Høgskolen i Østfold

Løsningsforslag, første og ikke helt komplette utkast (småbugs kan forekomme)! EKSAMEN

Emnekode: ITF20006 000	Emne: Algoritmer og datastrukturer
Dato: 19. mai 2010	Eksamenstid: 09:00 til 13:00
Hjelpemidler: 8 A4-sider (4 ark) med egne notater	Faglærer: Gunnar Misund

Oppgavesettet består av 5 sider inklusive denne forsiden. Kontroller at oppgaven er komplett før du begynner å besvare spørsmålene.

I implementasjonsoppgaver gir Java-kode best uttelling, men godt dokumentert pseudo-kode kan også brukes. Gjør dine egne forutsetninger hvis du føler behov for det.

Oppgavesettet består av 4 oppgaver, med tilsammen 13 deloppgaver. Hver hovedoppgave er vektet, og alle deloppgavene i en oppgave teller likt. Alle oppgavene skal besvares. Ved sensur teller hver deloppgave like mye. Karakteren fastsettes dog på basis av en helhetsvurdering av besvarelsen.

SKRIV TYDELIG :-)

Sensurdato: 16. juni 2010

Karakterene er tilgjengelige for studenter på studentweb senest 2 virkedager etter oppgitt sensurfrist. Følg instruksjoner gitt på: www.hiof.no/studentweb

Oppgave 1: 20%

A) Hva er effektiviteten i O-notasjon til følgende funksjon (begrunn svaret):

$F(n) = 13 + (14n^2 + \log_3 n) / n$

Leddet med høyest grad er $14n^2/n = 14n$, vi stryker konstanten 14, og vi får $O(n)$, lineær orden.

B) Forklar kort hva binærsøk er, og redegjør for effektiviteten til slike søk.

Et binærsøk forutsetter et sortert mengde. Vi starter med å dele mengden i to like store delder, og finner ut om det vi søker etter må ligge i den «minste» eller «største» mengden. Deretter fortsetter vi å søke på samme måte i den mengden som elementet må befinne i. For hver iterasjon deler vi altså mengden vi søker i to like store deler, og i verste tilfelle finner vi ikke elementet, og har da søkt gjennom et logaritmisk antall delmengder.

Ordenen er altså $O(\log n)$.

C) Implementer metoden

```
void delete(int[] arr, int idx)
```

som sletter element nr `idx` *i tabellen* `arr`*. Angi metodens effektivitet, inkludert begrunnelse.*

```
void delete(int[] arr, int idx) {
  // We are not asked to return a new array
  // We just shift the last part of the array one element left
  // If idx is last element, we do nothing :)
  for (int i = idx; i < arr.length-1; i++) {
      arr[i] = arr[i+1];
  }
}
```

Vi kopierer elementene med med indeks større enn idx en plas til venstre, ett for ett. I verste tilfelle er idx lik 0, må vi altså flytte alle elementene...dette gir linær orden, $O(n)$.

D) Implementer metoden

```
void delete(ListNode head, ListNode tail, ListNode del)
```

som fjerner noden med referansen `del` i en dobbeltlenket liste der første node er `head`, og siste node er `tail`. Du kan anta at `del` ikke er første eller siste node. Angi metodens effektivitet, inkludert begrunnelse. Se vedlegg.

```
public static void delete(ListNode head, ListNode tail, ListNode del)
{

      del.prev.next = del.next;

      del.next.prev = del.prev;

}
```

Vi har allerede referansen til element som skal slettes, så vi trenger ikke lete den fram. Dermed trengere vi bare å «bypass»'e den. Dermed blir det konstant orden, $O(1)$.

E) Betrakt funksjonen

```
void func (int[] arr) {
      int i = arr.length-1;
      int j = 0;
      while(i >= 0) {
            for (int k = 0; k <= i; k++) {
                  j += arr[k];
            }
            i--;
      }
}
```

Angi funksjonens effektivitet, inkludert begrunnelse.

Den ytre løkka går kjøres for hvert element i tabellen. I første gjennomløp kjører den indre løkka N ganger der N er antall elementer i arr. Neste gang kjøres den N-1 ganger, også videre, og siste gang kjøres den 1 gang. Dette gir totalt N+(N-1)+(N-2)+ .... + 1 ganger, dvs. summen av de N første heltallene, som jo er $(N^2-N)/2$, som gir oss kvadratisk orden, $O(N^2)$.

Oppgave 2: 30%

*A) Foreleser forteller begeistret at han har pønsket ut en genial sorteringsmetode, som ved en kløktig kombinasjon av trær og grafer viser seg å være av **lineær** orden. Hva tror du om dette?*

Foreleser er på bærtur, det er ikke mulig å finne en sorteringsalgoritme (for en generell input uten spesielle særtrekk) som gir bedre enn loglineær orden, $O(NlogN)$.

