FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Gevorderde algoritmen

Bert De Saffel

Master in de Industriële Wetenschappen: Informatica Academiejaar 2018–2019

Gecompileerd op 8 november 2018

UNIVERSITEIT
GENT

Inhoudsopgave

Hoofdstuk 1

Efficiënte zoekbomen

Vooraleer efficiënte zoekbomen behandeld kunnen worden moet er eerst een fundamentele operatie besproken worden: **rotaties**. Rotaties wijzigen de vorm van de boom, maar behouden de inorder volgorde van de sleutels. Dit is nodig omdat de eigenschap van een binaire zoekboom steeds voldaan moet zijn, namelijk dat het linkerkind kleiner is, en het rechterkind groter, dan de ouder. Een rotatie is $O(1)$ omdat enkel pointers verplaatst moet worden. ToDo: foto van rotatie

1.1 Rood-zwarte bomen

Een rood-zwarte boom is een **binaire zoekboom** waarbij bovendien:

- Elke knoop rood of zwart gekleurd is.
- Elke virtuele knoop zwart is. Een virtuele knoop is een ontbrekend kind (nullpointer), die geen gegevens bevatten maar wel een kleur hebben (zwart).
- De wortel zwart is (een rode wortel kan zonder problemen zwart gemaakt worden).
- Elke mogelijke weg vanuit een knoop naar een virtuele knoop evenveel zwarte knopen heeft **(zwarte diepte)**.
- De hoogte h, kan uit de voorgaande definities, afgeleidt worden aangezien elke deelboom met wortel w en zwarte diepte z tenminste $2^z - 1$ inwendige knopen bevat.

$$1 + 2 + ... + 2^{z-1} = 2^z - 1$$

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

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

1.1.1 Operaties

Zoeken is equivalent met een gewone binaire boom en is dus $O(\lg n)$. De interessante operaties zijn toevoegen en verwijderen, die beiden zowel bottom-up als top-down kunnen gebeuren:

- **Bottom-up.** Een bottom-up rood-zwarte boom zal eerst een knoop toevoegen of verwijderen, en nadien de boom herstellen.
  - **Toevoegen.** Een knoop toevoegen gebeurt op dezelfde manier als bij een normale binaire zoekboom. Een nieuwe toegevoegde knoop krijgt altijd een rode kleur, omdat de zwarte diepte herstellen moeilijker is. Bij het toevoegen van een knoop kunnen er zich zes gevallen voordoen, waarvan er drie het spiegelbeeld zijn van elkaar. Hier wordt veronderstelt dat de ouder p van de toegevoegde knoop c het linkerkind is van grootouder g, en dus de broer b van p het rechterkind is van g.
    1. Indien b rood is, kan eenvoudig p en b zwart gemaakt worden, terwijl g rood gemaakt wordt. Indien g een zwarte ouder heeft is de situatie opgelost. Is dit niet het geval, wordt het probleem opgeschoven naar boven, alsof het lijkt dat g de toegevoegde knoop is want die is nu rood. De ligging van c ten opzichte van p heeft in dit geval geen impact.
    2. Indien b zwart is, kunnen er zich twee gevallen voordoen:
       (a) Indien c aan de uitwendige kant ligt van p, liggen de drie knopen g, p en c op een lijn en moet er een rotatie naar rechts uitgevoerd worden. Deze rotatie wordt gevolgd door p zwart en g rood te kleuren.
       (b) Indien c aan de inwendige kant ligt van p, dan moet enkel p en c naar links geroteerd worden, zodat we het vorige geval krijgen (c is nu wel de ouder van p).
  - **Verwijderen.** Ook wordt deze operatie eerst uitgevoerd zoals bij een normale binaire zoekboom. Indien de ffysisch te verwijderen knoop rood is, is verwijderen eenvoudig aangezien de zwarte hoogte ongewijzigd blijft.
- **Top-down.** Een top-down rood-zwarte boom zal op de zoekweg ook al de boom herstellen.
  - **Toevoegen.** Op de weg naar beneden mogen we geen rode broers toelaten, aangezien de nieuwe knoop kind van beide kan zijn. Wanneer er op de zoekweg een zwarte knoop c met twee rode kinderen voorkomt, kan c rood gemaakt worden en zijn kinderen zwart, indien de ouder p van c ook rood is, en c ligt aan de butienkant, moet p geroteerd worden, zodat p de ouder is van c en g, de oorspronkelijke ouder van p. Ligt c aan de binnenkant, dan wordt eerst c geroteerd, zodat deze de ouder wordt van p, gevolgd door een bijkomende rotatie rond c zodat c als kinderen p en g heeft. De knoop c wordt terug zwart gemaakt en g wordt rood gemaakt.
  - **Verwijderen.**

