# Hoofdstuk 1

# Efficiënte zoekbomen

## 1.1 Inleiding

De uitvoeringstijd van de meeste operaties op een binaire zoekboom met hoogte $h$ is $O(h)$. Deze hoogte is sterk afhankelijk van de toevoegvolgorde van de gegevens. Als elke toevoegvolgorde even waarschijnlijk is, dan is de verwachtingswaarde voor de hoogte van een zoekboom met $n$ gegevens $O(\lg n)$. Maar in het slechtste geval is die hoogte $O(n)$, zodat we eigenlijk met een soort gelinkte lijst te maken hebben, met alle gevolgen vandien. Elke toevoegvolgorde even waarschijnlijk onderstellen is niet zeer realistisch. Daarom werd er reeds vroeg gezocht naar manieren om de efficiëntie van zoekbomen te verbeteren. Er zijn verschillende mogelijkheden:

- *Elke operatie steeds efficiënt maken.* De vorm van *evenwichtige* zoekbomen ('balanced search trees') blijft steeds perfect of nagenoeg perfect. De operaties zijn dan ook altijd efficiënt:
  - De eerste evenwichtige bomen waren 'AVL-bomen'. (Genoemd naar de initialen van hun uitvinders, Adel'son-Vel'skiĭ en Landis, 1962.) Bij deze bomen mag het hoogteverschil tussen de twee deelbomen van elke knoop nooit groter zijn dan één. Daartoe slaat men in elke knoop zijn hoogte op. (Eigenlijk volstaat het hoogteverschil tussen zijn deelbomen, en daarvoor zijn in principe slechts twee bits nodig.) Toevoegen of verwijderen van een knoop kan het evenwicht in de boom verstoren, zodat er structuurwijzigingen moeten gebeuren om dat te herstellen. Daarbij mag echter niet aan de fundamentele eigenschap van een zoekboom geraakt worden. In de praktijk worden AVL-bomen niet vaak meer gebruikt, omdat hun implementatie ingewikkeld is, en de structuurwijzigingen soms veel werk vereisen. ($O(\lg n)$ rotaties, zie later.)
  - Later werden '2-3-bomen' uitgevonden (Hopcroft, 1970), waarvan elke (inwendige) knoop twee of drie kinderen heeft (vandaar de naam), en alle bladeren dezelfde diepte hebben. Bij toevoegen of verwijderen wordt dit ideale evenwicht behouden door het aantal kinderen van de knopen te manipuleren. Deze bomen zijn tevens de voorlopers van de 'B-trees' uit hoofdstuk 3. Analoog definieert men '2-3-4-bomen',

waarop de operaties wat eenvoudiger uitvallen, omdat de drie soorten knopen meer flexibiliteit bieden.

Knopen van verschillende soorten gebruiken, die bovendien van soort kunnen veranderen, geeft praktische problemen bij implementatie. (Dat geldt niet voor B-trees, zie later.) Beide bomen kunnen echter door equivalente *binaire* bomen voorgesteld worden ('Binary B-trees', Bayer, 1971, en 'Symmetric binary B-trees', Bayer, 1972). Daarbij hebben de bladeren niet meer dezelfde diepte: het perfecte evenwicht wordt opgegeven, maar de hoogte blijft $O(\lg n)$. Deze ideeën leidden uiteindelijk tot rood-zwarte bomen, de meest gebruikte evenwichtige zoekbomen[1].

- *Elke reeks operaties steeds efficiënt maken.* Bij 'splay trees' wordt de vorm van de boom meermaals aangepast, zodat die nooit slecht blijft. Met als gevolg dat elke *reeks* opeenvolgende operaties gegarandeerd efficiënt is, ook al kan een individuele operatie zeer traag uitvallen. Met andere woorden, *uitgemiddeld over de reeks* ('geamortiseerd') is de performantie per operatie goed, ook in het slechtste geval.

- *De gemiddelde efficiëntie onafhankelijk maken van de operatievolgorde.* Door gebruik te maken van een random generator zorgen 'randomized search trees' ervoor dat de boom random is en ook blijft, onafhankelijk van de volgorde van toevoegen en verwijderen. Met als gevolg dat de verwachtingswaarde van de performantie van hun operaties steeds $O(\lg n)$ is. De eenvoudigste zoekboom uit deze familie is een 'treap'.

## 1.2 Rood-zwarte bomen

### 1.2.1 Definitie en eigenschappen

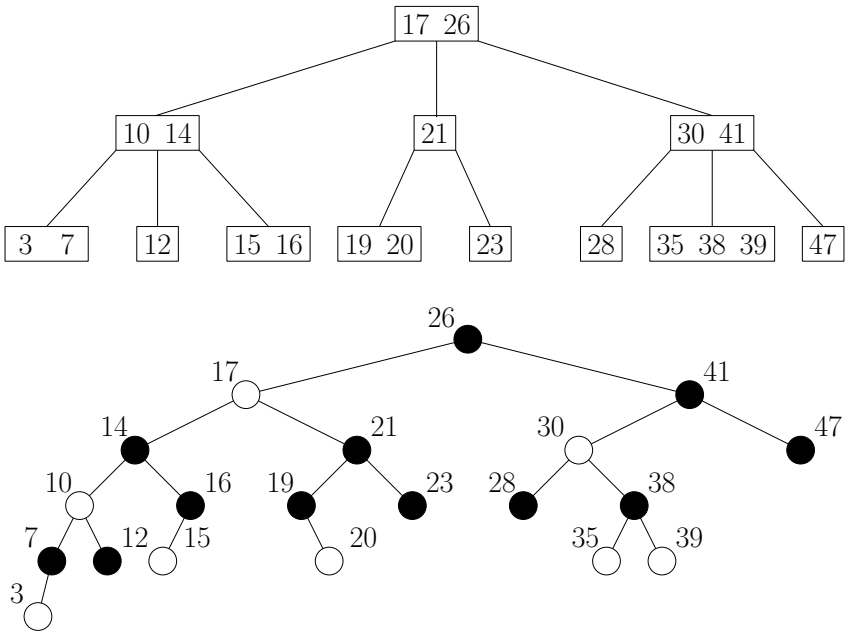
#### Definitie

Een rood-zwarte boom ('red-black tree', Guibas en Sedgewick, 1978) is een binaire zoekboom die een 2-3-4-boom simuleert. Daartoe vervangt men een 3-knoop door twee, en een 4-knoop door drie binaire knopen. Daarbij definieert men twee soorten ouder-kind verbindingen. De originele verbindingen van de 2-3-4-boom noemt men zwart, de nieuwe verbindingen rood. Die kleur moet in principe opgeslagen worden bij de verbindingen (pointers), maar het is gemakkelijker om ze op te slaan bij de kinderen. De hoogte van de boom wordt laag gehouden door beperkingen op te leggen aan de manier waarop knopen op elke (dalende) weg vanuit de wortel gekleurd kunnen worden. Figuur 1.1 toont een 2-3-4-boom en een equivalente rood-zwarte boom.

We gebruiken dezelfde knoopstructuur als bij een gewone binaire zoekboom, met een nieuw veld voor de kleur. (Waarvoor in principe één bit volstaat.) Ontbrekende kinderen worden voorgesteld door nullwijzers, die we echter conceptueel beschouwen als wijzers naar uitwendige (dus kinderloze) knopen van de boom, die geen gegevens bevatten, maar wel een kleur hebben. Alle gewone knopen worden aldus inwendig.

Een rood-zwarte boom wordt dan gedefinieerd als een binaire zoekboom, waarbij bovendien:

[1] Associatieve containers uit zowel de Standard Template Library van C++ (`map`, `set`) als uit de Collections API van Java (`TreeMap`, `TreeSet`) worden geïmplementeerd met rood-zwarte bomen.




Figuur 1.1: 2-3-4-boom en equivalente rood-zwarte boom (wit stelt hier rood voor).

1. Elke knoop rood of zwart gekleurd is.
2. Elk blad (uitwendige knoop) zwart is.
3. Een rode knoop steeds twee zwarte kinderen heeft. (Zijn er steeds twee?)
4. Elke mogelijke (rechtstreekse) weg vanuit een knoop naar een blad evenveel zwarte knopen heeft. Dat aantal noemt men de *zwarte hoogte* van de knoop. Daarbij wordt de knoop zelf niet meegerekend (zoals bij de gewone hoogte), het (zwarte) blad echter wel.
5. De wortel zwart is. Deze vereiste is niet strikt noodzakelijk, maar wel handig. Een rode wortel kan trouwens steeds zwart gemaakt worden, zonder gevolgen voor de zwarte hoogten.

**Eigenschappen**

Uit deze definitie kunnen we afleiden dat een deelboom met wortel $w$ en zwarte hoogte $z(w)$ tenminste $2^{z(w)} - 1$ *inwendige* knopen (en dus gegevens) bevat. Dat is eenvoudig aan te tonen via inductie op de (gewone) hoogte van de deelboom:

1. *Hoogte nul.* Een deelboom met hoogte nul bevat enkel een blad, dat zwarte hoogte nul heeft. Er zijn dus geen inwendige knopen, wat inderdaad minstens $2^0 - 1 = 0$ is.
2. *Positieve hoogte.* Daarbij onderstellen we dat de eigenschap geldt voor elke kleinere hoogte. De wortel $w$ van een deelboom met positieve hoogte is een inwendige knoop die steeds twee kinderen heeft (eventueel bladeren). Nu is de zwarte hoogte van een rood kind $z(w)$, en van een zwart kind $z(w) - 1$. De hoogte van beide kinderen is kleiner dan

die van $w$, zodat bij onderstelling de eigenschap geldt voor de deelbomen waarvan ze wortel zijn. De deelboom met wortel $w$ heeft dan minstens

$$2(2^{z(w)-1} - 1) + 1 = 2^{z(w)} - 1$$

inwendige knopen, wat de eigenschap bewijst.

Stel nu dat een rood-zwarte boom met $n$ inwendige knopen een hoogte $h$ heeft (blad meegerekend). Dat er geen twee opeenvolgende rode knopen op elke weg vanuit een knoop naar een blad mogen voorkomen, impliceert dat de langste van die wegen hoogstens tweemaal zo lang is als de kortste. Voor de wortel $w$ betekent dat $z(w) \geq h/2$, zodat uit de vorige eigenschap volgt dat

$$n \geq 2^{z(w)} - 1 \geq 2^{h/2} - 1$$

wat uiteindelijk leert dat

$$h \leq 2 \lg(n + 1).$$

De hoogte van een rood-zwarte boom met $n$ knopen is dus steeds $O(\lg n)$.

De boom is nu slechts bij benadering evenwichtig. Omdat de eisen hier minder streng zijn dan bij een AVL-boom, zijn de operaties op deze bomen eenvoudiger en efficiënter te implementeren.

### 1.2.2 Zoeken

Bij zoekoperaties speelt de kleur van de knopen geen rol, en wordt de rood-zwarte boom een gewone binaire zoekboom. Zijn hoogte is echter steeds $O(\lg n)$, zodat zoeken naar een willekeurige sleutel, zoeken naar de kleinste en de grootste sleutel, en zoeken naar de opvolger en de voorloper van een sleutel, die allemaal $O(h)$ waren, nu gegarandeerd $O(\lg n)$ worden. Zoeken is dus altijd efficiënt.

### 1.2.3 Toevoegen en verwijderen

Een element toevoegen aan een rood-zwarte boom, of er een uit verwijderen, zonder rekening te houden met de kleur, is dus ook $O(\lg n)$. Het is echter niet zeker dat deze gewijzigde boom nog rood-zwart zal zijn, zodat de efficiëntie van erop volgende operaties niet meer verzekerd kan worden. Gelukkig zal blijken dat toevoegen en verwijderen, met als resultaat een nieuwe rood-zwarte boom, toch efficiënt kunnen gebeuren, als er bepaalde kleur- en structuurwijzigingen aan de boom gebeuren.

Bij toevoegen kan men op twee manieren te werk gaan:

- Ofwel voegt men eerst toe zonder op de kleur te letten, zoals bij een gewone binaire zoekboom (een blad wordt vervangen door de nieuwe knoop). Daarna herstelt men de rood-zwarte boom, te beginnen bij de nieuwe knoop, en desnoods tot bij de wortel. We dalen dus af, om nadien langs dezelfde weg weer te stijgen. Ouderwijzers of een stapel zijn hiervoor vereist. Men spreekt van een 'bottom-up' rood-zwarte boom.

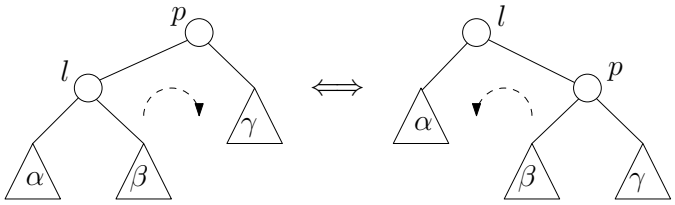
- Ofwel wordt de boom reeds aangepast langs de dalende zoekweg. Deze variant heet natuurlijk 'top-down', en blijkt in de praktijk efficiënter zowel in tijd als in plaats, omdat ouderwijzers noch stapel nodig zijn.

Ook verwijderen kan 'bottom-up' of 'top-down' gebeuren.

In beide soorten rood-zwarte bomen gebeuren de structuurwijzigingen met *rotaties*, die reeds bij AVL-bomen gebruikt werden.

#### 1.2.3.1 Rotaties

Rotaties wijzigen de vorm van de boom, maar behouden de inorder volgorde van de sleutels. (Waarom is dat nodig?) De kleur van de knopen blijft onveranderd. Een rotatie is een lokale operatie, waarbij twee *inwendige* knopen betrokken zijn. (Een ouder en een van zijn kinderen.) De rotatie 'draait' rond de ouder-kind verbinding. Er zijn twee soorten, een rotatie naar links voor een rechterkind en een naar rechts voor een linkerkind, en ze zijn elkaars inverse (figuur 1.2).




Figuur 1.2: Rotaties.

Een rotatie moet drie ouder-kind verbindingen aanpassen. (In beide richtingen, als er ouderwijzers zijn.) Bij een rotatie naar rechts van een ouder $p$ en zijn linkerkind $l$ bijvoorbeeld wordt het rechterkind van $l$ (als het bestaat) het linkerkind van $p$, de ouder van $p$ (als die bestaat) wordt de ouder van $l$, en $p$ wordt het rechterkind van $l$. Een rotatie is dus $O(1)$.

#### 1.2.3.2 Bottom-up rood-zwarte bomen

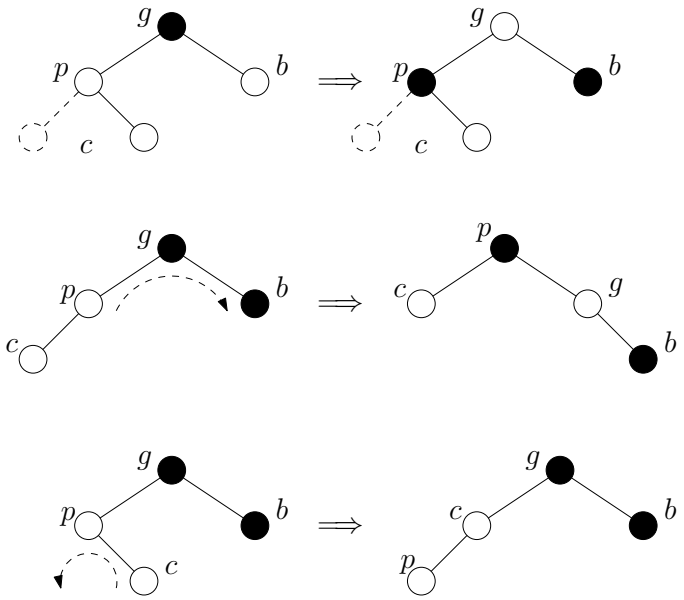
**Toevoegen**

De nieuwe knoop wordt dus eerst op de gewone manier toegevoegd. Maar welke kleur geven we hem? Een zwarte knoop kan de zwarte hoogte van veel knopen ontregelen, en een rode knoop mag enkel als de ouder zwart is. We kiezen toch maar voor rood, omdat de zwarte hoogte moeilijker te herstellen valt.

