2022

Verantwoordingsdocument

COMPLEXE DATASTRUCTUREN

GROEP 7: EMANUEL DE JONG & PETER VAN ASSENBERGH

Inhoudsopgave

Requirements

Operatie tijden lineaire datastructuur en binaire zoekboom

De opdracht is om de tijden van het zoeken in een lineaire datastructuur en een binaire zoekboom te meten en vergelijken. Beide datastructuren moeten dezelfde gegevens opgeslagen hebben. En de metingen moeten in verschillende hoeveelheden gegevens plaatsvinden.

Als uitbereiding meten we ook het plaatsen en verwijderen van elementen.

We hebben voor de volgende omgeving gekozen:

Onderwerp	Keuze	Reden
Lineaire datastructuur	Java's eigen LinkedList	Om zeker te stellen dat de datastructuur goed en geïmplementeerd is.
Binaire zoekboom	Java's eigen TreeMap (Red black tree)	Om zeker te stellen dat de datastructuur goed en geïmplementeerd is. Hoe er rekening mee dat red black trees zichzelf balanceren. Dit kost tijd tijdens het balanceren maar scheelt tijd tijdens andere operaties.
Argument voor zoeken	De index/key	Als er met de opgeslagen waarde gezocht word, word er worst case altijd door de hele datastructuur gelopen. Dus is dit geen goede vergelijking.
Argument voor verwijderen	De index/key	Als er met de opgeslagen waarde gezocht word, word er worst case altijd door de hele datastructuur gelopen. Dus is dit geen goede vergelijking.
Welk element als argument	Elke 1 keer	De operatie tijd voor elementen aan het begin /bovenkant, midden of einde/onderkant kan veel uitmaken bij sommige datastructuren. Als er willekeurig word gekozen, kan het resultaat oneerlijk en onbetrouwbaar zijn. Door over alles te lopen, krijgt men het gemiddelde tussen de best case en worst case van een datastructuur.
Hoeveelheid herhalingen	3	Initialisatie en willekeurige gebeurtenissen kunnen een meting verpesten. Als 1 van de 3 tijdens erg afwijkt, is deze waarschijnlijk ongeldig. Een hoger aantal herhalingen zou nog beter zijn, maar dit zou ons en waarschijnlijk de docenten te veel tijd kosten.
Hoeveelheid gegevens	10.000, 50.000 en 100.000	Waardes onder 10.000 geven geen groot verschil aan. Waardes boven de 100.000 duren te lang bij het meten.
Gegevens type	WaitingListEntries	Het gaat om het verschil in de metingen tussen de datastructuren, en niet de metingen apart. Beide datastructuren maken even veel objecten aan, dus maakt het type object niet uit. We hebben alleen voor WaitingListEntries gekozen omdat we er verder niks mee hebben gedaan.

De vergelijking kan via de menu optie 4 gestart worden. De code is te vinden in de functies in controllers.Statistics.

Resultaat

Dit verschilt natuurlijk per computer en heeft altijd wat deviaties. Dit is enkel een van de metingen die we hebben gedaan. De metingen zijn in nanoseconden.

Linked list

Iteratie 1					
10.000					
Plaats	1.000.700	Zoek	31.028.500	Verwijder	138.125.400
50.000					
Plaats	3.002.900	Zoek	696.633.200	Verwijder	3.161.874.900
100.000					
Plaats	3.002.900	Zoek	2.783.532.200	Verwijder	12.647.500.200
Iteratie 2					
10.000					
Plaats	1.000.800	Zoek	28.025.100	Verwijder	129.117.800
50.000					
Plaats	1.000.700	Zoek	684.621.700	Verwijder	3.143.858.500
100.000					
Plaats	2.727.600	Zoek	2.808.554.400	Verwijder	12.652.503.900
Iteratie 3					
10.000					
Plaats	1.000.700	Zoek	27.024.500	Verwijder	126.113.500
50.000					
Plaats	1.002.500	Zoek	683.621.500	Verwijder	3.143.859.100
100.000					
Plaats	3.003.300	Zoek	2.723.476.400	Verwijder	12.563.423.800

Bekijk de linked list complexiteit voor meer informatie.

