Kodprov 2018-10-24

1 Instruktioner

Öppna en terminal och kör följande kommandon:

1. `cd` (detta tar dig till din hemkatalog)
2. `mkdir kodprov 181024`
3. `cd kodprov 181024`
4. `curl http://wrigstad.com/ioopm/ byzantin.zip -o k.zip`
5. `unzip k.zip`

Nu har du fått ett antal filer och kataloger:

Objekt	Beskrivning
`uppgift1`	katalog med alla filer för uppgift 1
`uppgift2`	katalog med alla filer för uppgift 2
`Makefile`	en makefil för att lämna in kodprovet

1.1 Inlämning och rättning

Inlämning går till så här: ställ dig i katalogen `kodprov 181024`. Om du har tappat bort dig i filsystemet kan du skriva `cd; cd kodprov 181024`. Nu kan du skriva `make handin` för att lämna in kodprovet. När du kör detta kommando 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. Vid behov kan du köra `make handin` flera gånger – endast den sista inlämningen räknas.

Den automatiska rättningen kommer att gå till så att vi kör dina inlämningar mot 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 falleras)[1].

Om du har löst uppgifterna på rätt sätt och testfallen som du får ut passerar är du förhoppningsvis godkänd.

1.2 Allmänna förhållningsregler

- Samma regler som för en salstenta gäller: inga mobiltelefoner, inga SMS, inga samtal med någon förutom vakterna oavsett medium.
- Du måste kunna legitimera dig.
- Du får inte på något sätt titta på eller använda gammal kod som du har skrivit.
- 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.

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. Om du t.ex. implementerar en länkad lista av strängar kan testningen ske med hjälp av ett helt annat driver-program än det som delades ut på kodprovet.

I mån av tid har vi tidigare år tillämpat ett system där vi ger rest för mindre fel. Det är oklart om detta system tillämpas i år, men det betyder att det är värt att lämna in partiella lösningar, t.ex. en lösning som har något mindre fel.

Lycka till!

---

2 C-uppgift: Interned Strings

Din uppgift är att skriva färdigt filen `interned.c` så att den implementerar alla funktionerna i `interned.h` och passerar testerna i `driver.c`. Vad du måste göra är klart markerat med `///TODO` i `interned.c`.

Uppgiften går ut på att skriva klart ett bibliotek med sex publika funktioner som implementerar en teknik för att undvika flera kopior av samma sträng i ett program. Denna teknik kallas för *interned strings* vilket jag försvenskat till *internerade strängar*.

Biblioteket fungerar så att det finns två typer av strängar: *vanliga strängar* (som skapas som vanligt) och *internerade*

strängar (som endast kan skapas via `intstr_create()`). En internerad sträng är en null-terminerad array av tecken precis som en vanlig sträng, men sättet som de skapas på garanterar att om två internerade strängar är lika, så är de också identiska. Mer om det strax.

Varje gång en användare vill ha en internerad sträng anropar hen funktionen `intstr_create()` med den sträng som hen vill ha internerad, t.ex.

```
char *a = intstr_create("hej"); // skapa en interned string "hej"
```

Om det redan finns en internerad sträng för `"hej"` så returneras en pekare till denna sträng. Om inte, så skapar vi en kopia av `"hej"` som blir den nya internerade strängen och returnerar en pekare till denna. Följden blir att två internerade strängar `a` och `b` som är *lika* (dvs. `strcmp(a, b) == 0`) också är *identiska* (dvs. `a == b`):

```
char *a = intstr_create("hej"); // skapa en interned string "hej"
char *b = intstr_create("hej");
assert(a == b); // strängar med samma innehåll är
```

Inuti biblioteket används en **enkel hashtabell** för att hålla koll på vilka strängar som har internerats. För enkelhetens skull ligger denna hashtabell i en global variabel istället för att skickas in till alla funktioner.

Vid första anropet av `intstr_create()` för en sträng skapas ett entry i hashtabellen som håller i den internerade motsvarigheten. Efterföljande anrop till `intstr_create()` hittar entry:t och returnerar pekaren till den internerade strängen. **Denna kod är given.**

För att se om en sträng är internerad eller ej kan man använda funktionen `intstr_is_interned()` (**denna skall du skriva**):

```
char *a = "hej";
intstr_is_interned(a); // returnerar false
char *b = intstr_create("hej");
intstr_is_interned(b); // returnerar true
intstr_is_interned(a); // returnerar true
```

Observera att `intstr_is_interned(b)` inte svarar på om `b` *är* en internerad sträng, utan om det *finns* en likadan sträng internerad.

