

# Gevorderde algoritmen

Bert De Saffel

Master in de Industriële Wetenschappen: Informatica Academiejaar 2018–2019

Gecompileerd op 18 oktober 2019

# Inhoudsopgave

<b>I</b>	<b>Gegevensstructuren II</b>	<b>3</b>
<b>1</b>	<b>Efficiënte zoekbomen</b>	<b>4</b>
1.1	Inleiding . . . . .	4
1.2	Rood-zwarte bomen . . . . .	5
1.2.1	Definitie en eigenschappen . . . . .	6
1.2.2	Zoeken . . . . .	6
1.2.3	Toevoegen en verwijderen . . . . .	6
1.2.4	Rotaties . . . . .	7
1.2.5	Bottom-up rood-zwarte bomen . . . . .	7
1.2.6	Top-down rood-zwarte bomen . . . . .	10
1.2.7	Vereenvoudigde rood-zwarte bomen . . . . .	11
1.3	Splaybomen . . . . .	12
1.3.1	Bottom-up splayboom . . . . .	12
1.3.2	Top-down splayboom . . . . .	13
1.3.3	Performantie van splay trees . . . . .	15
1.4	Gerandomiseerde zoekbomen . . . . .	17
1.5	Skip lists . . . . .	17
<b>2</b>	<b>Toepassingen van dynamisch programmeren</b>	<b>18</b>
2.1	Optimale binaire zoekbomen . . . . .	18
2.2	Langste gemeenschappelijke deelsequentie . . . . .	21
<b>3</b>	<b>Samenvoegbare heaps</b>	<b>23</b>
3.1	Binomiale queues . . . . .	23
3.1.1	Structuur . . . . .	23
3.1.2	Operaties . . . . .	23

3.2	Pairing heaps . . . . .	25
-----	-------------------------	----

Deel I

# Gegevensstructuren II

# Hoofdstuk 1

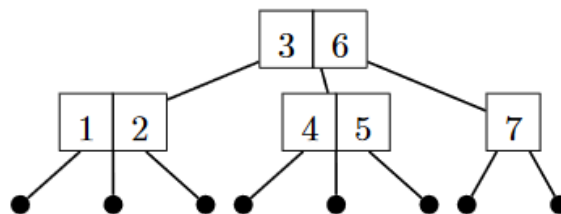
## Efficiënte zoekbomen

### 1.1 Inleiding

- Uitvoeringstijd van operaties (zoeken, toevoegen, verwijderen) op een binaire zoekboom met hoogte  $h$  is  $O(h)$ .
- De hoogte  $h$  is afhankelijk van de toevoegvolgorde van de  $n$  elementen:
  - In het slechtste geval bekomt men een gelinkte lijst, zodat  $h = O(n)$ .
  - Als elke toevoegvolgorde even waarschijnlijk is, dan is de verwachtingswaarde voor de hoogte  $h = O(\lg n)$ .
- **! Geen realistische veronderstelling.**
- Drie manieren om de efficiëntie van zoekbomen te verbeteren:
  1. **Elke operatie steeds efficiënt maken.** (Hoogte klein houden)
    - (a) AVL-bomen.
      - Hoogteverschil van de tweede deelbomen van elke knoop wordt gedefinieerd als:

$$\Delta h \leq 1$$

- $\Delta h$  wordt opgeslagen in de knoop zelf.
- (b) 2-3-bomen (figuur 1.1).

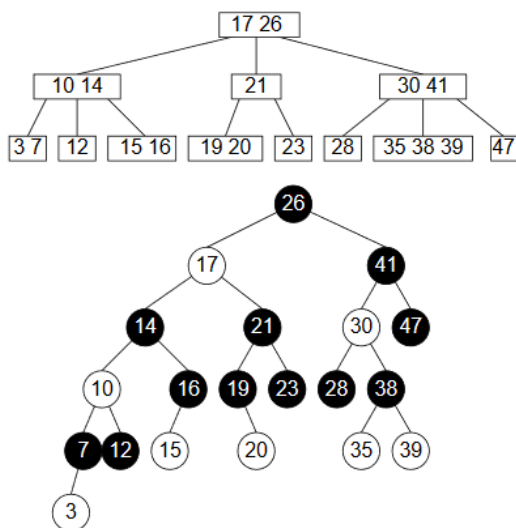


Figuur 1.1: Een 2-3-boom.

- Elke knoop heeft 2 of 3 kinderen en dus 1 of 2 sleutels.
- Elk blad heeft dezelfde diepte.
- Bij toevoegen of verwijderen wordt het ideale evenwicht behouden door het aantal kinderen van de knopen te manipuleren.

- (c) 2-3-4-bomen.
  - Analooq aan een 2-3-boom, maar elke knoop heeft 2, 3 of 4 kinderen.
  - ! Per knoop moet er plaats voorzien zijn voor 3 sleutels, wat onnodig veel geheugen vraagt.
- (d) Rood-zwarte bomen (sectie 1.2.1).
- 2. **Elke reeks operaties steeds efficiënt maken.**
  - (a) Splaybomen (sectie 1.3).
    - De vorm van de boom wordt meermaals aangepast.
    - Elke reeks opeenvolgende operaties is gegarandeerd efficiënt.
    - Een individuele operatie kan wel traag uitvallen.
    - *Geamortiseerd* is de performantie per operatie goed.
- 3. **De gemiddelde efficiëntie onafhankelijk maken van de operatievolgorde.**
  - (a) Gerandomiseerde zoekbomen (sectie 1.4).
    - Gebruik van een random generator.
    - De boom is random, onafhankelijk van de toevoeg- en verwijdervolgorde.
    - Verwachtingswaarde van de hoogte  $h$  wordt dan  $O(\lg n)$ .

## 1.2 Rood-zwarte bomen



Figuur 1.2: Een 2-3-4-boom en equivalent rood-zwarte boom (wit stelt hier rood voor).

- Simuleert een 2-3-4-boom (fig 1.2).
  - Een knoop in een 2-3-4 boom worden 1, 2 of 3 knopen in een rood-zwarte boom.
  - Een 2-knoop wordt een zwarte knoop.
  - Een 3-knoop wordt een zwarte knoop met een rood kind.
  - Een 4-knoop wordt een zwarte knoop met twee rode kinderen.
- Een rood-zwarte boom is gemakkelijker te definiëren als er afgestapt wordt van het 2-3-4-boom concept.

### 1.2.1 Definitie en eigenschappen

- **Definitie:**

- Een binaire zoekboom.
- Elke knoop is rood of zwart gekleurd.
- Elke virtuele knoop is zwart. Een virtuele knoop is een knoop die geen waarde heeft, maar wel een kleur. Deze vervangen de nullwijzers.
- Een rode knoop heeft steeds twee zwarte kinderen
- Elke mogelijke weg vanuit een knoop naar elke virtuele knoop bevat evenveel zwarte knopen. Dit aantal zwarte knopen wordt de *zwarte hoogte* genoemd
- De wortel is zwart.