Er is dus een probleem als de ouder ook rood is: er zijn dan twee opeenvolgende rode knopen. We zullen deze storing trachten te verwijderen door enkele rotaties en kleurwijzigingen, nadat we ze eventueel eerst naar boven in de boom opgeschoven hebben, desnoods tot bij de wortel.

Aangezien de ouder $p$ van de nieuwe knoop $c$ rood is, kan hij geen wortel zijn (want die is steeds zwart). Er is dus een grootouder $g$, die zwart is (want de boom waaraan we $c$ toevoegen was rood-zwart, en die heeft geen opeenvolgende rode knopen). We krijgen dan *zes* mogelijke gevallen, die uiteenvallen in twee groepen van drie, naar gelang dat $p$ een linker- of rechterkind is van $g$. De operaties voor beide groepen zijn elkaars spiegelbeeld, zodat we slechts één groep moeten bespreken.

*Onderstel dat $p$ een linkerkind is van $g$.* We kunnen de rode ouder $p$ elimineren ofwel door hem zwart te maken (dat vereist een zwarte knoop die rood moet worden, ter compensatie), ofwel door hem weg te nemen via een rotatie naar rechts (als dit niet opnieuw twee opeenvolgende rode knopen oplevert in de rechterdeelboom). Er zijn dan ook twee hoofdgevallen, naar gelang van de kleur van de broer $b$ van $p$ (figuur 1.3):




Figuur 1.3: Toevoegen aan een bottom-up rood-zwarte boom.

1. *De broer $b$ van $p$ is rood.* De ligging van $c$ ten opzichte van zijn ouder $p$ is hier onbelangrijk. De rode ouder $p$ kan zwart worden, als er aan dezelfde kant een zwarte knoop rood wordt, ter compensatie. Wijzigingen moeten zo lokaal mogelijk blijven, zodat we de zwarte grootouder $g$ rood maken. Maar daardoor nemen we ook een zwarte kleur weg aan de andere kant, wat gelukkig kan opgevangen worden door de rode broer $b$ zwart te maken.

   Als de rood geworden $g$ een zwarte ouder heeft, of geen ouder, dan is het probleem opgelost. Is deze overgrootouder echter rood, dan zijn er opnieuw twee opeenvolgende rode knopen. Het probleem werd dus opgeschoven, in de richting van de wortel. Ofwel wordt het dan opgelost door een van de andere gevallen, ofwel wordt het verder omhoog geschoven. In het slechtste geval gaat dat opschuiven door tot bij de wortel, die dan nog enkel (opnieuw) zwart moet gemaakt worden.

2. *De broer $b$ van $p$ is zwart.* Initieel is $b$ een blad, maar als dit geval later hogerop voorkomt zal $b$ een gewone knoop zijn. Nu nemen we de rode ouder $p$ weg via een rotatie naar rechts, die de rechterdeelboom een extra knoop bezorgt. Omdat $p$ nu bovenaan komt, en dus invloed heeft op twee wegen, neemt die best de zwarte kleur van $g$ over. De overgebrachte knoop $g$ moet dan rood worden om de zwarte hoogte aan die kant niet te ontregelen.

   Dit werkt echter enkel als $c$, $p$ en $g$ op één lijn liggen. (Ga eens na.) Er zijn dus nog twee mogelijkheden, naar gelang dat $c$ een linker- of rechterkind is van $p$:

   (a) *Knoop $c$ is een linkerkind van $p$.* Hier liggen de drie knopen op één lijn, zoals gewenst. We roteren ouder $p$ en grootouder $g$ naar rechts, maken $p$ zwart, en $g$ rood. Er liggen nu twee rode knopen aan weerszijden van een zwarte ouder, zodat het probleem opgelost is.

   (b) *Knoop $c$ is een rechterkind van $p$.* Om de drie knopen op één lijn te krijgen, roteren we $p$ en $c$ naar links. Dan krijgen we het vorige geval (mutatis mutandis).

In totaal zijn er dus hoogstens twee rotaties nodig om de boom te herstellen, eventueel voorafgegaan door $O(\lg n)$ keer opschuiven (beperkt door de hoogte van de boom). Zowel roteren als opschuiven zijn $O(1)$, en het initiële afdalen is $O(\lg n)$, zodat toevoegen steeds $O(\lg n)$ is.

**Verwijderen**

Verwijderen is zoals gewoonlijk wat ingewikkelder. Bovendien wordt de implementatie vlug onoverzichtelijk omdat men steeds moet nagaan of bepaalde knopen wel bestaan. (Bladeren zijn conceptueel zwart, maar er is geen knoop om hun kleur op te slaan!) Daarom gebruikt men in dergelijke gevallen vaak een speciale 'null'knoop, die een blad (nullpointer) voorstelt. De structuur van deze knoop is dezelfde als die van de gewone knopen. Zijn kleur is steeds zwart, zijn andere velden kan men naar believen gebruiken. Bemerk wel dat slechts één nullknoop álle bladeren voorstelt: alle nullwijzers worden door een wijzer naar deze knoop vervangen. Een kind kan nu steeds naar zijn ouder verwijzen, ook als het een blad is. Verwijderen maakt eveneens gebruik van rotaties, waarvan de implementatie ook wat vereenvoudigt met de nullknoop.

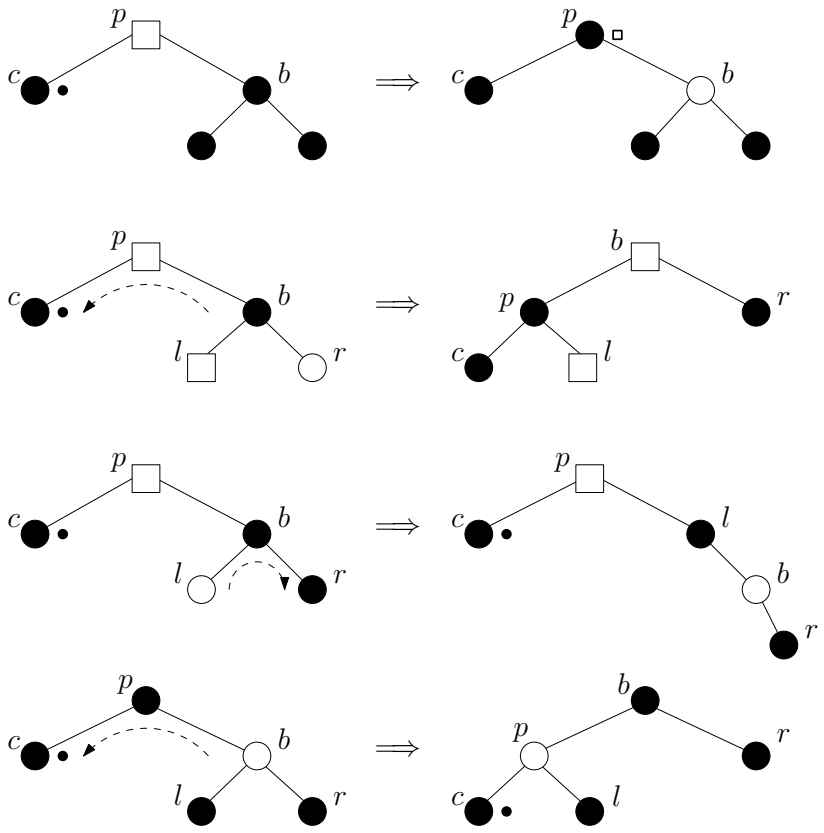
Eerst verwijdert men de knoop (of zijn voorloper of opvolger), zoals bij een gewone zoekboom. Als de fysisch te verwijderen knoop rood is, dan heeft hij geen 'echte' kinderen (enkel zwarte bladeren). Verwijderen is dan eenvoudig, en zonder gevolgen voor de zwarte hoogten. Als de fysisch te verwijderen knoop zwart is, dan heeft hij ofwel geen 'echte' kinderen, ofwel één rood kind. Dat rood kind kan de zwarte kleur van zijn verdwenen ouder overnemen, zodat de zwarte hoogten intact blijven. (Waarom kan dat niet met een eventueel rode ouder?) Als er geen 'echte' kinderen zijn, mag de zwarte kleur toch niet zomaar verdwijnen, of de zwarte hoogten komen in het gedrang. Daarom geeft men die kleur aan een van de 'kinderen' (een zwart blad), dat dan 'dubbel' zwart wordt. Dat is niet in overeenstemming met de definitie van een rood-zwarte boom, zodat we deze anomalie moeten verwijderen.

In bepaalde gevallen zal men het probleem naar boven moeten opschuiven, zodat niet alleen een blad, maar ook een inwendige knoop dubbel zwart kan worden. Het elimineren van een dubbel zwarte kleur moet dus voor een willekeurige knoop $c$ opgelost worden. (Door een

zwarte nullknoop te gebruiken, vervalt het onderscheid tussen een blad en een inwendige knoop. Als ouder krijgt dat blad dan zijn vroegere grootouder, de ouder van de verwijderde knoop.)

Als de dubbel zwarte knoop $c$ wortel is, dan kan de overtollige zwarte kleur gewoon verdwijnen, omdat de wortel in geen enkele zwarte hoogte meetelt. Stel dus dat $c$ geen wortel is, met ouder $p$. Dan zijn er *acht* mogelijke gevallen, die uiteenvallen in twee groepen van vier, naar gelang dat $c$ een linker- of rechterkind van $p$ is. De operaties voor beide groepen zijn elkaars spiegelbeeld, zodat we slechts één groep moeten bespreken.

*Onderstel dat $c$ een linkerkind is van $p$.* We kunnen het overtollige zwart elimineren ofwel door het samen met een vrijgekomen zwarte kleur uit de rechterdeelboom naar boven op te schuiven (dat vereist dat er rechts een knoop rood kan worden), ofwel door de linkerdeelboom een extra knoop te bezorgen, via een rotatie naar links, en die zwart te maken. Beide gevallen vereisen dat $c$ een zwarte broer heeft. Er zijn dus twee hoofdgevallen, naar gelang van de kleur van deze broer $b$ (figuur 1.4):




Figuur 1.4: Verwijderen uit een bottom-up rood-zwarte boom.

1. *Broer $b$ van $c$ is zwart.* De kleur van hun ouder $p$ is willekeurig. Of $b$ zijn kleur afstaat, of roteert, hangt af van de kleur van zijn kinderen (eventueel zwarte bladeren). Om rood te kunnen worden moet $b$ immers twee zwarte kinderen hebben, en wanneer $b$

roteert moet zijn verdwenen zwarte kleur gecompenseerd worden door een van zijn rode kinderen zwart te maken. Er zijn dus nog drie gevallen, naar gelang van de kleur van de kinderen van $b$:

(a) *Broer $b$ heeft twee zwarte kinderen.* Dan kan $b$ rood worden, en de vrijgekomen zwarte kleur schuiven we samen met het overtollige zwart van $c$ door naar hun ouder $p$. De zwarte hoogte op beide takken blijft aldus ongewijzigd. Een rode ouder $p$ kan zwart worden, zodat alles in orde is.

Een zwarte ouder $p$ wordt echter dubbel zwart, zodat het probleem naar boven opgeschoven werd. Ofwel wordt het daar opgelost door een van de andere gevallen, ofwel wordt het verder opgeschoven, desnoods tot bij de wortel, waar dat overtollige zwart gewoon verdwijnt.

(b) *Broer $b$ heeft een rood rechterkind.* De kleur van het linkerkind $l$ van $b$ is willekeurig. Nu bezorgen we de linkerdeelboom een extra knoop (via rotatie naar links), zodat die de overtollige zwarte kleur van $c$ kan overnemen. Daarbij verliest de rechterdeelboom echter een zwarte knoop. Gelukkig is er het rode rechterkind $r$ van $b$, dat zwart kan worden. (Dit werkt niet met een rood linkerkind.) We roteren dus $p$ en $b$ naar links, en $b$ krijgt de (willekeurige) kleur van $p$, die zelf zwart wordt, net als $r$. Daarmee is het probleem van de baan.

(c) *Broer $b$ heeft een zwart rechterkind en een rood linkerkind.* We herleiden dit tot het vorige geval door de (nieuwe) broer van $c$ een rood rechterkind te bezorgen. Daartoe roteren we $b$ en zijn rood linkerkind $l$ naar rechts, maken $b$ rood, en $l$ zwart. De nieuwe broer $l$ van $c$ is zwart, en heeft nu een rood rechterkind $b$. Dat is precies het vorige geval (mutatis mutandis).

2. *Broer $b$ van $c$ is rood.* Hun ouder $p$ moet dan zwart zijn. We herleiden dit tot het eerste hoofdgeval, door $c$ een nieuwe zwarte broer te bezorgen. Dat moet een *inwendige* knoop zijn, want we hebben zijn kinderen nodig. Daarvoor gebruiken we het zwarte linkerkind $l$ van broer $b$, die immers twee inwendige zwarte kinderen heeft. (De zwarte hoogte van $b$ is minstens twee, omdat $c$ dubbel zwart is.) We roteren ouder $p$ en broer $b$ naar links, maken $b$ zwart, en $p$ rood. De dubbel zwarte knoop $c$ heeft nu een nieuwe zwarte broer $l$, en een rood geworden ouder $p$. Dat is dus het eerste hoofdgeval.

Het eerste hoofdgeval lost het probleem op met één of twee rotaties, of schuift het naar boven. Het tweede hoofdgeval verricht één rotatie, om bij het eerste hoofdgeval terecht te komen. Daar wordt het ofwel opgelost met één of twee bijkomende rotaties, ofwel met twee kleurwijzigingen (zonder op te schuiven, omdat de ouder van $c$ rood is, en dus probleemloos zwart kan worden). Een naar boven opgeschoven probleem wordt ofwel daar opgelost met maximaal drie rotaties, of nog verder opgeschoven.

Verwijderen heeft dus hoogstens drie rotaties nodig om de boom te herstellen, eventueel voorafgegaan door $O(\lg n)$ keer opschuiven (beperkt door de hoogte van de boom). Zowel roteren als opschuiven zijn $O(1)$, en het initiële afdalen is $O(\lg n)$, zodat ook verwijderen steeds $O(\lg n)$ is.

#### 1.2.3.3 Top-down rood-zwarte bomen

Bij deze bomen moeten we nooit naar boven terugkeren, zodat ouderwijzers noch stapel vereist zijn.

**Toevoegen**

Net zoals bij de bottom-up versie vervangt de nieuwe knoop een blad (nullwijzer), en maken we hem rood. Dat geeft enkel problemen als zijn ouder ook rood is. We hebben gezien dat dit met rotaties en kleurwijzigingen kan opgelost worden als de oom van de nieuwe knoop zwart is. Bij een rode oom echter moesten we soms terugkeren in de boom, en dat is hier onmogelijk. Daarom zullen we er tijdens het afdalen voor zorgen dat deze oom steeds zwart zal zijn.

Op de weg naar beneden mogen we daarom geen rode broers toelaten, want de nieuwe (rode) knoop zou een kind van een van beide kunnen worden. Wanneer we dus voorbij een *zwarte knoop met twee rode kinderen* komen, dan maken we die knoop rood en zijn kinderen zwart. Als die nieuwe rode knoop zelf een rode ouder heeft is er een probleem, dat echter weer met rotaties en kleurwijzigingen kan opgelost worden, omdat zijn oom gegarandeerd zwart is. Rode broers hoger op de zoekweg werden immers op dezelfde manier geëlimineerd. Daarbij kunnen nieuwe rode broers gecreëerd worden, maar die liggen dan hoger in de boom, zodat ze hier geen problemen meer veroorzaken. (Bemerk trouwens dat door het zwart maken van de rode broers, de zoektocht naar beneden tenminste twee generaties zwarte knopen zal ontmoeten.)