För varje interned string finns en räknare som håller koll på hur många gånger `intstr_create()` har anropats på den strängen. Denna räknare kallas för *strängens refcount* och vi kan få reda på en strängs refcount med funktionen `intstr_refcount()`. Initialt är en internerad strängs refcount 1. **Denna kod är given.**

```
char *a = "hej";
intstr_refcount(a); // returnerar 0 -- a är inte internerad
char *b = intstr_create("hej");
intstr_refcount(b); // returnerar 1
char *c = intstr_create("hej");
intstr_refcount(b); // returnerar 2
intstr_refcount(c); // returnerar 2
```

Funktionen `intstr_destroy()` används för att ange att man är klar med en sträng (**denna skall du skriva**). Varje gång `intstr_destroy()` anropas på en internerad sträng minskas dess refcount med 1:

```
char *b = intstr_create("hej");
intstr_refcount(b); // returnerar 1
char *c = intstr_create("hej");
intstr_refcount(b); // returnerar 2
intstr_destroy(b);
intstr_refcount(b); // returnerar 1
intstr_destroy(b);  // nu blir refcount 0
```

Anrop till `intstr_destroy()` med icke-internerade strängar ignoreras.

När en internerad strängs refcount når 0 i samband med ett anrop till `intstr_destroy()` tas den bort ur hashtabellen. Det betyder att nästa gång `intstr_create()` anropas för samma sträng så är det som om det var första gången vi anropade `intstr_create()` för den strängen.

```
char *b = intstr_create("hej");
intstr_refcount(b); // returnerar 1
intstr_destroy(b);
b = intstr_create("hej");
intstr_refcount(b); // returnerar 1
```

Utöver ovan nämnda funktioner finns också `intstr_init()` som skall anropas innan det första anropet till `intstr_create()` och vars syfte är att initiera biblioteket. **Denna kod är given.**

Sist finns `intstr_done()` (**denna skall du skriva**) som skall anropas efter sista användandet av en interned string i

programmet, och före `return` i `main()` och som tar bort alla entries i hashtabellen. Typiskt börjar ett program som använder detta bibliotek med ett anrop till `intstr_init()` först i `main()` och slutar med ett anrop till `intstr_done()`. Testerna gör detta korrekt förstås. Det är inte meningen att `intstr_init()` och `intstr_done()` skall anropas fler än en gång per program och det är ingenting som du behöver göra något med.

Din uppgift är att skriva färdigt `interned.c` så att den implementerar alla funktionerna i `interned.h` och passerar testerna i `driver.c`. Vad du måste göra är klart markerat med `///TODO` i `interned.c`.

2.1 Tips 1: funktioner som förväntar sig internerade strängar

Funktioner som förväntar sig internerade strängar bör endast använda sig av `==` vid sökning i hashtabellen.

2.2 Tips 2: upprepad logik

Mer än hälften av funktionerna gör en sökning i hashtabellen för att hitta ett entry. Du kan antingen återanvända (mycket av) logiken i `intstr_create()` eller extrahera en generell "find-funktion" från den. Observera att om du vill ha en find-funktion som också fungerar för borttagning behöver du antingen en "find previous" eller dubbelpekare.

2.3 Tips 3: Läs `interned.h`

Funktions-dokumentationen i `interned.h` är bra hjälp.

2.4 Icke-funktionella krav

1. Funktioner som opererar på hashtabellen skall använda sig av den givna stränghashfunktionen för att hitta rätt bucket att söka i (som vanligt i en hashtabell).
2. Entries i buckets behöver *inte* vara sorterade.
3. Du får använda en dummy om du vill – i så fall kan du skapa dessa i `intstr_init()` eller genom att ändra typen på `strings` till `entry_t` istället för `entry_t *` (beroende på vad du föredrar).
4. Ditt program får inte läcka minne. Efter att `intstr_done()` har körts skall allt heap-minne som biblioteket använt vara frigjort. Vidare skall `intstr_destroy()` städa bort entries och internerade strängar löpande under programmets gång, när refcount går ned till 0.

2.5 Testa din lösning (funktionella tester)

Du kan kompilera din lösning mot `driver.c` som utför enklare operationer på listan och rapporterar eventuella funna fel. Detta gör du med hjälp av den medföljande makefilen enligt följande:

`make test`
: kompilerar `driver.c` och `interned.c` till `a.out` och kör

`make memtest`
: som `make test`, men kör i valgrind för att hitta minnesfel

❗ Danger

Du får inte ändra i någon annan fil än `interned.c` eftersom det är den enda fil som lämnas in.

---

3 Java-uppgift: Reseplanerare

Uppgiften går ut på att utöka en existerande reseplanerare. Den centrala datastrukturen i programmet är en graf vars noder är stationer och vars bågar är resvägar mellan stationerna. Varje båge har en vikt som motsvarar restiden för sträckan. Noderna är instanser av klassen `Node` och bågarna instanser av klassen `Edge` (det sista skall du ändra på). Klassen `Network` håller i alla noder i en mängd.