- **Eigenschappen:**

- Een deelboom met zwarte hoogte  $z$  heeft tenminste  $2^z - 1$  inwendige knopen. Dit is de deelboom waarvan elke knoop zwart is.
- De hoogte  $h$  van een rood-zwarte boom met  $n$  knopen is steeds  $O(\lg n)$ , want:
  - ◊ Er zijn nooit twee opeenvolgende rode knopen op elke weg vanuit een knoop naar een virtuele knoop  $\rightarrow z \geq h/2$ .
  - ◊ Substitutie in de eerste eigenschap geeft:

$$\begin{aligned} n &\geq 2^z - 1 \geq 2^{h/2} - 1 \\ \rightarrow h &\leq 2 \lg(n + 1) \end{aligned}$$

### 1.2.2 Zoeken

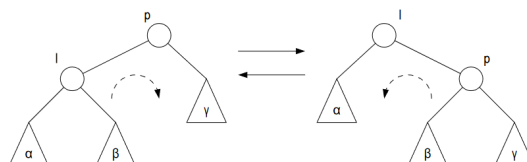
- De kleur speelt geen rol, zodat de rood-zwarte boom een gewone binaire zoekboom wordt.
- De hoogte van een rood-zwarte boom is wel geëigend  $O(\lg n)$ .
- Zoeken naar een willekeurige sleutel is dus  $O(\lg n)$ .

### 1.2.3 Toevoegen en verwijderen

- Element toevoegen of verwijderen, zonder rekening te houden met de kleur, is ook  $O(\lg n)$ .
- ! Geen garantie dat deze gewijzigde boom nog rood-zwart zal zijn.
- Twee manieren om toe te voegen:
  1. **Bottom-up:**
    - Voeg knoop toe zonder rekening te houden met de kleur.
    - Herstel de rood-zwarte boom, te beginnen bij de nieuwe knoop, en desnoods tot bij de wortel.
    - ! Er zijn ouderwijzers of een stapel nodig om naar boven in de boom te gaan.
    - ! Multithreading is niet mogelijk. Alle threads die een bottom-up rood-zwarte boom gebruiken moeten gelocked worden bij een toevoeg-of verwijderoperatie.
  2. **Top-down:**
    - Pas de boom aan langs de dalende zoekweg.

- ! Als de ouder van de toe te voegen knoop reeds zwart is, dan moet er niets aan de boom aangepast worden (want elke nieuwe knoop is altijd rood). Top-down houdt hier geen rekening mee en heeft toch reeds de boom aangepast tegen dat de knoop toegevoegd wordt.
- ✓ Geen ouderwijzers of stapel nodig.
- ✓ Multithreading wel mogelijk. Bij het afdalen naar elke knoop zijn enkel nog de deelbomen van die knoop nodig om de boom te herstellen.

### 1.2.4 Rotaties



Figuur 1.3: Rotaties

- Een rotatie wijzigt de vorm van de boom, maar behouden de in-order volgorde van de sleutels.
- Er moeten enkel pointers aangepast worden, en is dus  $O(1)$ .
- **Rechtste rotatie** van een ouder  $p$  en zijn linkerkind  $l$ :
  - Het rechterkind van  $l$  wordt het linkerkind van  $p$ .
  - De ouder van  $p$  wordt de ouder van  $l$ .
  - $p$  wordt het rechterkind van  $l$ .
- **Linkse rotatie** van een ouder  $p$  en zijn rechterkind  $r$ :
  - Het linkerkind van  $r$  wordt het rechterkind van  $p$ .
  - De ouder van  $r$  wordt de ouder van  $p$ .
  - $p$  wordt het linkerkind van  $l$ .

### 1.2.5 Bottom-up rood-zwarte bomen

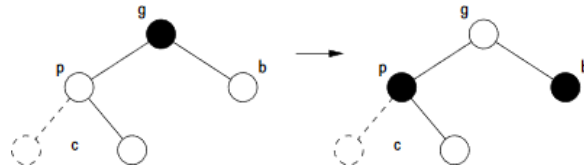
#### Toevoegen

- De knoop wordt eerst op de gewone manier toegevoegd.
- Welke kleur geven we die knoop?
  - **Zwart:** dit kan de zwarte hoogte van veel knopen ontregelen.
  - **Rood:** dit mag enkel als de ouder zwart is.
  - Kies voor rood omdat zwarte hoogte moeilijker te herstellen valt.
- Als de ouder zwart is, dan is toevoegen gelukt.
- Als de ouder rood is wordt deze storing verwijderd door rotaties en kleurwijzigingen door te voeren.



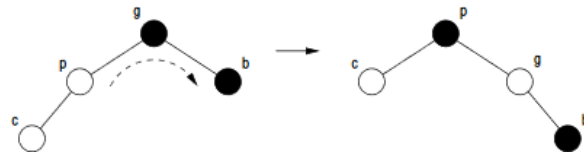
- Vaststellingen:
  - De ouder  $p$  van de nieuwe knoop  $c$  is rood.
  - De grootouder  $g$  van  $c$  is zwart want  $p$  is rood.
- Er zijn zes mogelijke gevallen, die twee groepen van drie vormen, naar gelang dat  $p$  een linker- of rechterkind is van  $g$ .
- We onderstellen dat  $p$  een linkerkind is van  $g$ .

1. **De broer  $b$  van  $p$  is rood.**



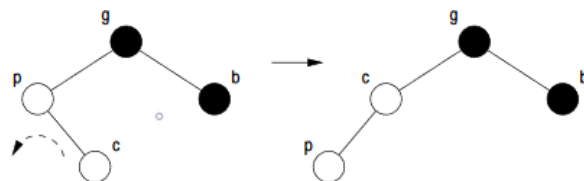
Figuur 1.4: Rode broer.

- Maak  $p$  en  $b$  zwart.
  - Maak  $g$  rood.
  - Als  $g$  een zwarte ouder heeft, is het probleem opgelost.
  - Als  $g$  een rode ouder heeft, zijn er opnieuw twee opeenvolgende rode knopen.
  - Het probleem wordt opgeschoven in de richting van de wortel.
2. **De broer  $p$  van  $p$  is zwart.**
- (a) **Knoop  $c$  is een linkerkind van  $p$ .**



Figuur 1.5: Rode broer.

- Roteer  $p$  en  $g$  naar rechts.
  - Maak  $p$  zwart.
  - Maak  $g$  rood.
- (b) **Knoop  $c$  is een rechterkind van  $p$ .**

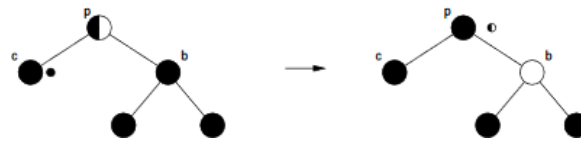


Figuur 1.6: Rode broer.

- Roteer  $p$  en  $c$  naar links.
  - We krijgen nu het vorige geval.
- Hoogstens 2 rotaties om de boom te herstellen, voorafgegaan door eventueel  $O(\lg n)$  opschuiven.
  - Roteren en opschuiven is  $O(1)$ , en afdalen is  $O(\lg n)$  zodat toevoegen  $O(\lg n)$  is.

