Løsningsforslag til kapittel 6, 7, 8 og 9 i *Introduction to Algorithms* (Cormen et al.)

Ola Natvig (olanatv krøllalfa stud.ntnu.no)

1. oktober 2007

Forord

Dette dokumentet er ment som hjelp for å kontrollere svar på treningsoppgaver i læreboka *Introduction to Algorithms* av *Thomas Cormen et al.*. Det kan ha sneket seg inn noen feil i løsningene, hvis du finner en feil (også skrivefeil osv..) ikke nøl med å ta kontakt.

Kapittel 6

Seksjon 6.1 - Treningsoppgaver

6.1 - 1

En heap av høyde h har minst et element på dybde h og maks et fullt nivå h. En heap er et binært tre. Antall noder i et fullt binært tre er gitt ved:

$$n = 2^{h+1} - 1$$

Dette gir for en heap av høyde h:

$$2^h \leq n \leq 2^{h+1} - 1$$

6.1 - 2

Høyden til et tre med n mellom 2^h og $2^{h+1} - 1$ noder har høyde h.

$$h \leq \lg(n) \leq \lg(2^{h+1} - 1) < h + 1.$$

Siden $lg(n)$ er mindre enn $h + 1$ rundes $lg(n)$ ned til h □

6.1 - 3

Hvis heap-egenskapen gjelder over hele heapen har roten for ethvert sub-tre i heapen være større eller lik begge barna sine. Denne gjelder rekursivt og roten for subtreet må derfor være større eller lik alle sine etterkommere.

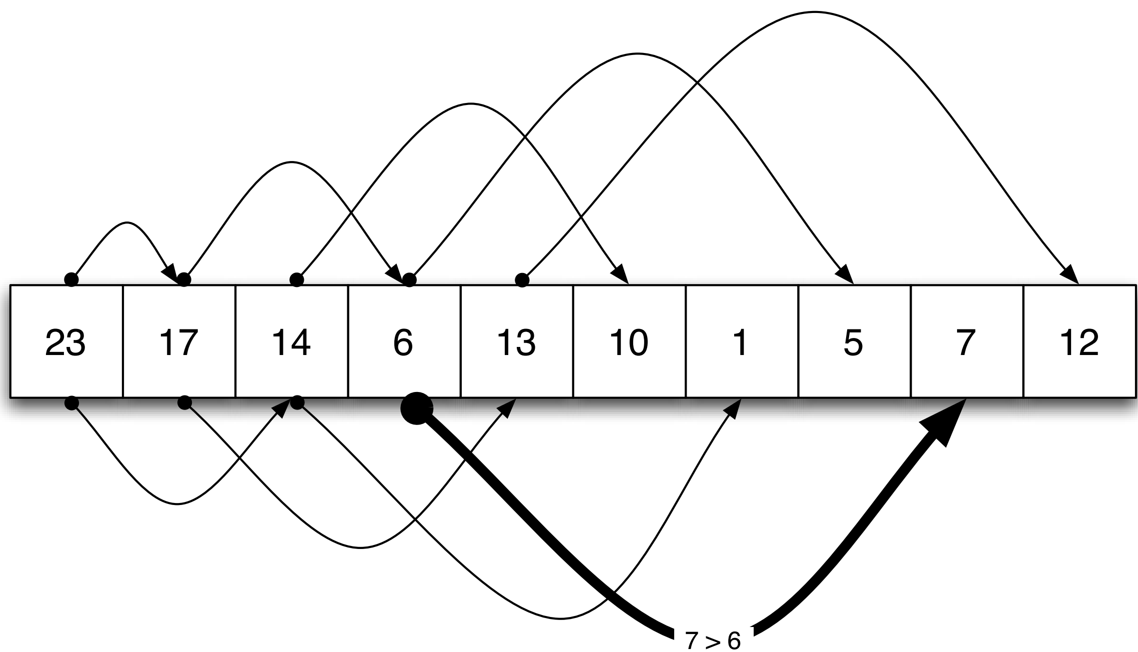
6.1 - 4

Gitt at alle elementer er ulike kan ikke det minste elementet i en maks-heap ha noen barn, det minste elementet må derfor ligge i det laveste nivået.

6.1 - 5

I et stortert array gjelder alltid: $X_i \leq X_{i+1}$, dette gjør at også : $X_i \leq X_{2i}$ og $X_i \leq X_{2i+1}$ gjelder. Derfor er et stortert array alltid en min-heap.

