Lösningsförslag och bedömningskommentarer DoA-tentamen

Niclas Börlin

2023-08-23

Contents

1 Uppgift 1) Terminologi (10p)

a) En osorterad lista är ordnad

- **Sant**.
- En lista är alltid ordnad, osorterad eller sorterad.

b) En heterogen datatyp är en sammansatt datatyp där elementen är av olika datatyper

- **Sant**.
- "...kan vara av olika dataatyper..." hade kanske varit ännu tydligare

c) En oriktad graf med en sammanhängande komponent utan cykler är ett träd

- **Sant**.
- En graf med cykler är inget träd
- En graf med >1 sammanhängande komponenter är inget träd

d) I en komplett graf är alla noder grannar till alla andra

- **Sant**.
- Icke att förväxla med ett komplett träd.

e) Det kan finnas fler elementvärden än element i en Lista

- **Falskt**.
- Vi kan ha fler **element** än elementvärden men inte tvärtom. Listan [3,4,3] innehåller 3 element men bara 2 elementvärden.

f) En konstruerad datatyp är också en implementerad datatyp

- **Falskt**.
- En **konstruerad** datatyp kräver en beskrivning av hur den är konstruerad internt, t.ex. att en mängd är lagrad i en lista. Det kräver inte att det finns implementerad kod.

g) En Stack fungerar enligt principen FIFO (First In First Out)

- **Falskt**.
- En Stack fungerar enligt LIFO (Last In First Out).
- En Kö fungerar enligt FIFO.

h) Mängd är en homogen datatyp

- **Sant**.

i) Ett Binärt träd är ett träd där varje nod har max två barn

- **Sant**.

j) Ett fullt binärt träd är välbalanserat

- **Falskt**
- I ett fullt binärt träd har varje nod endera noll eller två barn, vilket uppfylls av en del kraftigt obalanserade träd.

2 Uppgift 2) Pseudokod SetDiff

```
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)
```

2.1 2a) Pseudokod (14p)

Låt datatypen `Mängd` vara konstruerad som en `Riktad lista` med gränsytan till vänster. Skriv en algoritm i pseudokod med huvudet `Set-diff(s, t: DList(val))` som tar två listor `s` och `t` och returnerar mängddifferensen mellan `s` och `t`, dvs. en lista med alla element i `s` som inte finns i `t`. Dubletter är tillåtna i in- och ut-listorna. Efter körningen ska inlistorna `s` och `t` vara oförändrade.

Operatorn ← (<-, vänsterpil, tilldelning, kopiering) är definierad för datatypen `val`. Dessutom finns en funktion `Equal(a, b: val)` definierad som returnerar `True` om elementvärdena `a` och `b` anses vara lika.

Var noggrann med att visa hur du tar hand om alla returvärden som behövs från alla funktioner. Förutom de definierade funktionerna ovan får du bara använda funktioner i gränsytorna till `DList`. För fulla poäng ska lösningen vara modulariserad där lösningen delas in i två eller flera icke-triviala algoritmer.

Det som kommer att bedömas är

- att pseudokoden löser uppgiften under de givna förutsättningarna,
- att pseudokoden är fri från språkspecifika konstruktioner (inga i++ eller for i in range(...), etc.),
- att koden är korrekt indenterad,
- att koden är rimligt kommenterad,
- om koden är modulariserad, och
- om koden har optimal komplexitet ($g(n)$ är viktigast).

2.1.1 Lösningsförslag

```
Algorithm Find(v: val, s: DList)
// Return True if an element with value v is found in s, otherwise False

p ← DList-first(s)
while not DList-isend(p, s) do
  if Equal(v, DList-inspect(p, s)) then
    return True
  p ← DList-next(p, s)

return False


Algorithm Set-diff(s, t: DList)
// Return DList with elements of s not in t.

// Output list
d ← DList-empty()
// Check each position of s
p ← DList-first(s)
while not DList-isend(p, s) do
  v ← DList-inspect(p, s)
  if not Find(v, t) then
    // Value did not appear in t, insert into output
    d ← DList-insert(DList-first(d), v)
  // Advance in s
  p ← DList-next(p, s)

return d
```

2.1.2 Bedömning

De vanligaste sakerna som gett avdrag:

