UPPSALA UNIVERSITET

Programkonstruktion

Moment 8
Om abstrakta datatyper och binära sökträd

Tabeller

En viktig tillämpning är tabellen – att ifrån en ”nyckel” kunna ta fram ett ”tabellvärde”. Ett typiskt exempel är en telefonkatalog:

Avdelningen för datalogi

Andersson, Arne, professor i datalogi, 1022
Björklund, Henrik, doktorand, 1023
Bol, Roland, universitetslektor, 7606
Eriksson, Lars-Henrik, universitetslektor, 1057
Flener, Pierre, universitetslektor, 1028
Fomkin, Ruslan, doktorand, 1062

Implementering av tabellen

En tabell implementeras naturligen som en lista:

```
val datalogi =
[("Andersson, Arne",("professor i Datalogi",1022)),
 ("Björklund, Henrik",("doktorand",1023)),
 ("Bol, Roland",("universitetslektor",7606)),
 ("Eriksson, Lars-Henrik",("universitetslektor",1057)),
 ("Flener, Pierre",("universitetslektor",1028)),
 ("Fomkin, Ruslan",("doktorand",1062)),
 ......] : (string * (string*int)) list;
```

Listans typ blir alltså `(nyckeltyp*värdetyp) list`
En invariant är att varje nyckel förekommer *högst en gång*.

Operationer på tabellen

- Skapa en tom tabell (`empty`)
- Lägg in en uppgift i tabellen (`update`)
- Hämta ett värde från tabellen (`lookup`)
- Tag bort en uppgift från tabellen (`remove`)

Exempel:

```
lookup("Bol, Roland",datalogi) =
                SOME ("universitetslektor",7606)
lookup("Anka, Kalle",datalogi) = NONE
```


Tabellprogrammet – `lookup`

```
(* lookup(key,table)
   TYPE: ''a*(''a*'b) list -> 'b option
   PRE: table är en korrekt tabell.
   POST: SOME v om key finns i table och v är
         motsvarande värde. NONE annars. *)
(* VARIANT: längden hos table *)
fun lookup(_,[]) = NONE
  | lookup(key,(key',value)::rest) =
      if key = key' then
        SOME value
      else
        lookup(key,rest);
```


Tabellprogrammet – `empty, update`

```
val empty = [];

(* update(key,value,table)
   TYPE: ''a*'b*(''a*'b) list -> (''a*'b) list
   PRE: table är en korrekt tabell.
   POST: Tabellen table uppdaterad med värdet
         value för nyckeln key *)
(* VARIANT: längden av table *)
fun update(key, value, []) = [(key,value)]
  | update(key, value, (key',value')::rest) =
      if key = key' then
        (key,value)::rest
      else
        (key',value')::update(key,value,rest);
```

Observera att `update` respekterar datastrukturinvarianten – att varje nyckel förekommer högst en gång i tabellen.

Tabellprogrammet – `remove`

```
(* remove(key,table)
   TYPE: ''a*(''a*'b) list -> (''a*'b) list
   PRE: table är en korrekt tabell.
   POST: Tabellen table med uppgiften om nyckeln
         key borttagen *)
(* VARIANT: längden av table *)
fun remove(key, []) = []
  | remove(key, (key',value)::rest) =
      if key = key' then
        rest
      else
        (key',value)::remove(key,rest);
```

Observera att även `remove` respekterar datastrukturinvarianten.

Tabelltyper

Observera att tabellfunktionerna alla blir polymorfa:

```
empty: (''a*'b) list
lookup : ''a*(''a*'b) list -> 'b option
update : ''a*'b*(''a*'b) list -> (''a*'b) list
remove : ''a*(''a*'b) list -> (''a*'b) list
```

Anledningen är att programmet aldrig utför någon *operation* på nycklar eller värden (bara likhetsjämförelser på nycklar) och därför inte bryr sig om vilka typer dessa har.

Tabellerna får alltså typer på formen `(''a*'b) list`.

Exempeltabellen hade typen `(string*(string*int)) list`, vilket är en *instans* av den polymorfa tabelltypen ovan.

Tabellkomplexitet

En tabell med *n* nycklar representeras av en lista med *n* element.

För att slå i tabellen måste man i genomsnitt söka igenom halva listan – dvs *n*/2 rekursiva anrop av `lookup`.

En lista med en miljon nycklar kräver i genomsnitt 500 000 anrop av `lookup` för att göra *en* tabellslagning. Detta är inte bra i praktiken.

