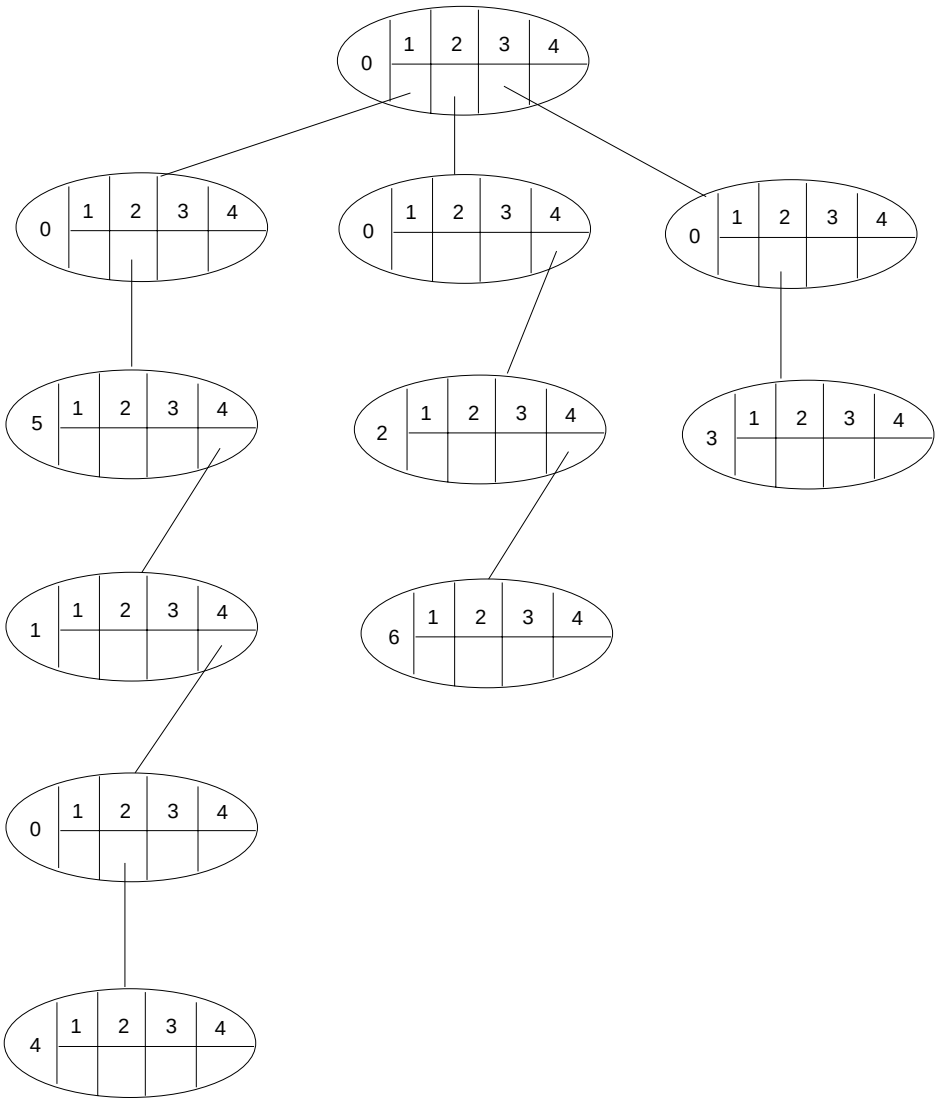
IN 115

Fasitforslag til Eksamen 1997
Omskrevet til Java

1. april 2000

Oppgave 1

1-a



Figur 1: Treet etter innsetting

1-b

I oppgaveteksten ble det definert hva det vil si at en bokstav finnes, nemlig at nesteb-pekeren assosiert til bokstaven var forskjellig fra null. Dette burde man da benytte. I simula-fasitforslaget er det to løsninger av denne oppgaven. Jeg har bare tatt for meg den hvor man tar en og en bokstav og ser hva som som skal gjøres her:

```
  /** Sette inn alle ordene i matrisen */
  static void insertAll() {
    for (int i = 1; i < lengde.length; i++)
      insert(i,rot);
  }

  /** Sette inn et ord i treet */
  static void insert(int nr, Node r) {
    int le;
    Node hj;

    le = lengde[nr];
    hj = r;

    for (int j = 1; j <= le; j++) {
      if (hj.nesteb[finnr(Ord[nr][j])] == null)
hj.nesteb[finnr(Ord[nr][j])] = new Node();

      hj = hj.nesteb[finnr(Ord[nr][j])];
    }

    if (hj.index == 0)
      hj.index = nr;

    //Hvis ordet alt var der gjør vi ingenting
  }
```