B) Forklar hvordan utvalgssortering foregår. Anvend metoden på følgende sekvens, og illustrer hvert trinn.

13 - 8 - 56 - 1 - 9 - 41 - 18

Diskuter metodens effektivitet.

Vi bygger opp en sortert sekvens ved gjentatte ganger finne og ta ut minste element i input-mengden og legge det til på slutten av den sorterte mengden.

INPUT	SORTERT
13 – 8 – 1 – 9 – 41 – 18	
13 – 8 – 9 – 41 – 18	1
13 – 9 – 41 – 18	1 – 8
13 – 41 – 18	1 – 8 – 9
41 – 18	1 – 8 – 9 – 13
41	1 – 8 – 9 – 13 – 18
	1 – 8 – 9 – 13 – 18 – 41

Orden er kvadratisk, $O(N^2)$, fordi vi for hver innsetting må gå gjennom hele det som er igjen av input, som altså blir lik summen av de N første heltall.

C) Implementer metoden ListNode sort(ListNode head) som returnerer den første noden i en sortert, dobbeltlenket liste, der nodene er hentet fra listen med start head. Sorteringen skal foregå etter utvalgsprinsippet. Forklar hva som eventuelt skjer med den originale input-lista. Se vedlegg.

Denne oppgaven kan være litt fiklete hvis man skal dekke alle spesialtilfellene, og en blanding av java/kommentarer/pseudokode kan nok være greiest for de fleste, og vil også gi bra uttelling.

Den originale lista blir «ødelagt», fordi disse referansene nå inngår i den sorterte lista, dvs. at prev og next referansene er endret.

```
public static ListNode sort(ListNode head) {
    // If empty or single item input, return without doing anything :)
    if (head == null || head.next == null) {
        return head;
    }
    ListNode sortedHead = null;
    // Keep track of last node in sorted list
    // Otherwise algorithm will be of cubic order!
    ListNode sortedTail = new ListNode();
    ListNode inputHead = head;
    while (inputHead != null) {
        // Find smallest
        ListNode curr = inputHead;
        ListNode smallest = curr;
        while (curr != null) {
            if (curr.val < smallest.val) {
                smallest = curr;
            }
            curr = curr.next;
        }
        // Remove smallest from input
        if (smallest.next == null
              && smallest.prev == null) { // Treat single item
            inputHead = null;
        }
        else if (smallest == inputHead) { // Treat head as special case
            inputHead = smallest.next;
            inputHead.prev = null;
        }
        else if (smallest.next == null) { // Treat tail as special case
            smallest.prev.next = null;
        }
        else {
            smallest.prev.next = smallest.next;
            smallest.next.prev = smallest.prev;
        }
        // Append smallest in sorted list
        if (sortedHead == null) { // Treat first append as special case
            smallest.prev = null;
            smallest.next = null;
            sortedHead = smallest;
            sortedTail = smallest;
        } else {
            sortedTail.next = smallest;
            smallest.prev = sortedTail;
            sortedTail = smallest;
        }
        // Terminate the list
        sortedTail.next = null;
    }
    return sortedHead;
}
```

Oppgave 3: 20%

A) Bygg opp en MaxHeap ved å sette inn disse elementene (i gitt rekkefølge):

B - G - Y - F - A - Z

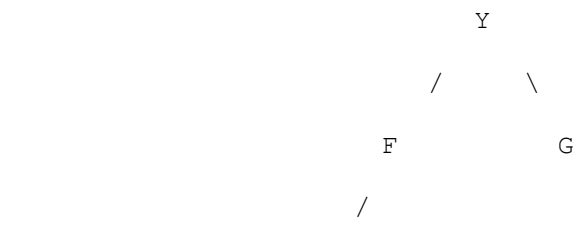
Fjern deretter det største elementet. Illustrer de ulike trinnene ved både innsettingene og fjerningen.

