FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Gevorderde algoritmen

Bert De Saffel

Master in de Industriële Wetenschappen: Informatica Academiejaar 2018–2019

Gecompileerd op 25 september 2019

UNIVERSITEIT
GENT

Inhoudsopgave

Deel I

Gegevensstructuren II

Hoofdstuk 1

Efficiënte zoekbomen

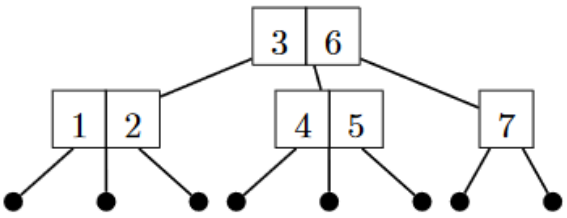
1.1 Inleiding

- 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:

$$\Delta h <= 1$$

     - Δh 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.
- Analoog aan een 2-3-boom, maar elke knoop heeft 2, 3 of 4 kinderen.

(d) Rood-zwarte bomen (sectie 1.2.1).

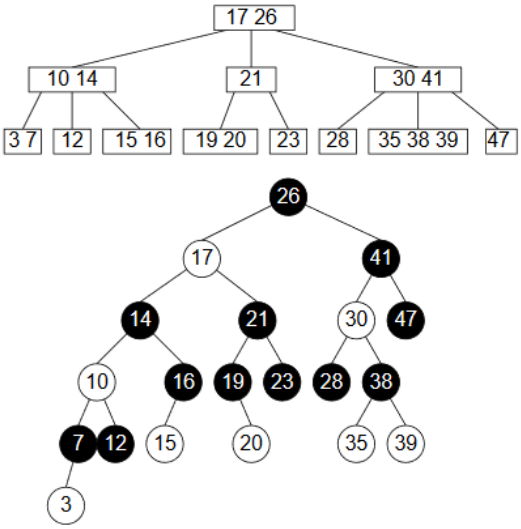
2. **Elke reeks operaties steeds efficiënt maken.**

(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 definieren 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 een virtuele knoop bevat evenveel zwarte knopen. Dit aantal wordt de *zwarte hoogte* genoemd
  - De wortel is zwart.
- **Eigenschappen:**
  - Een deelboom met wortel w en 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:

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

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 gegerandeerd $O(\lg n)$.
- 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.

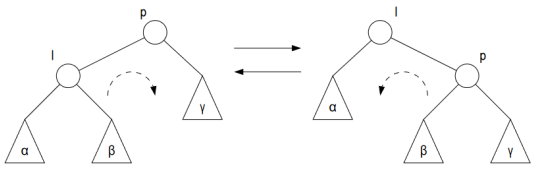
   ! Multithreading is niet mogelijk. Alle threads die een bottom-up rood-zwarte boom gebruiken moeten gelocked worden bij een toevoeg-of verwijderoperatie.
2. **Top-down:**
   - Pas de boom aan langs de dalende zoekweg.

! 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 inorder 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.

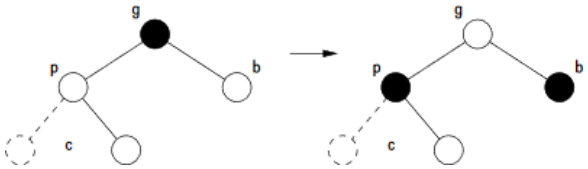
1.2.5 Bottom-up rood-zwarte bomen

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.**

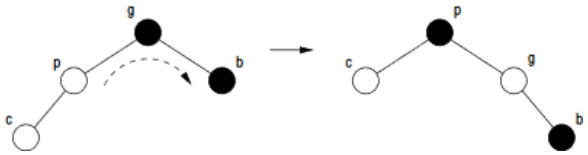


Figuur 1.4: Rode broer.

   - 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.**

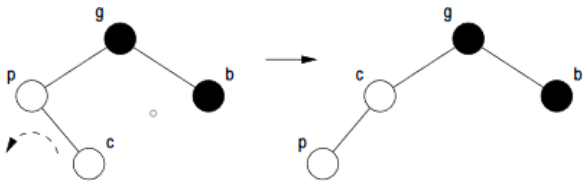
   (a) **Knoop c is een linkerkind van p.**



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 $O(\lg n)$ opschuivingen.
- Roteren en opschuiven is $O(1)$, en afdalen is $O(\lg n)$ zodat toevoegen $O(\lg n)$ is.

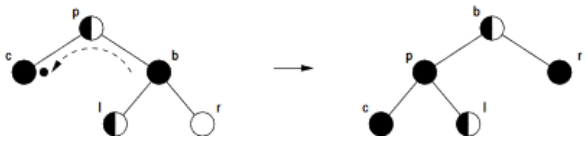
Verwijderen

- 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 drie mogelijkheden:
  1. **De knoop heeft 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, herstelt 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.

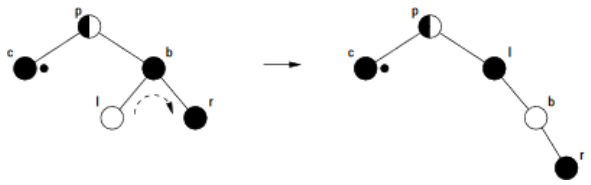
     (b) **Broer b heeft een rood rechterkind.** De kleur van het linkerkind l van b is willekeurig.



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 $O(\lg n)$.