1-c

```
  /** Rekursiv metode for alfabetisk utskrift */
  static void alfaliste(Node no, int plass) {
    if (no.index != 0)
      System.out.println((new String(ford)).substring(0,plass));
    for (int j = 1; j <= 4; j++) {
      if (no.nesteb[j] != null) {
ford[plass] = finnb(j);
alfaliste(no.nesteb[j], plass+1);
```

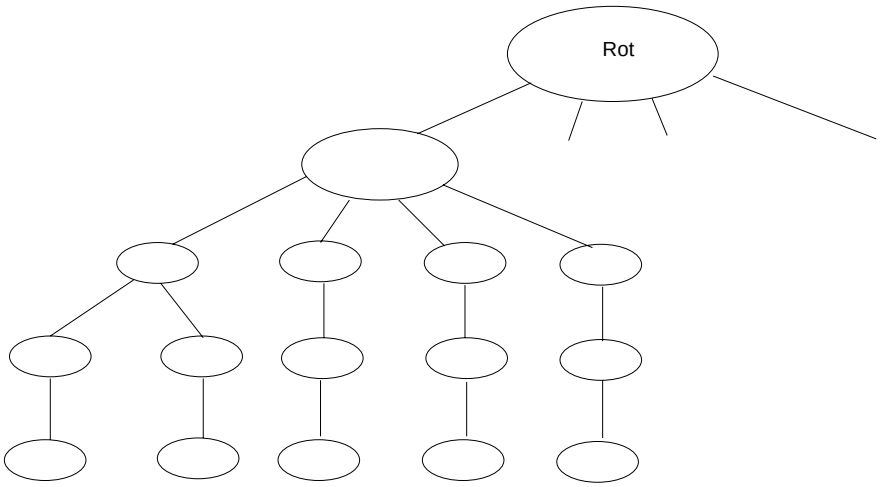
```
      }
    }
  }
```

1-d

Det minimale antall noder får vi ved at enhver node utenom roten "holder" et ord (index!=0). Da får vi 21 noder.

Dette kan oppnåes på mange måter. En måte er å bare se på alle kombinasjoner av 1 -og 2 bokstavert sekvenser.

Det maksimale treet får man når så få noder som mulig deler prefiks og lengden på ordene er maksimal, altså 4. Prinsippet her er at spredning høyt oppe i treet gir flere noder. (Ingen eller lite felles prefiks.) Vi får da maksimalt antall noder ved å splitte så mye som mulig fra rot og nedover. Vi får 5 ord i hvert av subtrærne til rot på følgende måte:



Figur 2: Maksimalt tre

der bare løvnodene har indeks forskjellig fra 0. Dette gir maksimalt 15*4 + 1 = 61 noder.

Et vanlig binærtre med verdier i nodene må ha 20 noder.

Den minimale høyden er når treet er så balansert som mulig, og vi får da høyde 4.

Den maksimale høyden får vi ved et helt skjevt tre og den er da 19

(løvnodene har høyde 0).

Søking i denne strukturen er optimal: Vi får en søkelengde som er identisk med lengden av ordet. Uansett hvilken teknikk man bruker (også hashing!) for å søke etter tekster må man på en eller annen måte behandle hver bokstav i ordet. Problemet kan bli plassforbruk, da hele alfabetet vil kreve nokså store noder.

1-e

Den opplagte måten å optimalisere treet på i situasjonen gitt av oppgaveteksten er å "avslutte" ordet før tiden ved å la indeks peke inn i matrisen. Man må passe på at indeks er lik 0 for alle nodene nedover i den enkle stien. Vi kan altså kutte en sti ved en node hvis noden har kun ett subtre *og* indeks = 0, og subtreet har kun ett subtre og indeks = 0, og der dette igjen har kun ett subtre *og* indeks = 0, osv inntil vi treffer på løvnoden som må ha indeks som peker på slutten av ordet.