Als er onderweg een rotatie moet gebeuren, zijn daar vier generaties knopen bij betrokken, want om twee opeenvolgende rode knopen (ouder en kind) te elimineren moeten ouder en grootouder roteren, waarbij ook de overgrootouder moet aangepast worden (als die bestaat). Tijdens de afdaling moeten er dus steeds (wijzers naar) vier opeenvolgende knopen bijgehouden worden.

Toevoegen daalt enkel in de boom, waarbij rotaties en kleurwijzigingen kunnen gebeuren. De performantie is dus steeds $O(\lg n)$.

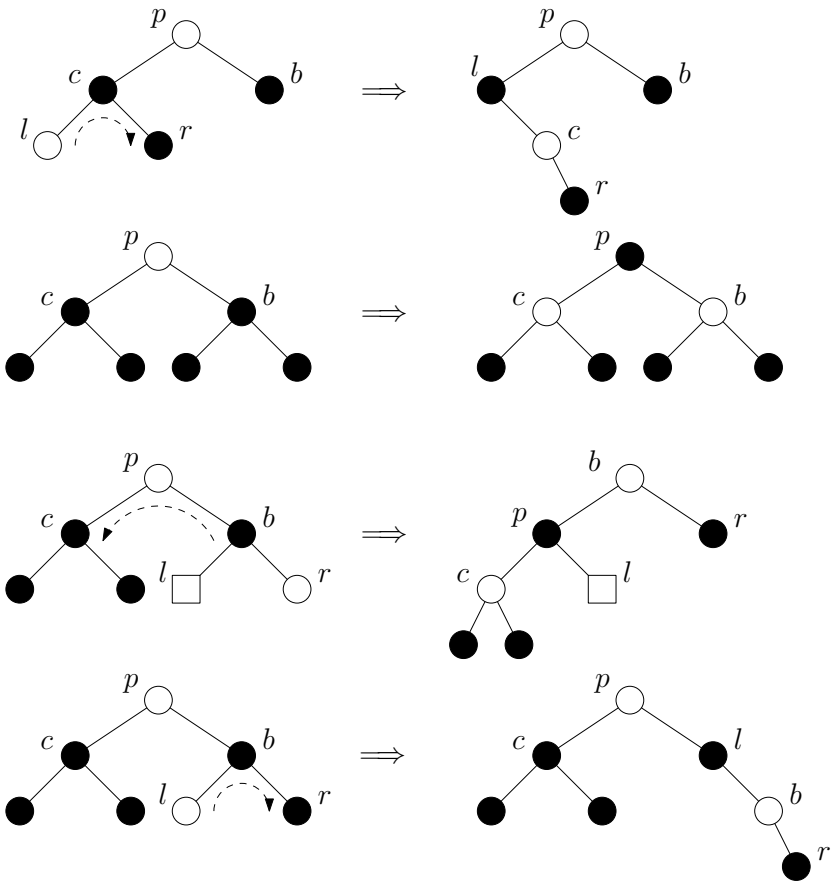
**Verwijderen**

Zoals bij een gewone binaire zoekboom moet een te verwijderen knoop met twee echte kinderen vervangen worden door zijn voorloper of opvolger. De zwarte hoogte van de fysisch te verwijderen knoop is dus één, omdat minstens een van zijn kinderen een zwart blad (nullpointer) is. Om geen problemen te krijgen met de zwarte hoogte, moeten we ervoor zorgen dat deze knoop rood is. Maar dan is ook zijn tweede kind steeds een zwart blad, want een rode knoop mag geen rood kind hebben.

Omdat we niet weten wanneer we deze knoop op de dalende zoekweg zullen vinden, maken we elke volgende knoop op die weg rood. Stel dus dat we tijdens het afdalen in een rode of rood gemaakte knoop $p$ beland zijn, en dat we nog verder moeten afdalen. We zullen dan bij een (uiteraard) zwart kind $c$ terechtkomen, dat rood moet worden, waarna we opnieuw in de beginsituatie zijn. (We zullen ons later bezighouden met het begin, bij de wortel). Naar gelang dat $c$ een linker- of rechterkind is van $p$ zijn er twee groepen van mogelijke gevallen te onderscheiden. Opnieuw zijn de operaties elkaars spiegelbeeld, zodat we slechts één groep

moeten bespreken.

*Stel dat $c$ een linkerkind is van $p$.* Ofwel kan het probleem opgelost worden via een van de rode kinderen van $c$, ofwel moet er een beroep gedaan worden op broer $b$. Er zijn dus twee hoofdgevallen (zie figuur 1.5):




Figuur 1.5: Verwijderen uit een top-down rood-zwarte boom.

1. *Knoop $c$ heeft minstens één rood kind.* Als we geluk hebben moeten we verder naar een rood kind, zodat we weer in de beginsituatie zijn.

   Als we echter naar een zwart kind moeten afdalen, of als $c$ de (fysisch) te verwijderen knoop is, dan maken we $c$ rood door hem samen met zijn rood kind te roteren, en hun beide kleuren om te keren. We zijn dan opnieuw in de beginsituatie, of we kunnen $c$ zonder meer verwijderen.

2. *Knoop $c$ heeft twee zwarte kinderen.* (Dat geldt ook als $c$ geen inwendige kinderen heeft.) Om $c$ rood te maken zullen we nu een beroep moeten doen op zijn (zwarte) broer $b$. Ofwel geeft $c$ zijn zwarte kleur door aan ouder $p$, maar dan moet $b$ ook rood kunnen worden. Ofwel brengen we een knoop over (via rotatie) om de zwarte kleur van $c$ over

te nemen, en zullen we het verlies van een zwarte knoop aan de andere kant moeten compenseren. (Bemerk de analogie met het elimineren van een dubbel zwarte knoop bij bottom-up verwijderen.) Er zijn dus nog drie mogelijkheden, afhankelijk van de kleur van de kinderen van $b$ (die nooit een blad is):

(a) *Broer $b$ heeft twee zwarte kinderen.* Dan maken we $b$ en $c$ rood, en $p$ zwart.

(b) *Broer $b$ heeft een rood rechterkind.* De kleur van het linkerkind is willekeurig. Omdat $c$ rood moet worden, is er een bijkomende zwarte knoop in de linkse deelboom nodig, die we bekomen via een rotatie naar links. Daarbij verliest de rechterdeelboom een zwarte knoop, waarvan de kleur bewaard wordt door het rood rechterkind $r$ van $b$ zwart te maken. We roteren dus $p$ en $b$ naar links, maken $c$ en $b$ rood, en zowel $p$ als $r$ zwart.

(c) *Broer $b$ heeft een zwart rechterkind en een rood linkerkind.* In principe zorgen we ervoor dat de (nieuwe) broer van $c$ een rood rechterkind krijgt, zodat we in het vorige geval terechtkomen. We roteren dus het rood linkerkind $l$ en $b$ naar rechts en keren hun kleuren om. We zijn dan in het vorige geval, dat $p$ en $l$ naar links roteert, en de kleuren van $p$, $c$, $b$ en $l$ omkeert.

Het kan echter eenvoudiger, aangezien de kleuren van $b$ en $l$ tweemaal omkeren, zodat we ze beter onveranderd laten. Het volstaat dus om $l$ en $b$ naar rechts te roteren, dan $p$ en $l$ naar links te roteren, en tenslotte $c$ rood te maken en $p$ zwart.

Ofwel kan de rood geworden knoop $c$ nu (fysisch) verwijderd worden, ofwel moeten we verder afdalen, zodat we weer in de beginsituatie zijn.

Blijft de vraag hoe we beginnen bij de (zwarte) wortel. Die kunnen we beschouwen als een zwarte knoop $c$ waarheen we moeten vanuit een fictieve rode ouder. Afhankelijk van de kinderen van de wortel krijgen we dan dezelfde hoofdgevallen als hierboven. Het tweede hoofdgeval (beide kinderen zwart) is echter eenvoudiger, omdat de wortel steeds (tijdelijk) rood mag worden, want hij heeft geen broer waarmee hij rekening moet houden.

Net als toevoegen gebeurt verwijderen door af te dalen in de boom, waarbij rotaties en kleurwijzigingen kunnen gebeuren. De performantie is dus ook steeds $O(\lg n)$.

### 1.2.4 Vereenvoudigde rood-zwarte bomen

De implementatie van rood-zwarte bomen is omslachtig door de talrijke speciale gevallen. Daarom gebruikt men soms eenvoudiger varianten. Zo mag bij een AA-boom (Andersson, 1993) enkel een *rechterkind* rood zijn. Ook 'Binary B-trees' hadden reeds dezelfde beperking ten opzichte van 'Symmetric binary B-trees', zonder echter de rood-zwarte terminologie te gebruiken. Deze beperkingen reduceren het aantal gevallen, maar behouden toch de (asymptotische) efficiëntie.

## 1.3 Splay trees

In tegenstelling tot evenwichtige zoekbomen garanderen splay trees (Sleator en Tarjan, 1985) niet dat elke afzonderlijke operatie efficiënt is. In plaats daarvan zorgen ze ervoor dat elke *opeenvolgende reeks* operaties steeds efficiënt is, zodat uitgemiddeld over de reeks de performantie per operatie goed uitvalt. (Dat gemiddelde is dus *geen* verwachtingswaarde, zoals bij gewone zoekbomen. Men noemt dit 'amortized efficiency', want uitgesmeerd over de reeks.) Aangezien men efficiënte zoekbomen niet gebruikt om er slechts enkele operaties op uit te voeren, vormen ze een interessant alternatief voor de meer ingewikkelde rood-zwarte bomen.

Concreter, wanneer men een reeks van $m$ operaties verricht op een initieel ledige splay tree, waaronder $n$ keer toevoegen (zodat $m \geq n$), dan is de performantie van deze reeks $O(m \lg n)$, ook in het slechtste geval. Uitgemiddeld over de reeks is dat dus $O(\lg n)$ per operatie. (Voor een individuele operatie is dat slechter dan de $O(\lg n)$ in het slechtste geval bij rood-zwarte bomen, maar beter dan een verwachtingswaarde van $O(\lg n)$, aangezien deze niet uitsluit dat elke operatie in een reeks slecht kan zijn.)

Aangezien individuele operaties nu inefficiënt mogen zijn, zal men na een dergelijke operatie de vorm van de boom moeten aanpassen, zoniet zou het herhalen van diezelfde operatie een slechte reeks opleveren. De basisidee van een splay tree bestaat erin om elke knoop die gezocht, toegevoegd of verwijderd wordt, tot *wortel* van de boom te maken, zodat volgende operaties op die knoop (of zijn omgeving) zeer efficiënt worden. (In de praktijk blijkt immers de kans groot dat men dezelfde knoop kort daarna opnieuw nodig heeft.) Dit is vergelijkbaar met gelinkte lijsten die zelf hun gemiddelde zoektijd verbeteren, door elk gevonden element vooraan de lijst te plaatsen. (Zie Gegevensstructuren I in Algoritmen I.) Een knoop wortel maken is wel iets ingewikkelder, omdat de boom een binaire zoekboom moet blijven. Deze zogenaamde *splay-operatie* gebeurt dus opnieuw met rotaties.

De weg naar een diep liggende knoop bevat veel knopen die ook relatief diep liggen. Terwijl we die knoop wortel maken, zullen we tegelijkertijd de structuur van de boom zo moeten aanpassen, dat ook de toegangstijd van deze knopen verbetert, zoniet blijft de kans bestaan dat een reeks operaties inefficiënt is.

Bij splay trees moet er geen hoogte van knopen of andere informatie over het evenwicht van de boom (zoals kleuren) bijgehouden worden. Dit spaart geheugen uit, en zorgt meestal voor meer eenvoudige implementaties. Bemerk echter dat ook zoeken de boom kan herstructureren. Net zoals bij rood-zwarte bomen bestaan er 'top-down' en 'bottom-up' versies van splay trees.

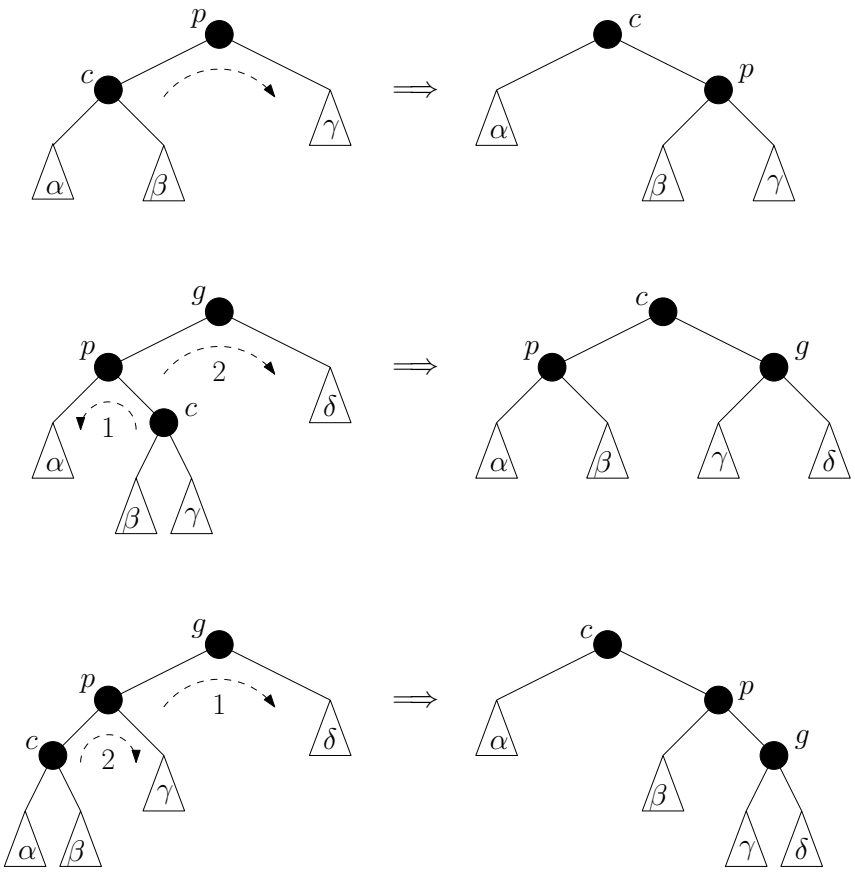
### 1.3.1 Bottom-up splay trees

Eerst zoeken we de knoop zoals bij een gewone zoekboom, daarna gebeurt de splay-operatie van onder naar boven. We hebben dus opnieuw ouderwijzers of een stapel nodig.

Een knoop kan naar boven gebracht worden door hem telkens met zijn ouder op de zoekweg te roteren, maar dat is niet voldoende om steeds een performantie van $O(m \log n)$ te verzekeren. Weliswaar wordt de knoop zelf wortel, maar de situatie voor de andere knopen op de zoekweg is nauwelijks verbeterd. Het is dan ook mogelijk om een reeks van $m$ operaties te vinden

die $\Theta(mn)$ duurt. (Voeg bijvoorbeeld de getallen $1, 2, 3, \ldots, n$ toe aan een ledige boom. Het resultaat is een gelinkte lijst, efficiënt opgebouwd in $O(n)$. De knopen in dezelfde volgorde opzoeken vereist echter $\Theta(n^2)$ bewerkingen, en bovendien is de boom na afloop weer precies dezelfde gelinkte lijst, waarna we van vooraf aan kunnen herbeginnen.)

We zullen de rotaties dus zorgvuldiger moeten aanpakken. Nog steeds gebeuren die van onder naar boven langs de zoekweg voor een knoop $c$, tot die wortel is. Maar we onderscheiden nu de volgende mogelijkheden (zie figuur 1.6):




Figuur 1.6: Bottom-up splay.

1. *De ouder van $c$ is wortel.* Dan roteren we beide knopen. Dit eenvoudig geval noemt men 'zig'.

2. *Knoop $c$ heeft nog een grootouder.* Er zijn dan *vier* gevallen, die uiteenvallen in twee groepen van twee, naar gelang dat ouder $p$ een linker- of rechterkind is van grootouder $g$. Eens te meer zijn de operaties elkaars spiegelbeeld, en behandelen we slechts één groep.

