Algoritmen en datastructuren 2
Project - De beste binaire zoekboom

Elias Nijs

02008758
Universiteit Gent

ABSTRACT

In dit verslag bespreek ik mijn uitvoering van het project voor Algoritmen en Datastructuren 2 aan de Universiteit Gent 2021. De opgave van dit project bestond erin een Semi-splayboom, een binaire zoekboom, met balancering op aanvraag, en een zoekboom volgens eigen heuristiek optimaal te implementeren aan de hand van benchmarks en theoretische inzichten.

1. Theoretische oefeningen

1.1 Boom 1

1.1.1 Vraag 1

Bewerkingen
Slechtste geval: telkens aan dezelfde kant een node toevoegen
Diepte boom: floor(log(n))

We krijgen dan: log(n) + (1 + log(n)) + (2 + log(n)) + (3 + log(n))

$$\text{Wat het volgende wordt: } \sum_{i=0}^{n-1} (i + \log(n))$$

Zo bekomen we dan een complexiteit voor de bewerkingen van: O(nlogn)

Opbouwen boom
1. get nodes: traverse all nodes in order -> O(n)
2. place with divide and conquer -> O(n)

We bekomen dan een complexiteit van: O(n)

Totaal
We bekomen dan O(nlogn)
Dus O(log(n)) per bewerking.

1.1.2 Vraag 1
Slechtste geval eerst log n toevoegingen aan dezelfde kant en daarna telkens degene aan het uiteinde zoeken.

We krijgen dan terug: O(logn)
Opbouwen is dan terug: O(n)

We bekomen dan: O(nlog(n))
Dus O(log(n)) per bewerking.

1.2 Boom 1

1.2.1 Vraag 1
/

1.2.2 Vraag 1
/

2. De basis binaire zoekboom

Hier zal eerst de basis binaire boom kort overlopen worden zodanig dat de basisfunctionaliteiten hiervan niet meer moet besproken worden tijdens de uiteenzetting van de meer complexe bomen. Deze meer complexe bomen implementeren immers de onderstaande functies mits enkele zeer kleine aanpassing om andere functies in deze te faciliteren.

2.1 Componenten

De basis binaire zoekboom bestaat uit alle verwachte functies, namelijk: root(), size(), search(<value>). add(<value>) en remove(<value>). root() geeft de root van de boom terug, size() de grootte, met search(<value>) kan een waarde opgezocht worden, met add(<value>) kan een waarde toegevoegd worden aan de boom en tot slot kan met remove(<value>) een waarde verwijderd worden uit de boom.

2.2 Enkele design beslissingen

De eerste design beslissing die een vermelding waardig is, is de beslissing om de size als variabele bij te houden in de boom. Dit zorgt ervoor dat wanneer de grootte van de boom opgevraagd wordt, de hele boom niet node per node zal moeten afgegaan worden. Het effect op hoe overzichtelijk de code is, is verwaarloosbaar wanneer deze grote winst in rekening gebracht wordt.

Een andere beslissing die genomen werd, is om recursieve functies maximaal te benutten. Deze zijn overzichtelijker en lenen zich intuïtief meer naar het probleem. Wanneer de boom overlopen moet worden, wordt telkens gebruik gemaakt van recursie. (Een bijkomstige reden waarom hiervoor gekozen werd, is om nog meer intuïtie te verwerven wanneer hiermee gewerkt wordt).

Een laaste belissing, die een vermelding waardig is, is hoe de iterator geimplementeerd is. De iterator werkt aan de hand van een fifo (first in, first out) geïmplementeerd met een door java utils verschafte ArrayDeque. Wanneer de iterator opgevraagd zal een fifo aangemaakt worden en zullen de nodes daarna in volgorde op deze fifo geplaatst worden.

De implementatie van de resterende zaken is zeer straightforward en zal hier dus niet besproken worden.

3. Semi-splay

3.1 The algorithm

Semi-splay werd geïmplementeerd zoals op pagina 15 in de cursus beschreven staat. Wanneer een actie uitgevoerd wordt, wordt eerst het splaypad, dat een globale variabele is, terug vrijgemaakt en vervolgens worden tijdens de actie de overlopen nodes 1 voor 1 toegevoegd aan dit splaypad. Er werd gekozen om van het splaypad een globale variabele te maken omdat deze zeer vaak wordt opgeroepen. Wanneer de actie voorbij is zal er een functie splay opgeroepen worden die vervolgens het aangemaakte pad zal overlopen en er zolang mogelijk splay bewerkingen op uitvoeren. Het splaypad werd na benchmarken geoptimaliseerd door gebruik te maken van een stack, een datastructuur die zich hier beter naar leent vermits we telkens de laatst toegevoegden nodig hebben.