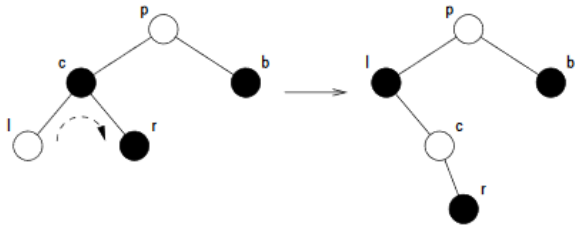
Verwijderen

- 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 bolgende 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.

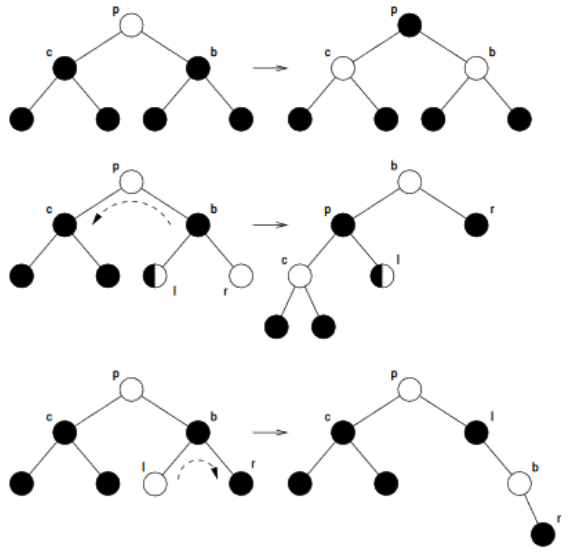
1. **Knoop c heeft minstens één rood kind.**



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

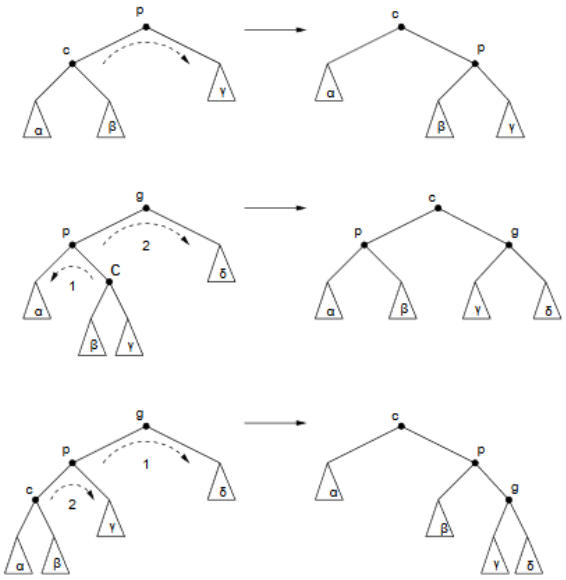
- Garanderen dat elke reeks opeenvolgende operaties efficiënt is.
- Als we m operaties verrichten op de splay tree, waarbij n keer toevoegen, dan is de performantie van deze reeks $O(m \lg n)$.
- Uitgemiddeld is dit $O(\lg n)$.
- Individuelde operaties mogen inefficiënt zijn, maar de boom moet zo aangepast worden zodat een reeks van die operaties efficiënt zijn.
- **Basisidee:** Elke knoop die gezocht wordt, toegevoegd of verwijdert wordt, zal de wortel worden van de boom, zodat opeenvolgende operaties op die knoop efficiënt zijn.
- Een willekeurige knoop tot wortel maken gebeurd 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 wordt 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 rotaties:

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 linker- 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.

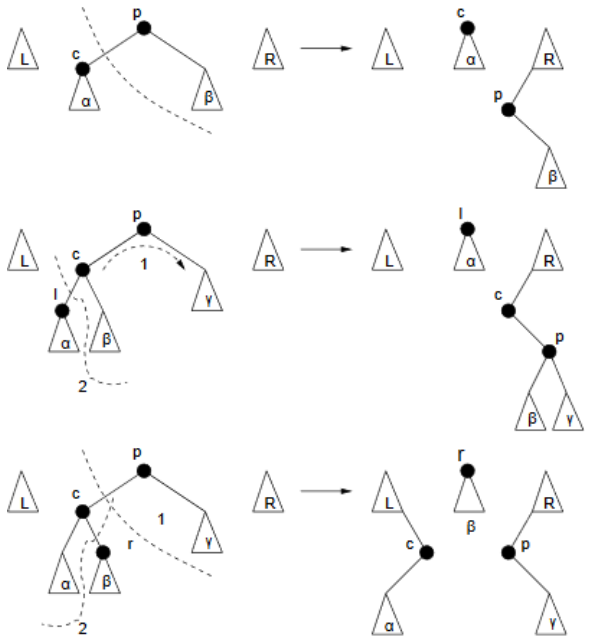
(b) **Knoop c is een linkerkind van p.**

- 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.

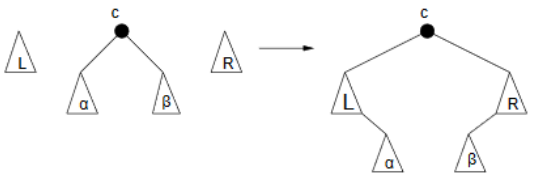
1.3.2 Top-down splayboom

- 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** Φ gebruikt.
- Elke mogelijke vorm van een splayboom krijgt een reeël 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 Φ_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 individuelde 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 Φ_m zeker niet kleiner is dan de beginpotentiaal Φ_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 dergerlijke bovengrenzen, maar kan niet vereenvoudigt 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 $p_1, p_2, ...p_k$ op de zoekweg gewijzigt. 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

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

      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 verwijdert worden.