1.2 Splaybomen

Een splayboom is een normale binaire zoekboom, waarbij er een splayoperatie gedefinieerd is: een reeks van operaties zodat de meest recente opgevraagde knoop α in de wortel staat. Bij splaybomen heeft zoeken ook een bottom-up en top-down versie.

1.2.1 Operaties

- **Bottom-up.** De splayoperatie bij een bottom-up splayboom maakt gebruik van drie rotaties: **zig**, dat enkel uitgevoerd wordt indien de ouder p van α de wortel is van de boom. Deze rotatie is een normale rotatie zodat α de wortel wordt, en p één van de kinderen van α, **zig-zig**, dat uitgevoerd wordt indien p nog een ouder g heeft en α aan de inwendige kant light. De eerste rotatie roteert α naar de buitenkant, zodat deze de ouder wordt van p, gevolgd door een rotatie die α de ouder maakt van p en g en **zig-zag**, dat uitgevoerd wordt indien α, p en g op dezelfde lijn liggen (dus α is uitwendig). De eerst rotatie roteer g, zodat p de ouder is van α en g, gevolgd door een rotatie die c de ouder maakt van p (die nog steeds g als kind heeft).

  - **Zoeken.** Bottom-up zoeken is equivalent met een normale binaire zoekboom, gevolgd door de splayoperatie die de gezochte knoop tot wortel maakt. Als de gezochte sleutel niet bestaat, wordt de splayoperatie uitgevoerd op zijn voorloper of opvolger.
  - **Toevoegen.** Toevoegen zal eerst de fysische knoop toevoegen aan de boom. Deze toegevoegde knoop wordt dan met de splayoperatie tot wortel gemaakt.
  - **Verwijderen.** Eerst wordt de fysisch te verwijderen knoop verwijdert. De ouder van deze knoop wordt nu via de splayoperatie tot wortel gemaakt. De ouder is ook de laatste knoop op de zoekweg, zodat indien de te verwijderen knoop niet bestaat, nog steeds deze ouder tot wortel gemaakt wordt.

- **Top-down.** Een top-down splayboom maakt geen gebruik van rotaties, zodat er geen nood is aan ouderwijzers of stapels. De splayoperatie bij een top-down splayboom deelt de boom op in drie zoekbomen: L, die alle sleutels kleiner dan de sleutels in M bevat, en R, die alle sleutels groter dan de sleutels in M bevat. Initieël is M de oorspronkelijke boom en zijn R en L ledig. De zoekweg begint bij de wortel van M, en er wordt voor gezorgt dat de huidige knoop op de zoekweg steeds de wortel van M is ,zodat op het einde van het zoeken, de gezochte sleutel (of zijn voorloper of opvolger) de wortel is van de uiteindelijke splayboom.

