Databashanterare

En stor del av en programmerares arbete går ut på att ta hand om gammal kod, ofta skriven av någon annan. I den här uppgiften får ni koden till en fungerande men illa skriven databashanterare. Målet är att förbättra det, först genom modularisering och abstrahering, och till sist genom att byta ut databasens implementation mot en mer effektiv sådan.

Partiell lista över innehållet

- Abstrahering
- Dynamiska datastrukturer
- Enkel I/O

Introduktion

Filen `db.c` innehåller koden till en mycket enkel databashanterare. Programmet använder en länkad lista där varje nod innehåller två strängar – en nyckel (`key`) ett värde (`value`). Genom terminalinmatning kan en användare sedan slå upp nycklar och få ut motsvarande värde, ändra på en nyckels tillhörande värde, lägga in ett nytt nyckel-värde-par (givet en unik nyckel), ta bort ett nyckel-värde-par samt skriva ut databasen i dess nuvarande form.

Programmet börjar med att läsa in innehållet i en fil där varannan rad tolkas som en nyckel och varannan rad som tillhörande värde. Om filen `input` innehåller raderna

```
Apa
Djur
Banan
Frukt
Cirkus
Roligt
```

och programmet körs (efter att ha kompilerats till filen `db`) med kommandot

```
foo> ./db input
```

så kommer programmet skapa följande lista:

$$\langle(\texttt{"Cirkus"}, \texttt{"Roligt"}), (\texttt{"Banan"}, \texttt{"Frukt"}), (\texttt{"Apa"}, \texttt{"Djur"})\rangle$$

Om användaren slår upp nyckeln `"Apa"` så svarar programmet med värdet `"Djur"`.

Databaser

För att testa ert program finns ett antal färdiga databaser med namn och telefonnummer som ni kan använda. Dessa ligger i katalogen `/it/kurs/imperoopmet/databases` på institutionens filsystem. Databasen SWE.db är tänkt att användas för prova hur programmet funkar. Databaserna `TINY.db`, `SMALL.db`, `LARGE.db` och `HUGE.db` är automatiskt genererade[1] databaser som kan användas för prestandamätning.

För att undvika att kursen tar upp en massa hårddiskutrymme för påhittade namn och telefonnummer så ser vi gärna att ni använder de större filerna där de ligger (och inte kopierar dem till er hemkatalog). Det är också därför som programmet aldrig sparar några modifierade databaser till hårddisken, utan bara skriver dem till `stdout`. Självklart får ni skapa egna mindre testdatabaser om ni vill!

[1] Titta gärna på Python-skriptet `generate_db.py` som ligger i samma mapp

Uppgiften

Koden i `db.c` är inte så bra skriven. Förutom funktionen `readline` som används av kompatibilitetsskäl[2] så består hela programmet av en enda lång funktion, vilket gör koden svårläst. Dessutom delar olika sektioner av programmet på variabler och minnesutrymme, vilket gör att det kan bli svårt att ändra en liten detalj utan att behöva ändra på hela programmet.

Er uppgift är att förbättra koden, utan att lägga till eller ta bort någon funktionalitet.

1. Börja med att lära känna programmet genom att kompilera och köra det med någon liten databas, till exempel `SWE.db`. Titta på koden samtidigt som programmet körs (eller stega igenom det med `gdb`) och ta reda på vilken del av koden som gör vad. Skriv (tillfälliga) kommentarer i koden vid behov.

2. Dela sedan upp programmet i lämpliga funktioner. Vad som menas med ”lämplig” är inte alltid helt lätt att definiera, men en bra tumregel är att en funktion ska göra precis en sak (och ha ett namn som på ett bra sätt beskriver vad den gör). Använd funktionsabstraktion både för att göra programmet mer lättläst och för att unvika upprepning av kod.

3. Hela programmet är just nu väldigt medvetet om att databasen är implementerad som en länkad lista. Det innebär att man måste gå igenom hela programmet om man till exempel skulle vilja byta ut listan mot en annan datastruktur. Det är också svårare att testa så att datastrukturen är korrekt implementerad.

