Laboration 2: Implementation av stack, kö och lista

version 2009-03-06 D. Howie, reviderad jackson 2012-02-27, perjee 2015-11-26

Syfte

Syftet med laborationen är att ge erfarenhet av implementation av en länkad lista samt av arraybaserade stackar och köer. Uppgiften ska också ge träning i att använda rekursion som programmeringsverktyg.
Till uppgiften hör en litteraturhandledning och en ZIP-fil som innehåller skelettkod i Java.

Förberedelse

Läs igenom hela uppgiftsbeskrivningen innan du sätter igång att arbeta. Annan nyttig läsning är föreläsningsanteckningarna från föreläsning 3-5 samt relevanta delar i kurslitteraturen. Se litteraturhandledningen till denna uppgift.

1. Skapa ett testprogram (t.ex. JUnit-test) som låter användaren se hur de olika operationer som specificeras av interfacet `List` kan användas; se vidare under *Genomförande*.
2. Implementatera en länkad lista: Skriv en klass som implementerar interfacet `List`. Använd *runtime exceptions* för att signalera undantag/fel till användaren av listan. Testa din länkade lista med hjälp av testprogrammet. Dokumentera testresultatet med en skärmdump och lägg till det kommenterade testprogram till laborationsrapporten.
3. (a) Skriv en arraybaserad implementation av en stack. Skapa en robust implementation genom att använda *runtime exceptions* för undantagshantering. Testa din implementation med hjälp av ett testprogram och dokumentera som ovan.
   (b) Skriv en arraybaserad implementation av en FIFO-kö, med hjälp av en cirkulär array. Använd *runtime exceptions* för undantagshantering. Testa din implementation och dokumentera testresultaten som i deluppgift (a).
4. Skapa ett interface `ExtendedList` som är en utökning av interfacet `List`, och implementera interfacet i den länkade listan. `ExtendedList` ska deklarera följande metoder:
   - insert (sätt in på en angiven position, första element har index 0)
   - remove (ta bort från en angiven position, första element har index 0)
   - get (titta på angiven position, första element har index 0)

Gäller för alla programmeringsuppgifter: Dokumentera de klasser som ni har skrivit med hjälp av j*avadoc*.

Genomförande

Filen `aod_lab2.zip` finns som en länk i Blackboard tillsammans med denna laborationsbeskrivning. Arkiv-filen innehåller ett par javainterface (`IndexedList.java`, `List.java`, `Stack.java`, `Queue.java`) och en undantagsklass (`ListIsEmptyException`) i paketet `se.hig.aod.lab2` (under `src`). Spara ner ZIP-filen och packa upp den på lämpligt ställe på ditt konto / din dator. Det finns även filen `se.hig.aod.lab2.jar` som innehåller tillhörande klassfiler.

1. Testprogram för en lista

Skapa ett testprogram som visar på hur man kan använda de metoder som finns. Det skall även testa att de undantag som man måste kunna hanterar verkligen uppstår. Metoderna som skall implementeras specifieras av interfacet `se.hig.aod.lab2.List`. Observera att du alltså ***inte*** ska arbeta mot `java.util.List`!

Tips: "Stubbar"

För att kunna kompilera hela projektet kan du redan nu skapa en stomme för klassen `LinkedList` som implementerar interfacet `List`. Låt klassen tillhöra samma paket som listinterfacet, dvs. `se.hig.aod.lab2`.

Detta kan enkelt göras i Eclipse:
Skaffa en ny klass (New -> Class). Bestäm att klassen ska implementera interfacet `List`.
Obs: Klassen är en generisk klass, dvs. namnet måste ha en typparameter: `LinkedList<T>`.

I den nya klassen saknas en rad metoder, och det signaleras i första raden av klassdefinitionen genom en problem-symbol. När man klickar på den, visas flera optioner. Dubbelklicka på 'Add unimplemented methods', resten gör Eclipse.

Låt metoderna i klassen vara tomma "stubbar", enligt följande mönster:

```
public T removeFirst() { return null; }
```

På så sätt kan du kompilera klassen trots att den inte är färdigskriven.

Låt testklassen innehålla en referens till ett `List`-objekt, och låt den referera till ett nytt objekt av typen `LinkedList`: Definiera som instansvariabel t.ex. `List<Integer> testList.`

I metoden av testklassen där du instansierar `LinkedList` följer:

```
testList = new LinkedList<Integer>();
```

Kompilera så att du ser att ditt testprogram anropar listmetoderna på rätt sätt (utan felmeddelanden).

2. Implementation av en länkad lista

Skapa klassen `ListNode` (som representerar en nod i listan) och implementera metoderna i klassen `LinkedList`.

Tips 1: Åtkomstbegränsning

Låt listklassen och dess metoder vara åtkomliga för alla användare, men se däremot till att åtkomligheten för listnodklassen är begränsad. Det kan åstadkommas på flera olika sätt:

1. Listnodklassen osynlig utanför paketet: Båda klasserna görs fristående, men listnodklassen görs otillgänglig utanför paketet. Det kan åstadkommas genom att använda synlighetsmodifieraren "default" (vare sig *public* eller *private*) i listnodklassen och dess datafält och metoder. I så fall är det endast klasser i samma paket (till exempel listklassen) som kan använda sig av listnoder och deras metoder.
2. Inre klass: Genom att låta listnodklassen vara en *inre klass* till listklassen kan åtkomsten begränsas så att endast listklassen kan komma åt listnodklassen och dess metoder.

