

schijnlijkheidsverdeling over een bepaald interval. Dat interval wordt dan onderverdeeld in m gelijke delen ('buckets'), zodat nagaan in welk deelinterval een sleutel valt $O(1)$ is. De te rangschikken sleutels worden verdeeld over de deelintervallen, en door de uniforme waarschijnlijkheidsverdeling bevatten die elk gemiddeld evenveel sleutels. Als men de sleutels in elk deelinterval rangschikt, is het probleem opgelost.

Door m voldoende groot te kiezen, en bovendien evenredig met n ($m = \Theta(n)$), zorgt men ervoor dat elk deelinterval gemiddeld slechts een klein aantal sleutels bevat, dat onafhankelijk is van n. Het ligt dan voor de hand om elk deelinterval te rangschikken met insertion sort.

Aangezien het aantal sleutels dat in een deelinterval terechtkomt onbekend is (en maximaal n), moet men elk deelinterval implementeren met een gelinkte lijst of een dynamische tabel. Alternatief is om te werken zoals bij telsort: eerst doorloopt men de tabel om het aantal elementen in elke bucket te tellen, daarna weet men welke deel van de doeltabel met elke bucket overeenkomt.

Wat is de performantie van deze methode? Ze moet eerst $\Theta(n)$ deelintervallen initialiseren, dan de sleutels over die deelintervallen verdelen, dan de deelintervallen afzonderlijk rangschikken, en ze tenslotte in de juiste volgorde terugkopiëren naar de oorspronkelijke tabel. Behalve het rangschikken van de deelintervallen zijn al deze termen $\Theta(n)$, en als n_i het aantal sleutels in deelinterval i voorstelt, dan is de vereiste tijd

$$T(n) = \Theta(n) + \sum_{i=1}^{m} O(n_i^2)$$

De tweede term hangt natuurlijk af van de sleutelverdeling:

- In het *slechtste geval* komen alle sleutels in hetzelfde deelinterval terecht. De performantie wordt dan bepaald door de tweede term, en is $O(n^2)$. Aangezien de kans p dat een sleutel in een deelinterval valt gelijk is aan $1/m = \Theta(1/n)$, en onafhankelijk van waar de andere sleutels terechtkomen, wordt de waarschijnlijkheid voor dat slechtste geval $\Theta(1/n^n)$, bijzonder klein dus. Maar ook in minder extreme gevallen kan de methode slecht performeren, als het aantal elementen in een of meerdere deelintervallen te groot uitvalt.
- In het *gemiddelde geval* is de methode echter $\Theta(n)$. De gemiddelde waarde van $T(n)$ is immers

  $$E[T(n)] = \Theta(n) + \sum_{i=1}^{m} E[O(n_i^2)]$$

  waarbij we gebruik maken van de linearieit van de gemiddelde waarde.[2] Aangezien de definitie van een bovengrens ook een constante bevat, kunnen we die-

[2] Voor n toevalsvariabelen x_i (die niet noodzakelijk onafhankelijk moeten zijn) en constanten c_i geldt dat $E\left[\sum_{i=1}^{n} c_i x_i\right] = \sum_{i=1}^{n} c_i E[x_i]$.

zelfde eigenschap gebruiken om E en O om te wisselen:

$$E[T(n)] = \Theta(n) + \sum_{i=1}^{m} O(E[n_i^2])$$

Of een sleutel in deelinterval i terechtkomt of niet, is een toevalsexperiment met twee resultaten (een Bernoulli trial). Die experimenten zijn onafhankelijk, en worden n keer herhaald. Nu is toevalsvariabele n_i het aantal keer dat het experiment gunstig uitviel, en ze heeft dus een binomiale waarschijnlijkheidsverdeling. De kans dat n_i gelijk is aan k is dan

$$\Pr\{n_i = k\} = \binom{n}{k} p^k (1-p)^{n-k}$$

De gemiddelde waarde van deze verdeling is $E[n_i] = np$ en de variantie $V[n_i] = np(1-p)$. Omdat $E[n_i^2] = V[n_i] + E^2[n_i]$ krijgen we dat $E[n_i^2] = np(np + 1 - p)$, of ook, met $p = \Theta(1/n)$, dat $E[n_i^2] = \Theta(1 - 1/n)$. Daarmee bekomen we tenslotte dat

$$E[T(n)] = \Theta(n) + \sum_{i=1}^{m} \Theta(1 - 1/n) = \Theta(n)$$

Bemerk dat bucket sort niet enkel gemiddeld lineair is als de sleutels uniform verdeeld zijn. Het is voldoende dat het deelinterval van een sleutel in $O(1)$ kan bepaald worden, en dat het rangschikken van alle deelintervallen samen gemiddeld lineair is, m.a.w. als $E\left[\sum_{i=1}^{m} O(n_i^2)\right] = \Theta(n)$.

De methode rangschikt duidelijk *niet ter plaatse*, want ze gebruikt een tabel van deelintervallen. Ze is echter wel *stabiel*, omdat insertion sort dat ook is.

HOOFDSTUK 6
DE SELECTIE-OPERATIE

Een belangrijke toepassing, verwant met rangschikken, is het zoeken naar de *mediaan* van een aantal elementen. Dat is het middelste element als we die elementen zouden rangschikken: er zijn evenveel elementen kleiner dan de mediaan als er groter zijn. (Als het aantal elementen even is, zijn er twee medianen.) De mediaan wordt vaak gebruikt in de statistiek.

Natuurlijk kan men dit probleem oplossen door de elementen werkelijk te rangschikken en het middelste te nemen, maar er zijn snellere methodes. De mediaan zoeken is eigenlijk een speciaal geval van de 'selectie'-operatie: zoek het k-de kleinste element van n elementen ($1 \leq k \leq n$).

Geen enkel algoritme kan garanderen dat een element het k-de kleinste is, zonder de $k-1$ kleinere elementen gevonden te hebben (en dus de $n-k$ grotere), zodat de selectiealgoritmen gewoonlijk zonder veel extra moeite de k kleinste elementen opleveren (maar niet noodzakelijk in volgorde).

Afhankelijk van de waarde van k, zijn er een aantal mogelijkheden:

(1) Voor zeer kleine k kunnen we gerust k maal het kleinste element zoeken, en het op zijn juiste plaats zetten, zoals bij selection sort. Deze operatie is lineair, zodat de performantie $\Theta(kn)$ wordt. Deze methode is ook van toepassing bij grote k, nagenoeg gelijk aan n (dan zoekt men uiteraard telkens het grootste element). De performantie is dan $\Theta((n-k)n)$.
Speciale gevallen hiervan zijn het minimum ($k = 1$) of het maximum ($k = n$) van n elementen zoeken. Daarvoor zijn steeds $n-1$ sleutelvergelijkingen vereist. Wanneer men echter *tegelijk* het minimum en het maximum nodig heeft, volstaan $\lceil 3n/2 \rceil - 2$ sleutelvergelijkingen. Daartoe neemt men telkens een paar elementen, die eerst *onderling* vergeleken worden. Het grootste wordt daarna vergeleken met het voorlopig maximum, het kleinste met het voorlopig minimum. (Telkens meer dan twee elementen samen nemen kan dit resultaat echter niet verbeteren.) De recursieve oplossing heeft trouwens dezelfde (asymptotische) performantie.
Nog een ander interessant speciaal geval is zoeken naar de grootste twee van n elementen. Die kunnen samen gevonden worden met slechts $n - 2 + \lceil \lg n \rceil$ sleutelvergelijkingen, door een collectie kandidaten voor het tweede grootste element bij te houden.

(2) a. Voor ietwat grotere k, kan men beter een efficiëntere methode gebruiken om telkens het kleinste element te vinden, en dus de k eerste stappen van

heapsort toepassen (met een dalende heap). Dat geeft een performantie van $O(n + k \lg n)$.

b. Een andere mogelijkheid bestaat erin een stijgende heap bij te houden met de voorlopig k kleinste elementen. Die wordt geïnitialiseerd met de eerste k elementen, en de resterende elementen worden één voor één vergeleken met het grootste element in de heap. Als een element kleiner is dan dit maximum, dan hoort het thuis in de heap en vervangen we de wortel. Uiteindelijk zal de heap de k kleinste elementen bevatten, met het gezochte k-de kleinste element bij de wortel. De performantie is nu $O(k + (n - k) \lg k)$.

(3) Voor nog grotere k bestaat er echter een methode die gemiddeld *lineair* is, gebaseerd op de partitie van quicksort (eveneens Hoare, 1962). Deze partitie verdeelt immers de tabel in de elementen kleiner dan (of gelijk aan) het spilelement, en deze groter dan (of gelijk aan) het spilelement. We hebben hier wel een implementatie van de partitie nodig die het spilelement op zijn definitieve positie i plaatst ($1 \leq i \leq n$). Om het k-de kleinste element te vinden gaan we dan als volgt te werk:

- Als $k = i$, dan hebben we wat we zochten.
- Als $k < i$ dan moeten we verder zoeken in de linkse deeltabel, en eveneens naar het k-de kleinste element.
- Als $k > i$ dan moeten we verder zoeken in de rechtse deeltabel, maar nu naar het $(k - i)$-de kleinste element.

Het zoeken in de deeltabellen gebeurt natuurlijk weer via recursie. (De methode is echter niet écht recursief, want deze staartrecursie kan gemakkelijk verwijderd worden.)

Net zoals bij quicksort hangt de performantie af van het resultaat van de partities, maar daarnaast ook van de positie van k ten opzichte van die van het spilelement, in al deze partities:

- In het *beste* geval zijn de twee delen telkens even groot. Als we voor de eenvoud onderstellen dat n een macht van twee is, krijgen we

$$T(n) = cn + T(n/2)$$

En als deze halvering stelselmatig gebeurt, en k pas op het einde gelijk wordt aan de spilelementpositie i (dat is zeker een bovengrens), geldt ook dat

$$\begin{aligned} T(n/2) &= c(n/2) + T(n/4) \\ T(n/4) &= c(n/4) + T(n/8) \\ T(n/8) &= c(n/8) + T(n/16) \\ &\vdots \\ T(2) &= c(2) + T(1) \end{aligned}$$

Alles optellen geeft dan dat

$$T(n) = c(n + n/2 + \cdots + 8 + 4 + 2) + T(1)$$

De som tussen de haakjes is gelijk aan $2n - 2$, zodat $T(n) = O(n)$.

- In het *slechtste* geval echter krijgen we

$$T(n) = cn + T(n-1)$$

Als we veronderstellen dat k steeds in het grootste deel valt, en pas op het einde gelijk wordt aan de spilelementpositie i, dan geldt voor alle volgende partities dat

$$T(n-1) = c(n-1) + T(n-2)$$

$$\vdots$$

$$T(2) = c(2) + T(1)$$

Zodat tenslotte blijkt dat

$$T(n) = c(n + (n-1) + \cdots + 4 + 3 + 2) + T(1)$$

Dit is hetzelfde $O(n^2)$ gedrag als bij quicksort. En opnieuw kan dit grotendeels vermeden worden door het spilelement zorgvuldig te kiezen.

- Het *gemiddeld* geval is hopelijk niet zo slecht. Voor de eenvoud van de analyse onderstellen we opnieuw dat het *spilelement random* gekozen wordt. Bovendien moeten we dan geen veronderstellingen maken over de waarschijnlijkheidsverdeling van de invoerpermutaties. Aangezien elke waarde voor de uiteindelijke spilelementpositie i even waarschijnlijk is, wordt de gemiddelde tijd om het k-de kleinste element uit n elementen te vinden dan

$$T(n,k) = cn + \frac{1}{n}\left(\sum_{i=1}^{k-1} T(n-i, k-i) + 0 + \sum_{i=k+1}^{n} T(i-1, k)\right)$$

De situatie in de deeltabellen is immers analoog aan die in de volledige tabel, en er moet niets meer gebeuren als $i = k$ (c is een constante).

De waarde van k die $T(n,k)$ maximaal maakt geeft ons dan een bovengrens voor $T(n)$:

$$T(n) \leq cn + \frac{1}{n}\max_k \left\{\sum_{i=1}^{k-1} T(n-i) + \sum_{i=k+1}^{n} T(i-1)\right\}$$

Ook de $T()$-waarden in het rechterlid zijn nu maximale waarden. Aanpassen van de indices geeft dan

$$T(n) \leq cn + \frac{1}{n}\max_k \left\{\sum_{i=n-k+1}^{n-1} T(i) + \sum_{i=k}^{n-1} T(i)\right\}$$

Om deze recursieve betrekking op te lossen gebruiken we de *substitutiemethode*: we 'raden' de functie voor $T(n)$ en tonen ze dan aan via inductie.

Hier onderstellen we dat $T(n)$ lineair is: $T(n) \leq an$ voor een of andere constante a. Dat geldt zeker voor n gelijk aan één. Geldt dat ook voor n groter dan één? Voor de inductiestap onderstellen we dat dit geldt voor alle waarden kleiner dan n. En dus voor alle $T()$-waarden in het rechterlid. We kunnen daarom elke $T(i)$ vervangen door ai:

$$T(n) \leq cn + \frac{a}{n} \max_k \left\{ \sum_{i=n-k+1}^{n-1} i + \sum_{i=k}^{n-1} i \right\}$$

De uitdrukking in het rechterlid is maximaal voor $k = \lfloor (n+1)/2 \rfloor$. Dit levert twee termen van de vorm

$$\sum_{i=p}^{n-1} i = \frac{(n-1)(n-2)}{2} - \frac{(p-1)(p-2)}{2}.$$

waarbij, naargelang het geval, p gelijk kan zijn aan $(n+1)/2$ (n oneven), $n/2$ of $n/2+1$ (n even). In elk van die gevallen is $(p-1)(p-2) \geq (n-2)(n-4)/4$ en met die waarde bekomen we, voor een kleine constante c_1, dat

$$T(n) \leq cn + \frac{3a}{4} n + c_1,$$

waaruit blijkt dat de onderstelling klopt, en bovendien dat we $a \sim 4c$ mogen nemen. De gemiddelde uitvoeringstijd is dus inderdaad $O(n)$.

HOOFDSTUK 7
UITWENDIG RANGSCHIKKEN

7.1 INLEIDING

Zelfs met de huidige grootte van het inwendig geheugen, zijn er toepassingen die zoveel gegevens moeten rangschikken (als zelfstandige operatie, of als onderdeel van een databankoperatie), dat ze beroep moeten doen op het uitwendig geheugen (zie bijvoorbeeld Vitter [18], en Graefe [9]). Aangezien de toegang tot dat uitwendig geheugen veel trager is dan tot het inwendig geheugen, wordt de performantie gedomineerd door de invoer/uitvoeroperaties. Het aantal invoer/uitvoeroperaties, en de manier waarop ze georganiseerd worden, is dan ook van primordiaal belang. Een configuratie met een groot aantal schijven kan echter de bandbreedte zo groot maken dat ook de vereiste CPU-tijd belangrijk wordt. De beschikbare hardwareconfiguratie speelt dus een veel grotere rol dan bij inwendig rangschikken.

De standaardmethode voor uitwendig rangschikken is *uitwendig samenvoegen* (*external mergesort*), en ze bestaat uit twee fasen: inwendig rangschikken en uitwendig samenvoegen. De gegevens worden eerst opgedeeld in stukken, die men inwendig gerangschikt, en weer uitschrijft. Daarna voegt men deze gerangschikte stukken (of reeksen[1]) uitwendig samen op de manier van mergesort.

7.2 INWENDIG RANGSCHIKKEN

Dat inwendig rangschikken kan op twee manieren gebeuren. De eerste manier leest telkens zoveel mogelijk gegevens in, rangschikt die met een efficiënte methode (meestal quicksort), en schrijft ze weer uit. Deze methode heet dan ook ‘load-sort-store’. Elke reeks (behalve eventueel de laatste) is hier even groot. En tijdens het rangschikken gebeuren er geen invoer/uitvoeroperaties.

De tweede manier gebruikt het inwendig geheugen als een soort filter, waarbij continu gegevens worden ingelezen, gegevens met kleinere sleutels worden doorgelaten en in volgorde uitgeschreven, en gegevens met grotere sleutels tijdelijk worden tegengehouden. Elk uitgeschreven gegeven is het kleinste van de gegevens in het filter

[1] Zo’n gerangschikt stuk heet in het Engels een ‘run’. Bij gebrek aan een gangbare equivalente term spreken we hier van een ‘reeks’.

('selection'), het is minstens zo groot als het vorig uitgeschreven gegeven, en het wordt vervangen door een volgend ingelezen gegeven (*'replacement'*). Deze methode heet dan ook *'replacement selection'*. De grootte van de geproduceerde reeksen kan nu variëren. Gemiddeld zijn ze tweemaal zo groot als bij de eerste methode.[2] Het rangschikken verloopt nu *gelijktijdig* met de invoer/uitvoeroperaties.

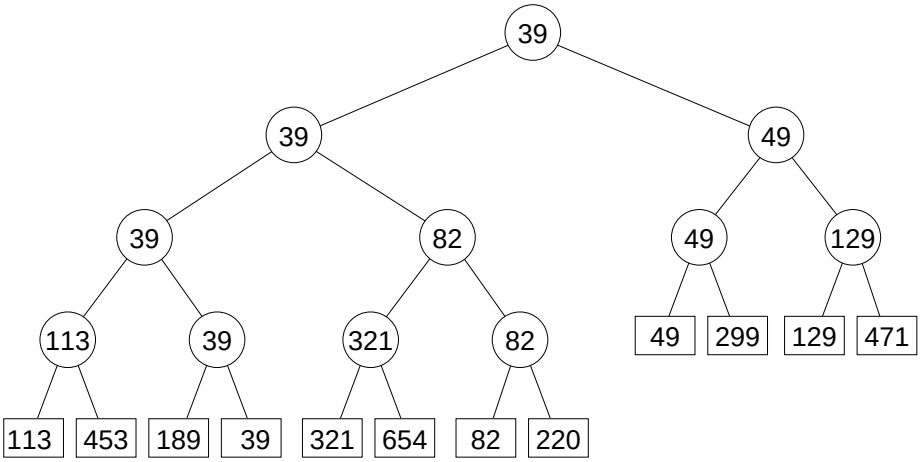
Replacement selection onthoudt steeds de sleutel van het laatst uitgeschreven gegeven. De sleutel van het nieuw ingelezen gegeven wordt daarmee vergeleken, en als hij minstens even groot is, hoort hij thuis in de huidige reeks, zoniet in de volgende. De huidige reeks wordt afgesloten wanneer alle aanwezige sleutels kleiner zijn dan de laatst uitgeschreven sleutel, en dus tot de volgende reeks behoren.

Replacement selection kan het uitschrijven van een ingelezen sleutel heel lang uitstellen, maar kan hem nooit verder vooruit plaatsen dan het maximaal aantal gegevens n in het inwendig geheugen. Wanneer elke sleutel initieel wordt voorafgegaan door minder dan n grotere sleutels, dan produceert replacement selection slechts één reeks, zodat samenvoegen overbodig wordt. Dat gebeurt dus zeker in het zeldzame geval dat de gegevens reeds gerangschikt waren. Maar ook in minder gunstige gevallen maakt replacement selection goed gebruik van reeds aanwezige ordening in de gegevens, in tegenstelling tot load-sort-store.

Omdat telkens de kleinste sleutel uit een grote collectie sleutels vereist is, ligt het voor de hand om replacement selection efficiënt te implementeren met een (binaire) *heap* (zie bijvoorbeeld Knuth [11], en Weiss [19]). Een nieuw ingelezen gegeven dat behoort tot de huidige reeks wordt aan de heap toegevoegd. Anders maakt men de heap één element kleiner, en wordt het nieuw element in de vrijgekomen plaats opgeslagen, in afwachting van de volgende reeks. Wanneer de heap ledig wordt, is zijn plaats volledig ingenomen door wachtende gegevens. Daarmee maakt men dan een nieuwe heap, en begint de volgende reeks.

De meeste implementaties gebruiken echter een *selectieboom* (zie figuur 7.1). Net zoals een binaire heap is dit een complete binaire boom, maar het is bovendien een volle binaire boom want elke inwendige knoop heeft twee kinderen. Bovendien hebben de uitwendige knopen (de bladeren) een andere functie dan de inwendige. Alle gegevens zitten in de bladeren, en elke inwendige knoop bevat de kleinste sleutel uit zijn deelboom, samen met een verwijzing naar het blad waar het overeenkomstig gegeven staat. Voor n gegevens heeft de boom dus n uitwendige en $n - 1$ inwendige knopen. (Bij selectiebomen gebruikt men meestal een terminologie ontleend aan tornooien, en als winst betekent een kleinste sleutel hebben, dan is dit een boom van *winnaars*. De tornooiwinnaar staat bij de wortel.) De gegevens van de huidige reeks en die van de volgende zitten nu in dezelfde boom. Om ze van elkaar te onderscheiden gebruikt men een samengestelde sleutel, die bestaat uit het reeksnummer gevolgd door de eigenlijke sleutel. (Voor de eenvoud onderstellen we dat alle sleutels in de figuren tot dezelfde reeks behoren.) Sleutels worden nu *lexicografisch* vergeleken: eerst het reeksnummer, dan pas de eigenlijke sleutel. (De boom kan men initialiseren met fictieve gegevens uit

[2] Dit aantonen is niet zo eenvoudig. Minder formeel is een analogie met een sneeuwruimer, zie Knuth [11].



Figuur 7.1. Een selectieboom met winnaars.

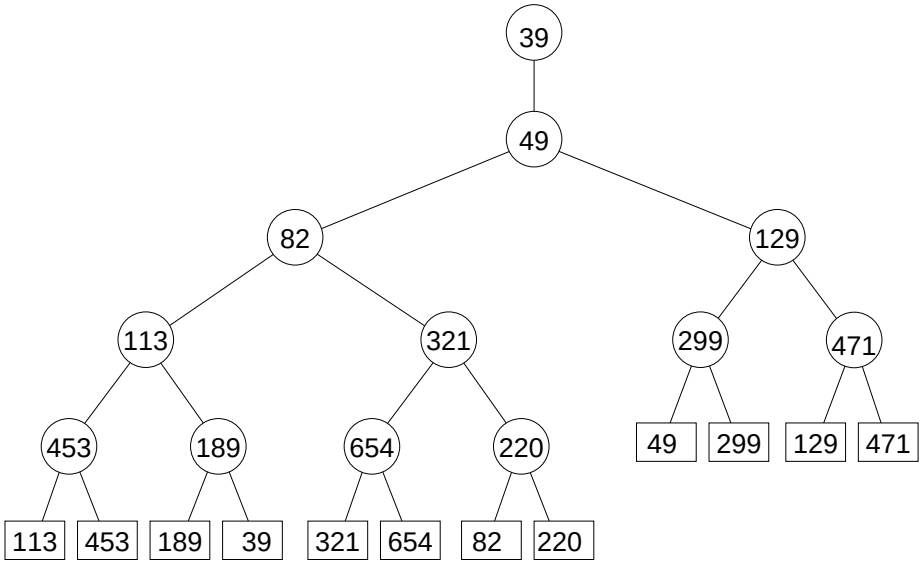
een nulde reeks, die nooit uitgeschreven wordt. Ook op het einde, als er geen nieuwe gegevens meer zijn, gebruikt men fictieve gegevens die nooit worden uitgeschreven.)

Wanneer men het gegeven met de kleinste sleutel wegschrijft, komt een nieuw gegeven in zijn plaats, en dus in zijn blad. (Vandaar dat sleutels vergezeld gaan van een verwijzing naar het blad waar ze vandaan komen.) De grootte van de boom is dus constant. Aanpassen van de boom gebeurt langs een weg naar boven, tot bij de wortel: voor een boom met n gegevens is dat dus $\Theta(\lg n)$. (Bemerk dat vervangen van de wortel bij een binaire heap een dalende weg volgt.)

De sleutels die op die weg moeten getest worden, horen bij gegevens die hun 'wedstrijd' tegen de weggeschreven sleutel verloren. Een alternatieve implementatie gebruikt dan ook een boom van *verliezers*, zoals in figuur 7.2 (zie Knuth [11]). Elke inwendige knoop bevat nu de grootste van de kleinste sleutels uit zijn deelbomen. (De verliezer van de wedstrijd tussen de winnaars van beide deelbomen.) De wortel krijgt een speciale ouder met de tornooiwinnaar. Omdat elk ander gegeven precies eenmaal een wedstrijd verloren heeft, bevat elke inwendige knoop een andere verliezer.

Wanneer de winnaar weggeschreven wordt, komt een nieuw ingelezen sleutel weer in het overeenkomstig blad terecht. Aanpassen van de boom gebeurt opnieuw langs een weg naar boven. De winnaar moet men echter apart bijhouden, omdat hij niet meer in een inwendige knoop wordt opgeslagen.

Implementaties van replacement selection trachten de selectieboom zo klein mogelijk te maken. Want een stijgende weg volgen in een grote selectieboom zorgt voor veel cachefouten, aangezien er geen verband is tussen opeenvolgende wegen. De knopen dicht bij de wortel komen op veel wegen voor, en zullen dus in de cache blijven, maar dat geldt niet voor de lager gelegen knopen. Dat kan bijvoorbeeld als volgt (zie Lar-



Figuur 7.2. Een selectieboom met verliezers.

son [12]):

- De gegevens van de knopen worden in een aparte geheugenzone opgeslagen, zodat de boom enkel verwijzingen bevat. Bij veel toepassingen (bijvoorbeeld databanksystemen) ligt de grootte van die geheugenzone vast. Gegevens met een verschillende grootte zorgen dan voor problemen. Een weggeschreven gegeven kan immers niet altijd vervangen worden door een nieuw ingelezen gegeven, omdat de vrijgekomen plaats ontoereikend blijkt. Of er komt plaats vrij voor meerdere nieuwe gegevens. Men kan dat oplossen door selecteren en vervangen te ontkoppelen: nieuwe gegevens worden aan de boom toegevoegd zolang er plaats is, dan worden er gegevens weggeschreven tot er weer voldoende plaats is voor het volgende nieuw gegeven, enz. Met als gevolg dat het aantal knopen van de selectieboom variabel wordt.

- Het aantal knopen van de boom kan beperkt worden door een minireeks gegevens aan elk blad toe te wijzen. Deze minireeksen worden vooraf intern gerangschikt met quicksort. Een weggeschreven gegeven wordt nu vervangen door het volgende gegeven uit de minireeks bij zijn blad. Het aanpassen van de boom blijft gelijk. Nieuwe gegevens worden niet een voor een toegevoegd aan de boom, maar eerst verzameld in een minireeks. Eens vol, wordt een minireeks gerangschikt, na toekenning van reeksnummers aan haar gegevens. Men koppelt ze dan aan een blad, en voegt haar kleinste sleutel toe aan de boom. Een ledige minireeks wordt niet vervangen, maar verdwijnt uit de boom. Ook hier is het aantal knopen van de selectieboom niet meer constant.

Tenslotte moeten we rekening houden met het aantal beschikbare schijven:

- Met slechts één schijf is load-sort-store aangewezen. Want diezelfde schijf bevat zowel het invoerbestand als de reeksen, wat niet geschikt is voor replacement selection dat continu reeksen produceert.
- Met twee of meer schijven kiest men echter best voor replacement selection, omdat load-sort-store slechts één schijf tegelijk gebruikt. Wanneer men over meerdere schijven beschikt, past men vaak *'disk striping'* toe. Daarbij worden opeenvolgende blokken van dezelfde reeks *gelijktijdig* op dezelfde cilinders van opeenvolgende schijven opgeslagen. (Modulo het aantal schijven.) Eigenlijk simuleert men daarmee één schijf met veel grotere blokken.

7.3 UITWENDIG SAMENVOEGEN

De reeksen moeten nu nog samengevoegd worden. Aangezien ze gerangschikt zijn, gebeurt dat op dezelfde efficiënte manier als bij mergesort. Elke reeks is echter minstens zo groot als het (beschikbaar) inwendig geheugen, zodat er slechts plaats is voor een (klein) gedeelte van elke betrokken reeks. Dubbele buffering wordt gebruikt om de CPU-aktiviteit te overlappen met schijfoperaties.

Voor eenzelfde aantal gegevens hangt de samenvoegtijd af van het aantal initiële reeksen, en van het aantal reeksen dat men gelijktijdig kan samenvoegen. Replacement selection produceert gewoonlijk minder reeksen dan bijvoorbeeld load-sort-store. En hoe meer reeksen men gelijktijdig kan samenvoegen, des te langer de resulterende reeksen, en des te minder samenvoegfasen er vereist zijn.

Voor veel toepassingen zal één samenvoegfase volstaan. Als er meerdere fasen nodig zijn, tracht men zo snel mogelijk lange reeksen te bekomen. Een goede heuristiek voegt dan ook zoveel mogelijk korte reeksen samen.

Meerdere reeksen efficiënt samenvoegen kan gebeuren met dezelfde selectieboom als bij replacement selection. Elk blad van de boom wordt nu gekoppeld aan een invoerreeks. Wanneer men een gegeven wegschrijft is het blad vanwaar het kwam bekend, zodat het kan vervangen door een nieuw gegeven uit dezelfde invoerreeks.

Gegevensuitwisseling met een schijf gebeurt in blokken. Stel dat zo'n blok b gegevens bevat, dat het beschikbaar inwendig geheugen m blokken groot is, en dat er in totaal n blokken gegevens moeten gerangschikt worden. (Voor de eenvoud gaan we ervan uit dat alle gegevens even groot zijn.) Men kan aantonen dat het optimaal aantal schijfoperaties om n blokken gegevens te rangschikken met d schijven $\Theta((n/d)\log_m n)$ bedraagt, ongeacht de gebruikte methode. Kunnen we dit optimum halen met uitwendig samenvoegen? Dat hangt af van het aantal beschikbare schijven:

- Met een selectieboom kan men $O(m)$ reeksen samenvoegen, zodat er $O(\log_m n)$ samenvoegfasen nodig zijn, die elk n blokken moeten inlezen en uitschrijven. Wanneer men slechts één schijf gebruikt, heeft uitwendig samenvoegen dan ook $\Theta(n \log_m n)$ schijfoperaties nodig. In dit geval is de methode dus optimaal.

- Werken met meerdere schijven wordt veel eenvoudiger wanneer men ze behandelt als één grotere schijf, door disk striping te gebruiken. Elke parallelle operatie leest of schrijft nu d blokken, alsof we werken met één schijf met blokgrootte db. Het aantal schijfoperaties bij uitwendig samenvoegen wordt dan

$$\Theta\left(\frac{n}{d}\log_{m/d}\frac{n}{d}\right) = \Theta\left(\frac{n}{d}\frac{\log(n/d)}{\log(m/d)}\right)$$

  Dat is echter een factor

$$\frac{\log(n/d)}{\log n}\frac{\log m}{\log(m/d)} \approx \frac{\log m}{\log(m/d)}$$

  meer dan het optimum. Voor grotere d wordt die noemer klein, zodat het verschil belangrijk kan zijn.

  Als disk striping niet toelaat om het optimum te halen, zit er niets anders op dan de d schijven onafhankelijk te gebruiken. Omdat elke samenvoegfase niet meer dan n/d schijfoperaties mag uitvoeren, moet elke parallelle leesoperatie d blokken opleveren, maar dat is enkel mogelijk als die op *verschillende* schijven staan. De samen te voegen reeksen werden echter geproduceerd in een vorige ronde, en bij het verdelen van hun blokken over de schijven is het onmogelijk om te voorspellen in welke volgorde ze later zullen ingelezen worden. Het is zelfs niet uitgesloten dat alle d blokken op dezelfde schijf staan. Er werden verscheidene methodes bedacht die met onafhankelijke schijven werken, maar door hun grotere complexiteit blijkt disk striping in de praktijk vaak toch efficiënter. Een methode die het wel beter doet is SRM ('simple randomized merge sort'), waarbij elke nieuwe stripe op een random gekozen schijf begint. (Voor meer details zie bijvoorbeeld Knuth [11], waar de methode 'randomized striping' genoemd wordt.)

DEEL 3

GEGEVENSSTRUCTUREN I

INLEIDING

Efficiënte algoritmen kunnen gebruik maken van zeer eenvoudige gegevensstructuren (we hebben al enkele voorbeelden gezien), terwijl eenvoudige algoritmen soms zeer complexe gegevensstructuren vereisen.

