Övningar i kursboken:
Ch 7.9: E1-4 8.2: E1-3 8.3: E1-4 8.6: E1-6, 8-9 8.7: E1
9.1: E1-8, 12-13 9.2: E1-5

***Obs**: I uppgift 3.1-3 nedan ska du använda BC5. Jfr instruktioner till lab 2.*

3.1 Koda och testa en ADT för en *tabell*, med följande definition:
Data:
- Tabellen lagrar, på bestämda platser (index), ett begränsat antal element av viss typ.

Operationer:
- Skapa en tom tabell.
- Placera ett nytt element på en plats i tabellen som entydigt kan associeras med elementet (t ex genom en elementnyckel).
- Sök ett element i tabellen.
- Töm tabellen.
- Genomlöp tabellen och gör något med varje element som påträffas, utan att förändra det.

Använd c- och h-filer på samma sätt som i *lab 1.1*.

Specifikation:
- Tabellen ska vara en *hashtabell*.
- Låt elementdata (en **`struct`**) bestå endast av nyckeln (heltal).
- Använd de funktionsprototyper som finns i kursboken (se *Ch 8.7* sid 360-61), samt en egen prototyp för **`TraverseTable()`** som genomlöper tabellen.
- Ange lämplig tabellstorlek (**`HASHSIZE`**) om max 10 element ska lagras i tabellen
- Placera element med nyckel mellan -100 och 100 i tabellen (för att förenkla tillåter vi att fler än ett element kan få samma nyckel). Fortsätt till dess att tabellen är full. När ska detta anses inträffa? Använd valfri hashmetod samt valfri metod för indexökning (sid 355-59).
- Skriv ut hela tabellen, med blanksteg för tom plats, med hjälp av **`TraverseTable()`**.
- Gör några olika nyckelsökningar i tabellen för att kolla att den fungerar som den ska.

Antag att du vill lägga till en operation:
- Ta bort ett element med viss nyckel. Varför är denna operation problematisk?

3.2 Förändra din hashtabell i *lab 3.1* så att tabellens storlek kan bestämmas av användaren under körning. Istället för **`#define HASHSIZE ..`** i **`hash.h`**-filen använder du en global variabel **`hashsize`** i **`hash.c`**-filen. Variabeln kan sedan tilldelas önskat värde i din **`main`**-fil, varefter **`CreateTable`** använder variabeln för dynamisk minnesallokering av tabellarrayen.
Typdefinition för hashtabellen blir alltså: **`typedef TableEntry* HashTable;`**
Notera att **`CreateTable`** i detta fall måste använda en **`HashTable`**-*pekare* som parameter.

Testa på samma sätt som i *lab 3.1*.

3.3 ***Obligatorisk***: Skriv ett program som visar antalet nyckeljämförelser (*probes*) vid sökning i en hashtabell.

Specifikation:
- Skapa en hashtabell enligt *lab 3.2*.
- Användaren ska ges möjlighet att testa din tabell. Antal element (max 10) anges av användaren varefter slumpade heltal mellan 15 och 25 läggs i en liten testtabell av lämplig storlek. Skriv ut tabellen med mellanslag på tomma platser. Låt användaren därefter

ange ett element som ska sökas. Skriv ut resultat (tabellindex) samt antal nyckeljämförelser. Användaren ska kunna fortsätta att testa tills han/hon är nöjd.
- En större tabell med 10000 element ska sedan testas. Skapa en tabell av lämplig storlek.
- Slumpa 10000 st heltal mellan 15000 och 25000 och lägg i tabellen.
- Sök en slumpvald nyckel (heltal mellan 15000 och 25000) i tabellen. Upprepa (med ny nyckel) minst 100 gånger.
- Skriv ut medelvärdet (flyttal) av antalet jämförelser/sökning och den totala processortiden (flyttal) för de 100 sökningarna.
- Testsökningar i ny tabell ska kunna göras om användaren önskar.
- *Obs*: Se till att frigöra minne varje gång en tabell är färdiganvänd!

Jämför resultatet med *lab 2.5*. Undersök vad som händer när du minskar tabellstorleken och låter dess värde vara ett primtal resp icke-primtal. Kan du sätta tabellstorleken till antalet värden som ska lagras? Slutsatser?

3.4 ***Valfri***: Se kursboken, *Ch 8.6 P3,* sid 367. Du ska skriva ett program som läser in ett ämnes kemiska formel (Na , H_2O , $Al_2(SO_4)_3$ …) varefter molekylvikten skrivs ut.
Filen **`period.txt`** innehåller atomvikten för de flesta grundämnen i det periodiska systemet.

3.5 Skriv ett program som infogar *n* st slumptal i ett *binärt sökträd*, och därefter skriver ut talen i stigande ordning. Testa med t ex 10, 15 och 20 st slumptal. Töm trädet mellan varje test.
Lägg dina prototyper resp kod för trädfunktionerna i en h- resp c-fil, på samma sätt som i *lab 1.1*. Använd lämpligen följande trädoperationer:
- Skapa ett tomt träd.
- Kolla om trädet är tomt.
- Infoga ett element på rätt plats i trädet.
- Genomlöp trädet i stigande ordning med avseende på elementnyckeln och gör något med varje element utan att förändra trädet.
- Töm trädet (jfr *Ch 9.1 E7*, sid 394).

3.6 Lägg till följande trädoperation i *lab 3.5*:
- Beräkna trädets höjd (jfr *Ch 9.1 E6*).

Använd *rekursiv* metod.

3.7 ***Obligatorisk***: Skriv ett program som med hjälp av ett binärt sökträd skapar en telefonlista. Varje element innehåller efternamn, förnamn samt telefonnr (inkl rikt-nr). Använd lämpligen hela namnet som nyckel.

Specifikation:
- Använd trädtypen (inkl operationer) från *lab 3.5*, med elementtypen modifierad.
- Läs in minst 10 element till ett binärt sökträd. Elementen bör inte vara ordnade före inläggningen. Varför?
- Användaren ska sedan kunna göra följande:
  - Skriva ut listan i bokstavsordning m a p efternamnet.
  - Söka ett namn i listan och få en utskrift av posten om den finns.
  - Lägga en ny post till listan.
  - Skriva ut trädets höjd.

3.8 ***Valfri***: Ett flygbolag använder sig av följande metod vid överbokningar. Om en plats blir ledig i en fullbokad tur kommer platsen att gå till den som står först i en *prioritetskö*. Placeringen i kön avgörs av hur många som redan står i kön när man bokar och om man är medlem i bolagets bonusklubb. I det senare fallet tas hänsyn till hur långt man rest med bolaget under det senaste året och hur många år man har varit medlem. Beräkningen av prioritetsvärdet p sker med formeln:

$p = a + b - c$

där a = sträcka (mil, hundratal) , b = antal år och c = aktuell köstorlek (inkl nyinsatt)

Filen **`reserv.txt`** innehåller namnen på ett antal personer som vill resa till en datamässa i Las Vegas med ett flyg som tyvärr är fullbokat. Namnen står i den ordning som personerna har satt upp sig på reservlistan. För medlemmar i bonusklubben finns även sträckan samt antal medlemsår angivna.

Skriv ett program som läser in filen och som hanterar prioritetskön m h a en *heap*. Följande ska kunna göras och upprepas, med bevarade heap-egenskaper:

- Skriva ut prioritetskön.
- Skriva ut heapen i nivåordning.
- Ta bort den förste resenären ur kön (lämpligen med utskrift).
- Sätta in en ny förhoppningsfull resenär.