Algoritmer och datastrukturer
version perjee 26 november 2015

Laboration 3: Prioritetsköer

Syfte

Laborationen handlar om att bli bekant med den abstrakta datatypen prioritetskö både från programmeringens och analysens synvinkel. Därför omfattar den uppgifter som ger färdigheter i att utveckla ett programmerings-gränssnitt till datatypen och implementera gränsnittet i form av en Java-klass. Under laborationens gång ska två olika implementeringar av dataypen undersökas experimentellt. Syfte är att visa tidsåtgången när datamängder med olika storlekar bearbetas samt diskutera resultaten för att utveckla kunskaper om hur valet av datastrukturen kan influera prestandan av implementationen.

Förberedelse

Läs de relevanta avsnitten i kurslitteraturen (se litteraturhandledningen) och föreläsningsanteckningarna till föreläsning 8.

Uppgifter

1. Undersök tidsåtgången för insättning och borttagning av ett antal element i en redan existerande prioritetskö. Jämför tidsåtgången som uppstår när en HEAP-baserad kö används med den av en kö som är baserad på ett binärt sökträd.

2. Skapa ett Java-interface för den abstrakta datatypen Prioritetskö. Prioritetskön ska kunna lagra instanser av klasser som implementerar interfacet `Comparable`. Interfacet ska innehålla följande operationer:

   - Töm prioritetskön (töm den på element och återlämna minnet) - `clear`.
   - Undersök om prioritetskön är tom - `isEmpty`.
   - Undersök om prioritetskön är full - `isFull`.
   - Returnera antalet element i prioritetskön - `size`.
   - Sätt in ett element i prioritetskön (i prioritetsordning) - `enqueue`.
   - Ta bort första elementet (det element som har högst prioritet) från prioritetskön och returnera det till användaren - `dequeue`.
   - Titta på första elementet (det element som har högst prioritet) utan att ta bort det - `getFront`.

3. Skapa ett testprogram (rekommendation: använd `JUnit`) som kan användas för att testa de olika operationer som kan utföras på prioritetskön. Låt programmet sätta in element av typen `Comparable` (till exempel `Integer`-objekt) i prioritetskön.

4. Implementera en prioritetskö antingen

   (a) baserad på en heap eller

   (b) baserad på ett binärt sökträd.

   Testa din implementation med hjälp av testprogrammet. Använd 'runtime exceptions' för hantering av fel/undantagsfall. Du kan anta att lågt värde i element innebär hög prioritet.

Uppgift 1 kan komma att kräva en viss beräkningstid, men är oberoende av andra uppgifter. Därför är det kanske bra att börja tidigt med uppgiften - se nedan.

Genomförande

1. Tidsåtgång för insättning i och borttagning av element från en prioritetskö

Här är din uppgift att bestämma experimentellt prestandan av två redan existerande implementeringar av en prioritetskö: Den ena använder sig av en HEAP och den adra av ett BST. Klasserna finns kompilerade i filen `se.hig.aod.lab3.jar` (se nedan).

Du ska undersöka hur tidsåtgången ändrar sig med växande storlek av värden som finns i kön när det sätts in ett konstant antal värden i kön eller tas bort från den. I föreläsning 8 kan du läsa att tidsåtgången för både insättning och borttagning är $O(log(N))$ ifall en heap används. För ett binärt sökträd är det också $O(log(N))$ under förutsättning att trädet är välbalanserat. (Under vilka omständigheter är detta fallet?)

Gör en empirisk undersökning av tidsåtgången för insättnings- och borttagningsoperationerna på två olika prioritetsköer:

1. Heap-baserad kö: Insättning/borttagning av osorterat data.
2. Heap-baserad kö: Insättning/borttagning av sorterat data.
3. Heap-baserad kö: Insättning/borttagning av sorterat data (omvänd ordning).
4. BST-baserad kö: Insättning/borttagning av osorterat data.
5. BST-baserad kö: Insättning/borttagning av sorterat data.
6. BST-baserad kö: Insättning/borttagning av sorterat data (omvänd ordning).