Hoewel gegevens en algoritme nauw verweven zijn, is het toch nuttig om beide begrippen conceptueel te scheiden, en een algoritme expliciet gebruik te laten maken van een of andere gegevensstructuur. Apart gedefinieerde gegevensstructuren kunnen ook gemakkelijker als 'bouwstenen' bij meer ingewikkelde algoritmen aangewend worden. Bovendien laat dit toe om een gegevensstructuur door een andere te vervangen, als dat de performantie van een algoritme verbetert. Deze visie sluit overigens aan bij de objectgeoriënteerde manier om software te ontwerpen.[3] Toch zullen we ons zelden bekommeren om de 'verpakking' van de gegevensstructuren, om hun 'interface' met de rest van een programma. We zijn hier voornamelijk geïnteresseerd in de inwendige organisatie van de gegevensstructuren, en de efficiëntie van de bijbehorende operaties.

De meeste gegevensstructuren kunnen eigenlijk beschouwd worden als een soort verzamelingen. Een wiskundige verzameling verandert echter nooit, terwijl de door algoritmen gebruikte verzamelingen kunnen groeien, krimpen, of op nog andere manieren wijzigen. Het zijn dus potentieel *dynamische* verzamelingen.

De gegevens die in een dergelijke verzameling worden opgeslagen bestaan meestal uit twee delen: een *sleutel* als identificatie, en *bijbehorende informatie*. Sleutels moeten niet noodzakelijk verschillend zijn, in tegenstelling tot deze in wiskundige verzamelingen.

Sommige algoritmen voeren slechts eenvoudige operaties uit op hun verzamelingen, andere vereisen meer complexe operaties. Het zal wel niemand verwonderen dat de beste manier om een gegevensstructuur te organiseren afhangt van het soort operaties dat ze moet ondersteunen. De efficiëntie van deze operaties wordt gewoonlijk bepaald in functie van het aantal opgeslagen elementen n.

Er zijn twee soorten gegevensstructuren:

- Bij sommige gegevensstructuren speelt de sleutel van de gegevens geen rol: ze worden louter als *opslagplaats* gebruikt (in het Engels noemt men ze dan ook 'containers'). Ofwel kunnen de gegevens enkel opgevraagd worden in een bepaalde volgorde, afhankelijk van de toevoegvolgorde, of van een aan de gegevens

[3] Zie cursus Objectgeoriënteerd Programmeren.

toegekende prioriteit. Ofwel worden de gegevens opgeslagen op bekende plaatsen in de structuur. Ondanks hun eenvoud worden deze structuren vaak gebruikt.

- Andere gegevensstructuren noemt men *woordenboeken* ('dictionaries'), omdat ze de 'woordenboekoperaties' zoeken, toevoegen, en verwijderen ondersteunen.[4] Zowel zoeken als verwijderen gebruiken de sleutel om het gegeven te lokaliseren.

  Vaak laten deze gegevensstructuren ook operaties toe die de *volgorde* van de sleutels gebruiken. (Dit vereist uiteraard dat er een ordening op de sleutels gedefinieerd werd.) Zo kan men naar het gegeven met de *grootste* of de *kleinste* sleutel in de verzameling zoeken, of naar het gegeven met de *voorloper* of de *opvolger* van een gegeven sleutel. Dat maakt het ook mogelijk om alle gegevens gerangschikt op te vragen.

[4] De Standard Template Library van C++ gebruikt de term 'container' ook voor de woordenboekstructuren zoals `map`.

HOOFDSTUK 8
CONTAINERS

Deze gegevensstructuren maken geen gebruik van de sleutels van de gegevens. Ofwel zijn de gegevens enkel toegankelijk in een bepaalde volgorde, ofwel worden de gegevens opgeslagen op welbepaalde plaatsen.

8.1 TABEL

Soms wenst men gegevens op te slaan zonder hun sleutels te gebruiken, maar met volledige controle over de plaats waar ze terechtkomen. Een tabel gebruiken ligt dan voor de hand, omdat de toegang niet aan restricties onderhevig is, en alle elementen even efficiënt bereikbaar zijn. Tabellen worden immers rechtstreeks ondersteund door het inwendig geheugen van de computer. (Het volledige computergeheugen kan beschouwd worden als een tabel, waarbij het (hardware)adres overeenkomt met de tabelindex.)

Een tabel heeft een vaste grootte, maar het is niet altijd eenvoudig om die op voorhand goed te schatten. Als dan bij de uitvoering blijkt dat een tabel te klein is, kan men een nieuwe grotere tabel reserveren, en alle elementen uit de oorspronkelijke tabel kopiëren naar de tweede. De eerste tabel wordt daarna vrijgegeven.

Hoewel toevoegen aan een (ongeordende) tabel normaal $O(1)$ is, heeft toevoegen aan een volle tabel dan een heel wat slechtere performantie. Wanneer eenzelfde operatie sterk uiteenlopende performanties vertoont, heeft het weinig zin om afzonderlijke operaties te bekijken. Aangezien gegevensstructuren niet gebruikt worden om er slechts één operatie op uit te voeren, kan men de performantie van een *reeks* opeenvolgende operaties bepalen, en die delen door het aantal operaties in de reeks. Het resultaat noemt men de 'geamortiseerde efficiëntie' van de operatie, want ze werd uitgemiddeld over de reeks. Op die manier kunnen we aantonen dat de geamortiseerde efficiëntie van toevoegen aan een dynamische tabel nog steeds $O(1)$ is.

Als performantie-eenheid nemen we de tijd voor een elementaire toewijzing aan een tabelplaats. Ook onderstellen we dat de tijd om een nieuwe tabel aan te vragen en de oude vrij te geven gedomineerd wordt door de tijd om te kopiëren. Gewoonlijk neemt men een nieuwe tabel die tweemaal zo groot is als de oude. (Ze uitbreiden met een vaste grootte is geen goed idee, want de geamortiseerde performantie wordt dan slechter.) Initieel is de tabel ledig, met grootte één. De tijd t_i voor de i-de toevoegoperatie is normaal gelijk aan één, maar wordt gelijk aan i als het aantal elementen dat reeds in

de tabel zat een macht van twee was. Want dan moeten we $i - 1$ elementen kopiëren, vooraleer het nieuw element kan toegevoegd worden. De totale tijd om n elementen toe te voegen wordt dan

$$\sum_{i=1}^{n} t_i = n + \sum_{j=0}^{\lceil \lg n \rceil - 1} 2^j < n + 2n = 3n$$

De sommatie in het rechterlid is immers een meetkundige reeks. De geamortiseerde tijd van een toevoegoperatie is dus drie eenheden, of $O(1)$.

8.2 LIJST

Net zoals bij een tabel is elk element van een (gelinkte) lijst toegankelijk, maar niet elk element is even efficiënt toegankelijk. De knopen van de lijst bevatten immers een wijzer die expliciet de volgende knoop aanduidt: er is geen rechtstreeks toegangsmechanisme. Verder bevat elke knoop een sleutel samen met (eventueel een wijzer naar) bijbehorende informatie. Een knoop van een *dubbel gelinkte lijst* voorziet bovendien nog een wijzer naar zijn voorloper.

Er zijn verschillende mogelijkheden om een lijst af te sluiten, maar de meestgebruikte manier is om de laatste wijzer een nullpointer te maken (een dubbel gelinkte lijst heeft er dan twee).

Een voordeel van de lijst is dat hij geen vaste grootte heeft, en steeds kan uitgebreid worden zolang er geheugen voor een nieuwe knoop overblijft. Om efficiënt uit te breiden heeft een tabel steeds meer geheugen nodig, dat bovendien *opeenvolgende* geheugenplaatsen vereist. Een nadeel van de lijst is dat zijn knopen over het geheugen verspreid kunnen liggen, wat voor veel cachefouten kan zorgen. Een ander nadeel is natuurlijk dat efficiënte toegang tot een lijst enkel aan de uiteinden kan gebeuren. Aangezien de lijst hier niet als woordenboek gebruikt wordt, zodat de sleutels van de opgeslagen gegevens geen rol spelen, heeft het ook weinig zin om interne knopen toe te voegen of te verwijderen.

8.3 STAPEL

Een stapel ('stack') ondersteunt de volgende operaties:

(1) *Toevoegen.* Traditioneel de 'push'-operatie genoemd.
(2) *Verwijderen.* Traditioneel de 'pop'-operatie genoemd. Verwijderen gebeurt in de *omgekeerde* volgorde van toevoegen: het verwijderde gegeven is steeds het laatst toegevoegde. Een stapel heet dan ook een 'LIFO'-structuur ('Last In First Out').

(3) *Testen of ledig.*
(4) *Te verwijderen gegeven opvragen.* Soms is het nodig om het eerstvolgend te verwijderen gegeven op te vragen, zonder het te verwijderen. Ook wel de 'top'-operatie genoemd.

Aangezien een stapel de toevoegvolgorde moet bijhouden, kan elke lineaire gegevensstructuur dienen om hem te implementeren:

- De eenvoudigste implementatie maakt gebruik van een tabel. De gegevens worden achter elkaar in de tabel opgeslagen, in de volgorde waarin ze aankomen. Omdat de grootte van een tabel niet wijzigt, moet het aantal gegevens op de stapel bijgehouden worden. Daarvoor gebruikt men een 'stapelwijzer' ('stack-pointer'), die meestal de index van het laatst toegevoegde element bevat (of een verwijzing naar die tabelplaats). Aanpassen van de stapelwijzer volstaat dan ook om een element te verwijderen. Toevoegen aan een volle tabel moet op een gepaste manier opgevangen worden.
- Ook met een (enkelvoudig) gelinkte lijst is de implementatie zeer eenvoudig. Toevoegen en verwijderen gebeuren immers vooraan de lijst. Hier is er geen probleem met een volle stapel.

Bij beide implementaties zijn alle operaties $O(1)$.

Na een tabel is een stapel wel de meest gebruikte gegevensstructuur. Omdat de operaties zo eenvoudig zijn wordt een stapel vaak gebruikt om gegevens bij te houden, waarvoor de volgorde van verwijderen onbelangrijk is.

8.4 WACHTRIJ

Een wachtrij ('queue') ondersteunt de volgende operaties:

(1) *Toevoegen.* Gewoonlijk de 'enqueue'-operatie genoemd.
(2) *Verwijderen.* Gewoonlijk de 'dequeue'-operatie genoemd. Verwijderen gebeurt in *dezelfde* volgorde van toevoegen: het verwijderde gegeven is steeds het langst aanwezige. Een wachtrij heet dan ook een 'FIFO'-structuur ('First In First Out').
(3) *Testen of ledig.*

Ook een wachtrij moet de toevoegvolgorde bijhouden, zodat elke lineaire gegevensstructuur kan dienen om ze te implementeren:

- De eenvoudigste implementatie maakt gebruik van een tabel, waarin de elementen opgeslagen worden in dezelfde volgorde als waarin ze werden toegevoegd. Bij de tabelimplementatie van een stapel gebeurt het verwijderen aan dezelfde

kant als het toevoegen. Hier echter moet men verwijderen aan de tegenovergestelde kant. Omdat beide operaties onafhankelijk van elkaar moeten kunnen verlopen, hebben we nu dus twee indices (wijzers) nodig: de ene wijst bijvoorbeeld het eerstvolgend te verwijderen element aan ('head'), de andere de plaats waar het volgende element moet toegevoegd worden ('tail').

Als we bij een stapel beurtelings een element toevoegen en verwijderen, blijft de stapelwijzer ongeveer ter plaatse. Bij de wijzers van de wachtrij is dat niet het geval: beide bewegen op nagenoeg constante afstand in dezelfde richting. Omdat we zo vlug de (vaste) grenzen van een tabel zouden overschrijden, implementeert men de wachtrij als een soort *ringvormige* tabel: indien een index de rand van de tabel bereikt, gaat hij verder aan de andere kant.

Oorspronkelijk is de wachtrij ledig. Hoe moeten beide wijzers geïnitialiseerd worden? Bij een wachtrij met één element staat de 'tail'-index steeds één element verder dan de 'head'-index (modulo de lengte van de wachtrij). Verwijderen we een element, dan worden beide gelijk. Hun waarde is willekeurig, we kunnen dus gerust nul nemen. Toevoegen aan een volle tabel moet weer op een gepaste manier opgevangen worden.

- Ook de implementatie met een (enkelvoudig) gelinkte lijst is eenvoudig. Toevoegen gebeurt achteraan de lijst, verwijderen vooraan. Hiervoor is één extra pointer naar de laatste wijzer in de lijst voldoende: enkel voor achteraan verwijderen is een dubbelgelinkte lijst nodig. Natuurlijk vervallen de problemen met de tabelindices en met de beperkte tabelgrootte.

Bij beide implementaties zijn alle operaties $O(1)$.

8.5 DEQUE

Een 'deque'[1] ('double-ended queue') is een combinatie van zowel een stapel als een wachtrij, die ook wel eens door een algoritme gebruikt wordt.[2]

Het is opnieuw een lineaire gegevensstructuur met beperkte toegang: alle operaties moeten aan de uiteinden van de structuur gebeuren, maar nu kan men aan beide kanten zowel toevoegen als verwijderen.

Occasioneel onderscheidt men nog een 'input-restricted deque', waarbij men slechts aan één kant mag toevoegen, en analoog, een 'output-restricted deque'.

De implementaties zijn vergelijkbaar met die van stapels en wachtrijen, en alle operaties zijn dan ook weer $O(1)$.

---

[1] Spreek uit zoals 'check'.

[2] Naast `vector` en `list` is `deque` ook een basisstructuur in de Standard Template Library van C++. Het is echter geen pure deque, omdat ook andere operaties toegelaten zijn, zoals bij een `vector` of een `list`.

8.6 PRIORITEITSWACHTRIJ

Soms wenst men de gegevens in een wachtrij niet te behandelen in de volgorde waarin ze zijn binnengekomen, maar volgens een of andere prioriteit die aan de gegevens werd toegekend. (De prioriteiten kunnen ook de gegevens zelf zijn.) De prioriteit is gewoonlijk een getal[3], en het enige gegeven dat men uit de wachtrij kan halen, is dat met de hoogste prioriteit. Deze structuur heet dan ook een prioriteitswachtrij (*priority queue*). Ze ondersteunt de volgende operaties:

(1) *Toevoegen.* Een gegeven samen met zijn prioriteit.
(2) *Verwijderen.* Het gegeven met de hoogste prioriteit.
(3) *Testen of ledig.*

Er zijn een aantal mogelijke implementaties:

- *Gerangschikte tabel.* De gegevens worden gerangschikt gehouden volgens hun prioriteit. Het element met de hoogste prioriteit verwijderen is dan $O(1)$, maar een element toevoegen is $O(n)$.
- *Ongeordende tabel.* Toevoegen is dan $O(1)$, maar het element met de hoogste prioriteit verwijderen is $O(n)$. In toepassingen waar er vaker moet toegevoegd worden dan verwijderd, is deze oplossing te verkiezen boven de eerste.
- *Binaire heap.* Een implementatie die een uitstekend compromis vormt tussen deze twee uitersten is een (binaire) heap, waarbij zowel toevoegen als verwijderen $O(\lg n)$ is. Het element met de hoogste prioriteit staat steeds bij de wortel van de (dalende) heap, zodat het verwijderen ervan neerkomt op verwijderen van de wortel, wat reeds behandeld werd bij heapsort (4.1.2). Ook een element toevoegen aan een heap werd daar besproken.

  Soms vereist men bijkomende operaties op elementen van prioriteitswachtrijen, die enkel efficiënt door heaps kunnen ondersteund worden als men de *plaats* van die elementen in de heaptabel kent. Dat is het geval bij het *verwijderen* van een gegeven element, verschillend van de wortel, of bij de *wijzigen van de prioriteit* van een gegeven element. In beide gevallen komt dat neer op een herstel van de heap te beginnen bij dat element, en dat is mogelijk in $O(\lg n)$ operaties. Een heap bevat echter nauwelijks informatie over de plaats van zijn elementen (behalve dat met hoogste prioriteit), zodat men verplicht is om eerst sequentieel te zoeken, wat gemiddeld $O(n)$ operaties vereist. Dit kan soms verholpen worden door een bijkomende gegevenstructuur, die de huidige plaats van elk element in de heap bijhoudt. Er bestaan andere soorten heaps, die specifiek ontworpen werden om ook dergelijke operaties zo efficiënt mogelijk te maken (Fibonacciheaps, bijvoorbeeld).

[3] Traditioneel hecht men de *hoogste* prioriteit aan het *kleinste* getal. Om de verwarring te beperken spreken we van kleine en grote (prioriteits)getallen, maar van hoge en lage prioriteiten.

- *d-heap.* Het principe van een binaire heap kan men uitbreiden tot een zogenaamde 'd-heap', die een complete d-wegsboom gebruikt. Elke knoop heeft nu d opvolgers, behalve eventueel op de onderste twee niveaus. De hoogte van een d-heap met n elementen is $\lfloor \log_d n \rfloor$, zodat toevoegen ongeveer een factor $\lg d$ sneller wordt dan bij een binaire heap. Afdalen wordt echter $O(d \log_d n)$, omdat nu telkens het grootste uit d elementen moet bepaald worden, wat het voordeel van een kleinere hoogte teniet doet. Grotere waarden voor d kiezen heeft dus enkel zin als toevoegen de meest frequente operatie is.

  Ook een d-heap kan door zijn eenvoudige structuur gemakkelijk in een tabel opgeslagen worden. Om snel de ouder of de kinderen van een element te vinden neemt men voor d vaak een macht van twee. Er zijn inderdaad aanwijzingen dat een 4-heap efficiënter is dan een binaire heap.

- *Andere implementaties.* Tenslotte is het soms nodig om twee prioriteitswachtrijen *samen te voegen* ('merge'). Als ze samen n elementen bevatten, dan is deze operatie met een tabelimplementatie $O(n)$. Er werden verschillende boomstructuren bedacht die ook deze operatie efficiënt ondersteunen.[4] Hun structuur is echter niet zo regelmatig als die van d-heaps ($d \geq 2$), zodat men ze niet in een tabel kan opslaan, maar moet implementeren met wijzers.

8.6.1 Een toepassing: DES

Een belangrijke toepassing van prioriteitswachtrijen vindt men bij *Discrete Event Simulation*. Hierbij gaat het over het simuleren van een systeem waarin verschillende processen grotendeels onafhankelijk van elkaar verlopen, maar op bepaalde punten toch interageren. Een voorbeeld daarvan is een productiesysteem met verschillende machines dat individuele werkstukken verwerkt. Elk werkstuk heeft een individueel traject (je kan bijvoorbeeld denken aan een werkplaats die metalen voorwerpen maakt. Elk van deze voorwerpen moet in een bepaalde volgorde bepaalde bewerkingen ondergaan zoals verbuigen, polijsten, ... Elke bewerking vereist een andere machine. Eén een bepaalde bewerking is begonnen wordt ze voortgezet. In deze is de productie van verschillende werkstukken onafhankelijk van elkaar. Is echter een bewerking afgelopen dan kan het zijn dat het werkstuk moet wachten voor het naar de volgende machine kan. Een simulatie van zo'n proces kan helpen om, bijvoorbeeld, te bepalen of het nuttig is een extra machine van een bepaalde soort aan te kopen en om een efficiënte *policy* in te voeren (soms is het efficiënt om een werkstuk te laten wachten zelfs als er een machine van de juiste soort vrij is).

DES werkt hier met discrete *events*, gebeurtenissen die op een bepaald ogenblik plaatsvinden. Een gebeurtenis die plaatsvindt kan andere gebeurtenissen creëren die op hetzelfde moment of later zullen plaatsvinden. Zo kan bevoorbeeld het beëindigen van een bewerking op een werkstuk de gebeurtenis creëren dat op dat ogenblik een ander werkstuk uit de wachtrij voor de machine gehaald wordt en dat de bewerking daarvan

[4] Zie Algoritmen II.

begint. Het begin van een bewerking creëert de gebeurtenis dat de bewerking voltooid wordt, maar wel op een later tijdstip bepaald door de eigenschappen van de bewerking.

Gebeurtenissen worden een voor een afgehandeld. Nu kan een gebeurtenis pas worden afgehandeld als de toestand van het systeem op het moment van de gebeurtenis bekend is en bijgevolg pas op het moment dat alle gebeurtenissen die vroeger in de tijd liggen zijn afgehandeld. Hiervoor wordt een prioriteitswachtrij gebuikt. Deze bevat alle nog niet afgehandelde gebeurtenissen. Bij het opstarten van de simulatie wordt de wachtrij opgevuld met gebeurtenissen die onafhankelijk zijn van andere. In ons voorbeeld gaat het om het binnenkomen van een werkstuk in het productiesysteem. De simulator haalt daarna telkens de gebeurtenis met de hoogste prioriteit (dit is: met het vroegste tijdstip) van de wachtrij en verwerkt deze. Indien daarbij nieuwe gebeurtenissen gecreëerd worden zet hij deze in de wachtrij.

Soms kunnen er hierbij problemen optreden omdat gebeurtenissen op hetzelfde ogenblik plaatsvinden waarbij de verwerkingsvolgorde belangrijk is. Dit kan vaak opgelost worden door gebeurtenissen met dezelfde tijd toch een enigszins verschillende prioriteit te geven.

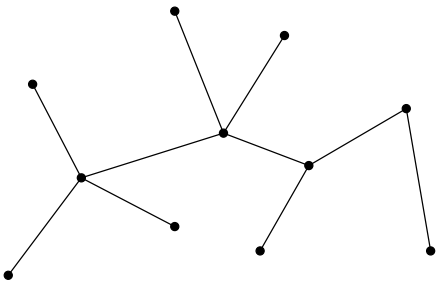
8.7 BOOM

8.7.1 Definities

Een boom is een van de belangrijkste gegevensstructuren, en veel algoritmen maken er expliciet of impliciet gebruik van. We hebben reeds kennis gemaakt met enkele bomen: heaps, partitiebomen en beslissingsbomen. In dit hoofdstuk geven we enkel de algemene definitie van een boom, gevolgd door een bespreking van de meest gebruikelijke manieren om hem in een computer voor te stellen, en enkele algemene operaties op bomen, die geen gebruik maken van de sleutels van de opgeslagen gegevens. Later komen dan woordenboekstructuren aan bod die gebruik maken van bomen.

Zeer algemeen definieert men een boom als een verzameling *knopen* (*nodes* of *vertices*) die onderling door *takken* (*edges*) verbonden zijn (zie figuur 8.1). Deze takken hebben geen specifieke richting. Belangrijk is wel dat de takken geen lussen mogen vormen. Elke knoop is bereikbaar vanuit elke andere knoop, en de gevolgde weg is zelfs uniek. De takken bepalen de *structuur* van de boom. De eigenlijke gegevens worden in de knopen opgeslagen. (En soms ook bij de takken.)

Bij de meeste bomen onderscheidt men één speciale knoop, de *wortel* (*root*) van de boom. Op wegen die vanuit de wortel vertrekken, worden de knopen nu beurtelings voorlopers (dichter bij de wortel) en opvolgers (verder van de wortel). Een rechtstreekse opvolger van een knoop noemt men zijn *kind*, en de rechtstreekse voorloper



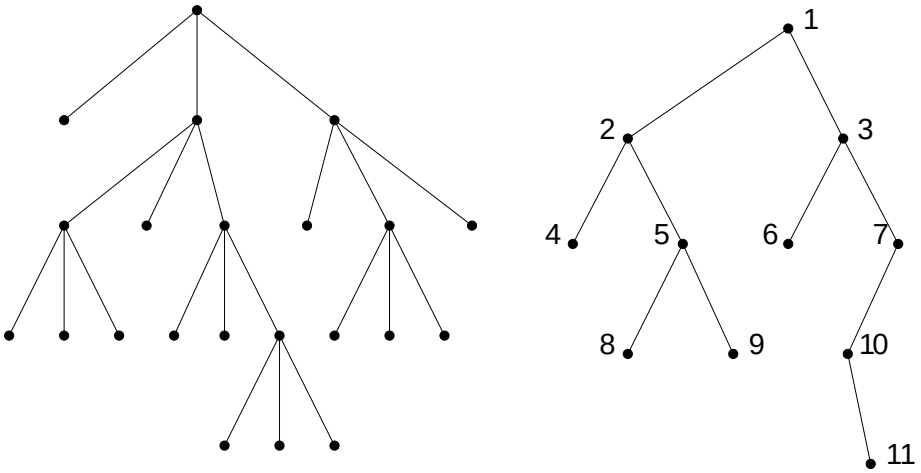
Figuur 8.1. Wortelloze Boom.

van een knoop is dan zijn *ouder* (*parent*). De wortel is de enige knoop zonder ouder. De kinderen van een knoop zijn elkaars *broers* (*siblings*). Knopen zonder kinderen noemt men de *bladeren* (*leaves*) van de boom (de terminologie is zeker niet consistent.) In plaats van bladeren spreekt men ook wel eens van de *uitwendige* of *terminale* knopen van de boom. De andere knopen zijn dan *inwendig* of *niet-terminaal*.

Elke knoop kan als wortel van een *deelboom* beschouwd worden. Die bevat dan alle opvolgers van deze knoop (gezien vanuit de wortel van de volledige boom). *Bomen zijn dus opgebouwd uit bomen*. Deze recursieve eigenschap zullen we later vaak kunnen gebruiken.
Sommige algoritmen werken met een collectie bomen. Dat noemt men uiteraard een *bos* ('forest'). Meestal kunnen de wortels van al deze bomen beschouwd worden als de kinderen van een fictieve knoop, zodat een bos dan herleid wordt tot een boom.

De lengte van de weg (het aantal takken) tussen de wortel en een knoop noemt men de *diepte* van de knoop. De wortel heeft dus diepte nul. De *hoogte* van een knoop is de lengte van de langste weg tussen die knoop en een blad in zijn deelboom. De hoogte van de boom is de hoogte van zijn wortel.

Het aantal kinderen van een knoop is de *graad* van de knoop. Bij een *geordende boom* is de volgorde van de kinderen van elke knoop belangrijk. Twee geordende bomen met dezelfde vorm kunnen dus toch verschillend zijn. Bij een *meerwegsboom* ('multiway tree') gaat men nog verder: alle inwendige knopen hebben potentieel k geordende kinderen, waarvan sommige mogen ontbreken. Elk kind krijgt een nummer, ook als het ontbreekt. Een knoop in een vijfwegsboom kan dus bijvoorbeeld drie kinderen hebben, het tweede, derde en vijfde. De ontbrekende kinderen vervangt men soms door knopen zonder kinderen. Alle inwendige knopen hebben in dat geval evenveel kinderen: een dergelijke boom noemt men *vol* (*full tree*). Links in figuur 8.2 staat een voorbeeld van een volle driewegsboom. Een zeer belangrijk speciaal geval van een meerwegsboom is de reeds bekende *binaire boom*, met k gelijk aan twee (rechts in figuur 8.2). Elke inwendige knoop maakt hier dus duidelijk onderscheid tussen zijn linker- en zijn rechterkind.



Figuur 8.2. Volle driewegsboom en binaire boom.

8.7.2 Voorstelling

Een binaire heap kan door zijn speciale structuur als complete binaire boom efficiënt in een tabel opgeslagen worden. De potentieel onregelmatige vorm van een algemene boom vereist dat elke knoop expliciet zijn kind(eren) aanduidt. Als het aantal knopen van een boom vastligt, kan men ze eveneens in een tabel opslaan, en de kinderen via indices aanduiden. Meestal zijn bomen echter *dynamische* structuren, met een variërend aantal knopen. Naar kinderen wordt dan gewoonlijk gerefereerd via wijzers.

Knopen van meerwegsbomen moeten expliciet ontbrekende kinderen aanduiden. Daarvoor zijn er verschillende mogelijkheden:

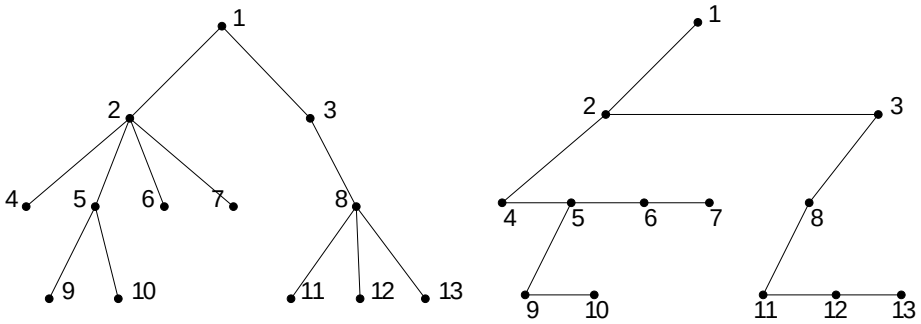
- Gebruik de nullwijzer.
- Aangezien het aantal wijzers naar ontbrekende kinderen groot is, tracht men ze soms nuttiger te gebruiken, door ze naar andere knopen in de boom te laten wijzen (afhankelijk van de operaties). Daartoe moet men ze kunnen onderscheiden van wijzers naar kinderen, bijvoorbeeld met behulp van een extra logische waarde.

De takken van een boom hebben geen richting, wijzers echter wel. Daarom bevat een knoop soms ook nog een wijzer naar zijn ouder, om bepaalde operaties efficiënter te maken. Sommige algoritmen gebruiken zelfs uitsluitend ouderwijzers, omdat ze enkel stijgen in bomen, nooit afdalen.

Een knoop van een binaire boom bevat dus een wijzer naar zowel het linker- als naar het rechterkind. Bij knopen van meerwegsbomen voorziet men een tabel met k wijzers.

Voor grote k neemt dit wel veel geheugen in beslag, zeker als het gemiddeld aantal aanwezige kinderen per knoop klein is.

Een variërend aantal gegevens wordt het best opgeslagen in een gelinkte lijst. Een geordende boom kan dus voorgesteld worden door in elke knoop een (enkelvoudig) gelinkte lijst van kinderen te voorzien. Omdat elke knoop zelf tot een dergelijke lijst behoort (die van zijn ouder), bevat hij ook een wijzer naar zijn rechterbroer, de opvolger in die lijst. We komen dus tot het verrassend resultaat dat elke knoop slechts twee wijzers bevat, één naar zijn linkerkind en één naar zijn rechterbroer. Met andere woorden, dit is een *binaire boom*, ook al is de betekenis van de twee wijzers in een knoop verschillend. Binaire bomen zijn dus zeer belangrijk, niet alleen omdat veel gebruikte



Figuur 8.3. Een geordende boom en zijn binair equivalent.

bomen inherent binair zijn, maar ook omdat een willekeurige geordende boom als een binaire kan voorgesteld worden.

8.7.3 Systematisch overlopen van bomen

Sommige algoritmen moeten alle knopen van een boom overlopen (en meteen ook alle takken), in een welbepaalde volgorde, onafhankelijk van de sleutels van de opgeslagen gegevens ('tree walk'). Alle gegevens in een tabel of een lijst overlopen is zeer eenvoudig, omdat het lineaire gegevensstructuren zijn. Bij bomen is dat niet zo evident, en zullen we een systeem moeten bedenken om alle knopen precies eenmaal te overlopen. We kunnen op twee manieren tewerk gaan: gebruik maken van de recursieve structuur van de boom, of niveau per niveau afhandelen:

(1) **Diepte eerst,** verder afgekort tot DEZ voor diepte-eerst zoeken. De recursieve structuur van een boom maakt een recursieve beschrijving van het te volgen traject zeer eenvoudig. Een niet-ledige boom (en dus ook elke deelboom) bestaat uit een wortel met daaronder deelbomen, één voor elk kind (voor een meerwegsboom ook een lege deelboom voor elk niet-bestaand kind). Diepte-eerst overlopen gebeurt dus systematisch door de wortel te bekijken en al deze deelbomen. We kunnen eerst de wortel bekijken en dan de deelbomen (zgn. *preorder*) of eerst de

deelbomen en dan de wortel (zgn. *postorder*). Bij meerwegsbomen, en meer in het bijzonder bij binaire bomen, is de volgorde van de kinderen bepaald en kunnen we de wortel ook zinvol *tussen* de kinderen plaatsen. Voor een binaire boom levert dat drie mogelijkheden:

a. Behandel eerst de wortel, en overloop daarna de linkse deelboom, gevolgd door de rechtse deelboom. Dat heet overlopen in 'preorder'. Een afwezig kind betekent natuurlijk een ledige boom, die geen werk vereist.
b. Overloop eerst de linkse deelboom, behandel de wortel, en overloop tenslotte de rechtse deelboom. Men spreekt dan van overlopen in 'inorder' (of *symmetric order*).
c. Overloop eerst de linkse deelboom, dan de rechtse, en behandel tenslotte de wortel. Dat heet overlopen in 'postorder' (of *endorder*).

