Exam

Niclas Borlin

2024-03-06

Contents

1 1. Grundbegrepp (10)

1.1 1.1 Positioner och Index

a) En Position kan användas tillsammans med flera listor.

- **Falskt**: En Position kan endast användas tillsammans med ”sin” lista

b) Ett Index kan användas tillsammans med flera fält.

- **Sant**: Ett index kan användas tillsammans med vilket fält som helst, förutsatt att indexvärdet ligger inom fältets gränser.

c) En Position i en lista blir odefinierad när listans struktur förändras.

- **Sant**: Alla ”gamla” positioner blir odefinierade efter en Insert() eller Remove().

d) Ett Index i ett Fält blir odefinierat när fältets struktur förändras.

- **Falskt**: Dessutom kan ett fälts struktur inte förändras.

e) En Lista kan innehålla flera positioner än element.

- **Sant**: En lista innehåller **alltid** en position mer än antalet element.

f) En Lista kan innehålla flera elementvärden än element.

- **Falskt**: Däremot gäller motsatsen: att det går att ha fler element än elementvärden, t.ex. i listan [1,2,2] som har 2 elementvärden och 3 element.

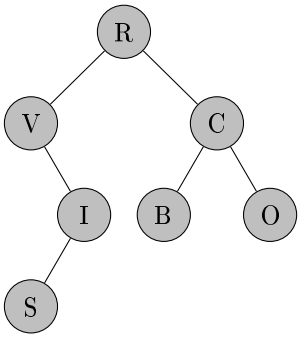
1.2 1.2 Egenskaper hos datatyper

Datatyp	Enkel	Sammansatt	Homogen	Ordnad
Heltal	X			
Fält		X	X	X
Post (Record)		X		
Lista		X	X	X
Tabell		X	X	

- Kommentar:
  - Heltal är en icke sammansatt datatyp, så den kan varken vara homogen eller heterogen, ordnad eller oordnad.
  - Post är heterogen då olika fält i posten kan ha olika typer. Post är också oordnad, då det inte spelar någon roll i vilken ordning fältens namn specificeras.
  - Tabell är homogen, då den innehåller element av typen nyckel-värde-par. Paret kan vara heterogent, men tabellen är homogen.

2 2. Träd

2.1 2.1 Traversering av binärt träd



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: `R,V,C,I,B,O,S`

b) I vilken ordning kommer noderna att besökas om trädet traverseras enligt djupet-först, preorder?

- Svar: `R,V,I,S,C,B,O`

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

- Svar: `V,S,I,R,B,C,O`

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

- Svar: `S,I,V,B,O,C,R`

2.2 2.2 Indentifiera trädalgoritmer

```
Algorithm foo(n: Node, T: BinTree)

if Has-left-child(n, T) then
   p ← 1 + foo(Left-child(n, T))
else
   p ← 0

if Has-right-child(n, T) then
   q ← 1 + foo(Right-child(n, T))
else
   q ← 0

return max(p, q)
```

```
Algorithm bar(n: Node, T: BinTree)

if Has-left-child(n, T) then
   bar(Left-child(n, T))

if Has-right-child(n, T) then
   bar(Right-child(n, T))

if not (Has-left-child(n, T) or Has-right-child(n, T)) then
   print(Label(n, T))
```

Betrakta algoritmerna `foo` och `bar` i panelen till vänster. I frågorna nedan är `T` ett godtyckligt binärt träd och `T1` är det binära träd som återfinns längst ner i panelen (och är samma träd som i förra uppgiften).

e) Beskriv i text vilken trädegenskap anropet `foo(Root(T), T)` beräknar:

- Svar: **Trädets höjd**.
- Bedömning: Gett halva poäng för ”längden på längsta grenen” eller ”längst vägen till ett löv”.
- Någon undrade vad funktionen `max()` gör. Den returnerar det högsta värdet (”maximum”) av dess argument.

f) Vad skulle anropet `foo(Root(T1), T1)` returnera?

- Svar: **3**.