Tidsåtgången för `lookup` är alltså proportionell mot antalet nycklar i tabellen. Komplexiteten hos `lookup` är O(*n*).

Samma resonemang gäller `update` och `remove`.

Abstrakta datatyper

Vid användningen av en tabell enligt ovan utförs alla uppgifter av de olika tabellfunktionerna. Ett program som använder tabeller behöver alltså *inte känna till* att en tabell är representerad som en lista.

Vill man ha en annan representation av tabellen (t.ex. för snabbare åtkomst!) så behöver man inte ändra programmet som *använder* tabellen, bara tabellfunktionerna har *samma funktionsspecifikation*! Det är också lättare att se till att invarianten uppfylls eftersom bara tabellfunktionerna kan skapa/ändra tabeller.

En datatyp med denna egenskap kallas för en *abstrakt datatyp* eftersom man har *abstraherat bort* hur datatypen *representeras* (listan i vårt fall) och bara ser på hur den *används* (funktionerna).

Stöd för dataabstraktion i ML

```
abstype datatypdeklaration
with
 deklarationer
end;
```

Konstruktorerna i datatypsdeklarationen blir *osynliga* och *oåtkomliga* utom mellan `with` och `end`.

```
abstype (''a,'b) table = Table of (''a*'b) list
with
  val empty = Table [];
  fun update(key,value,Table table) = ...;
  fun lookup(key,Table table) = ...;
  fun remove(key,Table table) = ...;
end;
```

`empty`, `update` etc. kallas för datatypens *primitiver*.

Olika sorters primitiver

Man brukar skilja på

- Konstruktorer – primitiver som konstruerar nya värden av den abstrakta datatypen (t.ex. `update`)
- Selektorer – primitiver som hämtar delar av värden (t.ex. `lookup`)
- Predikat – primitiver som gör tester eller jämförelser på värden.

Gränserna mellan dessa är dock inte skarp.

Intern dokumentation av abstrakta datatyper

Varje `abstype` skall ha intern dokumentation på samma sätt som en `datatype`.

Dessutom skall alla definitioner av värden (`val`) i den abstrakta datatypen också dokumenteras med
- Namn på värdet
- Värdets typ
- Beskrivning av vad värdet representerar

T.ex.:

```
(* empty
   TYPE: (''a*'b) table
   VALUE: En tom tabell *)
val empty = Table [];
```


Nackdelar med dataabstraktion

- Man kan bara göra sådant som är förutsett.
  Exempel: Om man vill veta om tabellen är *tom*, så går inte det! (En lösning kan vara att lägga till en primitiv som ger tillbaka all information i tabellen i ett bestämt format, t.ex. som en lista.)
- Man kan förlora prestanda i vissa fall.
  Exempel: Om man *vet* att en nyckel inte finns i tabellen så kan man lägga in en ny uppgift för nyckeln i *konstant tid* genom att skriva `(x,y)::table` i stället för `update(x,y,table)`.
- Eftersom konstruktorerna för den abstrakta datatypen inte går att använda (annat än av primitiverna) så kan man inte använda dem i matchning, vilket *kan* ge mer svårlästa program.

Likhetstyper

Abstrakta datatyper i ML är *inte likhetstyper*.
Värden med samma ”beteende” kan ha olika representation!

Exempel:

```
val tab1 = update("A",1,update("B",2,empty));
val tab2 = update("B",2,update("A",1,empty));
```

`tab1` och `tab2` är nu bundna till tabeller med *samma information*. Men `tab1`≠`tab2`, därför att listorna har elementen i *olika ordning*.

För att förhindra misstag *tillåter inte* ML jämförelser mellan värden av abstrakta datatyper (utanför `with` ... `end`).

För att jämföra sådana värden måste man skriva en primitiv för jämförelsen som då kan ta hänsyn till att representationen skiljer.

Sammanfattning av abstraktion

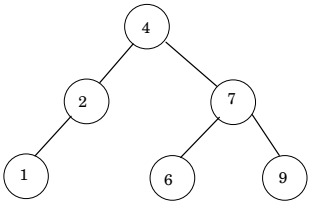
Definitionsabstraktion – ett värde ersätts av ett symboliskt namn
`x < maximum` i stället för `x < 100`
(vitsen är bättre läsbarhet och att behöver man ändra `maximum` räcker det med att ändra på ett ställe – där `maximum` definieras.)

