Laborationsrapport

Algoritmer och datastrukturer I , 7,5 hp
HT 2015

Laboration nr: 3
Prioritetsköer

av

Viktor Hanstorp (1994-04-13)

Institutionen för matematik, natur- och datavetenskap
Högskolan i Gävle

S-801 76 Gävle, Sweden

Email:
ndi14vhp@student.hig.se

Innehåll

1 Inledning

Labben ger erfarenhet i att arbeta med den abstrakta datatypen prioritetskö. Insikten som skall uppnås är hur bra prioritetsköer med ”Heap” och ”BST” är på olika sorterade indata. Detta uppnås med hjälp av empirisk analys.

2 Förutsättningar och krav

Reflekterande över empirisk data, skapande av ett prioritetskö-interface samt implementering av interfacet.

3 Implementering och test

För att göra empiriska tester så skapades klassen ”Benchmarker”. Klassen är till för att lättare kunna mäta exekveringstid.

”Benchmarker” tar emot en ”Logger” som optional parameter, och skriver ut till ”System.out” om man inte anger något annat.

För att skapa en benchmarker så kan man göra:

```
Benchmarker benchmarker = new Benchmarker()
{
    @Override
    void run() throws Exception
    {
          // Kod att köra
    }
};
```

Sedan kan man anropa ” benckmarker.benchmark([times])” där [times] är antalet gånger att köra ”run()”. Det som retuneras är ”BenchmarkResult” som inehåller tidsåtgången för alla körningar, samt metoder för att hämta ut ”max”, ”min” och medeltid.

Det finns också metoder som hjälper till:

```
public static BenchmarkResult runBenchmark([method], [times])
```

Där [method] är en lambda metod av typen

```
public interface BenchmarkMethod
{
    void run();
}
```

Och [times] är antalet gånger som den skall köras (”BenchmarkResult” är samma som ovan)

I ”benchmarker”:s ”run” metod så går det även att anropa ”log” och ”runUnbenchmarked” vilket stoppar tidsmätningen och skriver ut till ”loggern” eller kör en metod innan den fortsätter. All tid går inte att ta hänsyn till, det vill säga att om man anroppar ”runUnbenchmarked” så kommer det att gå tid innan tiden stoppas (anropande av metoden), dock så är denna tid konstant, och bör kunna bortses ifrån vid jämförelser.

Uppgift ”b” valdes (Binärt sökträd). Då användandet av binärt sökträd är ineffektivt (se Tabell 1 - Tidsåtgång) så valdes det att åtminstone implementera AVL[1].

3.1 Systembeskrivning

Trädet är uppbyggt av noder
- se.hig.aod.lab3.MyBSTPriorityQueue.MyBSTPriorityQueueNode

som är länkade med handtag
- se.hig.aod.lab3.MyBSTPriorityQueue.MyBSTPriorityQueueNodeHolder

till sina barn.
Med hjälp av dessa handtag så är det möjligt att traversera trädet åt båda hållen. Samt så gör dem det lättare att rotera noder.

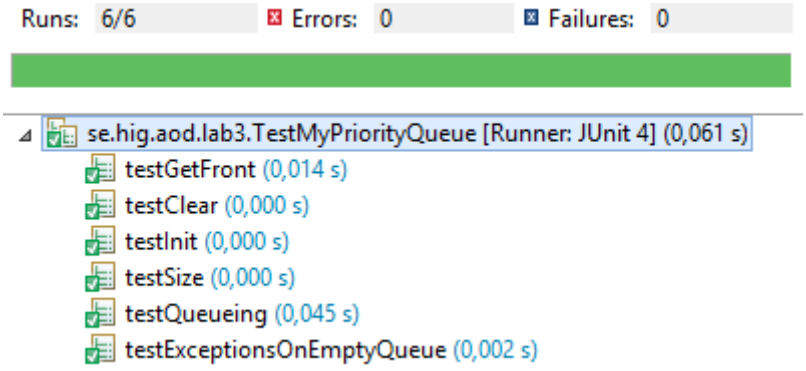
Ett huvud-exception
- se.hig.aod.lab3.MyPriorityQueue.MyPriorityQueueException

finns för kön. Exceptionet ärvs sedan av mer specifika exceptions
- se.hig.aod.lab3.MyPriorityQueue.
  - MyPriorityQueueIsFullException
  - MyPriorityQueueIsEmptyException
  - MyPriorityQueueNullNotAllowedException

Detta enligt rekommendation från lärare.

3.2 Tester

Metoder testades med JUnit



[1] F09 - AVL-träd (PDF) – Blackboard, Otillgängligt 2015-12-10

4 Diskussion

4.1 Diskussion av resultatet

	Heap						BST					
	Osort		Sort		Omv		Osort		Sort		Omv	
	In	Ut	In	Ut	In	Ut	In	Ut	In	Ut	In	Ut
10000	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**266**	**0**	**281**	**172**
20000	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**500**	**0**	**531**	**422**
40000	**0**	**15**	**0**	**0**	**0**	**0**	**0**	**0**	**969**	**0**	**1031**	**922**
80000	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1875**	**0**	**2125**	**1968**
160000	**15**	**15**	**16**	**0**	**0**	**0**	**0**	**0**	**4141**	**0**	**3938**	**4078**
320000	**15**	**0**	**16**	**16**	**32**	**0**	**0**	**0**	**7933**	**0**	**8282**	**8469**
640000	**16**	**0**	**31**	**0**	**31**	**0**	**16**	**0**	**22796**	**0**	**23846**	**18064**

Tabell 1 - Tidsåtgång

Resultaten av benchmarken kan ses i Tabell 1. Man kan se att en ”Heap” baserad prioritetskö har en försumbar tidsåtgång för alla testfall.

Binärt-sökträd har försumbar tidsåtgång för osorterat data, detta är då trädet är relativt balanserat och ger då en tidsåtgång på O(log(n)).

Insättningen på sorterad data tar dock längre tid då trädet urartar (fyller bara till vänster eller bara till höger) och då blir O(n).

Den försumbara tidsåtgången för borttagning ut binärt sökträd med sorterad indata beror på att då de minsta värdena stoppas in först så urartar trädet åt vänster, sedan när man tar ut värden så tar man bara roten, vilket går på O(1).

Den prioritetskö som skapades är väldigt ineffektiv jämfört med de som vi testade (antagligen på grund av ooptimerad AVL bearbetning), dock så var det lite ont om tid för att kunna hinna med optimeringar.

Prioritetskön hanterar inte dubbletter dock borde den se till att de är FIFO.

4.2 Diskussion av genomförandet

Började tidigt med benchmarkingen och fick det färdigt i god tid, dock så dröjde kanske kodandet och rapporten lite väl lång tid (på grund av annan inlämning samt distaraktioner).

5 Referenser

6 Bilagor

Källkoden finns tillgänglig i bifogad folder.