   *Stel dat $p$ linkerkind is van $g$.* Dan zijn er twee mogelijkheden:

   (a) *Knoop $c$ is rechterkind van $p$.* We voeren dan twee rotaties uit, een in elke richting: eerst $p$ en $c$ naar links, en dan $g$ en $c$ naar rechts. Daarom heet dit geval 'zig-zag'.

(b) *Knoop c is linkerkind van p.* Opnieuw roteren we tweemaal, telkens naar rechts: maar nu eerst $g$ en $p$ (de *bovenste* knopen), daarna $p$ en $c$. (Bemerk dat dit het enige verschil is met de vorige methode.) Omdat de drie knopen op één lijn lagen noemt men dit geval 'zig-zig'.

Hoewel het effect moeilijk te zien is op kleine voorbeelden, wordt de diepte van de meeste knopen op de toegangsweg nu ongeveer gehalveerd. Slechts enkele ondiepe knopen zakken hoogstens twee niveau's. Het is instructief om deze strategie eens toe te passen op het voorbeeld van hierboven.

Het specifiek gedrag van splay trees wordt slechts zichtbaar bij grotere bomen: inefficiënte operaties worden gevolgd door rotaties die gunstig zijn voor de volgende operaties, terwijl er na efficiënte operaties minder goed of zelfs slecht uitvallende rotaties kunnen gebeuren. Maar steeds blijft de performantie van de reeks goed.

Met deze splay-operatie verlopen de woordenboekoperaties als volgt:

- Zoeken gebeurt zoals bij een gewone zoekboom. Daarna wordt de laatste knoop op de zoekweg wortel, via een splay-operatie. Deze wortel bevat dan ofwel de gezochte sleutel, ofwel zijn voorloper of opvolger.
- Ook toevoegen gebeurt zoals bij een gewone binaire zoekboom. Daarna wordt de nieuwe knoop wortel, via een splay-operatie.

  Een alternatieve manier die (geamortiseerd) iets trager uitvalt, zoekt eerst de toe te voegen sleutel, zoals hierboven. De wortel bevat daarna ofwel de sleutel, ofwel zijn voorloper of opvolger. Als de sleutel niet aanwezig is (duplicaten voegen we immers niet toe), slaan we die op in een nieuwe wortelknoop, met als linkerkind (rechterkind) de voorloper (opvolger) met zijn linkse (rechtse) deelboom, en als rechterkind (linkerkind) zijn rechtse (linkse) deelboom.
- Ook verwijderen kan gebeuren zoals bij een gewone zoekboom. Daarna wordt de *ouder* van de fysisch verwijderde knoop wortel, via een splay-operatie. Als de sleutel niet gevonden wordt, zal toch weer de laatste knoop op de zoekweg wortel worden.

  Een alternatieve methode die (geamortiseerd) iets trager uitvalt, zoekt eerst de knoop met de te verwijderen sleutel, via een splay-operatie. Als daarna de wortel de gezochte sleutel bevat verwijdert men hem, en worden zijn deelbomen samengevoegd. Samenvoegen van splay trees (een 'join'-operatie) gebeurt door eerst het grootste (kleinste) element in de linkse (rechtse) deelboom te zoeken en dus wortel te maken, via een splay-operatie. Deze wortel heeft dan geen rechterkind (linkerkind), en op die plaats komt de andere deelboom terecht.

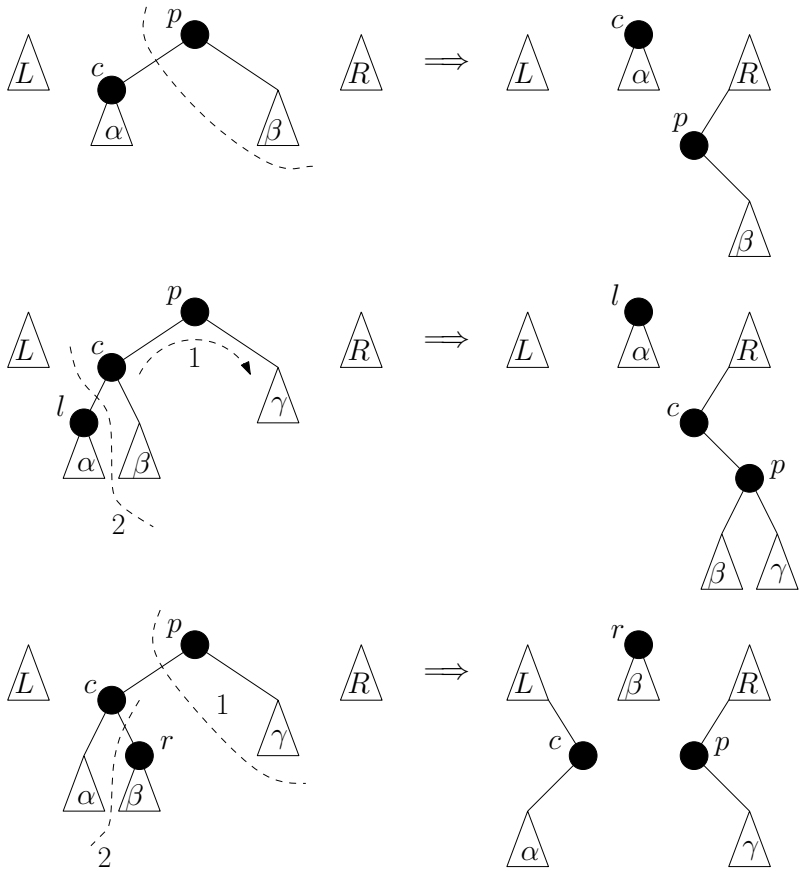
### 1.3.2 Top-down splay trees

De 'top-down' versie verricht de splay-operatie tijdens de afdaling, zodat de gezochte knoop wortel zal zijn als we hem bereiken. Er zijn dus geen ouderwijzers of stapel nodig, en in de praktijk blijkt deze versie eenvoudiger te implementeren en iets efficiënter.

Tijdens de afdaling wordt de boom in drie zoekbomen opgedeeld, zodanig dat alle sleutels in de linkse boom $L$ kleiner zijn dan de sleutels in de middelste boom $M$, en alle sleutels in de rechtse boom $R$ groter zijn dan die in $M$. Initieel is $M$ de oorspronkelijke boom, en zijn $L$ en $R$ ledig. Het zoeken (naar een sleutel, naar de plaats voor een nieuwe sleutel, of naar een te verwijderen sleutel) begint dan bij de wortel van $M$, en men zorgt ervoor dat de huidige knoop op de zoekweg steeds wortel blijft van $M$.

Stel dat we op weg naar beneden bij knoop $p$ terechtgekomen zijn (die dan wortel van $M$ geworden is), en dat we nog verder moeten (en kunnen). Afhankelijk van de richting die we uit moeten zijn er twee groepen van mogelijke gevallen, met eens te meer gespiegelde operaties, zodat we slechts één groep moeten behandelen.

*Stel dat we vanuit $p$ naar zijn linkerkind $c$ moeten.* Dan zijn er twee hoofdgevallen (zie figuur 1.7):




Figuur 1.7: Top-down splay.

1. *Linkerkind $c$ is de laatste knoop op de zoekweg.* Dat doet zich voor ofwel als we $c$ zoeken, ofwel omdat $c$ minstens één kind mist, en we net die kant op moeten. (Dat is dus zeker het geval als $c$ een blad is.) Knoop $p$ wordt dan het nieuwe kleinste element in $R$ (de rechtse deelboom van $p$ gaat mee), en de linkse deelboom van $p$ wordt de nieuwe $M$

(met $c$ als wortel).

Dit eenvoudigste geval noemt men opnieuw 'zig'.

2. *Linkerkind $c$ is niet de laatste knoop op de zoekweg.* Er zijn dan nog twee gevallen, naar gelang dat we afdalen naar het linkerkind $l$ of het rechterkind $r$ van $c$:

   (a) *Naar het linkerkind $l$ van $c$.* Dan *roteren* we eerst $p$ en $c$ naar rechts (de *bovenste* knopen eerst, zoals bij de bottom-up versie), daarna wordt $c$ het nieuwe kleinste element van $R$ (de nieuwe rechtse deelboom van $c$ gaat mee). De linkse deelboom van $c$ wordt de nieuwe $M$ (met $l$ als wortel).

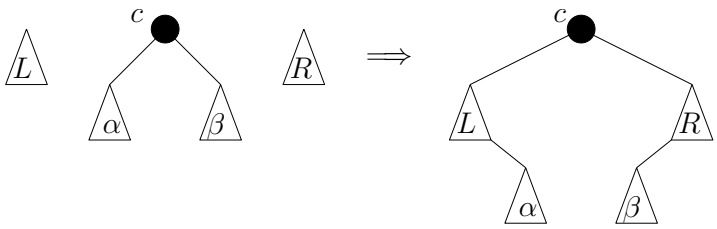
   Aangezien de drie knopen $p$, $c$ en $l$ op één lijn lagen noemt men dit geval opnieuw 'zig-zig'.

   (b) *Naar het rechterkind $r$ van $c$.* Dan wordt $p$ het nieuwe kleinste element van $R$ (de rechtse deelboom van $p$ gaat mee). Daarna wordt $c$ het nieuwe grootste element in $L$ (de linkse deelboom van $c$ gaat mee), en de rechtse deelboom van $c$ wordt de nieuwe $M$ (met $r$ als wortel).

   Hier lagen de drie knopen $p$, $c$ en $r$ niet op één lijn, zodat dit geval weer 'zig-zag' heet.

   Aangezien er geen rotatie gebeurt, kan men dit geval voor de eenvoud ook behandelen als een 'zig'. Dan wordt $p$ nog steeds het nieuwe kleinste element van $R$ (de rechtse deelboom van $p$ gaat mee), maar de linkse deelboom van $p$ wordt nu de nieuwe $M$ (met $c$ als wortel). Het voordeel van deze vereenvoudiging is dat er wat minder gevallen te onderscheiden zijn, het nadeel is dat we slechts één niveau in de boom zijn afgedaald in plaats van twee (er zijn dus meer iteraties nodig).

Als uiteindelijk de gezochte knoop $c$ wortel van $M$ geworden is, voltooien we de splay-operatie door de drie deelbomen samen te voegen tot één boom met wortel $c$ (zie figuur 1.8). Alle sleutels in de linkse deelboom van $c$ zijn groter dan de sleutels in $L$. We kunnen deze deelboom dus volledig onderbrengen in $L$, met zijn wortel op de meest rechtste plaats in $L$. En analoog komt de rechtse deelboom van $c$ terecht in $R$, met zijn wortel op de meest linkse plaats in $R$. Tenslotte wordt deze gewijzigde $L$ de nieuwe linkerdeelboom van $c$, en de gewijzigde $R$ zijn nieuwe rechterdeelboom.




Figuur 1.8: Samenvoegen na top-down splay.

De woordenboekoperaties verlopen nu als volgt:

- Zoeken maakt de knoop met de sleutel wortel (en als de sleutel niet gevonden wordt, dan de knoop met zijn voorloper of opvolger).
- Toevoegen gebeurt analoog met de alternatieve versie van bottom-up toevoegen. De voorloper of opvolger van de nieuwe sleutel wordt wortel (duplicaten worden niet toegevoegd), en de nieuwe knoop met de toegevoegde sleutel krijgt als linkerkind (rechterkind) de voorloper (opvolger) met zijn linkse (rechtse) deelboom, en als rechterkind (linkerkind) zijn rechtse (linkse) deelboom.
- Ook verwijderen gebeurt analoog met de alternatieve versie van bottom-up verwijderen. Eerst zoekt men de sleutel, en als hij gevonden wordt, en dus bij de wortel staat, dan verwijdert men die wortel en voegt zijn deelbomen samen (weer via een 'join'-operatie).

Na een top-down operatie is de boom niet noodzakelijk dezelfde als na de analoge bottom-up operatie, maar de geamortiseerde efficiëntie van alle operaties is opnieuw $O(\lg n)$.

### 1.3.3 Performantie van splay trees

De performantie-analyse van splay trees is niet zo eenvoudig, omdat de vorm van de bomen voortdurend verandert. Om aan te tonen dat elke reeks van $m$ operaties op een splay tree met maximaal $n$ knopen een performantie van $O(m \lg n)$ heeft, en dus een geamortiseerde efficiëntie van $O(\lg n)$ per operatie, gebruikt men een *potentiaalfunctie* $\Phi$. Omdat de vorm van de boom kan veranderen, krijgt elke mogelijke vorm een reëel getal toegewezen, zijn potentiaal. Operaties op de boom kunnen zijn vorm en dus ook zijn potentiaal wijzigen. Het is de bedoeling dat efficiënte operaties, die minder tijd gebruiken dan de geamortiseerde tijd per operatie, de potentiaal verhogen. Inefficiënte operaties spreken deze reserve aan en doen de potentiaal dalen. De geamortiseerde tijd van een individuele operatie wordt gedefinieerd als de som van haar werkelijke tijd en de toename van de potentiaal. Als $t_i$ de werkelijke tijd van de $i$-de operatie voorstelt, $a_i$ haar geamortiseerde tijd, en $\Phi_i$ de potentiaal na deze operatie, dan is $a_i = t_i + \Phi_i - \Phi_{i-1}$. De geamortiseerde tijd van een reeks van $m$ operaties is de som van de individuele geamortiseerde tijden:

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} (t_i + \Phi_i - \Phi_{i-1})$$

In deze som komen de meeste potentialen met tegengestelde tekens voor, zodat

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} t_i + \Phi_m - \Phi_0$$

waarbij $\Phi_0$ de beginpotentiaal voorstelt. Als we de potentiaalfunctie zo kiezen dat de eindpotentiaal niet kleiner is dan de beginpotentiaal, dan is de totale geamortiseerde tijd een *bovengrens* voor de werkelijke tijd van de reeks operaties.

Een geschikte potentiaalfunctie vinden die de berekeningen eenvoudig houdt en toch een zinvol resultaat oplevert, is niet altijd eenvoudig, en vaak moet men verschillende mogelijkheden uitproberen. De eenvoudigste keuze voor dit geval geeft elke knoop $i$ een gewicht $s_i$ gelijk aan

het aantal knopen in de deelboom waarvan hij wortel is (de knoop zelf inbegrepen), en neemt de som van de logaritmen van al deze gewichten:

$$\Phi = \sum_{i=1}^{n} \lg s_i$$

(Met een andere geschikte keuze voor de gewichten kan men bijkomende eigenschappen aantonen.) Voor de eenvoud noteert men $\lg s_i$ als $r_i$, de *rang* van de knoop $i$. De potentiaal van een ledige boom onderstellen we nul.

De performantie-analyse van bottom-up en top-down splay trees is volledig analoog. We beperken ons daarom tot de bottom-up versie. De performantie van een splay-operatie op een knoop is evenredig met de diepte van die knoop, en dus met het aantal uitgevoerde (enkelvoudige) rotaties. (Een zig verricht één rotatie, zowel zig-zag als zig-zig verrichten er twee.)

We zullen trachten aan te tonen dat de geamortiseerde tijd voor het zoeken naar een knoop $c$ gevolgd door een splay-operatie op die knoop gelijk is aan $O(1 + 3(r_w - r_c))$, waarbij $r_w$ de rang van de wortel en $r_c$ de rang van knoop $c$ in de originele boom voorstelt. (In de uiteindelijke boom is de rang van $c$ natuurlijk gelijk aan $r_w$.)

