Algoritmen en datastructuren 3: verslag

Janne Cools: 14/12/2022

Array trie

Voor de array trie heb ik de patricia trie geïmplementeerd. Hierbij hou ik voor elke top enkele variabelen bij. De kinderen van de top worden bijgehouden in een lijst die in de heap wordt gealloceerd en het aantal kinderen wordt in een aparte variabele opgeslagen. Als de top een blad is, wordt de string in dat blad bijgehouden in een variabele. Zo niet, dan bevat deze variabele gewoon een null-waarde. Daarnaast wordt het karakter bijgehouden waarmee de top vertakt is vanuit zijn oudertop en wordt een skip bijgehouden. Deze variabele skip is een string die de karakters voorstelt die overgeslagen worden in de boom. Hiermee worden de paden van de boom gecomprimeerd.

Bij deze trie worden 3 variabelen gealloceerd in de heap, namelijk de kinderen van een top, de eventuele string en de skip. Deze worden allemaal terug vrijgegeven wanneer de top verwijderd wordt of wanneer de methode arraytrie_free wordt opgeroepen.

Ik heb zelf een extra methode toegevoegd om een nieuwe ArrayTrie aan te maken (new_arraytrie). Hierbij worden het karakter, de string, de lengte van de skip en de string van de skip meegegeven. De parameter string kan ook een null-waarde zijn. Deze methode wordt gebruikt in de methode om een string toe te voegen aan de boom (arraytrie_add).

Aangezien de basistesten niet voldoende zijn om de werking van de trie te controleren, heb ik zelf nog testen toegevoegd. Voor het toevoegen van strings heb ik vooral gecontroleerd dat de skips op een correcte manier gemaakt en aangepast worden.
Bij de testen voor het verwijderen van een string heb ik naar de verschillende mogelijkheden gekeken voor de ouder van het blad dat verwijderd wordt. De ouder kan immers 2 of meer kinderen hebben, waarbij beide gevallen anders behandeld moeten worden. Het geval van 2 kinderen heb ik nog gesplitst in de situatie waarbij beide kinderen bladeren zijn, of het ene kind een blad is en het andere een interne top. Ten slotte heb ik gecontroleerd dat de methode correct wordt uitgevoerd als de ouder de wortel is.

Ternary trie

Bij de ternary trie houd ik de grootte van de trie bij en de wortel. Deze wortel is een ander datatype dat ik zelf gemaakt heb en wordt gebruikt door de ternary trie en de custom trie. Hierover wordt later nog meer uitleg gegeven.

Om de toevoegingsbewerking van deze trie te testen, heb ik vooral nagekeken dat paden correct gevormd worden bij het toevoegen van een string die gelijkaardig is aan een string die al in de boom zit. Daarnaast heb ik voor het verwijderen gecontroleerd dat paden terug verkort worden als ze leiden naar slechts één blad. Het is immers inefficiënt om een lang pad te hebben naar 1 blad, aangezien je hierdoor meer toppen moet bezoeken om dat blad te vinden. Daarnaast heb ik nagekeken dat de boom foutloos hervormd wordt wanneer een interne top geen equals-kind meer heeft en dus verwijderd kan worden. Hierbij moet deze top vervangen worden door zijn linker- of rechterkind.

Custom trie

Aangezien er op voorhand geen alfabet vastgelegd wordt, worden de tries aangevuld naarmate er meer strings worden toegevoegd. Dit zorgt er voor dat je bij de array trie de kinderen niet op voorhand kan sorteren volgens het karakter en je dus geen vaste index hebt voor de kinderen. Hierdoor is het minder efficiënt om een vertakking te volgen bij deze trie, aangezien je de karakters van alle kinderen moet vergelijken met het karakter van de nieuwe string om te weten welk kind je moet volgen.
Op dat vlak is de ternary trie efficiënter, aangezien je in elke top slechts één vergelijking maakt en dan doorschuift naar een top met een groter, gelijk of kleiner karakter.

Aan de andere kant is het nadeel bij de ternary trie dat de boom niet gecomprimeerd is en er dus een lang pad met weinig vertakkingen kan ontstaan naar een blad. Dit kan de boom onnodig groot maken en zal het zoeken naar een string vertragen.

Daarom heb ik voor de custom trie een combinatie van de beide tries gebruikt. Ik heb namelijk de ternary trie geïmplementeerd waarbij de boom wel gecomprimeerde paden bevat. Deze trie houdt zoals bij de ternary trie de grootte van de trie bij en de wortel, die opnieuw het datatype BinaryNode is.