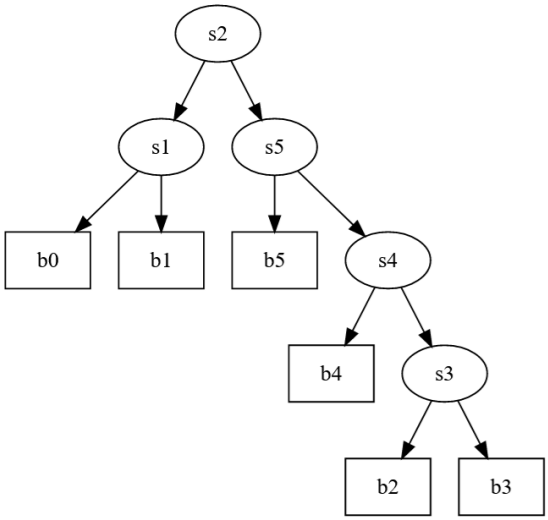
1.5 Skip lists

- 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 $s_1, \ldots, s_n$.
- De $n+1$ bladeren zijn $b_0, \ldots, b_n$.
  - 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 gorter 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

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

- 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 **deelproblemen 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 deelproblemen.
  - Zijn de deelproblemen onafhankelijk?
    - Ja want deelbomen hebben geen gemeenschappelijke knopen.
  - Zijn de deelproblemen overlappend?

    - Elke deelboom bevat een reeks opeenvolgende sleutels $s_i, \ldots s_j$ met bijhorende bladeren $b_{i-1}, \ldots, b_j$.
    - Deze deelboom heeft een wortel s_w waarbij $(i \leq w \leq j)$.
    - De linkse deelboom bevat de sleutels $s_i, \ldots, s_{w-1}$ en bladeren $b_{i-1}, \ldots, b_{w-1}$.
    - De rechtse deelboom bevat de sleutels $_{w+1}, \ldots, s_j$ en bladeren $b_w, \ldots, b_j$.
    - 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 $s_i, \ldots, s_j$ 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.

- We willen dus de kleinste verwachte zoektijd $z(i, j)$.
- Dit moet gebeuren voor alle i en j waarbij:
    - $1 \leq i \leq n + 1$
    - $0 \leq j \leq n$
    - $j \geq i - 1$
- 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 → 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 $s_i, \ldots, s_j$ 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 $z(i,i-1), z(i,i), \ldots, z(i,j-1)$ van de linkse deelboom nodig en de waarden $z(i+1,j), \ldots, z(j,j), z(j+1,j)$ van de rechtse deelbom 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 $s_i, \ldots s_j$ heeft $j - i + 1$ mogelijke wortels.
    - Elke test is $O(1)$.
    - Dit werk is evenredig met

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

  - **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:
  - $X = \langle x_0, x_1, \ldots, x_{n-1} \rangle$
  - $Y = \langle y_0, y_1, \ldots, y_{m-1} \rangle$
- Hoe LGD bepalen?
  - Is er een optimale deelstructuur?

        - Een optimale oplossing maakt gebruik van optimale oplossingen voor deelproblemen.
        - De deelproblemen zijn paren prefixen van de twee strings.
        - 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 $Z = \langle z_0, z_1, \ldots, z_{k-1} \rangle$ 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 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

Uitwendige gegevensstructuren

- Als de grootte van de gegevens de capaciteit van het intern geheugen overschrijdt, moeten deze gegevens opgeslagen worden in extern geheugen.
- We willen dat woordenboekoperaties nog steeds efficiënt uitgevoerd worden.
- Een harde schijf is veel trager dan een processor.
- Daarom moet het aantal schijfoperaties geminimaliseerd worden.

3.1 B-trees

- Uitwendige evenwichte zoekboom.
- Heeft een zeer kleine hoogte.
- Het aantal sleutels n is wel zeer groot.
- Er worden dus meerdere kinderen per knoop opgeslagen.
- Knopen kunnen best een volledige schijfpagina benutten.

3.1.1 Definitie

- Een B-tree heeft een orde m waarbij $m > 2$.
  - Elke inwendige knoop heeft hoogstens m kinderen.
  - Elke inwendige knoop, behalve de wortel, heeft minstens $\lceil m/2 \rceil$ kinderen.
  - Elke inwendige knoop met $k + 1$ kinderen bevat k sleutels.
  - Elk blad bevat hoogstens $m - 1$ en minstens $\lceil m/2 \rceil - 1$ sleutels.
  - De wortel bevat tenminste 2 kinderen, tenzij hij een blad is.
  - Als de wortel een blad is bevat hij minstens 1 sleutel.
  - Alle bladeren bevinden zich op hetzelfde niveau.
- Elke knoop bevat het volgende:
  - Een geheel getal k dan het huidig aantal sleutels in de knoop aanduidt.
  - Een tabel voor maximaal m pointers naar de kinderen van de knoop.

- Een tabel voor maximaal $m-1$ sleutels, die stijgend gerangschikt zijn.
  - Er is ook een tabel die bijbehorende informatie per sleutel bijhoudt.
  - De k geordende sleutels van de inwendige knoop verdelen het sleutelbereik in $k+1$ deelgebieden.
  - De sleutels uit de deelboom van het i-de kind c_i liggen tussen de sleutels s_{i-1} en s_i.
- Een logische waarde b die aanduidt of de knoop een blad is of niet.

- $2-3$ bomen ($m=3$) of $2-3-4$ bomen ($m=4$) zijn eenvoudige voorbeelden van B-trees. Normaal is m wel groter.