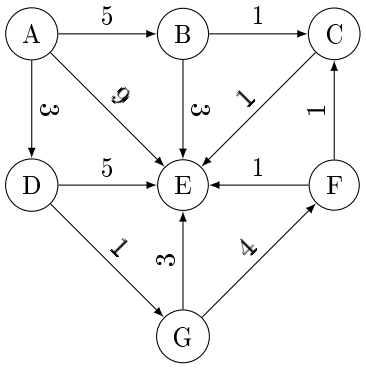
g) Beskriv i text vad `bar(Root(T), T)` skriver ut, och i vilken ordning:

- Svar: **Etiketterna för lövnoderna, från vänster till höger**.
- Bedömning: Ett poäng för ”löv”, ett poäng för ”från vänster till höger”.
- Några har förvirrats av att det stod `not (Has-left-child() or Has-right-child())`, vilket är sant för noder **utan** barn, dvs. löv.
- Texten innehöll ett typo ”Beskriv i text *var*...” som förvirrat några. Såg detta när jag rättat klart. Har gått igenom en gång till för alla som ligger nära en gräns.

h) Vad skulle anropet `bar(Root(T1), T1)` generera för utskrift?

- Svar: **SBO**

3 3. Grafalgoritmer, Dijkstra



Applicera Dijkstras algoritm på grafen till vänster. Starta i nod A. När algoritmen ska iterera över ett antal bågar, iterera över bågarna i stigande viktordning. I tabellen nedan, ange i vilken ordning noder får ett **förändrat** avstånd. Tänk på att en nod kan få förändrat avstånd flera gånger.

- Svar: (nc betyder "no change")

Steg	Aktiv nod	Nod	Avstånd
0	-	A	0
1	A 0	D	3
2		B	5
3		E	9
4	D 3	**G**	**4**
5		**E**	**8**
6	G 4	**E**	**7**
7		**F**	**8**
8	B 5	C	6
		E	(8)
	C 6	E	(7)
	E 7	-	-
	F 8	C	(9)
9		-	-

- Kommentar:
  - De inledande 3 förändringarna sker med A-noden som aktiv med avstånd 0.
    - När A är klar så består prioritetskön av (D 3, B 5, E 9).
  - Nästa aktiva när är D med avstånd 3.
    - Nästa förändring (nr 4) är att G-noden upptäcks och får avståndet 4.
    - Nästa förändring (nr 5) är att E-noden får avståndet uppdaterat från 9 till 8.
    - När D är klar så består prioritetskön av (G 4, B 5, E 8).
  - Nästa aktiva när är G med avstånd 4.
    - Nästa förändring (nr 6) är att E-noden får avståndet uppdaterat från 8 till 7.
    - Nästa förändring (nr 7) är att F-noden upptäcks och får avståndet 8.
    - När D är klar så består prioritetskön av (B 5, E 7, F 8).
  - Nästa aktiva när är B med avstånd 5.
    - Nästa förändring (nr 8) är att C-noden upptäcks och får avståndet 6.
    - Inga fler förändringar med B som aktiv.
    - När B är klar så består prioritetskön av (E 7, F 8).

- Nästa aktiva när är E med avstånd 7.
  - E har inga grannar, så inga förändringar med E som aktiv.
  - När E är klar så består prioritetskön av (F 8).
- Nästa aktiva när är F med avstånd 8.
  - Inga förändringar med F som aktiv.
  - När F är klar så består prioritetskön tom.
- Sista förändringen (nr 9) är alltså ”ingen”, dvs. streck.

4 4. Sortering

4.1 4.1 Sortering efter kombinationer av nycklar, sekvens

Vid sortering är det vanligt att värdena som sorteras består av Poster (*Records*), där något eller några fält används som nycklar vid sorteringen. En vanlig önskan är att posterna i första hand ska sorteras efter en *primär* nyckel, t.ex. efternamn, i andra hand efter en *sekundär* nyckel, t.ex. förnamn.

Den effektivaste lösningen är att skriva en egen jämförelsefunktion som kombinerar nycklarna för att avgöra vilken Post som ska komma först. Under vissa förutsättningar (se nästa fråga) kan vi dock uppnå samma resultat genom upprepad användning av en sorteringsalgoritm som endast kan sortera efter ett fält i posten.

Antag att anropet `Sort(l, 'firstname')` tar listan `l` av Poster och returnerar en kopia av listan sorterad efter förnamn. Antag på samma sätt att anropet `Sort(l, 'lastname')` returnerar en kopia av `l` sorterad efter efternamn.

a) Låt `l` innehålla den osorterade listan. Någon eller några av följande anrop/sekvenser av anrop ger oss en lista `s` sorterad i första hand efter efternamn och i andra hand efter förnamn. Ett eller flera svar är korrekt. Vilken/vilka? Denna fråga ger avdrag för fel svar.