6.1 - 6



Sekvens tegnet med heap-relasjoner.

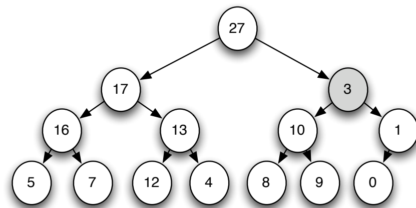
Vi ser at 6 er foreldrenode for 7, dette bryter med maks-heap-egenskapen. $(6 < 7)$

6.1 - 7

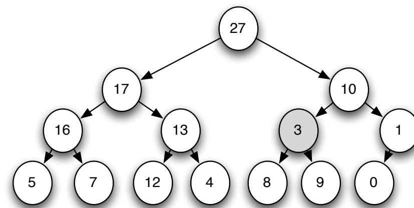
Den siste noden som ikke er en løvnode er foreldrenoden til node nummer n, alle noder etter denne noden er løvnoder. Foreldrenoden til node n er $\lfloor n/2 \rfloor$. De neste nodene $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2 ....., n$ er da løvnoder. $\square$.

Seksjon 6.2 - Treningsoppgaver

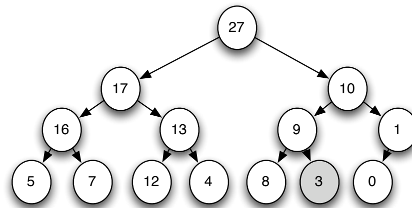
6.2 - 1



(a) Heapify(A, 3)



(b) Heapify(A, 6)



(c) Heapify(A, 13)

6.2 - 2

Istede for å velge ut den største av foreldre-noden og de to barna velges den minste. Dette er i grunnen bare å snu større enn tegn i linje 3 og 6 i psaudokoden på side 130 i cormen.

6.2 - 3

Max-Heapify(A, i) kalt på når $A[i]$ er større enn alle sine barn har ingen effekt, heap-egenskapen er allerede oppfylt.

6.2 - 4

Max-Heapify(A, i) kalt når $i > heap_size(A)/2$ har ingen effekt fordi node i da ikke har noen barn, og heap-egenskapen for en node uten barn alltid holder.

6.2 - 5

Siden max-heapify har kun et rekursivt kall og dette kallet er det siste kallet i funksjonen (tail-recursion) kan koden lett skrives om til en løkke.

Program 1 Iterativ versjon av Max-Heapify

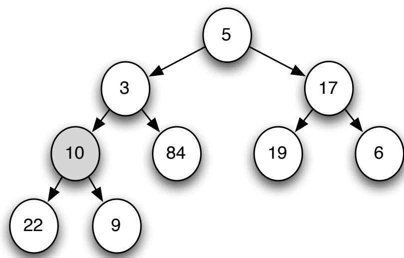
```
def Max-Heapify(A, i):
    while True:
        l = left(i), r = right(i)
        if l <= heap-size(A) and A[l] > A[i]: largest = l
        else: largest = i
        if r <= heap-size(A) and A[r] > A[largest]: largest = r
        if largest == i: return
        else:
            A[i], A[largest] = A[largest], A[i]
            i = largest
```

6.2 - 6

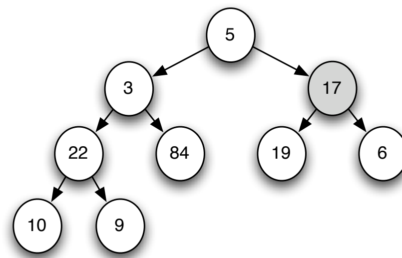
Gitt en heap av størrelse n og som er en min-heap med kun ulike verdier, da er alle heap-relasjoner i konflikt med maks-heap-egenskapen. Et max-heapify kall til rotnoden i heapen vil da bli kalt rekursivt helt ned til en løvnode i heapen. Lengden på en sti fra rotnoden til en løvnode i en heap er logaritmisk.

