Några svar till
TDDC70/91 Datastrukturer och algoritmer

2010-10-19

Följande är lösningsskisser och svar till uppgifterna på tentan. Lösningarna som ges här ska bara ses som vägledning och är oftast inte tillräckliga som svar på tentan.

1. (a) **Falskt**. Skipplistan har *förväntad* uppslagningstid $O(\log n)$ men inte i värsta fallet eftersom den är probabilistisk och alla element kan hamna på översta/nedersta nivån.

   (b) **Falskt**. Ett motexempel: $f(n) = g(n) = n^2$ och $h(n) = n^3$.

   (c) **Sant**. Enligt definitionen av Ω behöver vi hitta en reell konstant $c > 0$ och en heltalskonstant $n_0 \geq 1$ sådana att $n \log_2(n) \geq cn$ för $n \geq n_0$. Valet $c = 1$ och $n_0 = 2$ visar att $n \log_2(n) \geq cn$ för $n \geq n_0$ eftersom $\log_2(n) \geq 1$ i det intervallet.

   (d) **Sant**. $3^{\log(n^2)} = 3^{2\log(n)} = 9^{\log(n)} = n^{\log 9}$ som är polynomiskt.

2. (a) Den här algoritmen beräknar a^n. Körtiden för algoritmen är $O(n)$ eftersom

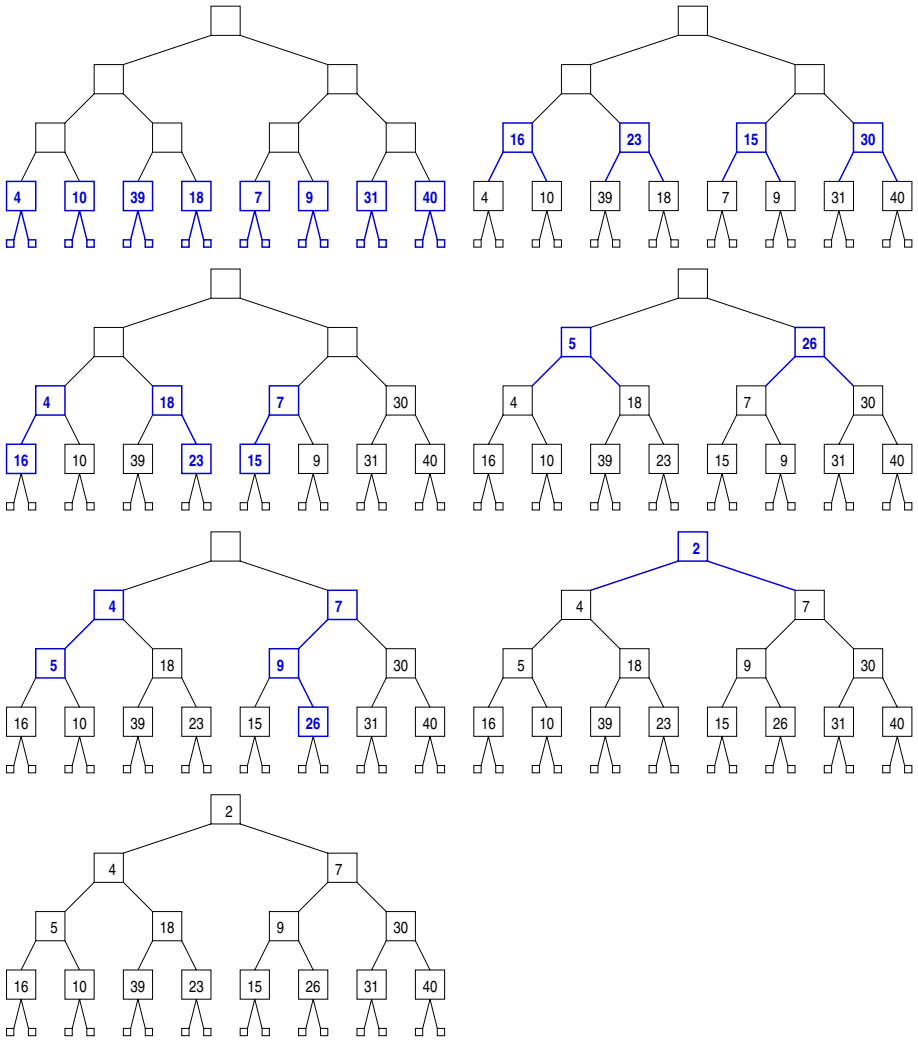
   - de initiala tilldelningarna tar konstant tid
   - varje iteration av **while**-looopen tar konstant tid
   - det blir exakt n iterationer