Sett inn rot:

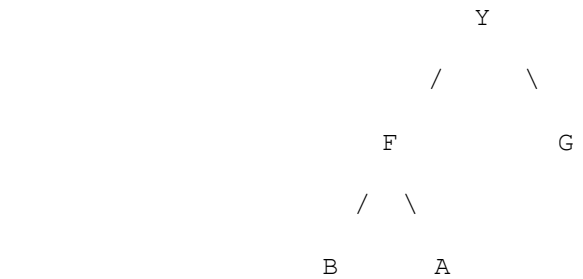
Sett inn G og boble opp:

Sett inn Y og boble opp:

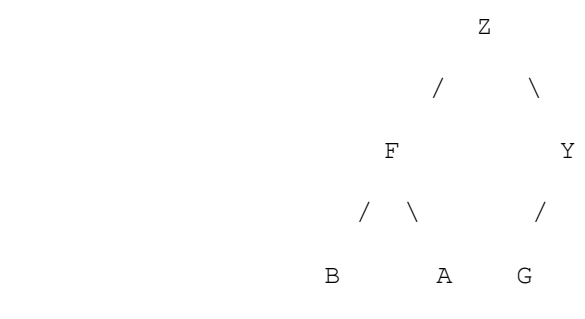
Sett inn F, og boble opp:



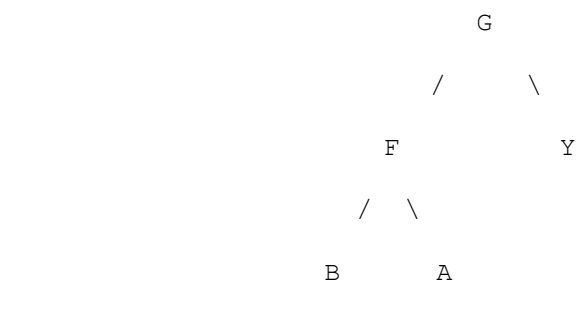
Sett inn A, ingen bobling:



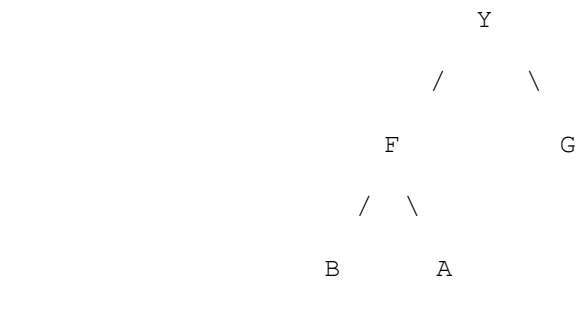
Sett inn, bobling opp til rota:



Ta ut Z (bytte med sist innsatte, G, slette sist inssatt)



Bobler nedover (med største barn)



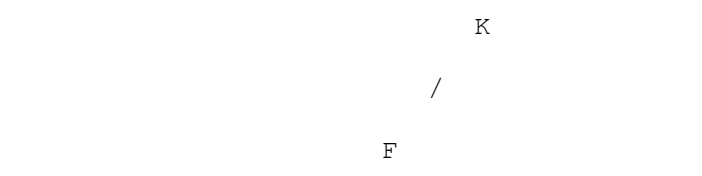
B) *Bygg opp et AVL-tre (binært, balansert søketre) ved å sette inn følgende elementer (i den gitte rekkefølgen):*

K - F - R - T - V - M

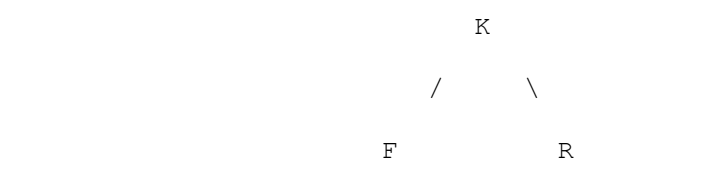
Illustrer de ulike trinnene i prosessen, og redegjør for hvilke regler du bruker.

Sett inn rot:

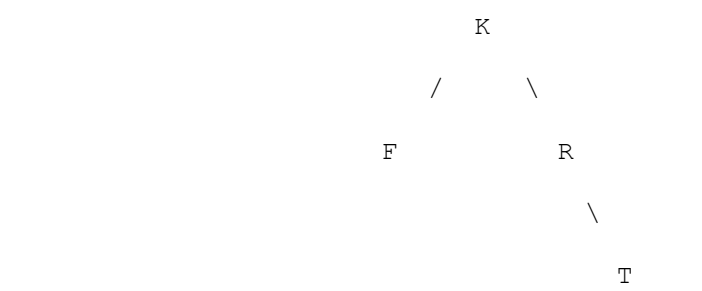
Sett inn F, balanse OK:



