Datastructuren en algoritmen II
Project 1

Titouan Vervack

25 November 2013

Legende

In dit verslag zullen enkele vaste stijlcodes voorkomen, hier opvolgend is een legende:

- **Klassen:** Alle klassen worden in het vet weergegeven.
- Methodes: Alle methodes worden onderlijnd.
- *Variabelen*: Alle variabelen worden cursief weergegeven.

1 Overlappende delen

AbstractTree

Aangezien binaire zoekbomen en semi-splay bomen een aantal gelijkaardige bewerkingen hebben maakte ik een abstracte bovenklasse, **AbstractTree** aan. Deze bevat een default constructor en de methodes contains, add, cost, printTree, size, balance, balancedBuild en treeToArray. Deze boom implementeert ook de interface **BST** zodat geen enkele andere klasse deze interface moet implementeren.

- AbstractTree(): De constructor initialiseert de variabelen $root, added$, $searched$ en $size$. $added$ houdt de laatste **Node** bij die wordt toegevoegd, dit hebben we enkel nodig voor semi-splay. $searched$ doet hetzelfde voor opzoeken. Het bijhouden van $added$ en $searched$ levert een zeer kleine overhead bij het toevoegen bij een binaire zoekboom.
- contains(int key): Kijkt of key voorkomt in de boom. Deze methode houd de laatst opgezochte **Node** bij in $searched$.
- add(int key): Maakt een **Node** aan en voegt deze toe aan de boom op de juiste plaats. Deze methode houd de laatst toegevoegde **Node** bij in $added$.
- add(**Node** $node$): Wrapperklasse voor add(int key) die een **Node** kan toevoegen en niet enkel een key. Wordt gebruikt voor het het balanceren van bomen.
- cost(): Berekent recursief de kost van de gehele boom.
- printTree(): Print de boom uit, wordt gebruikt om de structuur van de boom weer te geven.
- size(): Geeft de grootte van de boom terug.

- balance(): Balanceert de boom.
- balancedBuild(): Hulpmethode voor balance(). Splitst een lijst van keys in twee en voegt het middelste element toe aan de nieuwe boom.
- treeToArray(): Zet een boom om in een gesorteerde lijst van **Node**'s.

Node

Alle implementaties gebruiken **Node** om hun toppen voor te stellen. Node bevat de variabelen $key, parent, leftChild, rightChild$ en $weight$. Er bestaan accessor methodes voor alle variabelen, behalve setKey(int *key*). *parent* wordt enkel gebruikt bij semi-splay maar ook dit levert maar een zeer kleine overhead voor de binaire zoekbomen.

2 Sbst

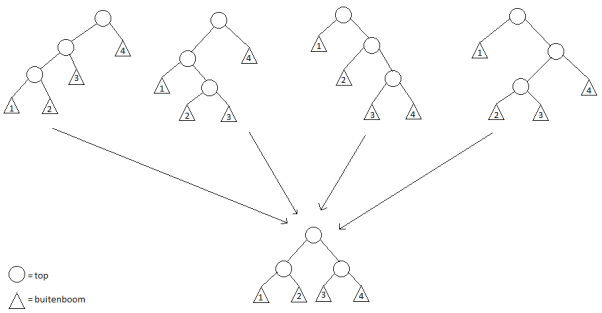
Voor semi-splay gebruiken we de abstracte bovenklasse **Sbst**. Deze klasse bevat wrapperklassen voor add(int *key*) en contains(int *key*) en erft **AbstractTree** over. Het enige dat **Sbst1** en **Sbst2** dan nog moeten implementeren is splay(**Node** *start*).

2.1 Sbst1

Dit was mijn eerste implementatie. Ik wist al hoe semi-splay kon gedaan worden, zoals we handmatig doen in de les. Maar dit betekent dat er veel conditionele expressies moeten gebruikt worden. Ik zocht dus eerst een algoritme dat met zo weinig mogelijk conditionele expressies werkte.

