Sorting & Searching, practicum 1

Efficiëntie van geavanceerde sorteeralgoritmes

Deze opdracht bouwt voort op opdracht 2 van het vak DataStructures. Als je die opdracht gemaakt hebt, dan kun je het deel dat de aantallen studenten aanpast en de tijd meet hergebruiken.

Je moet het volgende doen:

a) aan het programma een advanced sort algoritme (Merge Sort of Quick Sort) toevoegen en de big-O* bepalen
b) verbeteringen aan de gekozen advanced sort algoritme toevoegen en de big-O* bepalen (zie slide 21 en 31)
c) de lijst opslaan in een Binary Search Tree (hier hoef je dus géén big-O te bepalen).

* bij het bepalen van de big-O moet je de tijd meten die nodig is om een steeds oplopend aantal studenten te sorteren en onderzoeken of er een verband te ontdekken valt, en zo ja welk verband, met het aantal elementen.

a. Resultaten van studenten sorteren met een advanced sort

Je dient een programma te schrijven waarmee je efficiënt een lijst met studenten kunt laten zien, gesorteerd op bepaalde criteria. We gaan uit van een lijst met resultaten van een tentamen van een groep studenten, waarvan het studentnummer en het cijfer bekend is.

Hergebruik opdracht 2 van Datastructures (optioneel)

Als je opdracht 2 van Datastructures hebt gemaakt kun je het genereren van de studentenlijst met studentnummers en cijfers hergebruiken. Lees dan verder bij "Advanced sort toevoegen". Heb je de opdracht (nog) niet gemaakt, doe dan het volgende:

Om het programma te testen moet je zelf een lijst met studenten genereren. Voor deze opdracht heb je alleen een lijst van studenten met studentnummers en cijfers nodig. Je dient dus een class `Student` te maken die twee attributen heeft, studentnummer en cijfer.

Het programma moet een aantal studenten aan kunnen maken, standaard is 10.000, maar je moet dit gemakkelijk kunnen wijzigen in andere aantallen.

Bij het aanmaken krijgt elke student een studentnummer opeenvolgend te beginnen bij 50060001. Het programma geeft elke student random een cijfer met 1 decimaal (1,0 1,1 1,2 … 9,9 10,0). LET OP: elk cijfer heeft dezelfde kans om gekozen te worden.

Na het aanmaken moet je de lijst 'schudden', zodat de studentnummers niet meer op volgorde staan. Je kunt daarvoor de methode `schud(T[] waardes)` gebruiken uit de klasse `Schudder` die in het project zit dat je op de VLO staat.

Advanced sort toevoegen (Merge Sort of Quick Sort)

De aangemaakte lijst van studenten moet gesorteerd kunnen worden en het programma moet de (gesorteerde) lijst kunnen printen.

Je moet de gehele lijst met alle studenten kunnen sorteren op **cijfer van laag naar hoog**. Bij gelijke cijfers sorteer je op **studentnummer van hoog naar laag**.

Implementeer hiervoor naar keuze Merge Sort of Quick Sort. Maak gebruik van Comparable of een Comparator. Je moet de code van de sorteermethode van je keuze zelf implementeren (aan de hand van het boek) en toelichten in je verslag.

Voeg relevante code snippets toe in je verslag. Laat ook zien dat het sorteeralgoritme correct werkt door de output te tonen van een kleinere lijst studenten (bijvoorbeeld 50 studenten) voor en na sorteren.

Efficiëntie meten

Onderzoek de efficiëntie van het sorteeralgoritme door een aantal experimenten uit te voeren[1]. Genereer (in een loop) verschillende lijsten van 10.000, 20.000, 40.000, 80.000 en 160.000 studenten met hun resultaten. Sorteer vervolgens deze lijst op de gewenste manier.
Presenteer de resultaten van je experiment in een tabel en een grafiek.

Bereken op basis van je tests de tijdefficiëntie* van je implementatie en maak daarbij gebruik van de big-O. Laat zien hoe je de big-O hebt berekend. Vergelijk ook je praktische resultaten met de theoretische big-O van de algoritmes. Als er verschillen zijn, beredeneer dan hoe dat komt.

b. Verbetering toevoegen aan je gekozen algoritme

Merge Sort

Als je Merge Sort hebt gekozen, implementeer dan minstens één van de drie verbeteringen die op pagina 275 van het boek worden genoemd.

1) Voeg een cutoff toe voor kleine subarrays
2) Test of het array al gesorteerd is
3) Vermijd het kopiëren door de argumenten in de recursieve code te wisselen

Quick Sort

Als je Quick Sort hebt gekozen, implementeer dan de verbetering “Median-of-3 partitioning” die op pagina 296 van het boek staat.

Efficiëntie

Voeg relevante code snippets van je verbetering toe in je verslag.
Doe de big-O experimenten op dezelfde wijze als in onder deel a. en geef alle output. Vergelijk de resultaten met de resultaten bij a.

---

[1] Om de meetwaarden betrouwbaar te maken is het aan te raden om ieder experiment een aantal keer te doen en dan het gemiddelde te nemen.

c. Sla resultaten op in een Binary Search Tree en implementeer rank()

NB:Aan het begin van het practicum van week 2 wordt de Binary Search Tree (BST) behandeld. Je zult dus nog even moeten wachten met het maken met dit deel van deze opdracht.

Implementeer een Binary Search Tree (BST) die voldoet aan de API op bladzijde 398 van Sedgewick, zodat je de studentenlijst daarin kunt wegschrijven. Bij het toevoegen van een student aan de BST moet je het cijfer gebruiken om de plaats in de BST te bepalen, het studentnummer is de value. Onderzoek hoe je om moet gaan met het feit dat een cijfer meer dan één keer voor kan komen. Voeg de studenten uit de lijst toe nadat je de lijst geshuffeld hebt, zodat je niet een BST maakt, die totaal ongebalanceerd is.

Bij het zoeken op een cijfer moet je alle studentnummers geven van de studenten die dat cijfer gehaald hebben. **Toon aan dat dit werkt met een voorbeeld. Toon ook de code snippet van de door jou aangepaste methode `get()`.**

Implementeer de methode `rank()` en laat zien dat je bij een bepaald cijfer het aantal studenten kunt geven dat een lager cijfer heeft gehaald.

Voorbeeld: Stel dat de cijfers 10, 9, 9, 8, 8, 8, 7, 7, 6, 6, 6, 6, 6, 5, 3, 2 behaald zijn. De methode rank() geeft dan de volgende resultaten:

Cijfer	Rank
1	0
2	0
3	1
4	2
5	2
6	3
7	8
8	10
9	13
10	15