1. Användning av en position från en lista i en annan.
   - Att blanda ihop fält/index och lista/position ger 0p på uppgiften.
2. Om koden inte klarar tomma listor (-2p).
3. Inte ta emot returvärden från t.ex. `insert`.
4. Språkspecifika saker (olika avdrag)
5. Korrekt men icke-modulär lösning ger max 10p.
6. Felaktigt formulerade loopar (olika avdrag).
7. ”Kopiering” av lista med hjälp av tilldelningsoperatorn.
   - Tilldelningsoperatorn är definierad datatypen `val`, men inte för datatypen DList.
8. Saknar indentering (-4)
9. Löser ett annat problem (variabelt avdrag)
   - Några verkar ha antagit att listorna är sorterade.
   - Eller att man ska returnera skillnaden mellan element i respektive listor.

2.2 2b) Komplexitet (3p)

- Vilken tidskomplexitet har din algoritm? Antag att listorna s och t har ungefär lika många element n.
  - Rätt svar: $O(n^2)$.
  - För vart och ett av de (ungefär) n element i `s` krävs en sökning i `t`, dvs. att vi går igenom vart och ett av de (ungefär) n elementen i `t`.
- Bedömning:
  - Poäng på denna uppgift kräver att man lämnat in en lösning på 2a.

2.3 2c) Komplexitet för dublettfri version (6p)

Antag att du känner $g(n)$, c och n_0 för en implementation av `Set-diff` som tillåter dubletter i listorna. Antag vidare att du vet att listorna `s` och `t` alltid kommer att sakna dubletter. Hur kommer det att förändra komplexiteten hos `Set-diff`?

- Svar:
  - $g(n)$ kommer att förbli samma: För varje element i `s` måste vi fortfarande gå igenom varje element i `t`, dvs. $g(n) = n^2$.
  - c kan komma att förändras: Vi måste fortfarande gå igenom **alla** element i `s`. Sökningen i `t` efter ett elementvärde som **inte finns** kommer också att ta lika lång tid då vi måste gå igenom **alla** element i `t`. Skillnaden är att sökning efter elementvärden som **finns** i `t` kommer i genomsnitt att avbrytas snabbare om listan innehåller dubletter. En effektiv implementation avbryter sökningen så fort det **första** element med sökt värde hittats.
  - n_0 kan komma kan förändras: Eftersom c förändras kan n_0 också komma att förändras.
  - Jag anser för övrigt att denna uppgift tillhör de svårare på tentan.

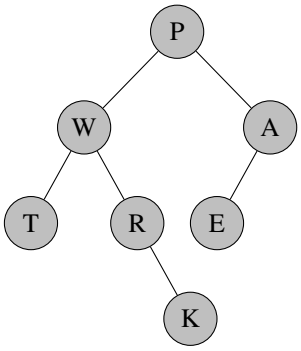
3 Uppgift 3) Operationskategorier Fält (9p)

Ange operationskategori för varje operation i gränsytan till den abstrakta datatypen Fält. Om operationen kan anses tillhöra flera kategorier, välj den längst till höger i tabellen nedan.

	Konstruktor	Inspektor	Modifikator	Navigator	Komparator
`Create(lo,hi)`	X				
`Low(a)`		X		(X)	
`High(a)`		X		(X)	
`Set-value(i, v, a)`			X		
`Has-value(i, a)`		X			
`Inspect-value(i, a)`		X			

Not: Strikt så är `Low` och `High` **inspektorer**, även om de returnerar ett index som används för att navigera i datatypen. Anledningen är att indextypen är **fristående** från fältet. Variabler av indextypen är inte begränsade till att representera positioner som t.ex. returvärdet från `List-first` är.

4 Uppgift 4) Traversering av binärt träd (12p)



Illustrationen till vänster visar ett binärt träd. Nedanstående frågor handlar om traversering av trädet. I svaren, skriv nodernas etiketter, separerade med kommatecken, t.ex. "A,B,C,D,E,F,G".

a) I vilken ordning kommer noderna att besökas om trädet traverseras enligt **bredden-först**?

- Svar: **P, W, A, T, R, E, K**
- Svep uppifrån och ner, från vänster till höger på varje nivå.

b) I vilken ordning kommer noderna att besökas om trädet traverseras. enligt **djupet-först, pre-order**?

- Svar: P, W, T, R, K, A, E
- Placera en streck klockan 9 på varje nod och följ trädets utsida moturs från toppen av roten.

c) I vilken ordning kommer noderna att besökas om trädet traverseras enligt **djupet-först, in-order**?

