Datastrukturer och algoritmer

Föreläsning 15

Innehåll

- Generella teorier
  - Abstrakta datatyper
  - Algoritmer

Innehåll

- Abstrakta datatyper
  - Lista, Cell, Fält, Tabell, Stack, Kö, Träd (ordnade och binära), Graf, Mängd, Lexikon, Prioritetskö, Heap, Trie, Binärt sökträd, AVL-träd, B-tree, Relation…
- Algoritmdesign
  - Traversering, Sökning, Sortering, Maximalt flöde, Minimalt uppspännande träd, Kortaste vägen etc
  - Tids- och rumskomplexitet
- Vad finns det för generella teorier?

Abstrakta datatyper

- Ett koncept för att kunna diskutera och jämföra olika typer av datastrukturer.
- Ligger på en hög abstraktionsnivå.
  - Främst intresserad av struktur och organisation, inte implementation.
- Operationerna ger datatypen karaktär och specifikationen visar datatypens uttrycksfullhet.

Operationskategorier (l)

- *Konstruktorer* - skapar/bygger upp och returnerar ett objekt av aktuell ADT
  - Grundkonstruktorer som saknar argument av den aktuella ADT:n
    - Empty, Make, Create
  - Vidareutvecklande konstruktorer som tar ett argument av den aktuella ADT:n
    - List-Insert, Stack-Push
  - Kombinerande konstruktorer som tar flera argument av den aktuella ADT:n
    - Set-Union

Operationskategorier (2)

- *Inspektorer* - Undersöker ett objekts inre uppbyggnad på olika sätt
  - Avläsning eller sondering av elementvärden eller strukturella förhållanden
    - Inspect-value, Stack-Top, Table-Lookup, Set-Choose
  - Test av olika extremfall av struktur och värden
    - Binary-tree-has-left-child, Set-member-of
  - Mätning av objekt
    - Isempty, Has-value

Operationskategorier (3)

- *Modifikatorer* – Ändrar ett objekt struktur och/eller elementvärden
  - Insättning, borttagning, tilldelning omstrukturering
    - Array-Set-Value, Table-Remove, Stack-Pop, Set-Insert
- *Navigatorer* – Används för att ta fram ett objekts struktur
  - Landmärken (kända positioner), lokala förflyttningar, traverseringar
    - List-First, List-End, List-next, Binary-tree-left-child
- *Komparatorer* – jämför objekt av den aktuella ADTn med varandra
  - Equal, Set-Subset

Uttrycksfullhet

- Abstrakt datatyp = objekt + konstruktion av gränsytan.
- Frågor att fundera kring vid skapande av ADT:
  - Vilken är värdemängden?
  - Vilka interna resp. externa egenskaper har objekten?
  - Vad ska man göra med objekten?
  - Specificera en gränsyta informellt och formellt.
  - Överväga olika implementationsmöjligheter.
- Kan jag göra det jag vill kunna göra med objekten??
  - Uttrycksfullhet

Uttrycksfullhet

- Datatypsspecifikationen har två roller:
  - Slå fast hur datatypen är beskaffad, vilka egenskaper den har.
  - Fungerar som en regelsamling för användningen av datatypen.
- Specifikationens uttrycksfullhet kan mätas med tre begrepp
  - Objektfullständighet
  - Algoritmfullständighet
  - Rik gränsyta

Objektfullständighet

- Är det svagaste kriteriet.
- Det ska vara möjligt att *konstruera* och *skilja* mellan *alla* objekt som anses höra till datatypen.
  - Man ska kunna skilja på två objekt A och B med en sekvens av operationer I $\circ O_1 \circ O_2 \ldots \circ O_n$ ($n \geq 0$)
    (I = Inspektor, O = Operation)
  - Om vi tittar på en tabell
    - Empty, Insert och Max (största definierade tabellvärdet) är inte objektfullständigt. Kan inte skilja på två tabeller med samma max-värde.
    - Empty, Insert och Lookup räcker!

Algoritmfullständighet (Expressive completeness)

- Starkare än (och implicerar) objektfullständighet.
- Man ska kunna *implementera alla* algoritmer i denna datatyp.
  - Dvs allt som man kan göra med datatypen ska också gå att implementera utifrån specifikationens operatorer.
    - Räcker att visa att man kan implementera ett test av likhet mellan två dataobjekt med hjälp av operationerna.
      - Vill man veta varför: Läs artikeln (Kapur, Srivas "Computability and implementability issues in abstract data types").
- Alltså: Algoritmfullständighet = objektfullständighet + likhetstest

Rik gränsyta (Expressive richness)

- Starkaste kriteriet, implicerar både objektfullständighet och algoritmfullständighet.
- Även om man har algoritmfullständighet så kan vissa algoritmer bli hopplöst ineffektiva.
- Krav: Man ska med hjälp av gränsytan kunna implementera speciella *analysfunktioner* som kan
  - Ta fram all information som krävs ur ett dataobjekt för att sedan kunna rekonstruera objektet med enbart komposition av analysfunktionerna.
  - De olika analysfunktionerna får varken innehålla iteration eller rekursion i sin definition.

