Föreläsning 11

Mängd, lexikon och
binära sökträd

Innehåll

- Mängd
- Lexikon
- Binära sökträd

Mängd

- Modell: En påse, men den är inte riktigt bra eftersom man tex kan ha mängder med gemensamma element.
- Organisation:
  - En *oordnad* samling av element som är av samma typ.
  - Grundmängden behöver inte vara ändlig (heltal) men dataobjekten är ändliga.
  - Kan inte innehålla två likadana värden.

Specifikation

```
abstract datatype Set(val)
   Empty()->Set(val)
   Single(v:val)->Set(val)
   Insert(v:val,s:Set(val))->Set(val)
   Union(s:Set(val),t:Set(val))->Set(val)
   Intersection(s:Set(val),t:Set(val))->Set(val)
   Difference(s:Set(val),t:Set(val))->Set(val)
   Isempty(s:Set(val))->Bool
   Member-of(v:val,s:Set(val))->Bool
   Choose(s:Set(val))->val
   Remove(v:val,s:Set(val))->Set(val)
   Equal(s:Set(val),t:Set(val))->Bool
   Subset(s:Set(val),t:Set(val))->Bool
```


Specifikation

- Alla metoder som finns i boken behövs inte.
  - `Empty, IsEmpty, Insert, Choose` och `Remove` skulle räcka.
  - Man har tagit med de vanliga matematiska mängdoperationerna.
- Vill man kunna ha mängder som förutom värden kan innehålla mängder av dessa värden (vanligt i matematiken) måste man skapa en ny datatyp generaliserad mängd.

Konstruktion av mängd

- Nästan oavsett konstruktion måste man kunna hantera det faktum att det inte får finnas dubbletter i en mängd.
- Mängd som lista har två alternativ:
  - Se till att listan inte har dubbletter (krav på `Set-Insert` och `Union`)
  - Låt listan ha dubbletter (krav på `Equal, Remove, Intersection, Difference`)

Konstruktion av mängd som lista

- Komplexitet:
  - Metoder som kräver sökningar i listan O(n)
  - Binära mängdoperationerna kräver (pga nästlad iteration) O(m*n)
  - Listan kan konstrueras på olika sätt. Sorterad lista tex är effektivare för de binära mängdoperationerna.

+ Grundmängden kan vara mycket stor

+ Delmängderna kan både vara mycket små och mycket stora utan utrymmesförluster (om man implementerar listan på rätt sätt)

- Flera operationer blir långsamma jämfört med andra val.

Konstruktion av mängd som bitvektor

- En bitvektor är en vektor med elementvärden av typen Bit = {0,1}
  - Ibland tolkas 0 och 1 som falskt resp. sant, dvs Bit identifieras som datatypen Boolean
- Grundmängden måste ha en diskret linjär ordning av elementen. (Man kan numrera dem.)
- Bit *k* i bitvektorn motsvarar det *k*:te elementet i grundmängden. Biten är 1 om elementet ingår i mängden.

Konstruktion av mängd som bitvektor

- Om bitvektorn finns implementerad som ett eller flera ord i datorns primärminne kan man utnyttja maskinoperationer.
  - Dvs man gör operationen *samtidigt* på alla element i vektorn. Detta gör många metoder effektiva.

+ Relativt effektiv i allmänhet, mycket effektiv som ovan.

- Grundmängden måste vara ändlig och i praktiken rätt så liten. Reserverar minne proportionellt mot grundmängdens storlek.

- Om man har många små mängder i en tillämpning blir det dålig effektivitet i minnesutnyttjandet.

Mängd konstruerad som boolesk funktion

- En mängd kan representeras av sin karakteristiska funktion $\chi_s(x)$:

$$\chi_s(x) = \begin{cases} 1 \text{ om } x \in S \\ 0 \text{ annars} \end{cases}$$

Lexikon

- En förenklad mängd där man tagit bort union, intersection, difference, subset och equal.
- Kvar är det man behöver för att kunna hålla ordning på en samling objekt:
  - empty
  - isempty(s)
  - insert(v, s)
  - remove(v, s)
  - memberOf(v, s)

Lexikon

- Ett lexikon är som en tabell utan tabellvärden.
- Saknar alltså metoder för att kombinera lexikon till nya lexikon.
- Lexikon är en *solitär* datatyp. Man kan bara göra modifieringar av isolerade dataobjekt.
- En icke-solitär datatyp har stöd för att kombinera/bearbeta två eller flera dataobjekt.