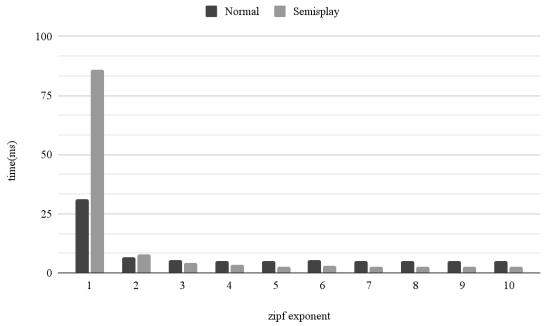
Verder werd ook geprobeerd om de vele if statements weg te werken door de 3 te splayen nodes in elke stap eerst te sorteren zodat we direct de middelste node hebben. Na dit echter geïmplementeerd te hebben werd er beslist om het idee toch niet te gebruiken en werd er teruggegaan naar de if statements. Het sorteerwerk plus de toch nog nodige if statements zorgde voor een te kleine optimalisatie die zo goed als verwaarloosbaar was.

3.2 Benchmarks

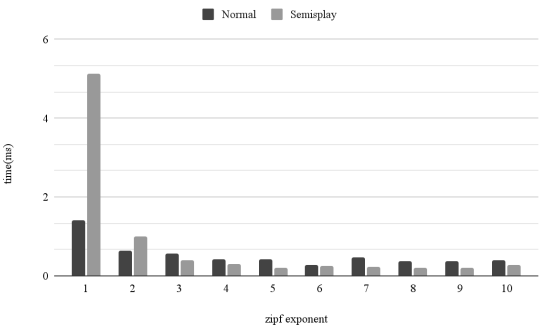
Hieronder enkele benchmarks van Semisplay tegenover een normale BST. (Een test kan gedaan worden door in de `Benchmarker.java` een test en de bomen te definieren.) In onderstaande grafiek kan men duidelijk zien dat semisplay wel degelijk sneller wordt dan een normale BST naarmate de verdeling dichter wordt. Men kan ook het nadeel van semisplay waarnemen, wanneer de verdeling nog wijd is, is de normale BST sneller dan semisplay.

Test parameters:
Amount: 10000, Range of elements: [0, 1000000], Zipf exponent: *variabele,* Times a test is a run: 100

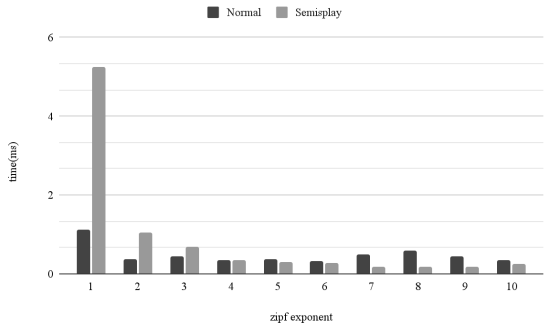
3.2.1 Add operation



3.2.2 Remove operation



3.2.3 Search operation



4. Optimal tree

4.1 The algorithm

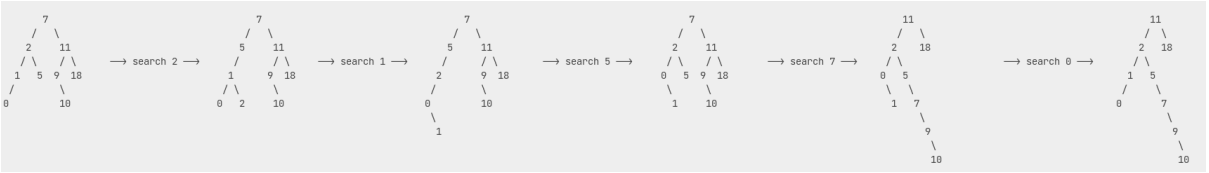
In de Optimal tree moesten twee functies geïmplementeerd worden die beiden een boom opbouwen aan de hand van meegegeven weights. Vermits de eerste functie in essentie een speciaal geval is van de tweede functie, werd de tweede functie gemaakt en wordt deze opgeroepen in de eerste met de juiste parameters. De keys en weights worden normaal doorgegeven en voor de external weights wordt een lijst, opgevuld met 0.0, doorgegeven.

