Kodprov 2019-12-10

1 Instruktioner

Öppna en terminal och kör följande kommandon:

1. `cd` (detta tar dig till din hemkatalog)
2. `mkdir kodprov191210`
3. `cd kodprov191210`
4. `curl http://wrigstad.com/ioopm19/misc/guteblandning.zip -o k.zip`
5. `unzip k.zip`

Nu har du fått ett antal filer och kataloger:
`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 `kodprov191210`. Om du har tappat bort dig i filsystemet kan du skriva `cd; cd kodprov191210`. 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 fallerar)[1].

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

[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.

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 ofta tillämpat ett system där vi ger rest för mindre fel. Det är oklart om detta system tillämpas för detta kodprov 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: Treemap

En tree map är – precis som en hash map – en avbildning från nycklar till värden. I många fall kan en hash map och en tree map implementera samma gränssnitt. Den stora skillnaden ligger i hur datastrukturerna är implementerade internt (en genomgång följer strax).

Denna uppgift går ut på att implementera en tree map med följande gränssnitt:

```
void *treemap_insert(treemap_t *t, int key, void *value);
void *treemap_lookup(treemap_t *t, int key);
bool treemap_has_key(treemap_t *t, int key);
int *treemap_keys(treemap_t *t);
int treemap_size(treemap_t *t);
```

Därutöver följande funktioner för att skapa och riva ned en treemap.

```
treemap_t *treemap_create();
void treemap_destroy(treemap_t *);
```

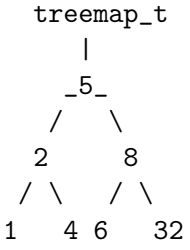
Viss kod är redan given.

Som namnet antyder bygger en treemap på *träd*. Uppgiften utgår inte ifrån att du har jobbat med träd förut – istället följer här en genomgång. Ett träd är uppbyggt som en sorterad länkad lista vars noder (aka länkar) har en nyckel (`int key`), värde (`void *value`) och därutöver två `next`-pekare som kallas `left` och `right`.

Sorteringen fungerar på följande sätt: Om vi står i en nod med nyckelvärdet k och följer `left`-pekaren kommer vi enbart att kunna hitta noder vars nycklar är *strikt mindre än* k. Om vi istället följer `right`-pekaren kommer vi enbart att kunna hitta noder vars nycklar är *strikt större än* k.

Detta gör det effektivt att söka i ett träd, men det är inte så viktigt för detta kodprov.

Nedan följer ett exempel på ett träd där endast nycklarna ritats ut för enkelhelt skull. Det torde vara enkelt att se att noderna i trädet uppfyller invarianten ovan, dvs. till vänster om varje nod finns bara noder med mindre nyckelvärden, och till höger finns bara noder med större nyckelvärden.



Om jag skall *leta* efter t.ex. nyckelvärdet 4 i trädet ovan börjar jag i *roten* av trädet och jämför dess nyckelvärde k med 4. Om $k > 4$ vet vi att lösningen ligger till vänster (vi brukar säga "i vänster subträd"), och vi skall alltså följa `left`-pekaren. Om $k < 4$ vet vi att lösningen ligger till höger (höger subträd), och vi skall alltså följa `right`-pekaren. Om $k = 4$ vet vi att vi har hittat rätt nod! Observera att detta är en rekursiv algoritm, efter att vi gått ned i vänster subträd i vårt exempel kan vi starta från början av denna beskrivning med `2` som *rot* (den är ju roten av subträdet) jämföra $2 < 4$ och se att lösningen ligger till höger (etc.).

Om vi vill *lägga till* något till trädet – säg nyckeln 3 och något värde v – utför vi exakt samma operation som ovan, tills dess att vi inser att nästa rekursiva steg skulle få oss att "trilla ut ur trädet". I exemplet med nyckelvärdet 3 kommer vi efter två steg att hamna i noden med nyckelvärdet 4 (gå vänster från 5 till 2, gå höger från 2 till 4). Eftersom $3 < 4$ kommer vi att vilja följa `left`-pekaren, men då märker vi att dess värde är `NULL`! Det betyder att vi har hittat platsen där trädet skall utökas – vi skapar en ny nod n med nyckeln 3 och värdet v och tilldelar resultatet till `left`-pekaren i 4.

Observera att n saknar subträd – dvs. `left` och `right` är `NULL`.

Ett tomt träd:

```
treemap_t
   |
  null
```

Om man följer algoritmen för att lägga till något i ett träd som gavs ovan kommer trädet inte automatiskt att förgrena sig jämnt (vi brukar säga att ett träd är balanserat eller inte). Det träd som du skall skapa skall **inte** vara balanserat, så du behöver inte tänka på sådana aspekter. Det betyder att om man skapar ett träd med nycklarna 1, 2, 3, 4, 5 i den ordningen skapas ett träd som ser ut så här:

```
treemap_t
   |
   1
    \
     2
      \
       3
        \
         4
          \
           5
```

Detta träd fungerar precis som en länkad lista: alla `left`-pekare är `NULL` och `right`-pekarna pekar på en nod med nästa nyckelvärde, förutom den sista noden vars `left` är `NULL`.

2.1 Uppgiften

Uppgiften går ut på att skriva klart implementationen av en treemap som finns i filen `yourcode.c`.

Du skall skriva all kod i filen `yourcode.c`. Filen `treemap.h` innehåller signaturerna för alla publika funktioner som skall finnas i `yourcode.c` samt

dokumentation för hur de olika funktionerna skall fungera. Det finns markeringar av typen `/// TODO` i filen som beskriver vad du skall göra.

Du kan förutsätta att `NULL` inte används som ett värde.

2.2 Icke-funktionella krav

- Du måste implementera hela programmet från grunden.
- Du får inte läcka minne eller läsa utanför initierat minne.
- Du skall skriva all din kod i `yourcode.c` som är den enda fil som lämnas in!

2.3 Testa din lösning

- `make compile` kompilerar `yourcode.c` mot testerna.
- `make test` kör testerna.
- `make memtest` kör testerna i valgrind för att hitta eventuella minnesfel.

---

3 Java-uppgift: Reseplanering

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, 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
```