3.1.2 Eigenschappen

- Het minimaal aantal knopen voor een boom met hoogte h is

$$1+2+2g+\cdots+2g^{h-1} = 1+2\sum_{i=0}^{h-1} g^i = 1+2\left(\frac{1-g^h}{1-g}\right)$$

  - De wortel van een minimale boom heeft slechts 1 sleutel en twee kinderen.
  - Elk kind heeft minimum $g = \lceil m/2 \rceil$ kinderen.

- De hoogte is bijgevolg $O(\lg n)$.
  - Elke knoop heeft minstens $g-1$ sleutels, behalve de wortel, die er minstens één heeft.

$$\begin{aligned} n &\geq 1+2(g-1)\left(\frac{g^h-1}{g-1}\right) \\ \rightarrow n &\geq 2g^h-1 \\ \rightarrow h &\leq log_g\left(\frac{n+1}{2}\right) \end{aligned}$$

- Een B-tree met n uniform verdeelde sleutels gebruikt ongeveer $\frac{n}{m \ln 2}$ schijfpaginas.

3.1.3 Woordenboekoperaties

Zoeken

- In elke knoop moet een meerwegsbeslissing genomen worden.
- De knoop moet eerst in het geheugen ingelezen worden.
- De sleutel wordt opgezocht in de gerangschikte tabel met sleutels.
  - Normaal zou binair zoeken efficiënter zijn, maar deze winst is vrij onbelangrijk.
  - Lineair zoeken kan bij kleine tabellen efficiënter uitvallen door het aantal cachefouten te minimaliseren.
- Er kunnen zich nu drie situaties voordoen:
  1. Als de sleutel in de tabel zit stopt het zoeken, met als resultaat een verwijzing naar de knoop op de schijf.
  2. Als de sleutel niet gevonden is en is de knoop een blad, dan zit de sleutel niet in de boom.

3. Als de sleutel niet gevonden is en de knoop is inwendig, wordt een nieuwe knoop in het geheugen ingelezen waarvan de wortel een kind is van de huidige knoop. Het zoekproces start opnieuw met deze knoop.

- Performantie:
  - Het aantal schijfoperaties is $O(h) = O(log_g n)$.
  - De processortijd per knoop is $O(m)$.
  - De totale performantie is $O(m \log_g n)$.

Toevoegen

- Toevoegen gebeurt **bottom-up**. Een top-down implementatie is ook mogelijk maar wordt minder gebruikt.
- De structuur van de boom kan gewijzigd worden.
- Toevoegen gebeurt altijd aan een blad.
- Vanuit de wortel wordt eerst het blad gezocht waarin de sleutel zou moeten zitten.
- Drie gevallen:
  - **De B-tree is ledig.**
    - De wortelknoop moet in het geheugen aangemaakt worden en gedeeltelijk ingevuld worden.
    - De knoop wordt dan naar de schijf gekopieërd.
  - **De B-tree is niet ledig.**
    - Het blad waarin de sleutel moet zitten wordt opgezocht. Er zijn dan twee gevallen.
    - **Het blad bevat minder dan m sleutels.**
      - De sleutel wordt in de juiste volgorde toegevoegd aan de tabel met sleutels.
    - **Het blad bevat m sleutels.**
      - Het blad wordt opgeslitst bij de middelste seutel.
      - Er wordt een nieuwe knoop op hetzelfde niveau aangemaakt, waarin de gegevens rechts van de middelste sleutel terechtkomen.
      - De middelste sleutel gaat naar de ouder.
      - Normaal gezien heeft de ouder plaats voor deze knoop, anders wordt er opnieuw geplitst.
- Performantie:
  - In het slechtste geval worden er $h + 1$ knopen gesplitst.
  - Een knoop splitsen vereist drie schijfoperaties en een processortijd van $O(m)$.
  - In het slechtste geval moet de boom tweemaal doorlopen worden.
    - Eerst om de sleutel te vinden.
    - Daarna eventueel tot de wortel splitsen.
    - ✓ Maar het aantal schijfoperaties per niveau is constant.
  - Het totaal aantal schijfoperaties is $\Theta(h)$.
  - De totale performantie is dan $O(mh) = O(m \log_g n)$.

Verwijderen

- Ook hier wordt enkel de **bottom-up** versie besproken.
- De gezochte sleutel kan zowel in een blad als in een inwendige knoop zitten.
  - **De sleutel zit in een blad.**
    - Er zijn geen kinderen meer dus kan de sleutel verwijderd worden.
    - Het kan zijn dat het blad nu te weinig sleutels heeft (minder dan $\lceil m/2 \rceil - 1$).
    - Er wordt een sleutel geleend van de ouder.
    - In het slechtste geval gaat dit ontlenen door tot aan de wortel.
    - Een sleutel ontlenen van een wortel die slechts één sleutel bevat maakt hem ledig, zodat de wortel verwijdert wordt.
  - **De sleutel zit in een inwendige knoop.**
    - De sleutel wordt vervangen door zijn voorloper of opvolger.
    - Als een knoop nu te weinig sleutels overhoudt, gebeurt er een **rotatie**.
      - Een sleutel van zijn broer gaat naar zijn ouder.
      - Een sleutel van de ouder gaat naar de knoop, die ook een kindwijzer van de broer overneemt.
      - Dit kan enkel als er een broer is die sleutels kan missen.
      - Als geen enkele broer een sleutel kan missen, wordt de knoop samengevoegd met een broer.
- Performantie:
  - Analoog aan toevoegen en is dan $O(m \log_g n)$.

3.1.4 Varianten van B-trees

- Nadelen van een gewone B-tree:
  - De bladeren moeten plaats reserveren voor kindwijzers die toch niet gebruikt worden.
  - Inwendige knopen kunnen gegevens bevatten en dat maakt verwijderen veel ingewikkelder.
  - Zoeken naar een opvolger van een sleutel kan $O(\log_g n)$ schijfoperaties vereisen.