Red black tree

Iteratie 1					
10.000					
Plaats	4.003.500	Zoek	2.001.800	Verwijder	3.002.900
50.000					
Plaats	5.004.600	Zoek	2.001.300	Verwijder	5.004.700
100.000					
Plaats	10.009.200	Zoek	4.003.600	Verwijder	4.003.500
Iteratie 2					
10.000					
Plaats	1.001.100	Zoek	1.001.200	Verwijder	999.300
50.000					
Plaats	3.002.800	Zoek	3.002.400	Verwijder	2.001.700
100.000					
Plaats	6.005.800	Zoek	5.004.700	Verwijder	3.002.400
Iteratie 3					
10.000					
Plaats	1.001.100	Zoek	1.001.100	Verwijder	1.001.200
50.000					
Plaats	3.002.300	Zoek	2.001.800	Verwijder	1.001.100
100.000					
Plaats	8.007.100	Zoek	4.004.000	Verwijder	3.002.400

Bekijk de red black tree complexiteit voor meer informatie.

Sorteeralgoritme

In RouteCalculator calcRouteKruskal gebruiken we een int comparing comparator voor de PriorityQueue.

In ComplaintDB sortByTotalCost maken we een comparator om de complaints op totale kosten te sorteren. Dit kan in oplopende of aflopende volgorde.

In RouteCalculator calcRouteFromSimpleTreeLoop sorteren we zelf de SimpleTree kinderen. Dit algoritme is erg traag met een tijd complexiteit van O(n^2). Maar dit maakt in ons geval niet uit omdat onze boom maximaal 4 kinderen heeft.

Datastructuur met optimale uitkomst

De database (DB) klassen slaan hun gegevens op in LinkedHashMaps. De sleutels zijn hierbij de IDs van de opgeslagen klassen en de waardes zijn de klassen zelf.

Het HashMap gedeelde van dit datastructuur heeft het voordeel dat je klassen in constante tijd terug kan vinden met hun ID.

Wat deze HashMap speciaal maakt is dat de entries met een dubbele link lijst zijn verbonden. Hierdoor zijn de entries in een volgorde.

ComplaintDB heeft een methode die zijn complaints op totale kosten sorteert. Hiervoor is een LinkedHashMap perfect. De complaints kunnen gesorteerd worden en in de goede volgorde aan een nieuwe ComplaintDB instantie toegevoegd worden. Deze instantie heeft nog alle functionaliteiten en voordelen en heeft de gewenste volgorde als er doorheen gelopen word.

Lineaire datastructuren

De lineaire datastructuren die we hebben gebruikt vallen onder 2 categorieën:

Arrays

Deze slaan elementen (of pointers naar de elementen toe) achter elkaar in het geheugen op. Het is terug te vinden in constante tijd met het nummer van de positie in de reeks. Het adres is simpelweg: de bytes van het element * de positie + het adres van de array.

Het nadeel aan arrays is dat elementen achter elkaar moeten staan. Als er geen plek meer is, moet alles gekopieerd worden naar een nieuwe plek in het geheugen.

We gebruiken arrays overal in de code. Zo well basis arrays als ArrayLists.

Lijsten

Lijsten bestaan uit nodes met de waarde om op te slaan en een of meerdere referenties naar andere nodes.

De meest bekende lijsten hebben nodes een referentie naar de volgende node of twee referenties in beide richtingen.

In tegenstelling tot arrays, kunnen lijsten theoretisch oneindig nieuwe elementen toevoegen. Ook hoeft er niets te verschuiven als er elementen in het midden toegevoegd of verwijderd worden. De referentie(s) van de aansluitende node(s) hoeven alleen aangepast te worden.

Het nadeel aan lijsten is dat er bij het zoeken over de nodes heen gelopen moet worden tot het juiste element is gevonden. Verder nemen de referenties ook plaats op.

We gebruiken lijsten o.a. in de database klassen en bij het meten van operatie duur van een LinkedList in Statistics.

Avl boom

We hebben twee versies hiervan geïmplementeerd. Een map met sleutels en waardes, en een set met alleen waardes. Deze klassen en hun bijbehorende node klassen zijn te vinden in de models.tree package.

In de map versie word de sleutel gebruikt voor het sorteren. In de set versie de waarde zelf. Buiten dit zijn de versies identiek, dus zullen we verder uitleg geven alsof het om een enkele datastructuur gaat.