  Top-down splaybomen kennen 6 operaties, waarvan er ook weer drie het spiegelbeeld zijn van elkaar. Veronderstel dat we vanuit een knoop p (die op dat moment de wortel van M is) naar het linkerkind c moeten, dan kunnen volgende gevallen zich voordoen:

  1. De knoop c is de laatste knoop op de zoekweg. Dit komt enkel voor indien c gezocht wordt, of als hij geen kind heeft in de richting dat gezocht moet worden. Knoop p, samen met zijn rechtste deelboom wordt het nieuwe kleinste element in R. De linkse deelboom van p wordt de niewe M met c als wortel. Dit geval wordt ook weer **zig** genoemd, maar heeft dus wel een heel andere implementatie dan de zig bij bottom-up splaybomen.
  2. Linkerkind c is niet de laatse knoop op de zoekweg.
     (a) Indien afgedaald moet worden naar het linkerkind l van c, dan wordt eerst p en c naar rechts geroteerd, daarna wordt c, samen met zijn rechtse deelboom, het nieuwe kleinste element in R. De linkse deelboom van c wordt de nieuwe M. Dit geval heet ook opnieuw **zig-zig**, aangezien de knopen op één lijn liggen.
     (b) Indien afgedaald moet worden naar het rechterkind r van c, dan wordt p, samen met zijn rechtste deelboom, het nieuwste kleinste element van R. Daarna wordt c het nieuwe grootste element in L, en de rechtste deelboom van c wordt de nieuwe M. Dit geval heet **zig-zag**.

Na deze operaties moeten de deelbomen nog samengevoegd worden, waarbij de wortel c is. Alle sleutels in de linkerdeelboom van c zijn groter dan die van L, dus kan L deze linkerdeelboom opnemen. Analoog zijn alle sleutels in de rechterdeelboom van c kleiner dan die van R, dus kan R deze rechterdeelboom opnemen. Het linkerkind van c wordt nu de wortel van L en het rechterkind wordt de wortel van R. De operaties verlopen nu als volgt:

- **Zoeken.** Deze operatie maakt de gezochte sleutel tot wortel, of indien deze niet gevonden wordt, door zijn voorloper of opvolger.
- **Toevoegen.** De voorloper of opvolger van de nieuwe sleutel wordt wortel, en de nieuwe knoop krijgt als linkerkind de voorloper met zijn linkse deelboom en als rechterkind zijn rechtste deelboom. Of alternatief, de nieuwe knoop krijgt als rechterkind de opvolger met zijn rechtste deelbom en als linkerkind zijn linkse deelboom.
- **Verwijderen.** Eerst wordt de sleutel gezocht, zodat die wortel wordt. Daarna wordt deze wortel verwijderd en worden de twee deelbomen terug samengevoegd.

1.2.2 Performantie

_ToDo: dit is niet belangrijk voor de test.

1.3 Randomized Search Trees

kans is klein dat hij dit vraagt

Hoofdstuk 2

Toepassingen van dynamisch programmeren

2.1 Langste gemeenschappelijke deelsequentie

Voor twee strings $X = \langle x_0, x_1, ...x_{n-1}\rangle$ en $Y = \langle y_0, y_1, ...y_{m-1}\rangle$ waarbij x_i en y_i individuele karakters zijn, kan men de langste gemeenschappelijke deelsequentie bepalen door een stringelementen weg te laten, zodat beide strings gelijk zijn. Dit probleem heeft een optimale deelstructuur: de deelproblemen zijn paren prefixen van de twee strings. Stel X_i de prefix met lengte i en X_0 de ledige prefix. Analoog geldt hetzelfde voor Y. Beschouw nu $Z = \langle z_0, z_1, ..., z_{k-1}\rangle$, dan zijn er drie mogelijkheden:

1. Als $n = 0$ of $m = 0$ dan is $k = 0$.
2. Als $x_{n-1} = y_{m-1}$ dan is $z_{k-1} = x_{n-1} = y_{m-1}$, met gevolg dat Z_{k-1} een LGD is van X_{n-1} en Y_{n-1}.
3. Als $x_{n-1} \neq y_{m-1}$:
   (a) Z is ofwel een LGD van X_{n-1} en Y ,met $z_{k-1} \neq x_{n-1}$, of,
   (b) Z is een LGD van X en Y_{m-1}, met $z_{k-1} \neq y_{m-1}$.