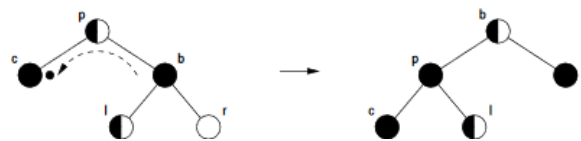
## Verwijderen

- Verwijder eerst de knoop zoals bij een gewone zoekboom.
- Als de te verwijderen knoop rood is, is er geen gevolg voor de zwarte hoogte en is de operatie klaar.
- Als de te verwijderen knoop zwart is, zijn er twee mogelijkheden:
  1. **De knoop heeft minstens één rood kind.** Dit rood kind kan de zwarte kleur overnemen, zodat de zwarte hoogten intact blijven.
  2. **De knoop heeft twee zwarte kinderen (virtueel of echt).** De zwarte kleur wordt aan één van de kinderen gegeven, zodat die **dubbelzwart** wordt.
- In het eerste geval is de operatie klaar. In het tweede geval moet de boom, die nu een dubbelzwarte knoop bevat, hersteld worden.
- Als de dubbelzwarte knoop  $c$  de wortel is, kan deze extra zwarte kleur verdwijnen.
- Als  $c$  geen wortel is, en ouder  $p$  heeft, dan zijn er acht mogelijkheden die in groepen van twee uiteenvallen naargelang  $c$  een linker- of rechterkind van  $p$  is.
- We veronderstellen dat  $c$  een linkerkind is van  $p$ .
  1. **De broer  $b$  van  $c$  is zwart.** De kleur van  $p$  is willekeurig. Hier zijn er drie gevallen mogelijk, afhankelijk van de kleur van de kinderen van  $b$ .
    - (a) **Broer  $b$  heeft twee zwarte kinderen.**



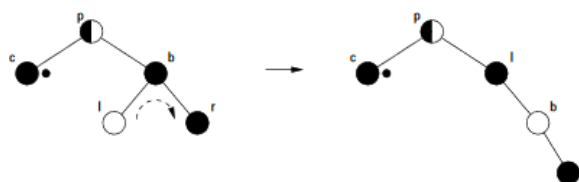
Figuur 1.7

- Knoop  $b$  kan rood worden.
- De extra zwarte kleur van  $c$  kan aan  $p$  gegeven worden.
  - ◊ Als  $p$  rood was, dan is de operatie gelukt.
  - ◊ Als  $p$  reeds zwart was, dan verschuift het probleem zich naar boven.
- (b) **Broer  $b$  heeft een rood rechterkind.** De kleur van het linkerkind  $l$  van  $b$  is willekeurig.



Figuur 1.8

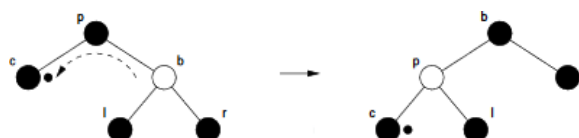
- Roteer  $p$  en  $b$  naar links.
- Knoop  $p$  krijgt de extra zwarte kleur van  $c$ .
- Het rechterkind  $r$  van  $b$  wordt zwart.
- Knoop  $b$  krijgt de oorspronkelijke kleur van  $p$ .
- (c) **Broer  $b$  heeft een zwarte rechterkind en een rood linkerkind.**
  - Roteer  $b$  en  $l$  naar rechts.



Figuur 1.9

- Maak  $b$  rood en  $l$  zwart.
- Dit is nu het vorige geval.

## 2. De broer $b$ van $c$ is rood.



Figuur 1.10

- Roteer  $p$  en  $b$  naar links.
- Maak  $b$  zwart en  $p$  rood.
- Dit is nu het eerste geval.
- Hoogstens 3 rotaties nodig om de boom te herstellen, voorafgegaan door eventueel  $O(\lg n)$  opschuivingen.
- Roteren en opschuiven is  $O(1)$ , en afdalen is  $O(\lg n)$  zodat verwijderen  $O(\lg n)$  is.

## 1.2.6 Top-down rood-zwarte bomen

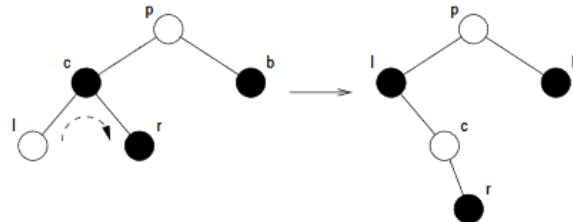
### Toevoegen

- Ook hier worden nieuwe knopen rood gemaakt.
- Op de weg naar beneden mogen er geen rode broers zijn.
- Als we een **zwarte knoop met twee rode kinderen** tegenkomen, dan maken we die knoop rood en zijn kinderen zwart.
- Als zijn ouder rood is, kan dit met rotaties en kleurwijzigingen opgelost worden.
- Toevoegen daalt enkel in de boom en is  $O(\lg n)$ .

### Verwijderen

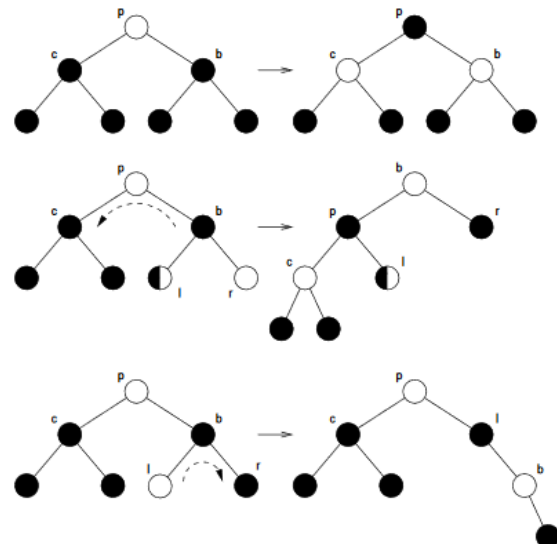
- De zwarte hoogte van de fysisch te verwijderen knoop is één, omdat minstens één van zijn kinderen virtueel is.
- Om geen problemen te krijgen met de zwarte hoogte moet deze knoop rood zijn, maar dan moet zijn tweede kind ook virtueel zijn.
- De zoekknoop kan eender waar in de boom zitten, daarom wordt elke volgende knoop op de zoekweg rood gemaakt.

- Tijdens het afdalen komen we in een rode of rood gemaakte knoop  $p$ .
  - Die heeft dan zeker een zwart kind  $c$ , dat rood moet worden.
  - Er zijn acht mogelijkheden die in groepen van twee uiteenvallen naargelang  $c$  een linker- of rechterkind van  $p$  is.
- We veronderstellen dat  $c$  een linkerkind is van  $p$ .
1. **Knoop  $c$  heeft minstens één rood kind.**



Figuur 1.11

- Als we naar een rode knoop moeten afdalen zitten we terug in de beginsituatie.
  - Als  $c$  de fysisch te verwijderen knoop is of als we naar een zwarte knoop moeten afdalen:
    - ◊ Roteer  $c$  samen met zijn rood kind zodat  $c$  nu als ouder zijn oorspronkelijk rood kind heeft.
    - ◊ Wijzig de kleur van  $c$  naar zwart.
    - ◊ Wijzig de kleur van zijn oorspronkelijk kind naar rood.
2. **Knoop  $c$  heeft twee zwarte kinderen.**



Figuur 1.12

### 1.2.7 Vereenvoudigde rood-zwarte bomen

- De implementatie is omslachtig door de talrijke speciale gevallen.