Funktionsabstraktion – ett uttryck görs oberoende av specifika data
`(fn x => x+1)y` i stället för `y+1`
(vitsen är att `(fn x => x+1)` kan namnges och användas i många olika sammanhang – ofta även bättre läsbarhet.)

Dataabstraktion – ett program görs oberoende av specifik datarepresentation
`update(x,y,table)` i stället för `(x,y)::table`
(vitsen är att man kan förändra representationen utan att ändra programmet, lättare att uppfylla invarianter, ofta bättre läsbarhet.)

Trädrepresentation

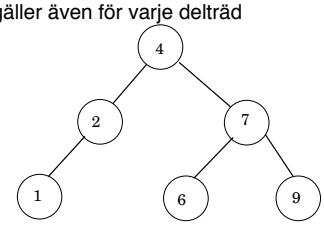
Tabellinformation kan lagras i träd i stället för i listor.



Varje nod innehåller en nyckel och motsvarande värde.
Detta träd representerar en tabell med nycklarna 1, 2, 4, 6, 7 och 9.
(Här och i fortsättningen visas bara nycklarna i figurerna – inte värden.)

Binära sökträd

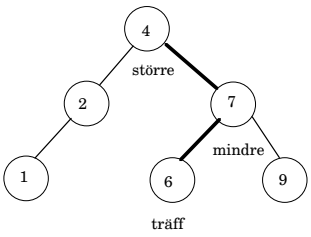
Ett binärt träd kallas för ett *binärt sökträd* om den har invarianten:
- Alla noder i vänster delträd har nycklar med lägre värden än rotnodens nyckel.
- Alla noder i höger delträd har nycklar med högre värden än rotnodens nyckel
- Dessa villkor gäller även för varje delträd



Sökning i binära sökträd

Sök efter nyckeln 6. Starta vid roten.

Vid varje nod väljer man vänster eller höger sökträd beroende på om det man söker efter är mindre eller större än nodens nyckel.



Deklaration av binära sökträd

```
abstype (''a,'b) table =
   Empty
 | Bintree of ''a*'b*(''a,'b) table*(''a,'b) table
(* REPRESENTATION CONVENTION:
     Ett tomt träd representeras av Empty. Varje icketomt
     träd representeras av Bintree(k,v,l,r), där k är
     nyckeln, v är tabellvärdet, l är vänster delträd och r
     är höger delträd.
   REPRESENTATION INVARIANT:
     Alla nycklar i vänster resp. höger delträd
     måste vara mindre resp. större än k  *)
with
  (* empty
     TYPE: (''a*'b) table
     VALUE: Ett tomt träd *)
  val empty = Empty;
  fun update .....;
  fun lookup .....;
  fun remove .....;
  fun toList .....;
end;
```


Funktionen `lookup`

```
(* lookup(key,table)
   TYPE: (Se kommentar nedan)
   PRE:  table är ett sökträd
   POST: SOME v om key finns i table och v är
         motsvarande värde. NONE annars. *)
(* VARIANT: Största djupet hos ett löv i table *)
  fun lookup(key,Empty) = NONE
    | lookup(key,Bintree(key',value,left,right)) =
        if key = key' then
          SOME value
        else if key < key' then
          lookup(key,left)
        else
          lookup(key,right);
```

Man skulle önska att typen hos `lookup` var

`''a*(''a,'b) table -> 'b option`, men i praktiken blir den

`int*(int,'a) table -> 'a option` (se bild 25)

Körexempel

```
lookup(6,Bintree(4,"v4",
                 Bintree(2,"v2",
                      Bintree(1,"v1",Empty,Empty),
                      Empty),
                 Bintree(7,"v7",
                      Bintree(6,"v6",Empty,Empty),
                      Bintree(9,"v9",Empty,Empty))))
—> if 6 = 4 then
     SOME "v4"
   else if 6 < 4 then
     lookup(6,Bintree(2,"v2",
                      Bintree(1,"v1",Empty,Empty),
                      Empty))
   else
     lookup(6,Bintree(7,"v7",
                      Bintree(6,"v6",Empty,Empty),
                      Bintree(9,"v9",Empty,Empty)))
```


Körexempel (forts.)

```
—> lookup(6,Bintree(7,"v7",
                    Bintree(6,"v6",Empty,Empty),
                    Bintree(9,"v9",Empty,Empty)))
—> if 6 = 7 then
     SOME "v7"
   else if 6 < 7 then
     lookup(6,Bintree(6,"v6",Empty,Empty))
   else
     lookup(6,Bintree(9,"v9",Empty,Empty))
—> if 6 = 6 then
     SOME "v6"
   else if 6 < 6 then
     lookup(6,Empty)
   else
     lookup(6,Empty)
—> SOME "v6"
```


Ett problem med typerna

Läser man in programmet för binära sökträd i ML så ser man att typen för t.ex. `lookup` blir

```
int * (int, 'a) table -> 'a option
```

...alltså *inte* polymorf i nyckeltypen!