B^+-tree

- Alle gegevens en bijhorende informatie zitten in de bladeren.
- Inwendige knopen worden gebruikt als index om de gegevens snel te lokaliseren.
- Bladeren en inwendige knopen hebben dus een verschillende structuur.
- Er is ook een **sequence set**, een gelinkte lijst van alle bladeren in stijgende sleutelvolgorde.
- De inwendige knopen moeten enkel sleutels bevatten en geen bijhorende informatie zodat de maximale graad groter is dan de bladeren.
- De bladeren moeten geen plaats reserveren voor kindwijzers, zodat ze meer gegevens kunnen bevatten.

Prefix B^+-tree

- Een variant van een B^+-tree voor strings.
- Strings kunnen echter veel plaats innemen.
- Om twee deelbomen van elkaar te onderscheiden wordt de kleinst mogelijke prefix bijgehouden.

B^*-tree

- In plaats van enkel gegevens over te brengen naar een buur tijdens het splitsen, worden de gegevens verdeeld over **drie** knopen.
- Beter gevulde knopen betekent een minder hoge boom.

3.2 Uitwendige hashing

- Waneer de volgorde van de sleutels niet belangreijk is.
- De woordenboekoperaties vereisen gemiddeld slechts $O(1)$.
- Er wordt een binaire trie (hoofdstuk 10) gebruikt.
  - Wanneer een sleutel gezocht wordt, worden de sleutels niet vergeleken maar wel de opeenvolgende bits van de sleutel.
  - Elke deelboom van een trie bevat alle sleutels met een gemeenschappelijk prefix.
  - Alle sleutels van een deelboom kan in één pagina ondergebracht worden.
  - Als de pagina vol geraakt, wordt de knoop (en dus de pagina) gesplitst, en beide pagina's krijgen een nieuwe trieknoop als ouder.
  - De vorm van een trie is onafhankelijk van de toevoegvolgorde.
  - Daarom wordt niet de bits van de sleutels gebruikt, maar de hashwaarde.
- Dit hoofdstuk bespreekt twee methoden: **extendible hashing** en **linear hashing** die beiden het zoeken in de trie elimineren.

3.2.1 Extendible hashing

- Het zoeken in de trie wordt geëlimineerd door de langst mogelijke prefix uit de trie als index te gebruiken in een hashtabel.
- Kortere prefixen komen overeen met meerdere tabelelementen die allemaal een verwijzing naar dezelfde pagina moet bevatten.
- Implementatie:
  - Er is een hashtabel die wijzers naar schijfpagina's bevat, waarbij elke schijfpagina maximaal m sleutels met bijbehorende gegevens bevat.
  - De hashwaarden zijn gehele getallen, waarvan het bereik bepaald wordt door de breedte w van een processorwoord.
  - De laatste d bits van die getallen dienen als indices in de hashtabel, zodat de tabel 2^d elementen bevat.
  - De **globale diepte** is d en is de lengte van het langste prefix in de trie.

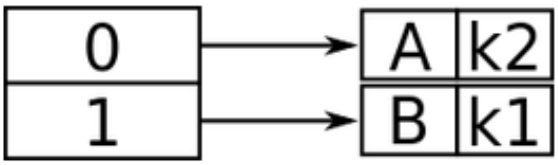
  - Alle sleutels waarvan de hashwaarde met dezelfde d bits eindigt komen bij hetzelfde tabelelement terecht.
  - Een pagina kan sleutels met hashwaarden bevatten waarvan de laatste d bits verschillend zijn.
  - Het aantal bits k is de **lokale diepte** en is het aantal waarmee al haar hashwaarden eindigen.

- De **woordenboekoperaties:**
  - **Zoeken.**
    - Bereken de hashwaarde van de sleutel.
    - Zoek de overeenkomstige pagina via de hashtabel.
    - Zoek sequentieel in deze pagina.
  - **Toevoegen.**
    - Als de pagina niet vol is moet gemiddeld helft van de elementen opgeschoven worden, maar dat is verwaarloosbaar.
    - Als de pagina vol is moet deze gesplitst worden.
    - Alle hashwaarden in die pagina beginnen met dezelfde k bits.
    - Er wordt daarom gesplitst op het volgende bit $k+1$. Alle elementen in de pagina waarbij die bit één is wordt overgebracht naar de nieuwe pagina.
    - De waarde van k wordt één groter zowel in de nieuwe pagina als in de oude pagina.
    - De hashtabel moet ook aangepast worden.
      - **Als k kleiner was dan d:** de helft van de wijzers van de oude pagina moeten naar de nieuwe pagina wijzen.
      - **Als k gelijk was aan d:** er was maar één wijzer naar de oude pagina. De waarde van d moet ook met één toenemen en de grootte van de hashtabel moet **verdubbelt** worden.
  - **Verwijderen.**
    - Zien we niet.

- Als er n uniform verdeelde sleutels opgeslagen zijn, dan is de verwachtingswaarde van het aantal pagina's $n/(m \ln 2) \equiv 1.44n/m$.