Programmeringen kan bli "fiklete", men ideen er søm følger: Når vi setter inn et nytt ord vil vi søke oss nedover i treet inntil hele ordet er sjekket eller til bokstaven ikke finnes. Den noden vi befinner oss i da er enten en *normalnode* eller en *optimalisert* node.

I det første tilfellet fortsetter vi innsettingen omtrent som for oppgave 1-b, bare at vi setter inn en optimalisert node som løvblad dersom dette er nødvendig.

I det andre tilfellet har vi fire mulige situasjoner: Det optimaliserte ordet er likt det vi søker på, eller de er ikke like og vi kan da finne et *splitt-punkt:* Vi får da tre situasjoner: Settinnord er prefiks til optord, optord er prefiks til settinnord, eller de har disjunkte "haler".

```
/** Optimalisert insert */
  static void optInsert(int nr, Node r) {
    int i, sp, lnr, lopt, opt, j;
    Node hj;

    i = 1;
    hj = r;
    lnr = lengde[nr];

    while (hj.nesteb[finnr(Ord[nr][i])] != null && i <= lnr) {
      hj = hj.nesteb[finnr(Ord[nr][i])];
      i++;
```

```
    }

    if (hj.index == 0 || lengde[hj.index] == (i-1)) {
      if (hj.index == 0) {
if (i <= lnr) {
  hj.nesteb[finnr(Ord[nr][i])] = new Node();
  hj = hj.nesteb[finnr(Ord[nr][i])];
}
hj.index = nr;
      }
    }
    else {
      sp = i;
      lopt = lengde[hj.index];
      opt = hj.index;

      while(Ord[nr][sp] == Ord[opt][sp] && sp <= lnr &&
    sp <= lopt)
sp++;

      if (sp > lnr && sp > lopt)
  // gjøre ikkenoe
;
      else {
for (j = i; j <= sp-1; j++) {
  hj.index = 0;
  hj.nesteb[finnr(Ord[nr][j])] = new Node();
  hj = hj.nesteb[finnr(Ord[nr][j])];
}
if (sp <= lopt) {
  hj.nesteb[finnr(Ord[opt][sp])] = new Node();
  hj.nesteb[finnr(Ord[opt][sp])].index = opt;
}
if (sp <= lnr) {
  hj.nesteb[finnr(Ord[nr][sp])] = new Node();
  hj.nesteb[finnr(Ord[nr][sp])].index = nr;
}
if (sp <= lopt && sp <= lnr)
  hj.index = 0;
else if (sp > lnr)
  hj.index = nr;
else
  hj.index = opt;
      }
```

```
    }
  }
```

1-f

Her kan selvfølgelig svarene variere en del. I oppgaven indikeres at vi har en øvre grense for hvor mange ord vi kan ha, men vi har ingen garanti for at matrisen er fylt opp.

Hvis man tar hensyn til den øvre grensen gitt i oppgaven er det greit å velge lukket hashing: Tabellstørrelsen bør være 1 1/2 til 2 ganger MAXO.

Hvis åpen hashing er valgt, bør forutsetningen være at man vet veldig lite om hvor mange ord det vil befinne seg i ord-databasen. Tabellstørrelsen her kan godt velges til MAXO.

Eksempler på "gode" hashfunksjoner vil være de som gir god spredning når man antar store tabellstørrelse. Et alternativ kan være funksjonen fra side 150 i Mark A. Weiss (den gamle pensumboken) som er et godt alternativ for tekster:

```
/* Hashfunksjon fra s150 i Mark A. Weiss */
private int hashValue(String str) {
  int isoval, result=0;

  for (int i = 0; i < str.length(); i++) {
    isoval = Character.getNumericValue(str.charAt(i));
    result = ((result*32) + Math.abs(isoval)) % hTab.length;
    }

  return result;
}
```

Denne har ulempen at man må beregne % ganske mange ganger. Det hadde vært raskere å ta en til slutt, men jeg er sannelig ikke sikker på om Java tillater overflow. Ellers bør ikke en tabellstørrelse være en toerpotens.