Seksjon 6.3 - Treningsoppgaver

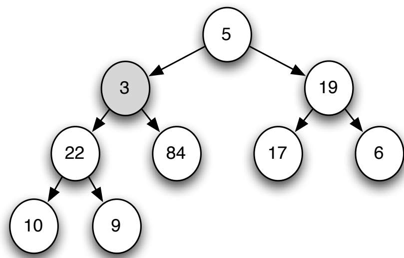
6.3 - 1



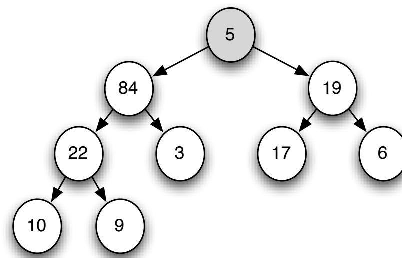
(a) Max-Heapify(A, 4)



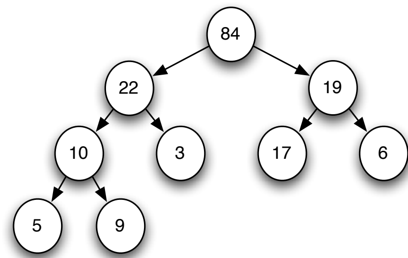
(b) Max-Heapify(A, 3)



(c) Max-Heapify(A, 2)



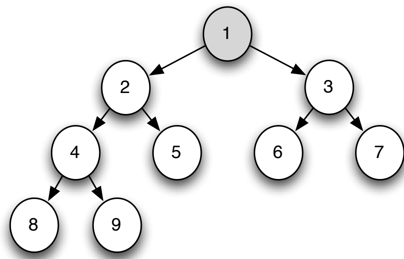
(d) Max-Heapify(A, 1)



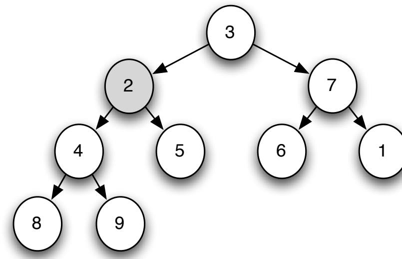
(e) A er en max-heap

6.3 - 2

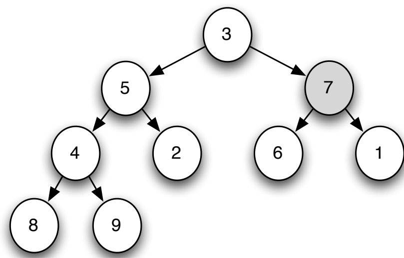
Grunnen til at build-max-heap går 'baklengs' gjennom treet er at dersom heapify kalles på en node der venstre og høyre sub-tre ikke er en 'korrekt' heap vil ikke heapify berøre mer enn et av subtreene. Når man går baklengs er alle subtrær heapify berører korrekte heaper.



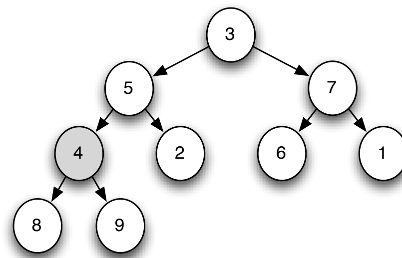
(a) Max-Heapify(A, 1)



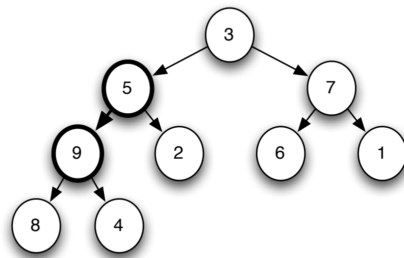
(b) Max-Heapify(A, 2)



(c) Max-Heapify(A, 3)



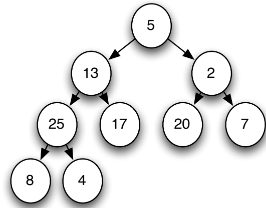
(d) Max-Heapify(A, 4)