De optimise functie met external weights werd eerst met een eigen implementatie gemaakt. Later, na benchmarks uit te voeren, bleek deze implementatie echter te traag en bleek er nog een fout in te zitten. Om een betere implementatie te kunnen maken werd er vervolgens naar een betere heuristiek/algoritme gezocht op het internet. Uiteindelijk werd deze video, gemaakt door Abdul Bari, gevonden: *https://www.youtube.com/watch?v=wAy6nDMPYAE*. Deze video zet een zeer mooie methode vanuit het dynamisch programmeren uiteen. Het idee achter dit algoritme is om een tabel op te stellen waarin voor elke cel, die overeenkomt met een range van index i to index j in de externalweights lijst, op basis van de vorige cellen de optimale root voor zijn range bijgehouden wordt. Wanneer deze tabel is opgesteld, kunnen we met divide and conquer telkens de cell die correspondeert met de huidige range opzoeken. Hierna krijgen we 2 nieuwe ranges waarmee we terug hetzelfde kunnen doen,... enzovoort, tot alle element geplaatst zijn.

5. My Tree

5.1 Prioritizing unsearched keys

Het idee achter de eerste zelfgemaakte boom was om het scenario waarin keys die al bezocht zijn een zeer kleine kans hebben om opnieuw opgezocht te worden te optimaliseren. De manier die bedacht werd om dit te bereiken was om een bezochte sleutel los te koppelen en aan de onderkant terug toe te voegen. Het bedachte algoritme gaat als volgt:



Search | Input: Node n

---

if n is not found **do**
    return

if n.left and n.right are null **do**
    return
else if n.left and n.right not null **do**
    go down both sides of n until n gets placed
    the tree on which n didn't get placed replaces the original position of n
else if n.left not null **do**
    place n at the bottom of n.left
    n.left replaces the original position of n
else do
    place n at the bottom of n.right
    n.right replaces the original position of n

Helaas na dit geïmplementeerd en geoptimaliseerd te hebben bleek het algoritme te traag te zijn in vergelijking met een normale BST. Het probleem is dat er te veel bewerkingen gedaan worden per search tegenover het aantal bewerkingen die er bespaard worden door deze. Er werd dus beslist om dit idee niet langer te volgen en een nieuwe boom te bedenken. Een belangrijke les die hieruit getrokken werd is dat de wanneer de boom gebalanceerd wordt zeer belangrijk is.

5.2 Left weight vs right weight

De tweede MyTree die geïmplementeerd werd is een boom waarin er geprobeerd wordt om te voorkomen dat 1 kant van de boom veel zwaarder is dan de andere kant.

5.2.1 Wanneer optimaliseren

Om dit te bereiken worden er drie extra variabelen bijgehouden: lw, rw en ratio. Wanneer een node toegevoegd wordt, wordt er op het einde gekeken of deze links of rechts van de root ligt en

afhankelijk van welke kant wordt dan lw of rw plus één gedaan. Een bijkomend idee is om het aantal dat bij lw of rw wordt opgeteld te laten afhangen van de absolute afstand tussen de node en de root, bv root is twaalf en node is twee, dan is de absolute afstand 10. Helaas valt dit niet te implementeren omdat de klasse Node hier geen ondersteuning voor biedt. Het type van deze klasse moet enkel een Comparable implementeren die aangeeft of de waarde groter, kleiner of gelijk is en niet hoeveel dit verschil is. Indien een node verwijderd wordt, zal lw of rw, terug afhankelijk van de positie van de node tegenover de root min één gedaan worden. Tot slot wordt op het einde van elke toevoeg actie gekeken of de tree ongebalanceerd is en indien dit het geval is wordt optimize() opgeroepen. De tree wordt links ongebalanceerd beschouwd indien lw/rw > ratio en rechts ongebalanceerd indien rw/lw > ratio.

5.2.2 Optimaliseren

Het idee om deze boom te optimaliseren is om de lichte kant naar beneden te laten vallen. Dit wordt dus bereikt als volgt:

Optimize left | Input: /

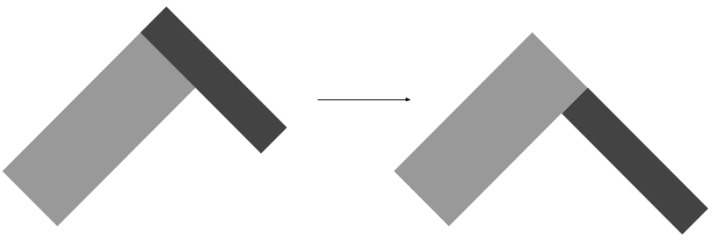
```
prevRoot      <- root
prevRoot.left <- null
newRoot       <- root.left
setRightMost(newRoot, prevRoot);
```

---

setRightMost left | Input: Node n, Node r

```
if n.right is not null do
        setRightMost(n.right, r)
        return
n.right <- r
```

Analoog voor optimize right.