Skapa en tabell över antalet element i kön och motsvarande tidsåtgång för insättning/borttagning:

	Heap						BST					
	Osorterat		Sorterat		Sorterat, omv.		Osorterat		Sorterat		Sorterat, omv.	
	Ins.	Utt.	Ins.	Utt.	Ins.	Utt.	Ins.	Utt.	Ins.	Utt.	Ins.	Utt.
10000												
20000												
40000												
80000												
160000												
320000												
640000												

Använd tabellen för att skapa ett **diagram över tidsåtgången som funktion av antalet element i kön**.

Material du ska använda:

ZIP-filen `aod_lab3_data.zip` innehåller `data_640000.txt` och `data_6400.txt`.

Data i filen `data_640000.txt` är avsedd för att fylla en prioritetskö. Använd filen i alla experiment, men läs olika antal värden från filen för att lagra de i kön (för antalen se tabellen).

Data i filen `data_6400.txt` ska däremot användas i själva experimentet: Alla värden som finns i filen ska läggas till en kö som redan innehåller ett antal värden.

Både den BST-baserade klassen `BSTPriorityQueue` och den HEAP-baserade klassen `HeapPriorityQueue` finns kompilerad i jar-filen `se.hig.aod.lab3.jar`. Lägg till jar-filen till projektet på samma sätt som i lab 1.

Varje experiment består av följande steg:

1. Skapa en instans av prioritetskön.
2. Lagra data från `data_640000.txt` i instansen.
3. Lägg till värden från `data_6400.txt` till instansen resp. ta bort 6400 värden från instansen och registrera exekveringstiden.

Med 'experiment' menas t.ex. mätning av tiden för insättning av 6400 element i en BST som innehåller 80000 element, som inte sorterades innan insättningen.

Observera att varje experiment måste utföras oberoende av alla andra experiment. Därför kan man inte återanvända data från ett annat experiment (t.ex. listan med värden för att fylla kön) eller själva kön i nästa steg - kön och listorna med värden måste skapas nytt varje gång.

Tips:

För at läsa data från textfilen kan du använda följande metod:

```
private ArrayList <Integer> loadList (String path, int size)
                                throws FileNotFoundException,
                                       IOException
{
    ArrayList <Integer> list = new ArrayList <Integer> ();
    int cnt = 0;

    BufferedReader in = new BufferedReader (new FileReader (path));
    String l;

    while ((l = in.readLine ()) != null && cnt < size)
    {
        list.add (Integer.parseInt (l));
        cnt++;
    }

    in.close ();
    return list;
}
```

Parametern `path` innehåller sökvägen till datafilen, `size` beskriver antalet element som ska hämtas från filen.

För att sortera data i ett `ArrayList`-objekt kan du använda `Collections`-klassen:

```
Collections.sort (data);
```

sorterar elementen i naturligt ordning,

```
Collections.sort (data, Collections.reverseOrder ());
```

sorterar elementen i omvänd ordning.

Exekveringstider kan mätas med hjälp av metoden `System.currentTimeMillis`:

```
long t1 = System.currentTimeMillis ();

// exekvera alla anropen - enqueue resp. dequeue

long exetime = System.currentTimeMillis () - t1;
```

Observera att du ska mäta tider bara för insättning av 6400 element ur filen `data_6400.txt` i en kö som redan innehåller ett antal element eller för att ta bort 6400 element ur en sådan kö.

Datamängder som måste hanteras i den här uppgiften är rätt så stora; det är möjligt att testprogrammet avbryter med en `StackOverflowError`. Ett sådant problem kan du lösa genom att ändra storleken av JavaVM:s stack.

När du använder kommandotolken, kan du lägga till parametern `-XssSTACKSTORLEK` till java-anropet:

```
java -Xss1000m JavaKlassAttExekvera
```

sätter storleken av stacken på 1000MB.