Oppgave 2

2-a

Faseinndelingen til den oppgitte grafen vil være:

```
Fase 1: 1
Fase 2: 2, 4, 5
Fase 3: 3, 6
Fase 4: 7
Fase 5: 8
```

2-b

Denne kan løses på mange måter, men de fleste løsningene vil basere seg på en **int innkanter[]** med plass til en plass til altall kanter til hver node. Denne blir talt ned ettersom nodene blir ferdigbehandlet.

I boken (den gamle) er det nevnt under kapittelet om korteste vei en teknikk som benytter seg av to bokser: De som er under prosessering og de som blir klare. Dette passer godt til fasene: Når vi behandler noder i fase **j** vil vi finne alle noder i fase **j+1**, og ingen andre kan klargjøres i fase **j**.

En enkel fifo-kø kan også benyttes, men man må på en eller annen måte markere fase-skillet. Dette kan gjøres ved hjelp av en hjelpevariabel i køelementene.

En tredje mulighet er å ikke benytte en kø i det hele tatt, da må man gå gjennom arrayen **innkanter[]** minst en gang for alle faser (To ganger i den mest rett frem løsningen). Denne løsningen er ikke så god som den første med hensyn på tidskompleksitet, og her ble det bedt om å ta hensyn til dette.

Her er et forslag med to "bokser":

```
public class Oppgave2_B {
  static int antn;
  static int innk[], tid[];
  static Kant node[];

  public static void main(String args[]) {
    antn = <et eller annet tall>
    innk = new int[antn];
    tid = new int[antn];
```

```
    node = new Kant[antn];

    <lese inn grafen på en eller annen måte>

  }


  /**
   * Skriv ut fasene i grafen tidligste avslutningstid
   */

  static void fase() {
    int faseStart, fase, nesteFs;
    Kant hj;
    KantElem b1, b2, hk;

    b1 = b2 = hk = null;

    //initialiserer innk[]
    for (int i = 0; i < antn; i++) {
      hj = node[i];
      while ( hj != null) {
innk[hj.til]++;
hj = hj.neste;
      }
    }

    //initialiserer b1 (startkøen)
    for (int i = 0; i < antn; i++) {
      if (innk[i] == 0) {
hk = new KantElem(i);
hk.neste = b1;
b1 = hk;
      }
    }

    faseStart = 0; nesteFs = 0; fase = 0;
    while (b1 != null) {
      System.out.print("Fase: " + fase + ": ");

      while (b1 != null) {
System.out.print(b1.node + "   " + faseStart+tid[b1.node] + "\n");

if (faseStart + tid[b1.node] > nesteFs)
```

```
  nesteFs = faseStart + tid[b1.node];

hj = node[b1.node];

while(hj != null) {
  innk[hj.til] = innk[hj.til - 1];

  if(innk[hj.til] == 0) {
    hk = new KantElem(hj.til);
    hk.neste = b2;
    b2 = hk;
  }
}
      }

      // b1 er tom - la oss gå til neste fase
      b1 = b2;
      b2 = null;
      faseStart = nesteFs;
      fase++;
    }
  }


}

class Kant {
  int til;
  Kant neste;
}


class KantElem {
  int node;
  KantElem neste;

  public KantElem(int node) {
    this.node = node;
  }
}
```

Hvis man ikke benytter kø kan man gå gjennom arrayen **innkanter[] (dvs innk[])** og markere noder og deres faser ved en negativ nummerering. Så når antall innkanter blir 0, markeres fasen ved det tilsvarende

negative nummer (dette finner man fra fasen til noden som ga opphavet til fjerning av den siste innkommende kanten).

Man kan også gå gjennom **innk[]** to ganger: Først markeres de som er klare (innkanter = 0) ved å sette -1 i arrayen. Dernest går man gjennom alle med -1 og behandler sidde. Ferdigbehandlet kan passelig være -2. Neste sveip tar så for seg alle med innk = 0 og setter til -1 osv..

Tidskompleksiteten på programmet over blir O(|E| + N) der |E| er antall kanter. Køene vi bruker er enkle lifo-lister og hver kant behandles bare en gang. Initialiseringen har orden O(N).

2-c

I programmet over ville vi ha fått en terminering der ikke alle nodene har blitt tildelt en fase. Dette vil skje i de aller fleste varianter.