Als $c$ de wortel is, dan moeten we niet afdalen in de boom en zijn er geen rotaties nodig. De werkelijke tijd is dan $O(1)$, en de potentiaal blijft gelijk, zodat dit triviaal geval alvast aan de eigenschap voldoet. Als we wel moeten afdalen, dan is het aantal enkelvoudige rotaties dat de splay-operatie zal uitvoeren gelijk aan de diepte van de knoop. De werkelijke tijd om één niveau te dalen en later de overeenkomstige (enkelvoudige) rotatie uit te voeren is $O(1)$. Omdat er een onbekend aantal zig-zags en/of zig-zigs gebeuren, eventueel gevolgd door één zig, bepalen we eerst de geamortiseerde tijd $a$ voor de drie gevallen apart. Aangezien telkens dezelfde knoop $c$ één of twee niveau's stijgt, trachten we resultaten te bekomen waarin enkel de rang van $c$ vóór en na de operatie voorkomt. De geamortiseerde tijd van de volledige splay-operatie is immers de som van de afzonderlijke tijden, en omdat de rang van $c$ na een operatie gelijk is aan de rang van $c$ vóór de volgende operatie, hopen we dat de meeste termen in die som wegvallen.

Het gewicht en de rang van knoop $i$ vóór een operatie noemen we $s_i$ en $r_i$, na de operatie $s'_i$ en $r'_i$. Vóór de operatie is $p$ de ouder van $c$, en $g$ de eventuele grootouder.

- Een zig kan enkel de rang van $c$ en $p$ wijzigen, zodat

  $$a = 1 + r'_c + r'_p - r_c - r_p$$

  of ook, aangezien $r'_p < r_p$

  $$a < 1 + r'_c - r_c$$

- De dubbele rotatie van een zig-zag kan enkel de rang van $c$, $p$ en $g$ wijzigen, zodat

  $$a = 2 + r'_c + r'_p + r'_g - r_c - r_p - r_g$$

  of, aangezien $r'_c = r_g$

  $$a = 2 + r'_p + r'_g - r_c - r_p$$

Om $r'_p + r'_g$ kwijt te spelen merken we op dat $s'_c > s'_p + s'_g$, en dus $r'_c > \lg(s'_p + s'_g)$. Om dit verder te vereenvoudigen, maken we gebruik van het feit dat de logaritmische functie concaaf is: voor positieve $a$ en $b$ geldt dat $\lg((a+b)/2) \geq (\lg a + \lg b)/2$. Dus is $\lg((s'_p + s'_g)/2) \geq (\lg s'_p + \lg s'_g)/2 = (r'_p + r'_g)/2$ zodat $r'_c > 1 + (r'_p + r'_g)/2$ of ook $r'_p + r'_g < 2r'_c - 2$. Dus wordt

$$a < 2r'_c - r_c - r_p$$

en omdat $r_p > r_c$ bekomen we uiteindelijk dat

$$a < 2(r'_c - r_c)$$

- Ook de dubbele rotatie van een zig-zig kan enkel de rang van $c$, $p$ en $g$ wijzigen, zodat opnieuw

  $$a = 2 + r'_c + r'_p + r'_g - r_c - r_p - r_g$$

  en weer met $r'_c = r_g$ wordt dit

  $$a = 2 + r'_p + r'_g - r_c - r_p$$

  Om dit te vereenvoudigen maken we gebruik van $s'_c > s'_g + s_c$, en met dezelfde eigenschap als hierboven wordt $r'_c > \lg(s'_g + s_c) \geq 1 + (r'_g + r_c)/2$ zodat $r'_g < 2r'_c - r_c - 2$. We krijgen dan

  $$a < r'_p + 2r'_c - 2r_c - r_p$$

  Met $r'_c > r'_p$ en $r_p > r_c$ bekomen we tenslotte dat

  $$a < 3(r'_c - r_c)$$

De bovengrenzen voor de drie soorten operaties bevatten dezelfde positieve term $r'_c - r_c$ (want $r'_c > r_c$), maar met verschillende coëfficiënten. De totale geamortiseerde tijd is een som van dergelijke tijden, die we maar kunnen vereenvoudigen als de coëfficiënten gelijk zijn. Omdat het hier gaat om bovengrenzen, maken we alle coëfficiënten gelijk aan de grootste waarde, die van de zig-zig operatie. In de som vallen de meeste termen nu weg, behalve de rang van $c$ vóór en na de volledige splay-operatie. Op het einde is $c$ wortel, met als rang inderdaad $r_w$. Daarmee is de eigenschap aangetoond.

We kunnen nu de geamortiseerde tijden van de woordenboekoperaties op een (bottom-up) splay tree met $n$ knopen bepalen:

- De geamortiseerde tijd voor zoeken gevolgd door een splay-operatie is $O(1 + 3\lg n)$, want $s_w$ is gelijk aan $n$, en voor een blad is $s_c$ één.
- Toevoegen (we nemen de snelste versie) daalt eerst af net als zoeken, en wijzigt met de nieuwe knoop enkel de rang van de knopen $p_1, p_2, \ldots, p_k$ op de zoekweg. Als $s_{p_i}$ het gewicht van knoop $p_i$ vóór het toevoegen van de nieuwe knoop voorstelt, en $s'_{p_i}$ het gewicht erna, dan is de potentiaaltoename gelijk aan

  $$\lg\left(\frac{s'_{p_1}}{s_{p_1}}\right) + \lg\left(\frac{s'_{p_2}}{s_{p_2}}\right) + \cdots + \lg\left(\frac{s'_{p_k}}{s_{p_k}}\right) = \lg\left(\frac{s'_{p_1}}{s_{p_1}}\frac{s'_{p_2}}{s_{p_2}}\cdots\frac{s'_{p_k}}{s_{p_k}}\right)$$

Nu is zowel $s_{p_{i-1}} \geq s_{p_i} + 1$ als $s'_{p_i} = s_{p_i} + 1$ en dus ook $s_{p_{i-1}} \geq s'_{p_i}$, zodat deze potentiaaltoename niet groter is dan

$$\lg\left(\frac{s'_{p_1}}{s_{p_k}}\right) \leq \lg(n+1)$$

De geamortiseerde tijd van toevoegen is dus $O(1 + 4\lg n)$.

- Een knoop fysich verwijderen doet de rang van de knopen op de zoekweg dalen, zodat het effect nooit positief is.

De woordenboekoperaties (splay-operaties inbegrepen) zijn dus allemaal zeker $O(1 + 4\lg n)$.

Tenslotte is de geamortiseerde tijd voor een reeks van $m$ woordenboekoperaties de som van de geamortiseerde tijden voor de individuele operaties. Met $n_i$ het aantal knopen bij de $i$-de operatie wordt die tijd $O(m + 4\sum_{i=1}^{m} \lg n_i)$, en als de boom maximaal $n$ knopen bevat is dat resultaat zeker $O(m + 4m\lg n)$ of eenvoudiger $O(m \lg n)$.

## 1.4 Randomized search trees

De verwachtingswaarde van de performantie van de woordenboekoperaties op een gewone zoekboom is $O(\lg n)$ als elke toevoegvolgorde even waarschijnlijk is. Dat is niet zeer realistisch. Bovendien geldt dit niet noodzakelijk wanneer men toevoegen afwisselt met verwijderen.

Randomized search trees ([**?**], [**?**]) maken gebruik van een random generator om het effect van de operatievolgorde te neutraliseren. Deze bomen blijven random, ook na elke operatie. De verwachtingswaarde van hun performantie hangt immers enkel af van de kwaliteit van de (hopelijk goede) random generator, en is dus steeds $O(\lg n)$.

Een treap (Aragon en Seidel, 1989) is de eenvoudigste boom van deze familie, en combineert een binaire zoekboom met een heap ('tree' + 'heap' = 'treap'). Elke knoop krijgt naast een sleutel (met eventueel bijbehorende informatie) ook nog een *prioriteit*, die bij het toevoegen door een random generator toegekend wordt. Een treap is een binaire zoekboom waarbij de prioriteit van de knopen ook nog aan de *heapvoorwaarde* voldoet: de prioriteit van een kind is minstens even groot als die van zijn ouder. (Een treap moet echter niet voldoen aan de *structuur*voorwaarde van een binaire heap: het is niet noodzakelijk een complete binaire boom.)

Wanneer alle sleutels verschillend zijn, en ook alle prioriteiten, dan is er met deze gegevens slechts één treap mogelijk. (De knoop met de kleinste prioriteit moet immers steeds de wortel zijn, en zijn sleutel verdeelt de sleutels in twee groepen. Dit geldt recursief voor elk van de deelbomen.) Met als belangrijk gevolg dat de vorm van een treap niet afhangt van de toevoegvolgorde van de sleutels, maar van de random gegenereerde prioriteiten. Bij een goede random generator is elke reeks van $n$ prioriteiten even waarschijnlijk, zodat een treap een random binaire zoekboom is.[2]

[2]Strikt genomen moeten alle prioriteiten dus verschillend zijn, maar daarvoor zorgen heeft meer nadelen dan voordelen.

De woordenboekoperaties verlopen als volgt:

- Zoeken houdt natuurlijk geen rekening met de prioriteiten, en is dus identiek aan zoeken in een gewone binaire zoekboom.
- Toevoegen van een knoop gebeurt als blad, zoals bij een gewone zoekboom. Pas daarna houdt men rekening met de prioriteiten, en wordt de knoop indien nodig naar boven geroteerd, om aan de heapvoorwaarde te voldoen, desnoods tot bij de wortel. Stijgen vereist natuurlijk ouderwijzers of een stapel. Elke rotatie herstelt de heapvoorwaarde tussen een kind en zijn ouder, zonder gevolgen voor de heapvoorwaarde elders in de boom (behalve eventueel hoger op de zoekweg, bij de grootouder). In tegenstelling tot rood-zwarte bomen en zelfs splay trees is het aantal gevallen hier dus heel beperkt.
- Een te verwijderen knoop krijgt de hoogst mogelijke prioriteit, zodat hij naar beneden kan geroteerd worden, en als blad uit de treap verdwijnt. Ook hier zijn er dus maar twee gevallen te onderscheiden.

De verwachtingswaarde van de efficiëntie van de woordenboekoperaties op treaps is $O(\lg n)$, wat beter is dan bij gewone zoekbomen, aangezien dat resultaat niet afhangt van de operatievolgorde. Hun performantie is niet zo goed als die van evenwichtige zoekbomen (want die halen dat resultaat in het slechtste geval), maar hun implementatie is veel eenvoudiger, onder meer omdat de prioriteit van een knoop nooit moet aangepast worden (in tegenstelling tot kleur of informatie over hoogte). Ook ten opzichte van splay trees kunnen treaps eenvoudiger en sneller uitvallen, omdat splay trees vaak veel rotaties uitvoeren (zelfs bij zoeken). Zoeken in een treap vereist immers geen rotaties, en men kan aantonen dat zowel toevoegen als verwijderen gemiddeld minder dan twee rotaties uitvoeren. Treaps kunnen echter geen goede performantie voor een reeks operaties garanderen.

# Hoofdstuk 2

# Optimale binaire zoekbomen

Stel dat de gegevens die in een binaire zoekboom moeten opgeslagen worden op voorhand gekend zijn, en niet meer wijzigen. Stel dat we ook weten met welke waarschijnlijkheid die gegevens gezocht zullen worden, en ook de afwezige gegevens, dan kunnen we trachten om de boom zo te schikken, dat de verwachtingswaarde van de zoektijd minimaal wordt.

De zoektijd wordt bepaald door de lengte van de zoekweg: voor een aanwezig gegeven is dat de diepte van de knoop, voor een afwezig gegeven de diepte van de ledige deelboom (een nullpointer). Daarom zullen we alle ledige deelbomen als bladeren van de zoekboom beschouwen. Alle aanwezige gegevens zijn dan in inwendige knopen opgeslagen.

De (gerangschikte) sleutels van de $n$ aanwezige gegevens noemen we $s_1, s_2, \ldots, s_n$, de $n+1$ bladeren $b_0, b_1, \ldots, b_n$. Elk blad staat voor alle afwezige gegevens die in een deelboom op die plaats hadden moeten zitten: $b_0$ staat voor alle sleutels kleiner dan $s_1$, $b_n$ voor alle sleutels groter dan $s_n$, en $b_i$ voor alle sleutels groter dan $s_i$ en kleiner dan $s_{i+1}$, voor $1 \leq i < n$.

De waarschijnlijkheid waarmee naar de $i$-de aanwezige sleutel $s_i$ gezocht wordt noemen we $p_i$. Het heeft weinig zin om de waarschijnlijkheid te kennen waarmee elk afwezig gegeven gezocht wordt. Voor veel afwezige sleutels zal de zoekweg immers bij hetzelfde blad eindigen. We moeten dus enkel de waarschijnlijkheid $q_i$ kennen waarmee alle afwezige sleutels voorgesteld door blad $b_i$ kunnen gezocht worden. Omdat zoeken dan steeds een inwendige knoop of een blad vindt, is

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

Als we de zoektijd van een knoop of blad gelijk stellen aan het aantal knopen op de zoekweg (de diepte plus één), dan is de verwachtingswaarde van de zoektijd

$$\sum_{i=1}^{n} (\text{diepte}(s_i) + 1)p_i + \sum_{i=0}^{n} (\text{diepte}(b_i) + 1)q_i$$

We moeten nu een zoekboom vinden die deze uitdrukking minimaliseert.

Tegenvoorbeelden tonen aan dat een boom met minimale hoogte niet noodzakelijk optimaal is, en dat een inhalige methode die in de wortel van elke deelboom de sleutel met de grootste

zoekwaarschijnlijkheid onderbrengt, ook niet altijd correct is. Alle mogelijke binaire zoekbomen onderzoeken is natuurlijk geen optie, aangezien hun aantal $\frac{1}{n+1}\binom{2n}{n} = \Omega(4^n/n^{3/2})$ bedraagt.

Gelukkig biedt dynamisch programmeren een uitkomst. Een optimalisatieprobleem komt in aanmerking voor een efficiënte oplossing via dynamisch programmeren, als het een optimale deelstructuur heeft, en de deelproblemen onafhankelijk maar overlappend zijn[1]. Laten we eens zien of dat hier het geval is:

- Als een zoekboom optimaal is, dan moet ook elk van zijn deelbomen optimaal zijn, want anders kunnen we een suboptimale deelboom vervangen door een beter exemplaar, wat een betere volledige boom oplevert. Een optimale oplossing bestaat dus uit optimale oplossingen voor deelproblemen. Zijn die deelproblemen ook onafhankelijk? Ja, want deelbomen hebben geen gemeenschappelijke knopen.

- Om te zien of de deelproblemen overlappend zijn, zodat veel optimale deelbomen meermaals in de oplossing gebruikt worden, moeten we eerst een oplossingsstrategie bedenken.

  Hoe kunnen we een optimale deelboom vinden? Elke deelboom bevat een reeks opeenvolgende sleutels $s_i, \ldots, s_j$ met bijbehorende bladeren $b_{i-1}, \ldots, b_j$. Een van die sleutels, bijvoorbeeld $s_w$ $(i \leq w \leq j)$, zal wortel zijn van de optimale deelboom. De linkse deelboom van $s_w$ bevat dan de sleutels $s_i, \ldots, s_{w-1}$ met de bladeren $b_{i-1}, \ldots, b_{w-1}$, en de rechtse deelboom de sleutels $s_{w+1}, \ldots, s_j$ met de bladeren $b_w, \ldots, b_j$. (Als $w$ gelijk is aan $i$, dan bevat de linkse deelboom de sleutels $s_i, \ldots, s_{i-1}$, wat natuurlijk betekent dat hij geen sleutels bevat, maar wel het blad $b_{i-1}$. Analoog, als $w$ gelijk is aan $j$ bevat de rechtse deelboom enkel het blad $b_j$.) Voor een optimale deelboom met wortel $s_w$ moeten deze beide deelbomen zelf optimaal zijn. We vinden dus de optimale deelboom door achtereenvolgens elk van zijn sleutels $s_i, \ldots, s_j$ als wortel te kiezen, de zoektijd voor de boom te berekenen gebruik makend van de zoektijden van zijn (optimale) deelbomen, en het minimum bij te houden. Daarbij komen veel deelbomen meermaals voor, zodat het inderdaad de moeite loont om vooraf hun zoektijden te bepalen en op te slaan.