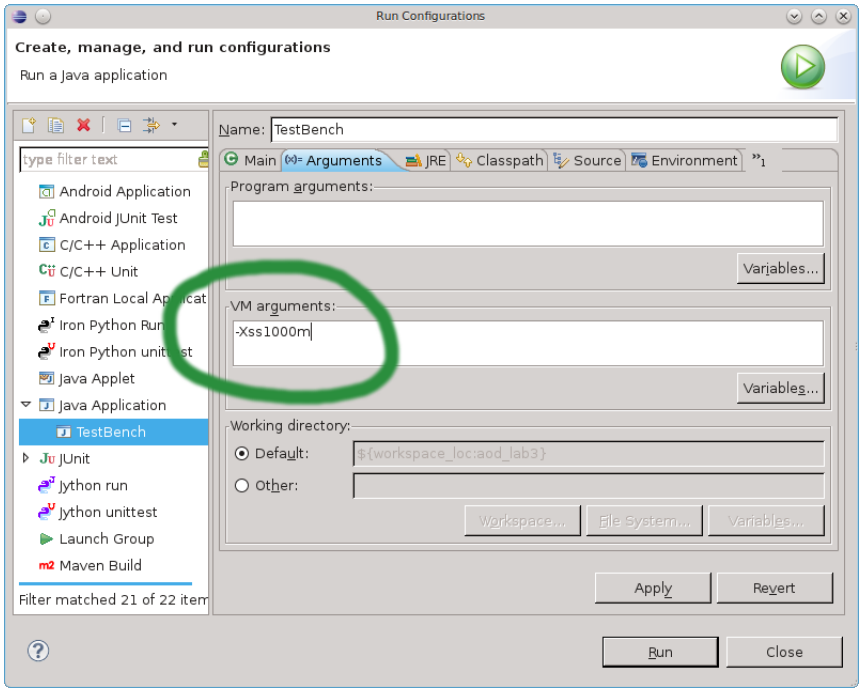

Figure 1: Storleken av stacken till JavaVM ändras till 1000MB.

I Eclipse kan du sätta parametern under 'Run configurations'. Fönstret når du i Eclipse via `Run → Run configurations...`

Om din dator inte klarar av datamängden, ska du använda delar av data. I så fall kan du testa köerna t.ex. med data som växer på följande sätt:

Serie 1:

5000, 10000, 20000, 40000, 80000, 160000, 320000 element

eller

Serie 2:

3000, 6000, 12000, 24000, 48000, 96000, 192000 element

Viktigt är att antalet element som sparas i kön innan experimentet sätts igång i varje steg fördubblas. Även antalet element som används under tidsmätning måste anpassas men det ska vara konstant i alla steg (t.ex. 2000 element till första serien eller 1000 element till andra). Elementen ska i varje fall tas ifrån `data_640000.txt` resp. `data_6400.txt`.

2. Interface för en prioritetskö

Java-interfacet ska du använda när du implementerar en prioritetskö. Klassen ska vara generisk; den ska kunna lagra objekt av klasser som implementerar `Comparable`. Detta krav kan uttryckas genom att definiera interfacet på följande sätt:

```
public interface PriorityQueue <T extends Comparable<?  super T>>
```

Tips:

- Metoden `enqueue` motsvarar en vanlig insättning i ett binärt sökträd.
- `dequeue` motsvarar operationen `removeMin` i ett binärt sökträd.

3. Testprogram för en prioritetskö

Skapa ett testprogram som kan användas för att testa de metoder som finns i det givna interfacet. Låt programmet använda prioritetskön för att till exempel lagra `Integer`-objekt. Det fungerar eftersom `Integer` implementerar interfacet `Comparable`.

Tips:

- Om du vill kunna kompilera ditt testprogram måste du skapa en ”stubb-klass” (med tomma metoddefinitioner) som implementerar interfacet (se lab 2).
- Glöm inte att förstöra kön innan programmet avslutas!

Skapa lämpliga testfall för att testa köns operationer. Utgå från testfallen vid testningen, och dokumentera utfallet av testerna genom att lägga skärmutskrifter som en bilaga till rapporten. Beskriv testfallen i rapportens resultatdel och hänvisa till bilagan med testresultaten, så att det tydligt framgår av rapporten att köoperationer fungerar. Exempel på lämpliga testfall:

- Ogiltiga operationer på en tom kö - t.ex. ta bort element, skriv ut element, titta på första element, osv.
- Ogiltiga operationer på en fylld kö - t.ex. lägg till element.
- ...