Tips 2: Felhantering med 'runtime exceptions'

Skapa **egna** *runtime exceptions* genom att ärva från klassen `RuntimeException`, och använd dessa exceptions där det är lämpligt. Dvs skapa egna exception för varje typ av fel.

Tips 3: Rekursiva metoder

För att implementera den rekursiva metoden `printListR` kan en rekursiv hjälpmetod `printR` implementeras i klassen `ListNode`. Metoden skriver ut den aktuella nodens element och därefter skriver den ut nodens svans rekursivt. Denna hjälpmetod kan sedan anropas av `printListR` i klassen `ListNode`. Metoden `printListR` blir alltså en så kallad "driver routine" som i sig inte är rekursiv.

Tips 4: Metoden contains

Metoden ska returnera **`true`** om listan innehåller det objekt som anges som inparameter till metoden, annars **`false`**. Metoden kan med fördel implementeras med den privata hjälpmetoden `findElement` som returnerar en referens till den första listnod som innehåller det sökta elementet. Om elementet inte finns i listan så returneras null.

En iterativ implementation av `findElement` är ganska så rättfram: Stega fram ett element i taget och jämför det sökta elementet med den aktuella nodens datafält, tills det sökta elementet hittas eller slutet på listan nås. Använd metoden `equals` för att jämföra element! Hur hanteras fallet att listan är tom?

Tack vare hjälpmetoden blir implementationen av `contains` mycket enkel: Om `findElement` returnerar **`null`** så fanns elementet inte i listan.

Tips 5: Metoden reversePrintList

Metoden `reversePrintList` kan implementeras antingen med hjälp av iteration eller med hjälp av rekursion. Du får själv välja vilken metod du vill använda. Här är några tips:

- Om du väljer en **iterativ** implementation kan du använda dig av en stack.
- Om du väljer en **rekursiv** implementation kan du göra ungefär på samma sätt som i den rekursiva `printListR`-metoden. Det räcker att lägga till en rekursiv hjälpmetod `revPrintR` i listnodklassen, och koden blir mycket elegant. (Tips: Om din rekursiva metod börjar bli väldigt lång och krånglig är du garanterat på fel spår... :)

3 . (a) Arraybaserad stackimplementation

Skapa en klass `ArrayStack` som implementerar interfacet `Stack` (från laboration 1) med hjälp av en array som underliggande datastruktur.

Du behöver inte skriva en stack-klass där instanserna kan växa vid behov, utan användaren av klassen ska kunna bestämma en maximal storlek. Lägg till en parameter till konstruktorn! Dessutom behövs det nu en metod `isFull` - till skillnad till klassen `ListStack (`som använde sig av en länkad lista som underliggande datastruktur) är det nu inte möjligt att sätta in ett godtyckligt antal element i en stack.

Tips: Felhantering med 'runtime exceptions'

Se till att de metoder där det kan uppstå fel kastar en *runtime exception*.

3. (b) Arraybaserad kö-implementation

Skapa en klass `ArrayQueue` som implementerar interfacet `Queue` (från laboration 1) med hjälp av en array.

Du behöver inte göra en kö som kan växa vid behov (som klassen `ArrayList`) utan du kan välja en maxstorlek på arrayen. Även här behövs nu en metod `isFull`. Använd en "cirkulär" array med "slå runt-teknik" (wrap around, tips: använd modulus-operatorn).

Tips: Felhantering med 'checked exceptions'

Se tipsen till uppgift 3.1.

4. Fler operationer på en länkad lista

Utöka interfacet `List` med följande operationer, och implementera dem i klassen `LinkedList`:

- `insert` (sätt in på en angiven position)
- `remove` (ta bort från en angiven position)
- `get` (titta på angiven position)

Diskutera gärna ditt förslag på utökat interface med din basgrupp. Om du vill kan du använda interfacet `IndexedList` som finns i den medföljande ZIP-filen.

5. Diskussionsfrågor

Att förbereda inför seminariet:
Studera Java-interfacet `se.hig.aod.lab2.List` som medföljer laborationen. Jämför det med det interface som skapades i laboration 1, och jämför med interfacet `java.util.List` som ingår i standardbiblioteket. Använd gruppens forum för att diskutera likheter och skillnader mellan interfacen `List` och `java.util.List`.

1. Vilka operationer är egentligen "standardoperationer" på en lista?
2. Är det några standardoperationer på en lista som inte finns med i något av interfacen?
3. Specificerar något av interfacen fler metoder än de som brukar räknas som standardoperationer?
4. Hur skulle en rekursiv version av hjälpmetoden `findElement` (se tipsen till uppgift 2) kunna se ut?

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 inlämningslänken i Blackboard som tillhör laborationen.

Lycka till! :)