- (X)
  - `t <- Sort(l, 'firstname')`
  - `s <- Sort(t, 'lastname')`
-
  - `t <- Sort(l, 'lastname')`
  - `s <- Sort(t, 'firstname')`
-
  - `s <- Sort(Sort(s, 'lastname'), 'firstname')`
- (X)
  - `s <- Sort(Sort(s, 'firstname'), 'lastname')`

- Kommentar:
  - Den sista sorteringen måste vara efter den primära nyckeln, dvs. efternamn.
  - Den näst sista sorteringen måste vara efter den sekundära nyckeln, dvs. förnamn.
  - I det generella fallet sorterar man alltså efter nycklarna i omvänd prioritetsordning.

4.2 4.2 Sortering efter kombinationer av nycklar, egenskaper

b) För att ovanstående lösning ska ge önskat resultat krävs att vår sorteringsalgoritm har en viss egenskap, vilken?

- Sorteringsalgoritmen är *in-place*.
- Sorteringsalgoritmen har komplexiteten högst n log n.
- Sorteringsalgoritmen är stabil.
- Sorteringsalgoritmen är rekursiv.
- Sorteringsalgoritmen använder ett pivåelement.

- Svar: Algoritmen måste vara **stabil**. Det betyder att när listan sorteras efter den primära nyckeln så bevaras den tidigare ordningen (som var baserad på den sekundära nyckeln).

5 5. Komplexitet

5.1 5.1 Tillväxthastigheter

Studera tillväxthastigheterna nedan. Ordna tillväxthastigheterna i ordning efter tillväxt för stora n. Notera att en långsam tillväxt ger en snabb algoritm och vice versa.

- Svar:

Växer långsammast (snabbast algoritm)	$O(1)$
.	$O(\log n)$
.	$O(n)$
.	$O(n \log n)$
.	$O(n^2)$
Växer snabbast (långsammast algoritm)	$O(2^n)$

5.2 5.2 Ordo c, n_0 (4 poäng)

Studera funktionen $T(n) = 3n^2 - 3n + 10$. Givet $T(n)$, vad blir c och lägsta n_0 för $g(n) = n^2$ enligt ordo-definitionen? Om något värde behöver avrundas, avrunda uppåt till närmast högre heltal.

- Svar: $c = 3$ ger $n_0 = 4$ enligt nedanstående tabell. Jag godkänner även $c = 4$, då med $n_0 = 2$ (halv poäng för $n_0 = 3$).
- Bedömning: 2p för c, 2p för n_0 om c korrekt.

n	$T(n)$	n^2	$3n^2$	$4n^2$
1	10	1	3	4
2	16	4	12	**16**
3	28	9	27	36
4	46	16	**48**	64
5	70	25	75	100

5.3 5.3 Ordo $g(n)$ (spetsfråga)

Studera funktionen $T(n) = 3n \log(n^2)$. Vilket av följande alternativ är det bästa $g(n)$ som uppfyller Ordo-definitionen?

- $g(n) = \log n$
- $g(n) = n \log n$
- $g(n) = n^2 \log n$
- $g(n) = \log n^2$
- $g(n) = n^2 \log n^2$
- Svar: $\log n^2 = 2 \log n$ och konstanten 2 försvinner in i c, så $g(n) = n \log n$ är rätt svar.

6 6. Listor

1. Skriv en algoritm (funktion) `Multiplicity(v: Value, l: DList)` som tar en riktad (och osorterad) lista `l` och ett värde `v` och returnerar *multipliciteten* för `v`, dvs. antalet gånger som `v` förekommer som värde i `l`. Algoritmen ska ha följande huvud: `Algorithm Multiplicity(v: Value, l: DList)`

2. Skriv en algoritm `Max-multiplicity(l: DList)` som använder `Multiplicity()` och returnerar den största multipliciteten i listan `l`. Algoritmen ska ha följande huvud: `Algorithm Max-multiplicity(l: DList)`

3. Antag att du har en riktad lista `l` som kan innehålla upprepade värden. Skriv en funktion `Most-common(l: DList)` som returnerar en ny lista som innehåller alla elementvärden som har högst multiplicitet i `l`. (Kallas för *typvärde* eller *mode* på statistikspråk.) Använd `Multiplicity()` och/eller `Max-multiplicity()` i din lösning! Algoritmen ska ha följande huvud: `Algorithm Most-common(l: DList)`

   Du kan välja två svårighetsgrader på denna uppgift. För x poäng så får din lösning returnera en lista med godtyckligt antal upprepningar av elementvärdena. Most-common på listan [3, 5, 9, 3, 7, 9] skulle då kunna returnera t.ex. [3, 9, 3, 9] eller [9, 3, 9]. För full poäng får utlistan inte innehålla upprepade värden, dvs. Most-common skulle returnera [3, 9] eller [9, 3] på listan ovan. I samtliga fall spelar ordningen på värdena ingen roll.