Konstruktion av lexikon

- Vi kan få betydligt effektivare konstruktioner när antalet och typen av metoder är begränsade.
- Konstruktioner av lexikon:
  - Lexikon som Hashtabell (har vi mer eller mindre redan tittat på)
  - Lexikon som Binärt sökträd
  - Lexikon som Trie (nästa föreläsning)
- Eftersom Lexikon är så likt Tabell kan dessa konstruktioner med små ändringar bli effektiva konstruktioner av en tabell.

Binärt sökträd

- Används för sökning i linjära samlingar av dataobjekt, specifikt för att konstruera tabeller och lexikon.
- Organisation:
  - Ett binärt träd som är sorterat med avseende på en sorteringsordning R av etikett-typen så att
    - I varje nod N gäller att alla etiketter i vänster delträd går före N som i sin tur går före alla etiketter i höger delträd.
    - Alla noder är definierade.

Informell specifikation

- Skiljer sig från ett vanligt binärt träd:
  - Alla noder måste ha etiketter.
    - Ta bort Create, Has-Label och Set-Label och inför Make som skapar rotnod med värde.
    - Insert-operationerna utökas med ett etikettvärde.
  - Man ska kunna ta bort inre noder också, inte bara löv.
    - Positionsparametern i Delete-node behöver inte peka på ett löv.
    - När man rycker bort en inre nod slits trädet sönder. Hur lagar man det?
  - Är nedåtriktat
    - Parent kan utelämnas.
  - Eftersom trädet är sorterat kan man inte få stoppa in ett nytt element vart som helst.
    - Måste uppfylla sorteringsordningen.

Varför sorterat träd?

- Det går snabbare att söka i strukturen…
- För binärt sökträd:
  - Kolla om det sökta värdet finns i den aktuella noden.
  - Om inte sök rekursivt nedåt i vänster eller höger delträd beroende på om det sökta elementet kommer före eller efter nodens värde i sorteringsordningen.
- Om det binära trädet är komplett:
  - Värstafallskomplexitet för sökning O(log n) för ett träd med n noder.
  - Hur ser man till att trädet blir komplett vid insättning?

Borttagning av nod i binärt sökträd

- Hur lagar man ett träd när man tar bort en nod mitt i?
  - Om den borttagna noden bara hade ett delträd:
    - Lyft upp det ett snäpp
  - Om den borttagna noden hade två delträd:
    - Välj noden med det minsta värdet i höger delträd som ersättning (alternativt noden med största värdet i vänster delträd).
- Detta är standardkonstruktionen, är upp till den som implementerar trädet.
  - De vanligaste tillämpningarna är inte beroende av denna detalj.
  - Viktigt att visar sitt beslut i specifikation och dokumentation.

Varför inte ändra gränsytan?

- Eftersom man inte får sätta in ett nytt element vart som helst så kanske man lika gärna kan ersätta insert-left och insert-right med en metod place som automatiskt placerar det rätt?
- På samma sätt ersätta delete-node(pos, bt) med remove(val, bt)?
- Bägge dessa metoder ligger på en högre abstraktionsnivå än övriga metoder i gränsytan.
  - Place implementerar man i huvudsak med hjälp av andra metoder i gränsytan vilket är lite märkligt.
  - Strukturen döljs (för oss) och blir mer lik en mängd.

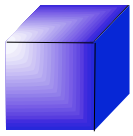
Tillämpningar av Binärt sökträd

- Framför allt till konstruktioner av Lexikon och Tabell.
- Notera att inorder-traversering av binärt sökträd ger en sorterad sekvens av de ingående elementen.
  - Kan alltså sortera element genom att stoppa in dem ett och ett i ett tomt Binärt sökträd och sedan traversera det.

Generaliseringar

C	N	R	S	T	U	Y

- Ett binärt sökträd underlättar sökning i en en-dimensionell datamängd.
- Lätt att generalisera detta till sökning i en 2-dimensionell datamängd (quadtree) eller så hög dimension man önskar (tex octtree).



Quadtree (Fyrträd)

- Organiserat som ett binärt träd med förgreningsfaktor 4 istället för 2.
- Tolkning (vanligast):
  - Rotnoden delar in den givna ytan (oftast kvadrat) i fyra lika stora kvadrater. Vart och ett av de fyra barnen delar i sin tur sin kvadrat i fyra osv. Inga koordinater behöver lagras i inre noder.
- Kan användas för att lagra lägesinformation för punktformiga grafiska objekt.

Quadtree – forts.

- Man kan också använda det för att representera kurvor och ytor.
  - Svarta kvadranter – fylls helt av objektet
  - Grå kvadranter – fylls delvis av objektet
  - Vita kvadranter – innehåller inte objektet
- Exempel på tillämpning
  - GIS
  - qtdemo i Matlab på peppar…