Semi-splay

Het semi-splay algoritme werkt in een bottom-up manier beginnend bij *added*. A.h.v. *added* bepalen we de drie toppen waarop we moeten splayen en houden ze bij in de array *splay*. Nu bewaren we ook de vier (lege) buitenbomen van deze toppen in de array *buitenbomen*. De splay toppen kunnen we gewoon sorteren om de juiste ordening te krijgen. De buitenbomen zijn iets moeilijker. Om te weten hoe de buitenbomen gesorteerd worden was **Nodewrapper** een noodzaak aangezien de buitenbomen null kunnen zijn. **NodeWrapper** heeft een **Node** *node* en een int *index*. We sorteren (m.b.v. **NodeWrapperComparator**) de buitenbomen op *index*. Als we gelijk welke ordening van drie toppen en zijn (lege) buitenbomen van links naar rechts bekijken, hoogte buiten beschouwing gelaten, dan zijn de buitenbomen al in de juiste volgorde. Dit is te zien in Figuur 1.



Figuur 1: Sortering van de buitenbomen

Dit van links naar rechts bekijken simuleren we m.b.v. de formule $depth * (-leftChild * rightChild)$. Hierbij is *depth* de diepte van de splay toppen, 1 zijnde de diepste, 2 de middelste en 3 de minst diepe. $leftChild$ is 1 als we bezig zijn over een linkerkind en anders is het 0. Hetzelfde geld voor *rightChild* maar dan uiteraard omgekeerd. Hierna moeten enkel nog de referenties van de toppen (ouders en kinderen) worden aangepast. De kost van een splay bewerking met dit algoritme ligt een constante hoger dan het normale algoritme aangezien er arrays van 3 en 4 elementen moeten gesorteerd worden.

2.2 Sbst2

Dit is mijn tweede implementatie. Hier gebruik ik de manier die we handmatig in de les toepassen. Ze is langer en bevat veel conditionele expressies maar ze is wel veel (een constante) sneller aangezien er geen sortering moet worden toegepast. De kost van dit algoritme is $O(log(n))$. Het toevoegen van een element vergt ook nog tijd $O(log(n))$ waardoor de totale complexiteit voor een element toe te voegen $O(2log(n)) = O(log(n))$ is.

3 Obst

Ik heb vier versies van obst geïmplementeerd. De vier versies zijn eigenlijk het verloop van recursie naar dynamisch programmeren. Aangezien elke versie een verbetering van de vorige optimize() (eigenlijk generateCostTable) bevat, zijn er enkele overlappende methodes. Daarom maakte ik de abstracte bovenklasse **Obst** aan. Deze bevat de methodes optimize() en al zijn hulpmethodes.

- optimize(): Zet de boom om in een boom met minimale kost. Hierin wordt de boom omgezet in een gesorteerde lijst van **Node**'s. Daarna worden de sommen van alle frequenties berekend. Tenslotte berekenen we de *cost* en de *roots* tabel en bouwen hiermee de nieuwe boom met minimale kost op.
- generateFreqSums(): Genereert alle mogelijke sommen van frequenties die voorkomen in de recursieve formule voor optimize().
- buildOptimalTree(): Bouwt een boom met minimale kost op a.h.v. een gesorteerde lijst van alle **Node**'s van de originele boom en een lijst met de wortels van de optimale subbomen.
- generateCostTable(): Genereert een tabel met alle kosten, *cost*, van de optimale subbomen en maakt ook de *roots* tabel aan.

3.1 Obst1

Zoals al mijn implementaties is dit gewoon de formule uit de opgave omgezet in code. Dit is de naïeve manier en gebruikt recursie. Het probleem hier is getMin(). Dit is een functie met een complexiteit van $O(n)$. Als optimalisatie worden wel alle frequentiesommen op voorhand berekend.

3.2 Obst2

Deze implementatie is dezelfde als **Obst1** maar als optimalisatie is getMin() ingeruild voor *min* in de for lus. Het berekenen van *min* in de for lus kan in tijd $O(1)$, veel beter dan de $O(n)$ van getMin() dus.