Om de implementatie van de custom trie te controleren, heb ik voor de toevoegbewerking testen gemaakt die gelijkaardig zijn aan de testen van de array trie. Ik voeg namelijk strings toe waardoor de variabele skip aangepast moet worden. Voor de testen voor het verwijderen heb ik mij gebaseerd op die van de ternary trie. Er wordt hierbij opnieuw nagekeken dat de trie correct hervormd wordt en dat de skip eventueel aangepast wordt.

BinaryNode

De BinaryNode is een datatype die een top voorstelt in de ternary trie en houdt variabelen bij voor het karakter van de top en de string als de top een blad is. Verder worden pointers naar de kinderen bijgehouden. Elke top heeft ten hoogste 3 kinderen naar toppen met een kleiner, groter of hetzelfde karakter als de huidige top.
Vervolgens bevat het datatype een variabele voor de ouder van de top. Aangezien de paden bij de ternary trie niet gecomprimeerd zijn, kan het dat veel toppen verwijderd moeten worden wanneer een string verwijderd wordt. Deze toppen worden dan op een bottom-up manier verwijderd, waardoor je de ouder van elke top nodig hebt en je deze dus moet bijhouden in een variabele.
Ten slotte wordt een skip bijgehouden, die enkel gebruikt wordt door de custom trie. Bij de ternary trie zal deze variabele gewoon de NULL-waarde hebben.

Door het gebruikt van dit extra datatype wordt code-duplicatie vermeden. Ik heb immers extra methodes gemaakt die door de beide tries gebruikt worden, zoals het vrijgeven van gealloceerd geheugen, het zoeken van een string in de trie en het hervormen van de trie door een top te vervangen door zijn linker- of rechterkind.