Anledningen är att ML måste storleksjämföra nycklar och därför känna till deras typ. Vårt program fungerar bara med en bestämd nyckeltyp. (Inte nödvändigtvis `int` – alla typer som kan storleksjämföras, t.ex. `string`, kan också användas.)

Detta problem kan man lösa med *högre ordningens funktioner* (kursmoment 9).

Tidkomplexitet för binära sökträd

Om alla ”nivåer” i trädet är fyllda så innehåller ett träd med *n* nivåer 2^n-1 noder. För att söka efter en nyckel behöver man bara besöka så många noder som motsvarar antalet nivåer.

Antalet rekursiva anrop för ett träd 2^n-1 noder är alltså *n*.

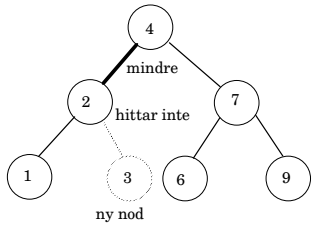
Tidsåtgången är proportionell mot logaritmen av antalet nycklar – O(log *n*).

Ett träd med en miljon nycklar kräver c:a 20 rekursiva anrop av `lookup` – jämför med 500 000 som krävdes när tabellen representerades som en lista.

Detta förutsätter att trädet är ”välbalanserat” – mera senare...

Uppdatering av binära sökträd

För att sätta in nyckeln 3: Sök efter den och sätt in den där den borde ha funnits:

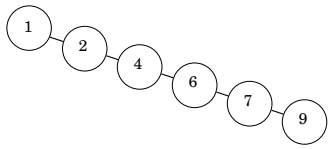


Funktionen `update`

```
(* update(key,value,table)
   TYPE: ''a*'b*(''a,'b) table -> (''a,'b) table
   PRE:  table är ett sökträd
   POST: Tabellen table uppdaterad med värdet
         value för nyckeln key *)
(* VARIANT: största djupet av ett löv i table *)
  fun update(key,value,Empty) =
        Bintree(key,value,Empty,Empty)
    | update(key,value,
             Bintree(key',value',left,right)) =
        if key = key' then
          Bintree(key,value,left,right)
        else if key < key' then
          Bintree(key',value',
                  update(key,value,left),right)
        else
          Bintree(key',value',
                  left,update(key,value,right));
```


Obalans

Sätter man i tur och ordning in 1, 2, 4, 6, 7 och 9 i ett tomt sökträd så får man:



Detta är ett kraftigt *obalanserat* träd. I praktiken en lista, vilket ger samma tidsåtgång för sökning som i en lista – linjärt i storleken.

För att binära sökträd skall vara praktiskt användbara måste de *balanseras om* efter uppdateringar.

Balanserade sökträd behandlas i nästa kurs (AD1).

Funktionen `toList`

Ibland vill man hämta ut informationen i ett träd som en lista.

```
(* toList table
   TYPE: (''a,'b) table-> (''a*'b) list
   PRE: (inget)
   POST: En lista av alla nyckel-värde-par i
         tabellen i inorder. *)
(* VARIANT: Största djupet av något löv i table *)
  fun toList Empty = []
    | toList(Bintree(key,value,left,right)) =
        toList left @ (key,value) :: toList right;
```

`toList` går igenom (*traverserar*) hela trädet för att samla information från noderna och bygga en lista.

I detta fall ”besöker” man vänster delträd först, sedan rotnoden, sedan höger delträd – *inorder*-traversering.

Mer om traversering

`toList` ”besöker” vänster delträd först, sedan rotnoden, sedan höger delträd – *inorder*-traversering.

Notera att listan som `toList` returnerar är *sorterad*. Genom att bygga ett sökträd med en mängd data och sedan göra om det till en lista i inorder så har man *sorterat* datamängden!

preorder-traversering – man besöker först rotnoden, sedan vänster delträd, sedan höger.

postorder-traversering – man besöker först vänster delträd, sedan höger, sedan rotnoden.