I din kod får du anta att likhetsoperatorn är definierat för enkla datatyper, t.ex. Heltal, men inte för datatypen Value. Däremot finns funktionen `Value-isequal(a, b: Value)` som returnerar True om `a` och `b` är lika.

Låt n vara antalet element i den riktade listan.

- Vad är komplexiteten för `Multiplicity()`? $O(n)$
- Vad är komplexiteten för `Max-multiplicity()`? $O(n^2)$
- Vad är komplexiteten för `Most-common()`? $O(n^2)$

6.1 Lösningsförslag

Notera all:

- `Multiplicity()` ska returnera 0 om värdet inte finns i listan.
- `Max-multiplicity()` för en tom lista ska returnera 0.
- `Most-common ()` för en tom lista ska returnera en tom lista.

```
Algorithm Multiplicity(v: Value, l: DList)
// Count the number of occurences of the value v in the list l

// Initialize counter
m ← 0

// Iterate over the input list...
p ← First(l)
while not Isend(p, l) do
    if Value-isequal(Inspect-value(p, l), v) then
        // Increment the counter for each matching value
        m ← m + 1
    // Advance in the list
    p ← Next(p, l)

// Return the count
return m
```

```
Algorithm Max-multiplicity(l: DList)
// Compute the largest multiplicity of any value in the list l

// Highest multiplicity seen so far
mx ← 0

// Iterate over the input list...
p ← First(l)
while not Isend(p, l) do
    // Compute multiplicity for this value
    mult ← Multiplicity(l, Inspect-value(p, l))
    // If this multiplicity is higher than any seen so far...
    if mult > mx then
        // ...remember this value
        mx ← mult

    // Advance in the input list
    p ← Next(p, l)

// Return the highest seen multiplicity
return mx
```

```
Algorithm Most-common-with-duplicates(l: DList)
// Returns a list with the most common values in the list l, i.e.,
// those values that have the largest multiplicity. The values will
// have the same multiplicity in the output list as in the input list.

// First compute the largest multiplicity in the list
m ← Max-multiplicity(l)

// Output list with values that have max multiplicity
t ← Empty()

// Iterate over the list...
p ← First(l)
while not Isend(p, l) do
    v ← Inspect-value(p, l)
    // If the multiplicity of this value equal the maximum...
    if Multiplicity(v, l) = m then
        // ...insert value into the output list. Since the order does not matter,
        // insert it first (simple).
        t ← Insert(v, First(t), t)

    // Advance in the input list
    p ← Next(p, l)

// Return the output list
return t
```

```
Algorithm Most-common-no-duplicates(l: DList)
// Returns a list with the most common values in the list l, i.e.,
// those values that have the largest multiplicity. The output list
// contains each value exactly once.

// First compute the largest multiplicity in the list
m ← Max-multiplicity(l)

// Output list with values that have max multiplicity
t ← Empty()

// Iterate over the list...
p ← First(l)
while not Isend(p, l) do
    v ← Inspect-value(p, l)
    // If the multiplicity of this value equal the maximum...
    if Multiplicity(v, l) = m then
        // ...and the values does not appear in the output list
        if Multiplicity(v, t) > 0 then
            // ...insert value into the output list. Since the order does not matter,
            // insert it first (simple).
            t ← Insert(v, First(t), t)

    // Advance in the input list
    p ← Next(p, l)

// Return the output list
return t
```

6.2 Bedömning

- Allvarliga
  - Använda heltal som position, dvs. blanda ihop index och positioner. Ger automatisk noll poäng på hela uppgiften.
  - Använder en position i flera listor. Ger noll poäng.
  - Sammanblandning av typer alt. typer och variabler, t.ex. returnera en lista när man ska returnera ett heltal. Ger stora avdrag.
  - C-specika lösningar, t.ex. ++. Ger stora avdrag.
  - Fel loopkonstruktioner, if där det borde vara while, etc.
  - Iterationsuppdatering next() utanför while-loopen i stället för inuti.
  - Flera lösningar där man jämfört värdet i en position med nästa, vilket alltid fick problem i slutet på listan (next(end()) är odefinierat).
  - Avancerar inte alltid i listan, t.ex. endast om likhet i Multiplicity.
  - Total avsaknad av kommentarer och indentering => noll poäng.
- Mindre allvarliga
  - Funktionerna ska klara en tom lista: Max-multiplicity ska då returnera 0, Most-common en tom lista.
    - Klarar en tom lista (-1p) (många Inspect(First(l), l) innan man kollat Isend(First(l),l).
  - Felanvändning av gränsytans funktioner, t.ex. Isend(l) utan positionsparameter
  - Tar ej hand om returvärden från t.ex. Insert()/Remove().
    - Använder positioner som ej är definierade efter Insert()/Remove().