Dit kan opgesteld worden als een recursieve vergelijking, waarbij $c[i, j]$ de lengte van de LGD voorstelt:

$$c[i,j] = \begin{cases} 0 & \text{als i} = 0 \text{ of j} = 0 \\ c[i-1][j-1] & \text{als } i > 0 \text{ en } j > 0 \text{ en } x_i = y_j \\ max(c[i, j-1], c[i-1, j]) & \text{als } i > 0 \text{ en } j > 0 \text{ en } x_i \neq y_j \end{cases}$$

Uitgewerkt op de woorden **LOUIS** en **ALOYSIUS**:

		A	L	O	Y	S	I	U	S
	0	0	0	0	0	0	0	0	0
L	0	0	1	1	1	1	1	1	1
O	0	0	1	2	2	2	2	2	2
U	0	0	1	2	2	2	2	3	3
I	0	0	1	2	2	2	3	3	3
S	0	0	1	2	2	3	3	3	4

Hoofdstuk 3

Uitwendige gegevensstructuren

3.1 B-trees

Een B-tree is een uitwendige evenwichtige meerwegszoekboom met een zeer kleine hoogte waarbij elk blad op dezelfde diepte zit. Meestal wordt er geheugenruimte voorzien voor de wortel en de meest recent gebruikte knopen. Vooraleer knopen kunnen bewerkt worden moeten deze eerst ingeladen worden aan de hand van een welbepaalde paginaindex, en door de grootte van een knoop, bevat het geheugen best zo weinig mogelijk knopen. Overbodige knopen worden terug weggeschreven, indien deze gewijzigd werden, en uit het geheugen verwijderd.

3.1.1 Definitie

Een B-tree van orde m, waarbij $m > 2$, wordt als volgt gedefinieerd:

- Elke inwendige knoop heeft hoogsten m kinderen.
- Elke inwendige knoop, behalve de wortel, heeft minstens $\lceil m/2 \rceil$ kinderen, tenzij hij een blad is.
- Elk blad zit op hetzelfde niveau

Elke knoop bevat dan:

- Een getal $k = m - 1$ dat aanduidt hoeveel sleutels in de knoop zitten.
- Een tabel met maximaal k sleutels, die niet dalend gerangschikt zijn. Een tweede tabel van dezelfde grootte voor de informatie bij elke sleutel bij te houden.
- Een tabel voor maximaal m wijzers naar de kinderen van de knoop.
- Een logische waarde b die aanduidt of de knoop een blad is.

3.1.2 Eigenschappen

Het minimaal aantal sleutels n kan eenvoudig berekend worden. veronderstel een boom met hoogte h en $g = \lceil m/2 \rceil$. De wortel van de minimale boom heeft slechts één sleutel en twee kinderen. Elk van die kinderen heeft minimaal g kinderen, die op hun beurt ook minimaal g kinderen hebben, enz. Het **aantal knopen** wordt dus:

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

Aangezien elke knoop minstens $g - 1$ sleutels heeft, behalve de wortel, kan n geschreven worden als (zie 2.3.3 Afschattingen met sommen Algoritmen I cursus):

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

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

$$h \leq log_g \frac{n+1}{2}$$

De hoogte is dus $O(\lg n)$.

3.1.3 Operaties

Zoeken

Elke knoop op de zoekweg moet ingelezen worden. Allereerst wordt er nagegaan of de sleutel in deze knoop zit (via lineair of binair zoeken). Als de gezochte sleutel niet in de knoop zit, moet de volgende knoop ingeladen worden. De wijzer van de volgende knoop staat op dezelfde index in de kindtabel als waar de niet gevonden sleutel zou moeten zitten in de sleuteltabel. Is de huidige knoop een blad en zit de gezochte sleutel niet in dit blad, dan zit de sleutel niet in de boom.