3.3 Obst3

Deze manier is iets slimmer, als de kost al gekend is wordt deze gewoon uit de tabel gehaald. Indien ze nog niet bestaat (dus als ze 0 is) dan wordt ze recursief berekend dit is dus gedeeltelijk dynamisch programmeren.

3.4 Obst4

Dit is de finale en beste versie die ik schreef. Hier is geen recursie aanwezig. In plaats van de *cost* tabel n voor n te overlopen en in te vullen wordt hier eerst de middendiagonaal van [0][0] tot [n][n] ingevuld. Eens dit gedaan is vullen we diagonaal per diagonaal (van het midden naar boven) in. De tijdscomplexiteit van dit algoritme is $O(n^3 + n) = O(n^3)$. $O(n)$ om de frequentiesommen te berekenen en $O(n^3)$ omdat er een drievoudige for lus wordt gebruikt in optimize(). De geheugencomplexiteit is $O(3n^2 + n) = O(n^2)$ aangezien er 3 dubbele arrays gebruikt worden: $cost, roots, freqSums$ en een enkele array: $tree$.

Nieuwe boom bouwen

Om de nieuwe boom op te bouwen gebruiken we buildOptimalTree(Node[] tree, int[][] freqSums, int[][] roots). $roots[x][y]$ bevat de index van de beste wortel voor de subboom opgebouwd uit sleutels x $\rightarrow$ y. De wortel van de nieuwe boom staat altijd in $tree[roots[size - 1][size - 1]]$. Daarna kunnen de kinderen altijd als volgt gevonden worden (met $index = roots[x][y]$):

- $leftChild = tree[roots[x][index - 1]]$
- $rightChild = tree[roots[index + 1][y]]$

Dit zijn de wortels die zorgen voor de minimale kost van de subbomen T[i, k - 1] en T[k + 1, j] die we berekenden in generateCostTable(). Aangezien elk element toegevoegd moet worden is de tijdscomplexiteit van dit algoritme $O(n)$.

4 Tests

4.1 Correctheidstest

Om zeker te zijn over de correctheid van mijn algoritmes heb ik getest op kleine voorbeelden. Zowel zelf bedachte als op het internet gevonden voorbeelden, aangezien deze al een oplossing hadden. Ook de test van de feedback werd uitgevoerd. Sommige implementaties heb ik stap voor stap met de debugger bekeken om de bugs er uit te halen. Hier vond ik ook enkele fouten die ik anders waarschijnlijk niet ging gevonden hebben. Als verdere test werden eerst grote datasets toegevoegd aan de bomen en werden er dan opzoekingen op gedaan om na te gaan of er nullpointer exceptions te vinden waren. Eens ik zeker was dat dit niet het geval was heb ik de implementaties tegen elkaar laten lopen en hun output vergeleken a.h.v. printTree().

