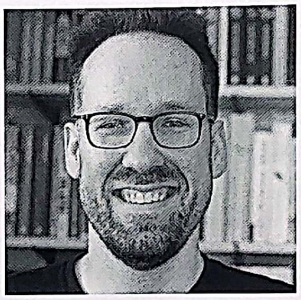
De algoritmes van het JAVA COLLECTIONS FRAMEWORK

"Does anyone actually use LinkedList? I wrote it, and I never use it." Dat tweette Josh Bloch, de schrijver van het Java Collections Framework, ooit [1]. Maar waarom niet? En wat is een LinkedList eigenlijk?



Jan Ouwens is senior ontwikkelaar bij Yoink en maker van EqualsVerifier. Hij studeerde Technische Informatica in Eindhoven, waar hij twee kansen had om Edsger Dijkstra te ontmoeten en ze allebei miste.

Het Java Collections Framework biedt tientallen verschillende datastructuren. Libraries zoals Guava, Eclipse Collections en Vavr voegen daar nog een groot aantal aan toe. Maar in de praktijk gebruik je vooral ArrayList, HashMap en soms HashSet. Dat heeft voor een groot deel te maken met de performancekenmerken van de algoritmes achter deze datastructuren. Dit artikel legt uit hoe dat zit, en wat je kunt winnen door te kijken naar die andere soorten Lists, Maps en Sets.

{ GEHEUGEN }

Eerst moeten we weten hoe geheugen werkt. Gelukkig hoeft dat voor ons verhaal niet in detail. Het is genoeg als je weet dat het geheugen bestaat uit een lange reeks van plaatsen die een waarde kunnen bevatten, en die je kunt benaderen via een unieke en opvolgende index. Denk aan een straat waarin elk huis zijn eigen huisnummer heeft. Een ArrayList slaat al zijn waardes achter elkaar op in een aaneengesloten blok geheugen (zie afbeelding 1). Een LinkedList is heel anders: die bestaat uit een reeks van cellen die elk op een willekeurige plaats in het geheugen zijn opgeslagen, en die elk een waarde bevatten en een pointer naar de volgende cel (zie afbeelding 2).

{ OPVRAGEN }

Stel nu dat je twee lijsten hebt: één ArrayList en één LinkedList, allebei met dezelfde inhoud, en stel dat je het zesde element uit die lijst wilt opvragen. In het geval van de ArrayList weet Java op welke plaats in het geheugen de lijst is opgeslagen. Die locatie plus 5 (want we beginnen bij 0 te tellen) is de locatie waar het zesde element is opgeslagen: 5. We hoeven dus maar één ding te doen om op de juiste geheugenlocatie te komen, namelijk twee getallen bij elkaar optellen.

Bij onze LinkedList is dat anders. Omdat je niet weet waar elke cel zich in het geheugen bevindt, zul je ze allemaal langs moeten gaan door telkens de pointer naar de volgende cel te volgen. Op die manier loop je van 0, via 1, 1, 2 en 3 naar het gezochte nummer 5. Nu moeten we twaalf dingen doen om op de juiste geheugenlocatie te komen, namelijk zes geheugenplaatsen uitlezen en zes keer springen naar de volgende plek in de lijst.
Het opvragen van een waarde in een lijst is een voorbeeld van een algoritme. De hoeveelheid dingen die je moet doen om het **algoritme** uit te voeren, heet **complexiteit**.