Voorbeeld

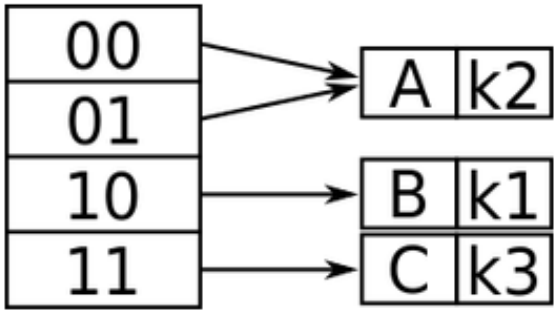
- Veronderstel een hashfunctie $h(k)$.
- Een pagina heeft slechts plaats om één hashwaarde op te slaan.
- De eerste i bits van elke hashwaarde wordt gebruikt als index in de hashtabel.
- De waarde van i is het kleinste getal zodanig dat elk item in de hashtabel uniek is.
- De volgende hashwaarden worden gebruikt:
  - $h(k_1) = 100100$
  - $h(k_2) = 010110$
  - $h(k_3) = 110110$
  - $h(k_4) = 011110$
- **k_1 en k_2 worden toegevoegd.**



Figuur 3.1

- Deze hashwaarden kunnen onderscheiden worden door het eerste bit, dus $d = 1$.
- De hashtabel bevat nu twee elementen (figuur 3.1), met twee wijzers: één naar pagina A en één naar pagina B.
- De lokale diepte k van beide elementen is gelijk aan d, $k = d = 1$.

- **k_3 wordt toegevoegd.**



Figuur 3.2

- Er zijn niet genoeg bits om k_3 te onderscheiden van k_1.
- Er wordt vergeleken op bit $k + 1$.
- Er wordt een nieuwe pagina C aangemaakt waarin de sleutel k_3 opgeslagen wordt.
- Aangezien k gelijk was aan d, verdubbelt de grootte van de hashtabel.

3.2.2 Linear hashing

- Er wordt geen hashtabel gebruikt door ervoor te zorgen dat pagina's opeenvolgende adressen hebben.
- De d eindbits van de hashwaarde worden niet gebruikt als index, maar rechtstreeks als adres van een pagina.
- Het gaat hier over **logische adressen**, die eenvoudig manipuleerbaar zijn en niet de **fysische adressen** die het besturingssysteem beheert.
- Er zijn 2^d adressen en evenveel pagina's.
- Als een pagina vol is wordt deze gesplitst, maar niet noodzakelijk de hele pagina.
- Pagina's worden in sequentiële volgorde gesplitst, of ze nu vol zijn of niet.
- Elke pagina die niet vol is (alle pagina's behalve de volle die het splitsen veroorzaakt heeft) krijgt een overflow pagina.

- Als de pagina aan de beurt is om te splitsen, worden zijn gegevens verdeeld over zijn overflow pagina en de pagina zelf.
- De **woordenboekoperaties:**
  - **Zoeken.**
    - Bereken de hashwaarde van de sleutel.
    - We moeten echter weten hoeveel eindbits er nodig zijn om de pagina te adresseren.
    - Er wordt een variabele p bijgehouden, die het adres van de volgende te splitsen pagina bijhoudt.
    - Het adres gevormd door de d eindbits wordt vergeleken met p.
    - Als $d < p$ dan is de gezochte pagina reeds gesplitst en moeten $d+1$ eindbits gebruikt worden. Anders volstaan d bits.
    - De sleutel kan in de pagina binair of lineair gezocht worden.
  - **Toevoegen.**
    - Eerst wordt de juiste pagina gelokaliseerd.
    - Als de pagina vol zit moet ze gesplitst worden.
    - Splitsen gebeurt sequentieël zodat $p = 0$ in het begin.
    - p wordt met één verhoogd tot alle 2^d pagina's gesplitst zijn.
    - De waarde van d wordt dan verhoogd met één, en p wordt terug 0.
    - Als pagina p gesplitst wordt, is het adres van de nieuwe pagina $p + 2^d$.
  - **Verwijderen.**
    -

Hoofdstuk 4

Meerdimensionale gegevensstructuren

- Gegevens met meer dan één sleutel zijn meerdimensionaal.
- Gegevensstructuren moeten toelaten om op al die sleutels, of in een bereik van meerdere sleutels te zoeken.
- De meeste gegevensstructuren zijn efficiënt voor een klein aantal dimensies.
- De gegevens worden zo gemodelleerd zodat ze een geometrische structuur vormen.
- Elke sleutel is een punt in een meerdimensionale Euclidische ruimte.
- Een meerdimensionaal punt zoeken is een speciaal geval van zoeken van alle punten in een meerdimensionale hyperrechthoek.
- Notatie:
  - Het aantal punten is n.
  - Het aantal dimensies is k.

4.1 Projectie

- Per dimensie wordt er een gegevensstructuur (bv gelinkte lijst) bijgehouden die de gesorteerde punten volgens die dimensie bijhoudt.
- Elk punt wordt dus geprojecteerd op elke dimensie.
- Zoeken in een hyperrechthoek gebeurt door een dimensie te kiezen en alle punten te zoeken die voor die dimensie binnen de hyperrechthoek liggen.
- Deze methode werkt als de zoekrechthoek een zijde heeft die de meeste punten uitsluit.
- De **gemiddelde performantie** is $O(n^{1-\frac{1}{k}})$.

4.2 Rasterstructuur

- De zoekruimte wordt verdeelt met behulp van een raster.
- Voor elk rastergebied (een hyperrechthoek) wordt een gelinkte lijst bijgehouden met de punten die erin liggen.
- De punten vinden die in een hyperrechthoek liggen komt neer op het vinden van de rastergebieden die overlappen, en welke van de punten in hun gelinkte lijsten binnen die rechthoek vallen.
- Het aantal rastergebieden is best een constante fractie van n, zodat het gemiddeld aantal punten in elk rastergebied een kleine constante wordt.

