Introduktion til algoritmik og datastrukturer
Vejledende løsninger

IT-højskolen i København

13. Juni 2000

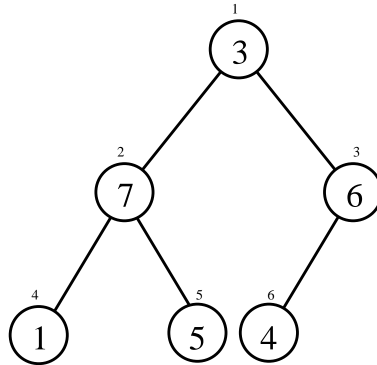
Dette eksamenssæt består af 3 opgaver, hver med 5 delopgaver. Alle opgaverne vægtes ens, og alle delopgaverne i hver opgave vægtes ens. Du har i alt 4 timer til din rådighed. Husk at angive sidetal, navn og studienummer på alle sider i din besvarelse. Eksamensættet består af 9 nummererede sider.

CLR refererer til "Introduction to Algorithms" af Cormen, Leiserson og Rivest, 18. tryk, 1997.

Opgave 1

Denne opgave handler om hobe og sortering.

a) Illuster udførelsen af HEAPIFY$(A, 1)$ for hoben A i figur 1. Anvend stilen fra figur 7.2 i CLR side 143.



Figur 1: Hoben A, der skal udføres HEAPIFY på.

Svar a) Først ombyttes 3 med 7, dernæst ombyttes 3 med 5 og algoritmen stopper.

b) QUICKSORT (CLR side 154) er defineret vha. PARTITION(A, p, r), som kan bytte rundt på tallene i array'et A fra indeks p til r. Illustrer beregningen af PARTITION$(A, 1, 6)$, hvor

$$A = \langle 7, 8, 9, 4, 5, 6 \rangle$$

I kan benytte Fig. 8.1 side 155 i CLR som en model for illustrationen.

Svar b) Der pivoteres omkring 7. Resultatet bliver $A = \langle 6, 5, 4, 9, 8, 7 \rangle$

c) Konstruer en sorteringsalgoritme, som har lineær kompleksitet, hvis tallene, der skal sorteres, allerede er *sorterede*; ellers er kompleksiteten $O(n \lg n)$. Skriv pseudo-kode.

Svar c) CHECK(A)

1. **for** $i = 2$ **to** LENGTH(A) **do**
2. **if** $A[i-1] \leq A[i]$ **return false**
3. **return true**

SORT1(A)

1. **if** CHECK(A) **return** A
2. **else** MERGESORT(A)

QUICKSORT er forventeligt mere effektiv end INSERTION-SORT (CLR side 3) for store inddata, men eksperimenter har vist, at INSERTION-SORT er mere effektiv end QUICKSORT, hvis der skal sorteres få tal.

d) Skriv pseudo-kode for en sorteringsalgoritme, der er en kombination af QUICKSORT og INSERTION-SORT. Algoritmen skal udnytte, at INSERTION-SORT i praksis er mere effektiv end QUICKSORT, hvis der skal sorteres færre end c tal, for en eller anden konstant c. Det er tilladt at henvise til kode i CLR.

Svar d) SORT2(A, p, r)

1. $d \leftarrow r - p$
2. **if** $d \geq c$ **then**
3. $\quad q \leftarrow$ PARTITION(A, p, r)
4. $\quad$ SORT2(A, p, q)
5. $\quad$ SORT2($A, q+1, r$)
6. **else** INSERTIONSORT(A, p, r)

e) Hvad er kompleksiteten af HEAPSORT, hvis tallene, der skal sorteres, er *identiske*?

Svar e) $O(n)$ tid da hvert kald til HEAPIFY tager $O(1)$ tid i dette tilfælde.

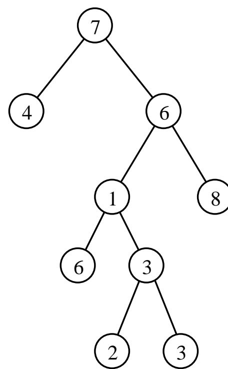
Opgave 2

Denne opgave handler om binære træer med ikke-negative (≥ 0) heltalsvægte i knuderne.

Vi definerer den *vægtede længde* af en vej (simple path) i et træ til at være summen af vægtene i knuderne på vejen, inklusive endeknuder. Dvs. den vægtede længde fra roden i træet i figur 2 til bladet med vægten 2 har en vægtet længde på 19.

Blandt flere veje i et træ er en *vægtet længste vej* en af vejene med den største vægtede længde. F.eks. har den vægtede længste vej (blandt alle mulige veje) i træet i figur 2 værdien 25.

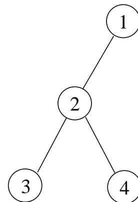
Vægt-dybden af et træ er værdien af en vægtet længste vej blandt vejene fra roden til et blad i træet. F.eks. er vægt-dybden 21 for træet i figur 2.