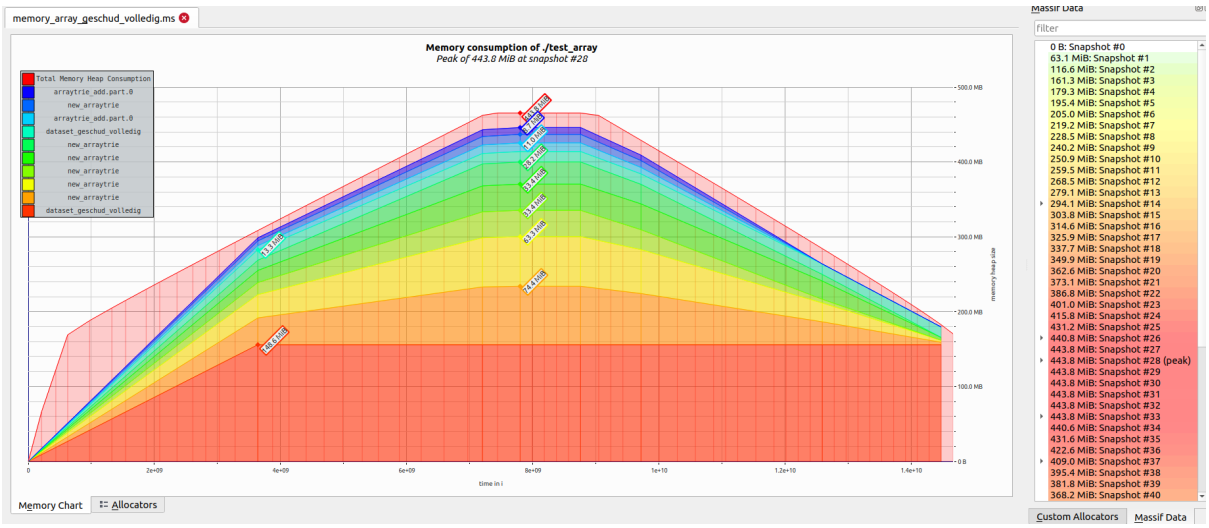
Geheugengebruik

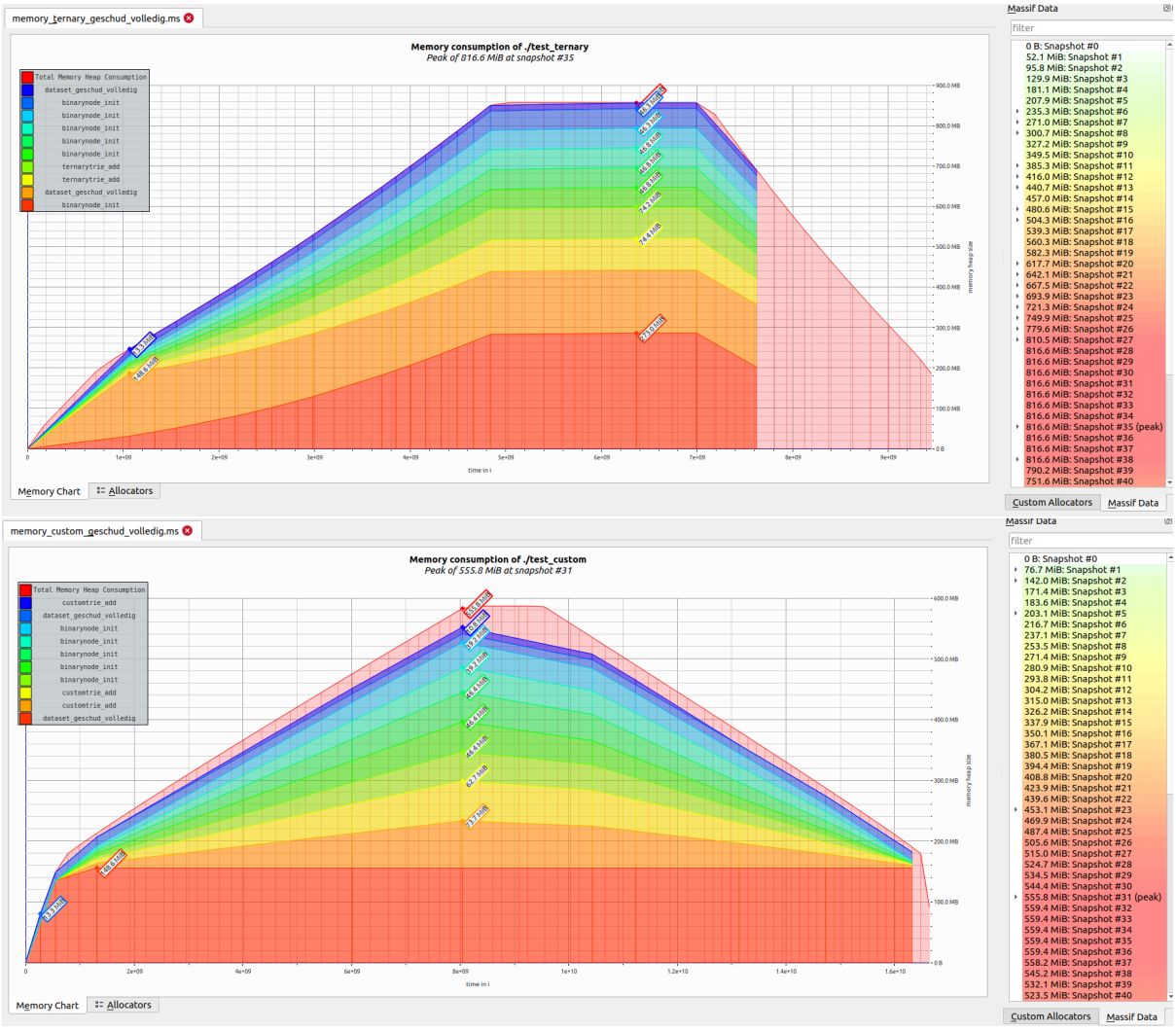
Zowel de array trie als de custom trie zijn gecomprimeerde bomen. Het overgeslagen deel van een string wordt in een top bijgehouden met een variabele "skip". Doordat deze niet gewoon de lengte van de deelstring bijhoudt maar de deelstring zelf, wordt er meer geheugen gebruikt.

Een variabele waarbij ik zo weinig mogelijk geheugen probeer te gebruiken, is de variabele "children"bij de array trie. Hierin worden de kinderen van een top bijgehouden in een lijst. Aangezien we het volledige alfabet niet op voorhand kennen, begin ik bij een lijst van grootte 0 en vergroot ik deze telkens met slechts één wanneer er een nieuw kind bijkomt. Op deze manier wordt er geen ongebruikt geheugen gealloceerd. Dit kan echter wel een effect hebben op de tijdscomplexiteit aangezien we de lijst elke keer moeten realloceren als er een nieuw kind bijkomt.

In de ternary trie worden de strings enkel in de bladeren bijgehouden en worden geen deelstrings opgeslagen in de interne toppen. Hierdoor wordt er minder geheugen gebruikt. Aan de andere kant zijn de paden van deze trie niet gecomprimeerd, waardoor er terug meer geheugen ingenomen wordt.

Bij het toevoegen, zoeken en verwijderen van een lijst van 1743797 strings werd het geheugengebruik gemeten en dit leidt tot de volgende figuren.





