F14 - Generella teorier

5DV149, 5DV150 Datastrukturer och algoritmer
Kapitel 9

Niclas Börlin
niclas.borlin@cs.umu.se

2020-03-04 Wed

Innehåll

- Abstrakta datatyper
  - `Lista`, `Cell`, `Fält`, `Tabell`, `Stack`, `Kö`, `Träd` (ordnat), `Graf`, `Mängd`, `Lexikon`, `Prioritetskö`, `Heap`, `Trie`, `Binärt sökträd`, `Relation`, `...`
- Vad finns det för generella teorier?
- Vad måste vi ta hänsyn till om vi vill göra en ny datatyp?
  - Exempel på datatyper att fundera på:
    - Kurs
    - Dokument
    - Arkiv
    - Modell
    - Läkarundersökning

Abstrakta datatyper (ADT)

- Ett koncept för att kunna diskutera och jämföra olika typer av datastrukturer.
- Hög abstraktionsnivå:
  - Intresserad av struktur och organisation, inte implementation.
- Operationerna ger datatypen *karaktär* och specifikationen visar datatypens *uttrycksfullhet*.
- Operationerna kan indelas i olika kategorier:
  - Konstruktorer
  - Inspektorer
  - Modifikatorer
  - Navigatorer
  - Komparatorer

Konstruktorer

- Skapar/bygger upp och returnerar ett objekt av aktuell ADT.
  - *Grundkonstruktorer* saknar argument av aktuell ADT:
    - `Empty()`
    - `Make()`
    - `Create()`
  - *Vidareutvecklande* konstruktorer tar *ett* argument av aktuell ADT:
    - `List-Insert(val, pos, List)`
    - `Stack-Push(val, Stack).`
  - *Kombinerande* konstruktorer tar *flera* argument av aktuell ADT:
    - `Set-Union(Set, Set)`

Inspektorer

- Undersöker ett objekts inre uppbyggnad:
  - Avläser eller sonderar elementvärden eller strukturella egenskaper:
    - `Inspect-value`
    - `Stack-Top`
    - `Table-Lookup`
    - `Set-Choose`
  - Test av olika extremfall för struktur eller värden:
    - `Binary-Tree-Has-Left-Child`
    - `Set-Member-Of`
    - `Isempty`
  - Mäter objekt:
    - `Array-Has-Value`

Modifikatorer

- Ändrar ett objekts struktur och/eller elementvärden:
  - Insättning, borttagning, tilldelning, omstrukturering:
    - `Array-Set-Value`
    - `Table-Remove`
    - `Stack-Push`
    - `Stack-Pop`
    - `Set-Insert`
- En del operationer kan räknas både som konstruktor och modifikator
  - `Stack-Push`

Navigatorer

- Används för att orientera sig i ett objekts struktur:
  - Landmärken (kända positioner), lokala förflyttningar, traverseringar, etc.
    - `List-First`
    - `List-End`
    - `List-Next`
    - `Binary-Tree-Left-Child`

Komparatorer

- Jämför objekt av aktuell ADT med varandra:
  - `Link-Equal`
  - `Set-Subset`

Gränsytor

- Vilka kategorier har följande operationer?
  - Tabell

```
abstract datatype  Table(arg,val)
      Empty() → Table(arg,val)
      Insert(x: arg,y: val,t: Table(arg,val))
                                  → Table(arg,val)
      Isempty (t: Table(arg,val)) → Bool
      Lookup (x: arg, t: Table(arg,val))
                                  → (Bool,val)
      Remove (x: arg, t: Table(arg,val))
                                  → Table(arg,val)
```

  - Riktad lista

```
abstract datatype DList(val)
auxiliary pos
  Empty() → DList(val)
  Isempty (l: DList(val)) → Bool
  First (l: DList(val)) → pos
  Next (p: pos, l: DList(val)) → pos
  Isend (p: pos, l: DList(val)) → Bool
  Inspect (p: pos, l: DList(val)) → val
  Insert(v: val, p: pos, l: DList(val))
                              → (DList(val),pos)
  Remove(p: pos, l: DList(val)) → (DList(val),pos)
```