- Svar: T, W, R, K, P, E, A
- Placera en streck klockan 6 på varje nod och följ trädets utsida moturs från toppen av roten.

d) I vilken ordning kommer noderna att besökas om trädet traverseras enligt **djupet-först, post-order**?

- Svar: T, K, R, W, E, A, P
- Placera en streck klockan 3 på varje nod och följ trädets utsida moturs från toppen av roten.

5 Uppgift 5) Prioritetskö, Hög

5.1 Uppgift 5a) Egenskaper för Hög (4p)

En Prioritetskö kan konstrueras på flera sätt, bl.a. som en **Hög** (Heap), en **Lista** eller en **Sorterad lista**.

- En **Hög** är ett specialfall av en annan datatyp **X**. Vilken?
  - Svar: Ett binärt träd
- Vilka speciella egenskaper måste datatypen **X** ha för att den ska utgöra en **Hög**?
  - Svar: Partiellt sorterad där etiketten för varje nod måste komma före etiketten för var och ett av dess barn enligt en sorteringordning **R**

5.1.1 Bedömning

- För fulla poäng kräver jag att man angett binärt träd och att sorteringsordningen ska vara uppfylld för **varje** förälder-barn-par.
- Det spelar dock ingen roll var man skrivit informationen.

5.2 Uppgift 5b) Komplexitet operationer: Hög (4p)

Ange komplexiteten för nedanstående operationer om Prioritetskön är konstruerad med hjälp av en **Hög**. Operationerna `Inspect-first` och `Delete-first` läser av värdet på respektive tar bort elementet som har högst prioritet i kön. Operationen `Update` används av en del grafalgoritmer efter att ett element i Prioritetskön har förändrats. Resultatet av operationen är att återställa den interna datatypen (Högen) till ett korrekt tillstånd.

	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
`Insert`		X			
`Inspect-first`	X				
`Delete-first`		X			
`Update`		X			

- `Insert`, `Delete-first` och `Update` jobbar alla längs en **gren** från roten till ett löv. Grenens längd är $O(\log n)$ om trädet är välbalanserat.
- `Inspect-first` läser av etiketten i roten, dvs. $O(1)$.

5.3 Uppgift 5c) Komplexitet operationer: Lista (4p)

Ange komplexiteten för nedanstående operationer om Prioritetskön är konstruerad med hjälp av en **Lista**. Operationerna `Inspect-first` och `Delete-first` läser av värdet på respektive tar bort elementet som har högst prioritet i kön. Operationen `Update` används av en del grafalgoritmer efter att ett element i Prioritetskön har förändrats. Resultatet av operationen är att återställa den interna datatypen (Listan) till ett korrekt tillstånd.

	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
`Insert`	X				
`Inspect-first`			X		
`Delete-first`			X		
`Update`	X				

- `Insert` behöver bara stoppa in elementet var som helst i Listan, t.ex. först, vilket har komplexitet $O(1)$.
- `Inspect-first` och `Delete-first` måste traversera listan för att hitta elementet med högst prioritet, vilket har komplexitet $O(n)$. I genomsnitt kommer halva listan att behöva traverseras, dvs. konstanten kommer att vara nära $c = 0.5$.
- `Update` behöver inte göra någonting; det finns inget krav på att listan ska vara sorterad. Att inte göra någontin har komplexitet $O(1)$.

5.4 Uppgift 5d) Komplexitet operationer: Sorterad lista (4p)

Ange komplexiteten för nedanstående operationer om Prioritetskön är konstruerad med hjälp av en **Sorterad lista**. Operationerna **Inspect-first** och **Delete-first** läser av värdet på respektive tar bort elementet som har högst prioritet i kön. Operationen **Update** används av en del grafalgoritmer efter att ett element i Prioritetskön har förändrats. Resultatet av operationen är att återställa den interna datatypen (den Sorterade listan) till ett korrekt tillstånd.