5.2.3 Benchmarking en conclusie

Nadat dit te implementeren werden er hier uiteraard ook benchmarks op uitgevoerd. Helaas was ook dit algoritme te traag. Zelf na te spelen met verschillende ratio's voor de boom was de maximum snelheid voor de boom, indien de ratio zeer hoog lag, net niet even snel als de normale BST. Het probleem met deze methode is dat de optimalisatie die uitgevoerd wordt niet altijd de boom

verbeterd en dit werd over het hoofd gezien bij het uitwerken. Veronderstel bijvoorbeeld het volgende scenario. Maak een nieuwe boom t. Voeg nu 100 toe. Voeg vervolgens 1 toe en nadien all getallen van 2 tot en met 99. voeg nu 101 toe. De tree zal nu een unbalance detecteren en zal vervolgens 100 naar onder verplaatsen, wat de boom minder optimaal maakt. Ook dit algoritme werd dus verwijderd. Een eventueel idee om dit te verhelpen zou kunnen zijn dat een node die verder weg van de root ligt indien we op de visualisatie van de boom de hoogte ineen klappen (zie hieronder) een groter gewicht krijgt dan een node die dicht bij de root ligt. Deze oplossing werd echter niet uitgeprobeerd vermits er ondertussen al aan een nieuw idee gewerkt werd.

```
    5
   /  \
  1    9     →   0 - 1 - 5 - 9
 /
0
```

5.3 Semi-optimize

De derde en laatste MyTree die geïmplementeerd geweest is. Het idee achter deze boom is dat deze zichzelf zal herbouwen aan de hand van weights wanneer een bepaald aantal nodes bezocht zijn.

5.2.1 Wanneer optimaliseren

De boom zal zich optimaliseren indien een bepaald aantal nodes van het totaal aantal nodes bezocht zijn. Belangrijk is dat elke node slechts 1 maal wordt meegerekend. Om dit makkelijk te kunnen oplossen wordt er gebruik gemaakt van een HashSet, een hashset garandeert immers uniciteit. De parameter die bepaalt hoeveel procent dit is, is de MAXUNSTABLE parameter. Deze parameter zal moeten ingevuld worden door degene die de boom implementeert.

5.3.2 Optimaliseren

Om de boom te herbouwen wordt gebruik gemaakt van weights. Elke node start met gewicht 1.0 houdt zijn gewicht bij. Wanneer een node bezocht wordt, zal deze zijn gewicht met een parameter SMLT vermenigvuldigd worden. Op deze manier zorgen we ervoor dat nodes die vaak bezocht worden bovenaan de boom komen te staan. Wanneer de boom herbouwd wordt, wordt er eerst een priorityqueue aangemaakt die 1 op gewicht en nadien op waarde sorteert. Hierna wordt een functie makeQueue opgeroepen die all nodes in boom zal ophalen en toevoegen aan de priorityqueue. Verder wordt de root ook nog op null gezet. Nadat deze stappen genomen zijn, zal het algoritme de elementen 1 voor 1 van de priority queue halen en toevoegen aan de boom. Indien er meerdere keys zijn met hetzelfde gewicht zullen deze eerst allemaal opgehaald worden en vervolgens met divide and conquer aan de boom worden toegevoegd. Doordat de divide and conquer telkens het midden neemt om toe te voegen, zal dit een optimale configuratie zijn.

5.2.3 Benchmarking en conclusie

Het algoritme zorgt in het algemeen wanneer nog geen optimalisatie gedaan is voor een kleine toename in tijd bij de zoekoperatie. Wanneer de optimalisatie gedaan wordt gaat er in het begin heel wat tijd verloren. Deze tijd wordt later echter ingehaald doordat de structuur van de boom beter is, dezelfde zoekopdrachten zullen vooral in tijd winnen. Helaas moet wel erkend worden dat dit algoritme in toepassingen weinig nut zal hebben doordat het een zeer niche situatie verbeterd.