Deluppgift 6 Associativ container

Uppgift

Du skall implementera en program-modul som fungerar som en associativ "data-container" där värden kan stoppas in för lagring. Jämför med en `std::map` i C++. Varje värde som stoppas in associeras med ett id (en "nyckel" unikt för det datat). Nyckeln används senare för att kunna hämta ut just det datat ur containern. Ett exempel på en associativ container är en ordlista. Stoppas svenka ord in med motsvarande engelska ord som nyckel kan man sedan lätt söka efter det engelska order (nycken) för att få ut den svenska översättningen (värdet knutet till nyckeln). Ett annat exempel är hur operativsystemet kan hålla reda på en viss resurs åt processer. Varje resurs processen reserverar får ett id och stoppas in i en associativ container där id blir nyckel. När resursen skall användas behöver processen bara ange det id den fick när resursen reserverades för att operativsystemet skall hitta rätt resurs. *Denna container kan senare användas för att hålla reda på filer eller processer.* De funktioner och datatyper som behövs för att implementera modulen beskrivs nedan.

En programmodul består av två filer. En header-fil (här `map.h`) som specificerar (deklarerar) de datatyper och funktioner modulen tillhandahåller, och en implementationsfil (här `map.c`) som definerar varje funktion. Dessutom behövs ett eller flera huvudprogram (`main.c, ...`) för att testa modulen noggrant. Huvudprogrammet inkluderar header-filen för modulen. Titta i uppgift 4 angående kompilering av program på flera filer.

Tänk på att kontrollera att givna argument till alla funktioner som ingår i modulen är giltiga innan de används. Det bör naturligtvis göras internt i funktionerna, annars måste ju kontrollerna skrivas om varje gång funktionen anropas. Anges felaktiga indata, t.ex. en nyckel som inte finns i samlingen, så betraktas det som fel och PANIC() anropas *om du inte kommer på något bättre sätt* att hantera felet.

I det följande hör varje deklaration till header-filen, medan implementationen av funktionerna som beskrivs hör till implementationsfilen (som vanligt alltså).

```
#define PANIC() exit(1)
```

En preprocessor-definition som kan användas om något går allvarligt fel eller om det är oklart vad som skall hända, t.ex. om listan är full. Detta makro finns i Pintos (men avslutar lite annorlunda).

```
typedef char* value_t;
typedef int key_t;
```

För att enkelt kunna byta ut den datatyp som används som nyckel, och datatypen på värdet som lagras med varje nyckel definieras "alias" för dessa typer. I fortsättningen används bara aliaset, aldrig den bakomliggande typen.

```
typedef enum {false, true} bool;
```

Om din kompilator inte tillhandahåller någon datatyp för bool automatiskt kan denna typdefinition behövas. Då får du typen `bool` och konstanterna `false` och `true`. Alternativt finns ibland en standardinkluderingsfil `<stdbool.h>` att använda.

`struct map; /* stores all objects in list or array */`

En datatyp för att representera hela containern (med alla nycklar och värden). En pekare till denna måste alltid skickas med som parameter till funktionerna som manipulerar containern för t.ex. insättning och sökning. För att representera containern internt har du huvudsakligen två aternativ:

Alternativ 1:
Du kan definiera en array av fast storlek som lagrar en samling värden. Nyckeln behöver inte lagras, utan indexet i arrayen fungerar som nyckel. Ungefär som en otroligt förenklad hashtabell. Denna lösning är speciellt effektiv då värdet för en nyckel söks, eftersom det då bara är att indexera arrayen med nyckeln. För att ytterligare förenkla förutsätter vi att värdet som skall lagras för varje nyckel är en pekare av något slag. Detta gör att oanvända positioner i arrayen kan hållas reda på genom att sätta dem till `NULL`. Tänk på att hantera fallen då listan är full. Följande deklarationer behövs:

```
/* symbolisk konstant för att lätt kunna ändra storleken
   i fortsättningen används denna då storleken behövs */
#define MAP_SIZE 128
struct map
{
  value_t content[MAP_SIZE];
};
```