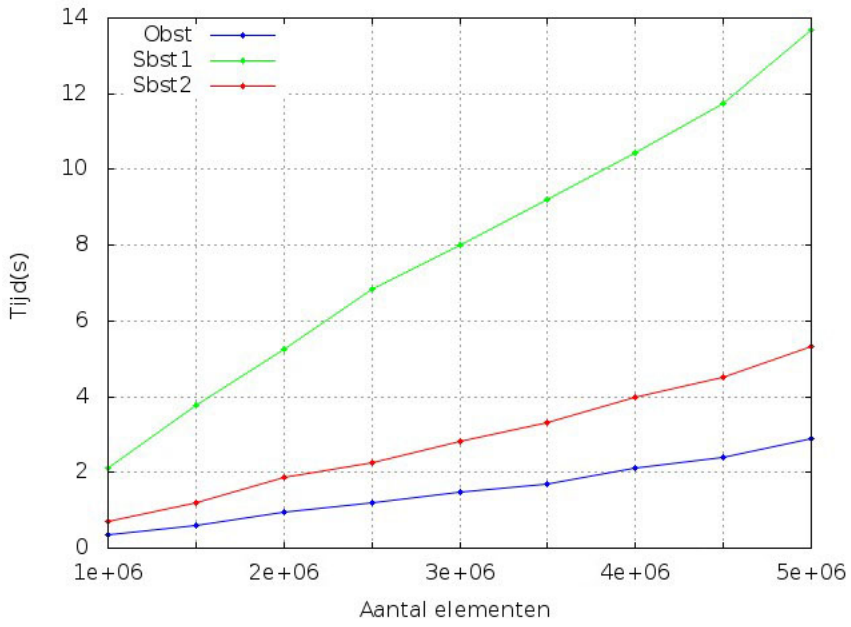
4.2 Testconfiguratie

De datasets die ik heb gegenereerd zijn deze met grootte $5.000 - 9.000$ in stappen van 500 om optimize() te testen. Deze werden gemaakt met `random_zipf` met ongelijkmatigheid $0 - 0,9$ in stappen van 0,1. Om sbst en obst te testen werden sets gemaakt met grootte $1.000.000 - 5.000.000$ in stappen van 500.000 en dezelfde verdeling als voor optimize(). Er zijn telkens 10 keer meer opzoekingen dan elementen aangemaakt. Alle tests, (behalve optimize() bij obst en add() en contains() op sbst), zijn 3 maal uitgevoerd en daarvan werd dan het gemiddelde

berekend. Alle tests bevinden zich in de klasse **Test**. Deze klasse bevat ook een methode warmup() waarmee de JVM wordt opgewarmd en een methode clean() waarmee geprobeerd wordt om de garbage collector zijn werk te laten doen. Deze tests worden dan opgeroepen vanuit de main. Om al mijn resultaten op te slaan wordt de **Logger** gebruikt. Deze klasse bevat een aantal gelijkaardige methodes die informatie in een mooi- en een puur data formaat uitprinten naar 2 verschillende bestanden.

4.3 Opbouwen van bomen

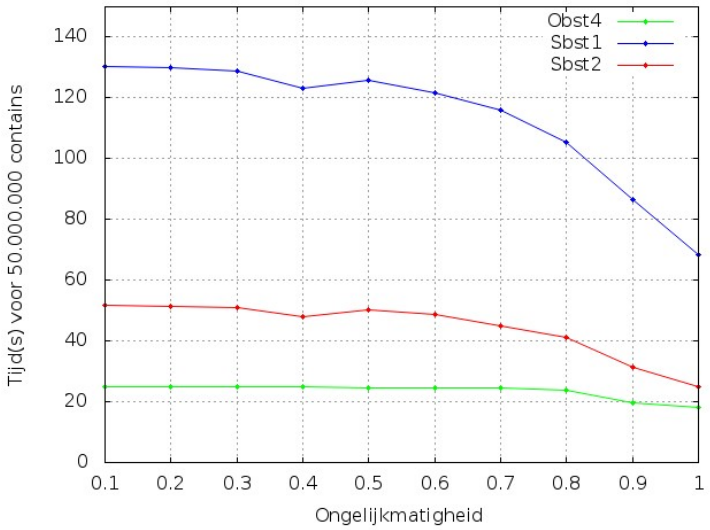
Eerst hebben we het verschil in opbouwtijd van alle bomen bekeken. **Obst1/2/3** worden niet getest omdat deze enkel verschillen in de implementatie van optimize(). We zien in figuur 2 dat **Obst4** sneller is dan **Sbst1** en **Sbst2**. Dit is logisch aangezien er bij sbst wordt toegevoegd zoals bij obst maar dan wordt er ook nog gesplayed. De reden waarom **Sbst1** zo veel trager is dan de andere 2 bomen hebben we gezien tijdens het bespreken van de complexiteit, deze is namelijk een grote constante trager dan **Sbst2**.