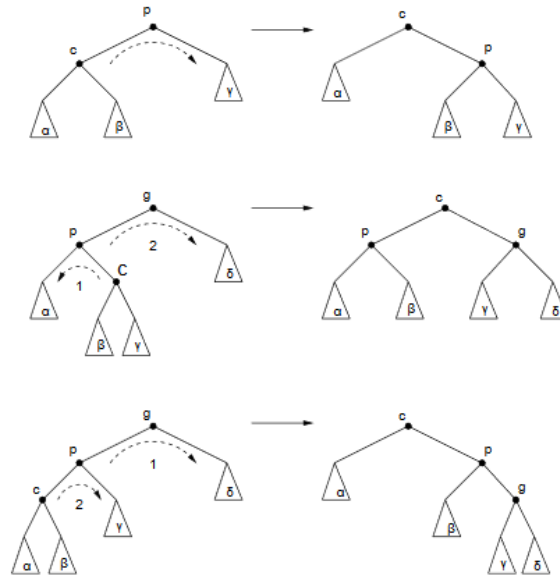
- Eenvoudigere varianten bestaan:
  - Een **AA-boom** geeft aan dat enkel een rechterkind rood moet zijn.
  - Een **Binary B-tree** beperkt het aantal gevallen maar behouden toch de asymptotische efficiëntie.
  - Een **left-leaning red-black-tree** stelt de eis dat een zwarte knoop enkel een rood rechterkind mag hebben als het reeds een rood linkerkind heeft.

## 1.3 Splaybomen

- Garanderen dat elke reeks opeenvolgende operaties efficiënt is.
- Als we  $m$  operaties verrichten op de splay tree, waarbij  $n$  keer toevoegen, dan is de performantie van deze reeks  $O(m \lg n)$ .
- Uitgemiddeld is dit  $O(\lg n)$ .
- Individuele operaties mogen inefficiënt zijn, maar de boom moet zo aangepast worden zodat een reeks van die operaties efficiënt zijn.
- **Basisidee:** Elke knoop die gezocht wordt, toegevoegd of verwijderd wordt, zal de wortel worden van de boom, zodat opeenvolgende operaties op die knoop efficiënt zijn.
- Een willekeurige knoop tot wortel maken gebeurt via de **splay-operatie**.
- De weg naar een diepe knoop bevat knopen die ook diep liggen. Terwijl we een knoop wortel maken, moeten de knopen op het zoekpad ook aangepast worden, zodat ook de toegangstijd van deze knopen verbetert, anders blijft de kans bestaan dat een reeks van operaties inefficiënt is.
- Er moet geen extra informatie bijgehouden worden voor knopen, wat geheugen uitspaart.
- De splay-operatie is gedefinieerd voor zowel bottom-up als top-down splaybomen.

### 1.3.1 Bottom-up splayboom

- De knoop wordt eerst gezocht zoals bij een gewone zoekboom.
  - De splay-operatie gebeurt van onder naar boven.
  - Een knoop kan naar boven gebracht worden door hem telkens te roteren met zijn ouder.
  - Om de toegangstijd van knopen op de zoekweg ook te verbeteren, zijn er drie verschillende mogelijkheden:
    1. **De ouder  $p$  van  $c$  is wortel.**
      - Roteer beide knopen zodat  $c$  de wortel wordt.
    2. **Knoop  $c$  heeft nog een grootouder.**
      - Er zijn vier gevallen, die uitvallen in groepen van twee, naar gelang dat  $p$  een linkerkind of rechterkind is van grootouder  $g$ .
      - We veronderstellen dat  $p$  linkerkind is van  $g$ .
- (a) **Knoop  $c$  is een rechterkind van  $p$ .**
- Roteer  $p$  en  $c$  naar links.

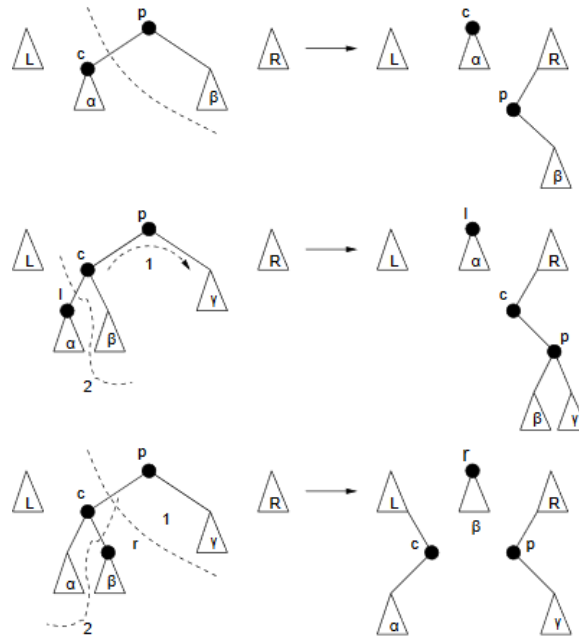


Figuur 1.13: Bottom-up splay.

- Roteer  $g$  en  $c$  naar rechts.
- (b) **Knoop  $c$  is een linkerkind van  $p$ .**
  - Roteer  $g$  en  $p$  naar rechts.
  - Roteer  $p$  en  $c$  naar rechts.
- De **woordenboekoperaties verlopen nu als volgt:**
  - **Zoeken.** De knoop wordt eerst gezocht zoals een gewone zoekboom. Daarna wordt deze tot wortel gemaakt via de splay-operatie.
  - **Toevoegen.** Toevoegen gebeurt ook zoals een gewone zoekboom. De nieuwe knoop wordt dan tot wortel gemaakt met de splay-operatie.
  - **Verwijderen.** Verwijderen gebeurt ook zoals een gewone zoekboom. Daarna wordt de ouder van die knoop tot wortel gemaakt met de splay-operatie.

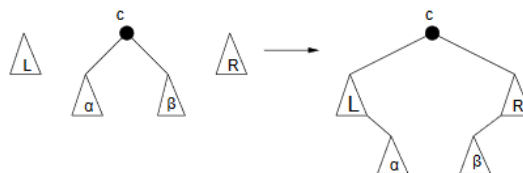
### 1.3.2 Top-down splayboom

- De splayoperatie wordt uitgevoerd tijdens de afdaling, zodat de gezochte knoop wortel zal zijn als we hem bereiken.
- De boom wordt in drie zoekbomen opgedeeld,  $L$ ,  $M$  en  $R$ .
  - Alle sleutels in  $L$  zijn kleiner dan die in  $M$ .
  - Alle sleutels in  $R$  zijn groter dan die in  $M$ .
- Eerst is  $M$  de oorspronkelijke boom en zijn  $L$  en  $R$  ledig.
- De huidige knoop op de zoekweg is steeds de wortel van  $M$ .
- Stel dat we bij een knoop  $p$  uitkomen, en dan nog verder moeten naar een knoop  $c$ .
- Er zijn dan twee groepen van 3 gevallen, afhankelijk of  $c$  een linker- of rechterkind is van  $p$ .