Figur 2: Binært træ med vægte.

a) Vis ved hjælp af et lille modeksempel, at den vægtede længste vej i et træ T ikke nødvendigvis går igennem roden på T.

Svar a) Se figur 3



Figur 3: Modeksempel.

I denne opgave betragter vi den sædvanlige repræsentation af et binært træ T, som brugt i CLR kapitel 13. Dvs. til hver knude v i træet T har vi felterne

left$[v]$, *right*$[v]$, og $p[v]$ for venstre barn, højre barn og forældreknuden (parent) henholdsvis. For roden r er $p[r] = \text{NIL}$ og for et blad w er både *left*$(w) = \text{NIL}$ og *right*$(w) = \text{NIL}$. Herudover lader vi *key*$[v]$ betegne *vægten* i knuden v. Bemærk, at vi *ikke* antager, at vægtene opfylder *binary-search-property* i denne opgave.

b) Lav en procedure VÆGTTILROD(v), der beregner den *vægtede længde* af vejen fra knuden v til roden af T. Proceduren skal køre i tid $O(h)$, hvor h er højden af T.

Svar b) VÆGTTILROD(v)

1. $s \leftarrow$ *key*$[v]$
2. **while** $p[v] \neq$**nil do**
3. $\quad v \leftarrow p[v]$
4. $\quad s \leftarrow s+$ *key*$[v]$
5. **return** s.

c) Lav en rekursiv procedure VÆGTDYBDE, der beregner vægt-dybden af et binært træ T.

Svar c) VÆGTDYBDE(v)

1. **if** v =**nil return** 0
2. **else return**
3. $\quad$ max(VÆGTDYBDE (*left*$[v]$), VÆGTDYBDE (*right*$[v]$))+*key*$[v]$

Vi er nu interesseret i at udvide datastrukuren for det vægtede binære træ T, så vi hurtigt kan finde vægt-dybden, når vægtene i knuderne kan opdateres løbende. Mere præcist er vi interesseret i at udvide datastrukturen for T, så vi kan understøtte følgende to operationer:

OPDATER(v, x): Ændrer T, så knuden v får vægten x.

AKTUELVÆGTDYBDE(): Returnerer den aktuelle vægt-dybde for T.

d) Beskriv en udvidelse af datastrukturen for et træ T, der understøtter de to operationer ovenfor. Køretiden for proceduren OPDATER skal være $O(h)$, hvor h er højden af T, mens AKTUELVÆGTDYBDE skal køre i tid $O(1)$.
Vink: Lav et ekstra felt til hver knude v, der indeholder den aktuelle vægt-dybde af *undertræet* med rod v.

Svar s) Vi tilføjer et felt *weight*$[v]$ til hver knude v, der indeholder den aktuelle vægtdybde af undertræet med rod i v. Operationerne bliver som følger.

AKTUELVÆGTDYBDE(): Returner *weight*$[v]$.

OPDATER(v, x): Lad r være roden af træet. Sæt *weight*$[v] \leftarrow x$ og for alle knuder w på vejen fra v til r sæt

$weight[w] \leftarrow \max(\ weight(left[w]),\ weight(right[w])) + key[v]$

Vi er nu interesseret i en anden udvidelse af datastrukturen for T. Først skal træet T initialiseres, så alle knuder initielt har vægt 1. Derefter skal vi håndtere følgende to operationer:

SÆTNUL(v): Ændrer vægten i knuden v til 0, dvs. sæt *key*$[v]$ til 0.

NULVEJ(u, v): Returnerer true, hvis og kun hvis den vægtede længde af vejen fra knude u til knude v er 0.

e) Beskriv en datastruktur for ovennævnte problem. For et træ T med n knuder skal løsningen give en *total* køretid på $O(n \lg^* n)$ for initialisering og udførelse af n SÆTNUL og NULVEJ operationer.

Svar e) Vi anvender Union-Find datastrukturen. For hver knude v associerer vi et element $e(v)$ og en vægt *weight*$[e(v)]$. Først opretter vi n mængder til hvert element med MAKE-SET operationer. Alle vægte er initielt sat til 1. Operationerne bliver som følger.

SÆTNUL(v): Sæt *weight*$[e(v)] \leftarrow 0$. For alle knuder w incident med v, hvis *weight*$[e(w)] = 0$ udfør UNION$(e(v), e(w))$.

NULVEJ(u, v): Returnerer true, hvis FIND-SET(u)=FIND-SET(v). Ellers returner false.

Vi har valgt at lade længden af den vægtede vej fra en knude til sig selv være 0. Det er ligetil at ændre datastrukturen hvis man ikke ønsker dette. Da $O(n)$ MAKE-SET, UNION og FIND-SET kan udføres i $O(n \lg^* n)$ har datastrukturen de ønskede tidskompleksiteter.