Uttrycksfullhet

- Gränsytans specifikation visar datatypens *uttrycksfullhet*.
  - Vad kan jag göra med objekten?
- Frågor att fundera kring vid skapandet av en ADT:
  - Vilken är värdemängden?
    - Vilken typ av värden vill jag lagra?
  - Vilka interna resp. externa egenskaper har objekten?
  - Vad vill man göra med objekten?
  - Specificera en gränsyta informellt och formellt.
  - Överväg olika implementationsmöjligheter.

Uttrycksfullhet

- Datatypsspecifikationen har två roller:
  - Beskriva datatypens *egenskaper*.
  - Fungerar som en *regelsamling* för användningen av datatypen.
- Specifikationens uttrycksfullhet kan mätas med tre begrepp:

  Nivå 0: (Objektfullständighet) Vi vill kunna skilja olika objekt åt.

  Nivå 1: (Algoritmfullständighet) Vi vill kunna jämföra objekt.

  Nivå 2: (Rik gränsyta) Vi vill kunna kopiera objekt effektivt.

Nivå 0: Objektfullständighet (1)

- Det ska vara möjligt att *konstruera* och *skilja* mellan *alla* objekt som hör till datatypen.
- Vi måste kunna inspektera allt som vi stoppar in i datatypen.
- Vi måste kunna skilja objekt åt om vi vet hur dom borde skilja sig åt.

Nivå 0: Objektfullständighet (2)

- Låt I stå för en inspektor och O_i för alla andra operationer.
- För att skilja mellan två objekt A och B måste det för alla A och B existera en sekvens av operationer

$$I \circ O_1 \circ O_2 \cdots O_n, n \geq 0,$$

som ger olika resultat om A och B är olika.

- Exempel:
  - `Front` $\circ$ `Dequeue` $\circ$ `Dequeue` går också att skriva `Front(Dequeue(Dequeue(q)))` och skulle ge olika resultat på köerna

$$q_1 = (1, 4, 9),$$
$$q_2 = (1, 4, 10).$$

Nivå 0, exempel

- Datatypen `Student` med konstruktorn `Create(name, address)`, men som enda inspektor `Inspect-Name` uppfyller inte nivå 0.
- En stack-gränsyta med endast funktionerna `Empty`, `Push` och `Max` (största värdet i stacken) uppfyller inte nivå 0.
  - Kan inte skilja på

$$A = (1, 28),$$
$$B = (5, 20, 28).$$

Nivå 0, exempel

- En tabell-gränsyta med endast funktionerna `Empty`, `Insert` och `Max` (största tabellvärdet) uppfyller inte nivå 0.
  - Kan inte skilja på

    A =((Rosor, 46), (Krysantemum, 28)).
    B =((Tussilago, 46), (Persilja, 15)).
- En tabell-gränsyta med endast funktionerna `Empty`, `Insert` och `Lookup` uppfyller däremot nivå 0.
  - För *A* och *B* ovan så ger `Lookup(A,Rosor)` annat resultat än `Lookup(B,Rosor)`.

Nivå 1: Algoritmfullständighet (*Expressive completeness*)

- Starkare än Nivå 0: objektfullständighet.
- Man ska kunna implementera *alla algoritmer* i denna datatyp:
  - Allt som man kan göra med datatypen ska också gå att implementera med specifikationens operatorer.
- Räcker med att visa att man kan implementera ett *likhetstest* mellan två dataobjekt med hjälp av operationerna.
  - Nivå 1 = Nivå 0 + likhetstest.
- Alltså: Det ska gå att skilja två olika objekt åt även om man inte vet vilka skillnader man letar efter.