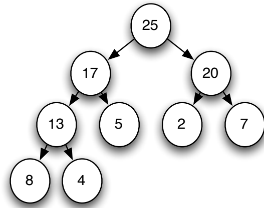
(e) A er ikke en max-heap

Seksjon 6.4 - Treningsoppgaver

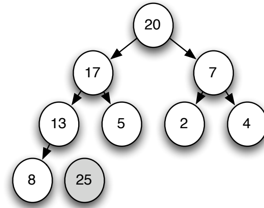
6.4 - 1



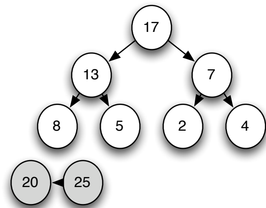
(a) input



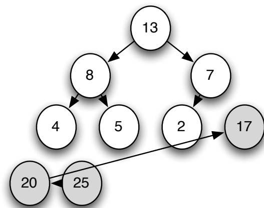
(b) max-heap



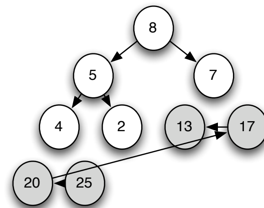
(c) 25 på riktig posisjon



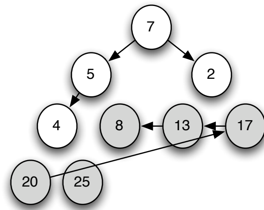
(d) 20 på riktig posisjon



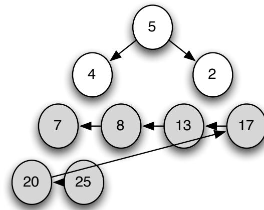
(e) 17 på riktig posisjon



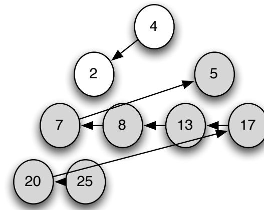
(f) 13 på riktig posisjon



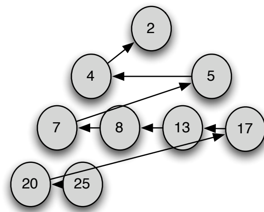
(g) 8 på riktig posisjon



(h) 7 på riktig posisjon



(i) 5 på riktig posisjon



(j) 4 og 2 på riktig posisjon

6.4 - 2

Initialisering: Før første iterasjon er i lik n og $A[1...i]$ er lik $A[1...n]$ og inneholder en maks-heap av de i minste elementene i $A[1...n]$ (d.v.s. en maks-heap av alle elementene i A). Arrayet $A[i+1..n]$ inneholder de 0 største elementene i A sortert. Invarianten holder ved initialisering. **Vedlikehold:** for hver runde i for-løkken i linje 2-5 velges det største av de i minste elementene i A og plasseres på posisjon i. Maks-heapen vedlikehodles av et *Max-Heapify* kall og i settes lik $i-1$. Etter hver slik iterasjon er $A[1..i]$ en maks-heap av de i minste elementene i A og $A[i+1....n]$ de $n-i$ største elemnetene i sortert rekkefølge. Invarianten holder. **Terminering:** Når $i = 1$ er $A[1]$ en max-heap med det minste elementet i A og $A[2....N]$ de $N-1$ største elemntene i A i sortert rekkefølge. Invarianten holder.

6.4 - 3

Build-Max-Heap tar lineær tid (Cormen s.135), heapsort gjør n kall til *Max-Heapify*, *Max-Heapify* har logaritmisk kjøretid (Cormen s.131-132) kjøretiden er derfor $O(n \lg n)$ for alle input.

6.4 - 4

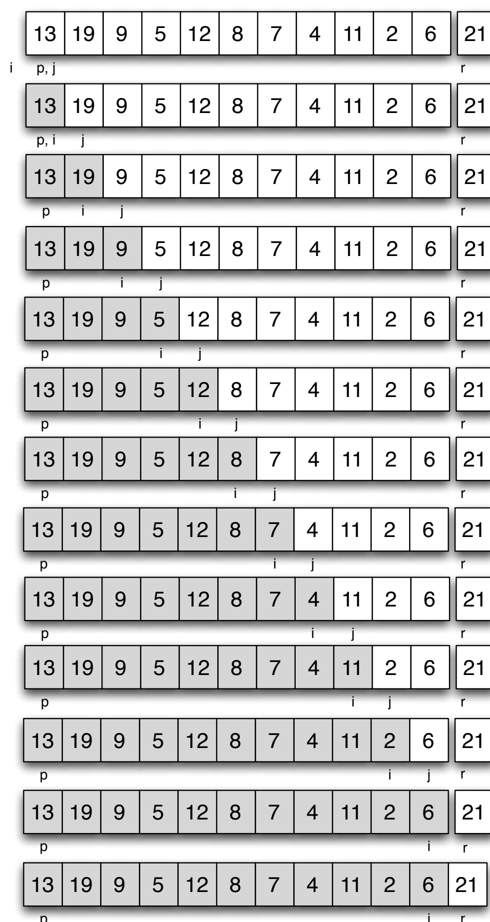
Heapsort er sortering med sammenlikning, worst-case kjøretid kan derfor ikke være bedre enn $\Omega(n \lg n)$.