	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
`Insert`			X		
`Inspect-first`	X				
`Delete-first`	X				
`Update`			X		

- `Insert` behöver traversera listan för att hitta var det nya värdet ska stoppas in, vilket har komplexitet $O(n)$. I genomsnitt behöver halva listan traverseras, alltså blir konstanten ungefär $c = 0.5$.
- `Inspect-first` och `Delete-first` behöver bara läsa av/ta bort det första elementet, vilket bägge har komplexitet $O(1)$.
- Efter att elementets värde uppderats behöver `Update` flytta elementet till rätt position i listan. Att flytta ett element i en lista kräver att man traverserar listan, dvs. $O(n)$. Det är troligt att de flesta förflyttningar är korta, så konstanten c kommer troligen att vara liten.

6 Uppgift 6) Identifiera algoritm

```
1  Algorithm foo(x: Array, u: Value, Is-equal, Is-related: Function)
2
3  a ← Low(x)
4  b ← High(x)
5  while a < b do
6    d ← b - a + 1
7    if d mod 2 = 1 then
8      c ← a + ((d - 1) / 2)
9    else
10     c ← a + d / 2
11   v ← Inspect-value(x, c)
12   if Is-equal(v, u) then
13     return c
14   else if Is-related(v, u) then
15     a ← c + 1
16   else
17     b ← c - 1
18
19 return Low(x) - 1
```

a) Panelen till vänster innehåller en algoritm i pseudokod. Algoritmen/funktionen `foo` tar ett fält `x` som antas vara sorterat enligt en sorteringsordning `R`. Sorteringsordningen `R` är implementerad av funktionen `Is-related(x, y)` och returnerar `True` om `x` är *relaterad* till `y`, dvs. om `x R y` är sant. Vidare så returnerar funktionen `Is-equal(x, y)` värdet `True` om `x` och `y` anses vara lika.

Algoritmen `foo` är känd under ett annat namn. Vilket?

- **Svar:** Binärsökning, binär sökning, binary search.
- Jag har gett 1p om man svarat "sökningsalgoritm".

b) Vad är komplexiteten för `foo` som en funktion av antalet element `n` i Fältet `x`?

- **Svar:** $O(\log n)$

c) Algoritmen innehåller en bug. På vilken rad finns buggen?

- **Svar:** Buggen finns på rad 5 som lyder `while a < b do`. Algoritmen terminerar när det finns ett element kvar att undersöka. Det betyder att om det sökta värdet finns i det elementet så kommer algoritmen inte att hitta det.

d) Vilken kod skulle stå på den raden för att åtgärda buggen?

- **Svar:** `while a <= b do`
- **Sånt som inte är buggar:**
  - Flera har svarat att rad 19 ska lyda: `return Low(x)`, vilket inte är korrekt. Algoritmen har två `return`-satser. På rad 13 har sökningen lyckats och index för det funna elementet returneras. På rad 19 har sökningen misslyckas och funktionen ska returnera ett ogiltigt index som felsignal. `Low(x)` är ett giltigt index, `Low(x) - 1` är det inte.

7 Uppgift 7) Graf

7.1 Uppgift 7a) Egenskaper för grafer

Antag att vi har en oriktad graf med 7 noder.

a) Hur många bågar måste den minst ha för att kunna vara en *komplett* graf?

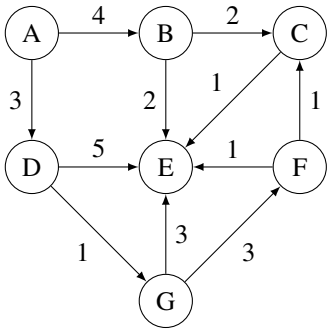
- **Svar:** I en komplett graf är alla noder grannar till alla andra noder. Det betyder att var och en av de 7 noderna har 6 grannar. Om grafen varit riktad hade det behövs 7*6=42 bågar för att göra den komplett. Exempelvis behövs bågen (a,b) och (b,a) för att göra a och b grannar till varandra. I en oriktad graf räcker det med bågen (a,b) för att göra a och b till grannar med varandra. Vi klarar oss alltså på hälften så många bågar, dvs. **42/2=21** bågar.

b) Hur många bågar måste den minst ha för att kunna vara en *sammanhängande* graf?

- **Svar:** En graf med n noder behöver minst $n - 1$ bågar för att vara sammanhängande, alltså **6** bågar.

7.2 Uppgift 7b) Dijkstras algoritm

Utför Dijkstras algoritm på grafen nedan med början i nod A. Lista noderna i den ordning som de tas bort från prioritetskön och för var och en det kortaste avståndet från A till noden. Om två avstånd är lika sorteras noderna i bokstavsordning ("lägst" bokstav kommer före i sorteringsordning).