Nivå 1, frågor

- Uppfyller `Queue`: {`Empty`, `Enqueue`, `Front`, `Dequeue`, `Isempty`} nivå 1?
- Uppfyller `Table`: {`Empty`, `Insert`, `Lookup`} nivå 1?

Fel i boken! (1)

- Boken påstår att `Table`: {`Empty`, `Insert`, `Lookup`} uppfyller nivå 1 (är algoritmfullständig).
- Detta är sant endast om vi gör ett antagande angående nyckeltypen, vilket?

Fel i boken! (2)

- Om vi antar att nyckeltypen är *ändlig* och *uppräkningsbar* så uppfyller `Table`: {`Empty`, `Insert`, `Lookup`} nivå 1.
- Hur ser algoritmen för likhetstestet ut?

Likhetstest, tabell

`Algorithm isEqual(A, B: table(key,val))`

- **for each** possible value `x` **in** key type
  - **if not** `lookup(A)=lookup(B)` **then**
    - **return** `false`
- **return** `true`

Nivå 2: Rik gränsyta (*Expressive richness*)

- Även om man uppfyller nivå 1 så kan vissa algoritmer bli hopplöst ineffektiva.
- Krav för nivå 2: Man ska med hjälp av gränsytan kunna implementera speciella analysfunktioner (*distinguished functions*) som uppfyller följande:
  - Objektet ska kunna *rekonstrueras* både till värde och struktur med enbart komposition av analysfunktionerna.
  - Analysfunktionerna får varken innehålla iteration eller rekursion i sin definition.
- Nivå 2: Nivå 1 + "effektiv kopiering".

Nivå 2, frågor (1)

- Uppfyller `Queue`: {`Isempty`, `Front`, `Dequeue`} nivå 2?
  - I så fall, hur ser algoritmen för effektiv kopiering ut?

Nivå 2, frågor (2)

- Uppfyller `Queue`: {`Isempty`, `Front`, `Dequeue`, `Empty`, `Enqueue`} nivå 2?
  - I så fall, hur ser algoritmen för effektiv kopiering ut?

Effektiv algoritm, kopiering kö

```
Algorithm Copy(q: queue)
// Returns the original queue and a copy of it
```

- `r <- Empty() // "Original"`
- `s <- Empty() // "Copy"`
- **while not** `Isempty(q)` **do**
  - `v <- Front(q)`
  - `q <- Dequeue(q)`
  - `r <- Enqueue(r, v)`
  - `s <- Enqueue(s, v)`
- **return** `(r,s)`

Nivå 2, frågor (3)

- Antag `arg` är en ändlig, uppräkningsbar datatyp.
- Uppfyller `Table(arg,val)`: {`Empty`, `Isempty`, `Insert`, `Remove`, `Lookup`} nivå 2?
  - I så fall, hur ser den effektiva kopieringsalgoritmen ut?

Nivå 2, frågor (4)

- Antag `arg` är en ändlig, uppräkningsbar datatyp.
- Antag `Choose` är en funktion som tar en tabell och returnerar ett godtyckligt nyckelvärde-tabellvärde-par från en icke-tom tabell.
- Uppfyller `Table(arg,val)`: {`Empty`, `Isempty`, `Insert`, `Remove`, `Lookup`, `Choose`} nivå 2?
  - I så fall, hur ser den effektiva kopieringsalgoritmen ut?

Effektiv algoritm, kopiering tabell

```
Algorithm Copy(t: table)
// Returns the original table and a copy of it
```

- `r <- Empty() // "Original"`
- `s <- Empty() // "Copy"`
- **while not** `Isempty(t)` **do**
  - `(k,v) <- Choose(t)`
  - `r <- Insert(r, k, v)`
  - `s <- Insert(s, k, v)`
  - `t <- Remove(t, v)`
- **return** `(r,s)`

Praktisk uttrycksfullhet

- Vi har teoretiska mått på uttrycksfullhet: Nivå 0-2.
- Måste man alltid uppfylla nivå 2?
  - Finns det tillfällen då man kan nöja sig med nivå 1 (eller 0)?
- Räcker det med att uppfylla nivå 2?
- Hur skapar man en gränsyta in praktiken?