   (b) Den här algoritmen beräknar också a^n. Körtiden för algoritmen är $O(\log n)$ av följande skäl:

   Initialiseringen och **if**-satsen med sitt innehåll tar konstant tid, så vi behöver lista ut hur många gånger **while**-loopen nås. Eftersom k minskar (antingen halveras eller minskas med ett) i varje steg och är lika med n initialt kommer loopen som värst att exekveras n gånger. Men vi kan göra en bättre analys.

   Notera att k halveras om k är jämnt och att k minskas med ett och halveras i nästa iteration om k är udda. Så åtminstone varannan iteration av **while**-loopen halverar k. Det går att halvera ett tal n som mest $\lceil \log n \rceil$ gånger innan det blir ≤ 1. (Varje gång vi halverar ett tal skiftar vi det åt höger med en bit, och ett tal n har $\lceil \log n \rceil$ bitar.) Om vi minskar talet med ett mellan att vi halverar det kan vi fortfarande inte halvera det fler än $\lceil \log n \rceil$ gånger. Eftersom vi bara kan minska k med ett mellan två iterationer som halverar k (om inte n är udda eller det är sista iterationen) får vi göra en interation som minskar k med ett högst $\lceil \log n \rceil + 2$ gånger. Så vi har som mest $2\lceil \log n \rceil + 2$ iterationer. Vilket ger oss tidskomplexiteten $O(\log n)$.

3. (a) Så här blir det, om vi låter de små fyrkanterna representera null (Om man tänker sig en arraybaserad heap behövs förstås inte null-pekarna.):



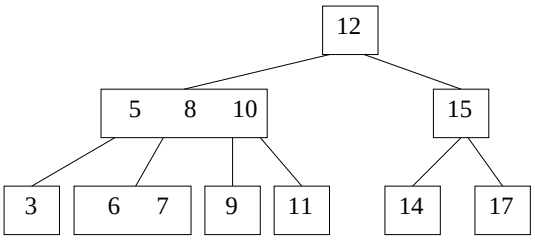
(b) Notera att om v är en nod i T som är större än x så är alla nycklar lagrade i dess delträd också större än x. Alltså är det tillräckligt att söka med start i roten efter noder i T som är mindre än eller lika med x. Vi visar här ett rekursivt sätt att göra detta på. Genom att använda en kö eller en stack går det att formulera iterativa lösningar.

Require: en heap T, en nod v i T, en frågenyckel x
Ensure: hittar nycklarna i delträdet av T rotat i v som är mindre än eller lika med x