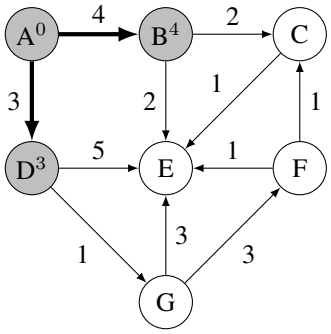


- **Svar:**

Steg	Nod	Avstånd
1	A	0
2	D	3
3	B	4
4	G	4
5	C	6
6	E	6
7	F	7

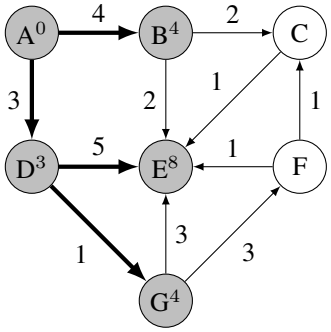
7.2.1 Beskrivning

- Innehållet i prioritetskön efter initieringen: `q = [ A(0) ]`
- Steg 1: Nod A plockas ut från prioritetskön. efter att alla grannar till A besökts ser grafen och prioritetskön ut så här (gråa noder har ”setts” av algoritmen):



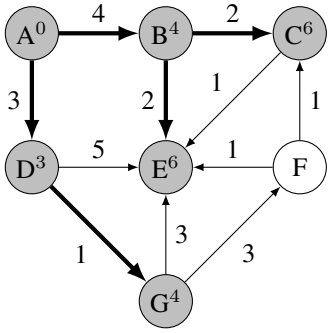
`q = [ (D,3) (B,4) ]`

- Steg 2: Nod D plockas ut från prioritetskön. efter att alla grannar till D besökts ser grafen och prioritetskön ut så här:



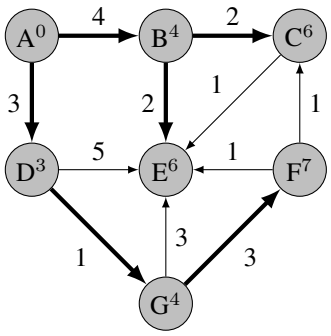
`q = [ (B,4) (G,4) (E,8) ]`

- Steg 3: Nod B plockas ut från prioritetskön. efter att alla grannar till B besökts ser grafen och prioritetskön ut så här:



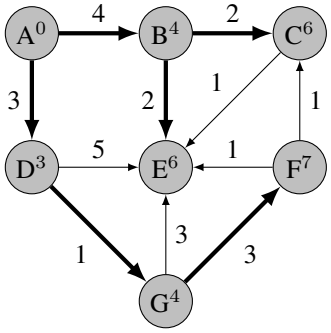
`q = [ (G,4) (C,6) (E,6) ]`

- Steg 4: Nod G plockas ut från prioritetskön. efter att alla grannar till G besökts ser grafen och prioritetskön ut så här:



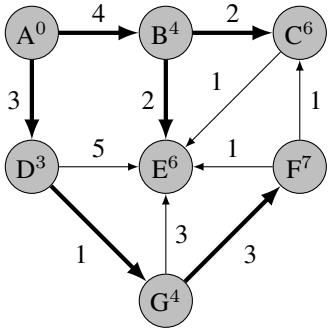

```
q = [ (C,6) (E,6) (F,7) ]
```

- Steg 5: Nod C plockas ut från prioritetskön. efter att alla grannar till C besökts ser grafen och prioritetskön ut så här:



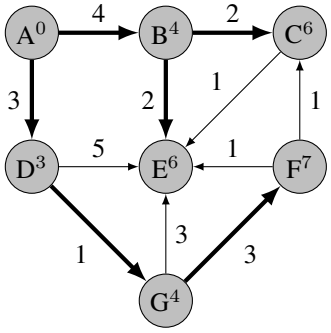

```
q = [ (E,6) (F,7) ]
```

- Steg 6: Nod E plockas ut från prioritetskön. efter att alla grannar till E besökts ser grafen och prioritetskön ut så här:




```
q = [ (F,7) ]
```

- Steg 6: Nod F plockas ut från prioritetskön. efter att alla grannar till F besökts ser grafen och prioritetskön ut så här:



`q = [ ]`