Instruktioner

Öppna en terminal och skriv omedelbart `mkdir kodprov180105`[1]. Gå in i denna katalog med `cd kodprov180105`. Hämta sedan kodprovet till din dator med följande kommando:

[1] Obs! . skall inte vara med i kommandona nedan!

```
curl -L --remote-name http://eliasc.github.io/ioopm/krokofant.zip
```

Nu får du en zip-fil med koden till uppgifterna. Packa upp den med `unzip krokofant.zip`, så får du ett antal filer och kataloger:

- `uppgift1` – filer för uppgift 1
- `uppgift2` – filer för uppgift 2
- `Makefile` – en makefil för att lämna in kodprovet

Inlämning

Ställ dig i katalogen `kodprov180105`. Om du har tappat bort dig i filsystemet kan du skriva `cd; cd kodprov180105`. Skriv `make handin` för att lämna in kodprovet. Då skapas en zip-fil med de filer som du har uppmanats att ändra i (inga andra), och denna fil sparas sedan på en plats där vi kan rätta provet.

Rättning

Vid rättningen körs din inlämningar mot mer omfattande testfall. Du har fått ut mindre omfattande testfall eller motsvarande i detta prov som du kan använda som ett stöd för att göra en korrekt lösning. Experiment med att lämna ut mer omfattande tester har visat sig skapa mer stress än hjälp (tänk fler testfall som fallerar)[2].

[2] Att lämna ut exakt samma test som används vid rättning är heller inte lämpligt, då det har förekommit fall då studenter försökt simulera en korrekt lösning genom att bara hacka output för specifika testvärden.

Om du har löst uppgifterna på rätt sätt och testfallen som du får ut passerar är du förhoppningsvis godkänd. Vi kommer att använda en blandning av automatiska tester och granskning vid rättning. Du kan inte förutsätta att den kod du skriver enbart kommer att användas för det driver-program som används för testning här.

I mån av tid har vi tidigare tillämpat ett system där vi ger rest för mindre fel. Huruvida rest tillämpas för ett visst kodprov beror på tid och resultat. Det betyder dock att det är värt att lämna in partiella lösningar, t.ex. en lösning som har något mindre fel.

Allmänna förhållningsregler

- Samma regler som för en salstenta gäller: inga mobiltelefoner, inga SMS, inga samtal med någon förutom vakterna oavsett medium, etc.
- Du måste kunna legitimera dig.

- Du får inte på något sätt titta på eller använda gammal kod som du har skrivit, eller röra filer som inte berör kodprovet.
- Du får inte gå ut på nätet.
- Du får inte använda någon annan dokumentation än man-sidor och böcker.
- Det är tillåtet att ha en bok på en läsplatta, eller skärmen på en bärbar dator. Inga andra program får köra på dessa maskiner, och du får inte använda dem till något annat än att läsa kurslitteratur.
- Du måste skriva all kod själv, förutom den kod som är given.
- Du får använda vilken editor som helst som finns installerad på institutionens datorer, men om 50 personer använder Eclipse samtidigt så riskerar vi att sänka servrarna.

Uppgift 1 (C)

Du får ut filen `queue.h` som innehåller funktionsprototyper med tillhörande dokumentation för en dubbel-ändad FIFO-kö, dvs. en kö där element stoppas in i en ände, och tas ut i den andra, i samma ordning som de stoppades in. (Notera att kön faktiskt tillåter att man tar element från båda ändarna.) Din uppgift är att implementera `queue.c` enligt instruktionerna här och i `queue.h`[3].

Liksom under kursen använder vi en *union* typedef:ad som `elem_t` för elementen. Notera att typen `queue_t` är typad som **`void`** i `queue.h`. Internt i `queue.c` kan du typomvandla detta till något annat. Du kan inte förutsätta något om antalet element man kan vilja lagra samtidigt i kön. Kön *skall* därför implementeras som en länkad datastruktur. För att kunna lägga till och ta bort element effektivt kommer du att behöva pekare till både första och sista länken, och varje länk kommer att behöva ha en pekare både till nästa och föregående element[4].

OBS! För att bli godkänd måste ditt program, förutom att vara korrekt, vara fritt från minnesläckage, och det får inte läsa oinitierat minne, skriva utanför allokerat minne eller skriva över data på stacken. Alla block allokerade av programmet skall vara explicit frigjorda i programmet innan det terminerar. Verifiera detta med `valgrind`. Du kan köra `make test` för att testa programmet[5], och `make memtest` för att köra testerna i valgrind. Observera återigen att testerna i makefilen är till för att hjälpa dig att hitta fel i din kod och att programmet passerar testerna inte nödvändigtvis betyder att programmet är korrekt eller att du är godkänd.

[3] Notera att utlämnade programmet inte kompilerar eftersom `queue.c` är tom.

[4] Det är en god idé att rita strukturen för att hålla tungan rätt i mun med alla pekare!