De binaire boom uit figuur 8.2geeft $\{1, 2, 4, 5, 8, 9, 3, 6, 7, 10, 11\}$ als opeenvolgende knoopnummers bij het in preorder overlopen. Bij inorder wordt deze volgorde $\{4, 2, 8, 5, 9, 1, 6, 3, 10, 11, 7\}$, bij postorder $\{4, 8, 9, 5, 2, 6, 11, 10, 7, 3, 1\}$. Binaire bomen worden bijvoorbeeld door compilers gebruikt om wiskundige uitdrukkingen voor te stellen. De bewerking is dan de informatie die bij een (inwendige) knoop hoort, en de argumenten zijn de kinderen (unaire operatoren hebben slechts één kind). Die kunnen op hun beurt uitdrukkingen zijn, en dus op dezelfde manier voorgesteld worden. Voor de uitdrukking $(a+b)*c-(d+e+f)/g$ bijvoorbeeld ziet die boom er uit zoals in figuur 8.4. Bemerk dat de haakjes verdwenen zijn, en dat er rekening gehouden werd met de prioriteit en de associativiteit van de operatoren.[5] Het overlopen van een dergelijke boom in preorder, inorder en postorder levert een lineaire notatie van de uitdrukking op, in resp. prefix-, infix- of suffixvorm. Zo kan de postfixvorm $a\ b + c * d\ e + f + g/-$ bijvoorbeeld gebruikt worden bij het evalueren van de uitdrukking door een stapelmachine.
Het programmeren van deze drie methodes in een taal zoals C++ die recursie toelaat is zeer eenvoudig. Ouderwijzers zijn daarbij overbodig: de recursiestapel houdt de terugweg bij. Recursieve oproepen voor ledige bomen kan men vermijden door ze vóór de oproep te testen.

(2) **Breedte eerst,** verder afgekort tot BEZ, voor breedte-eerst zoeken. De tweede manier overloopt de verschillende niveau's in de boom, van boven naar beneden, en elk niveau van links naar rechts. Dit heet dan ook overlopen in 'level order'. Deze methode kan geen gebruik maken van de recursieve structuur van de boom, en is dus op het eerste gezicht niet zo eenvoudig als de vorige methodes. De kinderen van elke behandelde knoop mogen slechts behandeld worden na alle resterende knopen op hetzelfde niveau, en vóór de kinderen van die knopen. Als we telkens de kinderen van een behandelde knoop in een *wachtrij* opslaan, en de volgende te behandelen knoop uit die wachtrij halen, respecteren we die volgorde.

Er bestaat een merkwaardige gelijkenis tussen recursief en in level order overlopen van een boom. Immers, om de recursie te implementeren gebruikt de compiler een *stapel*, waarop informatie over de nog te behandelen knopen in de juiste volgorde wordt

[5] Zie keuzevak 'Compilers'.

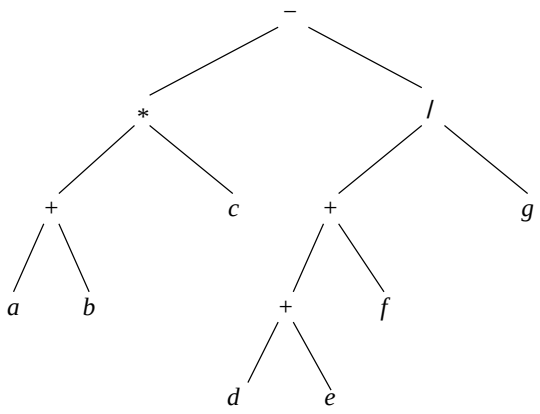
```
DEZbezoek(function<void(const T&)>preorder,
          function<void(const T&)>inorder,
          function<void(const T&)>postorder){
stack<pair<Binknoop<T>*, int>>ATW;//ATW: Af Te Werken
if ((*this)!=0){
    ATW.emplace((*this).get(),1);
    while (!ATW.empty()){
        Binknoop<T>* nuknoop=ATW.top().first;
        int pass=ATW.top().second;
        switch (pass){
            case 1:
                preorder(nuknoop->sl);
                ATW.top().second++;
                if (nuknoop->links)
                    ATW.emplace(nuknoop->links.get(),1);
                break;
            case 2:
                inorder(nuknoop->sl);
                ATW.top().second++;
                if (nuknoop->rechts)
                    ATW.emplace(nuknoop->rechts.get(),1);
                break;
            case 3:
                postorder(nuknoop->sl);
                ATW.pop();
        }
    };
};
};
```

Pseudocode 8.1. Expliciete stapel bij diepte-eerst.

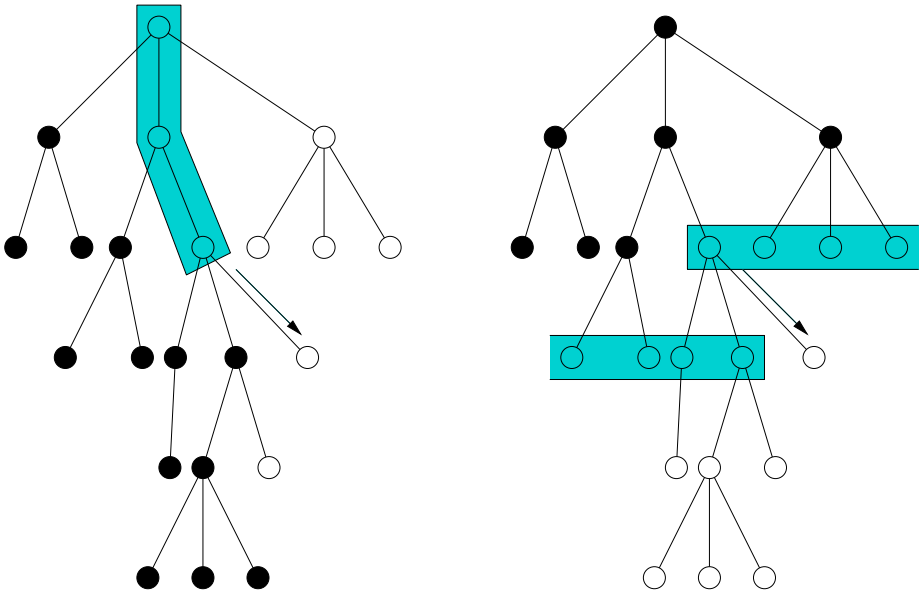
bijgehouden. (Er bestaan technieken om zelf die recursie te verwijderen, door expliciet gebruik te maken van een stapel.) In level order overlopen gebeurt op dezelfde manier, alleen moet men de stapel vervangen door een wachtrij.

Het overlopen van binaire bomen kan eenvoudig uitgebreid worden voor geordende bomen en meerwegsbomen, en zelfs voor bossen. Overlopen in preorder behandelt dan eveneens een knoop vóór al zijn kinderen, en in postorder na al zijn kinderen. Enkel overlopen in inorder heeft hier geen equivalent. De definitie van level order blijft natuurlijk ongewijzigd.

Wat gebeurt er als we de geordende bomen vervangen door hun binaire equivalenten, en deze op de verschillende manieren overlopen? De betekenis van de twee kin-



Figuur 8.4. Syntactische boom voor $(a + b) * c - (d + e + f)/g$.



Figuur 8.5. Diepte-eerst en breedte-eerst zoeken

deren in de binaire boom is immers verschillend, maar daar houden deze operaties geen rekening mee. Het is niet zo moeilijk om na te gaan dat preorder hetzelfde resultaat geeft op beide voorstellingen, postorder bij de geordende boom overeenkomt met inorder bij de binaire boom, en level order uiteraard een verschillend resultaat geeft. Voor de geordende boom uit figuur 8.3 bijvoorbeeld is de preorder volgorde $1, 2, 4, 5, 9, 10, 6, 7, 3, 8, 11, 12, 13$ dezelfde als die van zijn equivalente binaire boom, en zijn postorder volgorde $4, 9, 10, 5, 6, 7, 2, 11, 12, 13, 8, 3, 1$ is precies de in-

order volgorde van zijn equivalent.

8.7.4 Backtracking

8.7.4.1 Strings als gegevensstructuur

Een string is een aaneenschakeling van letters uit een bepaalde eindige karakterverzameling Σ. Het is gebruikelijk om strings te beschouwen waarbij de karakterverzameling een traditioneel alfabet is of tenminste omvat: zo bevat de traditionele byteset met ASCII-codering alle kleine letters en hoofdletters van het 'gewone' alfabet. Vaak is het echter interessant om strings te beschouwen met een niet-traditionele karakterverzameling. Zo zou een routeplanner een vaste verzameling van instructies kunnen hebben, $\{$*'ga rechtdoor', 'sla linksaf',* $\ldots\}$, die we als alfabet nemen. Daarbij is een routebeschrijving dan een string, waarbij we elke instructie als één enkele letter beschouwen.

Een tweede voorbeeld vinden we bij het winkelprobleem. Ik heb een boodschappenlijst van dingen die ik wil kopen. Ik heb ook een verzameling winkels die elk een of meerdere items op mijn lijst verkopen, waarbij veel items op mijn lijst in verschillende winkels te koop zijn. Ik wil nu mijn boodschappen doen door zo weinig mogelijk winkels te bezoeken. Variaties gebruiken een ander optimisatiecriterium: ik wil zo weinig mogelijk afstand afleggen, of ik wil zo weinig mogelijk geld uitgeven, waarbij ik voor elk bezoek aan een winkel een vaste kost bereken, en zo voorts. Bij het zoeken naar een oplossing is het handig om de verzameling winkels te beschouwen als het alfabet Σ. Strings over dit alfabet geven dan een lijst van winkels aan, waarbij de volgorde belangrijk kan zijn.

De gebruikelijke notatie voor de verzameling van alle strings over een alfabet Σ is $\Sigma*$. De $*$ in deze notatie verwijst naar de Kleene-sluitingsoperatie zoals die bij reguliere uitdrukkingen gebruikt wordt: Het is dus $\Sigma*$ en niet Σ^*.

Definities:

- Een *combinatorisch probleem* is een probleem van de volgende vorm: gegeven is een alfabet Σ en een geheel van voorwaarden. Gevraagd is een string (of alle strings) uit $\Sigma*$ die aan alle voorwaarden voldoet (of voldoen).
- Een *combinatorisch optimisatieprobleem* is een probleem van de volgende vorm: gegeven is een alfabet Σ, een geheel van voorwaarden en een evaluatiefunctie die aan elke string die aan de voorwaarden voldoet een waarde toekent. Gevraagd is een element (of alle elementen) uit $\Sigma*$ die aan alle voorwaarden voldoet (of voldoen) en volgens de evaluatiefunctie de beste score haalt.

We gaan ervan uit dat de voorwaarden gemakkelijk te controleren zijn[6]. Een string die aan alle voorwaarden voldoet noemen we een oplossing, ook bij een optimisatiepro-

[6] Voor wie dit strikt wil definiëren: ze moeten polynomiaal controleerbaar zijn. Hiervoor verwijzen we naar de literatuur over NP-problemen.

bleem. Daar zijn we dus niet tevreden met zomaar een oplossing: we willen de beste oplossing.

Het is duidelijk dat bijna alle problemen die we in deze cursus behandelen in een van de twee vormen kunnen gegoten worden. Zo is elk *toekenningsprobleem* van deze vorm. Bij zulk een probleem hebben we een aantal variabelen $v_0, v_1, \ldots, v_{n-1}$. Vraag is om aan elke variabele v_i een waarde toe te kennen uit een domein D_i zodanig dat het resultaat aan bepaalde voorwaarden voldoet. Nemen we als alfabet $\Sigma = \cup_{i=0}^{n-1} D_i$ dan is een oplossing van het probleem een string uit $\Sigma*$ van lengte n waarbij de i-de letter in D_i moet zitten en die aan ook aan alle andere voorwaarden voldoet.

Ook al bestaan er talrijke efficiënte algoritmen voor problemen in de meest diverse toepassingsgebieden, toch zijn er veel praktische problemen waarvoor nog geen efficiënt algoritme gevonden werd. (En hoogstwaarschijnlijk ook nooit zal gevonden worden. Dat is het beroemde P-NP-probleem.[7])

In principe zit er dan niets anders op dan alle kandidaat-oplossingen uit te proberen. Omdat die problemen combinatorisch zijn, is deze brutekrachtmethode enkel zinvol wanneer de 'afmetingen' van het probleem klein zijn. Als we een toekenningsprobleem hebben waarbij alle domeinen grootte d hebben dan zijn er in principe d^n strings die we moeten onderzoeken, wat zelfs voor vrij kleine waarden van d en n al ondoenbaar veel is. Zelfs programmatorisch is er een probleem: je kan in een programma wel n verneste for-lussen schrijven, maar alleen als n op voorhand gekend is. Om alle kandidaat-oplossingen uit te proberen moet men systematisch te werk gaan, zodat elke mogelijke oplossing precies eenmaal aan bod komt. Voor veel problemen kan de oplossing *incrementeel* geconstrueerd worden: men voegt achtereenvolgens letters toe, tot een volledige oplossing bekomen wordt. Hierbij is het belangrijk om zo snel mogelijk te ontdekken als men op een dood spoor zit, waarbij de al geconstrueerde string niet het begin kan zijn van een uiteindelijke oplossing. Basisidee is hier dat men, als men al een deelstring heeft, men een lijst opstelt van mogelijkheden voor de volgende letter die zo restrictief mogelijk is. Bij ons winkelprobleem gaan we bijvoorbeeld alleen nog winkels toevoegen die nog niet in de deelstring zitten. Beter echter is om alleen winkels toe te voegen die items verkopen die wel op onze boodschappenlijst staan maar niet verkocht worden door winkels die al in de deelstring zitten. Een goede keuze kan het verschil maken tussen een efficiënt programma en een dat onmogelijk veel tijd nodig heeft.

Backtracking is een systematische oplossingsmethode voor dit soort problemen. Impliciet wordt hierbij gebruik gemaakt van een boom die diepte-eerst wordt afgezocht (in principe zou men deze boom ook in level-order kunnen overlopen, maar de geheugenvereisten zijn veel hoger). Een knoop in die boom op diepte k komt overeen met een string in $\Sigma*$ van lengte k: de wortel komt dus overeen met de lege string. De knoop waar we ons in bevinden wordt voorgesteld door een kandidaat-deeloplossing. De boom wordt niet expliciet opgeslagen. Wel heeft men voor alle grijze knopen een lijst van kinderen in de vorm van een lijst van toe te voegen letters. In principe moet

[7] Zie Gevorderde Algoritmen.

men zelfs slechts een lijst met nog niet bezochte kinderen bijhouden. Deze lijsten worden bijgehouden op een stapel. Zoals bij gewoon diepte-eerst zoeken kan deze stapel expliciet geprogrammeerd worden, maar ook kan men gebruik maken van recursie.

Afdalen in de boom komt overeen met het toevoegen van een letter aan de deeloplossing, terug naar boven gaan met het wegnemen van een letter. Als bij het teruggaan blijkt dat er geen mogelijke letter meer is om toe te voegen moet men nog een stap verder teruggaan. Als alles goed gaat krijgen we een string die een oplossing voorstelt. Als we maar één oplossing zoeken kunnen we stoppen. Als we echter alle oplossingen willen, of we zoeken een optimale oplossing, moeten we verder gaan. Dan eindigen we als de deeloplossing terug de lege string geworden is[8]. Dit is ook het einde als er geen oplossing voor het probleem bestaat. De methode kan dus op haar stappen terugkeren, vandaar de naam '(chronological) backtracking'. Ze genereert alle kandidaat-oplossingen precies eenmaal, zodat de gezochte oplossing zeker gevonden zal worden, als ze bestaat.

Een (deel)oplossing is dus een string van waarden $\langle w_0, w_1, \ldots, w_k \rangle$. Elke waarde w_i behoort tot de verzameling V_i van kandidaat-waarden voor de i-de letter. Gesteld dat deze (deel)oplossing mag verlengd worden met een $k+1$-de letter, dan moeten we eerst de verzameling V_{k+1} van alle kandidaat-waarden voor deze letter bepalen, om daaruit de volgende waarde w_{k+1} te kiezen. Een oplossing is een string die aan alle voorwaarden voldoet. In veel gevallen weten we dat we een oplossing hebben omdat de deeloplossing een bepaalde lengte n heeft bereikt. Bij een toekenningsprobleem bijvoorbeeld hebben alle oplossingen dezelfde lengte n, maar toch kan het nodig zijn om te controleren of een kandidaat-deeloplossing van lengte n aan alle voorwaarden voldoet. Maar het is niet altijd op voorhand gekend hoe groot een oplossing is: het is zelfs mogelijk dat er twee oplossingen zijn waarbij de ene een prefix is van de andere.

Het verlengen wordt herhaald totdat een verzameling V_{k+1} ledig blijkt (of tot we een oplossing hebben, als we maar één oplossing zoeken). Dan moeten we op onze stappen terugkeren om nieuwe deeloplossingen uit te proberen. Terugkeren betekent een andere waarde kiezen voor variabele k, als tenminste nog niet alle waarden van V_k aan bod kwamen. Daarna proberen we opnieuw de deeloplossing te verlengen met variabele $k+1$, enz. (Bemerk dat een nieuwe keuze voor w_k een andere verzameling V_{k+1} kan opleveren dan daarvoor.)

Als echter V_k uitgeput is, dan moeten we terugkeren naar variabele $k-1$ om een nieuwe waarde w_{k-1} te kiezen, enz. Het backtrackmechanisme stopt als er geen nieuwe keuze meer kan gemaakt worden voor de eerste variabele. De pseudocode 8.2 illustreert de werkwijze voor een versie die niet stopt als ze een oplossing vindt.

Een tweede, recursieve, versie van backtracking staat in pseudocode 8.3.

Zelfs voor niet al te grote n kan de zoekboom zeer groot worden. Daarom tracht men de performantie van backtracking te verbeteren door het aantal knopen zoveel mogelijk te

[8] We gaan er hierbij van uit dat er maar een eindig aantal deeloplossingen is. Strikt genomen volgt dat niet uit de gegeven definities.

```
void backtracking () {
   int k = 0;
   bepaal V0;
   while (k >= 0) {
      while (Vk niet ledig) {
         wk = volgend element uit Vk;
         verwijder wk uit Vk;
         if (<w0,w1,...,wk> een oplossing is)
            verwerk <w0,w1,...,wk>;
         k++; // volgende positie
         bepaal Vk;
      }
      k--; // backtracking
   }
}
```

Pseudocode 8.2. Backtracking.

beperken, en zo weinig mogelijk knopen te onderzoeken. Telkens wanneer we op onze stappen terugkeren wordt de onderliggende deelboom niet verder onderzocht, zodat we de zoekboom als het ware *snoeien* (*pruning*). Snoeien kan leiden tot spectaculaire performantieverbeteringen. Mogelijke verbetertechnieken zijn:

- *Variabelen ordenen.* Bij een toekenningsprobleem kan de volgorde van de variabelen bij het toekennen van waarden grote invloed hebben op de grootte van de boom. Daarom behandelt men de variabelen best volgens *stijgend* aantal mogelijke waarden. De grootte van de verzamelingen V_i kan ofwel op voorhand bepaald worden (statisch), ofwel telkens als er een nieuwe waarde aan een variabele wordt toegekend (dynamisch), en dan enkel voor de nog resterende variabelen.

- *Waarden ordenen.* De volgorde waarin de waarden in V_k worden toegevoegd aan de deeloplossing heeft geen invloed op de grootte van de boom, maar wel op de volgorde waarin die doorzocht wordt. Als men alle mogelijke oplossingen wenst, speelt dat geen rol, maar wel als tevreden is met één oplossing, die dan sneller kan gevonden worden. Waarden worden best toegekend volgens *dalend* aantal mogelijkheden dat ze open laten: de 'gemakkelijkste' waarden eerst.

- *Verder terugkeren.* ('Backjumping'.) In plaats van terug te keren naar de vorige variabele, zoals bij (chronological) backtracking, keert men terug naar de variabele die de deeloplossing ongeldig maakte. Stel bijvoorbeeld dat de variabelen x en y reeds een waarde kregen, maar dat elke waarde voor de volgende variabele z onverenigbaar is met de waarde van x. Dan is het zinloos om terug te keren naar y, en opnieuw alle waarden voor z uit te proberen. Men keert beter terug naar x.

```
void backtracking (String& deeloplossing) {
   //k=lengte deeloplossing
   if (deeloplossing is oplossing)
      verwerk(deeloplossing);
   bepaal Vk;
   // Alle kandidaten voor positie k uitproberen
   for (alle w uit Vk) {
      backtracking(deeloplossing+w);

   }
}

void backtracking () {
   backtracking (legestring);
}
```

Pseudocode 8.3. Backtracking, recursieve versie.

- *Vooruit testen.* ('Forward checking'.) Bij het toekennen van een nieuwe waarde aan een variabele, wordt nagegaan of er nog minstens één mogelijke waarde overblijft voor alle resterende variabelen. Zoniet geeft men meteen een volgende waarde aan de variabele.

- *Symmetrieën.* Een laatste mogelijkheid om te snoeien bestaat erin om zoveel mogelijk gebruik te maken van symmetrieën in het probleem. Deelbomen die dezelfde oplossingen opleveren die we reeds gevonden hebben, of die er eenvoudig mee verwant zijn, kunnen we buiten beschouwing laten. Het is echter niet altijd eenvoudig om symmetrieën te ontdekken. Een voorbeeld waar dit kan is het oorspronkelijke winkelprobleem, waarbij alleen het aantal winkels telt. Vermits de volgorde van de winkels in de oplossing onbelangrijk is kan men elke winkel van een volgnummer voorzien. Bij het bepalen van V_k kan men eisen dat elk element van V_k een groter volgnummer heeft dan w_{k-1}.

- *Snoeien bij optimisatie*. Als men een voorlopig beste oplossing heeft kan het zijn dat een deeloplossing geen betere oplossing kan opleveren dan deze die men reeds gevonden heeft en moet men niet verder gaan in die richting. Een voorbeeld is het oorspronkelijke winkelpobleem: als een deeloplossing meer winkels heeft dan een al gevonden oplossing kan men ze weglaten. Deze vorm van snoeien staat bekend als 'branch-and-bound'. Daarbij past men soms tientallen 'heuristieken' (vuistregels) toe om de beste oplossing van een deelboom te 'voorspellen'. Ook hier kan men trachten de waarden te ordenen die aan een variabele moeten toegekend worden, zodat de beste oplossingen zo vroeg mogelijk gevonden worden. Zo kan men substantiële stukken van de boom snoeien. Daarbij maakt men opnieuw gebruik van allerlei 'heuristieken', afhankelijk van

het probleem.

Ten slotte: een probleem kan te groot zijn om met backtracking op te lossen, ook met de hiervermelde verbeteringen. Bij optimisatieproblemen komt het voor dat men wel redelijk snel vrij goede oplossingen kan vinden door het gebruik van heuristieken en/of metaheuristieken. Gezien het praktische belang van sommige van deze problemen is en wordt daarover veel onderzoek verricht.

8.8 GRAAF

8.8.1 Definitie en voorstelling

Een graaf is een abstract model, dat bestaat uit een verzameling van n knopen (*vertices, nodes*) en een verzameling van m verbindingen (*edges*). De knopen stellen objecten voor waartussen verbindingen kunnen bestaan, en we zullen ze hier meestal nummeren van 0 tot en met $n-1$.[9] De verbindingen worden aangeduid met de nummers van hun eindknopen, zoals bijvoorbeeld de verbinding (i, j) of $v_{i,j}$.

Meestal nemen we aan dat er maar één verbinding (i, j) bestaat van een knoop i naar een knoop j. Soms breidt men de definitie van een graaf uit om meervoudige verbindingen tussen twee knopen toe te laten en zelfs verbindingen van een knoop naar zichzelf. In dit geval spreken we van een *multigraaf*.

Verbindingen kunnen expliciet een *richting* aanduiden: in dat geval spreekt men van een gerichte graaf. Een verbinding wordt dan aangeduid door zijn begin- en eindknoop, *in die volgorde*: verbinding (i, j) is dus niet dezelfde als verbinding (j, i). Bij een ongerichte graaf bestaat dat onderscheid niet: beide stellen dezelfde verbinding voor. Men kan een ongerichte graaf beschouwen als een *symmetrische* gerichte graaf door de ongerichte verbinding tussen i en j te beschouwen als een verkorte notatie voor een gerichte verbinding (i, j) samen met de gerichte verbinding (j, i)

De *graad* van een knoop van een ongerichte graaf is het aantal buren van die knoop. Bij gerichte grafen heeft elke knoop een *ingraad* (het aantal inkomende verbindingen) en een *uitgraad* (het aantal uitgaande verbindingen).

De verbindingen kunnen bepaalde eigenschappen hebben, zoals een naam, een voorwaarde (denk daarbij aan de wachters van bijvoorbeeld een UML-statendiagram), een afstand, een kost, een tijd, enz. In dat geval kent men aan elke verbinding een *etiket* toe en men spreekt van een *geëtiketteerde* graaf. Als de etiketten getallen zijn die men

[9] In de praktijk hebben knopen vaak namen, zodat een bijkomende gegevensstructuur vereist is om namen af te beelden op gehele getallen.

zinvol kan optellen en met nul vergelijken dan noemt men de etiketten *gewichten* (vaak een reëel of geheel getal) die die hoedanigheid voorstellen. Men spreekt dan van een gewogen graaf.

Algoritmen maken meestal gebruik van een van de twee basisvoorstellingen van grafen:

- *Burenmatrix.* Deze stelt een verbinding (i, j) voor door het element op rij i en kolom j van een vierkante $(n \times n)$ burenmatrix. Dat element kan een logische waarde hebben, die het al dan niet bestaan van de verbinding aangeeft, of kan het etiket van de verbinding bevatten, waarbij een niet bestaande verbinding dan aangeduid wordt door een speciaal etiket, zoals bijvoorbeeld $0, +\infty$, of $-\infty$.

  Deze matrixvoorstelling is uiteraard zeer geschikt om snel te testen of er een verbinding bestaat tussen twee knopen. Bovendien kan men snel te weten komen van welke knopen een gegeven knoop buur is.

  Matrices nemen echter vlug veel geheugen in beslag, en zelfs hun initialisatie vereist reeds $\Theta(n^2)$ operaties, zodat deze voorstelling enkel bruikbaar is voor kleinere n, of voor een *dichte* graaf met relatief veel verbindingen, want dan moeten er toch $\Omega(n^2)$ gegevens opgeslagen worden. Bij ongewogen grafen kan men desnoods plaats winnen door als matrixelementen de afzonderlijke geheugenbits te gebruiken.

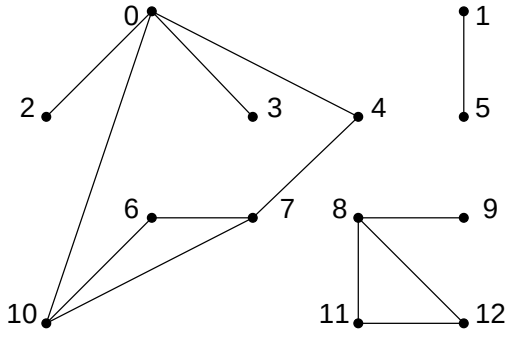
  Een burenmatrix voor een ongerichte graaf is symmetrisch, zodat het soms de moeite loont om slechts de helft van de matrix op te slaan.

- *Burenlijsten.* Deze voorstelling is het meest geschikt voor een *ijle* graaf met relatief weinig verbindingen ($m \ll n^2$). In de praktijk zijn de meeste grote grafen ijl, zodat deze voorstelling meest gebruikt wordt. (Vlakke grafen, die men kan tekenen zonder dat de verbindingen snijden, zijn steeds ijl.) De buren van elke knoop worden nu in een burenlijst opgeslagen (een tabel of lijst), en de graaf wordt voorgesteld door een tabel van n burenlijsten. Deze voorstelling reserveert dus enkel plaats voor de verbindingen die effectief aanwezig zijn. Gewoonlijk is de volgorde van de buren in een lijst willekeurig.

  Hoe men de lijst opslaat hangt, zoals altijd, af van het gebruik dat men ervan maakt. Gaat het om een graaf waarbij men de burenlijst vaak linear moet doorlopen dan maakt men bijvoorbeeld gebruik van een gelinkte lijst. Weet men bij constructie van de graaf reeds hoe groot de lijsten zijn dan is een tabel efficiënter qua tijd- en geheugengebruik. Moet men vaak opzoeken of twee knopen een verbinding hebben dan kan het gebruik van een rood-zwarte boom interessant zijn. Nagaan of er een verbinding bestaat tussen twee knopen is hier minder efficiënt dan bij een burenmatrix.

  Bij gewogen grafen bevat de burenlijst ook nog het gewicht van de verbinding naar elke buur. Bij een ongerichte graaf komt elke verbinding (i, j) tweemaal in deze lijsten voor: knoop i staat in de burenlijst van knoop j, en vice versa.

Figuur 8.6 toont een ongerichte, niet samenhangende graaf, en figuur 8.7 bevat zijn burenmatrix en burenlijsten.



Figuur 8.6. Ongerichte graaf.

0	2	10	3	4
1	5			
2	0			
3	0			
4	0	7		
5	1			
6	10	7		
7	6	10	4	
8	11	12	9	
9	8			
10	7	6	0	
11	12	8		
12	8	11		

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	1	1	1	0	0	0	0	0	1	0	0
1	0	0	0	0	0	1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	1	0	0	0	0	0
5	0	1	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	1	0	0	1	0	0
7	0	0	0	0	1	0	1	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	0	1	0	1	1
9	0	0	0	0	0	0	0	0	1	0	0	0	0
10	1	0	0	0	0	0	1	1	0	0	0	0	0
11	0	0	0	0	0	0	0	0	1	0	0	0	1
12	0	0	0	0	0	0	0	0	1	0	0	1	0

Figuur 8.7. Burenlijsten en burenmatrix voor de ongerichte graaf van figuur 8.6.

Om efficiëntieredenen kunnen deze basisvoorstellingen uitgebreid worden met allerlei bijkomende gegevensstructuren. Het zoveelste compromis tussen plaats en tijd.

De efficiëntie van graafalgoritmen hangt in het algemeen zowel af van het aantal knopen n als van het aantal verbindingen m. Bij de matrixvoorstelling echter domineert meestal het aantal knopen, omdat het de grootte van de matrix bepaalt.

8.8.2 Systematisch overlopen van grafen

Net zoals bij bomen komt het vaak voor dat elke knoop van een graaf moet behandeld worden. Bij bomen is er slechts één mogelijke weg tussen twee knopen. Bij een graaf kan een knoop meestal langs veel verschillende wegen bereikt worden, en moet men erop letten dat elke knoop slechts eenmaal behandeld wordt.

Bij bomen valt het wellicht niet op dat samen met de knopen ook alle takken overlopen worden (bij elke knoop, behalve de wortel, hoort eigenlijk één tak). Ook bij grafen worden samen met de knopen alle verbindingen systematisch onderzocht, en dit levert essentiële informatie op over de structuur van de graaf.

De meest gebruikte methode is *diepte-eerst zoeken*, een uitbreiding van het recursief overlopen van bomen. Ook overlopen in level order heeft hier zijn tegenhanger, *breedte-eerst zoeken*.

8.8.2.1 Diepte-eerst zoeken

Deze systematische methode om alle knopen en verbindingen te onderzoeken ligt aan de basis van een verrassend groot aantal toepassingen.[10] De methode is analoog aan recursief overlopen van (algemene) bomen (in preorder of postorder): beginnend bij de wortel, worden eerst alle kinderen van een bezochte knoop afgewerkt, vooraleer de knoop zelf afgewerkt (definitief verlaten) wordt. Een graaf heeft echter geen wortel, men kan dus bij elke knoop beginnen. Een knoop heeft ook geen kinderen, maar buren. En om te vermijden dat knopen meermaals behandeld worden, moet men bijhouden welke buren reeds ontdekt werden (nog in behandeling of al afgewerkt), en enkel de niet ontdekte buren behandelen. Om dat te kunnen nagaan kan men voor elke knoop een logische waarde bijhouden. Pseudocode 8.4 toont een minimale versie, te interpreteren als lidfuncties van een graafobject.

```
void behandel_knoop (int i) {
  ontdekt[i] = true; // Voorlopig is dit alles
  for (alle buren j van knoop i)
     if (!ontdekt[j])
        behandel_knoop(j);
}

void diepte_eerst_zoeken () {
  // Initialisatie: nog geen knopen gezien
  vector<bool> ontdekt(n,false);
  // Behandel alle knopen
  for (int i = 0 ; i < n ; i++)
     if (!ontdekt[i])
        behandel_knoop(i); // Begin een nieuwe boom
}
```

Pseudocode 8.4. Diepte-eerst zoeken.