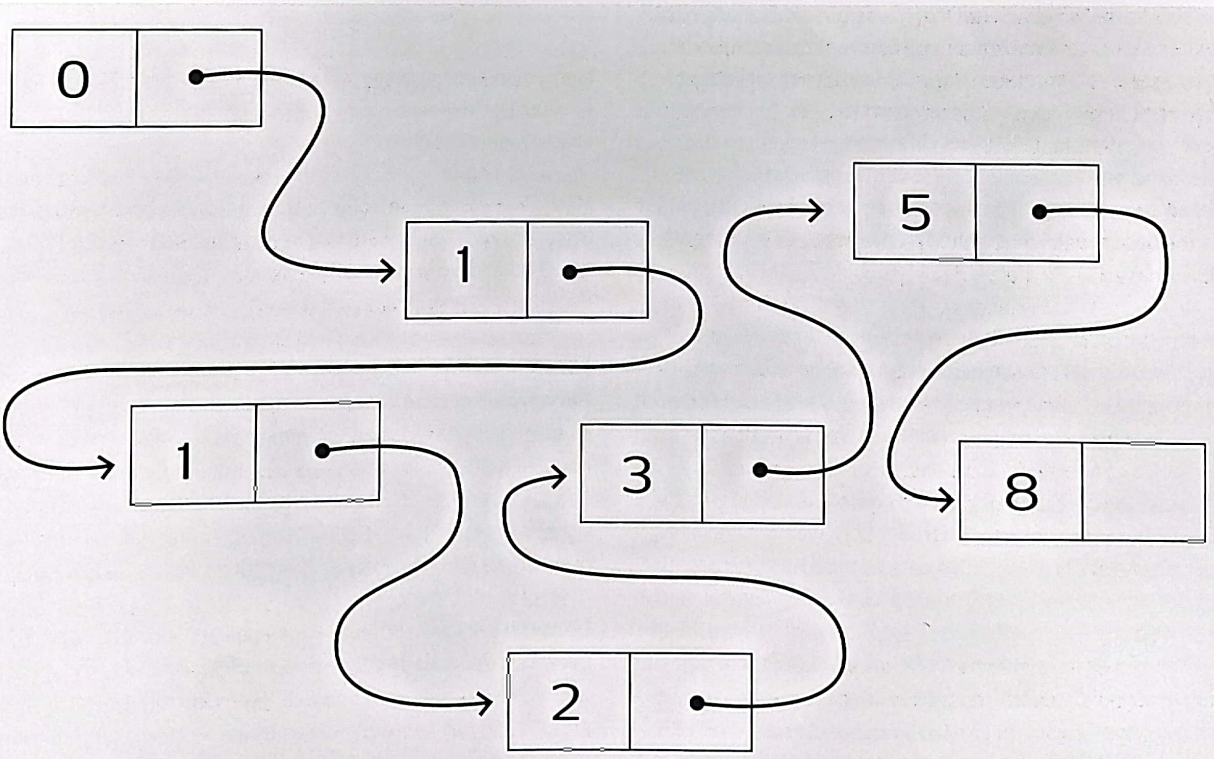
{ BIG O }

In de informatica gebruiken we de **Big O notation** om complexiteit uit te drukken. Daarin zijn we best wel lui, want we laten een hoop informatie weg. We hebben gezien dat we voor het opvragen in een ArrayList altijd maar één handeling hoeven te verrichten. In de Big O notation zeggen we dan dat dit algoritme "orde 1" heeft: O(1). Een ander woord hiervoor is dat het algoritme constant is.

Bij het opvragen in een LinkedList is de complexiteit afhankelijk van het aantal elementen in de lijst. Om het zesde element op te

					0	1	1	2	3	5
8	13	21	34	55	89	144				

Afbeelding 1: Geheugen met daarin een ArrayList. De lege geheugenplaatsen zijn gereserveerd voor andere dingen.



Afbeelding 2: Schematische weergave van een LinkedList.

zoeken moesten we 12 dingen doen. Het is makkelijk om na te gaan dat je, om het tiende element op te zoeken, 20 dingen moet doen, en om het tweede element op te zoeken 4 dingen moet doen. Oftewel, als je het N-de element wil opvragen, moet je 2 x N dingen doen. Volgens de Big O notation heeft dit algoritme daarom een "orde N": O(N). Een ander woord hiervoor is dat het algoritme lineair is. Het maakt hierbij niet uit dat je voor elk element twee dingen moet doen, en het maakt ook niet uit of je het eerste element wil opvragen, of juist het laatste.