Concreter geformuleerd, komt elk deelprobleem neer op het bepalen van de kleinste verwachte zoektijd $z(i, j)$ voor een (deel)boom met sleutels $s_i, \ldots, s_j$ en bijbehorende bladeren $b_{i-1}, \ldots, b_j$. En dit moet gebeuren voor alle $i$ en $j$ zodat $1 \leq i \leq n+1$, $0 \leq j \leq n$, en $j \geq i-1$. (Als $j = i-1$ dan is de deelboom ledig, zie hoger.) De gezochte oplossing zal dan $z(1, n)$ zijn.

Om $z_w(i, j)$ voor een deelboom met wortel $s_w$ te vinden $(i \leq w \leq j)$ maken we natuurlijk gebruik van de optimale zoektijden van zijn eigen deelbomen, $z(i, w-1)$ en $z(w+1, j)$. Deze zoektijden zijn echter berekend ten opzichte van de wortels van die deelbomen. Als kinderen van $s_w$ wordt de diepte van al hun knopen (en bladeren) één groter. Hun bijdrage tot de zoektijd neemt dan toe met de som van de zoekwaarschijnlijkheden van al hun sleutels en bladeren. Voor een willekeurige deelboom met sleutels $s_i, \ldots, s_j$ is die som

$$g(i, j) = \sum_{k=i}^{j} p_k + \sum_{k=i-1}^{j} q_k$$

[1] Zie Grafen I in Algoritmen I.

Met $s_w$ als wortel krijgen we dan dat

$$z_w(i,j) = p_w + (z(i,w-1) + g(i,w-1)) + (z(w+1,j) + g(w+1,j))$$

of eenvoudiger

$$z_w(i,j) = z(i,w-1) + z(w+1,j) + g(i,j)$$

Voor een optimale deelboom moet $z(i,j)$ minimaal zijn. We moeten dus achtereenvolgens elke sleutel van de deelboom tot wortel maken, en het minimum bepalen. De index $w$ doorloopt daarvoor alle waarden tussen $i$ en $j$ (inbegrepen):

$$z(i,j) = \min_{i \le w \le j} \{z_w(i,j)\} = \min_{i \le w \le j} \{z(i,w-1) + z(w+1,j)\} + g(i,j)$$

Dat minimum bepalen heeft enkel zin wanneer $i \le j$, dus als de deelboom sleutels bevat. Wanneer $j = i-1$ heeft de deelboom geen sleutels, maar enkel het blad $b_{i-1}$. In dat geval is $z(i,i-1) = q_{i-1}$. (Idem als $i = j+1$: $z(j+1,j) = q_j = q_{i-1}$.)

Nu zijn we niet enkel geïnteresseerd in de minimale verwachte zoektijd, maar natuurlijk ook in de boom zelf. Daartoe volstaat het de index $w$ van de wortel van elke optimale deelboom bij te houden. Voor de deelboom met sleutels $s_i, \ldots, s_j$ $(1 \le i \le j \le n)$ noemen we die index $r(i,j)$.

Een recursieve implementatie van $z_w(i,j)$ zou veel deeloplossingen meermaals bepalen. Ook bij de berekening van alle waarden $g(i,j)$ komen veel deelsommen meermaals voor. Daarom bepalen we de deeloplossingen bottom-up en slaan we de resultaten op in tabellen, zodat ze hogerop beschikbaar zijn. De implementatie van het volledige algoritme gebruikt drie tabellen:

- Een tweedimensionale tabel $z[1..n+1, 0..n]$ voor de waarden $z(i,j)$. Beide grenzen gaan niet van 1 tot $n$, omdat we plaats nodig hebben voor $z(1,0)$ (een deelboom met enkel blad $b_0$), en voor $z(n+1,n)$ (een deelboom met enkel blad $b_n$). Een driehoeksmatrix zou eigenlijk volstaan, want steeds is $j \ge i-1$.
- Een tweedimensionale tabel $g[1..n+1, 0..n]$ voor de waarden $g(i,j)$, om efficiëntieredenen. Immers, voor de bepaling van $z(i,j)$ zoeken we het minimum voor alle mogelijke indices $w$ van de wortel, waarbij telkens dezelfde waarde $g(i,j)$ voorkomt. Die steeds opnieuw berekenen zou veel onnodig werk betekenen. Om deze tabel (nóg een driehoeksmatrix) in te vullen maken we gebruik van de betrekkingen $g(i,j) = g(i,j-1) + p_j + q_j$ (voor $i \le j$) en $g(i,i-1) = q_{i-1}$.
- Een tweedimensionale tabel $r[1..n, 1..n]$ voor de indices $r(i,j)$ van de wortels van de optimale deelbomen. Ook hier volstaat een driehoeksmatrix, want steeds geldt dat $1 \le i \le j \le n$.

Het algoritme moet ervoor zorgen dat de waarden $z(i,j)$ en $g(i,j)$ in de juiste volgorde bepaald worden. Voor beide is dat mogelijk door eerst alle elementen $z(i,i-1)$ en $g(i,i-1)$ op de diagonaal in te vullen (de triviale gevallen), en dan achtereenvolgens de elementen op elke evenwijdige diagonaal, in de richting van de tabelhoek linksboven. Immers, voor $z(i,j)$ zijn

alle mogelijke waarden $z(i,i-1), z(i,i), \ldots, z(i,j-1)$ van linkse deelbomen nodig, en ook alle mogelijke waarden $z(i+1,j), \ldots, z(j,j), z(j+1,j)$ van rechtse deelbomen. En al die waarden staan op diagonalen onder deze van $z(i,j)$. Voor $g(i,j)$ volstaat één waarde $g(i,j-1)$ op de diagonaal er onder. Aangezien $z(i,j)$ gebruik maakt van $g(i,j)$ moet deze waarde eerst bepaald worden. De pseudocode 2.1 geeft wat meer details.

```
// Gegeven n, en de reele tabellen p[1..n] en q[0..n]
double z[1..n+1,0..n]; // Minimale verwachtingswaarde van de zoektijden
int g[1..n+1,0..n];
int r[1..n,1..n]; // Indices van de wortelelementen van optimale deelbomen
// Initialisatie van de diagonalen van z en g
for(int i=1 ; i<=n+1 ; i++)
   z[i,i-1]=g[i,i-1]=q[i-1];
// Invullen van de tabellen g,z en r
for(int k=1 ; k<=n ; k++)
   for(int i=1 ; i<=n-k+1 ; i++){
      int j=i+k-1;
      g[i,j]=g[i,j-1]+p[j]+q[j];
      // Index w van wortelelement zoeken dat minimum oplevert
      z[i,j]=oneindig;
      for(int w=i ; w<=j ; w++){
         double t=z[i,w-1]+z[w+1,j]+g[i,j];
         if(t<z[i,j]){
            z[i,j]=t;
            r[i,j]=w;
         }
      }
   }
// Resultaat: z[1,n], en de wortelindices in tabel r
```

Pseudocode 2.1: Bepalen van een optimale binaire zoekboom

Rest nog de performantie. Aangezien de drie vernestelde herhalingen elk niet meer dan $n$ iteraties kunnen uitvoeren, is dit algoritme zeker $O(n^3)$. Maar wat is de ondergrens? Het meeste werk gebeurt in de binnenste herhaling, waar alle mogelijke wortels voor elke deelboom getest worden. Een deelboom met sleutels $s_i, \ldots, s_j$ $(1 \leq i \leq j \leq n)$ heeft $j-i+1$ mogelijke wortels, en elke test is $O(1)$. Dat werk is dus evenredig met $\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i+1)$, en omdat $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$, ook evenredig met $n^3$. De totale performantie is dus ook $\Theta(n^3)$. Door gebruik te maken van een bijkomende eigenschap kan men het aantal te testen wortels sterk reduceren, zodat de performantie $\Theta(n^2)$ wordt (Knuth, 1971, zie [**?**]).

# Hoofdstuk 3

# Uitwendige gegevensstructuren

Wanneer de omvang van de gegevens te groot is voor het intern geheugen, moeten ze in een extern geheugen opgeslagen worden (meestal magneetschijven). Hoe moeten deze gegevens nu georganiseerd worden, om er efficiënte woordenboekoperaties en (eventueel) sequentiële operaties (volgorde) op uit te voeren?

Er moet natuurlijk een prijs betaald worden voor de grote en goedkope opslagcapaciteit van de schijf. De toegangstijd tot een gegeven wordt nu niet meer bepaald door electronische eigenschappen, maar door de mechanische performantie van de schijf: haar rotatiesnelheid en de tijd nodig voor het verplaatsen van de lees/schrijfkop. Gemiddeld is deze tijd typisch enkele milliseconden lang, en processoren kunnen in dezelfde tijd honderdduizenden instructies uitvoeren. Deze situatie verbetert er niet op, omdat processoren steeds sneller worden, terwijl wel de opslagcapaciteit van schijven spektakulair stijgt, maar niet hun toegangstijd.

Eens de informatie gelokaliseerd werd, kunnen er grote hoeveelheden gegevens snel overgebracht worden. Lezen van en schrijven naar een schijf gebeurt dan ook in vrij grote informatieblokken (pagina's).

De performantie van een programma dat gebruik maakt van een schijf, wordt dus voornamelijk bepaald door het aantal schijfoperaties. Het loont dan ook de moeite om eventueel het algoritme ingewikkelder te maken, als dat maar resulteert in minder schijfoperaties.

Net zoals inwendige gegevensstructuren maken uitwendige gegevensstructuren gebruik van boomstructuren of van hashing. Wanneer de volgorde van de gegevens belangrijk is, gebruikt men meestal een uitwendige versie van een evenwichtige zoekboom, de B-tree.[1] Als volgorde geen rol speelt, is een uitwendige versie van een hashtabel een efficiënt alternatief. Het artikel van Vitter [**?**] geeft een overzicht.

[1] Met de huidige grootte van de inwendige geheugens, gebruikt men zelfs een B-tree als *inwendige* gegevensstructuur.

## 3.1 B-Trees

Een 'B-tree' (Bayer en McCreight, 1972) is een uitwendige evenwichtige zoekboom.[2] De efficiëntie van de meeste operaties op zoekbomen wordt begrensd door de hoogte van de boom, en dus zorgt men ervoor dat een B-tree onder alle omstandigheden perfect in evenwicht blijft, en bovendien een zeer kleine hoogte heeft. Aangezien het aantal sleutels $n$ zeer groot is, en de minimale hoogte van een binaire boom $\lfloor \lg n \rfloor$, zullen we bomen met meer dan twee kinderen per knoop moeten gebruiken als we een zeer kleine hoogte willen. Omdat bovendien de informatie-overdracht van en naar een schijf in pagina's gebeurt, nemen we knopen die een volledige schijfpagina beslaan, en we geven ze zoveel kinderen als ze kunnen bevatten. Zo krijgen we knoopgraden gaande van 50 tot enkele duizendtallen, afhankelijk van de sleutels en de schijf.

Een knoop van een B-tree moet men eerst in het inwendig geheugen inlezen, vooraleer er mee kan gewerkt worden. Gewijzigde knopen moet men nadien natuurlijk op de schijf aanpassen. Om schijfoperaties uit te sparen voorziet men gewoonlijk geheugenruimte voor de meest recent gebruikte knopen. Aangezien elke operatie slechts een gering aantal knopen gebruikt (de hoogte en dus de afgelegde weg in de boom zijn immers heel klein), zal zeker de wortel en soms ook het volledige eerste niveau van de boom in het geheugen aanwezig blijven.

De op te slagen gegevens bestaan zoals gewoonlijk uit een sleutel met bijbehorende informatie. Dikwijls wordt de sleutel enkel vergezeld van een wijzer naar de bijbehorende informatie, maar voor de eenvoud zullen we onderstellen dat sleutel en informatie samen blijven. De sleutels moeten ook geordend kunnen worden, zoals bij elke zoekboom.

Een B-tree is dus een meerwegszoekboom waarvan alle bladeren dezelfde diepte hebben. Bij toevoegen en verwijderen van sleutels wordt het perfecte evenwicht van de boom behouden door het aantal kinderen in de knopen te manipuleren. (Zoals bij zijn voorloper, de 2-3 boom.) Ingrijpende wijzigingen aan de boomstructuur zijn dan ook zeldzaam. (Gelukkig maar, gezien de traagheid van schijfoperaties.) Om te vermijden dat de boom te hoog wordt, zorgt men ervoor dat elke (inwendige) knoop steeds genoeg kinderen heeft.

### 3.1.1 Definitie

Een B-tree van orde $m$ wordt als volgt gedefinieerd:

- Elke inwendige knoop heeft hoogstens $m$ kinderen.
- Elke inwendige knoop, behalve de wortel, heeft minstens $\lceil m/2 \rceil$ kinderen. De wortel heeft minstens twee kinderen, tenzij hij een blad is.
- Elke inwendige knoop met $k$ kinderen bevat $k-1$ sleutels. De bladeren bevatten hoogstens $m-1$ en minstens $\lceil m/2 \rceil - 1$ sleutels. Als de wortel een blad is bevat hij minstens één sleutel, tenzij de B-tree ledig is.

[2] De naam werd nooit verklaard. Misschien komt de 'B' van 'Boeing', de toenmalige werkgever van de uitvinders. Omwille van deze en nog andere bijdragen zou men ze gerust 'Bayer-trees' kunnen noemen.

- Alle bladeren bevinden zich op hetzelfde niveau.

Elke knoop bevat dan:

- Een tabel voor maximaal $m$ 'wijzers' naar de kinderen van de knoop. Bij een blad blijft deze tabel ongedefinieerd.
- Een tabel voor maximaal $m-1$ sleutels, die stijgend (niet dalend) gerangschikt zijn. Een tweede tabel, van dezelfde grootte, met hun bijbehorende informatie, of een verwijzing ernaar. De $k-1$ geordende sleutels van een inwendige knoop verdelen het sleutelbereik in $k$ deelgebieden. De sleutels van elk deelgebied zitten in een deelboom met als wortel een kind van deze knoop. De sleutels uit de deelboom van het $i$-de kind $c_i$ liggen dus tussen de sleutels $s_{i-1}$ en $s_i$. (Deze in de deelboom van $c_1$ zijn natuurlijk kleiner dan $s_1$, en deze in de deelboom van $c_{k+1}$ zijn groter dan $s_k$.)
- Een geheel getal $k$ dat het huidig aantal *sleutels* in de knoop aanduidt. (Niet elke knoop heeft kinderen.)
- Een logische waarde $b$ die aanduidt of de knoop een blad is of niet. Door het perfecte evenwicht is er geen behoefte aan 'nullwijzers' voor de ontbrekende kinderen van een blad: de logische waarde duidt immers aan of we ons op het laagste niveau bevinden.

De eenvoudigste B-trees zijn de 2-3 bomen (orde 3) en de 2-3-4 bomen (orde 4).[3] Normaal is $m$ echter veel groter, zodat de hoogte van de boom (zeer) klein blijft. Bemerk dat de knopen plaats moeten reserveren voor de maximale grootte van hun tabellen. De waarde van $m$ hangt dus af van de grootte van een schijfpagina, van de grootte van de sleutel en van die van zijn bijbehorende gegevens.

### 3.1.2 Eigenschappen

Zoals bij elke zoekboom is de langste afgelegde weg bij woordenboekoperaties op een B-tree evenredig met zijn hoogte. Toegang tot elke knoop op die weg vereist echter een schijfoperatie. (En een tweede als die gewijzigd wordt.) Gelukkig blijkt die hoogte zeer klein, ook voor heel veel sleutels.

Stel dat de boom niet ledig is, en hoogte $h$ heeft. Wat is dan het minimaal aantal sleutels $n$ dat hij bevat? (Bemerk dat, in tegenstelling tot binaire zoekbomen, het aantal sleutels *niet* gelijk is aan het aantal knopen.) De wortel van een minimale boom heeft slechts één sleutel, en dus twee kinderen (als $h > 0$). Elk van die kinderen heeft minimaal $g = \lceil m/2 \rceil$ kinderen, en die op hun beurt ook elk minimaal $g$ kinderen, enz. Het aantal *knopen* is dus ten minste

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

[3] Bij de originele definitie van een B-tree was $m$ oneven. Nog andere definities vereisen dat $m$ even is.

