Algoritmer - Laboration 1

Inledning

Den här laborationen behandlar balanserade träd. Syftet är att träna er på att implementera de viktigaste operationerna i några relativt avancerade datastrukturser. Första delen av laborationen utförs i språket Haskell, och den andra i språket C.

Då uppgiften behandlar vanliga datastrukturer så finns det förstås väldigt mycket material tillgängligt på nätet. Tanken med laborationens utformning är dock att ni inte skall behöva söka annat material än det som givets på föreläsningen, och det som finns i boken, ***avseende datastrukturerna***. Självklart får ni däremot förstås fritt leta och utnyttja material om språken Haskell och C. Reglerna för den här laborationen är därför enkla – ni skall inte utnyttja andra källor än de ovan angivna avseende implementeringar av datastrukturerna. Specifikt är det uttryckligen förbjudet (och det kommer att betraktas som fusk) att lämna in kod som utgår från en lösning funnen på nätet. Det är däremot *inte* förbjudet att utgå från stegvisa beskrivningar av de olika operationerna, alternativt pseudokod – men i så fall skall de källor (websidor) ni utgått från anges i inlämningen. Självklart är det, som vanligt, även förbjudet att dela lösningar mellan grupperna. Vid inskick skall gruppen även garantera att de inte utnyttjat otillåtet material genom att i det inskickade dokumentet sist infoga meningen:

”Vi [gruppens medlemmar] lovar och svär att vi följt instruktionerna avseende plagiering, och därvid utvecklad vår laborationslösning själva.”

Uppgift 1 Sökträd i Haskell

Den här delen av laborationen behandlar alltså AVL-träd och implementationen skall ske i Haskell. Uppgiften består av ett antal steg, vilka kommer att leda er fram till en fullständig lösning. Observera att ni skall utgå från det givna programskelettet, och att ni i princip inte behöver lägga till fler funktioner än de som nämns. I uppgiften ingår även ett antal frågor, vilka skall besvaras i ett Word-dokument (motsvarande) och skickas med inlämningen. Det är tillåtet att rita skisser för hand, men dessa skall i så fall vara läsliga och de måste scannas in så att hela inlämningen kan ske elektroniskt.

Del A: Binära sökträd

Som en uppvärmning skall vi arbeta med vanliga binära sökträd. Börja med att ladda in BinTree_START.hs. Som synes är detta en modul i Haskell, vilken då förstås skall hantera binära sökträd. Om vi börjar med att betrakta inledningen så hittar vi följande:

```
module BinTree (BinTree,emptyTree,treeMember,treeInsert,
                        treeDelete, buildTree,inorder,treeSort) where

treeMember   :: (Ord a,Show a) => a -> BinTree a -> Bool
treeInsert   :: (Ord a,Show a) => a -> BinTree a -> BinTree a
treeDelete   :: (Ord a,Show a) => a -> BinTree a -> BinTree a
buildTree    :: (Ord a,Show a) => [a] -> BinTree a
inorder      :: (Ord a,Show a) => BinTree a -> [a]
treeSort     :: (Ord a,Show a) => [a] -> [a]

data (Ord a) => BinTree a = EmptyBT
                          | NodeBT a (BinTree a) (BinTree a)
    deriving Show
```

Det här säger då att vi har en modul BinTree vilket innehåller deklarationer för och exporterar `BinTree`, `emptyTree`, `treeMember`, `treeInsert`, `treeDelete`, `buildTree`, `inorder`, och `treeSort`. Just nu är dock denna rad bortkommenterad och ersatt med en annan där inga funktioner exporteras. Efterhand som ni implementerar funktionerna skall ni därför utöka listan av exporterade funktioner igen.

Efter detta ges deklarationer, inklusive typer, för de sex olika funktionerna. Även dessa är nu bortkommenterade.

Fråga 1) Vad innebär bokstaven a i dessa typdeklarationer?
Fråga 2) Vad innebär `(Ord a,Show a)` i deklarationerna av funktionerna?

Därefter kommer själva datatypen, vilken då säger att ett binärt sökträd antingen är ett tomt träd (konstanten `EmptyBT`) eller en nod med ett värde och två underträd: `NodeBT a (BinTree a) (BinTree a)`

Nästa rad: `emptyTree = EmptyBT` inför sedan bara ett alternativt skrivsätt för `EmptyBT`

Nu är vi äntligen beredda att skriva den första funktioen `treeMember`, vilken förstås skall avgöra om ett visst element finns i trädet eller inte. Som synes av koden nedan och i filen är det mesta av funktionen redan klart:

```
treeMember v' EmptyBT                  = False
treeMember v' (NodeBT v lf rt) | v==v' = True
                               | v'<v  = ??
                               | v'>v  = ??
```

Notera hur funktionen är strukturerad med mönstermatchning. Första raden säger att v' inte är medlem i ett tomt träd. Andra raden skiljer sedan ut de tre fall där trädet inte är tomt. Första alternativet är att v==v' och då fanns förstås v' i trädet. De två alternativ som återstår att implementera är de rekursiva då sökningen sker i antingen vänster eller höger underträd beroende på om v' är större eller mindre än v. Färdigställ och testa funktionen treeMember. För att pröva den finns några träd deklarerade som t1-t3.