Alternativ 2: (rekommenderas för de som kan lite mer)
Du kan använda den länkade lista som finns i PintOS (uppgift 4) för att hålla reda på samlingen värden och dess nycklar. Här behövs ingen fast storlek eftersom listan kan växa dynamiskt, alltså slipper du hantera fallen då listan är full. Å andra sidan måste minnet allokeras och återlämnas korrekt, och eftersom listan saknar index måste nyckeln lagras tillsammans med varje värde. En struct definieras till detta.

```
struct association
{
  key_t   key;   /* nyckeln */
  value_t value; /* värdet associerat med nyckeln */
  /* list-element för att kunna sätta in i listan */
  struct list_elem elem;
};

struct map
{
  /* listan med alla lagrade associationer */
  struct list content;

  /* räknas upp varje gång en ny nyckel behövs */
  int next_key;
};
```

```
void map_init(struct map* m);
```

Denna funktion motsvarar en konstruktor i C++ och måste anropas manuellt varje gång en variabel av typen `struct map` skapas. Alla datamedlemmar i `struct map` ska här initiearas till kända värden som ger en tom container.

```
key_t map_insert(struct map* m, value_t v);
```

En funktion för att sätta in en ny associering. Värdet anges som parameter. Värdet läggs till samlingen och den nyckel det fick returneras. Den som tar emot den returnerade nyckeln skall senare kunna använda den för att få fram värdet igen.

```
value_t map_find(struct map* m, key_t k);
```

Då värdet för en viss nyckel söks anropas dennna funktion. Värdet returneras om det hittas, annars returneras `NULL`. Det är upp till den som anropar att kontrollera att något hittades.

```
value_t map_remove(struct map* m, key_t k);
```

En funktion för att ta bort en association ur samlingen. Fungerar som `map_find`, men tar även bort det funna värdet ur samlingen.

Utöver ovanstående basfunktionalitet kommer det att finnas behov av lite mer avancerade funktioner. Nedanstående nyttjar funktionspekare (jämför med lambdafunktioner eller funktionsobjekt i C++) för att vara generella, men det går även bra att senare implementera just de specifika varianter som behövs istället.

```
void map_for_each(struct map* m,
                  void (*exec)(key_t k, value_t v, int aux),
                  int aux);
```

Tanken är att funktionen skall gå igenom hela samlingen associationer (alla nycklar och värden) och anropa `exec` för varje association. Pekaren till funktionen `exec` anges som som andra argument. Den tredje parametern skickas direkt vidare till varje anrop av `exec`. Den kanske inte behövs i alla lägen, men kan vara bra att ha ibland.

Tittar vi på andra parametern ser vi att det är en pekare till en funktion som tar tre parametrar och returnerar `void`. Just detta syntax ser lite struligt ut i C, men tänk bort parenteserna runt ordet exec och stjärnan framför så ser du att det är en normal funktionsdeklaration man gjort om till pekare.

När man vet vilken funktion som behöver anropas på varje lagrad association kan man enkelt hårdkoda det funktionsanropet istället för att använda en funktionspekare som andraparameter. Om du tycker det är enklare så går det bra att göra så.

```
void map_remove_if(struct map* m,
                   bool (*cond)(key_t k, value_t v, int aux),
                   int aux);
```

Skall fungera precis som `map_for_each`, men om funktionen `cond` returnerar `true` när den anropats skall associationen (nyckel och värde) tas bort från samlingen. Bra att ha om alla eller vissa associationer skall tas bort och man dessutom vill behandla varje värde på något sätt precis innan borttagning, t.ex. avallokera minne eller stänga filer. Eftersom värdet förutsätts vara en pekare av något slag är det stor chans att den är dynamiskt allokerad. Dessutom kanske den pekar ut en struct som innehåller medlemmar som behöver avallokeras eller hanteras på annat sätt.

Även denna kan hårdkodas utan funktionspekare om så önskas.

Testprogram

Ett påbörjat testprogram finns bland de givna filerna.