Figuur 2: Tijd voor het opbouwen van bomen

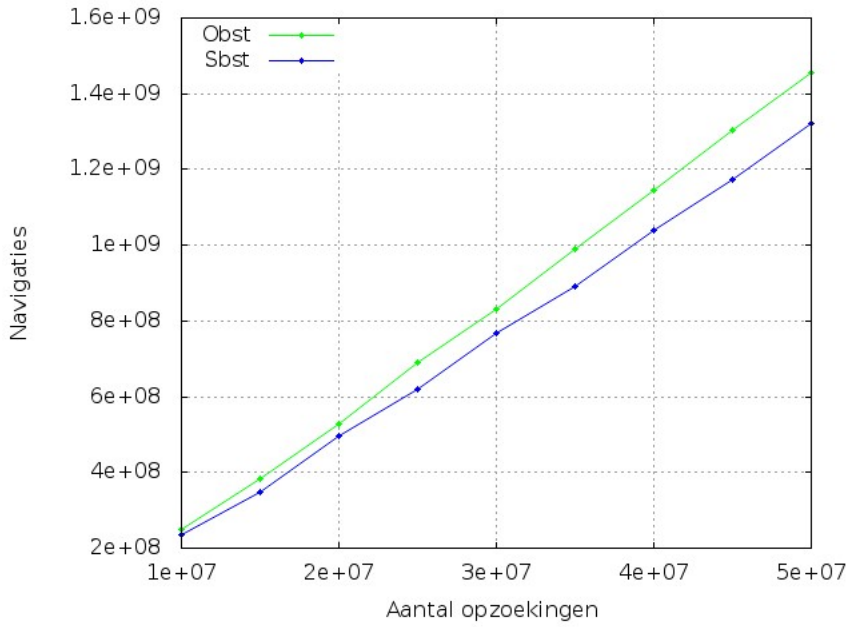
4.4 Opzoeken in bomen

Voor het opzoeken hebben we ook rekening gehouden met de ongelijkmatigheid. We zien in figuur 3 dat er bij alle bomen minder tijd nodig is om de zoekbewerkingen te voltooien als de verdeling van de zoekbewerkingen minder gelijkmatig is. Bij sbst is dit te wijten aan het feit dat een bepaalde verzameling van sleutels (deze die vaak worden opgezocht) naar boven wordt gesplayed wat ervoor zorgt dat er telkens minder en minder tijd nodig is om deze vaak bezochte sleutels te vinden. Bij obst is dit echter toevallig, het had evengoed langer kunnen duren. Aangezien er bij obst geen aanpassingen gebeuren wanneer we zoeken, hangt de opzoektijd gewoon af van waar in de boom de vaker bezochte sleutels liggen. Liggen deze bovenaan in de boom, dan zal de opzoektijd verminderen, als deze diep in de boom zitten zal de opzoektijd vergroten. Vervolgens vergeleken we wat er gebeurde wanneer we bij een gelijkmatige/ongelijkmatige (figuur 4 en 5) verdeling opzoekingen deden maar dit keer meten we de navigaties en niet de tijd



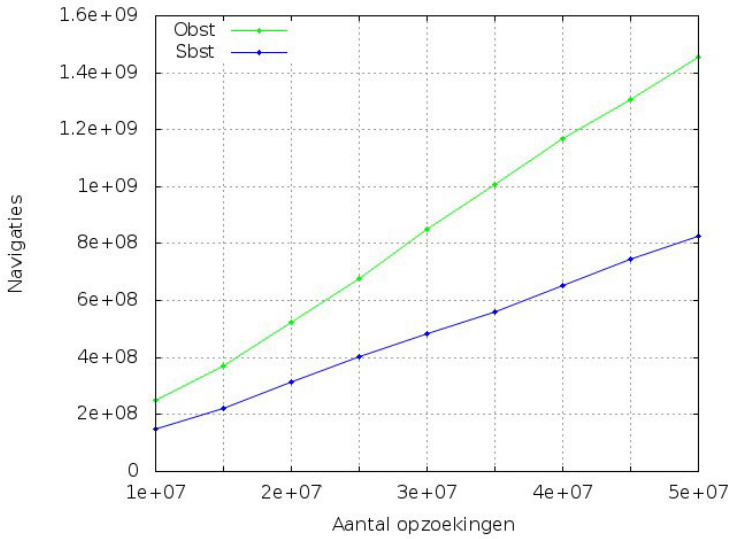
Figuur 3: Tijd voor te zoeken in een boom van 50.000.000 elementen

