2020-02-19

Carl Granström

Algoritmer och DATASTRUKTURER - Uppsala Universitet

Programmeringsuppgift 4

Analys av tidskomplexitet

# Inledning

Jag har valt att paketera mina mätningar i en egen klass som jag kallar Measurement. I measurement-klassen har jag fyra fält som motsvarar de fyra kolumnerna i cvs-filen med mätningar: date, time, temperature och approvalColor. De tre första är uppenbara, medan approvalColor är av *char*-typ och håller antingen ’G’ för Kontrollerade och godkända värden eller ’Y’ för misstänkta eller aggregerade värden enligt den bifogade filen *legend-smhi-opendata.txt*.

Measurement-klassen implementerar hashCode, equals och compareTo för att tillåta att den används tillsammans med datastrukturer på ett enkelt sätt.

## Datastrukturer

De datastrukturer som övervägts är några av de som ingår i Java Collections API: List, Map och Set. Dessa har dock olika implementationer med olika tidskomplexitet för sina operationer enligt nedan:

List

* LinkedList
* ArrayList

Map

* HashMap
* TreeMap (Red-Black Tree implementation)

Set

* HashSet
* TreeSet (Red-Black Tree implementation)

Eftersom de underliggande datastrukturerna är desamma för Map och Set så kan vi välja en av dessa. Då de metoder vi vill använda söker baserat på datum och tid så förefaller det logiskt att använda Map snarare än Set ur ren läsbarhetssynpunkt, dessutom är de flesta Set-implementationer inte ordnade vilket förefaller osmidigt då vår data är i ordning från början samt behöver använda sig av den ordningen för de flesta operationerna. Detta leder till att jag beslutade mig för att närmre granska och överväga LinkedList, ArrayList, HashMap och TreeMap.

Andra strukturer så som till exempel Stack och Queue gör själva implementationen av vissa operationer väldigt intuitiv, men har samma tidskomplexitet som de underliggande datastrukturerna, i fallet med Stack är detta Vector, vilket är en Dynamisk Array som har samma tidskomplexitet som ArrayList(amortiserad analys för put()-operationen).

## Operationer

För att avgöra vilken datastruktur som är lämplig så måste de operationer som ska användas granskas så att dessa kan jämföras med datastrukturernas tidskomplexitet för operationerna i fråga.

### loadData()

Jag har valt att inte räkna in tiden det tar att föra över data till datastrukturen i mitt övervägande om vilken struktur som är lämpligast. Ibland är det självfallet så att själva överförandet av data till lämplig struktur dominerar körtiden hos en algoritm, vilket man som utvecklare absolut behöver vara medveten om. Att den aktuella datan redan är sorterad ger ju även vissa datasstrukturer en fördel.

### averageTemperatures()

Averagetemperatures innehåller flera operationer som är relevanta att ta i beaktan.

1. Hitta första värdet med rätt startdatum
2. Medan datumet <= slutdatum
   1. Repetera 24 ggr: addera temperaturen till temperaturSumma
   2. medeltemperatur = temperaturSumma / 24
   3. Skriv ut datum och medeltemperatur

Observera att listan nedan ignorerar operationer som är O(1) för samtliga strukturer eller som på andra sätt är irrelevant för analysen. Listan är därför något förenklad.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Steg | **LinkedList** | **ArrayList** | **HashMap** | **TreeMap** |
| **Hitta startdatum** | indexOf(Object)  get(int index)  O(n + n) = **O(n)** | indexOf(Object)  get(int index)  O(n + 1) = **O(n)** | get(key)  **O(1)** | get(key)  **O(log(n))** |
| **addera temperaturen till slutsumma och gå till nästa tid (24 ggr)** | next()  **O(1)** | get(int index)  **O(1)** | get(key)  **O(1)** | get(key)  **O(log(n))** |

*Tabell 1: Tidskomplexitet för averageTemperatures*

Det förefaller som att det borde gå att både hitta objektet och returnera det utan att iterera över listan två gånger för LinkedList, men det är inte omedelbart uppenbart hur.

För TreeMap så kan vi välja att antingen använda get 24 gånger för O(24 log(n)) eller att bygga en iterator för O(n) och sedan använda next() vilken ger O(1). Att konstruera iteratorn tar då så pass lång tid att det inte är att föredra.

En sista observation är att alla de aktuella värdena behöver adderas ihop (det vill säga antalet dagar\*24) om vi kallar detta värde för *m* (observera dock att m <= n eller m ⊆ n)så får vi en tidskomplexitet som måste gångras med m. Men då worst case är att m = n om vi vill beräkna för hela tidsperioden så kan vi använda n istället för m.