Eigenlijk is het best gek dat we in een vak dat bekend staat om zijn pietluttigheid, zo achteloos zijn over iets zo fundamenteels. Het gaat erom dat we willen kunnen zien hoe snel het aantal bewerkingen toeneemt als de hoeveelheid data groeit. Een constant algoritme blijft even groot als de hoeveelheid data groeit, en een lineair algoritme groeit lineair mee met de hoeveelheid data. Er zijn ook algoritmes die kwadratisch groeien ($O(N^2)$), zoals sommige sorteeralgoritmes, en er zijn algoritmes die exponentieel groeien (O(2^N)), zoals algoritmes om wachtwoorden te kraken [2].

{ INVOEGEN EN TOEVOEGEN }

Goed, terug naar de Java Collections. Opvragen is niet het enige wat we doen in een List. Stel dat je een element wil invoegen aan het begin van een lijst. Nu zijn de rollen van de ArrayList en de LinkedList omgedraaid!

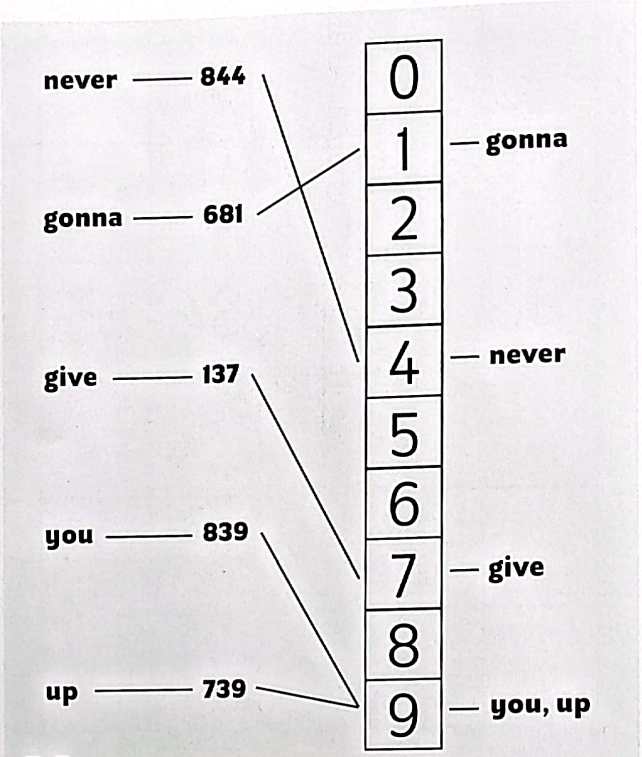
Om bij een ArrayList een element in te voegen aan het begin, zul je elk element een plaats moeten opschuiven, zodat er aan het begin een plekje vrijkomt. Dat is O(N). Maar bij een LinkedList maak je een nieuwe cel. Die laat je verwijzen naar het eerste element van de bestaande lijst, en je vertelt je LinkedList dat je nieuwe cel voortaan de eerste cel is van de lijst. Dat zijn twee handelingen, maar omdat we lui zijn, noemen we dat O(1), omdat het aantal handelingen niet verandert naarmate de lijst langer wordt.

Bij het toevoegen en weghalen aan het einde van de lijst, is de ArrayList weer in het voordeel, omdat je eerst het einde van de lijst moet vinden (O(N) voor een LinkedList!), en er in een ArrayList niks meer geschoven hoeft te worden.

{ LISTS }

Met deze informatie kunnen we beredeneren waarom we in Java eigenlijk altijd naar de ArrayList grijpen, en niet naar de LinkedList. Dingen die we veel doen met lijsten, zijn toevoegen aan het eind in plaats van aan het begin (O(1) voor ArrayList en O(N) voor LinkedList), itereren over de elementen (O(N) voor beide soorten lijst), en random access (O(1) vs O(N)).

Invoegen aan het begin van een lijst, waar een LinkedList goed in is, doen we alleen in heel specifieke situaties, bijvoorbeeld als we een queue moeten implementeren: in een queue plaats je nieuwe



Afbeelding 3: Schematische weergave van een HashSet met tien buckets.

elementen aan het begin van de lijst, en haal je ze aan het einde er weer af. LinkedList is dan ook een bekende implementatie van Java's Queue interface.