in seconden. Hier zien we duidelijk het voordeel van semi-splay. Deze is in alle gevallen sneller



Figuur 4: Navigaties bij een gelijkmatige verdeling voor een boom van 50.000.000 elementen

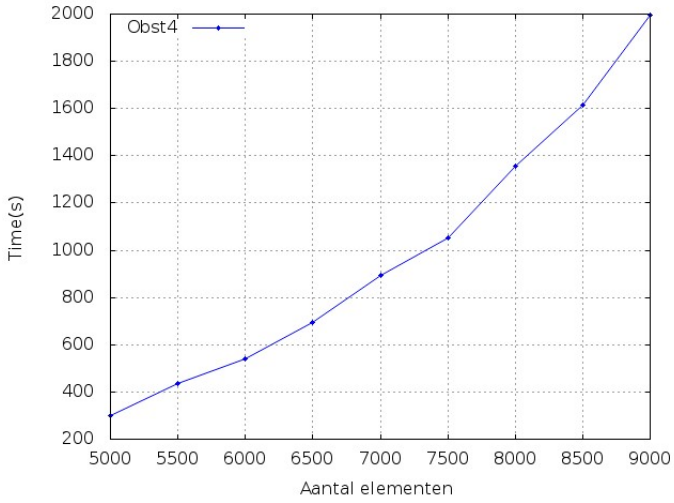
omdat hij beter gebalanceerd is. In realiteit is hij echter trager omdat hij ook nog eens moet splayen. Als de boom en/of de ongelijkmatigheid en/of het aantal opzoekingen groot genoeg is, dan zal sbst wel beter presteren. We zien dit al gebeuren als we kijken naar figuur 5.



Figuur 5: Navigaties bij een ongelijkmatige verdeling voor een boom van 50.000.000 elementen

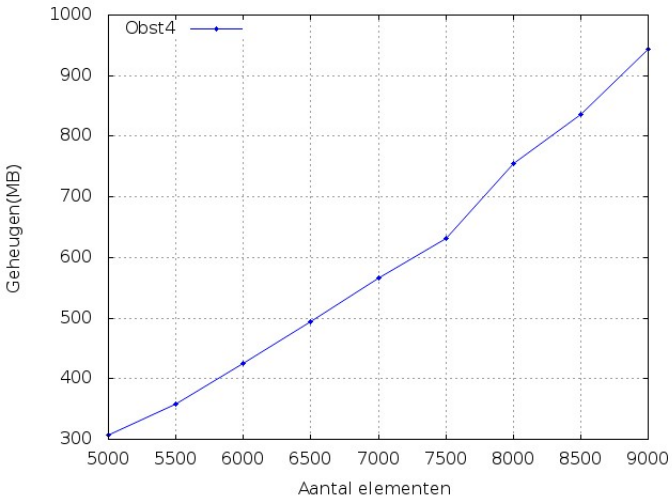
4.5 Optimaliseren van bomen

Obst1/2/3 worden niet gebruikt om dat deze implementaties maar acceptabele tijden kunnen geven voor bomen met ongeveer 20 elementen. We hebben eerst gekeken naar het tijds- en geheugenverbruik van optimize(). We zien dat deze stijgt zoals we bespraken in het theoretisch gedeelte. We zien in figuur 6 dat het gigantisch lang duurt om een boom optimaal te balanceren.