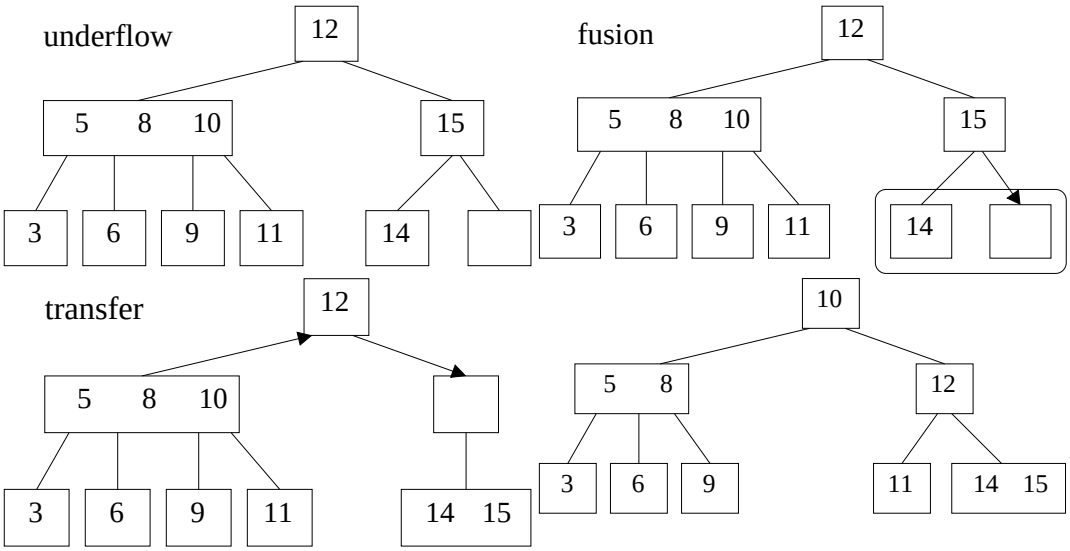
```
procedure findSmaller(T, v, x)
    if v ≠ null then
        if (v.key()) ≤ x then
            reportKey(v.key())
            findSmaller(T, T.leftChild(v), x)
            findSmaller(T, T.rightChild(v), x)
```

Exekveringstiden för algoritmen blir $O(k)$ av följande skäl. När vi flyttar oss nedåt i trädet tittar vi på varje nod med en nyckel mindre än eller lika med x exakt en gång. De enda övriga noder vi tittar på är deras barn. Givet k hittade nycklar har de k motsvarande noderna i heapen $2k$ barn. Trots det faktum att några av deras barn också är bland noderna vars nycklar är $\leq x$ ger detta oss fortfarande den sökta övre gränsen: vi tittar på $3k$ noder, vilket är $O(k)$ noder.

4. (a)



(b) Med kursbokens terminologi får vi:



5. (a) Med linjär sondering placerar vi ett element på platsen med index $h(k)$ eller första lediga platsen därefter (om $H[h(k)]$ redan är upptagen).

En liten hjälptabell:

Nyckel	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Bokstav	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
mod 13	0	1	2	3	4	5	6	7	8	9	10	11	12	0	1	2	3	4	5	6	7	8	9	10	11	12

Sonderingssekvenserna:

E	A	S		Y	Q	U	E	S	T		I	O	N
4	0	5	11	3	7	4	5	6		8	1	0	
						5	6	7		9		1	
						6	7	8	10			2	
							8	9					

Så hashtabellen blir till slut:

0	1	2	3	4	5	6	7	8	9	10	11	12
A	O	N	Q	E_1	S_1	E_2	U	S_2	T	I	Y	

(b) Med dubbel hashning använder vi strategin att om h avbildar k på en plats $H[i]$, med $i = h(k)$, som redan är upptagen provar vi iterativt platserna $H[(i + j \cdot h'(k)) \bmod 13]$ för $j = 1, 2, 3, \ldots$.

Så hashtabellen blir till slut:

0	1	2	3	4	5	6	7	8	9	10	11	12
A	I	N	Q	E_1	S_1	T	U	S_2	E_2		Y	O

Med sonderingssekvenserna:

```
E A S  Y Q U E S T  I  O  N
4 0 5 11 3 7 4 5 6  8  1  0
           9 0      4  5  3
             8      0  9  6
                    9  0  9
                    5  4 12
                    1  8  2
                      12
```

6. Den här algoritmen kommer att använda en sorteringsalgoritm – vilken sorteringsalgoritm som helst med $O(n \log n)$ tidskomplexitet i värsta fallet duger (t.ex. merge-sort).

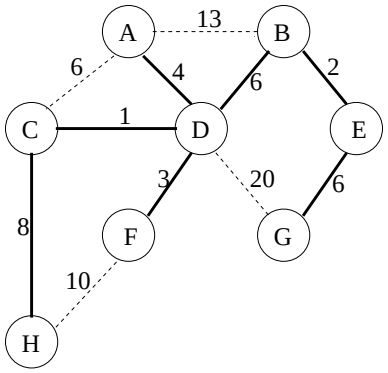
   Vi ger en grov beskrivning av algoritmen följt av pseudokod.