[5] Ett tips är att börja med att fylla i alla funktioner i `queue.c` så att programmet kompilerar, och sen implementera funktionerna en och en i ordningen som de testas!

Uppgift 2 (Java)

Du får ut en fullt fungerande implementation av en mängd, implementerad som ett träd (`TreeSet.java`). Trädet har en iterator som skapar en lista av alla noder i trädet (insamlade med en inorder-traversering) och sen itererar över den listan istället för själva trädet. Det här gör implementationen enkel, men det är onödigt minneskrävande för stora träd. Din uppgift är att byta ut implementationen av iteratorn så att den inte behöver lagra alla noder i trädet samtidigt. Den nya iteratorn kommer att använda sig av en stack som du också måste implementera.

Alla ändringar sker i filen `TreeSet.java`, och det är också den enda filen som lämnas in. Du kan köra `make test` för att köra testerna, och `make test-soft` för stänga av assertions, så att körningen inte avbryts när ett test misslyckas. Testerna kommer att hjälpa dig på traven med vad som ska göras[6]. Byt därför inte namn på klasserna, eftersom det kan göra att testerna slutar fungera som de ska!

[6] Kör du till exempel `make test-soft` på den utlämnade koden kommer testen att berätta att du använder ursprungslösningen och att du inte har implementerat en stack än.

Stacken

Du ska implementera en generisk klass `Stack` (en skiss finns påbörjad i slutet av `TreeSet.java`). Stacken ska implementera interface:et `StackI` som finns i filen `StackI.java`. Detta interface innehåller också kort dokumentation för varje metod. Notera att du inte får använda datastrukturer från Javas standardbibliotek (till exempel `LinkedList`) för att implementera stacken.

En stack är en datastruktur som erbjuder två operationer: `push`, som lägger till ett element i stacken, och `pop`, som plockar ut det senast tillagda elementet. Att anropa `pop` på en tom stack ska i den här implementationen leda till att undantaget `StackUnderflowException` kastas (detta undantag finns fördefinerat i en egen fil). I interface:et `StackI` finns även metoder för att titta på det översta elementet i stacken utan att plocka ut det (`top`), och att se om stacken är tom (`isEmpty`).

Notera att det inte finns några specifika tester för stacken, men det är lätt att skriva själv (till exempel i en `main` metod i klassen `TreeSet`). Om du pushar elementen 1, 2 och 3 i den ordningen och sedan poppar tre gånger ska du få tillbaka elementen 3, 2 och 1 i den ordningen. När stacken är tom ska `isEmpty` förstås returnera `true`[7].

[7] Det är värt att skriva dessa tester så att det inte leder till buggar när iteratorn ska implementeras!

Iteratorn

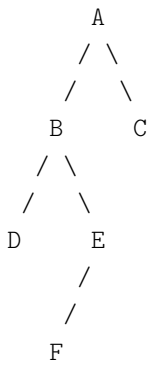
En iterator används för att iterera över en datasamling. I det här fallet finns bara två operationer[8]: `next`, som returnerar nästa element och "flyttar fram" iteratorn ett steg, och `hasNext`, som kollar om det finns fler element kvar. Notera att `next` returnerar datasamlingens *element*,

[8] Notera att `remove`, som vissa iteratorer erbjuder, inte behöver implementeras här!

och inte noderna i datasamlingen.

Den nya iteratorn kommer som sagt att använda sig av en stack. När iteratorn skapas pushas rotnoden och alla noder som går att nå genom att bara gå åt vänster i trädet. När `next` anropas poppas noden `n` från stacken. Därefter pushas `n.right` och alla noder som går att nå genom att bara gå åt vänster från `n.right`. Slutligen returneras `n`:s element. När stacken är tom finns det inga fler element, och då ska `hasNext` returnera `false`.

Som exempel, anta att trädet har följande utseende:



När iteratorn skapas kommer noderna `A`, `B` och `D` att pushas på stacken (i den ordningen). Första gången `next` anropas poppas `D` från stacken, och `D`:s element returneras (eftersom `D` inte har några barn kommer inga fler noder att pushas just då). Andra gången `next` anropas kommer `B` att poppas, `E` och `F` pushas till stacken, och `B`:s element att returneras. Följande två anrop till `next` kommer att poppa `F` respektive `E` och returnera deras element. Ytterligare ett anrop kommer att poppa `A`, pusha `C`, och returnera `A`:s element. Ett sista anrop till `next` kommer att poppa `C` och returnera dess element. Efter det finns det inga element kvar (då ska `hasNext` returnera `false`).

Notera att testerna testar `TreeSet`:s metoder som många är implementerade med hjälp av iteratorn. Om iteratorn inte fungerar kommer det att leda till att tester som inte explicit nämner iteratorn inte går igenom. Det betyder inte att du behöver ändra på några metoder utanför iterator-klassen! Om du kör fast, skriv mindre tester som bara testar iteratorn.