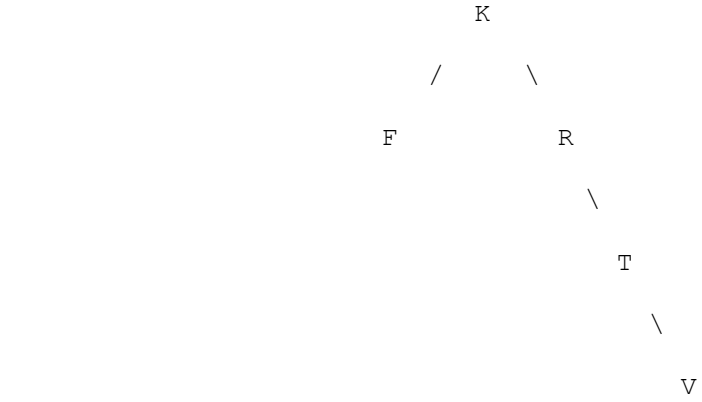
Sett inn R, balanse OK:



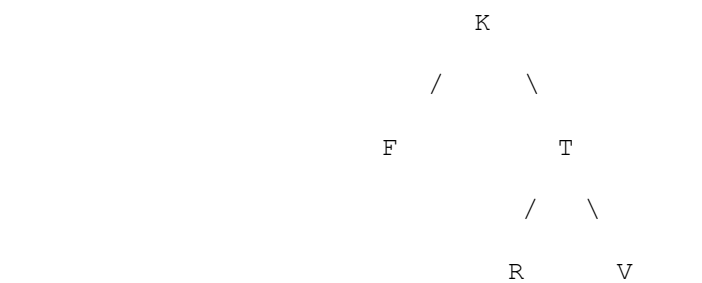
Sett inn T balanse OK:



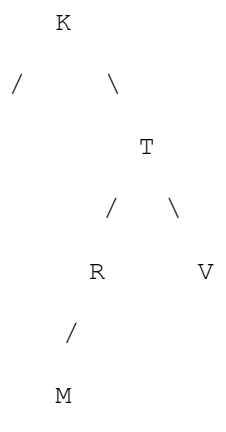
Sett inn T, R har +2 i balanse (type høyre/høyre):



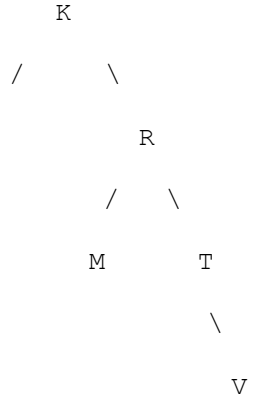
Må ta en enkel venstre rotasjon:



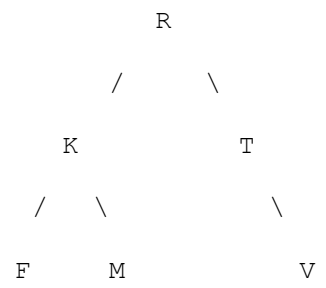
Setter in M, får ubalanse (+2) i K, type venstre/høyre



Første enkel høyre rotasjon om T

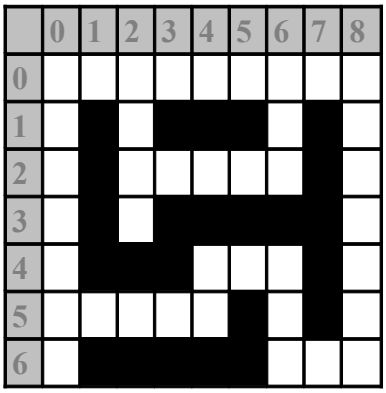


Deretter enkel venstre rotasjon om K



Oppgave 4: 30%

Vi har gitt følgende labyrint på matriseform (9 kolonner i x-retning, 7 rader i y-retning, svart markerer blokkerte ruter, grå markerer indeksene for kolonnene og radene):



Den rekursive metoden **move** kan benyttes for å finne en vei gjennom labyrinten (se vedlegg).

A) Hva blir skrevet ut til skjerm ved kallet `move(maze, 0, 0, 8, 6)`*, der* `maze` *er labyrinten over? Du kan stoppe når du har kommet fram til mål. Forklar kort hva som skjer videre etter at man har funnet fram til mål første gang. Bruk gjerne illustrasjoner.*

```
1: 0,0
2: 0,1
3: 0,2
4: 0,3
5: 0,4
6: 0,5
7: 0,6
OBS: Her er det backtracking!
8: 1,5
9: 2,5
10: 3,5
11: 4,5
12: 4,4
13: 5,4
14: 6,4
15: 6,5
```

```
16: 6,6
17: 7,6
Found!
```