Fråga 3) Hur ser de olika träden t1-t3 ut, dvs. rita upp de tre träden.

Nästa funktion är treeInsert, vilken förstås skall sätta in ett element i trädet. Funktionen följer samma mönster som treeMember, och återigen återstår bara de rekursiva anropen. Observera att om vi försöker sätta in ett element som redan finns så förändras inte den noden.

```
treeInsert v' EmptyBT                       = NodeBT v' EmptyBT EmptyBT
treeInsert v' (NodeBT v lf rt) | v'==v      = NodeBT v lf rt
                               | v' < v     = ??
                               | otherwise  = ??
```

Fråga 4) Hur ser de olika träden t1-t3 ut om ni sätter in: 4 i t1, 1 i t2 respektive 8 i t3.

För att bygga ett helt träd från en lista med heltal finns funktionen:

```
buildTree lf = foldr treeInsert EmptyBT (reverse lf)
```

Fråga 5) Vad är `foldr` för funktion? Specifikt vad är dess signatur. Förklara signaturen med egna ord. Slutligen, varför måste vi vända på listan för att få den önskade effekten?

Vi kommer nu till `treeDelete`, som är den mest komplexa funktionen. Det svåraste fallet är då noden som skall tas bort har två barn, och då kan man faktiskt lösa det hela på lite olika sätt – kravet är ju att det resulterande trädet fortfarande skall vara ett binärt sökträd. Vi väljer där följande strategi: Den nod A som skall tas bort ersätts med den nod B i A:s högra underträd som har ett minimalt värde. Därefter tas B bort (med `treeDelete` förstås).

Fråga 6) Vilka principiellt olika alternativ finns det egentligen då det väl är dags att ta bort en nod? Rita olika träd och visa vad som måste göras vid en borttagning. Specifikt – övertyga er om och ge ett tydligt exempel på fallet då noden som skall tas bort har två barn.

Om ni betraktar koden för `treeDelete` så är de olika alternativen beskrivna, men ni skall förstås komplettera med koden för genomförandet. För att hantera det sista fallet kan det vara bra att ha en funktion `minTree` som helt enkelt returnerar det minsta värdet i ett träd, så börja med att skriva denna hjälpfunktion.

```
-- value not found
treeDelete v' EmptyBT                       = ??
```

```
-- one descendant
treeDelete v' (NodeBT v lf EmptyBT) | v'==v = ??
treeDelete v' (NodeBT v EmptyBT rt) | v'==v = ??
-- two descendants
treeDelete v' (NodeBT v lf rt)
    | v'<v  = ??
    | v'>v  = ??
    | v'==v = ??
```

Fråga 7) Vid användande av `treeInsert` respektive `treeDelete`, skapas det då nya träd eller modifieras de existerande? Motivera! Koppla gärna detta till ett resonemang om data i Haskell.

Som avslutning på denna del skall ni sedan skriva funktionerna `inorder` (vilken löper igenom trädet i inorder och samlar trädets element i en lista) och `treeSort`, vilken förstås tar in en lista och sorterar den genom att lägga in alla element i ett BinTree och sedan anropa `inorder`.

Fråga 8) Skicka med ett körningsexempel (skärmdump) där ni visar er treeSort.

Del B: AVL-träd

Vi övergår nu till AVL-träd. Som ni säkert inser är i princip de exporterade operationerna de samma som för de binära sökträden. Det som skiljer är istället den interna hantering som måste ske för att garantera att träden hålls balanserade, för att på så sätt uppnå bra prestanda vid insättning och uppslag.

Ladda in filen AVLTree_START.hs. Som synes är här implementation väldigt rudimentär. Datatypen är i alla fall definierad enligt:

```
module AVLTree(AVLTree,emptyAVL) where

data (Ord a,Show a) => AVLTree a = EmptyAVL
                                 | NodeAVL a (AVLTree a) (AVLTree a)
    deriving Show
```

Från det här ser vi att vi alltså valt att **inte** lagra de olika delträdens höjd i noderna, vilket kanske annars är det vanligaste alternativet. Detta innebär att vi istället behöver kunna räkna ut höjden för ett AVL-träd. Skriv därför en funktion `height` med den uppenbara funktionaliteten. Observera att det här är en intern funktion som inte exporteras av modulen.

Vi skall nu ta oss an det som är speciellt för AVL-träd nämligen rotationerna. Längst ner i filen finns fyra minimala träd definierade, vilka alla fyra kräver olika rotationer för att åter bli balanserade.

Fråga 9) Rita upp de fyra träden och visa vilka rotationer som skall genomföras, samt resultatet av dessa.