Elke knoop heeft minstens $g-1$ sleutels, behalve de wortel, die heeft er minstens één. Er komt dan

$$n \geq 1 + 2(g-1)\left(\frac{g^h - 1}{g-1}\right)$$

zodat

$$n \geq 2g^h - 1$$

of tenslotte, na het nemen van de logaritme met basis $g = \lceil m/2 \rceil$ (een constante)

$$h \leq \log_{\lceil m/2 \rceil} \frac{n+1}{2}.$$

De hoogte is dus $O(\lg n)$, zoals bij een rood-zwarte boom, maar de (verborgen) constante is een factor $\lg\lceil m/2 \rceil$ kleiner.

Men kan aantonen dat een B-tree met $n$ uniform verdeelde sleutels gemiddeld ongeveer $n/(m \ln 2)$ schijfpagina's gebruikt, zodat elke pagina gemiddeld voor ongeveer 69 procent gevuld is ($\ln 2 \approx 0.69$).

### 3.1.3 Woordenboekoperaties

#### 3.1.3.1 Zoeken

Zoeken gebeurt langs een weg vanuit de wortel in de richting van een blad, zoals bij een binaire zoekboom. Natuurlijk moet er nu in elke knoop op die weg een meerwegsbeslissing genomen worden.

Elke knoop op die weg moet eerst in het geheugen ingelezen worden (de wortel uiteraard niet meer). De sleutel wordt dan opgezocht in de gerangschikte tabel met sleutels. De snelste manier is natuurlijk binair zoeken, maar de winst is vrij onbelangrijk naast de schijfoperaties. Lineair zoeken in een gerangschikte tabel kan overigens ook wat efficiënter uitvallen.[4]

Vinden we de sleutel, dan stopt het zoeken, met als resultaat een verwijzing naar de knoop op de schijf (de opslag in het geheugen is immers maar tijdelijk), en de plaats van de sleutel in die knoop. Vinden we de sleutel niet, en is de knoop een blad, dan is de sleutel niet in de boom aanwezig. Bij een interne knoop echter moeten we verder zoeken in de juiste deelboom, waarvan de wortel een kind is van de huidige knoop. De wijzer naar dat kind staat bij dezelfde index in de kindertabel als waar de niet gevonden sleutel zou moeten staan in de sleuteltabel. Dat kind wordt dan ingelezen, en het zoekproces wordt op die nieuwe knoop herhaald.

Tijdens het afdalen moet elke knoop (behalve de wortel) ingelezen worden. Het aantal schijfoperaties is dus $O(h) = O(\log_{\lceil m/2 \rceil} n)$. De processortijd per knoop is $O(m)$, en dus in totaal $O(m \log_{\lceil m/2 \rceil} n)$. Bij binair zoeken zou dat $O(\lg n)$ worden. De verborgen constante is echter veel kleiner dan die voor de schijfoperaties.

[4] Recent gebruikt men interpolerend zoeken om het aantal cachefouten te minimaliseren. Interpolerend zoeken werkt echter slecht als de sleutels niet uniform verdeeld zijn, zodat men allerlei technieken aanwendt om dit probleem te omzeilen [**?**].

#### 3.1.3.2 Toevoegen

Door het aantal sleutels te wijzigen kunnen aanpassingen aan de boomstructuur nodig worden. Net zoals bij rood-zwarte bomen kunnen die bottom-up of top-down gebeuren. In het slechtste geval vereist de bottom-up versie dat de hoogte van de boom tweemaal doorlopen wordt, wat veel erger is dan bij een inwendige boom. Dit slechtste geval is echter zeldzaam (zie hieronder), en als men bovendien de meest recente gebruikte knopen in het inwendig geheugen bijhoudt, vervalt dit nadeel. We bespreken hier dan ook de bottom-up versie, omdat die het meest gebruikt wordt.[5]

De initialisatie van een ledige B-tree vereist wel wat meer werk dan die van een inwendige zoekboom. Men moet de wortelknoop in het geheugen aanmaken en gedeeltelijk invullen, en dan op de schijf kopiëren. Een ledige wortel is een blad, dat bovendien geen sleutels bevat. Zoals voor elke nieuwe knoop moeten we er eerst plaats voor reserveren op de schijf, vooraleer we kunnen kopiëren. Een verwijzing naar die plaats moet ook permanent bijgehouden worden. (Een andere knoop kan immers wortel worden.)

Toevoegen gebeurt steeds aan een blad want toevoegen aan een inwendige knoop zou een extra kind met een bijbehorende deelboom vereisen. Vanuit de wortel zoeken we dus het blad waarin de sleutel zou moeten zitten, en we voegen hem op de juiste plaats in de tabel toe (met zijn bijbehorende informatie).

Als dat blad dan $m$ sleutels bevat, wordt het gesplitst bij de middelste sleutel (nummer $\lceil m/2 \rceil$). Daarbij wordt een nieuwe knoop op hetzelfde niveau aangemaakt (in het inwendig geheugen en op de schijf), waarin de gegevens rechts van de middelste sleutel terechtkomen. De middelste sleutel zelf (met zijn bijbehorende informatie) gaat naar de ouder, vergezeld van een verwijzing naar de nieuwe knoop. Een deel van de tabellen in de ouder moet dus opgeschoven worden.

Gewoonlijk heeft die ouder plaats voor deze extra sleutel. Als dat niet het geval is, moet de ouder op zijn beurt gesplitst worden. Als een te splitsen knoop kinderen heeft, moeten die samen met de sleutels over de originele en de nieuwe knoop verdeeld worden.

Soms tracht men het opsplitsen van knopen zo lang mogelijk uit te stellen, door gegevens over te brengen naar een broerknoop, en ze over beide knopen zo goed mogelijk te verdelen. Dit is natuurlijk enkel mogelijk als die broer plaats heeft voor extra sleutels. Omdat de inorder volgorde van de sleutels intact moet blijven, gebeurt dat overbrengen weer via een soort rotatie: een sleutel gaat van de knoop naar zijn ouder, die een sleutel afstaat aan de broer. Die krijgt bovendien een kindwijzer van de knoop (als die kinderen heeft).

In het slechtste geval moet elke knoop op de (terug)weg naar de wortel gesplitst worden. Het kan ook nodig zijn om de wortel te splitsen. Er is dan echter geen ouder meer om de middelste sleutel in onder te brengen. Die moet dus aangemaakt worden, en wordt de nieuwe wortel. (Met slechts één sleutel, maar dat is toegelaten.) In tegenstelling tot binaire zoekbomen groeien B-trees dus bovenaan. (Om onderaan te groeien zou er een volledig niveau moeten bijkomen, wegens de perfecte vorm van de boom.) Alle nieuwe en alle gewijzigde knopen

[5] De top-down versie (Guibas en Sedgewick, 1978) is interessant voor een B-tree met meerdere gelijktijdige gebruikers. De knopen op de gevolgde weg kunnen dan immers vroeger vrijgegeven worden, zodat ze door anderen kunnen gebruikt worden.

moeten op de schijf aangepast worden.

In het slechtste geval worden er dus $h + 1$ knopen gesplitst. Dat is echter zeer zeldzaam. Immers, het totaal aantal splitsingen sinds de initialisatie van de boom is gelijk aan het aantal knopen, min $(h + 1)$. (Ga eens na.) Dat is veel minder dan het aantal toegevoegde sleutels. Want een boom met $p$ knopen bevat minstens $1 + (p-1)(\lceil m/2 \rceil - 1)$ sleutels, zodat $p \leq 1 + (n-1)/(\lceil m/2 \rceil - 1)$. Per toegevoegde sleutel is het gemiddeld aantal splitsingen dan ook kleiner dan $1/(\lceil m/2 \rceil - 1)$.

Een knoop splitsen vereist drie schijfoperaties (als de te splitsen knoop en zijn ouder reeds ingelezen zijn), en een processortijd van $O(m)$, omdat alle kopieer- en opschuifbewerkingen begrensd zijn door het maximaal aantal kinderen in een knoop. In het slechtste geval moeten we tweemaal de volledige hoogte van de boom doorlopen, maar er gebeurt slechts een constant aantal schijfoperaties per niveau. Het totaal aantal schijfoperaties is dus $\Theta(h) = \Theta(\log_{\lceil m/2 \rceil} n)$. Per knoop is de processortijd $O(m)$ (want we moeten zoeken in of toevoegen aan een tabel met grootte $O(m)$, naast eventueel nog $O(m)$ voor opsplitsen), met dus een totaal van $O(mh) = O(m \log_{\lceil m/2 \rceil} n)$.

#### 3.1.3.3 Verwijderen

De te verwijderen sleutel kan zowel in een blad als in een inwendige knoop zitten. Maar uiteindelijk moeten we steeds uit een blad verwijderen, want anders zou er samen met de sleutel een kind en de bijbehorende deelboom moeten verdwijnen.

Ook hier bespreken we een bottom-up versie: als het blad te weinig sleutels overhoudt, moet er soms een sleutel aan de ouder ontleend worden. In het slechtste geval gaat dit ontlenen door tot bij de wortel. Een sleutel ontlenen aan een wortel met slechts één sleutel maakt die wortel ledig, zodat die verwijderd moet worden: een B-tree krimpt dus ook bovenaan.

Wanneer de te verwijderen sleutel in een inwendige knoop zit, mogen we de structuur van die knoop niet wijzigen. Daarom vervangen we de sleutel door zijn voorloper of opvolger, die dan werkelijk verwijderd wordt (want te vinden in een blad). We moeten dus eerst helemaal afdalen, om dan achteraf nog een wijziging aan te brengen in een hoger gelegen knoop. Dat is echter zeldzaam, omdat de meeste sleutels van een B-tree in de bladeren zitten.

Wanneer een knoop te weinig sleutels overhoudt, kan men trachten een sleutel te ontlenen aan een broerknoop. Dat gebeurt met dezelfde rotatie als bij toevoegen: de sleutel van de broer gaat naar zijn ouder, een sleutel van de ouder gaat naar de knoop, die ook een kindwijzer van de broer overneemt (als die kinderen heeft). Omdat hierbij drie knopen moeten aangepast worden, brengt men soms meerdere sleutels over van een broer, zodat ze elk ongeveer even veel sleutels krijgen. Opnieuw verwijderen uit dezelfde knoop geeft dan niet meteen weer problemen.

Natuurlijk kan men enkel sleutels ontlenen aan een broer die sleutels kan missen. Als dat niet het geval is, kan men eventueel hetzelfde proberen bij de andere broer, als die bestaat. Maar als geen broer een sleutel kan missen, dan gebeurt het omgekeerde van splitsen: de knoop wordt samengevoegd met een broer, zodat er een knoop uit de boom verdwijnt. Hun ouder verliest dan een kind, zodat de sleutel tussen de twee broers moet verdwijnen: hij wordt ook

toegevoegd aan de samengevoegde knoop. Met als mogelijk gevolg dat de ouder te weinig sleutels overhoudt.

Bij roteren of samenvoegen zijn drie knopen betrokken, zodat drie schijfoperaties vereist zijn (naast de leesoperaties). In het slechtste geval moeten we tweemaal de volledige hoogte van de boom doorlopen, maar er gebeurt slechts een constant aantal schijfoperaties per niveau. Het totaal aantal schijfoperaties is dus $\Theta(h) = \Theta(\log_{\lceil m/2 \rceil} n)$. Per knoop is de processortijd $O(m)$ (want we moeten zoeken in of verwijderen uit een tabel met grootte $O(m)$, naast eventueel nog $O(m)$ voor samenvoegen), met dus een totaal van $O(mh) = O(m \log_{\lceil m/2 \rceil} n)$.

### 3.1.4 Varianten van B-trees

#### 3.1.4.1 $B^+$-tree

Een gewone B-tree heeft enkele nadelen. Zo moeten de bladeren plaats reserveren voor kindwijzers die toch niet gebruikt en dus niet ingevuld worden. Dat inwendige knopen gegevens kunnen bevatten maakt bijvoorbeeld verwijderen ingewikkelder, en zowel toevoegen als verwijderen moet uiteindelijk toch in een blad gebeuren. Tenslotte kan zoeken naar de opvolger van een sleutel $O(\log_{\lceil m/2 \rceil} n)$ schijfoperaties vereisen.

Een veel gebruikte B-tree variant is de $B^+$-tree (Knuth 1973), die alle gegevens (sleutels en hun bijbehorende informatie) in de bladeren opslaat. De inwendige knopen worden gebruikt als een *index* die toelaat om snel gegevens te lokaliseren. Met als gevolg dat bladeren en inwendige knopen een verschillende structuur krijgen, en ook in grootte mogen verschillen. Bovendien maakt men een *gelinkte lijst* van alle bladeren, in stijgende sleutelvolgorde (de 'sequence set').

Het scheiden van de index en de sequence set heeft belangrijke gevolgen. Omdat inwendige knopen uitsluitend dienen om de juiste weg te vinden, bevatten ze enkel sleutels (zonder bijbehorende informatie) en kindwijzers. Hun maximale graad kan daardoor groter worden. De bladeren moeten niet langer plaats reserveren voor kindwijzers, zodat ze meer gegevens kunnen bevatten. Het aantal gegevens in een blad wordt ook tussen twee grenzen gehouden, die dus meestal verschillen van deze voor de inwendige knopen. Door de knopen beter te gebruiken wordt de hoogte van de boom kleiner, wat de woordenboekoperaties sneller maakt. De sequence set zorgt ervoor dat zoeken naar de opvolger van een sleutel hoogstens één schijfoperatie vereist. Ook sequentieel overlopen van alle gegevens is nu efficiënter, want elke knoop wordt slechts eenmaal bezocht, zodat er plaats voor slechts één knoop in het inwendig geheugen nodig is. Een $B^+$-tree is dus zeer geschikt voor zowel random als sequentiële operaties.

De woordenboekoperaties verlopen nagenoeg zoals bij een gewone B-tree. Zoeken gaat nu steeds van de wortel naar een blad, en stopt dus niet bij een inwendige knoop die de gezochte sleutel bevat. Toevoegen gebeurt in een blad. Als dat blad moet gesplitst worden, blijft de middelste sleutel in een van de twee knopen, en gaat er enkel een *kopie* van die sleutel naar de ouder. Ook verwijderen gebeurt steeds in een blad. Zolang dat blad genoeg sleutels overhoudt, moeten hoger gelegen knopen niet gewijzigd worden, zelfs als ze een kopie van de verwijderde sleutel bevatten. De enige taak van de sleutels in de inwendige knopen is immers

te leiden naar het juiste blad. Het kan echter wel nodig zijn om deze knopen aan te passen, wanneer een blad gegevens moet ontlenen, of samengevoegd wordt met een buur.

#### 3.1.4.2 Prefix B$^+$-tree

Wanneer de sleutels strings zijn, kunnen ze in de inwendige knopen teveel plaats innemen. De enige functie van een sleutel in een inwendige knoop is echter de sleutels in twee deelbomen van elkaar te onderscheiden. Daarom gebruikt een Prefix B$^+$-tree (Bayer en Unterauer, 1977) daarvoor een zo kort mogelijke string, vaak een prefix van een van de te onderscheiden strings. Met als gevolg meer plaats voor sleutels in de inwendige knopen, zodat de boom minder hoog wordt, wat de zoektijd ten goede komt.

#### 3.1.4.3 B$^*$-tree

Bij een gewone B-tree stelt men soms het splitsen van een knoop uit, door gegevens over te brengen naar een buur. Wanneer die buur ook vol is, splitst een gewone B-tree de knoop in twee knopen die dan elk ongeveer half gevuld zijn. Een B$^*$-tree (Knuth, 1973) verdeelt de sleutels van de knoop en de volle buur over de *drie* knopen, zodat die elk voor ongeveer twee derden gevuld zijn. De wortel heeft echter geen buur, zodat men wacht met splitsen tot ook hier twee knopen voor twee derden kunnen gevuld worden. Dat betekent dat men toelaat dat de wortel tot vier derden van de normale grootte opgevuld wordt. Beter gevulde knopen betekent een minder hoge boom, en dus een kleinere zoektijd.