[10] De eerste gedocumenteerde toepassing dateert van lang voordat er sprake was van computers, en diende om een uitweg te vinden uit een doolhofstructuur (Tarry, 1895).

Het behandelen van een knoop behandelt ook alle knopen die van daaruit bereikbaar zijn. Als er na het behandelen van de eerste knoop nog knopen overblijven, moet men een resterende knoop behandelen, en daarna eventueel nog meer, tot uiteindelijk alle knopen behandeld werden.

Tijdens de uitvoering van diepte-eerst zoeken kan men conceptueel de knopen in drie groepen onderverdelen, die men door drie kleuren kan kenmerken: de nog niet ontdekte knopen (wit), de knopen die reeds ontdekt zijn maar nog niet volledig afgewerkt (grijs), en deze die afgewerkt zijn (zwart). Oorspronkelijk zijn alle knopen wit, na afloop zijn ze allemaal zwart. Het behandelen van een knoop begint met hem grijs te kleuren (hij was immers wit), en nadat zijn burenlijst doorlopen werd, wordt hij zwart gemaakt. Soms moet diepte-eerst zoeken meer weten dan of een knoop reeds ontdekt is, en kunnen deze kleuren ook effectief bijgehouden worden.

Behalve de knopen onderzoekt diepte-eerst zoeken ook systematisch alle *verbindingen* van de graaf. De verbindingen vanuit een in behandeling zijnde (en dus grijze) knoop worden onderverdeeld in vier soorten:

- Een verbinding met een nog niet ontdekte (en dus witte) knoop is een boomtak (*tree edge*). Elke knoop kan immers slechts eenmaal ontdekt worden, en (behalve de beginknoop) vanuit slechts één andere knoop. Als men de beginknoop tot wortel maakt, dan vormen de boomtakken (met hun eindknopen) inderdaad een boom.

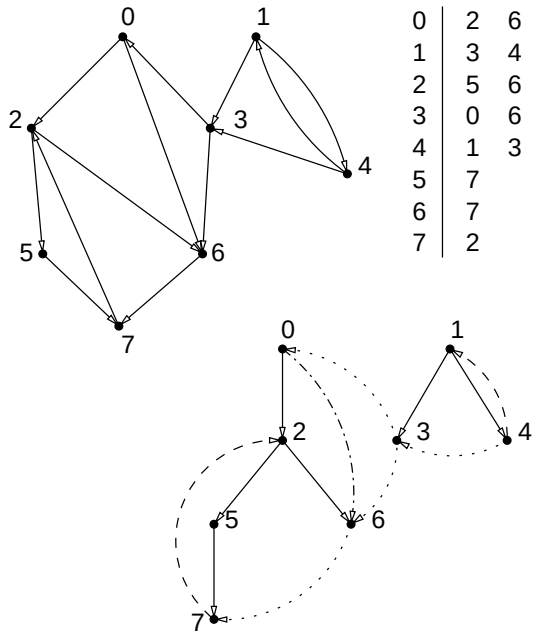
  Wanneer die boom niet alle graafknopen bevat, dan zijn er meerdere bomen, die samen een bos vormen. Een boom die alle knopen van een (deel)graaf bevat, en waarvan de takken (deel)graafverbindingen zijn, noemt men een overspannende boom (*spanning tree*)van die (deel)graaf.

- Een verbinding met een grijze knoop heet een terugverbinding (*back edge*). Die grijze knoop ligt steeds in dezelfde boom als de behandelde knoop, en is er bovendien een voorloper van. (Waarom?)

- Een verbinding met een zwarte opvolger (en dus in dezelfde boom), noemt men een heenverbinding (*forward edge*).

- Een verbinding met een zwarte knoop die geen opvolger is, heet een dwarsverbinding (*cross edge*). Die knoop ligt ofwel in dezelfde boom (in een nevendeelboom), of in een andere boom van het bos. Aangezien men de diepte-eerst bomen gewoonlijk van links naar rechts ‘tekent’, net zoals de boomtakken binnen dezelfde boom, wijzen dwarsverbindingen steeds van rechts naar links.

Figuur 8.8 toont een gerichte graaf, zijn burenlijsten en zijn diepte-eerst bos, met boomtakken (volle lijn), terugverbindingen (streeplijn), heenverbindingen (punt-streeplijn) en dwarsverbindingen (stippellijn). Boomtakken en terugverbindingen herkennen is eenvoudig, maar voor heen- en dwarsverbindingen volstaat de kleur van de eindknoop niet, en moet men bijvoorbeeld zijn preordernummer gebruiken om ze van elkaar te onderscheiden.



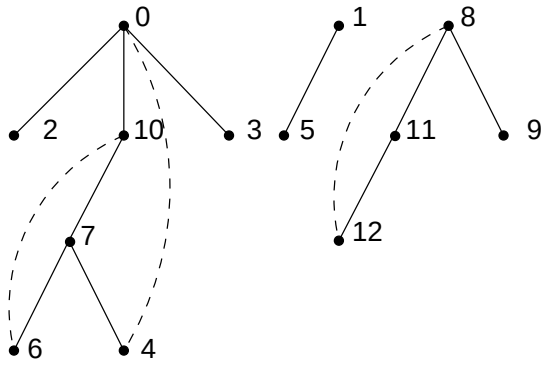
Figuur 8.8. Gerichte graaf met burenlijsten en diepte-eerst bos.

Aangezien een opvolger vanuit een voorloper kan bereikt worden via de boomtakken, betekent een terugverbinding een *lus* in de graaf. Een graaf zonder terugverbindingen is dus lusvrij. Bemerk dat alle knopen van een lus steeds tot dezelfde boom behoren.

De voorstelling van ongerichte grafen bevat elke verbinding tweemaal, zodat diepte-eerst zoeken elke verbinding dan ook tweemaal tegenkomt (eenmaal in elke richting). De classificatie van een verbinding moet bij de eerste keer gebeuren. Met als gevolg dat ongerichte grafen geen heen- en dwarsverbindingen kennen, omdat die verbindingen reeds vanuit de andere richting als terugverbinding, resp. boomtak, geclassificeerd werden. Let wel: in een ongerichte graaf is een verbinding van een kind naar zijn ouder geen terugverbinding, maar een boomtak die voor de tweede keer ontmoet wordt. Figuur 8.9 toont het diepte-eerst bos van de ongerichte graaf uit figuur 8.6.

Bepaalde toepassingen maken gebruik van de volgorde waarin knopen door diepte-eerst zoeken ontdekt worden. Dit komt neer op het in *preorder* nummeren van de knopen in de boom, en kan eenvoudig ingebouwd worden in de recursieve implementatie van `behandel()`. Andere toepassingen vereisen de volgorde waarin de knopen door diepte-eerst zoeken afgewerkt worden (alle buren van een knoop werden dan overlopen). Dat bekomt men door de boom te nummeren in *postorder*, wat al even gemakkelijk te doen is.

Naast de bepaling van het pre- of postordernummer kunnen er nog andere operaties op



Figuur 8.9. Diepte-eerst bos voor de ongerichte graaf van figuur 8.6.

knopen gebeuren, zowel vóór het overlopen van de buren, als erna, of zelfs ertussen. Het eenvoudige schema van diepte-eerst zoeken kan aldus op verschillende manieren aangevuld worden, naargelang de toepassing. We zullen daar nog enkele voorbeelden van zien.

De efficiëntie van diepte-eerst zoeken wordt bepaald door de voorstelling van de graaf:

- In beide gevallen vereist de initialisatie $\Theta(n)$ operaties.
- Bemerk dat zoeken naar een nieuwe startknoop niet telkens vooraan herbegint, maar vanaf de plaats waar het de vorige keer gestopt is. De tweede `for` verricht dan slechts $\Theta(n)$ testen in plaats van $O(n^2)$.
- Elke knoop wordt eenmaal ontdekt, en dus ook eenmaal behandeld, wat $\Theta(n)$ operaties vergt.
- Van elke knoop worden alle buren onderzocht. Bij een voorstelling met burenlijsten moeten dus alle burenlijsten overlopen worden, zodat het aantal operaties evenredig is met het aantal verbindingen. (De totale lengte van de burenlijsten van een gerichte graaf is gelijk aan m, bij een ongerichte graaf echter $2m$.) Bij een burenmatrix vinden we de buren van een knoop door een volledige rij van de matrix te overlopen, zodat het aantal operaties nu evenredig is met de grootte van de matrix.

In totaal krijgen we dus $\Theta(n+m)$ voor burenlijsten, en $\Theta(n+n^2) = \Theta(n^2)$ voor een burenmatrix. Voor ijle grafen nemen burenlijsten niet alleen minder plaats in dan een burenmatrix, ze maken diepte-eerst zoeken ook efficiënter.

8.8.2.2 Breedte-eerst zoeken

Zoals reeds vermeld is dit een uitbreiding van overlopen in level order van een boom. Daarbij worden eerst alle kinderen van de wortel behandeld (diepte één), dan alle kinderen van deze knopen (diepte twee), enz. Het aantal takken dat men vanuit de wortel moet volgen om een knoop te bereiken, bepaalt dus zijn niveau. Bij grafen is dat weer niet zo eenvoudig, omdat knopen langs meerdere wegen kunnen bereikt worden. De niveaus worden nu gedefinieerd door het *kleinste* aantal verbindingen om hun knopen te bereiken, vanuit een startknoop. Om de knopen in de juiste volgorde te behandelen kunnen we net zoals bij bomen een *wachtrij* gebruiken, waarin nu echter enkel nog *niet behandelde* knopen terechtkomen.

Wanneer we een knoop behandelen, plaatsen we zijn nog niet ontdekte buren in de wachtrij (die zijn dan reeds ontdekt, maar nog niet behandeld). Dan halen we de volgende te behandelen knoop uit de wachtrij. Dat betekent dat de wachtrij enkel knopen uit (hoogstens) twee opeenvolgende niveaus bevat.

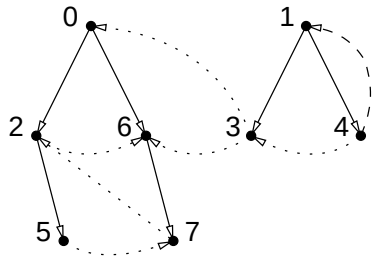
Tijdens de uitvoering van breedte-eerst zoeken kunnen we de knopen opnieuw conceptueel in drie groepen onderverdelen (met bijbehorende kleuren): de nog niet ontdekte knopen (wit), de reeds ontdekte maar nog niet behandelde knopen in de wachtrij (grijs), en de reeds behandelde knopen (zwart). Oorspronkelijk zit enkel de (grijze) wortel in de wachtrij, en zijn alle andere knopen wit. Na afloop zijn alle knopen zwart.

Ook hier kunnen we alle verbindingen klasseren, wanneer de buren van een ontdekte knoop overlopen worden. Bij ongerichte grafen ontmoeten we elke verbinding tweemaal, zodat deze classificatie opnieuw de eerste maal moet gebeuren:

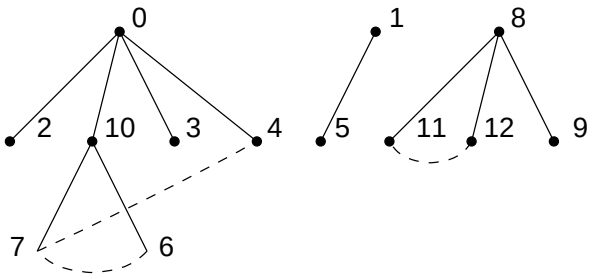
- Een verbinding met een nog niet ontdekte knoop is weer een *boomtak*.
- Een verbinding met een voorouder (dus in dezelfde boom), is een *terugverbinding*. Een ongerichte graaf heeft nooit terugverbindingen. (Waarom?)
- Een verbinding met een knoop die geen voorouder is (en geen kind), is een *dwarsverbinding*. Bij ongerichte grafen verbindt een dwarsverbinding knopen van dezelfde boom, op gelijk niveau, of met één niveau verschil. Bij gerichte grafen kan een dwarsverbinding leiden naar een knoop van dezelfde boom, op hetzelfde niveau, of op één niveau lager, of op elk hoger niveau (behalve uiteraard naar de wortel, want dat is een voorouder), en ook naar elke knoop van een andere (links gelegen) boom.

Heenverbindingen kunnen hier niet voorkomen. (Waarom?)

Figuur 8.10 toont het breedte-eerst bos van de gerichte graaf uit figuur 8.8, en figuur 8.11 toont dat bos voor de ongerichte graaf van figuur 8.6. De boomtakken vormen opnieuw een overspannende boom van de graaf. Bemerk dat een terugverbinding betekent dat de gerichte graaf een lus heeft. (Het omgekeerde is echter niet noodzakelijk waar.) Ook een dwarsverbinding bij een ongerichte graaf vormt een tweede



Figuur 8.10. Breedte-eerst bos voor de gerichte graaf van figuur 8.8.



Figuur 8.11. Breedte-eerst bos voor de ongerichte graaf van figuur 8.6.

verbinding tussen twee knopen (naast de boomtakken), en dus een lus. Breedte-eerst zoeken kan dus eveneens dienen om lussen in ongerichte grafen op te sporen.

De knopen in een wachtrij worden volgens het criterium 'first in, first out' geselecteerd. We zullen later enkele algoritmen bespreken, waar deze drie soorten knopen eveneens voorkomen, maar met een ander selectiecriterium (en dus ook een andere gegevensstructuur). Trouwens, ook diepte-eerst zoeken kan zo geïnterpreteerd worden, met een (impliciete) stapel voor de nog niet behandelde buren. (Of met een expliciete stapel, als de recursie verwijderd werd.)

Tenslotte bepalen we de efficiëntie:

- De initialisatie (alle knopen niet ontdekt) is $\Theta(n)$.
- Elke knoop wordt eenmaal op de wachtrij geplaatst en er weer afgehaald. Operaties op een wachtrij zijn $O(1)$, zodat dit voor alle knopen samen $\Theta(n)$ geeft.
- Alle buren van elke knoop worden overlopen, wat samen $\Theta(m)$ operaties vergt voor burenlijsten en $\Theta(n^2)$ voor een burenmatrix.

Afhankelijk van de voorstelling is het algoritme dus $\Theta(n + m)$ resp. $\Theta(n^2)$.

HOOFDSTUK 9
EENVOUDIGE WOORDENBOEKEN

Woordenboeken gebruiken de sleutels om gegevens op te zoeken, en dus ook om ze op een geschikte plaats op te slaan, waar ze liefst snel kunnen teruggevonden worden. Voor de eenvoud zullen we onderstellen dat alle sleutels *verschillend* zijn. De methodes kunnen echter vrij gemakkelijk voor duplicaten uitgebreid worden.

In dit hoofdstuk komen de lineaire woordenboekstructuren aan bod. De volgende twee hoofdstukken behandelen zeer belangrijke woordenboeken, hashtabellen en binaire zoekbomen.

9.1 TABEL

9.1.1 Rechtstreeks adresseerbare tabel

Wanneer er een een-eenduidig en eenvoudig verband bestaat tussen de sleutels en de tabelindices, dan is het niet nodig om de sleutels zelf op te slaan. De tabelindex van de bijbehorende informatie kan immers afgeleid worden uit de sleutel, en omgekeerd. Men spreekt dan ook van een *rechtstreeks adresseerbare tabel* ('direct addressable table').[1] Het ontbreken van een gegeven met een bepaalde sleutel zal ofwel moeten blijken uit een speciale code voor de (afwezige) bijbehorende informatie, ofwel uit een extra logische waarde. De bijbehorende informatie kan, als ze een vaste grootte heeft en niet veel plaats vergt, in de tabel zelf opgeslagen worden. Anders is het beter om een wijzer naar deze informatie te voorzien, zodat de tabel niet te groot wordt, en er enkel plaats gereserveerd wordt voor de aanwezige elementen.

Alle woordenboekoperaties zijn hier $O(1)$.

Als meerdere gegevens toch dezelfde sleutel hebben (duplicaten), dan kan bij de overeenkomstige tabelindex een gelinkte lijst voorzien worden met hun bijbehorende informatie.

[1] Bij specifieke toepassingen spreekt men soms van een *frequentietabel* of een *aanwezigheidstabel*. Andere namen zoals 'key addressable table' kunnen voor verwarring zorgen met associatieve gegevensstructuren, zoals de `map` in de Standard Template Library van C++.

Een belangrijke veralgemening van deze methode komt aan bod in hoofdstuk 10.

9.1.2 Ongeordende tabel

De woordenboekoperaties verlopen als volgt:

(1) *Zoeken.* De elementen van de tabel moeten overlopen worden tot de gezochte sleutel eventueel gevonden wordt. Dit heet dan ook *sequentieel* of lineair zoeken. Voor een afwezige sleutel doet deze methode n sleutelvergelijkingen. Voor een aanwezige sleutel, als elke sleutel met evenveel kans gezocht wordt, zijn er gemiddeld

$$(1 + 2 + \cdots + n)/n = (n + 1)/2$$

vergelijkingen nodig. In elk geval is sequentieel zoeken $O(n)$, en wordt daarom enkel gebruikt voor kleine tabellen, of wanneer er niet dikwijls in de tabel moet gezocht worden. Voor de andere gevallen bestaan er betere methodes die de sleutels in de tabel *herschikken* in een beter bruikbare volgorde.

(2) *Toevoegen.* Gebeurt achteraan en is dus $O(1)$.

(3) *Verwijderen.* Nadat de sleutel gevonden werd, kan men het element overschrijven met het laatste element. De sleutelvolgorde is hier immers onbelangrijk. Verwijderen zelf is dus $O(1)$.

9.1.3 Tabel geordend volgens zoekkans

Onderstellen dat elke sleutel met dezelfde waarschijnlijkheid zal gezocht worden is vaak niet realistisch. Indien men *op voorhand* weet dat bepaalde sleutels vaker zullen gezocht worden dan andere, kan men de sequentiële methode sneller maken door juist deze sleutels vooraan in de tabel te plaatsen. De optimale volgorde is natuurlijk volgens dalende waarschijnlijkheid.

Kent men echter deze waarschijnlijkheidsverdeling niet, dan kan men de vorige methode toch enigszins benaderen:

- Door de frequentie bij te houden waarmee elk element gezocht wordt, en de gegevens gerangschikt te houden volgens dalende frequentie.
- Door een gevonden element steeds met zijn voorloper (indien die bestaat) te *verwisselen* (‘transpose’). Op die manier krijgen we een dynamische verbetering van de volgorde, die goed blijkt te werken. Omdat het natuurlijk enige tijd duurt vooraleer de gegevens in de goede volgorde staan, is deze methode enkel bruikbaar voor een tijdsinvariante waarschijnlijkheidsverdeling.

9.1.4 Gerangschikte tabel

(1) *Zoeken.* Dit is al behandeld. Zoals gezien is binair zoeken $O(\lg n)$, veel efficiënter dan zoeken in een niet-gesorteerde tabel.
(2) *Toevoegen.* Nadat de plaats voor de nieuwe sleutel gelokaliseerd werd, moet gemiddeld de helft van de tabel opgeschoven worden, en in het slechtste geval zelfs de volledige tabel. Toevoegen is dus $O(n)$.
(3) *Verwijderen.* Net als bij toevoegen moet gemiddeld de helft van de tabel opgeschoven worden, en in het slechtste geval de volledige tabel. Verwijderen is dus ook $O(n)$.

9.2 LIJST

9.2.1 Ongeordende lijst

De woordenboekoperaties verlopen als volgt:

(1) *Zoeken.* Zoals bij een ongeordende tabel is zoeken sequentieel, en dus ook $O(n)$.
(2) *Toevoegen.* Gebeurt hier vooraan, en is dus $O(1)$.
(3) *Verwijderen.* Na zoeken is verwijderen nog $O(1)$. Bij een enkelvoudig gelinkte lijst heeft men daarbij de voorloper nodig.

9.2.2 Lijst geordend volgens zoekkans

De beste volgorde is opnieuw volgens dalende zoekwaarschijnlijkheden. Kent men die niet op voorhand, dan kan men trachten de optimale volgorde dynamisch te benaderen door een gevonden sleutel om te wisselen met zijn voorloper ('transpose'), of meteen helemaal *vooraan* te plaatsen ('move-to-front'). In de praktijk blijkt de eerste methode meestal beter, de tweede past zich sneller aan. (Bemerk dat de tweede methode niet efficiënt is bij tabellen.)

9.2.3 Gerangschikte lijst

De woordenboekoperaties verlopen als volgt:

(1) *Zoeken.* De voordelen die de rechtstreekse toegang van tabelelementen bood, ontbreken hier. Er bestaat dus geen equivalent van binair of interpolerend zoeken. Rangschikken verkort enkel de gemiddelde zoektijd naar een ontbrekend element, maar de performantie wordt toch niet beter dan $O(n)$.

(2) *Toevoegen.* De lijst moet nu sequentieel overlopen worden om de nieuwe plaats te lokaliseren. (Bij een enkelvoudig gelinkte lijst heeft men daarbij de voorloper nodig.) Gemiddeld en in het slechtste geval is toevoegen dus $O(n)$.
(3) *Verwijderen.* Analoog als toevoegen.

HOOFDSTUK 10
HASHTABELLEN

10.1 DEFINITIE

Veel toepassingen vereisen een gegevensstructuur die enkel de drie woordenboekoperaties ondersteunt: zoeken, toevoegen, en verwijderen. Een typisch voorbeeld is de naamtabel ('symbol table') die door een compiler wordt bijgehouden: de sleutels van de gegevens zijn de namen van de verschillende elementen die in een programma voorkomen, zoals variabelen, types, functies en procedures, en de naamtabel houdt allerlei informatie over die elementen bij.

Een hashtabel is een efficiënte implementatie van een dergelijke gegevensstructuur. Het is eigenlijk een veralgemening van de bekende rechtstreeks adresseerbare tabel (zie 9.1.1), waarbij er een een-eenduidig verband bestond tussen de sleutels en de tabelindices. Omdat deze tabel plaats voorzag voor elke mogelijke sleutel, waren de woordenboekoperaties heel eenvoudig te implementeren, en zeer efficiënt: $O(1)$.

De rechtstreeks adresseerbare tabel wordt echter onbruikbaar als het aantal *mogelijke* sleutels groot is. Het beschikbare geheugen kan immers ontoereikend zijn voor een tabel met die afmetingen. (Een tabel vereist opeenvolgende geheugenplaatsen.) En zelfs als er genoeg geheugen is, kan het werkelijk aantal opgeslagen sleutels zo klein zijn ten opzichte van het aantal mogelijke sleutels, dat het grootste deel van de tabel niet (nuttig) gebruikt wordt. Alleen al de initialisatie van een dergelijke tabel vereist relatief veel werk.

Een hashtabel is een typisch compromis tussen plaats (geheugenruimte) en tijd (uitvoeringstijd). Immers, mocht plaats geen rol spelen, dan zouden we natuurlijk een rechtstreeks adresseerbare tabel gebruiken. En als de performantie onbelangrijk was, dan volstond zoeken in een ongeordende tabel die enkel de opgeslagen sleutels bevat. Een hashtabel is niet alleen van vergelijkbare grootte als deze ongeordende tabel, maar is bovendien nagenoeg even efficiënt als de rechtstreeks adresseerbare tabel. De prijs die we daarvoor betalen is dat enkel de *gemiddelde* performantie uitstekend is, niet die voor het slechtste geval.

Aangezien er veel meer mogelijke sleutels zijn dan tabelplaatsen, kan er geen een-eenduidig verband meer bestaan tussen sleutels en indices. Met elke sleutel komt nog steeds één tabelindex overeen, maar niet omgekeerd. Deze index wordt uit de sleutel afgeleid via een *hashfunctie*.

Als deze hashfunctie voor elke op te slagen sleutel een verschillende index oplevert, dan volstaat een tabel met grootte n voor even zoveel sleutels.[1] Als de sleutels op voorhand gekend zijn, kan men inderdaad steeds een hashfunctie construeren die daarvoor zorgt (een 'perfecte' hashfunctie), ook al is dat niet triviaal. Voor sleutels die echter kunnen wijzigen (door verwijderen en toevoegen), faalt deze (statische) aanpak. Geen enkele hashfunctie kan immers perfect zijn voor elke verzameling van n sleutels. Want aangezien het aantal mogelijke sleutels groter is dan n, moet elke hashfunctie minstens twee sleutels aan dezelfde tabelindex toewijzen, zodat ze niet perfect kan zijn voor elke verzameling van n sleutels die deze twee sleutels bevat.

Wanneer twee sleutels bij dezelfde index terechtkomen spreekt men van een *conflict* ('collision'). Omdat perfecte hashfuncties niet mogelijk zijn bij sleutelwijzigingen, stelt men zich tevreden met bijna perfecte hashfuncties, die gemiddeld slechts een gering aantal conflicten veroorzaken. Elke hashtabel moet dan ook een mechanisme voorzien om conflicten op te vangen.

Een hashtabel implementeren vereist dus twee nagenoeg onafhankelijke keuzes: een geschikte hashfunctie, en een mechanisme om conflicten op te vangen.

10.2 HET OPVANGEN VAN CONFLICTEN

10.2.1 Chaining

Zoals de Engelse naam suggereert, maken deze methodes gebruik van gelinkte lijsten.

10.2.1.1 Separate chaining

Deze methode brengt alle sleutels die bij dezelfde tabelindex terechtkomen onder in een gelinkte lijst. De elementen van de hashtabel zijn dus lijsten. Elke lijstknoop bevat een sleutel met zijn bijbehorende informatie.

De implementatie van de woordenboekoperaties is eenvoudig:

(1) *Zoeken.* Via de hashwaarde van de gezochte sleutel komen we bij een gelinkte lijst terecht, waarin we enkel sequentieel kunnen zoeken. In het slechtste geval hebben alle sleutels in de tabel dezelfde hashwaarde, zodat ze allemaal in dezelfde lijst zitten. Zoeken is dan $O(n)$, zoals bij een gewone gelinkte lijst. Gelukkig is het gemiddeld geval veel beter.

[1] Een alternatieve naam voor een hashtabel, die nu niet meer gebruikt wordt, is 'scatter storage'.

Het gemiddeld gedrag hangt af van de kwaliteit van de hashfunctie, waarover later meer. Het na te streven ideaal is dat elke sleutel met dezelfde waarschijnlijkheid bij elk van de m tabelindices kan terechtkomen, en dit onafhankelijk van waar de andere sleutels terechtkwamen. Als we dit onderstellen, dan heeft het aantal elementen in elke lijst een binomiale waarschijnlijkheidsverdeling (zoals bij bucket sort), zodat de gemiddelde lengte van elke lijst n/m is. (Let wel, dit is niet het rekenkundig gemiddelde van alle lijstlengten, want dat is natuurlijk steeds n/m, maar de verwachtingswaarde van elke individuele lijstlengte.) De verhouding n/m noemt men de *bezettingsgraad* α ('load factor') van de hashtabel. Bemerk dat α hier zowel groter dan, kleiner dan als gelijk aan één kan zijn.
Zoeken naar een afwezige sleutel in een gelinkte lijst is gemiddeld wat sneller als de lijst gerangschikt is. We hebben er echter alle belang bij om de lijsten (zeer) kort te houden, zodat rangschikken niet de moeite loont. Als de sleutel dus afwezig is, moeten we steeds een volledige lijst doorlopen. Aangezien we met dezelfde kans bij elke lijst kunnen terechtkomen, en elke lijst een gemiddelde lengte α heeft, is de gemiddelde zoektijd $\Theta(1 + \alpha)$. (De 1 staat voor de constant onderstelde tijd nodig voor de evaluatie van de hashfunctie, die gewoonlijk niet verwaarloosbaar is tegenover α, zie onder.)
Bij zoeken naar een aanwezige sleutel is de situatie wat ingewikkelder, aangezien de lijsten niet meer met dezelfde waarschijnlijkheid doorzocht worden. Hoe meer sleutels een lijst bevat, des te waarschijnlijker dat de aanwezige sleutel erin gezocht wordt. Toch kan men aantonen dat de gemiddelde zoektijd eveneens $\Theta(1 + \alpha)$ bedraagt.
We kunnen deze resultaten als volgt interpreteren: door de sleutels over m gelinkte lijsten te verdelen in de plaats van ze allemaal in één lijst op te slaan, wordt de gemiddelde zoektijd een factor m kleiner.

(2) *Toevoegen.* Aangezien de lijsten niet gerangschikt zijn, wordt een nieuw element vooraan toegevoegd. Dat heeft het bijkomend voordeel dat het snel gevonden wordt, wanneer men het kort nadien zoekt. (De meeste programma's vertonen immers deze 'temporal locality'.) Toevoegen is dus $O(1)$.

(3) *Verwijderen.* Verwijderen van een reeds gevonden element is ook $O(1)$.

Indien we nu voor al deze operaties een gemiddelde performantie van $O(1)$ wensen, dan moet $\alpha = O(1)$ worden, en dus $m = O(n)$. Dat betekent dat de grootte van de hashtabel minstens evenredig moet zijn met het aantal opgeslagen sleutels. In de praktijk neemt men gewoonlijk $m \approx n$. (Maar ook de hashfunctie speelt een rol in de keuze van m, zoals we zullen zien.)

10.2.1.2 Coalesced chaining

In plaats van een tabel van lijsten gebruikt coalesced chaining een tabel van lijstknopen: een tabel dus die op elke plaats niet alleen een waarde kan opslaan, maar ook een verwijzing naar een ander element van de tabel.

Wat er juist gebeurt kan het gemakkelijkst worden uitgelegd aan de hand van toevoe-

gen. Als we een element willen toevoegen berekenen we de hashwaarde en kijken op de gepaste plaats in de tabel (noemen we deze A). Als deze leeg is dan vullen we ze met het het element. Nu hoort er bij de tabel een wijzer naar een lege plaats L die gebruikt kan worden bij collisies. Als A al bezet is dan plaatsen we het element in deze plaats. Nu moeten we er ook voor zorgen dat we het element nog kunnen terugvinden. Als A een nullwijzer bevat overschrijven we die met het adres van L; anders is A het begin van een lijst: er zit een verwijzing in naar een volgende plaats in de tabel, die eventueel ook wijst naar een volgende plaats, ... We komen echter uit op een plaats in de tabel met een nullwijzer en overschrijven die met het adres van L. Tenslotte moeten we nog een verwijzing vinden naar de volgende lege plaats die een collisie kan opvangen.

Een plaats die opgevuld werd bij een collisie kan deel uitmaken van verschillende lijsten (omdat ze als index een geldige hashwaarde heeft of omdat een element geschrapt is en de plaats later opnieuw gebruikt wordt). Hierdoor gaan die lijsten samenklitten (vandaar 'coalesced').

Coalesced chaining heeft net zoals open adressering (zie onder) het voordeel dat knopen niet meer dynamisch moeten aangemaakt (en later eventueel verwijderd) worden. Dat is echter ook een nadeel wanneer het aantal op te slagen elementen niet op voorhand gekend is.

Coalesced chaining heeft een aantal varianten. Zo kan men het samenklitten van de lijsten uitstellen door de hashtabel onder te verdelen in een adreszone ('address region') en een berging ('cellar'). De hashfunctie genereert dan uitsluitend indices in de adreszone, en bij een conflict wordt een knoop uit de berging gebruikt om de lijst te verlengen. Pas wanneer alle knopen van de berging opgebruikt zijn, gebruikt men vrije knopen uit de adreszone om lijsten te verlengen.