Opgave 3

For alle delopgaverne i denne opgave gælder det, at A er en tabel (array) bestående af n heltal. Det første tal i tabellen er $A[0]$, og det sidste tal i tabellen er $A[n-1]$.

I figur 4 er vist en tabel med 8 tal. For denne tabel er $n = 8$, $A[0] = 15$ og $A[n-1] = 314$

15	20	31	150	210	214	217	314

Figur 4: En tabel med 8 tal.

a) Lad A være en (ikke nødvendigvis sorteret) tabel med n tal. Beskriv, hvorledes man i $O(n \lg n)$ tid kan afgøre, om der er to ens tal i tabellen.

Svar a) Sorter tabellen og undersøg i lineær tid om to elementer $A[i-1]$ og $A[i]$ er ens for $2 \leq i \leq n$.

b) Lad A være en (ikke nødvendigvis sorteret) tabel med n heltal, hvor alle de n tal har en værdi mellem 1 og $3n$. Hvor hurtigt kan de n heltal sorteres?

Svar b) Ved at anvende f. eks. COUNTING-SORT kan A sorteres i $O(n+3n) = O(n)$.

c) Lad A være en sorteret tabel med n tal. Lad SUCC(x) være en procedure, der returnerer det mindste tal y fra A, som er større end x. Hvis et sådan tal ikke findes returneres ∞. SUCC(170) på tabellen i figur 4 er således 210. Beskriv, hvorledes SUCC kan konstrueres, så den har tidskompleksiteten $O(\lg n)$.

Svar c) Udfør en binær søgning efter elementet.

Lad NEXT(i) være en procedure der givet et indeks i fra en tabel A returnere det mindste indeks j, som er størrer end i, hvor tallet $A[j]$ er lige. Hvis et sådan

indeks ikke eksistere returneres $n - 1$. Betragt tabellen i figur 4. Her vil f.eks. NEXT(0) returnere 1, NEXT(1) vil returnere 3 og NEXT(5) vil returnere 7. Antag, at hvis NEXT(i) returnerer j, så har NEXT kompleksiteten $O(1 + j - i)$. Betragt følgende procedure AMOR: AMOR(k)

1. $x \leftarrow 0$
2. **for** $j \leftarrow 1$ **to** k
3. **do** $x \leftarrow$ NEXT(x)

d) Lad A være en tabel med n tal. Hvad returnerer et kald til NEXT($n - 1$)? Hvad er kompleksiteten for et kald til NEXT($n - 1$)? Hvad er kompleksiteten for et kald til AMOR(k)? Du skal for hver af følgende kompleksitetsmål $O(n)$, $O(k)$, $O(n + k)$ og $O(n * k)$ angive og begrunde om kompleksitetsmålet udtrykker kompleksiteten for AMOR(k).

Svar d) NEXT($n - 1$) returnerer $n - 1$ ifølge definitionen og tager derfor $O(1)$ tid. Kompleksiteten for AMOR(k) er $O(n + k)$ da vi højst gennemløber A en gang og højst udfører k kald til NEXT

I den næste delopgave vil vi betragte en tabel A, hvor alle tallene i tabellen til at starte med er 1. Endvidere antager vi, at tabellen indeholder n tal, hvor n kan skrives som $2^k + 1$ for et heltal $k \geq 0$ (dvs. n er et af tallene $2, 3, 5, 9, 17, 33, \ldots$).

I figur 5 er vist en tabel med 9 tal. For denne tabel er $n = 2^3 + 1 = 9$, $k = 3$, og $A[i] = 1$, for alle indices i.

1	1	1	1	1	1	1	1	1

Figur 5: En table med 9 et-taller.

Betragt følgende to procedurer NYETAL og HALVER:

NYETAL()

1. $i \leftarrow 0$

2. **while** $i \leq n - 1$

3. **do** $A[i] \leftarrow A[i] * 2$

4. $i \leftarrow i + A[i]$

HALVER(k)

1. **for** $j \leftarrow 1$ **to** k

2. **do** NYETAL()

e) Lad A være en tabel med 9 tal, som alle er 1. Tegn situationen efter et kald til NYETAL. Lad nu A være en tabel med n tal, hvor $n = 2^k + 1$ for et helttal $k \geq 0$. Hvad er kompleksiteten af et kald HALVER(k)?

Svar e) Efter et kald til NYETAL er $A = \langle 2, 1, 2, 1, 2, 1, 2, 1, 2 \rangle$. Et kald til HALVER resulterer i k kald til NYETAL. NYETAL fordobler længden af intervallet den springer hver gang den bliver kaldt. Følgelig bliver kompleksiteten af HALVER(k) $= O(n/2 + n/4 + \cdots) = O(n)$.