4. Implementationer av en prioritetskö

Du får välja vilken uppgift du vill lösa, (a) eller (b).

(a)

Skriv klassen `HeapPriorityQueue` som en heapbaserad implementation av en prioritetskö. Heapar lämpar sig bra för implementation av prioritetsköer eftersom de fungerar så att det minsta (alternativt det största, beroende på hur man väljer att implementera heapen) elementet ligger högst upp i heapen. Dessutom är tidsåtgången för insättning och borttagning av element i genomsnittsfallet lägre än för en listbaserad implementation.

Låt klassen använda sig av en arraybaserad heap för lagring av prioritetskköns element. Du behöver inte göra en dynamisk arrayimplementation, det vill säga en implementation som kan växa om arrayen blir full, men det är givetvis tillåtet att göra det om du vill.

Testa implementeringen på fel genom att använda testprogrammet du skapade i steg 2.

Skriv dessutom ut alla element i prioritetskön till ett terminalfönster. Elementet med högst prioritet ska skrivas ut först, därefter kan ordningen vara ospecificerad.

Använd 'runtime exceptions' för hantering av undantagsfall i prioritetskön.

Tips:

Vid utskrift av elementen i heapen kan du använda "bredden först", vilket i detta fall är lätt att implementera med hjälp av en enkel iteration. Observera att utskriftsordningen då blir ospecificerad, men att det minsta elementet alltid skrivs ut först.

(b)

Den mest naturliga implementationen av en prioritetskö är kanske att lagra elementen i en sorterad lista, där minsta elementet[1] då ligger först i listan. Vi ska inte studera det implementationssättet i denna uppgift. Istället ska vi välja en trädbaserad implementation.

Skapa klassen `BSTreePriorityQueue` som implementerar interfacet från uppgift 1. Låt klassen använda sig av en dynamisk implementation av ett binärt sökträd. Representera noderna i en generisk klass `TreeNode` som kan lagra objekt av klasser som implementerar `Comparable`. Se till att åtkomligheten till trädnodklassen är begränsad. Du behöver inte implementera fler operationer på det binära sökträdet än vad som behövs för att implementera operationerna i prioritetskö-interfacet. (Undantag: För att kunna skriva ut elementen behövs en metod som inte är definierad i interfacet.) Testa implementeringen på fel genom att använda testprogrammet du skapade i steg 1.

Skriv ut alla element i prioritetskön till ett terminalfönster. Elementet med högst prioritet ska skrivas ut först, därefter kan ordningen vara ospecificerad.

[1] Eller största elementet, om vi vill att högt elementvärde ska betyda hög prioritet.

5. Diskussionsfrågor

Att förbereda inför seminariet:

1. Hur bör prioritetskön hantera eventuella dubbletter?
2. Vilken är tidsåtgången i värsta fallet respektive ”genomsnittsfallet” för enqueue-operationen om vi använder
   (a) en listbaserad implementation?
   (b) en BST-baserad implementation?
   (c) en heapbaserad implementation?
3. Vilken är tidsåtgången i värsta fallet respektive ”genomsnittsfallet” för dequeue-operationen om vi använder
   (a) en listbaserad implementation?
   (b) en BST-baserad implementation?
   (c) en heapbaserad implementation?
4. Diskutera resultatet av den empiriska undersökningen (se uppgift 4) i förhållande till den teoretiska analysen.

Använd Ordo-notation för att ange tidsåtgången. Det är inte nödvändigt att genomföra några formella bevis, men försök att motivera svaren med informella resonemang. Ta gärna hjälp av kurslitteraturen och/eller annat referensmaterial som stöd för dina resonemang.

Redovisning

Programmeringsuppgifterna och rapportskrivning utförs gruppvis.

Packa ihop laborationsrapporten, skärmdumpar, källkoden och javadoc-dokumentationen i en jar-fil eller zip-fil, och ladda upp filen via inlämningslnken i Blackboard som tillhr laborationen.

Lycka till! :)