Toevoegen

Enkel de **bottom-up** methode wordt besproken aangezien de top-down versie minder vaak gebruikt wordt, maar wel handig is indien meerdere gebruikers aan de boom moeten, omdat knopen op de zoekweg dan vroeger worden vrijgegeven. Toevoegen gebeurt altijd in een blad en vormt geen probleem zolang dit blad nog plaats heeft. Is dit blad vol, wordt de knoop gesplitst rond de middelste sleutel van knoop. Sleutels die zich rechts van deze middelste sleutel bevinden, worden in een nieuwe knoop ondergebracht, dat ook een blad wordt. Sleutels die zich links van de middenste sleutel bevinden, blijven in het blad. De middelste sleutel zelf wordt nu verwijderd van de knoop, en toegevoegd bij de ouderknoop van de gesplitste knoop, zodat hetzelfde proces zich kan herhalen. Elke splitsing kost drie schijfoperaties. In het slechtste geval wordt het probleem helemaal tot de wortel opgeshoven, zodat er een nieuwe knoop wordt aangemaakt, met slechts één element, die nu de wortel wordt van de B-tree.

Verwijderen

Ook bij verwijderen wordt enkel de **bottom-up** methode besproken. Een sleutel wordt enkel maar verwijderd indien deze in een blad zit. Deze strategie moet dus de te verwijderen sleutel vervangen met zijn voorloper of opvolger omdat die altijd in een blad zitten, maar de meeste sleutels zitten in bladeren, zodat dit meestal geen probleem vormt. Wanneer een knoop te weinig sleutels heeft $< \lceil m/2 \rceil$ dan kan men proberen sleutels over te nemen van één van de twee broerknopen. De sleutel van de broer gaat naar zijn ouder, een sleutel van de ouder gaat naar de knoop, die ook een kindwijzer van de broer overneemt. Omdat hier drie knopen worden aangepast, is het beter om meerdere sleutels op die manier te roteren, zodat elke knoop evenveel sleutels heeft. Indien geen van beide broers een sleutel kan afstaan, wordt de knoop samengevoegd met een broer, zodat de ouder een kind verliest. De sleutel die ervoor zorgde dat deze knoop bereikbaar was, wordt toegevoegd aan de samengevoegde knoop en verwijderd uit de ouderknoop.

3.1.4 Varianten

- **B$^+$-tree.** Deze variant zal enkel sleutels opslaan in de bladeren zodat inwendige knopen enkel gebruikt worden als index om deze sleutels te lokaliseren. Bovendien is er een gelinkte lijst van alle bladeren in stijgende sleutelvolgorde. Omdat inwendige knopen enkel dienen als index, moeten ze minder informatie bevatten. Bladeren moeten ook geen plaats reserveren voor kindwijzers, zodat ze meer gegevens kunnen bevatten.
- **Prefix B$^+$-tree.** Deze variant wordt gebruik indien de sleutels strings zijn. De inwendige knopen bevatten een zo kort mogelijke string, meestal een prefix van de te onderscheiden strings.
- **B*-tree.** Deze variant zal bij de splitsoperatie de gegevens over drie knopen verdelen, in plaats van twee knopen. Beter gevulde knopen betekent een minder hoge boom.

3.2 Uitwendige Hashing

3.2.1 Extendible hashing

Deze methode bevat een hashtabel in het geheugen. Deze tabel bevat wijzers naar de schfijpagina's, die maximaal m sleutels met bijhorende gegevens kunnen bevatten. De hashwaarde zijn gehele getallen, met als bereik de breedte van een processorwoord w. De laatste d bits worden gebruikt als indicies in de hashtabel, zodat de tabel 2^d elementen bevat. Deze d is dan ook de globale diepte van de hashtabel en komt overeen met de langste prefix. Alle sleutels waarvan de hashwaarde op dezelfde d bits eindig, komen in dezelfde pagina. Meerdere tabelelementen mogen naar dezelfde pagina verwijzen, daarom wordt er bij elke pagina ook de lokale diepte k bijgehouden: het aantal bits waarmee al haar hashwaarden eindigen. Op die manier bevat elke pagina 2^{d-k} elementen.