Hieruit kan afgeleid worden dat de ternary trie het meeste geheugen inneemt. De oorzaak is waarschijnlijk dat er meer toppen bijgehouden worden door de ongecomprimeerde paden.

De custom trie gebruikt een 100 megabyte meer geheugen dan de array trie. De array trie houdt zijn kinderen immers bij in een lijst, terwijl de custom trie 3 kinderen bevat per top. Hierdoor zal de custom trie meer toppen bijhouden, wat kan leiden tot meer geheugengebruik.

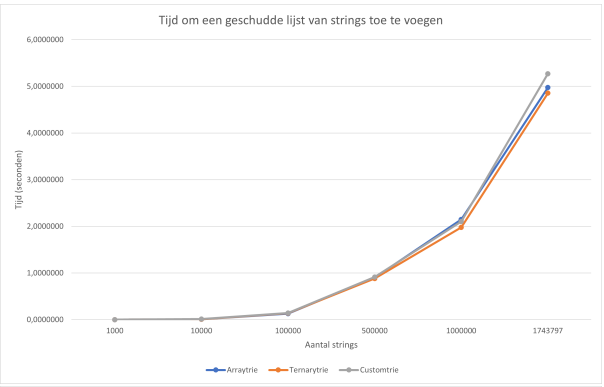
Tijdscomplexiteit

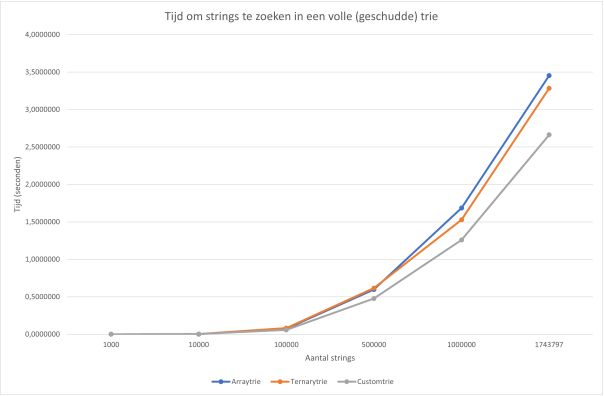
Om de tijdscomplexiteit te bepalen, heb ik de datasets gebruikt. Ik heb zowel geschudde deelsets gebruikt als ongeschudde deelsets. Hiermee kan ik vergelijken hoe de tries presteren met willekeurige strings ten opzichte van gelijkaardige strings. Voor beide situaties heb ik deelsets gebruikt met 1000, 10000, 100000, 500000, 1000000 en 1743797 strings.

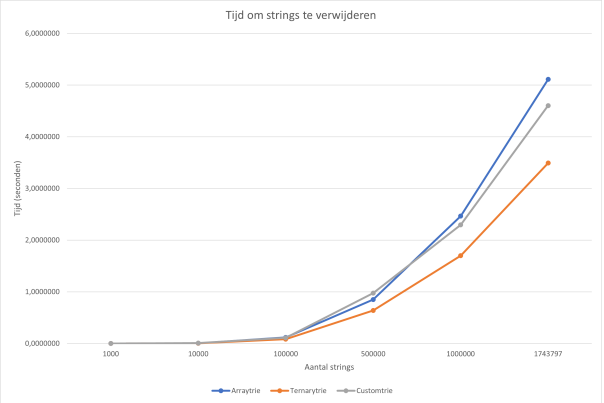
Geschudde datasets

Bij de geschudde datasets heb ik de tijden van het toevoegen, zoeken en verwijderen in

aparte grafieken geplaatst.