Figuur 1.14: Top-down splay.

- We veronderstellen dat  $c$  een linkerkind is van  $p$ .
  1. **Knoop  $c$  is de laatste knoop op de zoekweg.**
    - Knoop  $p$  wordt het nieuwe kleinste element in  $R$  samen met zijn rechtse deelboom.
    - Knoop  $c$  wordt de wortel van  $M$ .
  2. **Knoop  $c$  is niet de laatste knoop op de zoekweg.**
    - **We moeten verder afdalen naar het linkerkind  $l$  van  $c$ .**
      - ◊ Roteer  $p$  en  $c$  naar rechts.
      - ◊ Knoop  $c$  wordt het kleinste element in  $R$  samen met de rechtse deelboom van  $c$ .
      - ◊ De linkse deelboom van  $c$  wordt de nieuwe  $M$  met als wortel  $l$ .
    - **We moeten verder afdalen naar het rechterkind  $r$  van  $c$ .**
      - ◊ Knoop  $p$  wordt het kleinste element in  $R$  samen met de rechtse deelboom van  $p$ .
      - ◊ Knoop  $c$  wordt het nieuwe grootste element in  $L$ .
      - ◊ De rechtse deelboom van  $c$  wordt de nieuwe  $M$  met als wortel  $r$ .
- Als de gezochte knoop  $c$  wortel van  $M$  is, wordt de splayoperatie afgerond met een **join-operatie**.



Figuur 1.15: Samenvoegen na top-down splayen.

- De **woordenboekoperaties verlopen nu als volgt:**

- **Zoeken.** De knoop met de gezochte sleutel wordt tot wortel gemaakt. Als de sleutel niet gevonden wordt dan is zijn opvolger of voorloper de wortel.
- **Toevoegen.**
- **Verwijderen.**

### 1.3.3 Performantie van splay trees

- Niet eenvoudig aangezien vorm van de boom vaak verandert.
- We willen aantonen dat een reeks van  $m$  operaties op een splay tree met maximaal  $n$  knopen een performantie van  $O(m \lg n)$  heeft.
- Er wordt een **potentiaalfunctie**  $\Phi$  gebruikt.
- Elke mogelijke vorm van een splayboom krijgt een reëel getal toegewezen aan de hand van deze potentiaalfunctie.
- Efficiënte operaties die minder tijd gebruiken dan de geamortiseerde tijd per operatie doen het potentiaal stijgen.
- Niet-efficiënte operaties die meer tijd gebruiken dan de geamortiseerde tijd per operatie doen het potentiaal dalen.
- De geamortiseerde tijd van een operatie wordt gedefinieerd als de som van haar werkelijke tijd en de toename van het potentiaal.
  - Stel  $t_i$  de werkelijke tijd van de  $i$ -de operatie.
  - Stel  $a_i$  de geamortiseerde tijd van die operatie.
  - Stel  $\Phi_i$  het potentiaal na deze operatie.

$$\rightarrow a_i = t_i + \Phi_i - \Phi_{i-1}$$

- De geamortiseerde tijd van een reeks  $m$  operaties is de som van de individuele geamortiseerde tijden:

$$\begin{aligned} \sum_{i=1}^m a_i &= \sum_{i=1}^m (t_i + \Phi_i - \Phi_{i-1}) \\ &= t_1 + \Phi_1 - \Phi_0 + t_2 + \Phi_2 - \Phi_1 + t_3 + \Phi_3 - \Phi_2 + \cdots + t_m + \Phi_m - \Phi_{m-1} \\ &= \Phi_m - \Phi_0 + \sum_{i=1}^m t_i \end{aligned}$$

- Als de potentiaalfunctie zo gekozen wordt zodat het eindpotentiaal  $\Phi_m$  zeker niet kleiner is dan de beginpotentiaal  $\Phi_0$ , dan vormt de totale geamortiseerde tijd een **bovengrens** van de werkelijke tijd want de boom zal zeker niet slechter zijn.
- De eenvoudigste potentiaalfunctie geeft voor elke knoop  $i$  een gewicht  $s_i$  die gelijk is aan het aantal knopen in de deelboom waarvan hij wortel is. De potentiaal van de boom is dan de som van de logaritmen van deze gewichten:

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

- We noemen  $\lg s_i$  de rang  $r_i$  van knoop  $i$ .



- Performantie-analyse van bottom-up splayboom:
  - Performantie is evenredig met de diepte van de knoop, en dus met het aantal uitgevoerde rotaties.
  - We willen aantonen 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 is en  $r_c$  de rang van de gezochte knoop.

- ◊ Als  $c$  reeds de wortel is, dan is  $r_w = r_c$  en blijft het potentiaal dezelfde.

$$O(1 + 3(r_w - r_c)) = O(1)$$

- ◊ Anders moeten zoveel splay-operaties uitgevoerd worden als de diepte van de knoop (moet niet gekend zijn).
  - \* Een zig wijzigt de rang van  $c$  en  $p$

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

- \* Een zig-zag wijzigt de rang van  $c$ ,  $p$  en  $g$

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

- \* Een zig-zig wijzigt de rang van  $c$ ,  $p$  en  $g$

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

- De bovengrenzen voor de drie operaties bevatten dezelfde positieve term  $r'_c - r_c$  maar met verschillende coëfficiënten.
- De totale geamortiseerde tijd is een som van dergelijke bovengrenzen, maar kan niet vereenvoudigd worden als coëfficiënten niet gelijk zijn.
- Aangezien het bovengrenzen zijn, wordt de grootste coëfficiënt genomen.
- In de som vallen de meeste termen nu weg, behalve de rang van  $c$  voor en na de volledige splay-operatie.
- De geamortiseerde tijden van de woordenboekoperaties op een bottom-splay tree met  $n$  knopen zijn nu:

- ◊ **Zoeken.**  $O(1 + 3 \lg n)$  want  $s_w = n$ .

- ◊ **Toevoegen.**  $O(1 + 4 \lg n)$ .

Op de zoekweg worden de rang van knopen  $p_1, p_2, \dots, p_k$  op de zoekweg gewijzigd. Stel  $s_{p_i}$  het gewicht van knoop  $p_i$  voor het toevoegen en  $s'_{p_i}$  het gewicht van knoop  $p_i$  na het toevoegen. De potentiaaltoename is dan

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

Deze is nooit groter dan

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

- ◊ **Verwijderen.** Het effect van verwijderen is nooit positief.