- **Zoeken.** Zoeken van de sleutel komt neer op het hashen van deze sleutel, en de overeenkomstige schijfpagina te vinden, en dan deze pagina sequentieel te doorzoeken.

- **Toevoegen.** Toevoegen gebeurt analoog, waarbij gemiddeld de helft van de gegevens in een pagina moeten opschuiven. Indien de pagina vol geraakt moet er gesplitst worden. Deze splitsing gebeurt volgens de waarde van bit $k + 1$. Gegevens waarvoor die bit 1 is worden overgebracht naar een nieuwe gecreëerde pagina, beide met een k dat één groter is als de oorspronkelijke pagina. Nu zijn er nog twee gevallen:
  * $k - 1 \leq d$: De helft van de wijzers naar de oude pagina moeten vervangen worden door de nieuwe pagina, maakt niet uit de welke.
  * $k - 1 = d$: Er was slechts één wijzer naar de oude pagina, en omdat k nu groter is dan d, moet d ook met één toenemen, en de grootte van de hashtabel moet verdubbelt worden. Elke index wordt nu één bit langer, zodat er twee nieuwe indices uit onstaan. De tabelelementen bij beide indicies moeten naar dezelfde pagina verwijzen als de oorspronkelijke index.
- **Verwijderen.** Indien een pagina, na verwijdering van een element, samen met haar broer minder dan m sleutels bevat, moeten deze samengevoegd worden.

3.2.2 Linear hashing

Hoofdstuk 4

Meerdimensionale gegevensstructuren

Notatie: k = aantal dimensies en n = aantal punten.

4.1 Projectie

Deze methode gebruikt per dimensie een gegevensstructuur die alle punten gerangschikt bijhoudt volgens die dimensie. Zoeken gebeurt door een dimensie te kiezen, en alle punten te selecteren die binnen zijn zijde voor die dimensie vallen. Die punten worden dan sequentieel overlopen.

4.2 Rasterstructuur

Deze methode verdeelt de zoekruimte in regelmatige rastergebieden en kan geïmplementeerd worden met een meerdimensionale tabel. Elk rastergebied heeft een gelinkt lijst met punten die in dat gebied liggen, maar kan juist hierdoor onnodig veel geheugen innemen.

4.3 Quadtrees

Deze soort bomen verdeelt de zoekruimte in 2^k hyperrechthoeken en was origineel ontworpen voor 2 dimensies. Deze verdeling wordt opgeslagen in een 2^k-wegsboom. Elke knoop staat voor een gebied, dat onderverdeeld wordt in de 2^k deelgebieden van zijn kinderen. Voor grote k zijn quadtrees niet geschikt, daarom wordt enkel twee dimensies besproken.

4.3.1 Point quadtrees

In deze versie bevat elke knoop een punt. De coördinaten van dit punt delen het gebied op in 4 rechthoeken. De vorm is afhankelijk van de toevoegvolgorde, zodat slechtste geval $O(1)$ is. Zoeken naar een punt vergelijkt telkens het zoekpunt met de punten bij de opeenvolgende knopen, en daalt eventueel af naar het kind met het gepaste deelgebied.

4.3.2 Point-Region quadtrees

Deze vorm vereist dat de zoekruimte een rechthoek is omdat elke knoop de ruimte in vier gelijke rechthoeken verdeelt, zodat elk deel nul of één punt bevat. Inwendige knopen bevatten geen punten. Hier is de vorm onafhankelijk van de toevoegvolgorde, maar kan wel evenwichtig uitvallen. Het is onmogelijk om de hoogte en grootte in functie van het aantal punten uit te drukken.

4.3.3 k-d trees

Een k-d tree gebruikt een binaire boom, waarbij op elk niveau de dimensies afgewisselt wordt. Elke inwendige knoop bevat dan ook één punt, die de zoekruimte verdeelt in de dimensie voor dat niveau. Ideale opsplitsting bestaat uit gelijkmatige verdeling van de dimensies. In twee dimensies zal elk niveau dus afwisselend de x-dimensie en de y-dimensie beschouwen.