7 7. Huffman/LZ

7.1 7.1 Hjälpdatatyper (2+2 poäng)

Kompressionsalgoritmerna Huffman och LZ78 använder olika datatyper vid kodning (kompression) och dekodning (dekompression).

a) Vilken datatyp använder LZ78 för kodning?

- Rätt svar: Trie.
- Halv poäng för träd (Trie är ett sorts träd)
- Noll poäng för binärt träd.

b) Vilken datatyp använder LZ78 för dekodning?

- Rätt svar: Tabell
- Halv poäng för Fält

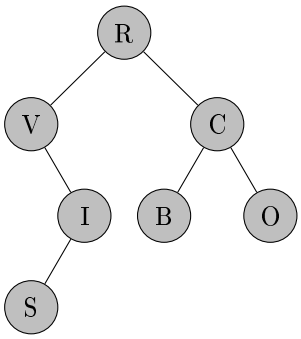
7.2 7.2 Dekodning

- Dekoda följande sträng som kodats av LZ-78-algoritmen:
  `(0, B)`, `(0, A)`, `(0, N)`, `(2, N)`, `(0, K)`, `(2, K)`, `(4, S)`.
- Svar: `BANANKAKANS` enligt följande tabell:

Input	Table	Output
	0	
(0, B)	1 B	B
(0, A)	2 A	A
(0, N)	3 N	N
(2, N)	4 AN	AN
(0, K)	5 K	K
(2, K)	6 AK	AK
(4, S)	7 ANS	ANS

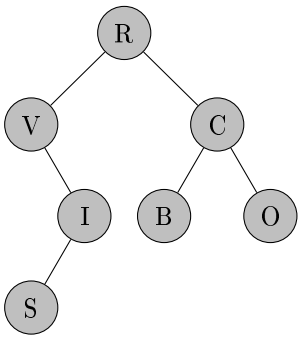
- Poäng: Gett nästan fulla poäng för nästan rätt.

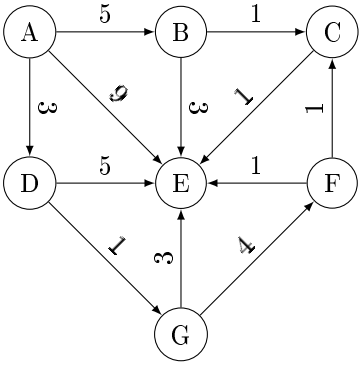
8 Illustrationer


R
V
C
I
B
O
S

```
Algorithm foo(n: Node, T: BinTree)

if Has-left-child(n, T) then
   p ← 1 + foo(Left-child(n, T))
else
   p ← 0

if Has-right-child(n, T) then
   q ← 1 + foo(Right-child(n, T))
else
   q ← 0

return max(p, q)
```

```
Algorithm bar(n: Node, T: BinTree)

if Has-left-child(n, T) then
   bar(Left-child(n, T))

if Has-right-child(n, T) then
   bar(Right-child(n, T))

if not (Has-left-child(n, T) or Has-right-child(n, T)) then
   print(Label(n, T))
```





9 Poängsättning

1	Grundbegrepp	
1.1	Positioner och Index	6
1.2	Egenskaper hos datatyper	5
2	Träd	0
2.1	Traversering	12
2.2	Identifiering av algoritmer	8
3	Grafer, grafalgoritmer	0
3.1	Dijkstras algoritm	12
4	Sortering	0
4.1	Sekvens	4
4.2	Egenskap	3
5	Komplexitet	0
5.1	Tillväxthastighet	9
5.2	Ordo c, n0	4
5.3	Ordo g(n)	2
6	Listor, pseudokod	0
6.1	Pseudokod	15
6.2	Komplexitet	6
7	LZ78	0
7.1	Hjälpdatatyper	4
7.2	Dekodning	10
	Summa	100