De waarde waarmee gesorteerd word moet de Comparable interface implementeren. Deze gebruiken we dan bij het vergelijken van twee nodes. Dit geeft de gebruiker de vrijheid zelf te beslissen hoe er gesorteerd word.

De boom is een form van een binaire boom. Dit betekend dat alle nodes links van een node lager en rechts van een node hoger moeten zijn.

Bij het plaatsen en verwijderen van nodes, word gekeken of de maximale hoogte van de linker en rechter kant niet meer verschillen dan een waarde van 1. Als dat wel zo is, worden er rotaties gedaan om de evenwicht te herstellen. Dit word zelf balanceren genoemd.

De map versie van deze datastructuur word in ons schema algoritme gebruikt. Maar zoals daar ook benoemd word, was dit geen goede keuze.

Graafstructuur

Een implementatie van een graaf en zijn subklassen is te vinden in de models.graph package.

GraphNode stelt een node voor en erft van de Location klas. Dit geeft het x en y waarden en functies om afstanden te berekenen.
GraphEdge stelt de verbinding tussen twee nodes voor en slaat de afstand tussen deze op.
Beide klassen hebben ook een uniek ID op basis van de node locaties.

De graaf slaat de nodes in een ArrayList op om makkelijk door de nodes heen te kunnen lopen. De verbindingen zitten in een HashMap met als keys de verbinding IDs. Dit is handig om snel een edge te kunnen vinden. Wij gebruiken het voornamelijk om in constante tijd te kijken of een edge al bestaat tijdens het toevoegen van een nieuwe edge.

Verder heeft de graaf 3 help methoden:

1. **getNodeConnections**: Voor een HashMap met per node een lijst van alle nodes waar het aan verbonden is.
2. **createEdges**: Maakt verbindingen tussen de nodes als er nog geen verbindingen zijn. Het resultaat is te vergelijken met een spinnenweb waarbij nodes verbonden zijn met de meest dichtbij zijnde nodes om hen heen. Dit is handig omdat we alleen gegevens hebben over locaties en niet over de verbindingen ertussen.
3. **calcTotalDistance**: Voor de afstand als we langs alle nodes gaan op volgorde van de node lijst. Het bruteforce algoritme gebruikt dit als meting van hoe goed een route is.

Graafalgoritmen

Er zijn 2 graafalgoritmen die we zelf hebben gemaakt.

Bruteforce

Als eerste is er het bruteforce algoritme om een route voor een schema te maken. Deze is te vinden in RouteCalculator calcRouteBruteforce.

Het probeert elke combinatie uit en houd altijd de beste route bij. Het geeft hiermee de optimaalste route maar levert een trage tijd complexiteit op.

De routes worden een voor een getest en worden weggegooid als ze langer zijn dan de beste route tot dan toe. Dit heet backtracking en zorgt ervoor dat niet alle routes in het geheugen opgeslagen hoeven worden.

Kruskal's algoritme

Als tweede is er een implementatie van Kruskal's algoritme. Deze is te vinden in RouteCalculator calcRouteKruskal.

In tegenstelling tot het bruteforce algoritme, is deze alleen een deel van de puzzel om een route te maken. Het word gebruikt om een minimum spanning tree (MST) van een graaf te maken. Simpel gezegd slaat het de connecties met de kortste afstand op met uitzondering van connecties die een cirkel creëren.

Het vind deze cirkels met onze DisjointSets (models.DisJointSet) en UnionFind (controllers.UnionFind) functies. Als de nodes van een connectie dezelfde set hebben, zouden ze een cirkel veroorzaken. Zo niet, worden de sets samengevoegd.

Voor een verdere optimalisatie hebben sets een rank. Bij het samenvoegen word met de rank beslist welke set in de ander gevoegd word.

Backtracking

Er zijn meerdere algoritmen met backtracking.

Het bruteforce algoritme is al in detail uitgelegd in Graafalgoritmen Bruteforce.

Schema algoritme

Het algoritme om schema's te maken (in Scheduler calcSchedule) word niet veel benoemd in dit document. Er zitten ontwerp fouten in die ervoor zorgen dat alleen naar een klein gedeelte van de mogelijkheden word gekeken. Toch willen we het hier benoemen omdat de backtracking erin op een slimme manier geïmplementeerd is.