Kapittel 7

Seksjon 7.1 - Treningsoppgaver

7.1 - 1

Oppgaven ber oss om å illusterer *Partition* på arrayet $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$ *Partition* funksjonen beskrevet i boka bruker det siste elementet i A som pivot, noe som i dette tilfellet gjør at A partisjoneres rundt sitt største element.



7.1 - 2

Partition har en indre løkke som går fra p til r, dvs den går n steg, Partition har derfor $\Theta(n)$ kjøretid.

7.1 - 3

Dette kan gjøres på flere måter, den enkleste er å snu sammenlikningen i linje 4 i *Partition*. (Cormen s. 146).

Seksjon 7.2 - Treningsoppgaver

7.2 - 2

Kjøretiden for *Quicksort* på et array A med kun like elemnter er $\Theta(n^2)$.

7.2 - 3

Når A er sortert i synkende rekkefølge velges alltid det minste elemntet i A som pivot element. Dette gjør at *Partition* gir en partisjon med $n-1$ elementer og en annen med null elementer. Kjøretiden for *Quicksort* blir da $\Theta(n^2)$.

7.2 - 4

Ved sortering av nesten sortert input vil den indre løkken i *Insertion-Sort* kjøre kort tilbake før den finner den riktige posisjonen til en verdi, dette gjør at *Insertion-Sort* har $\Theta(n)$ på nesten-sortert data. *Quicksort* derimot vil velge pivot elementer som gir dårlig partisjonering og vil derfor få $\Theta(n^2)$ kjøretid.(Se Cormen side 17 for *Insertion-Sort*)

7.2 - 5

$\lg_a(n) = \frac{\lg(n)}{\lg(a)}$ Treets maksimale og minimale høyde er $\lg_{1-a}(n)$ og $\lg_a(n)$

Seksjon 7.3 - Treningsoppgaver

7.3 - 1

Vi analyserer gjennomsnittstilfellet av randomiserte algoritmer fordi vi ofte ikke har noe konkret input som fremkaller worst-case kjøretid.

7.3 - 2

Vi får like mange kall til den tilfeldig tallgeneratoren som vi får kall til *Partition*. I verste tilfelle har vi ensidig partisjonering (største eller minste pivot) da får vi $\Theta(n)$ kall til partition. I beste tilfelle får vi balansert partisjonering og et rekurensstre med høyde $\lg(n)$. Partisjonering vil bli kalt i alle nodene i dette treet bortsett fra løvnodene. Vi har $2^{\lg(n)} - 1$ interne noder og antall kall til partition blir også da $\Theta(n)$.

Kapittel 8

Seksjon 8.1 - Treningsoppgaver

8.1 - 1

En løvnode i beslutningstreet til en sammenlikningsalgoritme må være minst n. Dette fordi sammenlikningsalgoritmen må gjøre minst n sammenlikninger for å i det hele tatt vite om input er sortert.

Seksjon 8.2 - Treningsoppgaver

8.2 - 1

A	6	0	2	0	1	3	4	5	1	3	2
C	2	2	2	2	1	1	1				
C	2	4	6	8	9	10	11				

(a) Bygg kumulatic C tabell

A	6	0	2	0	1	3	4	5	1	3	
B						2					
C	2	4	5	8	9	10	11				

(b) Plasser 2

A	6	0	2	0	1	3	4	5	1		
B						2		3			
C	2	4	5	7	9	10	11				

(c) Plasser 3

A	6	0	2	0	1	3	4	5			
B				1		2		3			
C	2	3	5	7	9	10	11				

(d) Plasser 1

A	6	0	2	0	1	3	4				
B				1		2		3		5	
C	2	3	5	7	9	9	11				

(e) Plasser 5

A	6	0	2	0	1	3					
B				1		2		3	4	5	
C	2	3	5	7	8	9	11				

(f) Plasser 4

A	6	0	2	0	1						
B				1		2	3	3	4	5	
C	2	3	5	6	8	9	11				

(g) Plasser 3

A	6	0	2	0							
B			1	1		2	3	3	4	5	
C	2	2	5	6	8	9	11				

(h) Plasser 1

A	6	0	2								
B		0	1	1		2	3	3	4	5	
C	1	2	5	6	8	9	11				

(i) Plasser 0

A	6	0									
B		0	1	1	2	2	3	3	4	5	
C	1	2	4	6	8	9	11				

(j) Plasser 2

A	6										
B	0	0	1	1	2	2	3	3	4	5	
C	0	2	4	6	8	9	11				

(k) Plasser 0

A											
B	0	0	1	1	2	2	3	3	4	5	6
C	0	2	4	6	8	9	10				

(l) Plasser 6, ferdig sortert

8.2 - 2

Vi går igjennom kildetabellen baklengs, av objekter med lik nøkkel vil derfor det av objektene som forekommer sist i kildetabellen settes inn sist i resultattabellen.