De grootteverhouding tussen die twee delen zorgt voor een waaier van mogelijkheden, met verschillende performanties. (De analyses zijn behoorlijk ingewikkeld.) Standaard coalesced chaining zoals hierboven beschreven gebruikt helemaal geen berging, zodat de lijsten veel vroeger zullen samenklitten. Wanneer men een berging gebruikt die toereikend is voor alle conflicten, blijven de lijsten volledig gescheiden: coalesced hashing wordt dan separate chaining. In principe kan deze grootteverhouding voor elke waarde van de bezettingsgraad optimaal gekozen worden. Een adreszone die 86% van het totaal inneemt blijkt echter een goed compromis voor veel bezettingsgraden.

De woordenboekoperaties verlopen als volgt:

(1) *Zoeken.* Zoeken begint in de adreszone, bij de index die de hashfunctie oplevert. En stopt als de knoop op die plaats de sleutel bevat, of ledig is. (Een knoop moet dus kunnen aanduiden dat hij ledig is.) Als de knoop een andere sleutel bevat, volgen we opvolgerwijzers tot we een knoop met de sleutel vinden, of het einde van de lijst.
(2) *Toevoegen.* Dit proces hebben we reeds kort besproken. Alleen het aanduiden van de lege plaats voor een collisie moeten we nog behandelen. De meest gebruikte oplossing is om de *laatste* lege plaats in de tabel hiervoor te gebruiken. Als deze

wordt opgevuld zoekt men gewoon de vorige lege plaats in de tabel. Werkt men met een berging dan staat deze achteraan in de tabel, met als gevolg dat eerst de knopen van de berging gebruikt worden, en pas daarna deze van de adreszone. (Bij standaard coalesced chaining zullen de laatste tabelknopen daardoor echter sneller opvullen, zodat de lijsten vroeger samenklitten. Door een berging te gebruiken houdt men de lijsten langer apart.)

(3) *Verwijderen.* Omdat de lijsten kunnen samenklitten, is verwijderen niet zo eenvoudig. Immers, wanneer we een knoop uit de lijst zouden verwijderen, wordt hij opnieuw ledig. Maar de lijst kan sleutels bevatten, die net via die tabelplaats zijn toegevoegd. Zoeken zou ze dus niet meer terugvinden. Men kan dit oplossen door de knoop in de lijst te laten, en hem te merken als verwijderd ('lazy deletion'). Zoeken moet deze plaats dan negeren (alsof ze bezet was door een verkeerde sleutel), en toevoegen moet ze opvullen (alsof ze vrij was). De zoektijd wordt natuurlijk nadelig beïnvloed door die latent 'aanwezige' sleutels. Daarom verkiest men gewoonlijk separate chaining (of een andere gegevensstructuur) als de toepassing vereist dat er sleutels verwijderd worden.

Ondanks het feit dat de lijsten samenklitten, blijkt dat ze kort blijven. Men kan aantonen (en experimenteel bevestigen) dat coalesced chaining met een goed gekozen grootteverhouding beter is dan andere hashtabellen, wanneer $\alpha > 0.6$.

10.2.2 Open adressering

Open adressering slaat[2] alle n gegevens in de hashtabel zelf op, zoals coalesced hashing. Maar in plaats van conflicten op te lossen met gelinkte lijsten, worden de alternatieve plaatsen voor de gegevens *berekend*. Plaats (wijzers naar opvolgers) wordt dus geruild voor tijd (berekening). Een berging wordt hier niet gebruikt: de alternatieve plaatsen liggen vast. Aangezien alle gegevens in de tabel terechtkomen, moeten we op voorhand een idee hebben van hun aantal, om de tabelgrootte te kunnen kiezen. De bezettingsgraad $\alpha = n/m$ is hier dus nooit groter dan één.

De woordenboekoperaties verlopen nu als volgt:

(1) *Zoeken.* Zoeken begint bij de index die de hashfunctie oplevert (de *primaire* index). En stopt als de sleutel daar aangetroffen wordt. Als die plaats ledig blijkt (een tabelelement moet kunnen aanduiden dat het geen sleutel bevat), is de sleutel afwezig. Als die plaats een andere sleutel bevat (er is dus een conflict), dan zoekt men op andere (berekende) plaatsen tot ofwel de sleutel gevonden wordt (andere sleutels worden overgeslagen), ofwel een ledige plaats.

(2) *Toevoegen.* Als zoeken een ledige plaats oplevert (dit onderstelt dat de tabel niet vol is), dan wordt de sleutel daar toegevoegd. Hoewel open adresseren in principe een hashtabel volledig kan opvullen, wordt haar performantie dan zo slecht (zie later) dat men ervoor zorgt dat er steeds voldoende plaats overblijft. Voor

[2] Dit is de oorspronkelijke naam. Separate chaining wordt ook wel 'open hashing' genoemd, en open addressing 'closed hashing'.

hetzelfde aantal op te slagen sleutels zal de tabel dus meer elementen bevatten dan die van separate chaining, maar door het wegvallen van de opvolgerwijzers kunnen de totale geheugenvereisten toch kleiner zijn. (Voor grote sleutels slaat men beter wijzers op.)
Net zoals coalesced chaining heeft open adressering het voordeel dat de plaats voor een nieuwe sleutel reeds bestaat, en niet zoals bij separate chaining dynamisch moet aangemaakt (en later eventueel verwijderd) worden, want dat vergt vaak wel wat tijd. Een nadeel is dan weer dat een tabel opeenvolgende geheugenplaatsen vereist, en ook niet meer (efficiënt) kan uitbreiden.

(3) *Verwijderen.* Net zoals bij coalesced chaining is verwijderen van sleutels hier niet zo eenvoudig. Een tabelplaats kan immers tot meerdere zoeksequenties behoren, en als we haar opnieuw als ledig aanduiden, kan de zoeksequentie voor andere sleutels daar voortijdig afgebroken worden. Men kan dit weer oplossen door een tweede speciale code te voorzien, die aanduidt dat de inhoud van een plaats verwijderd werd ('lazy deletion'). Zoeken moet deze plaats dan negeren (alsof ze bezet was door een verkeerde sleutel), en toevoegen moet ze opvullen (alsof ze vrij was). De zoektijd hangt nu niet langer uitsluitend af van de bezettingsgraad α (zie later), want ook de verwijderde sleutels zijn nog latent 'aanwezig'. Daarom verkiest men gewoonlijk separate chaining (of een andere gegevensstructuur) als de toepassing vereist dat er sleutels verwijderd worden.

10.2.2.1 Bepalen van zoeksequenties

Drie technieken worden gebruikt om de alternatieve plaatsen te berekenen die bij een conflict onderzocht worden. Ze garanderen allemaal dat indien nodig de volledige tabel doorzocht wordt. (De gegenereerde indices vormen dus een permutatie van $\{0, \ldots, m-1\}$.) In het ideale geval zou voor elke sleutel elke permutatie even waarschijnlijk moeten zijn[3], maar deze 'uniforme hashing' blijkt zeer moeilijk te realiseren, zodat men zich tevreden moet stellen met benaderingen. Al deze methodes genereren trouwens slechts een minieme fractie van de $m!$ mogelijke permutaties.

- *Lineair testen (linear probing).* De zoeksequentie bestaat hierbij uit de indices

$$(h(s) + i) \bmod m \qquad \text{voor } i = 0, \ldots, m-1$$

Als de sleutel s niet bij de primaire index $h(s)$ gevonden wordt ($h()$ is de hashfunctie), dan zoekt men telkens één plaats verder (modulo de grootte van de tabel). Vermits de primaire index $h(s)$ de zoeksequentie volledig bepaalt, zijn er slechts m verschillende sequenties mogelijk.

Deze methode heeft het voordeel van haar eenvoud, maar leidt tot *primaire clustering*: lange reeksen bezette plaatsen worden gevormd rond de primaire sleutels, waardoor de gemiddelde zoektijd toeneemt. Stel bijvoorbeeld dat de tabel half

[3] Voor elke sleutel ligt deze permutatie natuurlijk vast. Het zijn de sleutels zelf die gegenereerd worden door een toevalsproces.

vol is ($\alpha = 0.5$) en dat alle even indices bezet zijn en de oneven indices vrij. Dan vereist zoeken naar een niet-aanwezige sleutel gemiddeld 1.5 testen. Als echter de eerste $m/2$ indices bezet zijn, dan wordt dit aantal nagenoeg $m/8 = n/4$. (Ga eens na.)

De kans dat er clusters ontstaan is reëel, want de kans dat een vrije plaats bezet wordt door de volgende sleutel is $(i + 1)/m$ als de i vorige plaatsen bezet zijn, en slechts $1/m$ als de vorige plaats vrij is.

- *Kwadratisch testen ('quadratic probing').* De zoeksequentie wordt hier gevormd door de indices

$$(h(s) + c_1 i + c_2 i^2) \bmod m \qquad \text{voor } i = 0, \ldots, m-1$$

waarbij c_1 en c_2 constanten zijn, verschillend van nul. Dit blijkt veel beter te werken dan de vorige methode, maar om een permutatie van alle indices te bekomen moeten c_1, c_2, en de tabelgrootte m aan bepaalde voorwaarden voldoen.

Sleutels met dezelfde primaire index hebben nog steeds dezelfde zoeksequenties, zodat er ook hier slechts m verschillende sequenties zijn. De primaire clustering is verdwenen, en vervangen door een lichtere vorm (secundair clusteren), die nog slechts in theorie een licht nadeel kan opleveren.

- *Dubbele hashing ('double hashing').* Dit is een van de beste methodes voor open adresseren omdat de zoeksequenties het ideaal van uniforme hashing wat meer benaderen. De geteste indices zijn nu

$$(h(s) + ih'(s)) \bmod m \qquad \text{voor } i = 0, \ldots, m-1$$

waarbij zowel $h()$ als $h'()$ hashfuncties zijn. De indexsprongen ten opzichte van de primaire index $h(s)$ worden dus eveneens door de sleutel bepaald: de zoeksequentie hangt nu op twee manieren van de sleutel af. (Daarom is het geen goed idee om tweemaal dezelfde hashfunctie te gebruiken - dit geeft aanleiding tot een nog meer gecompliceerde vorm van clustering.)

De waarde van $h'(s)$ moet wel relatief priem zijn ten opzichte van m opdat alle indices zouden kunnen gegenereerd worden. (Als hun grootste gemene deler $d > 1$ zou zijn dan zou slechts een d-de deel van de tabel kunnen getest worden.) Wanneer m priem is kunnen we er bijvoorbeeld voor zorgen dat $0 < h'(s) < m$, en wanneer m een macht van twee is maken we $h'(s)$ steeds oneven.

Het aantal verschillende zoeksequenties is nu gevoelig groter: elk paar $(h(s), h'(s))$ bepaalt immers een sequentie en voor elke nieuwe sleutel s kunnen beide waarden onafhankelijk van elkaar veranderen, zodat er nu $\Theta(m^2)$ mogelijkheden zijn.

10.2.2.2 Performantie van open adressering

De analyse van deze drie zoekmethodes is moeilijk, en valt buiten het opzet van de cursus. Het ideale geval van *uniforme hashing* maakt de analyse veel eenvoudiger,

maar de resultaten zijn uiteraard optimistisch. Uitgebreide experimenten laten echter zien dat dubbele hashing dit ideaal goed benadert.

Opnieuw wordt de performantie volledig bepaald door de bezettingsgraad α (die hier, zoals gezegd, nooit groter is dan één):

- Hoeveel testen zijn er gemiddeld nodig om een *afwezige* sleutel te zoeken? Als p_i de waarschijnlijkheid voorstelt dat er *exact* i testen gebeuren voor een afwezige sleutel, dan is het gemiddeld aantal testen gelijk aan

$$\sum_{i=1}^{n+1} i p_i$$

  Als q_i de waarschijnlijkheid voorstelt dat er *tenminste* i testen gebeuren, dan kunnen we die som vereenvoudigen tot

$$\sum_{i=1}^{n+1} i p_i = \sum_{i=1}^{n+1} i(q_i - q_{i+1}) = \sum_{i=1}^{n+1} q_i$$

  want elke term q_i wordt i maal opgeteld en $i-1$ maal afgetrokken, en q_{n+2} is nul. Deze nieuwe som bevat geen producten meer, en de waarden q_i zijn gemakkelijker te bepalen dan p_i.

  Vooreerst gebeurt er altijd tenminste één test, zodat $q_1 = 1$. Er zal pas (tenminste) een tweede test gebeuren als de eerste test een bezette plaats vindt (met de verkeerde sleutel). Aangezien n van de m plaatsen bezet zijn, en uniforme hashing met dezelfde kans op elke plaats kan beginnen, is $q_2 = n/m$. Er zal pas (tenminste) een derde test gebeuren als de twee vorige geteste plaatsen bezet waren. Bij elke test vermindert echter het aantal nog te testen plaatsen, zodat ook de kans op het vinden van een bezette plaats wijzigt. Nu is de kans dat twee gebeurtenissen E_1 en E_2 zich samen voordoen, gelijk aan de kans dat E_1 zich voordoet, maal de kans dat E_2 zich voordoet *op voorwaarde dat* E_1 zich heeft voorgedaan. We moeten dus voorwaardelijke waarschijnlijkheden gebruiken:

$$P\{E_1 \cap E_2\} = P\{E_1\} \cdot P\{E_2|E_1\}$$

  Als gebeurtenis E_1 betekent dat de eerste geteste plaats bezet is, en E_2 dat de tweede geteste plaats bezet is, dan bekomen we dat

$$q_3 = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)$$

  aangezien er bij de tweede test nog $n-1$ plaatsen bezet zijn van de $m-1$ resterende plaatsen, en we opnieuw met dezelfde kans op elk van die plaatsen kunnen terechtkomen (wegens de uniforme hashing). Op dezelfde manier bekomen we dan dat

$$q_i = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)\cdots\left(\frac{n-i+2}{m-i+2}\right) \quad \text{voor } 1 < i \leq n+1$$

Aangezien $n \leq m$ is $(n-j)/(m-j) \leq n/m$ voor $0 \leq j < n$, zodat

$$q_i \leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1} \quad \text{voor } 1 \leq i \leq n+1$$

Het gemiddeld aantal testen voor een afwezige sleutel is dan hoogstens (want $\alpha < 1$)

$$\sum_{i=1}^{n+1} q_i \leq \sum_{i=1}^{n+1} \alpha^{i-1} < \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$

Voor $\alpha = 0.5$ is die waarde gelijk aan twee, voor $\alpha = 0.9$ is ze reeds tien.

- Een sleutel wordt pas toegevoegd als hij afwezig blijkt. Het gemiddeld aantal testen voor toevoegen is dus hetzelfde als voor een afwezige sleutel.

- Hoeveel testen vereist zoeken naar een *aanwezige* sleutel? De zoeksequentie om de sleutel te bereiken zal dezelfde zijn als wanneer hij werd toegevoegd, en dus nog afwezig was. Als de sleutel werd toegevoegd aan een tabel die reeds i sleutels bevatte ($0 \leq i < n$), dan was het gemiddeld aantal testen hoogstens $1/(1-i/m) = m/(m-i)$. Als elke aanwezige sleutel met dezelfde waarschijnlijkheid gezocht wordt, dan is het gemiddeld aantal testen voor een aanwezige sleutel

$$\frac{1}{n}\sum_{i=0}^{n-1} \frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1} \frac{1}{m-i} = \frac{1}{\alpha}\sum_{k=m-n+1}^{m} \frac{1}{k}$$

$$< \frac{1}{\alpha}\int_{m-n}^{m} \frac{dx}{x} = \frac{1}{\alpha}\ln\left(\frac{m}{m-n}\right) = \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$$

Voor $\alpha = 0.5$ is die waarde 1.4, voor $\alpha = 0.9$ is ze 2.6.

Bemerk dat het aantal testen sterk stijgt als α nadert tot één, dus als de tabel vol geraakt. Om een goede performantie te verzekeren, zorgt men er in de praktijk voor dat α nooit groter wordt dan 0.5. Dat vereist natuurlijk dat men op voorhand een goed idee heeft van het maximaal aantal elementen dat in de tabel moet opgeslagen worden. Dat is niet altijd zo, en dan maar een zo groot mogelijke tabel reserveren is bij complexe programma's vaak niet mogelijk. In dat geval kiest men ofwel voor de methode met (separate) chaining, omdat de gelinkte lijsten de overbezetting wat soepeler opvangen, ofwel voorziet men de mogelijkheid tot *rehashing*.

Rehashing creëert een nieuwe hashtabel met (ongeveer) dubbele grootte en bijbehorende nieuwe hashfunctie(s), en voegt alle elementen uit de eerste tabel toe aan de tweede. (Gewoon kopiëren gaat natuurlijk niet.) Tenslotte wordt de eerste tabel dan vrijgegeven (expliciet of impliciet). Aangezien de eerste hashtabel toch sequentieel moet overlopen worden, is dat een goede gelegenheid om de als verwijderd gemerkte elementen definitief te elimineren. Rehashing is natuurlijk tijdrovend, want $O(m)$, maar door de tabel evenredig met m te laten groeien, komt dit neer op een extra kost van $O(1)$ per toevoegoperatie. (Een geamortiseerde kost, zoals bij dynamische tabellen.) Gemiddeld is het effect dus gering, maar voor toepassingen die vereisen dat elke

operatie efficiënt verloopt, mag men niet het risico lopen dat er op een cruciaal moment een rehashing plaatsvindt.

Rehashing kan best gebeuren wanneer α een vooropgezette waarde bereikt (niet méér dan 0.5). Wachten tot wanneer toevoegen geen vrije plaats meer vindt is geen goed idee, omdat de performantie dan al geruime tijd slecht kan zijn.

10.3 HASHFUNCTIES

Een goede hashfunctie moet snel kunnen geëvalueerd worden (dat is onder meer afhankelijk van de processor), en moet het aantal conflicten zo klein mogelijk maken (dat hangt af van de sleutels).

Conflicten minimaliseren betekent dat de hashfunctie ervoor moet zorgen dat iedere tabelindex even waarschijnlijk is voor elke sleutel, en dit onafhankelijk van de indices voor de andere sleutels.[4] Nog anders gezegd, de waarschijnlijkheid dat twee verschillende sleutels dezelfde index opleveren moet gelijk zijn aan $1/m$. (Dat is trouwens de waarschijnlijkheid dat twee willekeurig gekozen indices gelijk zijn.) Deze eigenschap heet *enkelvoudige uniforme hashing* (om het onderscheid te maken met de uniforme hashing bij de zoek*sequenties* van open adresseren, die eigenlijk meervoudig is).

Een dergelijke functie ontwerpen is niet zo eenvoudig, omdat de waarschijnlijkheidsverdeling van de sleutels meestal niet bekend is. Dikwijls gebruikt men heuristische technieken om zo goed mogelijke functies te bekomen. Soms heeft men wel enig idee over de verdeling van de sleutels: een naamtabel van een compiler moet vaak werken met namen die nauw verwant zijn (‘max’, ‘maxx’, ‘maxy’, . . .). Een goede hashfunctie zorgt er in dit geval voor dat de kans klein is dat al die namen bij dezelfde index terechtkomen. In elk geval tracht men een functie te vinden waarvan de resultaten onafhankelijk zijn van enig patroon in de gebruikte sleutels. Soms móet een hashfunctie zelfs afwijken van het ideaal van de enkelvoudige uniforme hashing. Zo bijvoorbeeld bij open adressering met lineair testen, waarbij verwante sleutels het best ver uiteenliggende indices opleveren.

10.3.1 Vaste hashfuncties

Voorlopig onderstellen we dat de sleutels in een processorwoord passen, en dus positieve gehele getallen zijn of als zodanig kunnen geïnterpreteerd worden. Uitvoerige testen op typische gegevens hebben aangetoond dat twee soorten hashfuncties zeer goed werken. De ene is gebaseerd op delen, de andere op vermenigvuldigen:

- *Delen.* Deze vaak gebruikte methode zet een geheel getal s om in een index uit

[4] Bemerk evenwel dat een hashfunctie deterministisch is: de index voor elke sleutel ligt vast zodra de functie gedefinieerd werd.

$\{0, \ldots, m-1\}$ door de rest te berekenen bij deling door m:

$$h(s) = s \bmod m$$

Dit is eenvoudig en efficiënt, en zorgt ervoor dat willekeurige ('random') gekozen sleutels uit een uniforme verdeling gelijkmatig over de tabel verdeeld worden.

De grootte m van de hashtabel moet wel met enige zorg gekozen worden:

- Als m even zou zijn dan krijgen even sleutels even indices, en oneven sleutels oneven indices. Bij veel toepassingen zou dit voor veelvuldige conflicten zorgen.
- Als $m = 2^i$ dan bestaat de index uit de laatste i bits van de sleutel. Tenzij men op voorhand weet dat alle bitpatronen met lengte i op het einde van een sleutel even waarschijnlijk zijn, neemt men beter een hashfunctie die afhangt van *alle* bits uit de sleutel. Trouwens mochten sleutels echt 'random' zijn, dan zou om het even welk deel ervan voor de hashfunctie kunnen gebruikt worden.
- Machten van tien zijn evenmin een goede keuze voor m als de sleutels decimale getallen zijn, om analoge redenen.

Goede waarden voor m blijken bijvoorbeeld priemgetallen die niet te dicht bij machten van twee liggen. Het kan overigens nooit kwaad om eens na te gaan hoe gelijkmatig een hashfunctie sleutels verdeelt over de tabel, door ze vooraf op 'echte' gegevens uit te proberen.

- *Vermenigvuldigen.* Deze methode werkt in twee stappen: de sleutel wordt vermenigvuldigd met een constante fractie C ($0 < C < 1$), en het deel na de komma wordt behouden. Dit vermenigvuldigen we met m en we ronden af naar beneden. Kort genoteerd:

  $$h(s) = \lfloor m(sC - \lfloor sC \rfloor) \rfloor$$

  Bij deze methode blijkt de waarde van m niet kritisch te zijn. Daarom kiest men gewoonlijk $m = 2^i$ omdat de hashfunctie dan eenvoudig kan geïmplementeerd worden, en bovendien zonder vlottendekommabewerkingen. Stel dat de woordbreedte van de processor w is. Dan vermenigvuldigt men eerst de sleutel met het woord $\lfloor C2^w \rfloor$, wat een geheel resultaat van maximaal $2w$ bits oplevert (twee woorden), dat echter een factor 2^w te groot is. Het deel na de komma dat we moeten behouden staat dus in het minst significante woord van dat product. En vermenigvuldigen met m, gevolgd door het weglaten van het deel na de komma, betekent eenvoudig dat de gezochte hashwaarde bestaat uit de i meest significante bits van dat woord.

  Deze methode werkt voor elke waarde voor C, maar sommige waarden zijn beter, afhankelijk van de sleutels die gebruikt worden. Een diepere analyse suggereert dat

  $$C \approx (\sqrt{5} - 1)/2 = 0.61803\ldots \qquad \text{(de gulden snede)}$$

een goede kans maakt om behoorlijk te werken (zie Knuth [11]).

Voor grote sleutels die niet in een processorwoord passen, eventueel met variabele lengte (zoals strings), zijn er een aantal mogelijkheden, die soms kunnen ingepast worden in de eigenlijke hashfunctie. In principe zou men elk lang bitpatroon kunnen beschouwen als een (groot) geheel getal, en er dezelfde bewerkingen op uitvoeren als op woorden, door elk van die (hardware)bewerkingen in software te simuleren voor meerdere woorden. Dat is echter niet zeer efficiënt.

De afzonderlijke woorden van lange sleutels kunnen ook opgeteld[5] worden (modulo 2^w), of met bit-per-bit exclusieve-of operaties samengevoegd worden. Beide bewerkingen hebben het nadeel dat ze commutatief zijn: de woordparen (w_1, w_2) en (w_2, w_1) geven hetzelfde resultaat. Men kan dit oplossen door vooraf één van de woorden te roteren.

Strings worden vaak beschouwd als (grote) gehele getallen in een talstelsel met radix r, die men dan efficiënt evalueert met de bekende methode van Horner. Hoewel de keuze $r = 128$ (of 256) voor de hand ligt, kan die problemen geven wanneer m even is of een macht van twee. Daarom neemt men meestal een priemgetal. Vaak gebruikt men 37, 31 of ook wel 127. Om te vermijden dat deze getallen te groot worden, kan men de berekeningen modulo 2^w uitvoeren[6] (dat is zeer efficiënt - opgelet voor negatieve resultaten), of modulo m bij een hashfunctie met deling (het resultaat is dan toch hetzelfde). Als de strings lang zijn gebruikt men hierbij soms slechts een gedeelte (dat dan hopelijk 'random' genoeg is), zoals bijvoorbeeld enkel de oneven posities.

Een nog betere mogelijkheid voor sleutels die uit k elementen (woorden of karakters) bestaan, is onafhankelijke hashfuncties te gebruiken voor elk element, en de resultaten weer samen te stellen. Voor een sleutel $s = \langle e_1 e_2 \dots e_k \rangle$ is de hashfunctie dan

$$h(s) = (h_1(e_1) + h_2(e_2) + \dots + h_k(e_k)) \bmod m$$

Vooral bij strings kan dit efficiënt gebeuren door de waarden van deze hashfuncties vooraf in tabellen op te slaan. Als m een macht van twee is kunnen we de deling nog vermijden door de optellingen te vervangen door exclusieve-of operaties.

10.3.2 Universele hashing

Stel dat de hashfunctie vastligt. Dan is het steeds mogelijk dat een toepassing net die sleutels gebruikt die allemaal bij dezelfde tabelindex terechtkomen, met alle gevolgen van dien. De enige manier om dit te vermijden is random een hashfunctie te kiezen bij ingebruikname van een hashtabel, zodat het gedrag van de hashtabel onafhankelijk wordt van de gebruikte sleutels. De hashtabel kan zich dan bij elke uitvoering anders gedragen, ook voor dezelfde sleutels. Deze werkwijze heet *universele hashing*.

[5] Voor strings is het optellen van de waarden van de afzonderlijke *karakters* geen goed idee. Als de hashtabel vrij groot is, leveren korte strings waarden in een te beperkt bereik op.

[6] In dat geval wordt als radix $r = 131$ aanbevolen, omdat $r^i \bmod 2^w$ dan een maximale cyclus heeft voor de gangbare woordlengten $8 \leq w \leq 64$.

Dit is weer een mooi voorbeeld van een 'randomized' algoritme (of gegevensstructuur). Het idee is vergelijkbaar met een versie van randomized quicksort, waar een random permutatie van de invoertabel garandeert dat geen enkele tabel stelselmatig de slechtste performantie veroorzaakt.

Stel dat er k mogelijke sleutels zijn. (Dat aantal kan zeer groot zijn. Denk bijvoorbeeld aan alle mogelijke strings met niet meer dan 32 karakters.) Een hashfunctie moet elk van die sleutels kunnen omzetten in een van de m tabelindices. Bij een ideale hashfunctie is de kans op een conflict $1/m$. We kunnen een ideale random hashfunctie construeren door de tabelindex voor elke mogelijke sleutel random te kiezen. Deze functie is echter niet praktisch, niet alleen omdat een randomgenerator k gehele getallen moet produceren, maar ook omdat al die getallen moeten opgeslagen worden. (Als er toch plaats is voor zo'n grote tabel, is een hashtabel niet eens nodig.) Universele hashing slaagt erin om telkens snel een goede hashfunctie te produceren, die bovendien weinig plaats inneemt.

Deze hashfunctie wordt random gekozen uit een zorgvuldig ontworpen familie van functies. Een dergelijke familie heeft de eigenschap dat een random gekozen functie ideaal is. Of concreter, de kans dat een willekeurig gekozen functie uit de familie voor twee verschillende sleutels dezelfde tabelindex oplevert is hoogstens $1/m$.

Er zijn verschillende mogelijkheden om een dergelijke familie op te stellen. Dat gaat bijvoorbeeld als volgt. Stel dat er k mogelijke sleutels zijn. Kies dan een priemgetal p, zodat $p \geq k$. Voor de eenvoud zullen we onderstellen dat de sleutels gehele getallen zijn. Met a een geheel getal tussen 1 en $p-1$ (inbegrepen), b een geheel getal tussen 0 en $p-1$ (inbegrepen), en s een gehele sleutel, definiëren we een hashfunctie van de familie als

$$h_{a,b}(s) = ((as + b) \bmod p) \bmod m$$

Deze familie bevat dus $(p-1)p$ hashfuncties. Een random hashfunctie kiezen komt dan neer op het genereren (en opslaan) van twee random getallen a en b. Ook de evaluatie van de hashfunctie is eenvoudig en dus snel. De (eenmalige) keuze van p hoeft evenmin lastig te zijn, aangezien er steeds een priemgetal ligt tussen k en $2k$, voor elke mogelijke k. In dat geval is $p = O(k)$, zodat de hashfunctie opslaan slechts $2 \lg p = O(\lg k)$ bits vereist.

Aantonen dat deze familie de vereiste eigenschap heeft valt buiten het opzet van deze cursus.

HOOFDSTUK 11
BINAIRE ZOEKBOMEN

11.1 DEFINITIE

Stel dat we gegevens wensen op te slaan, waarvan de sleutels *geordend* kunnen worden. Naast de woordenboekoperaties kunnen we dan ook geïnteresseerd zijn in de sleutelvolgorde, met operaties zoals zoeken naar het eerste of laatste gegeven, zoeken naar de voorloper of opvolger van een gegeven, of alle gegevens opvragen in gerangschikte volgorde. Wat is dan de beste gegevensstructuur om al deze operaties efficiënt te ondersteunen?

Hashtabellen zijn (gemiddeld) zeer efficiënt voor de woordenboekoperaties, maar ongeschikt voor rangschikken van de opgeslagen gegevens, of andere operaties in verband met hun volgorde.

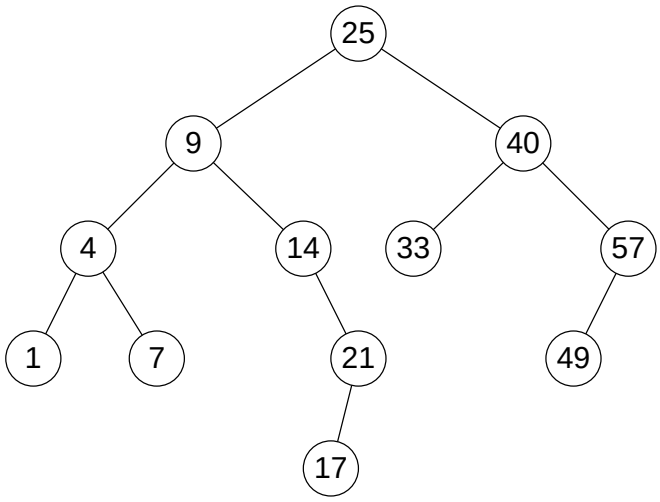
Als we de gegevens gerangschikt in een tabel opslaan, dan zijn alle operaties in verband met volgorde zeer eenvoudig en snel. Ook zoeken is zeer efficiënt via binair zoeken. Toevoegen of verwijderen is echter traag, omdat gemiddeld de helft van de tabel moet opgeschoven worden.

Elke iteratie van binair zoeken kiest de linker- of de rechterkant van het nog resterende deel van de tabel. Een schematische voorstelling van al deze keuzes heeft de vorm van een binaire boom, waarin het algoritme impliciet een weg volgt van boven naar beneden. Expliciet gebruik maken van een dergelijke boom leidt tot een van de belangrijkste gegevensstructuren: een binaire zoekboom. Die blijkt niet alleen efficiënt als woordenboek, maar ook zeer geschikt voor het bijhouden van de volgorde van de opgeslagen gegevens.

Een binaire zoekboom is een binaire boom, waarvan elke knoop een sleutel met bijbehorende informatie bevat. De sleutels in de knopen voldoen aan de volgende eigenschap:

> Alle sleutels in de linkse deelboom van elke knoop zijn kleiner dan (of even groot als) de sleutel van de knoop, en alle sleutels in de rechtse deelboom zijn groter dan (of even groot als) deze sleutel.

Een binaire zoekboom is dus recursief opgebouwd uit binaire zoekbomen, zoals te zien in figuur 11.1. Voor omvangrijke bijbehorende informatie is het beter de knoop ernaar



Figuur 11.1. Binaire zoekboom.

te laten verwijzen. Sommige operaties vereisen dat elke knoop ook een ouderwijzer bevat.