Employees hebben een Schedule (schema) met een lijst van Complaints (klachten). Dit zijn de klachten die ze die dag moeten oplossen. Klachten hebben een geschatte tijdsduur. De bedoeling is om de totale geschatte tijdsduur van de klachten zo dicht bij de dagelijkse werktijd te krijgen als mogelijk.

De klachten zijn in het algoritme in een AvlTreeMap opgeslagen. De totale tijd word per lijn van de root naar een leaf node bijgehouden. Alleen als de totale tijd hoger is dan de tot dan toe beste tijd, worden de klachten van de lijn verzamelt en terug gestuurd. Dit is erg efficiënt met opslag.

Specificatie technieken

Regex

Bij menu optie 1 voor het toevoegen van werknemers, kan de gebruiker een naam invullen. Deze naam word gevalideerd met een regex patroon.

Deze kijkt naar hoofdletters, woord lengte, spaties en ongeldige karakters.

```
"(" + // Everything in a group for the last quantify
"[A-Z]" + // 1 uppercase letter
"[a-z]{1,50}" + // 1-50 lowercase letters
"[ ]?)" + // 0-1 space for the next word
"+"; // All this 1-many times
```

Asserties

Overal in de code zijn asserties te vinden. De uitspraken in deze asserties zouden bij het gebruik van het programma niet fout moeten zijn. Toch is het goed om ze te testen en zo snel fouten te kunnen ontdekken.

De meeste asserties controleren parameters op onverwachte null waarden of ongeldige nummers. Ook zijn er asserties die naar de middenstand of het eind resultaat van een functie kijken.

Complexiteit

Bij de opslag complexiteit gaat het om objecten die (tijdelijk) tijdens de operatie worden gemaakt.

Datastructuren

Java.util.LinkedList

N = hoeveelheid elementen

Operatie	Tijd	Reden	Opslag	Reden
Plaats	O(n)	Worst case word het element aan het einde geplaatst en word er door alle elementen gelopen. Maar bij het plaatsen is er geen verschuiving of reallocatie nodig.	O(1)	Er word geen kopie van de bestaande elementen gemaakt. Alleen het nieuwe element word opgeslagen.
Zoek	O(n)	Worst case word het element aan het einde gezocht en word er door alle elementen gelopen.	O(1)	Er word geen kopie van de bestaande elementen gemaakt.
Verwijder	O(n)	Worst case word het element aan het einde verwijderd en word er door alle elementen gelopen. Maar bij het verwijderen is er geen verschuiving nodig.	O(1)	Er word geen kopie van de bestaande elementen gemaakt.

Java.util.TreeMap (Red-Black tree)

N = hoeveelheid nodes

Operatie	Tijd	Reden	Opslag	Reden
Plaats	O(log n)	De plek om te plaatsen is de hoogte of minder. De boom is gebalanceerd dus de hoogte is log. Balanceren is ook O(log n) maar bij O(2log n) word 2 vergeten.	O(1)	Er word geen kopie van de bestaande nodes gemaakt. Alleen de nieuwe node word opgeslagen.
Zoek	O(log n)	De plek om te zoeken is de hoogte of minder. De boom is gebalanceerd dus de hoogte is log.	O(1)	Er word geen kopie van de bestaande nodes gemaakt.
Verwijder	O(log n)	De plek om te verwijderen is de hoogte of minder. De boom is gebalanceerd dus de hoogte is log. Balanceren is ook O(log n) maar bij O(2log n) word 2 vergeten.	O(1)	Er word geen kopie van de bestaande nodes gemaakt.

Models.graph.Graph

De GraphNodes zijn opgeslagen in een ArrayList, dus de tabel gaat eigenlijk daar over.
N = hoeveelheid elementen

Operatie	Tijd	Reden	Opslag	Reden
Plaats	O(n)	Normaal constant. Maar bij overschrijding van de array grote word de array opnieuw gealloceerd.	O(n)	Normaal alleen het nieuwe element. Maar bij overschrijding van de array grote worden alle elementen gekopieerd.
Zoek	O(1) / O(n)	Constant als de index word gegeven want het memory adres is te krijgen met een keer som. Lineair als het element word gegeven want worst case is het aan het einde, dus worden alle elementen vergeleken.	O(1)	Er word geen kopie van de bestaande elementen gemaakt.
Verwijder	O(n)	Worst case word het eerste element verwijderd en moeten alle elementen verschoven worden.	O(1)	Er word geen kopie van de bestaande elementen gemaakt. Alleen de indexen van de elementen worden aangepast.