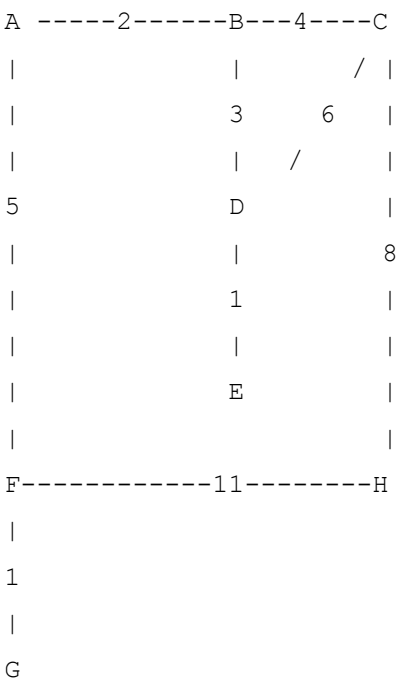
Etter dette fortsetter rekursjon ved å backtracke helt tilbake til start, og kjører rekursjonen via 1,0 og videre. Her kunne det vært lurt å tegne hele labyrinten, og skissere resten av rekursjonen ved å tegne piler eller lignende.

B) Forklar med tekst og illustrasjoner hvordan labyrinten over kan modelleres som et nettverk. Diskuteter spesielt hvordan vi kan minimere antallet noder.

Hver celle har 4 naboer, eller mindre hvis noen av nabocellene er blokkert eller utenfor labyrinten. Kantenes lengde blir da 1. Dette gir unødvendig mange noder. Det holder med å lage noder av kun de cellene der vi har et «kryss» eller «blindvei», og summere antallet celler imellom som lengden på kantene. La oss kalle nodene for:

```
A (0,0), B(2,0), C(6,0), D(2,2), E(2,3), F(0,5), G(0,6), og H(8,6)
```

Dette gir følgende nettverket (inkludert lengden på kantene):



C) Vi definerer lengden på en vei gjennom labyrinten som antallet flyttinger fra en celle til en nabocelle. Bruk Dijkstras algoritme for å finne korteste vei gjennom labyrinten over (fra `0,0` til `8,6`), basert på nettverksmodellen du har foreslått i forrige deloppgave. Forklar hva som skjer i hvert steg, gjerne ved å bruke en passende tabell.

- Ferdigbehandlede noder er streket under
- Oppdaterte noder (der veien er blir kortere enn før) er uthevet med fet skrift.

	a	b	c	d	e	f	g	h	Kommentarer
Init	a: 0	-: ∞	-: ∞	-: ∞	-: ∞	-: ∞	-: ∞	-: ∞	Avstanden fra a til a settes til 0. De andre avstandene settes til uendelig, og deres "via"-noden settes til "-"
1	a: 0	**a: 2**				**a: 5**			Vi velger noden med kortest avstandsverdi, dvs. a. Dernest oppdaterer vi avstanden til a ("via a") i alle nabonodene som ikke er ferdigbehandlet, dvs,. b og f. Noden a er etter dette ferdig behandlet, og vi markerer med å sette strek under den.
2		a: 2	**b: 6**	**b: 5**					Vi velger node b (kortest vei). For alle naboene til b som ikke er ferdigbehandlet, dvs c og d, sjekker vi om veien til a er kortere via b enn den til nå korteste veien, og det er den, og vi oppdaterer.
3						a: 5	**f: 6**	**f: 16**	Vi velger en av de med kortest avstand, f.eks. node f. For alle naboer som ikke er ferdigbehanldet (g og h). Vi oppdaterer begge.
4				b: 5	**d: 6**				d har nå kortest avstand. Vi oppdaterer e, men ikke c
5			b: 6					**c: 14**	Vi velger f.eks. c, og må oppdatere h
6					d: 6				Vi velger f.eks. e, ingen naboer er ubandlet.
7							f: 6		g har kortest avstand, men ingen ubandlede naboer.
8								*c: 14*	kun h gjenstår, og vi er framme!

Vedlegg

```
public class ListNode {
      public int val;
      public ListNode prev;
      public ListNode next;
}

public void move(boolean[][] maze, int sx, int sy, int dx, int dy) {
      // Blocked cells are initially marked as "false";
      // sx/sy represents the current move,
      // and dx/dy is the destination cell
      if (!valid(maze, sx, sy)) {
            // Return if we are outside the labyrint
            // or the cell value is false
            return;
      }

      // Assume we have a global counter, initially set to zero
      moveNo++;
      System.out.println(""+moveNo+": "+sx+","+sy);
      if (sx == dx && sy == dy) {
            System.out.println("Found!");
            return;
      }
      maze[sx][sy] = false;
      move(maze,  sx,     sy + 1, dx, dy);
      move(maze,  sx - 1, sy,     dx, dy);
      move(maze,  sx,     sy - 1, dx, dy);
      move(maze,  sx + 1, sy,     dx, dy);
}
```

Slutt på oppgavesettet.