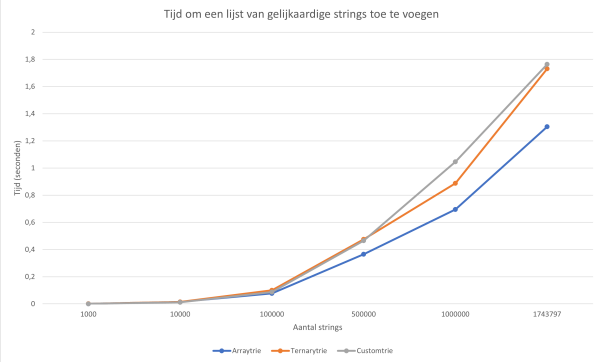


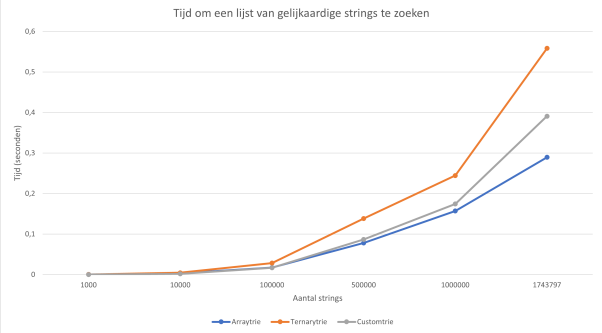


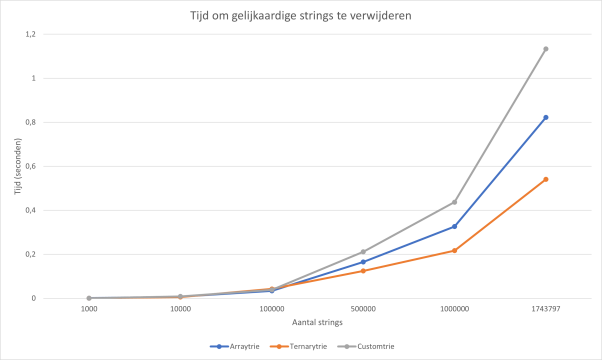
Uit de eerste grafiek kan afgeleid worden dat alle tries ongeveer even goed presteren voor het toevoegen van willekeurige strings.
Daarnaast presteert de custom trie het beste bij het zoeken van de strings en hebben de ternary trie en array trie een gelijkaardige tijd. Bij de array trie moet je immers bij elke top de lijst van kinderen overlopen om het kind te zoeken dat je moet volgen, wat een impact heeft op de uitvoeringstijd. Aangezien je bij de custom trie in een top enkel het karakter moet vergelijken en je op basis daarvan direct weet welk kind je moet volgen (links, rechts of midden), zal deze trie beter presteren. De custom trie is ook sneller dan de ternary trie door het werken met de skips. Hierdoor moet de custom trie immers minder toppen overlopen.
Aan de andere kant heeft de ternary trie de beste tijd voor het verwijderen van strings. Deze trie moet immers enkel toppen verwijderen en moet geen strings (voor de skip) aanpassen, wat een positieve invloed heeft op de uitvoeringstijd.

Gelijkaardige strings

Ook bij de gelijkaardige strings heb ik meerdere grafieken gebruikt voor de verschillende bewerkingen.







Hier zijn de resultaten anders en heeft de custom trie in het algemeen eerder de slechtste tijdscomplexiteit. Enkel bij het zoeken naar strings presteert deze trie beter dan de ternary trie. Het gebruik van de skips en gecomprimeerde paden van de array trie en de custom trie tonen bij zoekbewerkingen immers een significant voordeel. Door het werken met gelijkaardige strings kunnen geskipte deelstrings groot zijn waardoor er minder toppen gebruikt moeten worden en er dus minder toppen overlopen moeten worden bij het zoeken naar een string.
Aan de andere kant wordt het verwijderen van strings minder snel uitgevoerd bij de array trie en de custom trie dan bij de ternary trie. De oorzaak hiervan is waarschijnlijk dat je bij deze tries de skips in toppen opnieuw moet bepalen.
Bij het toevoegen van strings heeft de array trie de beste uitvoeringstijd. Bij deze trie moet je namelijk de karakters van de string met minder toppen vergelijken om te weten waar je het blad moet invoegen. Elke interne top bevat immers een ongeordende lijst van zijn kinderen waarbij je de pointer naar het nieuw kind gewoon achteraan toevoegt.

In het algemeen kan er vastgesteld worden dat alle tries bij gelijkaardige strings een kleinere uitvoeringstijd hebben dan bij de geschudde datasets.

Conclusie

Op basis van deze benchmarks kan afgeleid worden dat de array trie het best presteert wanneer de input bestaat uit gelijkaardige strings. Hierdoor zullen de skips groot zijn waardoor je minder toppen moet overlopen en vergelijken. Wanneer de bewerkingen uitgevoerd worden op willekeurige strings presteert de custom trie het best als de bewerkingen vooral uit zoekbewerkingen bestaan. Er worden hierbij immers minder toppen overlopen door het gebruik van gecomprimeerde paden. Aan de andere kant is de ternary trie de optimale trie voor verwijderbewerkingen, aangezien deze trie enkel toppen verwijdert en geen deelstrings moet kopiëren om de nieuwe skips te bepalen. Voor toevoegbewerkingen presteren alle tries even goed en is er dus geen specifieke optimale trie.