2-d

Vi antar at **transform** lager en matrise der en kant er markert med 1 og ingen kant er markert med 0.

```
public class Oppgave2_D {
static int sti[][];

  public static void main(String args[]) {
    int Mnode[][]; //Matriserepresentasjonen av grafen
    int antalln;

    <initialisere Mnode an antalln>

    sti[][] = new int[antalln][antalln];
    transform(Mnode, antalln);
  }

  static void transform(int mnode[][], int antalln) {
    for (int k = 0; k < antalln; k++)
      for (int i = 0; i < antalln; i++)
if (mnode[i][k] > 0)
  for (int j = 0; j < antalln; j++)
    if (mnode[j][k] > 0 )
      if (mnode[i][k] + mnode[k][j] > mnode[i][j]) {
mnode[i][j] = mnode[i][k] + mnode[k][j];
sti[i][j] = k;
```

```
        }
    }
}
```

2-e

Den modifiserte Floyd-algoritmen vil gi oss lengden på den lengste veien i matrisen Mnode. Hvis det går en kant fra u til v i den opprinnelige grafen og Mnode[u][v] > 1 så finnes det en lengre vei fra u til v. Denne kanten kan da fjernes.

Vi må overbevise oss om at vi da ikke har fjernet en kant som blir brukt i en lengste vei.

(u,v) kan umulig bli brukt i den lengste veien fra u,v - da ville vi ha hatt sykler i grafen.

Anta så at vi har en lengste vei s fra a til b. Hvis s inneholder kanten (u,v) så kan vi finne s' slik at s' er lengre enn s ved å benytte den lengre veien fra u til v istedenfor kanten (u,v). Altså kan ikke (u,v) være med i en lengste vei.

Når det gjelder fase-inndelingen, så er en aktivitetsfase faktisk definert ut fra lengste vei: En aktivitet tilhører fase j+1 hvis og bare hvis den lengste av de lengste veiene fra startnoden er j. Siden fjerning av kanter ikke forandrer på lengste veier får vi samme faseinndeling.

2-f

Vi skal begrunne at algoritmen over virker. For det første så er det ikke sykler i grafen så alle veier er enkle. Invarianten vil da være:

Mnode(i,j) > 0 hvis og bare hvis den lengste veien fra i til j som bare bruker noder fra (1,2,...,k) finnes ved å bruke sti[i][j], og lengden på denne veien skal være lik Mnode[i][j].

Når k=0 (altså for første gjennomløp av den ytre løkken) vil vi ha at Mnode[i][j] > 0 hvis og bare hvis det går en kant fra i til j, dvs veien bruker ingen mellomnoder. Altså er invarianten OK før vi starter den ytterste løkka.

Vi antar så at invarianten er opprettholdt etter m gjennomløp av den ytterste løkka.

Nå øker vi k til m+1. Vi skal også anta at for alle verdier av i og j som vi har sett på til nå i det m+i'te gjennomløpet av ytterste løkke, så bidrar disse til bevaring av invarianten.

De innerste programsetningene vil da for alle i og j der vi vet av det går en vei fra i til m+1 sjekke om det går en vei fra m+1 til j og i såfall hvis summen av veiene er lengre enn Mnode[i][j] oppdatere matrisen.

Vi vet at en vei fra i til j som bare benytter seg av nodene (1,2,3,...,m,m+1) og som *benytter* m+1 vil bestå av en vei fra i til m+1 og en vei fra m+1 til j som *ikke* benytter m+1 som mellomnoder. (Grafen innehar ikke sykler, alle veier er enkle). Lengden på de lengste og selve stiene finner vi da i Mnode og sti ut fra induksjonshypotesen om at m gjennomløp opprettholder invarianten og at tidligere par (i',j') i dette gjennomløp bidrar til opprettholdelse. Vi har bare enkle veier i grafen vår, så vi er helle rikke avhengig av behandling av senere par i',j' for det m+1'te gjennomløp med hensyn på (i,j) for det m+1'te gjennomløp.

Vi får altså oppdatert Mnode[i][j] og sti[i][j] hvis og bare hvis vi har funnet en lengre enkelt vei gjennom m+1 som bare benytter seg av nodene (1,2,....,m,m+1), og dette er den lengste sådanne.

Enhver enkelt vei vil bestå av en sekvens av noder som vi besøker på veien, så den invarianten vi har sett på vil etter det **antalln**'te gjennomløpet implisere av Floyd-varianten er korrekt.