8.2 - 3

Algoritmen fungerer fortsatt, C tabellen setter av plass til alle objekter med en gitt nøkkelverdi, men den nye algoritmen er ikke stabil, den er konsekvent ustabil i det at av objekter med lik nøkkelverdi byttes innbyrdes posisjoner helt om.

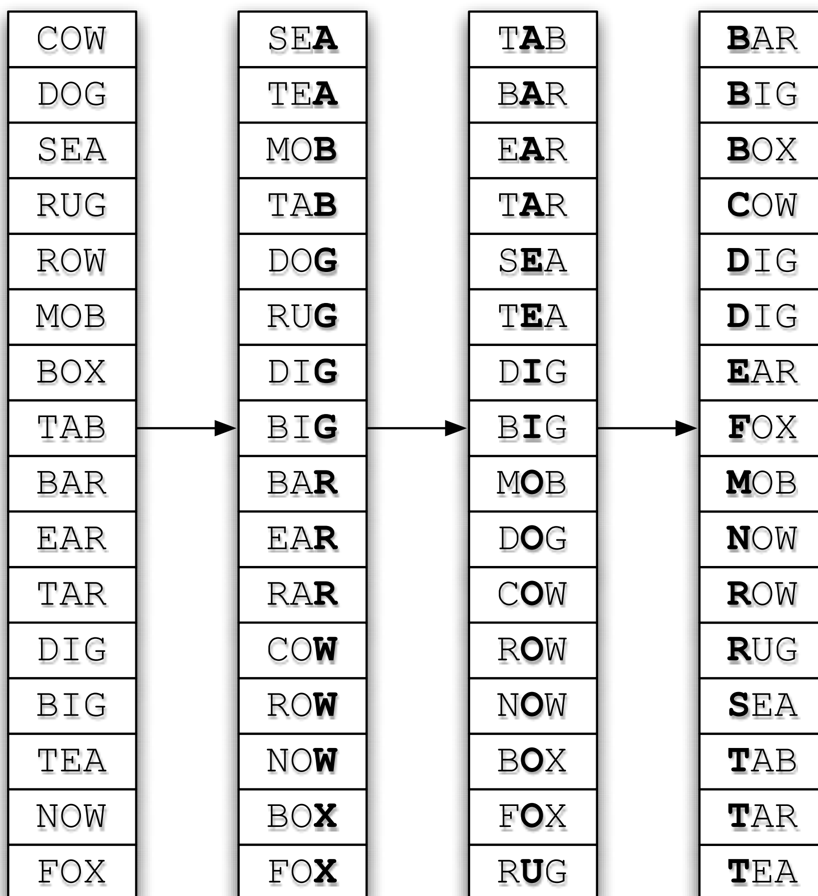
8.2 - 4

Lag den kumulative C tabellen fra tellesortering. Denne preproseseringen har kjøretid $\Theta(n+k)$ siden input telles i $\Theta(n)$ tid og tabellen C gjøres kumulativ i $\Theta(k)$ tid. Svar spørringer med: $C[b] - C[a-1]$ eller bare $C[b]$ dersom a er 0.

Seksjon 8.3 - Treningsoppgaver

8.3 - 1

Radix-Sort på trebokstavsord.