4.3 Quadtrees

- Een quadtree verdeelt de zoekruimte in 2^k hyperrechthoeken, waarvan de zijden evenwijdig zijn met het assenstelsel.
- Deze verdeling wordt opgeslaan in een 2^k-wegsboom: elke knoop staat voor een gebied.
- Een quadtree is niet geschikt voor hogere dimensies: er zouden te veel knopen zijn.
- Deze cursus behandelt enkel twee dimensies en er worden enkel **twee-dimensionale punten** opgeslaan.

4.3.1 Point quadtree

- Elke inwendige knoop bevat een punt, waarvan de coördinaten de zoekruimte opdelen in vier rechthoeken.
  - Elk (deel)zoekruimte is de wortel van een deelboom die alle punten in de overeenkomstige rechthoek bevat.
- Woordenboekoperaties:
  - **Zoeken en toevoegen.**
    - Het zoekpunt wordt telkens vergeleken met de punten van de opeenvolgende knopen.
    - Als het zoekpunt niet aanwezig is, eindigt de zoekoperatie in een ledig deelgebied, maar kan het punt wel toegevoegd worden als inwendige knoop.
    - De structuur van een point quadtree is afhankelijk van de toevoegvolgorde, maar is in het gemiddelde geval $O(\lg n)$. In het slechtste geval is het $O(n)$.
  - **Toevoegen als de gegevens op voorhand gekend zijn.**
    - Er kan voor gezorgd worden dat geen enkel deelgebied meer dan de helft van de punten van die van zijn ouder bevat.
    - De punten worden lexicografisch geranschikt en de wortel is de mediaan.
    - Alle punten voor de mediaan vallen dan in twee van zijn deelbomen, deze erachter in de andere twee.
    - Bij elk kind gebeurt hetzelfde.
    - Deze constructie is $O(n \lg n)$.
  - **Verwijderen.**
    - Een punt verwijderen zorgt ervoor dat een deelboom geen ouder meer heeft.
    - Om dit op te lossen worden alle punten in die deelboom opnieuw toegevoegd aan de boom.

4.3.2 PR quadtree

- Point-region quadtree.
- De zoekruimte **moet een rechthoek zijn**.
  - De zoekruimte kan gegeven worden.
  - De zoekruimte kan ook bepaald worden als de kleinste rechthoek die alle punten omvat.
- Elke knoop verdeelt de zoekruimte in vier **gelijke rechthoeken**.
- De opdeling loopt door tot dat elk deelgebied nog één punt bevat.
- Inwendige knopen bevatten geen punten.
- Woordenboekoperaties:
  - **Zoeken.**
    - De opeenvolgende punten vanuit de wortel worden gebruikt om de rechthoek te vinden waarin het punt zou moeten liggen.
  - **Toevoegen.**
    - Als de gevonden rechthoek geen punt bevat kan het punt toegevoegd worden.
    - Als de gevonden rechthoek wel een punt bevat, moet deze rechthoek opnieuw opgesplitst worden tot elk van de punten in een eigen gebied ligt.
  - **Verwijderen.**
    - Een punt verwijderen kan ervoor zorgen dat een deelgebied ledig wordt.
    - Als er nog slechts 1 punt zit in één van de vier deelgebieden, kunnen deze deelgebieden samengevoegd worden.
- De vorm van een PR quadtree is wel onafhankelijk van de toevoegvolgorde.
- Er is geen verband tussen de hoogte h en het aantal opgeslagen punten n omdat een PR quadtree nog steeds onevenwichtig kan uitvallen.
- Er is wel een verband tussen de hoogte h en de kleinste afstand a tussen twee zoekpunten.
  - Stel z de grootste zijde van de zoekruimte.
  - De grootste zijde van een gebied op diepte d is dan $\frac{z}{2^d}$.
  - De maximale afstand tussen twee punten in dat gebied is de lengte van de diagonaal in dat gebied

$$\begin{aligned}\sqrt{\left(\frac{z}{2^d}\right)^2 + \underset{k-2}{\cdots} + \left(\frac{z}{2^d}\right)^2} &= \sqrt{k\left(\frac{z}{2^d}\right)^2} \\ &= \frac{z\sqrt{k}}{2^d}\end{aligned}$$

  - Op elke diepte d is

$$\begin{aligned}a &\leq \frac{z\sqrt{k}}{2^d} \\ 2^d &\leq \frac{z\sqrt{k}}{a} \\ d &\leq \lg\left(\frac{z\sqrt{k}}{a}\right) \\ d &\leq \lg\left(\frac{z}{a}\right) + \lg(\sqrt{k}) \\ d &\leq \lg\left(\frac{z}{a}\right) + \frac{\lg k}{2}\end{aligned}$$

- De hoogte h is de maximale diepte van een inwendige knoop plus één:

$$h \leq d + 1 \leq \lg\left(\frac{z}{a}\right) + \frac{\lg k}{2} + 1$$

- Performantie:
  - Op elk niveau bedekken de gebieden van de inwendige knopen de verzameling punten, en al deze gebieden bevatten punten.
  - Per niveau is het aantal inwendige knopen $O(n)$.
  - Het totaal aantal inwendige knopen i in een boom met hoogte h is $O(hn)$.
  - Elke inwendige knoop heeft 4 kinderen, zodat het aantal bladeren $3i + 1$ is.
  - Het aantal knopen is ook $O(hn)$.
  - De constructietijd van de boom is $O(hn)$.