Wanneer men een binaire zoekboom in *inorder* overloopt verkrijgt men de sleutels van de elementen in gerangschikte volgorde. Aangezien een recursieve inorder-procedure in elke knoop tweemaal opgeroepen wordt, is rangschikken van n gegevens dus $\Theta(n)$. Dat is zeer snel voor rangschikken door vergelijken van willekeurige sleutels, maar houdt natuurlijk geen rekening met de constructie van de boom.

11.2 ZOEKEN

11.2.1 Zoeken naar een sleutel

Voorlopig zullen we voor de eenvoud onderstellen dat alle sleutels *verschillend* zijn. Door de recursieve structuur van de zoekboom, is het eenvoudig om het zoekproces recursief te formuleren (en te implementeren). Eerst wordt de zoeksleutel vergeleken met de sleutel van de wortel. Als deze sleutels verschillen, zoekt men verder in de linkse of rechtse deelboom, naar gelang dat de sleutel kleiner of groter is dan die bij de wortel. In deze deelboom herhaalt zich hetzelfde proces, tot ofwel de sleutel gevonden wordt, ofwel een ledige deelboom, wat betekent dat de sleutel niet in de zoekboom aanwezig is.

De geteste knopen vormen een weg vanuit de wortel in de richting van een blad, en de langst mogelijke weg komt overeen met de hoogte h van de boom. Zoeken is dus $O(h)$. Bemerk dat we steeds dalen, zodat ouderwijzers hiervoor niet nodig zijn.

11.2.2 Zoeken op volgorde

Deze operaties hebben uiteraard enkel zin op een niet-ledige boom.

11.2.2.1 Zoeken naar minimum en maximum

Ook dit zoekproces is eenvoudig recursief te formuleren. Ofwel bevindt het kleinste element van een niet-ledige zoekboom zich in de linkse deelboom van de wortel, ofwel, als die deelboom ledig is, bij de wortel zelf. Dit komt dus neer op het volgen van linkse kindwijzers vanuit de wortel, tot bij een knoop zonder linkerkind. Die bevat dan de gezochte sleutel. Het maximum zoeken is natuurlijk analoog.

Beide operaties volgen een weg vanuit de wortel naar beneden, en de lengte van die weg is niet groter dan de hoogte van de boom. Ze zijn dus $O(h)$. Aangezien we dalen, zijn ouderwijzers opnieuw niet nodig.

11.2.2.2 Zoeken naar opvolger en voorloper

Gegeven een knoop (de boom is dus niet ledig), zoek de opvolger van de sleutel in de knoop, als die bestaat. We kunnen twee gevallen onderscheiden:

- *De rechtse deelboom van de gegeven knoop is niet ledig.* Het overlopen van de boom in inorder rangschikt de sleutels, en daarbij komen de sleutels van de rechtse deelboom onmiddellijk na die van de knoop. De kleinste sleutel in deze deelboom is dan de opvolger van de knoop. Het probleem herleidt zich dus tot het vorige.
- *De rechtse deelboom is ledig.* Dat betekent dat de knoop het maximum bevat van een of meerdere deelbomen. De opvolgerknoop moet de grootste van deze deelbomen als linkerkind hebben.

  Om die te vinden moeten we dus ouderwijzers volgen, tot de eerste die rechts 'afslaat'. Bestaat die niet, dan is de sleutel van de opgegeven knoop de grootste in de boom. (Tot waar moeten we zoeken om dit te kunnen besluiten?)

De voorloper van een knoop wordt uiteraard op een analoge manier gevonden. Hier gebruiken we voor het eerst ouderwijzers. Vanuit de knoop volgen we ofwel een weg naar beneden (minimum), ofwel een weg naar boven, hoogstens tot bij de wortel van de boom. De maximale lengte van die wegen is opnieuw de hoogte van de boom. Beide operaties zijn dus ook $O(h)$.

De sleutels in een binaire zoekboom kunnen nu op een tweede manier gerangschikt worden, door bijvoorbeeld het minimum te zoeken, en dan al zijn opvolgers. (Is de performantie dan ook $\Theta(n)$, zoals bij overlopen in inorder?)

11.3 TOEVOEGEN

Toevoegen van een knoop moet niet alleen een nieuwe binaire boom opleveren, maar ook de basiseigenschap van de sleutels van een binaire zoekboom intact houden. Stel voorlopig dat we enkel toevoegen als de nieuwe sleutel zich nog niet in de boom bevindt. We moeten de sleutel opslaan op een plaats waar hij door een zoekoperatie kan gevonden worden. We zoeken dus naar de sleutel tot we bij een ledige deelboom terechtkomen, waarin de sleutel zich zou moeten bevinden. Deze ledige boom vervangen we dan door een nieuwe knoop zonder kinderen, waarin de sleutel en zijn bijbehorende informatie worden opgeslagen. Aangezien toevoegen eigenlijk zoeken is, gevolgd door een constant aantal bewerkingen, hebben ze (asymptotisch) dezelfde efficiëntie: $O(h)$.

Bekijken we nu even het probleem van *even grote* sleutels. Om zoeken eenvoudig te houden, kan men afspreken om gelijke sleutels bijvoorbeeld steeds in de rechtse deelboom op te slaan. Maar dat kan de hoogte van de boom onnodig groot maken, met alle gevolgen voor de performantie van de meeste operaties. (Wat gebeurt er bijvoorbeeld als we n gelijke sleutels aan een ledige boom toevoegen?) Om dit te vermijden zijn er een aantal mogelijkheden:

- Voorzie slechts één knoop voor alle duplicaten, die hun gemeenschappelijke sleutel bevat, en een *gelinkte lijst* met al hun bijbehorende informatie. Dit is wellicht de eenvoudigste oplossing, die tevens de hoogte van de boom minimaal houdt. Als er weinig duplicaten zijn, nemen de extra wijzers in elke knoop wel relatief veel plaats in beslag.
- Voorzie een *logische waarde* in elke knoop (in principe één bit, meestal één byte). Als tijdens het toevoegen de nieuwe sleutel even groot is als die van de knoop, dan voegen we hem toe aan de linkse of aan de rechtse deelboom, naargelang de logische waarde bij de knoop. Bovendien wordt die waarde geïnverteerd, zodat de volgende keer de andere deelboom gekozen wordt. Dit zorgt voor een meer evenwichtige boom.
- Ook hier gebruikt men een logische waarde, zoals hierboven. Deze wordt echter niet opgeslagen, maar telkens gegenereerd door een *randomgenerator*. Als die goed is, wordt gemiddeld de helft van de sleutels even groot aan die van de knoop links ervan opgelagen, de andere helft rechts.

11.4 PERFORMANTIE VAN ZOEKEN EN TOEVOEGEN

De lengte van de afgelegde weg bij zoeken en toevoegen is nooit groter dan de hoogte van de boom. Maar in het slechtste geval is deze hoogte $n-1$, zodat de performantie van zoeken en toevoegen dan $O(n)$ wordt.

Wat is de performantie in het gemiddeld geval? Daarbij onderstelt men dat de boom 'random' werd opgebouwd, met andere woorden, dat elke volgorde waarin de sleutels aan de boom kunnen toegevoegd worden even waarschijnlijk is. Dan kan men aantonen dat de gemiddelde hoogte van de boom $O(\lg n)$ is, maar dat is vrij lastig.[1] De tijd voor elke zoek- en toevoegoperatie wordt bepaald door de *diepte* van de gezochte of toegevoegde knoop, en de gemiddelde diepte van een knoop bepalen is een stuk eenvoudiger. (Maar een zwakker resultaat. Waarom?)

Om deze gemiddelde diepte te vinden, bepalen we de gemiddelde som van de diepten van alle knopen. Deze som noemt men de *inwendige weglengte*, en de gemiddelde inwendige weglengte voor een verzameling binaire zoekbomen met n knopen noteren we als $D(n)$. (Bemerk dat $D(1)$ nul is. Ook $D(0)$ onderstellen we nul.)

Aangezien alle mogelijke toevoegvolgordes even waarschijnlijk zijn, kan elke sleutel met dezelfde waarschijnlijkheid wortel zijn. Alle kleinere sleutels komen dan in de linkse deelboom terecht, en elk van hun toevoegvolgordes is ook even waarschijnlijk. (Idem voor de grotere sleutels in de rechtse deelboom.) Als bijvoorbeeld de i-de sleutel (in inorder volgorde) wortel is, dan bevat de linkse deelboom $i-1$ sleutels, en zijn gemiddelde inwendige weglengte is $D(i-1)$. Met die wortel is de gemiddelde inwendige weglengte van de volledige boom dus $D(i-1)+D(n-i)+n-1$, omdat de inwendige weglengte van de deelbomen ten opzichte van hun wortel gedefinieerd is. De gemiddelde inwendige weglengte voor de volledige boom wordt dan

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

Nu komt elke term in die som tweemaal voor. We krijgen dan

$$D(n) = n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} D(i)$$

Om deze full historybetrekking op te lossen vermenigvuldigen we eerst met n

$$nD(n) = n^2 - n + 2\sum_{i=0}^{n-1} D(i)$$

en werken dan de som weg door er dezelfde uitdrukking voor $n-1$ van af te trekken:

$$nD(n) - (n-1)D(n-1) = n^2 - n - (n-1)^2 + (n-1) + 2D(n-1)$$

[1] Ter vergelijking, de gemiddelde hoogte van een (gewone) binaire boom met n knopen is $O(\sqrt{n})$.

We bekomen dan

$$nD(n) = (n+1)D(n-1) + 2n - 2 < (n+1)D(n-1) + 2n.$$

Delen door $n(n+1)$ geeft

$$\frac{D(n)}{n+1} = \frac{D(n-1)}{n} + \frac{2}{n+1}$$

Stellen we $f(n) = D(n-1)/n$ dan geeft 2.4 op pagina 13 dat $f(n)$ $O(\lg n)$ is en dus dat $D(n) = O(n \lg n)$.

Als elke sleutel met dezelfde waarschijnlijkheid gezocht wordt, dan is de gemiddelde lengte van de zoekweg (en dus eventueel ook de toevoegweg) $O(\lg n)$.

Deze resultaten voor zowel de gemiddelde hoogte als de gemiddelde inwendige weglengte gelden echter alleen wanneer er geen elementen verwijderd worden. Er is weinig bekend over de gemiddelde eigenschappen van de boom wanneer men toevoegen afwisselt met verwijderen.

11.5 VERWIJDEREN

Als er niet veel sleutels moeten verwijderd worden, dan is de eenvoudigste oplossing opnieuw 'lazy deletion', zoals bij hashtabellen, omdat dit de structuur van de boom niet wijzigt. Lazy deletion is echter alleen bruikbaar in uitzonderlijke gevallen. Verwijderen is eenvoudig (gewoon een logische variabele omschakelen), maar zoeken en toevoegen moeten er rekening mee houden. Wanneer er veel sleutels verwijderd worden, moeten hun knopen effectief uit de boom verdwijnen. De vorm van een binaire zoekboom wordt bepaald door de opgeslagen sleutels. Het hoeft ons dan ook niet te verwonderen dat het verwijderen van een sleutel soms herstructurering zal vereisen, en dus ingewikkelder uitvalt dan de overige operaties. We zullen dit fenomeen nog vaker tegenkomen.

De te verwijderen knoop moet uiteraard eerst gezocht worden. Daarna zijn er drie mogelijke gevallen, naargelang het aantal kinderen van de knoop:

- *Geen kinderen.* Deze knoop kan zonder meer verwijderd worden. Dat vereist natuurlijk aanpassen van zijn ouder, als die bestaat.
- *Eén kind.* Dan wordt zijn ouder (als die bestaat) de nieuwe ouder van dat kind. Het wordt een linkerkind als te schrappen knoop een linkerkind was, en vice versa. Als de verwijderde knoop echter de wortel is, wordt het kind de nieuwe wortel.
- *Twee kinderen.* Dat is wat ingewikkelder, omdat er slechts één wijzer naar die knoop vrijkomt, terwijl er twee kinderen ouderloos worden. Omdat echter de

rechtse deelboom van de knoop niet ledig is, bevat hij de opvolger van de knoop, die bovendien geen links kind heeft. We kunnen dus deze opvolger uit de deelboom verwijderen (geval één of twee), en zijn gegevens (sleutel met bijbehorende informatie) kopiëren naar de te verwijderen knoop. Deze laatste wordt dus niet fysisch verwijderd (zijn drie wijzers blijven ongewijzigd), enkel zijn gegevens worden overschreven.

De basiseigenschap van binaire zoekbomen blijft daarbij intact, omdat de sleutel van de opvolger zeker groter is dan alle sleutels in zijn nieuwe linkse deelboom, en ook zeker kleiner dan alle sleutels uit zijn nieuwe rechtse deelboom, waarvan hij het minimum was. Ook ten opzichte van zijn nieuwe ouder is alles in orde, omdat alle wijzigingen zich in een deelboom van die ouder hebben voltrokken.

Als de gegevens in een knoop omvangrijk zijn kan kopiëren te traag uitvallen. Men kan dan de opvolger uit de deelboom verwijderen, en op de plaats van de te verwijderen knoop zetten (wijzers aanpassen), die dan uit de boom verdwijnt.

In plaats van de opvolger kan men ook de voorloper verwijderen. Steeds eenzijdig verwijderen kan de boom behoorlijk onevenwichtig maken, zodat men soms afwisselend verwijdert.

De twee eerste gevallen vereisen slechts een constant aantal operaties, terwijl het derde geval eerst nog een opvolger moet zoeken. Verwijderen is dus ook $O(h)$.

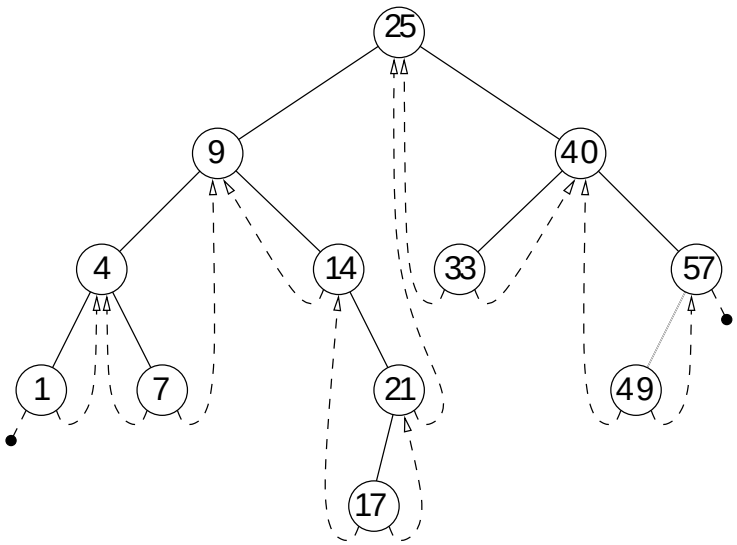
11.6 THREADED TREE

Een binaire boom met n knopen heeft steeds $n + 1$ nullwijzers. Die kunnen nuttiger gebruikt worden, door ze naar andere knopen in de boom te laten verwijzen. Natuurlijk moet men dan een onderscheid kunnen maken tussen een gewone wijzer naar een kind of deze speciale wijzers. Dat kan in principe met twee bits per knoop.

Een ‘threaded tree’ (Perlis en Thornton, 1960) is een binaire zoekboom waarbij een voormalige rechtse nullwijzer naar de *opvolger* (in inorder) van de knoop wijst, en een voormalige linkse nullwijzer naar de *voorloper*, zoals in figuur 11.2. Deze wijzers (‘threads’) verwijzen steeds naar hoger gelegen knopen in de boom. (Waarom?)

Het is duidelijk dat daarmee de opvolger (voorloper) van elke knoop kan gevonden worden, want ofwel is die het minimum van de rechtse deelboom (het maximum van de linkse deelboom), ofwel wordt er rechtstreeks naar verwezen (bij een ledige deelboom). Ouderwijzers die voornamelijk hiervoor gebruikt werden, zijn dus overbodig. Een dergelijke boom gebruikt niet alleen de talrijke nullwijzers, maar spaart ook nog een wijzer per knoop uit (in ruil voor twee extra bits).

Het groot voordeel van threaded trees is dat er geen stapel meer nodig is om de boom in inorder te overlopen. (Die is ook niet nodig als er ouderwijzers zijn.) Dat spaart geheugen (en vermijdt problemen met een te grote stapel) en tijd (stapeloperaties). In



Figuur 11.2. Threaded tree.

gewone bomen zonder ouderwijzers kunnen we immers nooit vanuit een knoop bij hoger gelegen knopen terechtkomen, zonder de weg bij te houden die we afgelegd hebben om die knoop te bereiken. Bij overlopen in inorder houdt de stapel precies die weg bij. Via een thread kan men echter ook een opvolger of voorloper vinden, zonder die weg te moeten doorlopen.

Het voordeel van threaded trees zou natuurlijk verdwijnen mocht het invullen en aanpassen van de threads niet efficiënt kunnen gebeuren. Gelukkig is dat niet het geval en zijn toevoegen en verwijderen bijna even gemakkelijk als bij gewone zoekbomen. Bij *toevoegen* van een niet-wortel wordt de nieuwe knoop aan een oude knoop gehangen, bijvoorbeeld als een linkerkind (rechterkind is analoog), waarbij we twee threads moeten invullen. De opvolger van de nieuwe knoop is de ouderknoop en die kennen we, de voorloper van de nieuwe knoop was de voorloper van de ouderknoop, en zijn adres vinden we in de linkerkindpointer van de ouder. Verwijderen van een knoop zonder kinderen is het omgekeerde (een van de threads wordt gekopieerd naar de ouder). Bij één kind is het, zoals te verwachten, iets moeilijker. Neem aan dat dit een rechterkind is. Dan heeft de opvolger van de te verwijderen knoop een linkerthread naar de te verwijderen knoop. Die moet vervangen worden door de linkerthread van de te verwijderen knoop voor alles in orde is (waarom?). De som van de twee zoekoperaties (eerst naar de te verwijderen knoop, dan naar zijn opvolger) vereist $O(h)$ operaties, waarbij h de diepte is van de opvolger.

Veel toepassingen op binaire zoekbomen vereisen enkel de opvolgers van een knoop, nooit de voorlopers. In de praktijk vindt men dan ook vaker ‘right-threaded trees’, waarbij een ledige rechtse deelboom door een thread wordt voorgesteld, en een ledige

linkse deelboom door een nullwijzer. En natuurlijk bestaan er analoge 'left-threaded trees'.

11.7 HASHTABEL OF BINAIRE ZOEKBOOM?

Een vergelijking tussen een hashtabel en een binaire zoekboom dringt zich op:

- *Operaties.* Een hashtabel ondersteunt enkel de woordenboekoperaties, en heeft dus een meer eenvoudige structuur. Bovendien vereist ze niet dat er een ordening op de sleutels gedefinieerd is. Een binaire zoekboom laat ook allerlei operaties toe waarbij de volgorde van de sleutels een rol speelt. Men kan zelfs naar een niet volledig gespecificeerde sleutel zoeken, wat bij hashtabellen onmogelijk is, omdat de hashfunctie de volledige sleutel nodig heeft. Zo kan men zoeken naar de kleinste (grootste) sleutel groter (kleiner) dan een gegeven sleutel, of naar alle sleutels gelegen tussen twee opgegeven sleutels (een 'range search').
- *Performantie.* Bij een hashtabel zijn de woordenboekoperaties gemiddeld $O(1)$, onafhankelijk van het aantal opgeslagen sleutels. De operaties op een binaire zoekboom zijn gemiddeld $O(\lg n)$. In het slechtste geval zijn de operaties van beide gegevensstructuren $O(n)$, maar enkel bij binaire zoekbomen kan men ervoor zorgen dat ze ook in het slechtste geval $O(\lg n)$ worden[2].
- *Geheugenvereisten.* Een hashtabel is een statische structuur, die bovendien opeenvolgende geheugenplaatsen vereist. Men moet dus op voorhand haar grootte kunnen schatten, want rehashing is duur, en niet altijd mogelijk. Een binaire zoekboom is een dynamische structuur, waarbij het geheugen meer gefragmenteerd mag zijn. Het is dus niet nodig dat zijn grootte op voorhand bekend is.

[2] Zie Algoritmen II.

DEEL 4

GRAAFALGORITMEN I

INLEIDING

Heel veel problemen, wanneer men ze tot hun essentie herleidt, kunnen voorgesteld worden door *objecten* waartussen *verbindingen* (relaties) bestaan. Enkele voorbeelden:

- Een overzicht van de luchtvaartverbindingen tussen de Europese steden (objecten), voorzien van reistijden en prijzen. Die kan gebruikt worden om vragen te beantwoorden als 'Hoe vliegt men het snelst van Manchester naar Bremen?', of 'Hoeveel kost de goedkoopste vlucht tussen Praag en Marseille?'.
- De objecten zijn vakken gedoceerd aan een onderwijsinstelling. Twee vakken zijn onderling verbonden als ze door dezelfde docent onderwezen worden, of als een student beide wil volgen. Gevraagd een uurrooster op te stellen dat met deze beperkingen rekening houdt.
- Een groot project kan onderverdeeld worden in een aantal deeltaken, met geschatte duur. Deze taken worden dan de objecten, en een (gerichte) verbinding van taak a naar taak b beduidt dat a moet afgewerkt zijn vooraleer b kan aangevat worden. De vraag kan zijn: 'Wat is de kortste tijd nodig om alles af te werken?'. Of ook nog: 'Welke taken mogen zeker geen vertraging oplopen, zonder het hele project te vertragen?'.

Een graaf is een abstract model voor dergelijke problemen. Graafalgoritmen zijn dan ook cruciaal bij het oplossen van een grote verscheidenheid aan belangrijke en vaak moeilijke praktische problemen.

Het is niet altijd eenvoudig om in een concreet probleem een onderliggende graaf te herkennen. Dat vereist immers het weglaten van details, om het probleem tot zijn essentie te herleiden (abstractie). Gezien het praktisch belang van grafen, zijn veel graafproblemen uitgebreid onderzocht, en behoort een nieuw probleem meestal tot een bekende categorie. Voor veel graafproblemen zijn er echter geen efficiënte algoritmen bekend (en bijna zeker ook niet te vinden), zodat men zijn toevlucht moet nemen tot benaderende algoritmen of heuristische methodes.

Deze beperkte inleiding bespreekt enkele belangrijke basisalgoritmen op grafen, die vaak als onderdeel fungeren in de oplossing van meer ingewikkelde problemen. Ze zijn allemaal efficiënt. Meer gevorderde graafalgoritmen worden behandeld in de cursus Gevorderde Algoritmen..

HOOFDSTUK 12
MINIMALE OVERSPANNENDE BOMEN

12.1 DEFINITIE EN CONSTRUCTIE-EIGENSCHAP

Een overspannende boom ('spanning tree') van een ongerichte (samenhangende) graaf is een (wortelloze) boom met dezelfde knopen en met een deel van de verbindingen als takken. Zowel diepte-eerst als breedte-eerst zoeken genereren overspannende bomen, en het is duidelijk dat een graaf meerdere overspannende bomen kan hebben. Bij een gewogen graaf definieert men het gewicht van een overspannende boom als de som van de gewichten van zijn takken, en een minimale overspannende boom (MOB) heeft dan het kleinste gewicht van alle dergelijke bomen.[1] Ook een MOB is niet noodzakelijk uniek. (Wanneer wel?) Figuur 12.1 toont een graaf met een van zijn minimale overspannende bomen.

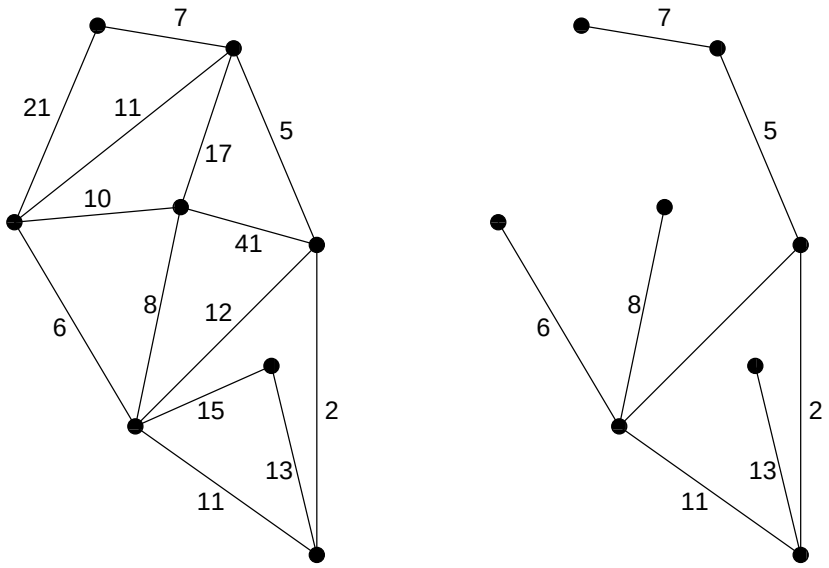
Minimale overspannende bomen hebben nogal wat toepassingen (directe en indirecte), en de algoritmen om ze op te stellen behoren tot de oudste graafalgoritmen.

Alle efficiënte algoritmen construeren een MOB tak per tak. Daarbij maken ze telkens een lokaal optimale keuze. Het eindresultaat is echter een globaal optimum. Dat is niet vanzelfsprekend: voor veel problemen vindt deze werkwijze enkel een suboptimale oplossing. Algoritmen die (globaal) optimale oplossingen vinden, door steeds lokaal optimale keuzes te maken (zonder ooit op hun stappen te moeten terugkeren), noemt men 'inhalig' (*greedy*).

De reeds gevonden takken vormen een deelverzameling van de graafverbindingen, die men telkens met een nieuwe verbinding uitbreidt, tot een MOB bekomen wordt. (Deze verzameling moet niet noodzakelijk een deelboom van de uiteindelijke MOB vormen.) Om die nieuwe verbinding te kiezen, kunnen we de volgende algemene eigenschap gebruiken (Tarjan, 1983), die gebruikt maakt van het begrip *snede*. Een snede S van een samenhangende graaf is een verzameling verbindingen die de knopenverzameling in twee niet-ledige stukken A en B verdeelt. Dit houdt in dat een verbinding (i, j) in S zit als en slechts als $i \in A$ en $j \in B$ of, omgekeerd, $j \in A$ en $i \in B$.

> Gegeven een deelverzameling D van graafverbindingen die behoren tot een MOB, en een graafverbinding v die niet tot D behoort. Opdat v samen met de verbindingen van D tot een MOB zou behoren, is nodig en

[1] Correcter zou '*lichtste* overspannende boom' zijn. Ook in de Engelse term 'minimal spanning tree' laat men 'weight' gewoonlijk weg.



Figuur 12.1. Ongerichte gewogen graaf en een van zijn minimale overspannende bomen.

voldoende dat er een snede S bestaat, die geen verbindingen van D bevat, en waarin v een lichtste verbinding is.

Ze aantonen is vrij eenvoudig:

- Is de eigenschap nodig? Als we onderstellen dat M een MOB is die de verbindingen van D en ook v bevat, bestaat er dan een snede S waarin v lichtst is?

  Wanneer we v uit M verwijderen krijgen we twee deelbomen, die de graafknopen in twee deelverzamelingen verdelen. De snede S die alle graafverbindingen tussen die twee deelverzamelingen bevat, heeft geen verbinding gemeen met D. Stel nu dat S een verbinding w bevat, lichter dan v. Als we v wegnemen uit M en vervangen door w dan krijgen we opnieuw een overspannende boom B, die bovendien lichter is dan M. Maar dan zou M geen MOB zijn, wat we onderstelden.

- Is de eigenschap voldoende? Onderstel dat de verbindingen van D tot een MOB behoren. Onderstel bovendien dat er een snede S bestaat, die geen verbindingen van D bevat, en waarin v een lichtste verbinding is. Behoren v en de verbindingen van D dan samen tot een MOB?

  Stel dat dit niet het geval is, dat de verbindingen van D samen met v tot geen enkele MOB behoren. Neem dan een MOB die D omvat. Als we v aan M

toevoegen ontstaat er een lus. Verbinding v verbindt knopen uit de twee deelverzamelingen van snede S. Dat geldt ook voor minstens een van de andere takken in die lus. Dus behoort die tak w ook tot S, en daarom niet tot D. Als we dan w uit M verwijderen, krijgen we een nieuwe overspannende boom B. Nu is B zeker niet zwaarder dan M, omdat v een lichtste verbinding van S is, en dus niet zwaarder dan w. Kortom, ook B is een MOB, die zowel de verbindingen van D als v bevat.

De algoritmen om een MOB te vinden verschillen in de manier waarop ze sneden kiezen die nieuwe MOB-verbindingen opleveren. De meest gebruikte zijn die van Prim en Kruskal.

12.2 HET ALGORITME VAN PRIM

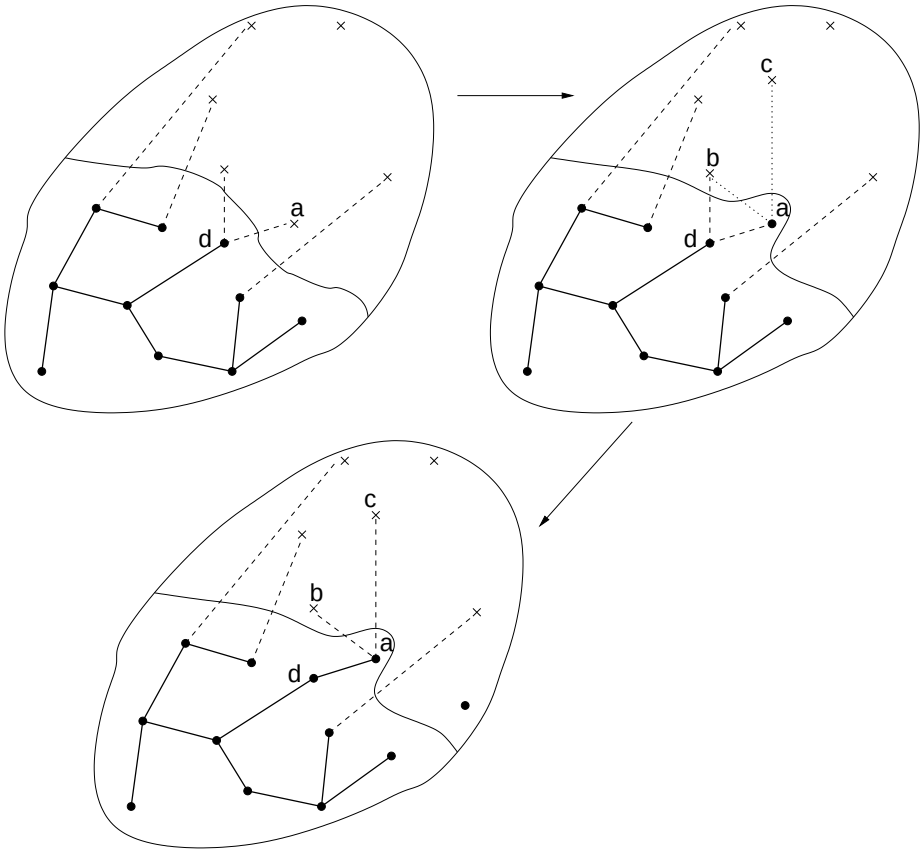
De verbindingen van D vormen hier een deelboom van de uiteindelijke MOB. We vertrekken van een willekeurig gekozen knoop; in het begin heeft D geen verbindingen. De snede S bestaat uit alle verbindingen tussen de knopen van D, en de rest van de graafknopen. S bevat dus geen verbindingen van D. De deelboom wordt telkens uitgebreid met een lichtste verbinding uit S.

In principe zouden we alle snedeverbindingen moeten onderzoeken, maar als een resterende knoop meerdere verbindingen met de MOB heeft, komt toch enkel de lichtste in aanmerking voor selectie. Met andere woorden, voor elke knoop die nog niet tot de MOB behoort, houden we de lichtste verbinding bij die hem verbindt met de MOB. (Resterende knopen die nog niet met de MOB verbonden zijn spelen voorlopig niet mee.)

Telkens wordt een lichtste verbinding en de bijbehorende nieuwe knoop in de MOB opgenomen. Buren van de nieuwe knoop die nog niet met de MOB verbonden waren, krijgen hun eerste (en dus voorlopig lichtste) verbinding. Buren die reeds met de MOB verbonden waren, kunnen een potentieel lichtere verbinding met de boom krijgen via de nieuwe knoop. Buren die reeds tot de MOB behoren mogen genegeerd worden, want hun verbinding behoort zeker niet tot de nieuwe snede. Heel dit proces wordt herhaald, tot de MOB alle knopen bevat. (De initiële MOB bevat één willekeurige knoop.)