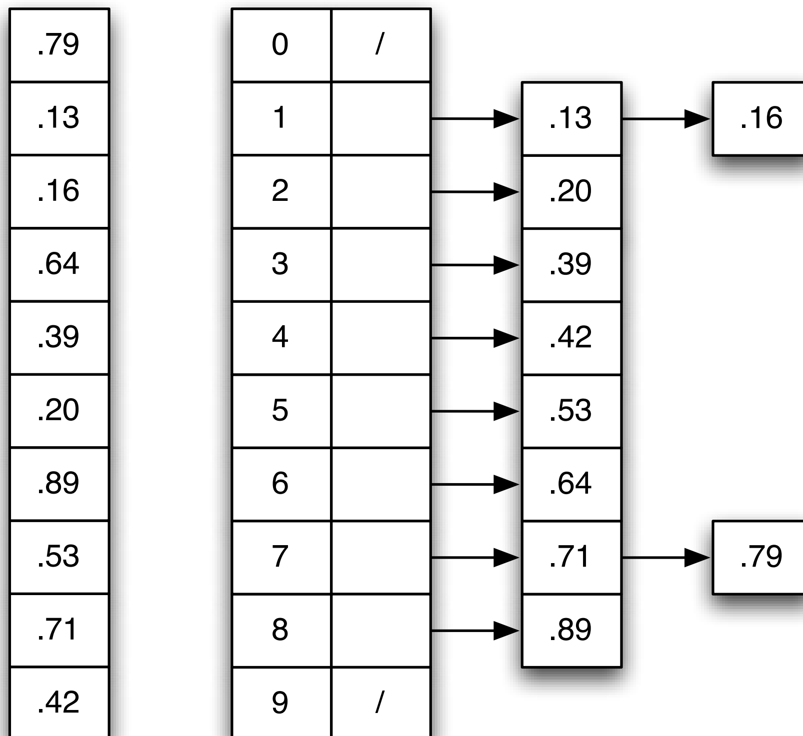


8.3 - 2

Insertion-Sort og *Merge-Sort* er stabile. For å gjøre hvilken som helst sorterings-algoritme stabil kan man f.eks. utvide hver nøkkel med objektets posisjon i input og bruke denne verdien når to nøkler er like. Dette krever $\Theta(n)$ ekstra minne.

Seksjon 8.4 - Treningsoppgaver

8.4 - 1



8.4 - 2

I verste tilfelle havner alle tallene i en bøtte, da vil *Insertion-Sort* kunne få $\Theta(n^2)$ kjøretid på den ene bøtta. For å unngå dette kan man bruke en $O(n \lg n)$ algoritme som *Merge-Sort* hvis det er mange elementer i en bøtte.

Kapittel 9

Seksjon 9.1 - Treningsoppgaver

9.1 - 1

Lag et binært tre med parvise sammenligninger av tallene, det minste tallet i paret blir med til neste runde osv. Vi finner det minste tallet med $n-1$ sammenligninger dvs. n løvnoder i et binært tre med $n-1$ interne noder der interne noder er sammenligninger. Stien fra roten av treet til det minste elemntet må inneholde det nest minste elementet, denne stien er $\lceil \lg(n) \rceil$ lang. Ved parhvis sammenlikning av elementene langs denne stien kan vi finne det nest minste elementet med $\lceil \lg(n) \rceil - 1$ sammenligninger. Det totale antall sammenligninger blir da: $n-1+\lceil \lg(n) \rceil - 1 = n + \lceil \lg(n) \rceil - 2$.

Seksjon 9.3 - Treningsoppgaver

9.3 - 1

Med grupper på k vil antall elementer mindre enn medianen av medianer være minst $\lceil \frac{k}{2} \rceil (\lceil \frac{1}{2} \lceil \frac{n}{k} \rceil \rceil - 2) \geq \frac{n}{4} - k$. Dette betyr at *Select* i verste fall vil bli kallt med maks $\frac{3n}{4} + k$ elementer. Følgende rekurens kan da settes opp:

$$T(n) \leq T(\lceil \frac{n}{k} \rceil) + T(\frac{3n}{4} + k) + O(n)$$

Ved substitusjon oppnår vi en nedre grense for k da algoritmen blir lineær. Antatt $T(n) \leq cn$ for all mindre n har vi:

$$\begin{aligned} T(n) &\leq c\lceil \frac{n}{k} \rceil + c(\frac{3n}{n} k) + O(n) \\ &\leq \frac{cn}{k} + \frac{3cn}{k} + ck + O(n) \\ &= cn(\frac{1}{k} + \frac{3}{4}) + ck + O(n) \\ &\leq cn \end{aligned}$$

Vi ser at den siste linkningen kun holder for $k \geq 4$, vi har altså vist at algoritmen vil ha lineær kjøretid så lenge man bruker grupper på 4 eller mer.

9.3 - 3

Ved å bruke median-av-medianer fremgangsmåte for å velge pivot element kan *Quicksort* få garantert $O(n \lg n)$ kjøretid.