- De geamortiseerde tijd voor een reeks van  $m$  woordenboekoperaties is de som van de geamortiseerde tijden voor de individuele operaties.
- Stel  $n_i$  het aantal knopen bij de  $i$ -de operatie wordt die tijd  $O(m + 4 \sum_{i=1}^m \lg n_i) = O(m + 4m \lg n) = O(m \lg n)$ .

## 1.4 Gerandomiseerde zoekbomen

- De performantie van de woordenboekoperaties op een gewone zoekboom is  $O(\lg n)$  als elke toevoegvolgorde even waarschijnlijk is.
- Gerandomiseerde zoekbomen maken gebruik van een random generator om de operatievolgorde te neutraliseren.
- Deze bomen blijven steeds random.
- Een **treap** is een gerandomiseerde zoekboom.
  - Elke knoop krijgt naast een sleutel ook een prioriteit, die door de random generator wordt toegekend als de knoop toegevoegd wordt.
  - De prioriteiten van de knopen voldoen aan de heapvoorwaarde: de prioriteit van een kind is maximaal even hoog als die van zijn ouder.
- De woordenboekoperaties:
  - **Zoeken.** Zoeken moet geen rekening houden met de prioriteiten en verloopt zoals een normale binaire zoekboom.
  - **Toevoegen.** Eerst wordt er normaal toegevoegd. De knoop wordt nadien naar boven geroteerd om aan de heapvoorwaarde te voldoen.
  - **Verwijderen.** De te verwijderen knoop krijgt de laagste prioriteit, zodat die naar beneden geroteerd wordt. Dit blad kan dan verwijderd worden.

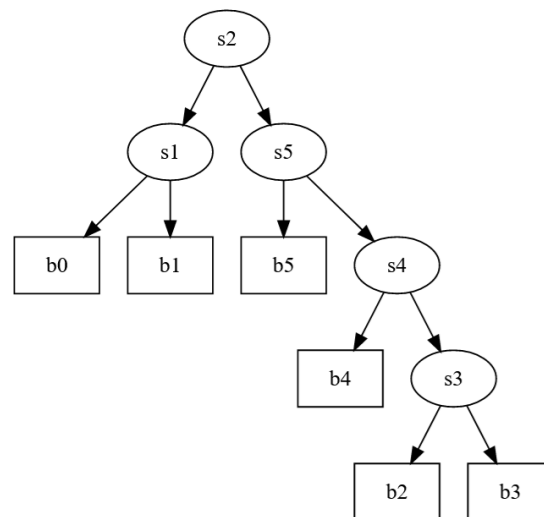
## 1.5 Skip lists

- Een meerwegszoekboom geïmplementeerd met gelinkte lijsten.
- Alle bladeren zitten op dezelfde diepte.
- Elke lijstknoop heeft plaats voor één sleutel en één kindwijzer.
- Een knoop met  $k$  kinderen bevat  $k - 1$  sleutels, zodat er één sleutelplaats over blijft.
- (zoekt gewoon eens een foto op)

## Hoofdstuk 2

# Toepassingen van dynamisch programmeren

### 2.1 Optimale binaire zoekbomen



Figuur 2.1: Een optimale binaire zoekboom met bijhorende kansentabel.

i	0	1	2	3	4	5
$p_i$		0.15	0.10	0.05	0.10	0.20
$q_i$	0.05	0.10	0.05	0.05	0.05	0.10

- Veronderstel dat de gegevens die in een binaire zoekboom moeten opgeslaan worden op voorhand gekend zijn.
- Veronderstel ook dat de waarschijnlijkheid gekend is waarmee de gegevens gezocht zullen worden.
- Tracht de boom zo te schikken zodat de verwachtingswaarde van de zoektijd minimaal wordt.
- De zoektijd wordt bepaald door de lengte van de zoekweg.

- De gerangschikte sleutels van de  $n$  gegevens zijn  $s_1, \dots, s_n$ .
- De  $n + 1$  bladeren zijn  $b_0, \dots, b_n$ .
  - Elk blad staat voor een afwezig gegeven die in een deelboom op die plaats had moeten zitten.
  - Het blad  $b_0$  staat voor alle sleutels kleiner dan  $s_1$ .
  - Het blad  $b_n$  staat voor alle sleutels groter dan  $s_n$ .
  - Het blad  $b_i$  staat voor alle sleutels groter dan  $s_i$  en kleiner dan  $s_{i+1}$ , met  $1 \leq i < n$
- De waarschijnlijkheid om de  $i$ -de sleutel  $s_i$  te zoeken is  $p_i$ .
- De waarschijnlijkheid om alle afwezige sleutels, voorgesteld door een blad  $b_i$ , te zoeken is  $q_i$ .
- De som van alle waarschijnlijkheden moet 1 zijn:

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

- Als de zoektijd gelijk is aan het aantal knopen op de zoekweg (de diepte plus één), dan is de verwachtingswaarde van de zoektijd van een binaire boom

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

- Deze verwachtingswaarde moet geminimaliseerd worden.
  - Boom met minimale hoogte is niet voldoende.
  - Alle mogelijke zoekbomen onderzoeken is ook geen optie, hun aantal is

$$\begin{aligned} \frac{1}{n+1} \binom{2n}{n} &\sim \frac{1}{n+1} \cdot \frac{2^{2n}}{\sqrt{\pi n}} \\ &\sim \frac{1}{n+1} \cdot \frac{4^n}{\sqrt{\pi n}} \\ &\sim \Omega\left(\frac{4^n}{n\sqrt{n}}\right) \end{aligned}$$

✓ Dynamisch programmeren biedt een uitkomst.

- Een optimalisatieprobleem komt in aanmerkingen voor een efficiënte oplossing via dynamisch programmeren als het:
  1. het een **optimale deelstructuur** heeft;
  2. de **deelp Problemen onafhankelijk maar overlappend** zijn.
- Is dit hier van toepassing?
  - Is er een optimale deelstructuur?
    - ◊ Als een zoekboom optimaal is, moeten zijn deelbomen ook optimaal zijn.
    - ◊ Een optimale oplossing bestaat uit optimale oplossingen voor deelp Problemen.
  - Zijn de deelp Problemen onafhankelijk?
    - ◊ Ja want deelbomen hebben geen gemeenschappelijke knopen.
  - Zijn de deelp Problemen overlappend?