Detta ger alla datastrukturerna en identisk tidskomplexitet på O(n).

### missingValues()

1. Hitta första värdet med rätt startdatum
2. Medan datumet <= slutdatum
   1. Repetera 24 ggr: kontrollera om tiden finns registrerad
   2. returnera saknade tider

Observera att då vår data redan är sorterad så tar det ingen extra tid för våra listor att returnera tiderna för datumen i ordning och för HashMap och TreeMap handlar det också bara om att inkrementera nycklarna med en timme 23 gånger totalt efter att första tiden för datumet hittats.

Att Map-strukturerna hittar saknade värden med hjälp av att get(key) returnerar *null* spelar ingen roll för tidskomplexitetsanalysen då detta inte ger sämre tidskomplexitet än hittade nycklar.

ListImplementationerna hittar saknade nycklar genom att jämföra den förväntade tiden med den som returneras av next() eller get(index)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Steg | **LinkedList** | **ArrayList** | **HashMap** | **TreeMap** |
| **Hitta startdatum** | indexOf(Object)  get(int index)  O(n + n) = **O(n)** | indexOf(Object)  get(int index)  O(n + 1) = **O(n)** | get(key)  **O(1)** | get(key)  **O(log(n))** |
| **kontrollera om tiden finns** | next()  **O(1)** | get(int index)  **O(1)** | get(key)  **O(1)** | get(key)  **O(log(n))** |

*Tabell 2: Tidskomplexitet för missingValues*

Även här måste vi dock iterera över alla värden i den angivna perioden, och får ett worst case med m = n, vilket ger O(n) för samtliga datastrukturer.

### approvedValues()

Den här operationen är lite extra intressant då den på ytan först kan förefalla vara O(n) då vi alltid måste kontrollera alla mätningar och deras värde på det här fältet. Om vi kan sortera datan på detta fält från början kan vi dock hitta ”brytpunkten” mellan ’G’ och ’Y’ i O(log(n)) och utifrån denna brytpunkt räkna ut hur många värden som är ’G’ respektive ’Y’.

I Tabell 3 nedan hittar du min skiss som försöker förklara hur jag tänker mig att missingValues kan få komplexiteten O(log(n)) i en sorterad lista som backas av en array eller i ett binärt träd.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| G | G | G | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | start |  |
| G | G | G | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | 16 | eftersom högra sidan börjar med Y, släng den |
| G | G | G | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | 8 | eftersom högra sidan fortfarande börjar med Y, släng den |
| G | G | G | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | 4 | eftersom högra sidan börjar med G, släng vänster sida och öka countG med 2 |
| G | G | G | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y |  | 2 | eftersom vänster sida innehåller G, öka countG med 1 (countG = 3), eftersom höger sidan börjar med Y, släng den. |
| G | G | G | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | 1 | countG = 3 |

*Tabell 3: Divide-and-Conquer med 16 mätvärden, 3 godkända och 21 icke godkända*

Utan att gräva mig alltför djupt ner i detaljer och edge cases så verkar det som att det skulle fungera att göra denna beräkning med Divide-and-Conquer och få O(log(n)) om listan/trädet är sorterat. Lägg märke till att illustrationen ovan räknar godkända mätningar snarare än icke godkända, men det spelar inte så stor roll för själva principen.

Dock går det inte att sortera listan efter dessa värden snabbare än O(n log(n)), så ovanstående operation får ändå en tidskomplexitet på O(n log(n)).

## Diskussion

Det förefallet som att vi för detta problem har två valmöjligheter. Den första lösningen är att vi skapar en HashMap för att njuta av något snabbare exekveringstid för averageTemperatures i de fall som vi inte är intresserade av hela perioden. Den andra lösningen är att redan från början skapa ett sorterat träd, vilket ger oss en obetydligt långsammare(skillnaden mellan O(1) och O(log(n)) är faktiskt väldigt liten) exekvering av vissa delar av averageTemperatures och missingValues men ger oss en möjlighet att, i de fall vi inte behöver bry oss om tiden det tar att befolka vår datastruktur, skapa ett träd sorterat på approvalColor och sedan kunna köra approvedValues i O(log(n)).

Min implementerade lösning håller sig till det förstnämnda alternativet eftersom jag uppskattar att averageTemperature är en vanligare operation än approvedValues.