Nedanstående ASCII-diagram visar ett litet nätverk med fyra kända platser från Göteborg, samt restiderna mellan dessa platser i minuter via det hypotetiska spårvagnsnätet "Prins Eugen". (Testerna på provet använder detta nätverk.)

```
// Modell 1

            5              7
Kortedala ------ Olskroken ----- Halta Lottas krog --, 4
    |                                    |          |
    '------------------------------------'          |
                    10                    S:t Sigfrids plan
```

När programmet startas (i klassen `TripPlanner`) läses en fil in som skapar nätverket. Detta sker genom att en serie stationer och deras vikter skickas till metoden `addLine()` (som du kommer att behöva utöka). Klassen `TripPlanner` använder sig av en funktion i `Network` för att hitta det kortaste avståndet mellan två platser i nätverket. Resvägen representeras i form av en ordnad lista av stationer (noder i nätverket) och sparas i `Trip` i instansvariabeln `route`. **Du behöver inte förstå denna algoritm för att lösa uppgiften, och algoritmen behöver heller inte ändras**.

Om jag till äventyrs befinner mig i Kortedala och vill till S:t Sigfrids plan kommer algoritmen att ge följande:

```
Kortedala
Halta Lottas krog
S:t Sigfrids plan
Total restid: 14 minuter
```

I nuvarande system modelleras inte det faktum att spårvagnsnätet består av flera olika linjer Vi återbesöker exemplet ovan med lite mer detaljer och låter `---...` avse en sträcka trafikerad av linje 1 och `===...` en sträcka trafikerad av linje 2.

```
// Modell 2

           5              7
Kortedala ===== Olskroken ===== Halta Lottas krog === 4
    |                                  |          ||
    '----------------------------------'          ||
                    10                    S:t Sigfrids plan
```

Diagrammet ovan visar att ettans spårvagn går mellan Kortedala och Halta Lottas krog, medan tvåans spårvagn trafikerar sträckan Kortedata, Olskroken, Halta Lottas krog, S:t Sigfrids plan. Om vi följer resplanen som gavs ovan måste vi alltså byta från ettan till tvåan vid Halta Lottas krog – **men i nuvarande system modelleras inte bytestiden**. Du kommer att åtgärda båda de fetstilta bristerna ovan genom att skapa nya klasser av bågar och ändra i `addLine()`.

Att åka tvåan från Kortedala till S:t Sigfrids plan utan byten tar 16 minuter. Såvida inte bytet vid Halta Lottas krog är snabbare än 2 minuter kommer denna resa följaktligen att vara snabbare.

Vi utökar därför modellen för att också innefatta bytestider genom att att representera spårnätverket ovan så här:

```
// Modell 3

           5              7
Kortedala ===== Olskroken ===== Halta Lottas krog === 4
    *                                  *          ||
    *                                  *          ||
  8 *                                8 *   S:t Sigfrids plan
    *                                  *
    *                                  *
Kortedala ----------------------Halta Lottas krog
                    10
```

Där `---...` är linje 1, `===...` är linje 2 och `***...` avser byten mellan linjer på samma station.

Stationer som knyter samman flera linjer förekommer nu flera gånger i grafen, en gång per linje (tänk "en gång per plattform"), och är sammankopplade med "bytes-bågar" med egen vikt som modellerar bytestiden (**för enkelhets skull alltid 8**). Så här skulle resan i exemplet ovan se ut om man inkluderar bytet. Notera att detta inte längre är den snabbaste resan från Kortedala till S:t Sigfrids plan:

```
Kortedala (start)
  |
  |  Linje 1, 10 minuter
  |
Halta Lottas krog
  *
  *  Byte, 8 minuter
  *
Halta Lottas krog
  ||
  || Linje 2, 4 minuter
  ||
S:t Sigfrids plan (stopp)
```

3.1 Uppgiften

Din uppgift är att utöka systemet med följande fyra "features":

3.1.1 Nya typer av bågar

Inför två nya typer av bågar med hjälp av subklassning: `Line` och `StopOver`. En `Line`-båge känner till vilken linje den tillhör och skall användas för att kunna skilja ut de olika spårvagnslinjerna. En `StopOver`-båge avser ett byte och har alltid vikten 8 (den fastslagna tiden för ett byte enligt det Göteborgska SpårvagnsVärket).

Observera att `Edge`-klassen inte har någon default-konstruktor. Om du subklassar `Edge` måste subklassens konstruktor anropa superklassens konstruktor. Syntaxen för detta är `super(...)` där `...` är argumenten.