- ◊ Elke deelboom bevat een reeks opeenvolgende sleutels  $s_i, \dots, s_j$  met bijhorende bladeren  $b_{i-1}, \dots, b_j$ .
- ◊ Deze deelboom heeft een wortel  $s_w$  waarbij  $(i \leq w \leq j)$ .
- ◊ De linkse deelboom bevat de sleutels  $s_i, \dots, s_{w-1}$  en bladeren  $b_{i-1}, \dots, b_{w-1}$ .
- ◊ De rechtse deelboom bevat de sleutels  $s_{w+1}, \dots, s_j$  en bladeren  $b_w, \dots, b_j$ .
- ◊ Voor een optimale deelboom met wortel  $s_w$  moeten deze beide deelbomen ook optimaal zijn.
- ◊ Deze wordt gevonden door:
  1. achtereenvolgens elk van zijn sleutels  $s_i, \dots, s_j$  als wortel te kiezen;
  2. de zoektijd voor de boom te berekenen door gebruikte maken van de zoektijden van de optimale deelbomen;
  3. de wortel te kiezen die de kleinste zoektijd oplevert.
- De kleinst verwachte zoektijd van een boom met sleutels  $s_i, \dots, s_j$  is  $z(i, j)$  en moet voor elke deelboom bepaald worden, dus voor alle  $i$  en  $j$  waarbij:
  - $1 \leq i \leq n + 1$
  - $i - 1 \leq j \leq n$
- De optimale boom heeft dus de kleinste verwachte zoektijd  $z(1, n)$ .
- Hoe  $z_w(i, j)$  bepalen voor een deelboom met wortel  $s_w$ ?
  - Gebruik de kans om in de wortel te komen.
  - Gebruik de optimale zoektijden van zijn deelbomen,  $z(i, w - 1)$  en  $z(w + 1, j)$ .
  - Gebruik ook de diepte van elke knoop, maar elke knoop staat nu op een niveau lager.
    - ◊ De bijdrage tot de zoektijd neemt toe met de som van de zoekwaarschijnlijkheden.

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

- Hieruit volgt:

$$\begin{aligned} z_w(i, j) &= p_w + (z(i, w - 1) + g(i, w - 1)) + (z(w + 1, j) + g(w + 1, j)) \\ &= z(i, w - 1) + z(w + 1, j) + g(i, j) \end{aligned}$$

- Dit moet minimaal zijn  $\rightarrow$  achtereenvolgens elke sleutel van de deelboom tot wortel maken.
  - De index  $w$  doorloopt alle waarden tussen  $i$  en  $j$ .

$$\begin{aligned} z(i, j) &= \min_{i \leq w \leq j} \{z_w(i, j)\} \\ &= \min_{i \leq w \leq j} \{z(i, w - 1) + z(w + 1, j)\} + g(i, j) \end{aligned}$$

- Hoe wordt de optimale boom bijgehouden?
  - Hou enkel de index  $w$  bij van de wortel van elke optimale deelboom.
  - Voor de deelboom met sleutels  $s_i, \dots, s_j$  is de index  $w = r(i, j)$ .
- Implementatie:
  - Een recursieve implementatie zou veel deeloplossingen opnieuw berekenen.
  - Daarom **bottom-up** implementeren. Maakt gebruik van drie tabellen:
    1. Tweedimensionale tabel  $z[1..n + 1, 0..n]$  voor de waarden  $z(i, j)$ .

- 2. Tweedimensionale tabel  $g[1..n+1, 0..n]$  voor de waarden  $g(i, j)$ .
- 3. Tweedimensionale tabel  $r[1..n, 1..n]$  voor de indices  $r(i, j)$ .
- Algoritme:
  1. Initialiseer de waarden  $z(i, i-1)$  en  $g(i, i-1)$  op  $q[i-1]$ .
  2. Bepaal achtereenvolgens elementen op elke evenwijdige diagonaal, in de richting van de tabelhoek linksboven.
    - ◊ Voor  $z(i, j)$  zijn de waarden  $z(i, i-1), z(i, i), \dots, z(i, j-1)$  van de linkse deelboom nodig en de waarden  $z(i+1, j), \dots, z(j, j), z(j+1, j)$  van de rechtse deelboom nodig.
    - ◊ Deze waarden staan op diagonalen onder deze van  $z(i, j)$ .
- Efficiëntie:
  - **Bovengrens:** drie verneste lussen  $\rightarrow O(n^3)$ .
  - **Ondergrens:**
    - ◊ Meeste werk bevindt zich in de binneste lus.
    - ◊ Een deelboom met sleutels  $s_i, \dots, s_j$  heeft  $j-i+1$  mogelijke wortels.
    - ◊ Elke test is  $O(1)$ .
    - ◊ Dit werk is evenredig met

$$\begin{aligned}
 & \sum_{i=1}^n \sum_{j=i}^n (j-i+1) \\
 \Rightarrow & \sum_{i=1}^n i^2 \\
 \Rightarrow & \frac{n(n+1)(2n+1)}{6} \\
 \Rightarrow & \Omega(n^3)
 \end{aligned}$$

- **Algemeen:**  $\Theta(n^3)$ .
- Kan met een bijkomende eigenschap (zien we niet in de cursus) gereduceerd worden tot  $\Theta(n^2)$ .

## 2.2 Langste gemeenschappelijke deelsequentie

- Een deelsequentie van een string wordt bekomen door nul of meer stringelementen weg te laten.
- Elke deelstring is een deelsequentie, maar niet omgekeerd.
- Een langste gemeenschappelijke deelsequentie (LGD) van twee strings kan nagaan hoe goed deze twee strings op elkaar lijken.
- Geven twee strings:
  - $X = \langle x_0, x_1, \dots, x_{n-1} \rangle$
  - $Y = \langle y_0, y_1, \dots, y_{m-1} \rangle$
- Hoe LGD bepalen?
  - Is er een optimale deelstructuur?
    - ◊ De deelproblemen zijn paren prefixen van de twee strings. De oplossing bij elk deelprobleem is de lengte van de langst gemeenschappelijke deelsequentie van deze twee prefixen.

- ◊ Het prefix van  $X$  met lengte  $i$  is  $X_i$ .
  - ◊ Het prefix van  $Y$  met lengte  $j$  is  $Y_j$ .
  - ◊ De ledige prefix is  $X_0$  en  $Y_0$ .
- Zijn de deelproblemen onafhankelijk?
  - ◊ Stel  $Z = \langle z_0, z_1, \dots, z_{k-1} \rangle$  de LGD van  $X$  en  $Y$ . Er zijn drie mogelijkheden:
    1. Als  $n = 0$  of  $m = 0$  dan is  $k = 0$ .
    2. Als  $x_{n-1} = y_{m-1}$  dan is  $z_{k-1} = x_{n-1} = y_{m-1}$  en is  $Z$  een LGD van  $X_{n-1}$  en  $Y_{m-1}$ .
    3. Als  $x_{n-1} \neq y_{m-1}$  dan is  $Z$  een LGD van  $X_{n-1}$  en  $Y$  of een LGD van  $X$  en  $Y_{m-1}$ .
- Zijn de deelproblemen overlappend?
  - ◊ Om de LGD van  $X$  en  $Y$  te vinden is het nodig om zowel de LGD van  $X$  en  $Y_{m-1}$  als van  $X_{n-1}$  en  $Y$  te vinden.
- De lengte  $c[i, j]$  van de LGD van  $X_i$  en  $Y_j$  wordt door een recursieve vergelijking bepaald:
 
$$c[i, j] = \begin{cases} 0 & \text{als } i = 0 \text{ of } j = 0 \\ c[i-1, j-1] + 1 & \text{als } i > 0 \text{ en } j > 0 \text{ en } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{als } i > 0 \text{ en } j > 0 \text{ en } x_i \neq y_j \end{cases}$$
- De lengte van de LGD komt overeen met  $c[n, m]$ .
- De waarden  $c[i, j]$  kunnen eenvoudig per rij van links naar rechts bepaald worden door de recursierelatie.
- Efficiëntie:
  - We beginnen de tabel in te vullen vanaf  $c[1, 1]$  (als  $i = 0$  of  $j = 0$  zijn de waarden 0).
  - De tabel  $c$  wordt rij per rij, kolom per kolom ingevuld.
  - De vereiste plaats en totale performantie is beiden  $\Theta(nm)$ .