Figuur 12.2 toont schematisch enkele stappen uit het algoritme. Onder de scheidingslijn bevindt zich de reeds gedeeltelijk geconstrueerde MOB, erboven liggen de resterende knopen. Sommigen daarvan zijn reeds verbonden met de MOB (verbindingen in streeplijn). De knoop a met de lichtste van die verbindingen wordt nu gekozen, zijn verbinding met knoop d van de MOB wordt definitief, en de scheidingslijn breidt uit. We onderzoeken dan de buren van a (verbindingen in stippellijn). Buur c wordt voor de eerste keer met de MOB verbonden, maar buur b is ook buur van d en had dus reeds een

verbinding met de MOB. Nu wordt enkel de lichtste van deze verbindingen behouden, en als dat (a, b) is, blijft b alleen via a met de MOB verbonden.



Figuur 12.2. Het algoritme van Prim voor de minimale overspannende boom.

Voor de opslag van een collectie gegevens waaruit telkens het minimum moet verwijderd worden, denken we natuurlijk aan een *prioriteitswachtrij*, geïmplementeerd met een heap. Die gegevens zijn knopen buiten de MOB, maar ermee verbonden, en hun prioriteit is het gewicht van hun lichtste verbinding met de MOB. (Om de verbinding te specificeren is ook nog de andere eindknoop in de MOB vereist. Deze voorlopige ouders kunnen in een aparte tabel bijgehouden worden.)

Het gewicht van een knoopverbinding moet echter ook efficiënt kunnen aangepast worden. Een gewone heap ondersteunt deze operatie niet. Men kan dat als volgt oplossen:

- De prioriteit van een element efficiënt aanpassen is slechts mogelijk als we zijn *plaats* in de heap kennen, wat normaal niet het geval is. De heaptabel met grootte

$O(n)$ telkens sequentieel doorzoeken is te traag, zodat men voor alle graafknopen deze posities apart moet bijhouden en ze indien nodig aanpassen. (Bijvoorbeeld in een tabel, geïndexeerd met het knoopnummer.) Een positie vinden (en achteraf aanpassen) wordt dan $O(1)$, en wijzigen (verminderen) van de prioriteit $O(\lg n)$.

- Als alternatief kan men een knoop met zijn oude prioriteit in de prioriteitswachtrij laten zitten, en de knoop nogmaals toevoegen met zijn nieuwe (lagere) prioriteit. De prioriteitswachtrij kan dan meerdere exemplaren van dezelfde knoop bevatten. Bij verwijdering van het minimum zullen we nu moeten testen of de bijbehorende knoop niet reeds in de MOB zit, zodat een volgend minimum moet verwijderd worden. Deze oplossing is 'netter' vanuit softwarestandpunt, en is zeker eenvoudiger te implementeren, maar heeft als nadeel dat de grootte van de prioriteitswachtrij $O(m)$ wordt. Voor de performantie van de afzonderlijke heapoperaties is dat niet zo belangrijk, omdat $m = O(n^2)$ en dus $\lg m = O(\lg n)$. Maar er moeten nu $O(m)$ elementen verwijderd worden in plaats van $O(n)$. Bovendien kunnen de geheugenvereisten te groot uitvallen.

Bemerk dat we hier opnieuw drie soorten knopen terugvinden, zoals bij diepte-eerst en breedte-eerst zoeken: de knopen die reeds tot de MOB behoren (zwart), deze die verbonden zijn met de MOB (de 'rand') en die dus in de prioriteitswachtrij zitten (grijs), en de rest (wit). Oorspronkelijk zit enkel de (grijze) wortel in de prioriteitswachtrij, en zijn alle andere knopen wit. En telkens wordt er een knoop uit de rand geselecteerd en overgebracht naar de MOB, waarna de rand zo nodig aangepast wordt. Het selectiemechanisme is nu echter niet LIFO (stapel) of FIFO (wachtrij), maar kleinste prioriteitsgetal eerst. Voor de rest zijn deze algoritmen zeer gelijkaardig.

Hoe efficiënt is dit algoritme? De initialisatie van enkele tabellen met informatie over de knopen is $\Theta(n)$. De rest hangt af van de gekozen oplossing voor het aanpassen van de prioriteiten:

- Wanneer elke knoop slechts eenmaal toegevoegd aan en verwijderd wordt uit de prioriteitswachtrij, vereist dit in totaal $O(n \lg n)$. Alle buren van elke verwijderde knoop moeten overlopen worden, en in het slechtste geval kan de prioriteit van een knoop zoveel maal aangepast worden als hij buur is (zijn graad), wat in totaal een bovengrens van $O(m \lg n)$ oplevert. Deze versie van het algoritme heeft dus een performantie van $O((n + m) \lg n)$, en omdat de graaf samenhangend is, zodat $n = O(m)$, krijgen we tenslotte $O(m \lg n)$.

  De prioriteit van de knopen kan meermaals aangepast (verminderd) worden, terwijl elke knoop slechts eenmaal toegevoegd en verwijderd wordt. Een element toevoegen of zijn prioriteit verminderen in een dalende heap betekent eventueel stijgen in de heap, het minimum verwijderen betekent eventueel dalen. Door de kleinere hoogte van een d-heap ten opzichte van een binaire heap wordt stijgen $O(\log_d n)$ in plaats van $O(\lg n)$, terwijl dalen $O(d \log_d n)$ wordt in plaats van $O(\lg n)$. Een geschikte keuze van d kan dus de performantie wat verbeteren. Een

substantiële verbetering kan in principe bekomen worden door een 'Fibonacci-heap' te gebruiken, die speciaal voor dit soort toepassingen bedacht werd. Zijn geamortiseerde performantie is slechts $O(1)$ voor toevoegen of verminderen van de prioriteit, en $O(\lg n)$ voor verwijderen. Daarmee verbetert de performantie tot $O(m + n \lg n)$. Maar de ingewikkelde implementatie van deze heap zorgt voor te grote verborgen constanten om praktisch bruikbaar te zijn.

- Als we duplicaten in de prioriteitswachtrij toelaten, kan die $O(m)$ elementen bevatten. Een prioriteit aanpassen wordt immers vervangen door opnieuw toevoegen. Toevoegen en verwijderen vereist nu in totaal $O(m \lg m)$, maar met $m = O(n^2)$ wordt dat $O(m \lg n)$. Het eindresultaat is dus (asymptotisch) hetzelfde als bij de eerste versie.

Het oorspronkelijke algoritme van Prim (1957) gebruikte geen prioriteitswachtrij, maar een gewone tabel. Daarin sequentieel zoeken naar de lichtste verbinding werd gecombineerd met het aanpassen van de gewichten van de buren van de laatst geselecteerde knoop. De performantie was dan ook $\Theta(n^2)$. Dit heeft nu enkel nog zin voor dichte grafen, voorgesteld door een burenmatrix.

12.3 DISJUNCTE DEELVERZAMELINGEN

12.3.1 Inleiding en definities

Gegeven een verzameling van n elementen, waarop een *equivalentierelatie* gedefinieerd is.[2] Stel dat deze verzameling in deelverzamelingen van equivalente elementen moet onderverdeeld worden (de *equivalentieklassen*). De doorsnede van elk paar equivalentieklassen is uiteraard ledig, ze zijn dus allemaal disjunct. Bovendien behoort elk element tot een equivalentieklasse (het is minstens equivalent met zichzelf). Een equivalentierelatie definieert dus een *partitie* van de verzameling. (Het omgekeerde geldt ook: elke partitie definieert een equivalentierelatie.)

Mochten we deze equivalentierelatie op voorhand kennen, dan kon de definitieve onderverdeling meteen gebeuren. Maar we krijgen deze informatie slechts met mondjesmaat, onder de vorm van paren equivalente elementen, het een na het ander, en na elk paar moet de (voorlopige) verdeling opgesteld worden. Men wil immers op ieder ogenblik kunnen nagaan of twee elementen in dezelfde (voorlopige) equivalentieklasse zitten. De verdeling is dus *dynamisch* in plaats van statisch.

Oorspronkelijk kennen we enkel de elementen, en behoort elk element dus tot zijn eigen deelverzameling (reflexieve eigenschap). Als we twee equivalente elementen (a, b)

[2] Zie cursus 'Discrete wiskunde'.

opgegeven krijgen, die tot verschillende deelverzamelingen behoren, dan worden al de elementen van beide deelverzamelingen equivalent (transitieve eigenschap). Die horen dus in een nieuwe deelverzameling, en de twee originele deelverzamelingen moeten verdwijnen. Als echter a en b reeds equivalent zijn (en dus in dezelfde deelverzameling zitten), dan verandert er uiteraard niets.

Waarom wordt dit onderwerp behandeld in de context van grafen? Omdat men de equivalentierelatie ook kan voorstellen als een ongerichte graaf, met de elementen als knopen, en verbindingen tussen de opgegeven paren equivalente elementen. Door de transitieve eigenschap zijn twee elementen equivalent als er een weg tussen beide bestaat, met andere woorden als ze met elkaar verbonden zijn: de equivalentieklassen zijn dus de *samenhangende componenten* van de graaf. Als de graaf opgegeven zou zijn, dan volstond diepte-eerst zoeken om deze componenten efficiënt te bepalen. De verbindingen worden echter dynamisch aan de graaf toegevoegd, zodat we telkens opnieuw diepte-eerst zoeken zouden moeten uitvoeren, om tussentijdse find-operaties mogelijk te maken. De meer efficiënte oplossingen hieronder maken dan ook geen gebruik van deze graaf.

Om te testen of twee elementen tot dezelfde deelverzameling behoren, kiest men een willekeurig element uit elke deelverzameling als haar vertegenwoordiger. Een zogenaamde *find*-operatie[3] op een element vindt de vertegenwoordiger van de deelverzameling waartoe het behoort, zodat twee find-operaties volstaan om na te gaan of twee elementen equivalent zijn. Het samenvoegen van de elementen uit twee deelverzamelingen noemt men een *union*-operatie. Na elke union-operatie is er een deelverzameling minder, zodat er hoogstens $n - 1$ union-operaties kunnen gebeuren.

Er bestaan verschillende manieren om (deel)verzamelingen voor te stellen, maar deze particuliere toepassing heeft haar eigen efficiënte implementatie. De onderverdeling is hier immers niet statisch: deelverzamelingen groeien of verdwijnen.

We zullen zien dat de twee vereiste operaties, find en union, elk *afzonderlijk* $O(1)$ kunnen gemaakt worden (in het slechtste geval), maar men kan aantonen dat dit niet tegelijkertijd mogelijk is. Toch slaagt men erin om dit ideaal resultaat goed te benaderen. De performantie van een willekeurige *reeks* van m operaties op n elementen (initialisatie, find- en union-operaties) kan immers gereduceerd worden tot nagenoeg $O(m)$, in het slechtste geval. (Met dus een geamortiseerde efficiëntie van $O(1)$ per operatie.)

Disjuncte deelverzamelingen kunnen dan bijvoorbeeld gebruikt worden om efficiënt na te gaan of een zeer grote (ongerichte) graaf samenhangend is[4]. Elke verbinding wordt immers slechts eenmaal behandeld, zodat het niet nodig is om de verbindingen op te slaan: ze kunnen sequentieel uit een bestand gehaald worden.

Er zijn meer algoritmen die dynamische equivalentieklassen gebruiken (zoals het algo-

[3] We behouden hier de Engelse namen. Deze manier van dynamisch onderverdelen noemt men ook wel 'union-findalgoritmen'.

[4] Een eenvoudig voorbeeld van een graafalgoritme dat uitwendig geheugen gebruikt (*external memory graph algorithm*). Er is nogal wat belangstelling voor (zeer) grote grafen, en deze klasse van algoritmen.

ritme van Kruskal, zie later), en die daarbij beroep doen op union-find operaties om ze efficiënt te beheren.

12.3.2 Efficiënte find-operatie

De n elementen kunnen genummerd worden van 0 tot $n-1$ (eventueel met behulp van een hashtabel om hun echte namen op te slaan). Als we voor elk element de vertegenwoordiger van de deelverzameling waartoe het behoort in een tabel bijhouden, wordt de find-operatie $O(1)$. We trachten nu ook de union-operatie zo efficiënt mogelijk te maken.

Als de elementen i en j tot verschillende deelverzamelingen behoren, waarop de union-operatie moet uitgevoerd worden, dan moet de tabel aangepast worden voor elk element uit een van beide verzamelingen. Aangezien we die elementen niet kennen, moeten we heel de tabel overlopen, zodat de $n-1$ mogelijke union-operaties samen $\Theta(n^2)$ bewerkingen vereisen. (De initialisatie van de n verzamelingen zorgt nog voor een bijkomende $\Theta(n)$-term.) Dat is zeer goed als er een vergelijkbaar aantal find-operaties gebeurt, want gemiddeld over de totale reeks operaties (geamortiseerd) worden beide operaties dan $O(1)$. Met minder dan $\Omega(n^2)$ find-operaties is deze performantie echter onaanvaardbaar.

Om dat te verbeteren houden we ook nog de elementen van elke verzameling bij in een *gelinkte lijst*. Die lijsten komen in een tabel, geïndexeerd met de vertegenwoordiger van de deelverzamelingen. Samenvoegen vereist dan enkel de concatenatie van twee lijsten, en het aanpassen van de find-tabel voor de elementen uit één van die lijsten. Toch blijft een performantie van $O(n^2)$ mogelijk, bijvoorbeeld door een telkens langer wordende lijst stelselmatig bij een lijst met slechts één element te voegen. (Ga eens na.)

Men kan dit vermijden door zorgvuldiger te werk te gaan bij de union-operatie. De *union-by-size heuristiek* bijvoorbeeld voegt steeds de kortste lijst toe aan de langere. Daartoe houdt men de lengte van elke lijst bij, en het is triviaal om die aan te passen. Aangezien elk element nu hoogstens $\lceil \lg n \rceil$ keer van verzameling kan veranderen (want telkens wordt zijn verzameling minstens tweemaal zo groot), vereisen $n-1$ union-operaties samen nog slechts $O(n \lg n)$ bewerkingen. Een reeks van m operaties (initialisatie, find- en union-operaties) heeft dan in het slechtste geval een performantie van $O(m + n \lg n)$.

12.3.3 Efficiënte union-operatie

De union-operatie kan zeer efficiënt gemaakt worden door de deelverzamelingen voor te stellen als bomen van een *bos*. De wortel is de vertegenwoordiger van elke deelverzameling. Omdat elk element zijn vertegenwoordiger moet kunnen terugvinden, maar niet omgekeerd, verwijst elke boomknoop enkel naar zijn ouder. (Bemerk dat er geen verband bestaat tussen de takken van dit bos en de verbindingen van de hierboven

vermelde graaf die de equivalentierelatie voorstelt.)

De union-operatie is nu zeer eenvoudig: de wortel van de ene boom wordt een kind van de wortel van de andere. Dat vereist $O(1)$ indien beide wortels gekend zijn (na find-operaties).

We trachten nu ook de find-operatie zo efficiënt mogelijk te maken. De find-operatie komt neer op het volgen van ouderwijzers van bij een knoop tot aan de wortel, en de hiervoor vereiste tijd is natuurlijk afhankelijk van de lengte van die weg. We hebben er dus alle belang bij om de hoogte van de bomen zo klein mogelijk te houden.

Deze eerste versie is helemaal niet goed, want $n-1$ union-operaties kunnen resulteren in één boom die eigenlijk een gelinkte lijst is met hoogte (lengte) $n-1$. Een find-operatie wordt dan $O(n)$, en een reeks van m operaties (initialisatie, union- en find-operaties) heeft in het slechtste geval een performantie van $O(mn)$. (Men vermoedt dat dat gemiddeld ook het geval is - zoals vaak bij gemiddelden is de analyse zeer complex.)

De performantie kan echter fel verbeterd worden door twee heuristieken te gebruiken:

- De eerste is dezelfde *union-by-size heuristiek* als bij de efficiënte find-operatie met gelinkte lijsten, en houdt het aantal elementen per boom bij. Door telkens de boom met minder elementen samen te voegen met deze met meer elementen, beperken we de hoogte van de bomen tot $\lceil \lg n \rceil$. (Om dezelfde redenen als vroeger. Een element kan hoogstens $\lceil \lg n \rceil$ maal van boom veranderen, en telkens neemt zijn diepte met één toe, beginnend vanaf nul, als wortel van zijn eigen boom.) De find-operatie wordt dus $O(\lg n)$, en een reeks van m operaties vraagt in het slechtste geval $O(m \lg n)$ bewerkingen (de $O(n)$-termen kunnen immers verwaarloosd worden). Als er veel meer find- dan union-operaties zijn dan is dit slechter dan de beste versie van de efficiënte find.

  Men kan aantonen dat het gemiddeld geval $O(m)$ bewerkingen vereist, wat dan neerkomt op een gemiddelde geamortiseerde performantie van $O(1)$ per operatie.

- De find-operatie zou optimaal zijn mocht elke knoop rechtstreeks naar de wortel van zijn boom wijzen. Dat zou echter de union-operatie inefficiënt maken, omdat daarbij dan alle knopen van één van de bomen moeten aangepast worden. Toch kan men dit ideaal benaderen door bij elke find-operatie alle knopen op de gevolgde weg naar de wortel te laten wijzen. (Dat vereist natuurlijk dat die weg tweemaal doorlopen wordt.) Een volgende find-operatie op een van die knopen zal dus optimaal zijn. (En kan ook nog voor andere knopen verbeterd zijn.) Hoe meer find-operaties er gebeuren, des te kleiner wordt de hoogte van de bomen.

  Deze heuristiek heet *path compression*, en kan samen met de eerste toegepast worden, omdat hij het aantal elementen in de bomen niet wijzigt.

De performantieanalyse van deze union/find-operaties is uiterst moeilijk, en zelfs nog niet volledig doorgrond. Zo is het niet duidelijk of de tweede heuristiek de *gemiddelde*

performantie van de eerste verbetert. Wat men echter wel weet is dat de performantie in het *slechtste* geval sterk verbetert door beide heuristieken samen te gebruiken. Voor een reeks van m bewerkingen (gesteld dat $m \geq n$) is de performantie immers $\Theta(m\alpha(m,n))$, waarbij $\alpha(m,n) \leq 4$ voor alle praktische waarden van n en m (Tarjan, 1975). Als men beide heuristieken gebruikt blijkt de geamortiseerde performantie van een union/find-operatie dus nagenoeg $O(1)$, ook in het slechtste geval.

12.4 HET ALGORITME VAN KRUSKAL

Dit algoritme bouwt een MOB op uit afzonderlijke deelbomen, die geleidelijk met elkaar verbonden worden tot er maar één boom meer overblijft. De oorspronkelijke (triviale) deelbomen zijn de knopen van de graaf. De lichtste graafverbinding die twee van die deelbomen verbindt, is de lichtste verbinding uit de snede gevormd door de knopen van een van die deelbomen, en alle andere knopen. Deze snede bevat ook geen enkele tak van die deelbomen. Die verbinding mag dus toegevoegd worden, want ze behoort tot een MOB. Daarom worden de graafverbindingen volgens stijgend (niet-dalend) gewicht onderzocht, en enkel als ze twee deelbomen verbinden worden ze toegevoegd.

Hoe komen we snel te weten of de eindknopen van een verbinding tot verschillende deelbomen behoren? Elk stuk is een deelverzameling van de verzameling knopen, en de stukken hebben geen gemeenschappelijke knopen. Het zijn dus disjuncte deelverzamelingen, die samen de verzameling knopen opdelen (een partitie). De relatie 'behoort tot hetzelfde stuk als' is de equivalentierelatie tussen de knopen. Aangezien elke nieuwe verbinding twee equivalente knopen aanduidt, moeten we snel kunnen nagaan of ze reeds tot hetzelfde stuk behoren. Anders moeten we die stukken samenvoegen. Dat zijn dus union/find-operaties op disjuncte deelverzamelingen. (Bemerk dat de takken van de union-find bomen *niet* noodzakelijk de takken van de MOB zijn. Die boom moet dus apart bijgehouden worden.)

Om de verbindingen volgens stijgend gewicht te kunnen onderzoeken, ligt het voor de hand om ze vooraf te rangschikken. Meestal echter moeten niet alle verbindingen onderzocht worden, omdat we kunnen stoppen nadat er $n-1$ geschikte verbindingen toegevoegd werden. (Bedenk eens een voorbeeld waarbij men ze allemaal moet testen.) Alle verbindingen vooraf rangschikken is dus overbodig, en aangezien we telkens de resterende verbinding met het kleinste gewicht nodig hebben, ligt het gebruik van een prioriteitswachtrij voor de hand.

Wat is de efficiëntie van dit algoritme? De opbouw van de heap is $O(m)$. In het slechtste geval moet elke verbinding uit de heap gehaald worden, dus maximaal $O(m \lg m)$ operaties, en voor elke verbinding zijn er twee find-operaties en hoogstens één union-operatie (met in totaal natuurlijk niet meer dan $n-1$ union-operaties), die samen $O(m+n)$ vergen. Blijft nog het opslaan van de $n-1$ takken van de MOB, maar dat is zeker $O(n)$. In totaal krijgen we dan $O(n+m+m \lg m)$, en aangezien $n = O(m)$ (de

graaf was immers samenhangend), wordt dat tenslotte $O(m \lg m)$. Meestal echter zal er veel minder tijd nodig zijn, omdat niet alle verbindingen aan bod komen. Bemerk dat het algoritme van Prim dezelfde asymptotische performantie had.

De oorspronkelijke methode van Kruskal (1956) gebruikte geen prioriteitswachtrij voor de verbindingen maar rangschikte ze, en ook de test of een nieuwe verbinding geen lus veroorzaakt was eenvoudiger en minder efficiënt.

12.5 CLUSTERING

Stel dat men een gegeven collectie objecten (bijvoorbeeld foto's, dokumenten, organismen) wil onderverdelen in clusters van gelijkaardige objecten. Dan moet men eerst een criterium kiezen om deze gelijkenis te kwantificeren. Vaak definieert men daarvoor een *afstand* tussen twee objecten: hoe groter de afstand, des te minder gelijkenis. Die afstand is natuurlijk zelden letterlijk te nemen: voor organismen kan dat de tijd zijn sinds ze afsplitsten van een gemeenschappelijke voorouder in de evolutie, voor beelden het aantal overeenkomstige pixels die meer dan een grenswaarde verschillen. Het probleem van clustering wordt dan te zorgen dat objecten in dezelfde cluster 'dicht' bij elkaar liggen, en deze in verschillende clusters 'ver' van elkaar.

Deze vage omschrijving vormt het vertrekpunt voor verschillende methodes van clustering. Een van de eenvoudigste maakt gebruik van minimale overspannende bomen. Gegeven een verzameling van n objecten $S = \{p_1, p_2, \ldots, p_n\}$. Gevraagd die te verdelen in k clusters $C = \{C_1, C_2, \ldots, C_k\}$. En dit aan de hand van een afstandsfunctie $d()$, met de volgende eigenschappen: als p_i en p_j twee objecten zijn, dan moet $d(p_i, p_i) = 0$, $d(p_i, p_j) > 0$ als $i \neq j$, en $d(p_i, p_j) = d(p_j, p_i)$. Wanneer men de *spreiding* ('spacing') van een clustering definieert als de kleinste afstand tussen twee objecten uit verschillende clusters, dan ligt het voor de hand om de clustering te zoeken met de grootste spreiding. Het is niet evident dat een oplossing efficiënt kan gevonden worden, want het aantal mogelijke clusteringen groeit exponentieel met n en k.

Clustering kan beschouwd worden als het construeren van een ongerichte graaf, met de objecten als knopen. De clusters zijn dan de samenhangende componenten. Dicht gelegen objecten zullen we zo snel mogelijk in dezelfde cluster onderbrengen, om te vermijden dat ze in verschillende clusters terechtkomen, die dan te dicht bij elkaar zouden liggen. We leggen dus een verbinding tussen paren objecten, volgens stijgende onderlinge afstand. Als twee objecten reeds in dezelfde cluster zitten heeft het geen zin om ze te verbinden, want dat wijzigt toch de samenhangende componenten niet. Er worden dan ook geen lussen gemaakt: de clustering blijft een verzameling (wortelloze) bomen, een bos. En elke toegevoegde verbinding voegt twee clusters samen. Deze methode heet 'single-link clustering' (een speciaal geval van *hierarchical agglomerative clustering*).

Dit is natuurlijk de constructie van een MOB met het algoritme van Kruskal, in een

complete graaf met de objecten als knopen en de afstanden als gewichten. Met dat verschil dat we stoppen wanneer er k samenhangende componenten zijn. De laatste $k-1$ verbindingen worden dus niet toegevoegd. (Men kan het ook zien als een MOB waaruit de $k-1$ zwaarste verbindingen verwijderd werden.)

Is de spreiding van deze clustering wel maximaal? De spreiding is de lengte d^* van de verbinding die het algoritme van Kruskal zou toegevoegd hebben, mochten we het niet onderbroken hebben. Bestaat er een andere clustering $C' = \{C'_1, C'_2, \ldots, C'_k\}$ met een grotere spreiding? Aangezien C' verschilt van C, moet er een cluster zijn van C die twee elementen bevat die tot verschillende clusters van C' behoren. Omdat beide elementen tot dezelfde cluster van C behoren, zijn ze verbonden via een weg waarvan alle verbindingen door Kruskal toegevoegd werden. Het gewicht van al die verbindingen is dus hoogstens d^*. In C' behoren deze elementen tot verschillende clusters: die weg tussen beide moet dus twee *opeenvolgende* knopen p en p' bevatten die ook tot verschillende clusters van C' behoren. Dus is $d(p, p') \leq d^*$. Nu is $d(p, p')$ de afstand tussen twee knopen uit verschillende clusters van C'. De spreiding is de kleinste afstand tussen twee dergelijke knopen. De spreiding van C' is dus ook niet groter dan d^*.

HOOFDSTUK 13
KORTSTE AFSTANDEN I

Het zal wel niemand verwonderen dat de kortste weg tussen twee knopen in een graaf (al dan niet gericht) belangrijk kan zijn. Met 'kortst' bedoelt men gewoonlijk 'met minimaal gewicht' (en het gewicht van een weg is de som van de gewichten van zijn verbindingen), maar soms is enkel het *aantal* verbindingen van belang. Termen zoals 'kortste afstand', 'dichtst gelegen', 'minimaal gewicht', mogen wel niet te letterlijk genomen worden. Als het gewicht écht de geometrische afstand voorstelt, zijn er vaak betere algoritmen beschikbaar, die gebruik maken van meetkundige eigenschappen.

Bij de meeste toepassingen zijn de gewichten positief. Negatieve gewichten kunnen ook wel eens voorkomen, maar de meeste algoritmen worden dan ingewikkelder (en trager). Bovendien mogen er dan geen lussen met negatief gewicht voorkomen, want anders is het begrip 'kortste afstand' niet goed gedefinieerd. We zullen hier steeds onderstellen dat gewichten positief zijn, tenzij anders vermeld.

Het probleem van de kortste afstanden vanuit één knoop naar alle andere knopen staat centraal:

- Geen enkel algoritme voor de kortste afstand tussen een paar knopen is (asymptotisch) efficiënter dan algoritmen die *alle* afstanden tussen één knoop en de overige knopen berekenen. Want om een oplossing voor het eerste probleem te vinden, lossen ze minstens gedeeltelijk het tweede probleem op.
- Een verwant probleem is het bepalen van de kortste afstand van alle knopen *naar* een bepaalde knoop, maar dat herleidt zich tot het vorige als men de richting van alle verbindingen omkeert.
- Wenst men de kortste afstand tussen *elk paar* knopen, dan kan men opnieuw het vorige probleem oplossen met elke knoop als vertrekpunt. In bepaalde gevallen zijn er echter betere oplossingen mogelijk.

13.1 KORTSTE AFSTANDEN VANUIT ÉÉN KNOOP

13.1.1 Niet-gewogen grafen

In een graaf zonder gewichten (of waarbij men de gewichten buiten beschouwing laat) wordt afstand gedefinieerd als het *aantal verbindingen* op de gevolgde weg. Om systematisch de kortste afstanden te vinden vanuit een gegeven knoop, volstaat het om eerst alle knopen te vinden die via één verbinding te bereiken zijn, dan deze waarvoor we twee verbindingen nodig hebben, enz. Dat zijn precies de niveaus van de boom opgebouwd door breedte-eerst zoeken, vanuit de gegeven knoop.[1]

De aanpassing aan breedte-eerst zoeken is minimaal. Telkens wanneer er een knoop ontdekt wordt, is zijn kortste afstand vanuit de wortel die van zijn ouder plus één. Deze afstanden kunnen voor alle knopen in een tabel bijgehouden worden.

13.1.2 Het algoritme van Dijkstra

In zijn oorspronkelijke vorm is dit net zoals het algoritme van Prim, dat er sterk mee verwant is, een van de oudste graafalgoritmen. Het vereist dat de *gewichten positief* zijn, wat in de praktijk ook meestal zo is. (Werkt Prim voor negatieve gewichten?)

Het algoritme steunt op een soortgelijke eigenschap als bij minimale overspannende bomen, en kan de kortste afstanden (en meteen ook de wegen) verbinding per verbinding construeren, zonder op zijn stappen te moeten terugkeren. Het is dus weer een standaardvoorbeeld van een inhalig algoritme.

De kortste wegen vanuit het vertrekpunt vormen eens te meer een (overspannende) boom van de graaf. Het algoritme bouwt die boom geleidelijk op, door weer drie groepen knopen te onderscheiden: deze die reeds deel uitmaken van de (partiële) boom (zwart), de randknopen die rechtstreekse buren zijn van een knoop uit de boom (grijs), en de rest (wit).

Voor de randknopen wordt nu de (voorlopige) kortste afstand *vanuit het vertrekpunt* bijgehouden (via een weg van boomknopen). Telkens wordt de randknoop met de kleinste voorlopige afstand geselecteerd, en in de boom opgenomen. Met deze nieuwe definitieve afstand wordt de rand zo nodig aangepast. Dit gaat door tot alle knopen in de boom zitten.

Waarom werkt dit? Onderstel dat de definitieve kortste afstanden tot de boomknopen reeds gevonden zijn. (Initieel is dat zeker zo, want de boom bevat dan enkel de startknoop, en alle gewichten zijn positief.) De voorlopig kortste afstand die voor elke randknoop bijgehouden wordt, is de kortste afstand naar die knoop via de boomknopen. Wanneer de randknoop met de kleinste voorlopige afstand geselecteerd wordt is dat meteen zijn definitieve kortste afstand. Want een kortere weg zou enkel mogelijk zijn via knopen die nog niet in de boom zitten. Maar dat is onmogelijk, want de afstand tot de andere randknopen is al minstens even groot, en de overige knopen zijn ofwel buren van die randknopen, of liggen nog verder. Die weg is dus zeker niet korter, en

[1] Dit was trouwens de eerste gedocumenteerde toepassing van breedte-eerst zoeken, en diende om de kortste uitweg uit een doolhofstructuur te vinden (Moore, 1959).

bovendien komt er nog een verbinding bij. (Bemerk dat deze redenering niet opgaat als gewichten negatief kunnen zijn.)

De voorlopige afstanden van de randknopen bepalen en aanpassen gebeurt als volgt. Wanneer er een nieuwe knoop in de boom wordt opgenomen, onderzoekt men al zijn buren. Is een buur reeds een randknoop, dan kan de weg via de nieuwe knoop korter zijn, en moet de voorlopige afstand van die buur aangepast worden. Een buur kan via de nieuwe knoop voor het eerst verbonden worden met de boom, en krijgt een voorlopige beginafstand. Met een buur die reeds in de boom zit tenslotte moet er niets gebeuren, want de kortste afstand van die buur was reeds bekend, en de afstand van de nieuwe knoop via die buur is zeker niet korter.

Opnieuw moeten we randknopen met afstanden bijhouden, zodat de knoop met de kleinste afstand efficiënt verwijderd kan worden. Een goede implementatie gebruikt dus een prioriteitswachtrij, net zoals het algoritme van Prim, en met dezelfde voorzieningen om de prioriteit van de knopen efficiënt te kunnen aanpassen. De performantie is dus ook $O(m \lg n)$.