Att utforma en gränsyta till en ADT (1)

- Utgå från den mentala modellen:
  - Vilka *data* vill vi kunna lagra i objektet?
  - Vad vill vi kunna *göra* med objektet?
- Applicerar de teoretiska begreppen.
  - Vill vi kunna skilja mellan objekt (nivå 1)?
  - Vill vi kunna kopiera objekt (nivå 2)?
- Exempel på datatyper:
  - Kurs
  - Dokument
  - Arkiv
  - Modell
  - Läkarundersökning

Att utforma en gränsyta till en ADT (2)

- Ofta blir målet att:
  - Uppfylla nivå 0 (annars kan vi inte plocka ut allt data vi stoppar in).
  - Uppfylla nivå 1 (annars finns det algoritmer som inte kan implementeras)
  - Operationerna är *primitiva*, dvs. inte kan implementeras av övrigare, enklare operationer i gränsytan.
  - Operationerna är *oberoende*, dvs. nivå 1 uppfylls inte om någon operation tas bort.
- Detta ger en stram yta med få operationer.
- Om vi får dålig prestanda kan vi alltid senare definiera extra operationer utifrån operationerna i grundgränsytan.

Specifikation för mängd

- Vilka funktioner behövs för en stram gränsyta?

```
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
```

Fördelar med en stram gränsyta

- Utbytbarhet (mellan implementationer):
  - Man kan börja med enkla implementationer och sedan byta ut mot allt effektivare.
- Portabilitet (mellan miljöer):
  - Mindre problem att flytta en datatyp med få operationer.
- Integritet (mot komplicerande/saboterande tillägg):
  - Mindre risk för att operationer läggs till som
    - strider mot grundidén med datatypen,
    - bara fungerar med aktuell implementation,
    - saboterar för andra operationer.
    - *One datatype to rule them all*
    - *Kan-vara-bra-att-ha-sjukan*

Programspråksstöd för ADTs

- Många språk ger mycket litet eller inget stöd alls. Då krävs:
  - Konventioner
    - Namngivning
    - Operationsval
  - God dokumentation av olika val som görs.
  - Disciplin
    - Inte gå in och peta i interna strukturer!

Blandat

Gränsyta för Kö, inkl. *Length*

```
abstract datatype  Queue(val)
  Empty() → Queue(val)
  Enqueue(v: val, q: Queue(val)) → Queue(val)
  Front (q: Queue(val)) → val
  Dequeue (q: Queue(val)) → Queue(val)
  Isempty (q: Queue(val)) → Bool
  Length (q: Queue(val)) → (int, Queue(val))
```

Pseudokod för Length(q)

```
Algorithm Length(q)
    Compute the length of a queue.
    Returns length and an identical queue.

r ← Empty() // This will be the replicated queue.
len ← 0
while not Isempty(q) do
    Copy first item in queue from q to r.
    r ← Enqueue(Front(q),r)
    Remove from q.
    q ← Dequeue(q)
    Increase counter.
    len ← len + 1
end
Return length and the replicated queue.
return (len, r)
```

Stack konstruerad som riktad lista

- Formulera operationerna i datatypen `Stack` med hjälp av operationerna i `Riktad lista`:

```
abstract datatype DList(val)
auxiliary pos
  Empty() → DList(val)
  Isempty (l: DList(val)) → Bool
  First (l: DList(val)) → pos
  Next (p: pos, l: DList(val)) → pos
  Isend (p: pos, l: DList(val)) → Bool
  Inspect (p: pos, l: DList(val)) → val
  Insert(v: val, p: pos, l: DList(val))
                                  → (DList(val),pos)
  Remove(p: pos, l: DList(val)) → (DList(val),pos)
```