## Hoofdstuk 3

# Samenvoegbare heaps

- Een samenvoegbare heap is een heap waarbij de samenvoegoperatie, het samenvoegen van twee heaps, efficiënt is zodanig dat de **heapvoorwaarde** nog steeds geldig is.
- De belangrijke samenvoegbare heaps zijn: **Binomial queues** en **Pairing heaps**.

### 3.1 Binomiale queues

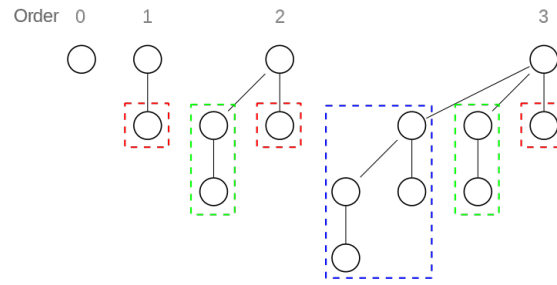
#### 3.1.1 Structuur

- Bestaat uit bos van binomiaalbomen.
- Een binomiaalboom  $B_h$  wordt recursief in functie van zijn hoogte  $h$  gedefinieerd.:
  - $B_0$  bestaat uit één knoop.
  - $B_h$  bestaat uit twee  $B_{h-1}$  bomen.
- De complete boom heeft  $2^h$  knopen, en op diepte  $d$  zijn er  $\binom{h}{d}$  knopen.
- Figuur 3.1 toont een aantal binomiaalbomen.
- Een prioriteitswachtrij met 13 elementen wordt voorgesteld als  $\langle B_3, B_2, B_0 \rangle$  want  $2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 13$ .
  - Er kan ook een binaire representatie gekozen worden:  $13 = (1101)_2$ . De bits die op 1 staan duiden een aanwezige binomiaalboom.

#### 3.1.2 Operaties

- **Minimum vinden:** Overloop de wortel van elke binomiaalboom. Het minimum kan ook gewoon bijgehouden worden.
- **Samenvoegen:** Tel de bomen met dezelfde hoogte bij elkaar op,  $B_h + B_h = B_{h+1}$ . Maak de wortel met de grootste sleutel het kind van deze met de kleinste. Bij het optellen moet er wel rekening gehouden worden met eventuele overdrachten.
  - **Voorbeeld:**
  - Er is een prioriteitswachtrij met 23 elementen  $= \langle B_4, B_2, B_1, B_0 \rangle$





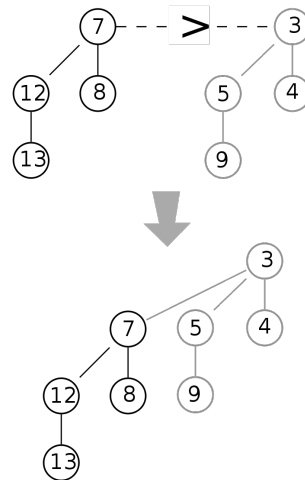
Figuur 3.1: Verschillende ordes van binomiaalbomen.

- Er is een prioriteitswachtrij met 13 elementen =  $\langle B_3, B_2, B_0 \rangle$
- Optellen geeft:

$$\begin{array}{ccccccc}
 & & B_4 & & B_3 & & B_2 & & B_1 & & B_0 \\
 & & B_4 & & & & B_2 & & B_1 & & B_0 \\
 & & & & B_3 & & B_2 & & & & B_0 \\
 \hline
 & & B_5 & & & & B_2 & & & & 
 \end{array}$$

Tabel 3.1: De binomiaalbomen boven de gestreepte lijn duiden de overdrachten aan.

Figuur 3.2 toont de samenvoegoperatie voor twee  $B_2$  bomen.

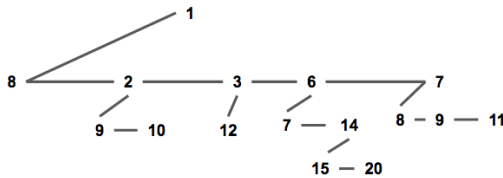


Figuur 3.2: Hier worden twee binomiaalbomen  $B_2$  samengevoegd. De boom met de waarde 7 voor de wortel wordt het linkerkind van de boom met waarde 3 voor de wortel. Het wordt een boom van orde  $B_3$ .

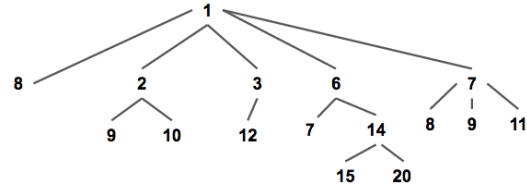
- **Toevoegen:** Maak een triviale binomiaalqueue met één knoop en voeg deze samen met de andere binomiaalqueue.
- **Minimum verwijderen:** Zoek binomiaalboom  $B_k$  met het kleinste wortelelement. Verwijder deze uit de binomiaalqueue. De deelbomen van deze binomiaalboom vormen een nieuw binomiaalbos die samengevoegd kan worden met de originele heap.

## 3.2 Pairing heaps

- Een algemene boom waarvan de sleutels voldoen aan de heapvoorwaarde.
- Elke knoop heeft een wijzer naar zijn linkerkind en rechterbroer (figuur 3.3 en 3.4). Als verminderen van prioriteit moet ondersteund worden, heeft elke knoop als linkerkind een wijzer naar zijn ouder en als rechterkind een wijzer naar zijn linkerkind.



Figuur 3.3: Een pairing heap.



Figuur 3.4: Dezelfde pairing heap, maar in boom-vorm.

De operaties op een pairing heap.

- **Samenvoegen:** Verlijk het wortelelement van beide heaps. De wortel met het grootste element wordt het linkerkind van deze met het kleinste element.
- **Toevoegen:** Maak een nieuwe pairingheap met één element, en voeg deze samen met de oorspronkelijke heap.
- **Prioriteit wijzigen:** De te wijzigen knoop wordt losgekoppeld, krijgt de prioriteitwijziging, en wordt dan weer samengevoegd met de oorspronkelijke heap.
- **Minimum verwijderen:** De wortel verwijderen levert een collectie van  $c$  heaps op. Voeg deze heaps van links naar rechts samen in  $O(n)$  of voeg eerst in paren toe, en dan van rechts naar links toevoegen in geamortiseerd  $O(\lg n)$ .
- **Willekeurige knoop verwijderen:** De te verwijderen knoop wordt losgekoppeld, zodat er twee deelheaps ontstaan. Deze twee deelheaps worden samengevoegd.