   Nästa steg är att flytta all kod som rör listan till en egen modul som kan kompileras och testas separat. Försök ta reda på vilka operationer som databashanteraren behöver, implementera dessa (ni kan förmodligen återanvända delar av redan existerande kod) och låt programmet anropa dessa istället för att manipulera listan direkt. Målet är att se till att databashanteraren är helt omedveten om vilken datastruktur som används!

4. Nu när själva databashanteraren är implementationsagnostisk kan vi byta ut den länkade listan mot en mer effektiv datastruktur! För att gå från linjär till logaritmisk tidskomplexitet på sökningarna i databasen kan vi till exempel välja att använda ett binärt sökträd. Implementera denna datastruktur i en egen modul och låt programmet använda den istället för den länkade listan. Om ni har abstraherat bort datastrukturen tillräckligt i tidigare steg och väljer samma namn på den nya datastrukturens operationer så borde ni kunna byta ut datastrukturen utan att ändra i `db.c` alls!

Tips

- Mycket av funktionsabstraktionen kan göras genom att ”klippa och klistra”. Kolla upp hur man gör det på bästa sätt med Emacs.
- Det är okej att lägga till och ta bort variabler vid behov, eller ändra på hur de används.
- Det är också okej att ändra på utskrifter eller inmatningar om ni tycker det gör programmet bättre på något sätt. Det enda viktiga är att det nya programmet stöder de fem menyvalen som orginalprogrammet stödde.
- Programmet är inte nödvändigtvis buggfritt från början! Var uppmärksam på eventuella fel i programmet och åtgärda dem om ni hittar dem.
- Tänk på att listmodulen förmodligen inte bör hantera någon in- och utmatning alls. Den funktionaliteten ska själva databashanteraren erbjuda. Skriv listmodulen så att den skulle kunna användas i ett annat program som behöver en lista (till exempel av er, på senare uppgifter i kursen).
- Fundera på hur man kan dölja datastrukturen så att programmet till synes bara använder en ”databas”, vars interna detaljer kan bytas utan att det märks utifrån.

[2] `fgets` behandlar avslutande radbrytning olika på olika system

Förslag till redovisningar kopplade till denna uppgift

Nedan beskriver vi några möjliga kunskapmål vars uppfyllnad kan redovisas som en del av denna uppgift. Listan är inte på något sätt uttömmande. Det är heller inte så att man måste redovisa de kunskapmål som nämns nedan i samband med denna uppgift, eller att denna uppgift är ”den bästa” att redovisa dessa i samband med.

A1 (Procedurell abstraktion) Den första delen av uppgiften går ut på att bryta ner main-funktionen i mindre funktioner och ge dem vettiga namn så att programmet blir lättare att läsa och förstå. Mer relevant för det här målet kan en uppgift nästan inte bli!

C7 (Modularisering) När ni separerar list-implementationen från koden som använder listan så är det modularisering ni håller på med. Eftersom det också är en sorts abstrahering så kan diskussionerna för A1 och C7 gå i varandra till viss del. Sökning i det binära sökträdet är ett exempel på ”divide and conquer”.

M38 (Länkade strukturer) Programmet innehåller redan en implementation av en länkad lista (som med fördel kan förbättras) och senare i uppgiften ska ni implementera ett binärt sökträd. Båda dessa är exempel på länkade strukturer.

D10 (Dokumentation) När programmet är klart ska ni ha minst tre moduler. Dessa bör förstås dokumenteras så att någon utomstående kan använda dem utan att behöva läsa och förstå all kod. När ni skriver dokumentation kommer ni kanske på flera funktioner som dessa moduler borde ha.

E11 (Genericitet) Om man vill kunna återanvända sina datastrukturer till annan kod så kan det vara en bra idé att göra dem generella (notera att detta inte är nödvändigt för att klara av uppgiften). Fundera på hur man skickar och tar emot data till/från modulerna om allt hanteras som void-pekare internt.