Models.tree.SimpleTree

Een SimpleTree heeft alleen kinderen. In de tabel heb ik het over meerdere SimpleTrees die aan elkaar gekoppelt zijn. Denk aan een linked list die meerdere connecties kan hebben.
N = hoeveelheid nodes

Operatie	Tijd	Reden	Opslag	Reden
Plaats	O(n)	De boom word niet gebalanceerd, dus worst case is het een lange lijn en word de node aan het einde geplaatst.	O(1)	Er word geen kopie van de bestaande nodes gemaakt. Alleen de nieuwe node word opgeslagen.
Zoek	O(n)	De boom word niet gebalanceerd, dus worst case is het een lange lijn en is de node aan het einde.	O(1)	Er word geen kopie van de bestaande nodes gemaakt.
Verwijder	O(n)	De boom word niet gebalanceerd, dus worst case is het een lange lijn en is de node aan het einde.	O(1)	Er word geen kopie van de bestaande nodes gemaakt.

Algoritmen

MST met kruskal en union find

N = hoeveelheid edges

Operatie	Tijd	Reden	Opslag	Reden
Find	O(log n)	Disjoint sets kunnen als bomen gezien worden. Worst case word er vanaf een leaf node naar de root gezocht. Met het rank systeem is de hoogte gebalanceerd, dus is dit log.	O(1)	Er word geen kopie of nieuw object gemaakt.
Union	O(log n)	Bij het samenvoegen word parent1's parent naar parent2 gezet (of andersom), wat constant is. Het zoeken van de parents is echter 2 keer Find.	O(1)	Er word geen kopie of nieuw object gemaakt.
Totaal	O(n log n)	Er word minimaal een van de 2 operaties voor elke edge uitgevoerd.	O(n)	Alle edges zijn in een lijst opgeslagen. Er is ook een disjoint set voor elke edge.

Bruteforce route berekening

N = hoeveelheid nodes

Operatie	Tijd	Reden	Opslag	Reden
Totaal	O(n^n)	Elke combinatie van aparte nodes word geprobeerd.	O(n^2)	Door backtracking worden routes een voor een uitgeprobeerd en verwijderd. Maar elke laag worden alle nodes gekopieerd. Dus worst case zijn er n lagen die ieder n nodes hebben.

Datastructuur keuzes en alternatieven

Database klassen

Voor het opslaan van de gegevens word een LinkedHashMap gegeven. Onderbouwing van deze keuze kan hier gevonden worden.

Als alternatief kan een normale HashMap gebruikt worden en kunnen sorteer functies een andere datastructuur teruggeven. We kunnen ook het snelle zoeken opgeven en als opslag een sorteerbare lijst gebruiken.

PriorityQueue in Kruskal's algoritme

De edges worden naar afstand gesorteerd en opgeslagen in een PriorityQueue. Dit is echter ook goed te doen met een Collection zoals een ArrayList en de Collection.sort methode.

Graph route in bruteforce algoritme

Tijdens het algoritme word de Graph klas gebruikt om de route bij te houden. De nodes worden in de graph opgeslagen in een ArrayList. De volgorde van die ArrayList word na het algoritme gebruikt om een echte Route klas te maken. Dit doen we om de route afstand te berekenen met de graph calcTotalDistance functie.

Maar dit zou net zo goed gedaan kunnen worden door de functie staties te maken en de route in de Route klas of een ArrayList te zetten.

Location klas

We hebben onze eigen klas voor het opslaan van coördinaten gemaakt. Zo hebben we een plek om coördinaat berekening methoden in te plaatsen die dan ook makkelijk zonder controller klas gebruikt kunnen worden.

Als we de methoden in een andere klas hadden gezet, hadden we echter ook een ingebouwde klas zoals java.awt.geom.Point2D kunnen gebruiken.