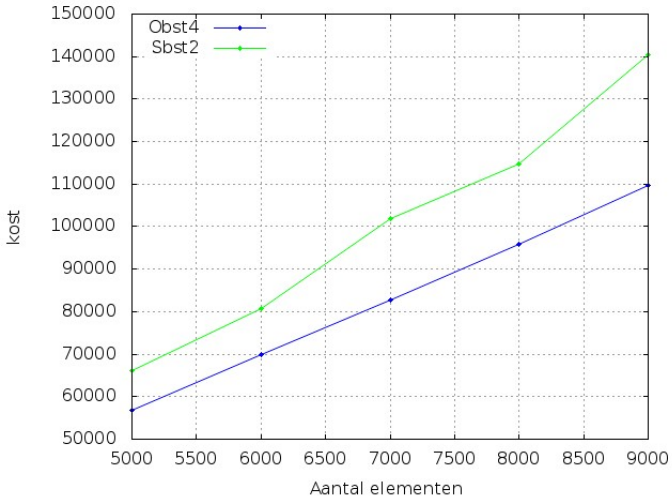
Figuur 6: Tijd nodig om te optimaliseren

Ook wordt er zeer veel geheugen voor maar een weinig aantal elementen gebruikt. Zie figuur 7.



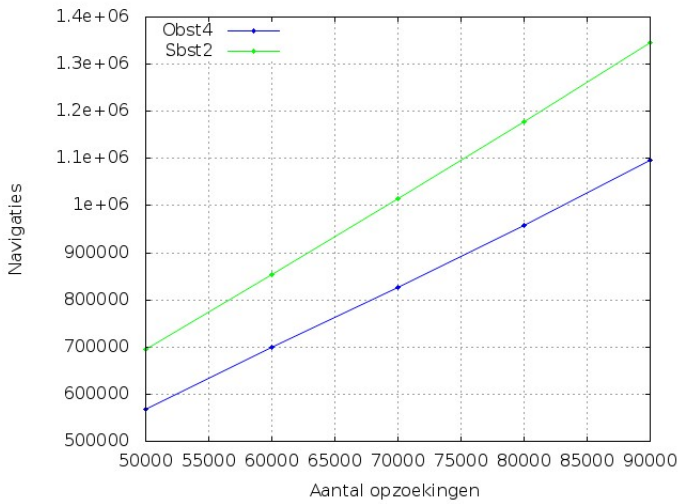
Figuur 7: Geheugen verbruik tijdens optimaliseren

Op het eerste zicht ziet optimize() er dus niet voordelig uit. Om hier iets aan te doen hebben we optimize() vergeleken met **Sbst2**. Eerst hebben we vergeleken wat het verschil in kost was. Dit is te zien in figuur 8. Hier zien we duidelijk dat de kost van obst redelijk wat onder deze van sbst ligt.



Figuur 8: Verschil in kost tussen obst en sbst

Wat dit dan als gevolg heeft voor het opzoeken van elementen is te zien in figuur 9. We zien dat de opzoekingen veel sneller gebeuren. Dit is zoals we zouden verwachten van semi-splay.



Figuur 9: Aantal navigaties bij een boom van 9.000 elementen

4.6 Besluit

We kunnen nu besluiten dat je best een binaire zoekboom gebruikt als het opzoeken en toevoegen van elementen in je dataset op alle momenten snel moet gebeuren en er nooit een moment van rust is. Als je weet dat de dataset zeer groot zal zijn of dat een bepaalde subset zeer vaak zal worden opgezocht, dan is een zelfbalancerende boom zoals sbst een goede keuze. Dit is omdat de zoekopdrachten minder en minder lang zullen duren. Het toevoegen zal wel iets langer duren. Als er echter bepaalde momenten zijn dat de dataset niet wordt geraadpleegd, bvb 's nachts is er geen toegang tot de dataset. Dan is het een goede keuze om tijdens die tijd optimize() uit te voeren. We besluiten hieruit dat er geen beste boom is maar dat elke boom optimaal is voor bepaalde situaties.