När ni nu skall koda de fyra rotationerna är det bra att utgå från de fyra situationerna som representeras av respektive träd. Observera att dessa fyra funktioner bara skall utföra rotationen, inte kontrollera om de behövs. Notera även att de dubbla rotationerna faktiskt utgörs av en enkel vänsterrotation och en enkel högerrotation. TIPS: Se till att ni har fullständigt klart för er hur dessa rotationer utförs konceptuellt så blir dessa funktioner kanske oväntat korta (en rad för varje!)

Fråga 10) Visa körningsexempel (skärmdump) för de fyra olika rotationerna på respektive träd t1-t4.

Vi övergår därefter till att tillverka funktioner som avgör om ett AVL-träd är obalanserat, och i så fall korrigerar detta genom att anropa rätt rotation. Mer konkret gör vi detta med två funktioner, en som kontrollerar om trädet är vänstertungt och en som avgör om det är högertungt. I filen återfinns den ena av dessa funktioner:

```
fixLeftHeavy at@(NodeAVL a bt@(NodeAVL b bl br) ar)
           | bh - arh < 2 = NodeAVL a bt ar
           | height bl < height br = rotateLeftRight at
```

```
            | otherwise = rotateRight at
            where       arh = height ar
                        bh = height bt
                        ah = height at
```

Fråga 11) Vad innebär tecknet `@` i Hakselkoden? Vad gör kommandot `where`?

Implementera nu funktionen `fixRightHeavy`.

Vi övergår därmed till själva insättningsfunktionen. I filen är mönstermatchningen gjort, så det som återstår är egentligen bara att fylla i rätt saker på högersidan, dvs. hur det nya trädet ser ut då ett nytt element x sätts in med `treeInsert`. Notera särskilt hur de båda balanseringsfunktionerna arbetar på detta träd (resultatet av insättningen).

```
treeInsert x EmptyAVL = NodeAVL x EmptyAVL EmptyAVL
treeInsert x (NodeAVL a al ar)
            | x < a = fixLeftHeavy (??)
            | otherwise = fixRightHeavy (??)
```

Fråga 12) Visa stegvis (skrärmdumpar) en körning med insättning i ordning av följande element 5, 6, 8, 3, 2, 4, 7 (exempel 6.6 i boken).

(Frivillig uppgift) Implementera slutligen `treeDelete x t` dvs. en operation som tar bort elementet x från AVL trädet t. En bra lösning på `treeDelete` ger samtliga gruppens medlemmar bonuspoäng till kursens midterms. Om ni aspirerar på detta skall ni även skicka med en skärmdump som visar när olika element tas bort ur ett AVLträd.

Algoritmer - Laboration 1

Uppgift 2 Sökträd i C

I den här uppgiften skall ni implementera s.k. 2-3 träd i C. Här skall ni skriva all kod själva. Observera att träden skall göras som en ADT – jämför kursen Data- och Programstrukturer.

Först och främst bör ni nu välja representation. Observera att valet av representation inte är helt självklart. Ett designval är hur ni skall hantera det faktum att trädet kommer att innehålla både 2-noder och 3-noder. En annan aspekt är att det kan vara fördelaktigt för det vidare arbetet att även ha en pekare till en nods föräldranod.

Fråga 13) Vilken representation väljer ni för ert träd?

Det är nu dags att utifrån er representation implementera de olika operationerna. För enkelhets skull behöver ni bara implementera `treeInsert`, samt ett bra sätt att skriva ut trädet.

Fråga 14) Visa stegvis med skärmdumpar konstruktionen av ett 2-3 träd från följande sekvens av inputs: 9, 5, 8, 3, 2, 4, 7. (Figur 6.8 i boken)

Den funktinalitet som skall finnas i programmet är en möjlighet att stegvis lägga till noder och skriva ut trädet efter hand – jämför den utdelade exempellösningen. Det skall dessutom även vara möjligt att på ett enkelt sätt lägga in en hel sekvens av värden och sedan skriva ut det resulterande trädet.

Som en frivillig uppgift får ni också implementera `treeDelete`. Även här ger en bra lösning på `treeDelete` samtliga gruppens medlemmar bonuspoäng till kursens midterms, och de grupper som anser sig värda detta skall skicka med ett körningsexempel.

Organisation

- Laborationen genomförs i grupper om 4-6 studenter.
- Handledning sker enbart på schemalagda tider och som drop-in.
- Vid inlämningen skall all källkod samt svar på frågorna (Word-dokument eller pdf) **mailas** till Ulf. Observera att allt skall sättas ihop till **ett** dokument.
- Ett körbart program för 2-3 träden, samt de modifierade Haskell-filerna skall bifogas mailet.
- Inlämningen skall avslutas med gruppens, av samtliga medlemmar underskrivna, försäkran om att ni följt ”SA code of conduct” dvs. att ni utvecklat lösningen själva och att alla i gruppen varit delaktiga.
- Vid rättning kommer framförallt funktionaliteten att bedömas, men faktorer som onödigt komplicerad kod, undermålig struktur etc. kan innebära att laborationen underkänns.
- **Examination 1 är 150504 klockan 09.00.**
- **Examination 2 är 150529 klockan 09.00.**
- **Examination 3 är i augusti.**