## 3.2 Uitwendige hashing

Wanneer men niet geïnteresseerd is in de volgorde van de sleutels, bestaan er meer eenvoudige, efficiënte alternatieven voor B-trees: uitwendige versies van hashtabellen [**?**] [**?**] [**?**]. Hun woordenboekoperaties vereisen immers (gemiddeld) slechts $O(1)$ schijfoperaties.

Een interne hashtabel met chaining moet van dezelfde grootteorde zijn als het aantal sleutels dat ze moet bevatten, en met open adressering is ze best tweemaal zo groot. Beide types kunnen dus duidelijk niet ongewijzigd gebruikt worden voor een zeer groot aantal sleutels.

Aangezien de informatie-overdracht bij een schijf toch in pagina's gebeurt, zou men aan elk hashtabelelement een schijfpagina kunnen toewijzen. Alle sleutels met dezelfde hashwaarde zouden dan in dezelfde pagina ondergebracht worden. Het streefdoel van $O(1)$ schijfoperaties lijkt dan gehaald. Maar wat als er zoveel conflicten zijn dat een pagina vol geraakt? Als we dezelfde technieken toepassen als bij inwendige hashtabellen, pagina's expliciet linken bij chaining, of alternatieve pagina's berekenen bij open adresseren, dan stijgt in beide gevallen het aantal schijfoperaties. En natuurlijk is rehashing van alle sleutels in al die schijfpagina's uitgesloten.

Een boom is een meer flexibele structuur dan een tabel. Aangezien alle gegevens in schijfpagina's opgeslagen zijn, moeten we zo snel mogelijk bij de juiste pagina terechtkomen. Wanneer men een sleutel zoekt in een (binaire) trie (zie later, in **??**), wordt die niet vergeleken met

sleutels in de knopen, maar worden zijn opeenvolgende *bits* gebruikt om het zoekproces te sturen. Elke deelboom van een trie bevat alle sleutels met een gemeenschappelijke prefix (die leidt naar de wortel van de deelboom). Aangezien de bladeren van de trie hier schijfpagina's zijn, die veel sleutels kunnen bevatten, kan men alle sleutels van een deelboom in één pagina onderbrengen, zodat de weg erheen in de trie korter wordt. Pas wanneer een pagina vol geraakt, wordt ze gesplitst, en beide pagina's krijgen een nieuwe trieknoop als ouder. De sleutels van de volle knoop worden dan op grond van de eerstvolgende bit tussen beide pagina's verdeeld. Die bit zal op de zoekweg getest worden bij hun nieuwe ouder.

De vorm van een trie is onafhankelijk van de toevoegvolgorde, en wordt volledig bepaald door de bits van de sleutels. Die boom kan dus zeer onevenwichtig uitvallen. Daarom gebruikt men niet de bits van de sleutels, maar van hun *hashwaarden*. Een goede hashfunctie zorgt er immers voor dat die hashwaarden (en dus hun bits) zo random mogelijk zijn. Met als gevolg een meer evenwichtige boom.

Beide methoden voor uitwendige hashing elimineren ook nog het zoeken in de trie. Extendible hashing vervangt die door een rechtstreeks adresseerbare tabel, linear hashing gebruikt pagina's met opeenvolgende (logische) adressen.

### 3.2.1 Extendible hashing

Deze methode (Fagin, Nievergelt, Pippenger en Strong, 1978) elimineert het zoeken in de trie, door het langst mogelijke prefix uit de trie als index te gebruiken in een hashtabel, die verwijzingen naar de overeenkomstige pagina's bevat. Kortere prefixen komen dan overeen met meerdere tabelelementen (elke index heeft immers evenveel bits), die allemaal een verwijzing naar dezelfde pagina moeten bevatten. Het vinden van een pagina is nu inderdaad $O(1)$.

Laten we de implementatie wat nader bekijken. Voorlopig onderstellen we dat de hashtabel opgeslagen is in het inwendig geheugen. Deze tabel bevat 'wijzers' naar de schijfpagina's, die maximaal $m$ (afhankelijk van de hardware) sleutels met bijbehorende gegevens bevatten. Net zoals bij een B-tree worden deze sleutels stijgend gerangschikt in een tabel opgeslagen.

Als hashwaarden gebruikt men gehele getallen, waarvan het bereik bepaald wordt door de breedte $w$ van een processorwoord. De laatste $d$ bits van die getallen worden als indices in de hashtabel gebruikt, zodat de tabel $2^d$ elementen bevat ($d < w$). Dat aantal bits $d$ (de 'globale diepte') is de lengte van het langste prefix in de trie, en kan dus variëren. (In tegenstelling tot een gewone hashtabel zijn hier (veel) meer hashwaarden dan tabelelementen.) Alle sleutels waarvan de hashwaarde met dezelfde $d$ bits eindigt komen dus bij hetzelfde tabelelement terecht, en worden in de overeenkomstige pagina opgeslagen.

Eenzelfde pagina kan sleutels met hashwaarden bevatten waarvan de laatste $d$ bits verschillend zijn, aangezien meerdere tabelelementen naar dezelfde pagina mogen verwijzen. Daarom houden we per pagina het aantal bits $k$ bij waarmee *al* haar hashwaarden eindigen (de 'lokale diepte'). De waarde van $k$ kan zelfs nul zijn, maar is zeker niet groter dan $d$. Het aantal tabelelementen dat naar dezelfde pagina wijst is dan $2^{d-k}$.

Extendible hashing voorziet enkel woordenboekoperaties (zoals gewone inwendige hashing):

- *Zoeken.* Zoeken van een sleutel komt neer op het berekenen van de hashwaarde, de pagina vinden via de hashtabel (en ze inlezen), en dan sequentieel (of eventueel binair) zoeken in de sleuteltabel. Zolang de hashtabel in het geheugen past, vereist dit slechts één schijfoperatie!

- *Toevoegen.* Toevoegen is analoog, en eenvoudig zolang de pagina niet vol is. Daarbij moet gemiddeld de helft van de gegevens in de tabel opgeschoven worden om de volgorde intact te houden, maar dat is verwaarloosbaar naast de tijd vereist voor een schijfoperatie.

  Een volle pagina moet echter gesplitst worden. Aangezien hier geen boomstructuur gebruikt wordt, kan splitsen ingrijpende gevolgen hebben voor de hashtabel.

  Omdat alle hashwaarden in de pagina met dezelfde $k$ bits eindigen, splitst men door de sleutels onder te verdelen volgens de waarde van bit $k+1$. Gegevens waarbij die bit één is worden overgebracht naar een nieuw gecreëerde pagina. Daarna wordt de waarde van $k$ één groter, zowel in de nieuwe als in de oude pagina. Ook de hashtabel moet aangepast worden:

  - Als $k$ kleiner was dan $d$ dan moet de helft van de wijzers naar de oude pagina (we weten die staan) verwijzen naar de nieuwe pagina.
  - Als $k$ gelijk was aan $d$, dan was er maar één wijzer naar de oude pagina. Aangezien $k$ nu groter wordt dan $d$, moet ook $d$ met één toenemen en dus *verdubbelt* de grootte van de hashtabel. Elke index wordt dan één bit langer, zodat er twee nieuwe indices uit ontstaan. De tabelelementen bij beide indices moeten naar dezelfde pagina verwijzen als de oorspronkelijke index. Enkel bij de gesplitste pagina wijst het ene tabelelement naar de oude pagina, het andere naar de nieuwe.

  Als de hashwaarden echter niet te onderscheiden zijn op grond van bit $k+1$ dan komen al hun sleutels in dezelfde pagina terecht (oud of nieuw). We trachten ze dan te verdelen met de volgende bit, en indien nodig met nog meer bits, tot er minstens één sleutel in een van de betrokken pagina's terechtkomt. De grootte van de hashtabel kan daarbij telkens verdubbelen.

  Deze methode werkt niet als er $m$ gelijke hashwaarden in dezelfde pagina terechtgekomen zijn. Ook wanneer de hashwaarden in een pagina teveel identieke eindbits hebben, wordt de hashtabel onnodig groot. Voor beide problemen is het dus belangrijk dat de hashfunctie goed is, zodat de toegekende bits zo 'random' mogelijk zijn.[6]

  Een belangrijke eigenschap van extendible hashing is dat de inhoud van de hashtabel enkel afhangt van de *waarde* van de opgeslagen sleutels, en niet van de volgorde waarin die sleutels toegevoegd werden. (Dat is niet zo verwonderlijk, want dat is een eigenschap van de structuur van tries.)

- *Verwijderen.* Verwijderen is lastiger dan toevoegen, net zoals bij een B-tree. Wanneer een pagina en haar ooit afgesplitste buur samen minder dan $m$ sleutels bevatten, kan men ze samenvoegen. (Best *minder* dan $m$, om te vermijden dat de samengevoegde pagina snel weer moet gesplitst worden.) Er verdwijnt dan een pagina, en haar hashtabelelement moet dan naar de andere pagina verwijzen (waarvan de $k$-waarde met één

[6] Om een al te grote hashtabel te vermijden, legt 'bounded index exponential hashing' (Lomet, 1983) een maximumgrootte op aan die tabel, en verdubbelt daarna de grootte van de pagina's, waar nodig.

vermindert). Zodra de hashtabel minstens twee verwijzingen naar elke pagina bevat (elke $k$ is dan kleiner dan $d$), kan ze gehalveerd worden ($d$ vermindert met één).

Soms maakt men het zich gemakkelijk door pagina's met weinig elementen te tolereren, wat in de praktijk vaak goed werkt.

Wanneer er $n$ uniform verdeelde sleutels opgeslagen zijn, kan men aantonen dat de verwachtingswaarde van het aantal pagina's $n/(m \ln 2) \approx 1.44n/m$ bedraagt, zodat elke pagina gemiddeld voor ongeveer 69 procent gevuld is. (Hetzelfde resultaat als bij B-trees.) De grootte van de hashtabel heeft een verwachtingswaarde van ongeveer $3.92(n/m)\sqrt[m]{n}$. Voor praktische waarden van $m$ is $\sqrt[m]{n} \approx 1$, zodat dit resultaat ongeveer $4(n/m)$ bedraagt.

Als de hashtabel te groot uitvalt voor het inwendig geheugen, zijn er een aantal mogelijkheden:

- De hashtabel kan op analoge manier in twee niveau's geïmplementeerd worden. We houden dan de wortel in het geheugen, en indexeren hem met een kleiner aantal eindbits om de delen van de eigenlijke hashtabel in het uitwendig geheugen te vinden. Zoeken vereist dan een extra schijfoperatie.
- Als alternatief kunnen we het maximaal aantal gegevens $m$ in elke pagina vergroten, door er naast sleutels enkel wijzers naar de bijbehorende informatie in op te slaan. Er zijn dan minder pagina's nodig, en dus wordt de hashtabel kleiner. Maar door de wijzers vereist zoeken ook hier een extra schijfoperatie.

### 3.2.2 Linear hashing

Deze methode (Litwin 1980) heeft zelfs geen hashtabel meer nodig om de trie te elimineren, door pagina's met opeenvolgende adressen te gebruiken. De $d$ eindbits van de hashwaarde worden niet meer als index in een hashtabel, maar rechtstreeks als adres van een pagina gebruikt. (Omdat het vereiste aantal pagina's meestal moeilijk te schatten valt, gebruikt men opeenvolgende *logische* adressen, en laat het aan het operating system over om deze om te zetten in fysische adressen.) Bij een hashtabel mag het aantal pagina's kleiner zijn dan $2^d$, omdat verschillende wijzers naar dezelfde pagina mogen verwijzen, maar nu zijn er $2^d$ adressen, en dus even veel pagina's. Wanneer $d$ toeneemt verdubbelt de hashtabel, maar niet het aantal pagina's. Hier zou het aantal pagina's moeten verdubbelen. Daarom gaat men anders te werk.

Wanneer een pagina vol is wordt er gesplitst, maar niet noodzakelijk de volle pagina. Pagina's worden in *sequentiële* volgorde gesplitst, of ze nu vol zijn of niet. Als een andere dan de volle pagina gesplitst wordt, krijgt deze laatste een overflow pagina toegewezen voor de overtollige gegevens. Als dan uiteindelijk deze pagina aan de beurt is om te splitsen, worden zowel haar gegevens als die in haar overflow pagina verdeeld over de twee pagina's, aan de hand van het $(d+1)$-de eindbit van hun hashwaarden. Met als gevolg dat de adressen van de oude en de nieuwe pagina voortaan $d+1$ bits tellen. De adressen van de nog niet gesplitste pagina's blijven echter $d$ bits lang.

Ook linear hashing voorziet enkel woordenboekoperaties (zoals gewone hashing):

- *Zoeken.* De hashwaarde van de sleutel wordt berekend, maar we moeten weten hoeveel eindbits daarvan nodig zijn om de pagina te adresseren. Daartoe wordt het (logisch) adres van de volgende te splitsen pagina bijgehouden in een variabele $p$ (zie onder, bij toevoegen). Het adres gevormd door de $d$ eindbits wordt dan vergeleken met dat in $p$ (wat ook $d$ bits telt). Als het kleiner is, dan is de gezochte pagina reeds gesplitst en moeten $d+1$ eindbits voor haar adres gebruikt worden. Anders volstaan de $d$ bits.

  De pagina zelf bevat net zoals bij extendible hashing een gerangschikte tabel van sleutels (met bijbehorende informatie), en houdt het huidig aantal sleutels bij (maximaal $m$). De sleutel wordt dan gelokaliseerd via binair of linear zoeken. Het is mogelijk dat er in plaats van in de (volle) pagina, in haar overflowpagina moet gezocht worden.

  Bij extendible hashing vereist zoeken precies één schijfoperatie als de hashtabel in het inwendig geheugen zit, anders twee. Bij linear hashing vereist zoeken steeds één schijfoperatie, tenzij er ook in een of meerdere overflowpagina's moet gezocht worden. Voor heel veel gegevens is de performantie van extendible hashing dus minder goed dan het beste geval van linear hashing, maar beter dan het slechtste geval. Wanneer men ook bij extendible hashing overflowpagina's introduceert om splitsen uit te stellen, wordt de vergelijking moeilijker. (Bij linear hashing zijn overflowpagina's noodzakelijk, bij extendible hashing optioneel.)

- *Toevoegen.* Toevoegen is de pagina lokaliseren zoals bij zoeken, en als deze niet vol is, het gegeven tussenvoegen in de gerangschikte tabel.

  Als de pagina vol is moet er gesplitst worden. Het adres van de volgende te splitsen pagina zit in $p$ (zie hoger). Omdat splitsen in sequentiële volgorde gebeurt, is $p$ initieel nul, en wordt bij elke splitsing geïncrementeerd tot alle $2^d$ pagina's gesplitst zijn. Dan wordt $d$ geïncrementeerd, wordt $p$ opnieuw nul, en begint de volgende expansieronde. Wanneer pagina $p$ gesplitst wordt, wordt het adres van de nieuwe pagina $p + 2^d$.

  Als echter de volle pagina niet gesplitst werd, dan wordt het gegeven toegevoegd aan een (eventueel nieuwe) overflowpagina.

- *Verwijderen.* Verwijderen is de pagina lokaliseren zoals bij zoeken, en het gegeven uit de tabel halen (eventueel uit een overflowpagina). Onderbezette pagina's verdwijnen in omgekeerde volgorde als waarin ze gecreëerd werden. (Het is dus niet noodzakelijk de pagina waaruit verwijderd werd die verdwijnt.) De gegevens van zo'n pagina komen dan terecht in de pagina waarvan ze origineel afgesplitst werd. Om te vermijden dat die bij toevoegen snel weer gesplitst wordt, moet hun gezamenlijk aantal gegevens kleiner zijn dan $m$.