5 Theoretische vragen

5.1 Vraag 1

Om het totaal aantal navigaties laag te houden heb ik 2 manieren bedacht:

- Elke keer we add oproepen kijken we in een hashmap of het element er al in zit. Dit heeft een constante kost. Als het element er nog niet in zit dan plaatsen we het nieuwe element in de boom en slaan het aantal navigaties op in de hashmap met als sleutel het nieuwe element. Bij contains() kijken we dan ook eerst of het te zoeken element in de hashmap zit. Zit het erin dan geven we gewoon de waarde van de hashmap (aantal navigaties nodig om aan de sleutel te geraken) terug. Bij sbst is dit echter niet mogelijk aangezien we niet weten hoe er moet gesplayed worden. Zit hij niet in de hashmap dan weten we dat de sleutel niet in de boom zit en geven we gewoon -1 terug. Elke keer een element wordt opgezocht dat nog niet in de boom zit verhogen we de waarde in de hashmap van dat element met 1. Als het dan ooit wordt toegevoegd stellen we deze waarde in als gewicht van de top. Hierdoor zal hij al direct hoog komen te staan in het geval deze sleutel al vaak werd gezocht. We moeten wel de definitie van contains() aanpassen. Als de sleutel namelijk nog niet in de boom zit dan kan er niet teruggegeven worden hoeveel navigaties nodig waren om dit element te vinden. De tijdscomplexiteit blijft dezelfde maar de er zal wel veel meer geheugen verbruikt worden.

- De tweede methode maakt gebruik van zogenaamde grafstenen. **Node** krijgt een extra veld *boolean dead* om aan te tonen of de node al bestaat of niet. Als we een element opzoeken dat niet in de boom staat wordt het gewoon toegevoegd op de plaats waar het zou moeten staan maar zetten we *dead* op true. Toppen die worden bezocht maar veld *dead* op true hebben krijgen gewoon +1 bij hun gewicht. Als een top wordt toegevoegd waarvan het veld *dead* op true staat, wordt *dead* op false gezet. Sbst gaat automatisch toppen die niet echt bestaan maar wel vaak bezocht worden bovenaan zetten. Als optimize() wordt uitgevoerd zullen de grafstenen ook hoger komen te staan. contains() moet worden aangepast zodat het een negatieve waarde teruggeeft als *dead* op true staat.

5.2 Vraag 2

Als vergelijkingen gebruikt worden i.p.v. navigaties en altijd eerst naar het linkerkind wordt gekeken, dan naar de parent en dan pas het rechterkind, dan zal de kost 1 zijn om naar links te gaan en 2 voor de andere gevallen. De formule moet nu als volgt aangepast worden:

$$c(T[i,j]) = \sum_{k=i}^{j} c(v_k) = \sum_{k=i}^{j} (v(p_k) + v(v_k)) * w(v_k), i <= j$$

Met $v(p_k)$ het aantal vergelijkingen van de parent en $v(v_k)$ het aantal vergelijkingen die nodig waren om te vinden in welke richting we op diepte k moesten bewegen: links, rechts of niet bewegen (als sleutel gelijk is aan v_k). Het dynamisch programmeren algoritme moet niet worden aangepast. **Node** moet wel een extra veld *int vergelijkingen* krijgen dat bijhoudt hoeveel vergelijkingen nodig zijn om dit element te bereiken. Wanneer we bij optimize() dan de optimale boom maken mag de boom niet hergeorganiseerd worden maar moet hij opnieuw opgebouwd worden. Dit kan in $O(nlog(n))$ waardoor het totale algoritme $O(n^3 + nlog(n)) = O(n^3)$ wordt. Dit is nog altijd even snel, op een constante na. Als we deze aanpassingen niet toepassen, zullen opzoekingen in de realiteit langer duren. Dit omdat optimize() en sbst de voorkeur zullen geven aan linkerpaden. Er gaan zich langere linkerpaden vormen. Dit zorgt er dus ook voor dat er minder elementen dicht bij de wortel kunnen staan.