Aangezien de kortste wegen een (eventueel gerichte) boom vormen, kunnen ze gereconstrueerd worden wanneer we de ouder van elke knoop opslaan. Daartoe volstaat het om naast de (voorlopige) kortste afstand van een randknoop, ook zijn onmiddellijke voorloper op die (voorlopige) kortste weg bij te houden. Telkens als de weg en de afstand aangepast worden, moet dat ook gebeuren met die voorloper.

De originele versie van Dijkstra (1959) gebruikte geen prioriteitswachtrij, maar een gewone tabel (zoals de originele methode van Prim), die dan sequentieel werd overlopen. Ook hier werd zoeken naar het volgende minimum gecombineerd met aanpassen van de afstanden. De performantie is dus weer $\Theta(n^2)$, wat nu nog enkel zin heeft voor dichte grafen.

13.2 KORTSTE AFSTANDEN TUSSEN ALLE KNOPENPAREN

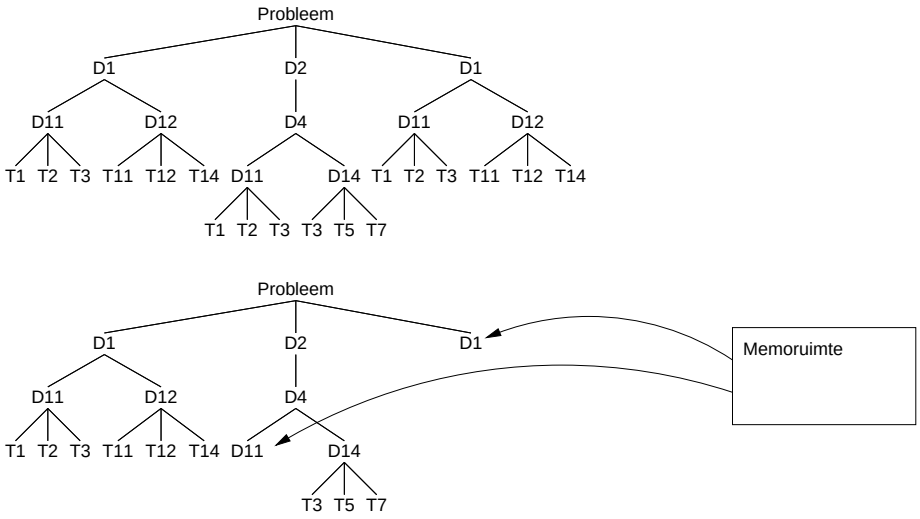
We kunnen dit probleem natuurlijk oplossen door op elke knoop het algoritme van Dijkstra toe te passen (op voorwaarde dat de gewichten positief zijn). Voor ijle grafen is de performantie dan $O(n(n + m) \lg n)$, voor dichte grafen $\Theta(n^3)$. Voor dichte grafen is het algoritme van Floyd-Warshall, hoewel ook $\Theta(n^3)$, in de praktijk sneller, en bovendien laat het *negatieve gewichten* toe. Als de methode van Dijkstra voor elke knoop herhaald wordt, berekent ze dikwijls afstanden die eigenlijk ook bruikbaar zijn voor een volgende keer (of keren), zodat er teveel werk gebeurt. Vermits toch alle afstanden gevraagd worden, houdt het algoritme van Floyd-Warshall[2] (Floyd, 1962) alle voorlopige afstanden in een matrix beschikbaar, en maakt er efficiënt gebruik van. Het is een mooi voorbeeld van 'dynamisch programmeren'.

[2] Dit algoritme werd door Floyd afgeleid uit het algoritme van Warshall voor de transitieve sluiting van een graaf (zie Algoritmen II).

13.2.1 Dynamisch programmeren

Dynamisch programmeren[3] is een belangrijke algoritmische methode om optimale oplossingen te vinden voor combinatorische problemen. Net zoals de verdeel-en-heersmethode vereist ze dat het op te lossen probleem eenvoudig kan opgelost worden als de oplossing van bepaalde gelijkaardige, maar eenvoudiger, problemen gekend is. De oplossing voor deze deelproblemen kan steunen op oplossingen van nog eenvoudiger deelproblemen; er is een onderste laag van problemen die zo eenvoudig zijn dat hun oplossing triviaal is. Zo ontstaat er een boom van problemen met als wortel het op te lossen probleem en als bladeren de triviale problemen, waarbij elke knoop kan worden opgelost als de oplossingen van de kinderen gekend is.

Tot zo ver is er geen verschil tussen dynamisch programmeren en verdeel-en-heers. De verdeel-en-heersmethode wordt dikwijls aangeduid als 'recursief oplossen', omdat ze vaak recursief geprogrammeerd worden. Ook dynamisch programmeren kan op recursieve wijze geïmplementeerd worden, maar er is een bijkomende factor: *memoisatie*. De term duidt erop dat, als een deelprobleem wordt opgelost, er een memootje wordt gemaakt met het resultaat. Als we dan later hetzelfde deelprobleem weer moeten oplossen dan moeten we deze oplossing niet meer zoeken, maar we kunnen ze aflezen van het gemaakte memootje. Dit heeft natuurlijk alleen zin als er overlapping is: deel-



Figuur 13.1. Verdeel-en-heers versus dynamisch programmeren

problemen die op verschillende plaatsen in de boom voorkomen. het moet mogelijk zijn om vlot te identificeren of een bepaald deelprobleem al is opgelost en zo ja, wat de oplossing is. De memoruimte kan dan ook verschillende vormen aannemen. Vaak

[3] De term 'programmeren' staat hier niet voor 'een computerprogramma schrijven', maar staat voor een 'planning' van de oplossing van een probleem, zoals trouwens ook in 'lineair programmeren'.

is het een één- of tweedimensionale tabel (als een probleem kan aangeduid worden met één of twee numerieke parameters), maar andere vormen zijn mogelijk. Ruimte en tijd zijn hier belangrijke factoren: het geheugengebruik mag niet excessief zijn en zoeken naar een al berekende oplossing moet uiteraard sneller gaan dan ze opnieuw berekenen. Fundamenteel zijn er twee methodes bij dynamisch programmeren. De beslissende factor is het antwoord op de vraag of we op voorhand gemakkelijk kunnen bepalen welke deelproblemen moeten opgelost worden en in welke volgorde. Vergelijken we volgende problemen:

(1) Berekening van de Fibonaccigetallen F_n. De triviale gevallen zijn hier $F_0 = 0$ en $F_1 = 1$. Duidelijk moet men om een gegeven F_n te berekenen alle F_i met $i < n$ berekenen.
(2) Berekenen van veralgemeende Fibonaccigetallen $G_n = G_{n-20} + G_{n-14}$, waarbij de triviale gevallen $G_0 \ldots G_{19}$ gegeven zijn. Hier is moeilijk te zien welke G_i moeten berekend worden.

Bij de *bottom-upmethode* berekenen we alle deeloplossingen in een volgorde die garandeert dat, als we bij een bepaald (deel)probleem aankomen, alle onderliggende deelproblemen zijn opgelost. Bij de *top-downmethode* houden we in onze memoruimte bij welke deelproblemen zijn opgelost en welke niet. Vertrekkend vanuit de wortel van onze boom berekenen we welke deelproblemen moeten worden opgelost. Voor elk van die deelproblemen kijken we in de memoruimte of de oplossing al gekend is. Is dit niet het geval dan lossen we het deelprobleem op (dit wordt meestal recursief geïmplementeerd) en noteren dit in de memoruimte voor we verder gaan.

Bij de bottom-upmethode is het niet altijd nodig alle berekende deelresultaten bij te houden. Bij de Fibonaccigetallen moeten we bijvoorbeeld maar twee tussenresultaten F_i en F_{i+1} onthouden en niet alle voorgaande getallen. Bovendien moeten we nooit opzoeken of we een resultaat al kennen, omdat altijd alle nodige deelresultaten die we nodig hebben al gekend zijn. Dit maakt dat de bottom-upmethode, als ze kan gebruikt worden, eenvoudiger en bijna altijd sneller is dan top-down. Ook de hierboven gegeven veralgemeende Fibonaccigetallen zullen sneller, en met minder geheugengebruik, berekend worden met bottom-up dan met top-down, zelfs als we alle G_i met $i < n$ berekenen (noteer dat we G_i met $n-i$ oneven nooit nodig hebben voor het eindresultaat).

Een probleem komt dus in aanmerking voor een efficiënte oplossing via dynamisch programmeren, als het een optimale deelstructuur heeft, en de deelproblemen overlappend zijn. Dat is bij het probleem van de kortste afstanden tussen alle knopenparen van een graaf het geval: een kortste weg bestaat uit kortste wegen, die ook deelwegen kunnen zijn van andere kortste wegen.

13.2.2 Het algoritme van Floyd-Warshall

Dit algoritme werkt met dynamisch programmeren. Het essentiële element is hoe de deelproblemen efficiënt geordend worden zodat we bottom-up kunnen werken met zo

weinig mogelijk geheugen- en tijdsgebruik.

De dichte graaf stellen we voor door een $n \times n$ burenmatrix G, waarvan elk element g_{ij} als volgt gedefinieerd wordt:

$$g_{ij} = \begin{cases} 0 & \text{als } i = j \\ \text{gewicht van verbinding } (i,j) & \text{als } i \neq j \text{ en de verbinding bestaat} \\ \infty & \text{als } i \neq j \text{ en de verbinding niet bestaat} \end{cases}$$

De gezochte oplossing zullen we opslaan in een $n \times n$ *afstandenmatrix* A waarvan element a_{ij} de kortste afstand tussen de knopen i en j bevat (of ∞ als er geen weg bestaat). Daarmee kennen we echter de wegen nog niet die bij deze minimale afstanden horen. Daarvoor is een derde $n \times n$ *voorlopermatrix* V vereist, waarvan het element v_{ij} de (onmiddellijke) voorloper is van knoop j op een kortste weg vanuit knoop i. De kortste wegen kunnen we hieruit gemakkelijk reconstrueren.

Initieel beschikken we enkel over de rechtstreekse afstand tussen elk paar knopen, wat dus voorlopig ook hun kortste afstand is. De kortste weg tussen elk knopenpaar mag uiteindelijk elke andere knoop bevatten. Daarom zullen we systematisch meer intermediaire knopen op elke weg toelaten, en telkens nieuwe voorlopig kortste afstanden bepalen aan de hand van de vorige.

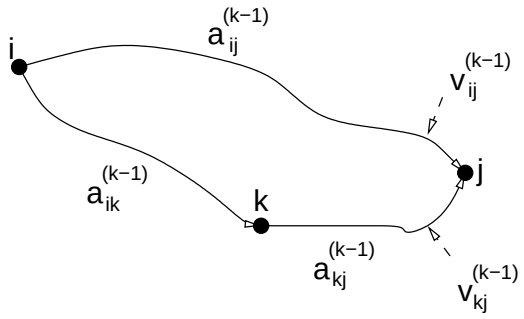
Stel dat we voorlopig kortste wegen hebben voor alle knopenparen, die enkel gebruik *mogen* maken van de intermediaire knopen $0, 1, 2, \ldots, k-1$. Als we nu ook knoop k toelaten, kunnen we dan de nieuwe kortste afstanden bepalen? Onderstel dat w_{ij} de voorlopig kortste weg is tussen i en j, die ook knoop k *mag* gebruiken:

- Als k geen intermediaire knoop van w_{ij} is, dan is w_{ij} de vorige kortste weg die k niet mocht gebruiken.
- Indien wel, dan kunnen we w_{ij} onderverdelen in de deelwegen w_{ik} van i naar k en w_{kj} van k naar j. Deelwegen van een kortste weg zijn ook kortste wegen, en noch w_{ik} noch w_{kj} bevatten k als intermediaire knoop. (Aangezien er geen lussen zijn op een kortste weg komt k slechts eenmaal voor.) De lengte van beide is dus reeds bekend, zodat we het nieuwe minimum kunnen bepalen.

Om de oplossingsmatrix A te vinden wordt een reeks afstandenmatrices $A^{(0)}, A^{(1)}, \ldots, A^{(n)}$ berekend. $A^{(0)}$ laat geen enkele tussenliggende knoop toe en is dus gelijk aan G. $A^{(n)}$ laat alle knopen toe en is dus de gezochte matrix A. Voor $k \geq 0$ wordt $A^{(k+1)}$ afgeleid uit $A^{(k)}$, want (zie figuur 13.2)

$$a_{ij}^{(k+1)} = \min\left(a_{ij}^{(k)}, a_{ik}^{(k)} + a_{kj}^{(k)}\right)$$

Bemerk dat het (gelukkig) niet nodig is om al die afstandenmatrices op te slaan. Twee opeenvolgende matrices volstaan, meer nog, er is maar één afstandenmatrix nodig, aangezien de berekening ter plaatse kan gebeuren. (Waarom?)



Figuur 13.2. Algoritme van Floyd-Warshall: eventueel kortere weg via knoop k.

De performantie van dit algoritme is duidelijk $\Theta(n^3)$. De verborgen constante is echter klein, zodat het zelfs voor middelmatig grote n goed bruikbaar is.

De voorlopermatrix V kan samen met A opgebouwd worden door een analoge reeks voorlopermatrices $V^{(0)}, V^{(1)}, \ldots, V^{(n)}$ te bepalen. De laatste matrix is dan de gezochte V. Voor de initiële matrix $V^{(0)}$ geldt

$$v_{ij}^{(0)} = \begin{cases} -1 & \text{als } i = j \text{ of } g_{ij} = \infty \text{ (geen voorloper)} \\ i & \text{als } i \neq j \text{ en } g_{ij} < \infty \end{cases}$$

en voor de andere matrices maken we gebruik van

$$v_{ij}^{(k+1)} = \begin{cases} v_{ij}^{(k)} & \text{als } a_{ij}^{(k)} \leq a_{ik}^{(k)} + a_{kj}^{(k)} \\ v_{kj}^{(k)} & \text{als } a_{ij}^{(k)} > a_{ik}^{(k)} + a_{kj}^{(k)} \end{cases} \qquad \text{voor } 0 \leq k < n$$

Merk op dat bij gelijkheid de voorloper ongewijzigd moet blijven. (Waarom?) Ook hier kan de berekening ter plaatste gebeuren, zodat één voorlopermatrix volstaat.

Een alternatieve manier bepaalt V rechtstreeks uit A en is eveneens $\Theta(n^3)$. (Hoe?)

HOOFDSTUK 14
GERICHTE LUSLOZE GRAFEN

Gerichte grafen die geen lussen bevatten[1] verschillen vrij sterk van gewone gerichte grafen. Vanuit elke knoop zien ze er immers uit als een boom, zodat men zou kunnen zeggen dat ze half graaf, half boom zijn. Door de afwezigheid van lussen vallen een aantal graafalgoritmen eenvoudiger uit dan in het algemeen geval. Efficiënt testen of een graaf lusvrij is kan bijvoorbeeld via diepte-eerst zoeken.

14.1 TOPOLOGISCH RANGSCHIKKEN

Topologisch rangschikken plaatst alle graafknopen zó op een horizontale as, dat alle verbindingen naar rechts wijzen. Met andere woorden, als een verbinding (i, j) tot de graaf behoort, staat knoop j steeds rechts van knoop i, en dat is natuurlijk onmogelijk als de graaf lussen heeft. Topologisch rangschikken kan dan ook gebruikt worden om te testen of een graaf lussen heeft.

Een topologische ordening is niet noodzakelijk uniek, omdat er knopen zonder onderlinge relatie kunnen voorkomen. We bespreken twee lineaire algoritmen voor dit probleem.

14.1.1 Via diepte-eerst zoeken

De zoveelste toepassing van deze methode. Wanneer diepte-eerst zoeken een knoop in postorder *afwerkt*, dan zijn alle verbindingen vanuit die knoop behandeld, en die leiden allemaal naar reeds afgewerkte knopen. Want boomtakken en heenverbindingen wijzen naar beneden, in de deelboom waarvan de knoop wortel is, en dwarsverbindingen wijzen naar links. Bovendien heeft een lusloze graaf geen terugverbindingen. Kortom, als een afgewerkte knoop op de as geplaatst wordt moeten alle vroeger afgewerkte knopen rechts daarvan liggen.

De *postordernummering* van de knopen geeft de volgorde van afwerken. Als men dus een knoop plaatst, moeten alle knopen met lagere postordernummers rechts daarvan liggen. Een dalende postordernummering geeft dan de plaatsingsvolgorde vanaf links.

[1] Engels *'directed acyclic graph'*, afgekort tot 'dag'.

Omdat postordernummeren niet veel meer is dan diepte-eerst zoeken, is de performantie $\Theta(n + m)$ voor ijle grafen.

14.1.2 Via ingraden

Dit algoritme begint met het berekenen van de *ingraad* van alle knopen. Een knoop zonder inkomende verbinding (ingraad nul) kan meteen helemaal links op de as geplaatst worden. Elke lusloze graaf heeft minstens één dergelijke knoop. (Waarom?) Wanneer men die knoop met al zijn verbindingen uit de lusloze graaf verwijdert, blijft er uiteraard een lusloze graaf over. Elke knoop met ingraad nul van deze nieuwe graaf kan dan rechts van de eerste geplaatst worden, want enkel de verwijderde knoop kan een verbinding naar deze knoop hebben. De tweede knoop wordt dan uit de graaf verwijderd, en een derde knoop met ingraad nul uit de nieuwe graaf wordt rechts van de tweede geplaatst. Want enkel de twee reeds geplaatste knopen kunnen een verbinding naar deze knoop hebben. Dit gaat door tot alle knopen geplaatst zijn. (Hoe zou dit algoritme reageren op de aanwezigheid van een lus?)

De enige informatie die we nodig hebben om een volgende knoop te plaatsen is de ingraad van de knopen. Verwijderen van een knoop komt dan neer op het verminderen van de ingraad van al zijn buren met één. Elke knoop waarvan de ingraad nul wordt of reeds nul was, kan dan geplaatst worden.

Telkens de tabel met ingraden doorzoeken naar knopen met ingraad nul is niet efficiënt. Een betere oplossing houdt alle knopen met ingraad nul bij, en neemt er telkens een willekeurige knoop uit. (Daarvoor kan men een eenvoudige gegevensstructuur als een stapel of een wachtrij gebruiken.) Wanneer we de ingraad van een knoop verminderen moeten we testen of die nul wordt, want dan moet hij bij de andere gevoegd worden.

Het berekenen van de ingraden is $\Theta(n) + \Theta(m) = \Theta(n + m)$. Elke knoop wordt eenmaal toegevoegd aan en verwijderd uit de stapel of wachtrij, wat in totaal $\Theta(n)$ is. Tenslotte moeten de buren van elke knoop behandeld worden, en dat is $\Theta(m)$. Voor ijle grafen is het algoritme dus $\Theta(n + m)$, net zoals het vorige.

14.2 KORTSTE AFSTANDEN VANUIT ÉÉN KNOOP

Bij deze grafen is de kortste afstand steeds gedefinieerd, ook als er negatieve gewichten zijn. Er zijn immers geen lussen, dus zeker geen lussen met negatief gewicht.

Het algoritme van Dijkstra vond de kortste afstanden tot de knopen in stijgende (niet dalende) volgorde, met behulp van een prioriteitswachtrij. Verder gelegen knopen konden immers de kortste afstand tot een knoop niet meer verbeteren, aangezien alle gewichten positief waren. Bij lusloze grafen is het veel eenvoudiger: knopen die rechts van een knoop liggen in de topologische volgorde kunnen zijn kortste afstand niet meer wijzigen, want al hun verbindingen wijzen naar rechts. Het volstaat dus om de graaf

topologisch te rangschikken, en dan de knopen in die volgorde te behandelen, beginnend bij de startknoop. (Bemerk dat dit niet noodzakelijk de eerste knoop is van de topologische volgorde.)

Opnieuw houden we voorlopige afstanden (en voorlopers) bij voor alle nog niet behandelde knopen. Als we een knoop behandelen kennen we zijn definitieve kortste afstand en voorloper. Deze kortste afstand kan de voorlopige afstanden van zijn buren verbeteren (want die liggen rechts van hem op de topologische as), en die moeten dan aangepast worden (samen met de voorlopers).

Topologisch rangschikken is $\Theta(n + m)$, initialisatie van de afstanden en voorlopers $\Theta(n)$, en de buren van elke knoop komen eenmaal aan bod, voor een bijkomende term $\Theta(m)$. Voor ijle grafen is de performantie dus $\Theta(n + m)$.

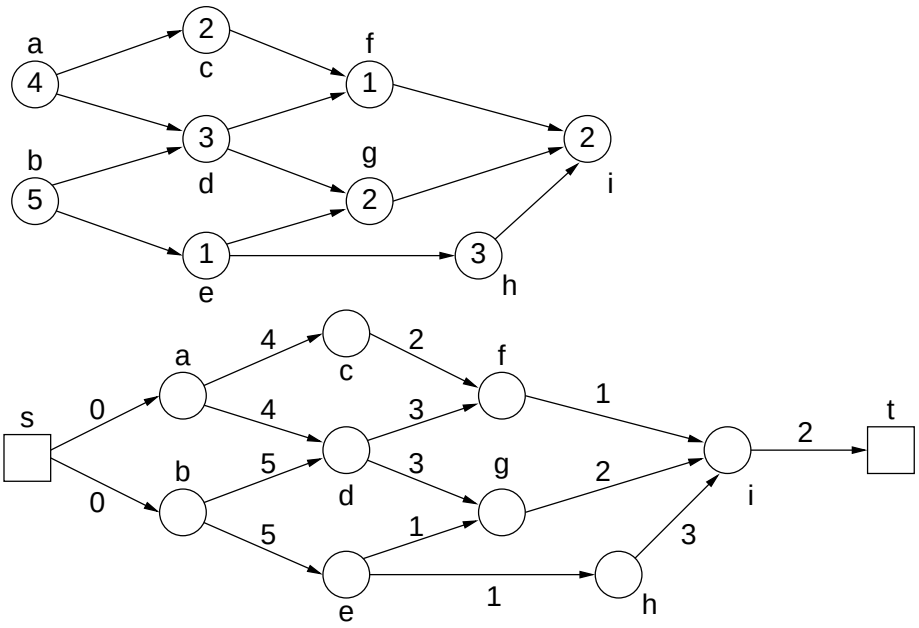
14.2.1 Projectplanning

Stel dat een complex project uit meerdere deeltaken bestaat, waarvan sommige moeten afgewerkt worden vóór andere, en dat de uitvoeringstijd voor elke taak gekend is. Dan kan men zich afvragen in hoeveel tijd alles kan afgewerkt worden, of hoeveel vertraging een bepaalde taak mag oplopen zonder het hele project te vertragen, enz. Praktische technieken voor dergelijke projectplanning zoals CPM en PERT[2] lossen dergelijke vragen op met behulp van lusloze grafen die de taken en hun onderlinge afhankelijkheid voorstellen. Voor de eenvoud zullen we hier onderstellen dat elke taak waarvoor de vereisten voldaan zijn, kan aangevat worden, onafhankelijk van andere taken die nog lopen.

Als men het probleem rechtstreeks in een graaf vertaalt, dan stellen de knopen de taken voor, met de voor elke taak vereiste tijd, terwijl de gewichtloze verbindingen enkel de afhankelijkheden uitdrukken. Willen we echter de standaardalgoritmen voor gewogen grafen gebruiken, dan moeten de *verbindingen* een gewicht (tijdsduur) krijgen, niet de knopen.

Gelukkig is het niet zo moeilijk om een equivalente graaf op te stellen. Dat kan op twee manieren: ofwel wordt elke taak een verbinding (meest gebruikt), ofwel blijft elke taak een knoop (eenvoudiger). We kiezen voor de tweede manier: de taak blijft bij de knoop, maar de taakduur wordt het gewicht van elk van zijn uitgaande verbindingen. Alle knopen zonder opvolgers worden verbonden met een nieuwe eindknoop, zodat ook hun taakduur aan deze verbindingen kan toegekend worden. Als er meerdere knopen met ingraad nul zijn, komt er een nieuwe startknoop bij, met verbindingen van gewicht nul naar al die knopen. In plaats van taken met een duur zijn de knopen nu gebeurtenissen geworden: het opstarten van de overeenkomstige taken. Figuur 14.1 stelt de oorspronkelijke graaf voor met de taakduur bij de knopen, en de nieuwe graaf met de taakduur bij de verbindingen.

[2] CPM ('Critical Path Method', E.I. du Pont de Nemours & Co.) en PERT ('Program Evaluation and Review Technique', Remington Rand), beide ontwikkeld tussen 1956 en 1958.



Figuur 14.1. Projectplanning: taakduur eerst bij de knopen en dan bij de verbindingen.

Deze nieuwe lusloze graaf met n knopen en m verbindingen stelt ons in staat om een aantal interessante gegevens te bepalen:

- *Minimale projectduur.* Elke weg tussen de begin- en eindknoop van het project stelt een opgelegde volgorde van taken voor, en de lengte van de weg is de totale tijd die daarvoor vereist is. Aangezien alle taken moeten voltooid worden, in de juiste volgorde, bepaalt de langste van die wegen de minimumtijd van het geheel.

  Langste afstanden zijn steeds goed gedefinieerd in lusloze grafen, zelfs met positieve gewichten. Langste afstanden kan men vinden door kortste afstanden te bepalen, nadat het teken van alle gewichten werd omgekeerd, of ook door het algoritme voor de kortste afstanden (licht) aan te passen (anders initialiseren, vergelijkingen omkeren). De performantie blijft natuurlijk dezelfde als voor kortste afstanden, dus $\Theta(n + m)$ bij ijle grafen.

- *Vroegste aanvangstijden.* Bij de bepaling van de langste afstand vanuit de beginknoop naar de eindknoop bekomen we ook de langste afstanden tot alle andere knopen. Als de tijdsoorsprong bij de beginknoop ligt, dan kennen we op die manier de vroegste aanvangstijd van elke knoop (en dus taak). Want een taak kan maar aangevangen worden als alle ervoor vereiste taken afgewerkt zijn.

- *Uiterste aanvangstijden.* Een ander interessant gegeven is de uiterste aanvangstijd van elke taak, zonder de minimumtijd voor het hele project te veranderen.

Dat betekent dat de uiterste aanvangstijd van de eindknoop gelijk moet zijn aan zijn vroegste aanvangstijd. (Bij de eindknoop is er natuurlijk niets meer te doen.) De uiterste aanvangstijd van alle andere knopen kan eenvoudig berekend worden als die van al hun buren gekend is. Voor elke knoop is die tijd immers de vroegste van de uiterste aanvangstijden van al zijn buren, min de taakduur van de knoop zelf.

Nu liggen alle buren van een knoop rechts ervan op de topologische as. Deze berekening kan dus voor alle knopen gebeuren in *omgekeerde topologische volgorde*. Aangezien de topologische volgorde reeds bepaald werd voor de minimale projectduur (en de vroegste aanvangstijden), blijft er nog een initialisatie van $\Theta(n)$ over en het behandelen van alle buren van alle knopen, wat $\Theta(m)$ vereist. In totaal krijgen we hiervoor dan een performantie van $\Theta(n + m)$.

- *Vertraging, en kritieke weg.* Tenslotte kan men met de vorige gegevens de vertraging (*'slack time'*) berekenen die elke taak *afzonderlijk* mag oplopen, zonder invloed op het gehele project. Die is natuurlijk gelijk aan het verschil van de uiterste aanvangstijd van die taak en zijn vroegste aanvangstijd.

  Taken die geen vertraging mogen oplopen zijn *kritiek*, en er bestaat steeds minstens één kritieke weg (*critical path*) waarop alle taken kritiek zijn.

Tabel 14.1 bevat al deze resultaten voor de taken uit figuur 14.1. Er is slechts één kritieke weg: s, b, d, g, i, t. Implementatie van vroegste en uiterste aanvangstijd zijn

taak	s	a	b	c	d	e	f	g	h	i	t
vroegste tijd	0	0	0	4	5	5	8	8	6	10	12
uiterste tijd	0	1	0	7	5	6	9	8	7	10	12
vertraging	0	1	0	3	0	1	1	0	1	0	0

Tabel 14.1. Resultaten voor figuur 14.1.

voorbeelden van twee mechanismen die gebruikt worden wanneer onderling van elkaar afhankelijke resultaten moeten berekend worden en die bekend staan als *pushing* (duwen) en *pulling* (trekken).

Bij het eerste mechanisme duwt een knoop die zijn resultaat kent dit vooruit naar zijn buren, bij het tweede mechanisme trekt een knoop die zijn resultaat wil kennen de resultaten van zijn voorgangers naar zich toe. Vroegste aanvang berekenen gebeurt, met een klassieke graafstructuur met burenlijsten, door te duwen (een knoop kent zijn opvolgers maar niet zijn voorgangers). Bij het berekenen van de uiterste aanvang moet een taak de uiterste aanvangsmomenten van zijn opvolgers kennen en trekt deze naar zich toe.

BIBLIOGRAFIE

[1] BENTLEY J.L. en McILROY M.D. Engineering a Sort Function. *Software - Practice and Experience*, 23(11):1249–1265, November 1993.

[2] BENTLEY J.L. en SEDGEWICK R. Fast Algorithms for Sorting and Searching Strings. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, Januari 1997.

[3] CIURA M. Best Increments for the Average Case of Shellsort. In R. Freivalds, editor, *FCT 2001*, LNCS, pages 106–117. Springer Verlag, 2001.

[4] CORMEN T.H., LEISERSON C.E., RIVEST R.L., en STEIN C. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition, 2009.

[5] ESTIVILL-CASTRO V. en WOOD D. A Survey of Adaptive Sorting Algorithms. *ACM Computing Surveys*, 24(4):441–475, December 1992.

[6] FRIGO M., LEISERSON C.E., PROKOP H. en RAMACHANDRAN S. Cache-oblivous Algorithms. In *Proc. 40th Annual Symposium on Foundations of Computer Science*, pages 285–297. IEEE Computer Society Press, Washington D.C., 1999.

[7] GOODRICH M.T. en TAMASSIA R. *Algorithm Design and Applications*. Wiley, Hoboken, NJ, 2015.

[8] GRAEFE G. B-Tree Indexes, Interpolation Search, and Skew. In *Proceedings of the Second International Workshop on Data Management on New hardware (DaMoN 2006)*. ACM, Juni 2006.

[9] GRAEFE G. Implementing Sorting in Database Systems. *ACM Computing Surveys*, 38(3):1–37, September 2006.

[10] JIANG T., LI M., en VITANYI P. The Average-case Complexity of Shellsort. In *Proceedings of ICALP'99*, Lecture Notes in Computer Science, pages 453–462. Springer Verlag, Berlin, 1999.

[11] KNUTH D.E. *The Art of Computer Programming, Vol 3, Sorting and Searching*. Addison-Wesley, Reading, MA, 2nd edition, 1998.

[12] LARSON P. External Sorting: Run Formation Revisited. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):961–972, Juli/Augustus 2003.

[13] LEVITIN A. *Introduction to the Design and Analysis of Algorithms*. Addison-Wesley, Reading, MA, 3rd edition, 2012.

[14] MANBER U. *Introduction to Algorithms. A Creative Approach*. Addison-Wesley, Reading, MA, 1989.

[15] MUSSER D.J. Introspective Sorting and Selection Algorithms. *Software-Practice and Experience*, 27(8):983–993, Augustus 1997.

[16] SEDGEWICK R. *Algorithms in C++, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching, Part 5: Graph Algorithms*. Addison-Wesley, Reading, MA, 3rd edition, 1998.

[17] STROUSTRUP B. *The C++ Programming Language*. Addison-Wesley, Boston, MA, 3rd edition, 1997.

[18] VITTER J.S. External Memory Algorithms and Data Structures: Dealing with Massive Data. *ACM Computing Surveys*, 33(2):209–271, Juni 2001.

[19] WEISS M.A. *Data Structures and Algorithm Analysis in C++*. Addison-Wesley, Reading, MA, 3rd edition, 2006.

[20] WILD S., NEBEL M. en NEININGER R. Average Case and Distributional Analysis of Dual-Pivot Quicksort. *ACM Transactions on Algorithms*, 11(3):article 22, Januari 2015.

[21] Yaroslavskiy V. Dual-Pivot Quicksort. *http://iaroslavski.narod.ru/ quicksort/-DualPivotQuicksort.pdf*, 2009.