Maar als je een queue nodig hebt, is dat vaak om dingen te delen tussen threads, en LinkedList is niet thread-safe. Meestal zul je dus voor een andere implementatie van Queue kiezen. Dat verklaart de tweet van Josh Bloch.

{ SETS }

Tijd om naar andere collections te gaan kijken, zoals naar de Set en de Map. Deze lijken erg op elkaar; sterker nog: in Java wordt de HashSet onder water geïmplementeerd met de HashMap waarvan de values genegeerd worden! Ook de TreeSet gebruikt onder water een Map. Omdat Sets conceptueel simpeler zijn dan Maps, negeren we de Maps in de rest van dit artikel.

De meest gebruikte implementatie van Set in Java is de HashSet. Deze datastructuur heeft de ongelooflijke eigenschap dat alle operaties O(1) zijn. Maar hoe dan!? Een HashSet is eigenlijk een array van genummerde buckets (zie afbeelding 3). Bij het toevoegen van een element wordt eerst de hashCode() van dat element berekend. Dit getal modulo het aantal buckets [3], is de bucket waarin het element wordt gestopt.

Bij het opvragen van een element gebeurt hetzelfde: de hashCode() wordt bepaald. Dit getal modulo het aantal buckets geeft de bucket waarin het element moet zitten, als het erin zit.
Let wel op: als er meer dan één element in een bucket zit, worden deze in een lijst geplaatst, en moet er in de lijst gezocht worden naar het element (en worden alle operaties plotseling O(N)). Als al je elementen dezelfde hashCode() hebben (wat mag volgens het contract van hashCode() [4]), had je qua complexiteit net zo goed een LinkedList kunnen gebruiken. Daarom is het heel belangrijk dat je hashCode() functie een goede spreiding heeft, zodat je elementen over zo veel mogelijk verschillende buckets verspreid worden.

{ ANDERE IMPLEMENTATIES }

Hoe zit het nu met al die andere implementaties van List, Set en Map die je kunt vinden in het Java Collections Framework? Ze brengen allemaal iets nieuws en interessants met zich mee, maar qua complexiteit zullen ze het doorgaans niet winnen van ArrayList, HashSet en HashMap. Dat betekent niet dat je er niks aan hebt!

TreeSet en TreeMap hebben bijvoorbeeld een vaste iteratievolgorde gebaseerd op de waarde van de elementen (dus niet op de volgorde waarin ze zijn toegevoegd!) De collections uit het java.util.concurrent package zijn iets trager, maar kunnen tegelijk benaderd worden vanuit verschillende threads, wat met de gewone collections niet kan.
En dan zijn er nog de collections uit externe libraries, zoals Google Guava [5], Vavr [6] en Eclipse Collections [7]. Het loont zeker de moeite om op hun websites te kijken welke eigenschappen die hebben, en te zien wat voor jouw project interessant kan zijn. Als je bijvoorbeeld een immutable list wil gebruiken, zijn die van Vavr en Eclipse Collections veruit superieur aan wat je krijgt van Java's eigen Collections.unmodifiableList.

{ CONCLUSIE }

In dit artikel heb ik in het kort geprobeerd uit te leggen wat de verschillen zijn tussen de verschillende collections, aan de hand van de complexiteit van de algoritmes waarmee ze geïmplementeerd zijn. Er valt natuurlijk veel meer over te vertellen dan in dit artikel past. Als je het onderwerp interessant vindt, raad ik je van harte aan om Jay Wengrow's "Common-Sense Guide to Data Structures and Algorithms" [8] te lezen: dit boek gaat op een speelse manier veel verder de diepte in. <

{ REFERENTIES }

1 **https://twitter.com/joshbloch/status/583813919019573248**
2 Dit is waarom het loont om lange wachtwoorden te gebruiken!
3 Eigenlijk had de HashSet net zo goed BucketSet kunnen heten...
4 **https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#hashCode()**
5 **https://github.com/google/guava/wiki**
6 **https://docs.vavr.io**
7 **http://www.eclipse.org/collections**
8 **https://pragprog.com/titles/jwdsal2/a-common-sense-guide-to-data-structures-and-algorithms-second-edition**