   - Först beräknar vi en ny sekvens C, med storlek n, sådan att för varje i mellan 0 och $n-1$ gäller $C[i] = m - A[i]$. Detta tar $O(n)$ tid.
   - Vi sorterar C, vilket tar $O(n \log n)$ tid.
   - Vi sorterar B, vilket tar $O(n \log n)$ tid.
   - Vi söker genom C och B från början och letar efter element som är lika. Eftersom C och B är sorterade behöver vi som värst göra $2n$ jämförelser. Därför tar det här steget $O(n)$ tid.
   - Om C och B har ett gemensamt element (säg b) finns det ett element a i A sådant att $b = m - a$. Notera att vi inte behöver hitta något sådant element enligt problembeskrivningen, det räcker att bekräfta att ett sådant element finns.

   Alltså blir exekveringstiden för den här algoritmen $O(n + n \log n + n \log n + n) = O(n \log n)$

```
Require: array A, array B, storlek n på A och B, heltal m
Ensure: avgör om det finns a ∈ A och b ∈ B sådana att a + b = m
  function isThereASum(A, B, n, m)
      låt C vara en ny array med storlek n
      for i ← 0 to n − 1 do
          C[i] ← m − A[i]
      sort(C)
      sort(B)
      cind ← 0
      bind ← 0
      while cind ≠ n och bind ≠ n do
          if C[cind] = B[bind] then
              return true
          if C[cind] ≤ B[bind] then
              cind ← cind + 1
          else
              bind ← bind + 1
      return false
```

7. (a)



Nästa nod	Uppskattat avst. från startnod							Båge från molnet till nästa nod
	B	C	D	E	F	G	H	
A	13	6	4	∞	∞	∞	∞	—
D	10	5	4	∞	7	24	∞	A–D
C	10	5	4	∞	7	24	13	D–C
F	10	5	4	∞	7	24	13	D–F
B	10	5	4	12	7	24	13	D–B
E	10	5	4	12	7	18	13	B–E
H	10	5	4	12	7	18	13	C–H
G	10	5	4	12	7	18	13	E–G

(b) Det här kan åstadkommas genom att modifiera Dijkstras algoritm. I stället för att representera den kortaste vägen från a till u låter vi märkningen $D[u]$ representera den maximala bandbredden över alla vägar från a till u. Den maximala bandbredden för en väg från a via u till en nod z som är granne till u är $\min\{D[u], w((u,z))\}$ så att relaxeringssteget uppdaterar $D[z]$ till $\max\{D[z], \min\{D[u], w((u,z))\}\}$.

Require: Viktad sammanhängande enkel graf G och två skiljda noder a och b
Ensure: Returnerar den maximala bandbredden över alla vägar från a till b

```
function maxBandwidth(G, a, b)
    initialisera D[a] ← ∞ och D[u] ← 0 för varje nod u ≠ a i G
    låt en prio-kö Q innehålla alla noder i G med D-värdena som nycklar
    while Q är icketom do
        u ← Q.removeMax()
        if u = b then                          ▷ hittade slutnoden — kan stanna här
            return D[u]
        else
            for each granne z till u som är i Q do
                d ← min{D[u], w((u, z))}
                if d > D[z] then
                    D[z] ← d
                    ändra z:s nyckelvärde i Q till D[z]
```

En avslutande **return**-sats är onödig eftersom $u = b$ vid något tillfälle i huvudloopen varför algoritmen terminerar och returnerar resultatet då.

Genom att representera grafen med en grannlista blir exekveringstiden densamma som för Dijkstras algoritm – $O((n+m)\log n)$, där n är antalet noder i G och m antalet bågar i G, om prioritetskön implementeras m.h.a. en heap och $O(n^2)$ om prioritetskön implementeras med en osorterad sekvens. (Det går att komma ner till $O(n \log n + m)$ genom att använda en ännu tjusigare datastruktur för prioritetskön.)