Algoritmer & Datastrukturer
Aflevering 3

Anders Bjerg Pedersen, Hold 2

7. marts 2007

Opgave 21-1

a)

Vi gennemgår proceduren på den givne sekvens med $n = 9$ og $m = 6$:

$Array\{\}:$	$extended\{\}:$
$\{4\}$	$\{\}$
$\{4, 8\}$	
$\{8\}$	$\{4\}$
$\{8, 3\}$	$\{4\}$
$\{8\}$	$\{4, 3\}$
$\vdots$	$3x$INSERT
$\{8, 9, 2, 6\}$	$\{4, 3\}$
$\vdots$	$3x$EXTRACT-MIN
$\{9\}$	$\{4, 3, 2, 6, 8\}$
$\{9, 1\}$	$\{4, 3, 2, 6, 8\}$
$\{9, 1, 7\}$	$\{4, 3, 2, 6, 8\}$
$\{9, 7\}$	$\{4, 3, 2, 6, 8, 1\}$
$\{9, 7, 5\}$	$\{4, 3, 2, 6, 8, 1\}$

b)

Vi ser hurtigt, at der ikke kan blive skrevet mere end ét element på hvad plads i *extracted*, da mængden K_j slettes efter hver gang, i er blevet fundet i denne mængde. Det er altså ikke muligt at finde to tal f.eks. i mængden K_5, da denne slettes, efter det første element er fundet og eventuelt extracted.

Derefter vil vi vise, at i er lig $\min U_j$, hvor $U_j = \bigcup_{k=1}^{j} K_k \backslash extracted[1, \ldots, j-1]$. Altså at i er det mindste element i foreningsmængden af $K_1, \ldots, K_j$, fraregnet elementerne i *extracted*. Antag hertil, at i ikke er det mindste element, altså at der findes et element $h \in U_j$ med $h < i$. Da vores loop kører opad efter i, vil vi allerede have behandlet dette h. Altså må det enten befinde sig i mængden K_{m+1} og dermed ikke i U_j, eller også må det være blevet extracted, hvormed det heller ikke ligger i U_j. Dette er i begge tilfælde

en modstrid, altså må i være det mindste element i U_j.

Og resten må stå hen i det uvisse...

c)

Implementationen af Extract-Min kan med fordel laves med disjoint set forests. Her ville K-mængderne blive repræsenteret af disjoint set forests, netop fordi vi har at gøre med de n forskellige tal i mængden $\{1, \ldots, n\}$. Hver element ville desuden have 3 properties: *key*, *previous* og *next*. *key* vil svare til vores j, dvs. det j der indgår i den mængde K_j, hvor elementet ligger. *prev* refererer til repræsentanten for mængden K_{j-1}, og tilsvarende refererer *next* til repræsentanten for mængden K_{j+1}.

Vi laver så lidt preprocessing ved at oprette ét set til hvert element i $\{1, \ldots, n\}$ med MAKE-SET(i), $i = 1, \ldots, n$ og derefter lave Unions, så vi opnår de forskellige K_j-mængder. Under disse Unions sættes den nye repræsentant til den af de to repræsentanter, der har den største værdi af *key*. Lad os for nemheds skyld antage, at a er den mindste repræsentant, og b er den største. Så sætter vi *previous* af den nye repræsentant til *previous* af den mindste og lader mængden "inden"den mindste pege på det nye set som *next*. De samme to ting gøres så "modsat"opad til den næste mængde.

For at implementere dette i algoritmen ser vi nu, at key[FIND-SET(i)] vil give os det j, vi har brug for i linie 2. For at finde den første ikke-tomme mængde efter j kan vi følge *next* fra FIND-SET(i). En løs implementation kunne altså se nogenlunde således ud:

Algorithm 1 OFF-LINE-MINIMUM(m, n)

```
1: for i ← 1 to n do
2:     j ← FIND-SET(i)
3:     if key[j] ≠ m + 1 then
4:         extracted[j] = i
5:         UNION(next[j], j)
6:         delete K_j
7: return extracted
```

Med hensyn til køretiden skal vi jo lave $n - m$ UNION-operationer i starten, efter vi har lavet n MAKE-SET-operationer. Da der maksimalt skal udtages m elementer i sekvensen, kan vi højst skulle bruge UNION m ekstra gange i **for**-loopet. Så kommer vi op på maksimalt n UNION-operationer. Desuden er der n FIND-SET-operationer i **for**-loopet. Hvad det lige giver af køretid, har jeg ikke kunnet finde frem til...