Rik gränsyta-exempel

- Stack-specifikationen har en rik gränsyta.
  - Isempty kan avgöra om stacken är Empty eller konstruerad som push(x, s) för något x och s. Top ger x och Pop ger s.
  - För vilken stack som helst kan ändliga kompositioner av dessa analysfunktioner
    - plocka ut vart och ett av elementen i stacken
    - hitta strukturen, ordningen på dem
    - utifrån detta kan stacken återskapas

Praktisk uttrycksfullhet

- Vi har teoretiska mått på uttrycksfullhet
  - Objektfullständighet, algoritmfullständighet och rik gränsyta.
- Måste man uppfylla alla tre kraven?
  - Ibland blir en rik gränsyta opraktisk, man saknar vissa operationer.
    - Utskrifter, längdfunktioner eller kopieringsfunktioner tex.
- Hur skapar man en gränsyta?

Att utforma en gränsyta

- Man utgår från de operationer som bidrar till att ge ADTn sin speciella karaktär.
- Sedan applicerar man de teoretiska begreppen. Lägger till vissa operationer och tar bort andra. Målet är att operationerna
  - Ger en objektfullständig gränsyta
  - Är primitiva (kan inte delas upp i mindre operationer)
  - Ger en algoritmfullständig gränsyta
  - Är oberoende, kan inte ta bort en enda operation och ändå ha kvar en algoritmfullständig gränsyta.
- Detta ger en rätt stram yta med få operationer.

Fördelar med en stram gränsyta

- Utbytbarhet
  - Man kan börja med enkla implementationer och sedan byta ut mot allt effektivare.
- Portabilitet
  - Mindre problem att flytta ett program med få op.
- Integritet
  - Mindre risk för att operationer läggs till som strider mot grundidén med ADTn.

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

Design av algoritmer

- Problemlösningsstrategier
  - Top-down
  - Bottom-up
- Typer av algoritmer (lösningstekniker)
  - ”Brute force”
  - Giriga algoritmer (Greedy-algorithms)
  - Söndra och härska (Divide and Conquer)
  - Dynamisk programmering

Brute force

- En rättfram ansats där man utgår direkt från problemställningen och de definitioner som finns där.
- Om problemet är kombinatoriskt så gör man en fullständig sökning
  - Genererar och numrerar alla tänkbara svar/lösningar
  - Välj den bästa lösningen
- Bra metod att starta med
  - Garanterar en korrekt lösning om en sådan finns
  - MEN garanterar inte effektivitet…
  - Ofta väldigt enkla algoritmer

Brute force: Exempel

- Linjär sökning
- Söka det minsta talet i en lista
  - Antar att alla element är det lägsta
  - Kolla alla mot alla
- Handelsresande problemet
  - Besöka alla städer bara en gång
  - På minst kostsamma sätt

Brute force (2)

- Många problem vet man inte av någon bättre lösning än brute force till.
- Ger ofta hög tillväxt på tidskomplexiteten
  - Speciellt för problem där antalet svar ökar snabbt med ökad problemstorlek
- Går ofta att effektivisera de naiva algoritmerna
  - Avbryta en sökning när man nått en lösning
    - Bättre med en lösning än den bästa.
  - Avbryta så fort man inser att vägen inte leder till en lösning

Giriga (Greedy) algoritmer

- METOD:
  - I varje steg titta på alla möjliga steg och välj den för tillfället bästa vägen.
- Bra för optimeringsproblem
  - I många fall får vi optimal lösning med en greedy algoritm.
    - Om den optimala lösningen kan nås via stegvisa lokala förändringar av starten
- Heuristisk metod vs. greedy
  - Garanti för optimal lösning ≈ greedy
- Bra alternativ till brute force algoritmer