4.4 K-d trees

- Vermijdt de grote vertakkingsgraad van een point quadtree door een binaire boom te gebruiken.
- Elke inwendige knoop bevat een punt, dat de deelzoekruimte slechts opsplitst in één dimensie.
- Opeenvolgende knopen gebruiken opeenvolgende dimensies om te splitsen.
- De opdeling kan doorgang tot slechts één punt in elk gebied is, of men kan vroeger stoppen en gelinkte lijsten bijhouden per gebied.
- Door de (eventueel random) afwisselende dimensies zijn er geen rotaties mogelijk om een dergelijke boom evenwichtig te maken. Daarom wordt verwijderen ook nooit echt gedaan, maar eerder met **lazy deletion**.
- Men kan wel af en toe een deelboom reconstrueren, en dan ook de te verwijderen knopen effectief verwijderen.

Hoofdstuk 5

Samenvoegbare heaps

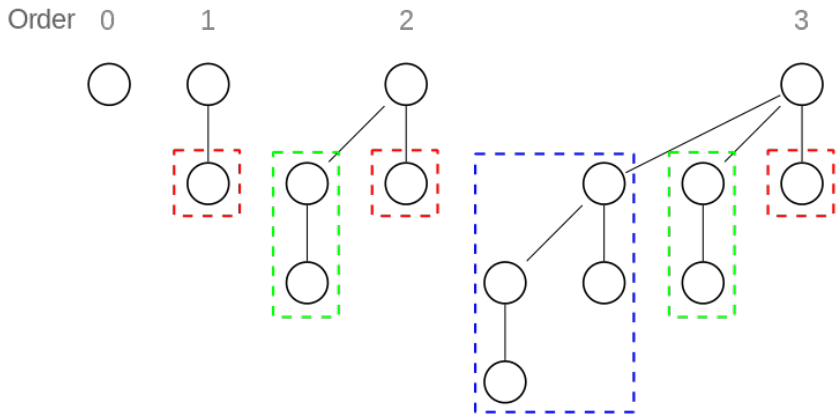
- Een samenvoegbare heap is een heap die geoptimaliseerd zijn om de 'join' operatie uit te voeren.
- De join operatie voegt twee heaps samen, zodat de **heapvoorwaarde** nog steeds geldig is.
- Een lijst van bekende heaps.
  - Leftist heaps.
    - Deze heaps proberen zo onevenwichtig mogelijk te zijn.
    - De linkerkant is diep en de rechterkant ondiep.
    - De operaties zijn efficiënt omdat al het werk in de rechterkant gebeurt.
  - Skew heaps.
    - Gelijkaardig aan een leftist heap, maar er is een vormbeperking.
    - In het slechtste geval kunnen individuele operaties $O(n)$ zijn.
  - Fibonacci heaps.
  - Relaxed heaps

De belangrijke samenvoegbare heaps zijn: **Binomial queues** en **Pairing heaps**.

5.1 Binomiale queues

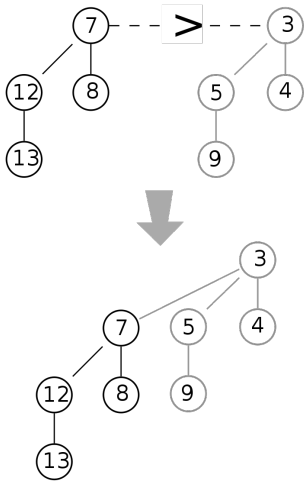
- Bestaat uit bos van binomiaalbomen.
- Binomiaalboom B_n bestaat uit twee binomiaalbomen B_{n-1}. B_0 bestaat uit één knoop.
- De tweede binomiaalboom is de meest linkse deelboom van de wortel van de eerste.
- Een binomiaalboom B_n bestaat uit een wortel met als kinderen $B_{n-1}, ..., B_1, B_0$ (zie figuur 5.1)
- Op diepte d zijn er $\binom{n}{d}$ knopen.
- Voorbeeld: Een prioriteitswachtrij met 13 elementen wordt voorgesteld als $\langle B_3, B_2, B_0 \rangle$.

De operaties op een binomiaalqueue:



Figuur 5.1: Verschillende ordes van binomiaalbomen.

- **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.

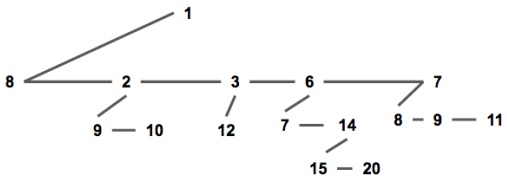


Figuur 5.2: Hier worden twee binomiaalbomen van orde 3 samengevoegd. De boom met de waarde 7 voor de wortel wordt het linkerkind van de boom met waarde 3 voor de wortel. Het wordt een boom van orde 4.

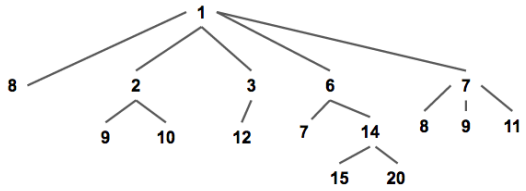
- **Toevoegen**: Maak een triviale binomiaalqueue met één knoop en voeg deze samen met de andere binomiaalqueue.
- **Minimum verwijderen**: Zoek binomiaalboom B_k met het kleinste wortelelement. Verwijder deze uit de binomiaalqueue. De deelbomen van deze binomiaalboom vormen een nieuw binomiaalbos die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

5.2 Pairing heaps

- Een algemene boom waarvan de sleutels voldoen aan de heapvoorwaarde.
- Elke knoop heeft een wijzer naar zijn linkerkind en rechterbroer (figuur 5.3 en 5.4). 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 5.3: Een pairing heap.



Figuur 5.4: Dezelfde pairing heap, maar in boomvorm.

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 onstaan. Deze twee deelheaps worden samengevoegd.