UNIVERSITEIT GENT

Optimale binaire zoekbomen
Tussentijdse – Feedback

Project Algoritmen en Datastructuren II

Academiejaar 2013-2014

1 Inleiding

Dit document bevat een lijst met de meest voorkomende fouten in de implementatie die werd ingediend voor de tussentijdse deadline. Aangezien de meeste fouten werden gemaakt tegen de returnwaarde van de `add` en `contains` methoden, staat er onderaan dit document een voorbeeld waarbij telkens de correcte waarde wordt teruggegeven.

Op vraag van enkele studenten werd ook een nieuwe versie van de klasse `NumberReader` ter beschikking gesteld. Je kan nu de methode `readFile()` oproepen om het hele bestand in te lezen. Deze methode is eenvoudiger en sneller dan het herhaaldelijk oproepen van de methode `next()`.

Belangrijk: de semi-splay bomen moeten de interface `BST` implementeren en dus ook alle methoden `cost()`, `balance()`, `size()`, ... Enkel de methode `optimize()` moet niet worden geïmplementeerd voor semi-splay bomen. Je kan voor sommige methoden natuurlijk overerving gebruiken om codeduplicatie te vermijden.

2 Fouten bij het tussentijds indienen

- De waarden teruggegeven door de methoden `add` en `contains` waren foutief. Een voorbeeld van de correcte returnwaarden is terug te vinden in dit document.
- De code stond in het verkeerde package. Ook de meegegeven interface werd aangepast door een andere package naam in te voeren.
- De klasse `Obst1` bevatte wel alle methoden, maar implementeerde niet de interface `BST` (ook niet onrechtstreeks).
- De code genereerde `Nullpointer` fouten op het voorbeeld beschreven onderaan dit document. Ook andere fouten (`ArrayIndexOutOfBoundsException`, `IllegalAccessException`) werden opgegooid bij het uitvoeren van onderstaand voorbeeld.
- De implementatie print soms nog debuginformatie uit. Dit kan best worden uitgeschakeld in de uiteindelijke versie.
- De `size()` methode was nog niet geïmplementeerd.

3 Correcte uitvoer van de te implementeren methoden

Hieronder staat een klein voorbeeld waarbij een reeks van methode-oproepen wordt uitgevoerd op een initieel lege boom. De correcte uitvoer voor de methoden `contains`, `add`, `cost` en `size` wordt telkens gegeven. Je kan dit voorbeeld gebruiken om te controleren of jouw implementaties de correcte waarden teruggeven. Aan de hand van het voorbeeld zou ook moeten duidelijk zijn hoe de uitvoer moet worden berekend in alle mogelijke situaties.

index	methode	uitvoer	uitleg
0	`size()`	0	De grootte van een lege boom is 0.
1	`contains(0)`	0	De navigatiekost in een lege boom is 0 (je navigeert niet of voert geen vergelijkingen uit). Het element wordt niet gevonden, dus is de returnwaarde niet positief.
2	`add(3)`	0	Toevoegen aan een lege boom vraagt geen navigatiekost. De returnwaarde is dus 0. De nieuwe wortel heeft gewicht 1.
3	`size()`	1	Een boom met 1 element heeft grootte 1.
4	`cost()`	1	De plaatsing van de wortel in stap 2 zorgt voor een gewicht 1 in de wortel. De totale kost is dus 1.
5	`contains(3)`	1	Het element in de wortel wordt opgezocht. Dit heeft navigatiekost 1 (aantal toppen die je bezoekt en waar je vergelijkingen uitvoert, diepte van de top +1).
6	`contains(1)`	-1	De navigatiekost is 1, doordat je in de wortel kan beslissen dat het element niet in de boom zit. De returnwaarde is negatief aangezien het element niet in de boom zit. De gewichten van de elementen worden niet gewijzigd.
7	`cost()`	2	De wortel heeft gewicht 2 (1 van de `add` methode in stap 2 en kost 1 van het opzoeken in stap 5).
8	`add(3)`	-1	De navigatiekost is 1, maar omdat het element al in de boom zat, wordt vermenigvuldigd met -1.
9	`cost()`	3	De wortel heeft kost 3 door bewerkingen in stappen 2, 5 en 8. Toevoegen is ook een zoekbewerking, waardoor in stap 8 ook het gewicht van element 3 met één wordt verhoogd.
10	`add(5)`	1	De navigatiekost is 1 omdat je in de wortel al kan beslissen waar de top moet worden toegevoegd. De top met element 5 krijgt meteen gewicht 1.

index	methode	uitvoer	uitleg
11	`add(6)`	2	De navigatiekost is 2. Je moet 2 toppen doorlopen (wortel en top met sleutel 5) om de top met sleutel 6 toe te voegen. Dit is ook gelijk aan de diepte van sleutel 5 plus 1, of gelijk aan de diepte van de nieuwe top.
12	`cost()`	8	De wortel heeft kost 3, sleutel 5 heeft diepte 1 en kost 1, sleutel 6 heeft diepte 2 en kost 1. Samen geeft dit $3 \cdot 1 + 1 \cdot 2 + 1 \cdot 3 = 8$.
13	`add(6)`	-3	Je moet 3 toppen bezoeken om te beslissen dat 6 reeds in de boom zat. Het gewicht van sleutel 6 wordt met 1 verhoogd.
14	`contains(6)`	3	Je bezoekt 3 toppen en vindt sleutel 6 op diepte 2.
15	`contains(7)`	-3	Je bezoekt 3 toppen en vindt dat het rechterkind van de top met sleutel 6 leeg is. De returnwaarde wordt dus met -1 vermenigvuldigd.
15	`add(5)`	-2	Analoog aan stap 13.
16	`add(1)`	1	Analoog aan stap 10.
17	`add(0)`	2	Analoog aan stap 11.
18	`contains(3)`	1	Analoog aan stap 5.
19	`contains(4)`	-2	Je bezoekt de wortel en zijn rechter kind met sleutel 5 (2 toppen) en vindt dat het linker kind van de top met sleutel 5 leeg is. De navigatiekost 2 wordt dus met -1 vermenigvuldigd.
20	`contains(2)`	-2	Analoog aan stap 19, maar nu bezoek je de wortel en zijn linker kind.
21	`contains(5)`	2	Navigatiekost is 2.
22	`add(2)`	2	Navigatiekost is 2.
23	`add(4)`	2	Navigatiekost is 2.
24	`size()`	7	De boom bevat sleutels 0 tot en met 6.
25	`cost()`	30	Na stap 12 (kost 8) werd sleutel 3 één keer opgezocht, sleutels 5 en 6 tweemaal (1 keer via `add` en 1 keer via `contains`). Er werden ook 4 extra sleutels toegevoegd. Samen geeft dit een kost van $4 \cdot 1 + (1+3) \cdot 2 + (1+1+1+3) \cdot 3 = 30$.