Exempel

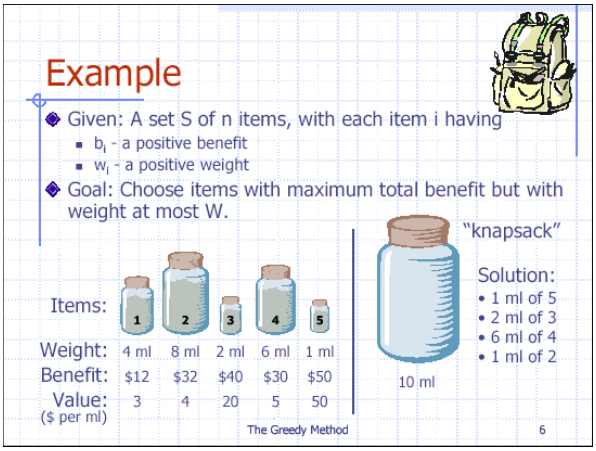
- Att lämna tillbaka växel
  - Minimalt antal mynt i växel
  - Ta alltid det största möjliga myntet i varje ”loop-varv”
- Minimalt uppspänt träd
  - Kruksals algoritm
  - Prims algoritm
- Kortaste vägen i en graf (Dijkstras algoritm)
- Huffman-kodning

”The Fractional Knapsack Problem”

- Givet, en mängd p med n element där element i har värde/förtjänst b_i (benefit) > 0 och en vikt $w_i > 0$
- Mål: Välja element med maximal förtjänst utan att den totala vikten blir mer än den maximala vikten W.
  - I ”The Fractional Knapsack Problem” får man ta bitar av elementen (fractions).
- Låt x_i vara mängden vi tar av element i

  Maximera: $\sum_{i \in S} b_i (x_i / w_i)$ med begränsningen: $\sum_{i \in S} x_i \leq W$

- Regel: För varje gång ta elementet med maximalt värde (förtjänst/vikten).
- O(n log n)



Söndra och härska (Divide and Conquer)

- METOD:
  - Söndra: Dela upp datat/problemet i två eller flera delar som löses rekursivt. Dessa delar bör vara ungefär lika stora.
  - Härska: Konstruera en slutlösning från dellösningarna.
- Leder till rekursiva algoritmer med minst två rekursiva anrop.
  - Kan vara en lösning när det är svårt hitta iterativa lösningar
  - Är ibland effektivare även om det finns iterativ lösning
  - Ibland beräknas en dellösning många gånger (= ineffektivt)
- O(n log n) är vanligt
- Merge-sort och Quick-sort
- Kan ställa krav på implementationen av ADT:n

Exempel: x^n

- Beräkna iterativt eg x*x*x*....*x ger algoritm som är O(n)
- Divide and conquer: vi kan bryta ner problemet och beräkna $x^{ciel(n/2)} * x^{floor(n/2)}$ rekursivt
  - Fast det ger inget :(
  - I de båda rekursiva anropen så beräknar vi i stort sett alltid samma värden... Kan vi utnyttja detta och vinna något?

Dynamisk programmering

- Undviker problemet med söndra och härska, dvs att samma problem kan lösas flera gånger.
- METOD:
  - Lös större och större problem tills man har löst sitt problem av storlek n, och använd information från föregångarna för i varje steg
  - Ställ upp en tabell som håller reda på redan kända lösningar.
    - För varje nytt anrop kollar man om man redan löst det problemet och hämtar det
    - Om inte löser man det och sätter in lösningen i tabellen.

Exempel

- 1-dimensionell
  - k! = fac(n) = 1*2*3*…*n
    - utnyttja att man vet (n-1)! när n! ska beräknas
- Multi-dimensionell dynamisk programmering
  - Matrisbaserad shortest path
  - 0-1 knapsack
    - Där får man alltså inte delar av elementen utan måste ta allt på en gång.

Exempelproblem:

- Du ska skapa ett program som spelar ett spel mot en person med följande regler.
- Det ligger 30 tändstickor på bordet. Spelarna får plocka upp 1, 2, eller 3 tändstickor varje gång och spelet pågår tills den sista stickan plockas upp. Den som tar upp sista stickan förlorar.
- Om datorn alltid får börja, hur ska den se till att den vinner?

Vilken algoritm?

- Brute-force
  - Testa alla varianter och leta fram den bästa…
- Giriga algoritmer
  - Hur vet man i varje steg vilken väg som är den bästa?
- Divide and Conquer
  - Hur ska man dela upp?
- Dynamisk programmering

Dynamisk programmering

- Vi börjar med problemet nedifrån och upp.
  - Målet: Datorns sista drag ska lämna *en* sticka kvar.
  - Hur kan jag hamna i det läget?
    - Om motspelaren startar ett spel med bara 5 stickor:
    - OK, hur få motspelaren att hamna med 5 stickor?
      - Se till att denne hamnar med 9 stickor!
    - Fortsätt tänka ”bottom-up”
      - 1, 5, 9, 13, 17, 21, 25, 29
      - Alltså ska datorn alltid starta med att bara dra en sticka!

Mot-spelare	Dator	Kvar
3	1	1
2	2	1
1	3	1

Mot-spelare	Dator	Kvar
3	1	5
2	2	5
1	3	5