Observera också att `Edge`-klassen har en hel del privata data och metoder som inte får override:as.

3.1.2 Ändra hur spårnätverket byggs upp

Metoden `addLine()` i klassen `Trip` lägger till en hel linje till nätverket (noder och bågar med vikter) i enlighet med Modell 1. Du skall ändra i `addLine()` så att nätverket byggs upp som i Modell 3.

En enkel algoritm för att åstadkomma detta för en given linje är att hålla reda på om en station n redan har lagts till i nätverket och i så fall skapa en kopia n' av stationen, länka samman n' med alla tidigare kopior $\{e_1, .., e_n\}$ av stationen med bytes-bågar och sedan ersätta n med n' i linjen. Här kommer samma beskrivning lite mer precist:

```
1. För varje nod n i linjen
   1.1. Låt E vara en mängd av alla existerande noder i nätverket med samma namn som n
   1.1.1. Om E inte är tom,
       1.1.1.1 skapa en ny nod n' med samma namn som n,
       1.1.1.2 för varje nod e i E, koppla ihop e och n' med en bytes-båge (en ny båge för
varje koppling),
       1.1.1.3 ersätt n med n' i linjen (alltså arrayen av noder).
2. Fortsätt med existerande logik som addLine() men fundera på vilken sorts båge som skall
användas..
```

Exempel: om linje 2 redan har lagts till och vi vill lägga till linje 1, kommer vi först att skapa en kopia av Kortedala, länka samman existerande Kortedala med kopian, och sedan uppdatera linje 1 så att den använder kopian istället för originalet.

Tips på bra metoder/funktionalitet i interfacet `Set`:

- `m.contains(n)` – returnerar `true` om `n` är med i mängden `m`
- `m.isEmpty()` – returnerar `true` om mängden `m` är tom
- `for (Node n : m) { ... }` – itererar över alla `n` i mängden `m`

Kod för att generera `E` ovan finns redan i `addLine()`.

3.1.3 Implementera funktionen `numberOfStopOvers()` i klassen `Trip`

Funktionen `numberOfStopOvers()` skall returnera antalet byten i en resa. Du kan se hur den anropas från `TripPlanner`.

3.1.4 Skriv färdigt funktionen `routeToString()` i klassen `Trip`

Denna funktion skall utökas med information om linjer, restid och byten. Du kan se hur den anropas från `TripPlanner`.

För varje *sträcka* (båge i nätverket) i en resa skall följande skrivas ut:

1. Linjen, om det är en linje – annars Byte
2. Restiden i minuter

Det vill säga, vi går från, till exempel detta:

```
"Kortedala
Halta Lottas krog
Halta Lottas krog
S:t Sigfrids plan"
```

Till detta:

```
"Kortedala
        Line 1, 10 minuter
Halta Lottas krog
        Byte, 8 minuter
Halta Lottas krog
        Line 2, 4 minuter
S:t Sigfrids plan"
```

OBS! Skriv *all* kod i filen `Trip.java` som redan innehåller en del kod. Nya klasser läggs i samma fil och skall därför inte vara `public`. Du kan söka på `TODO` i `Trip.java` för ledning.

OBS! Ingen del av uppgiften skall lösas genom att parsa strängar. Data om spårnätverket skall sparas i nätverket, inte i extra variabler i `Trip`.

3.2 Testning

Du kan testa din inlämning med hjälp av `make test`. Då kompileras och körs programmet i `TripPlanner.java` och kör två tester.

När du är klar bör programmets output vara följande:

```
Kortedala
        Linje 2, 5 minuter
Olskroken
        Linje 2, 7 minuter
Halta Lottas krog
        Linje 2, 4 minuter
S:t Sigfrids plan
Total restid: 16 minuter, 0 byten
```

Det finns också ett test där sträckan Olskroken–Halta Lottas krog med linje 2 för tillfället trafikeras av ersättningscykel och därför tar 17 minuter. För detta test förväntas följande output:

```
Kortedala
        Linje 1, 10 minuter
Halta Lottas krog
        Byte, 8 minuter
Halta Lottas krog
        Linje 2, 4 minuter
S:t Sigfrids plan
Total restid: 22 minuter, 1 byten
```

Questions about stuff on these pages? Use our Piazza forum.

Want to report a bug? Please place an issue here. Pull requests are graciously accepted (hint, hint).

Nerd fact: These pages are generated using org-mode in Emacs, a modified ReadTheOrg template, and a bunch of scripts.

Ended up here randomly? These are the pages for a one-semester course at 67% speed on imperative and object-oriented programming at the department of Information Technology at Uppsala University, ran by Tobias Wrigstad.

---

[1] 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.