Løsnings forslag i java In115, Våren 1996

Oppgave 1a

For å kunne kjøre Warshall-algoritmen, må man ha grafen på nabomatriseform, altså en boolsk matrise B, slik at `B[i][j]=true` hvis det går en kant fra i til j. Om grafen er på en annen rimelig form er en omarbeidelse til nabomatrise en grei sak, og den vil ta tid $O(n^2)$.

Ved å kjøre Warshall-algoritmen vil vi så forandre nabomatrisen slik at `B[i][j]=true` blir true dersom det finnes en rettet vei fra i til j. For å sjekke om en node 'x' er i TA-kjernen til behøver man bare å se om alle `B[x][j]` (rad 'x') er sanne. For å sjekke om noden 'x' er i FA-kjernen kan man tilsvarende se om alle `B[i][x]` (kolonne 'x') er sanne.

Kommentar: Om man initsialiserer nabomatrisen med `true` i `B[i][i]` vil alt gå som over. Om man initsialiserer med `false` i `B[i][i]`, kan man også få `false` i `B[i][i]` etter Warshall (for de 'i' som ikke er med i en løkke), og man må da hoppe over hoveddiagonalen i sjekkinga av linjer og kolonner på slutten. Dette kan vel imidlertid regnes som detaljer som ikke kreves i en *skisse*.

Tidsforbruket blir:

- Det man trenger til en mulig omarbeidelse til nabomatrise: $O(n^2)$.
- Det man trenger for å kjøre Warshall's algoritme: $O(n^3)$.
- Det man trenger for å finne de rader og kolonner som er bare `true`: $O(n^2)$

Totalt blir dette $O(n^3)$.

Oppgave 1b

Vi utfører ett dybde først søk fra hver node, og ser om vi fra den noden kan nå alle andre noder.

```
// Grafen er representert med noder på formen:
class Node
{
  String navn;
  Node etterf[];
  boolean merke;
}

// I tillegg har vi en liste over alle nodene og
// noen hjelpevariable
Node G[];
int antSett

// Metoden vi ønsker å lage
public void skrivTAK(Node G_[])
{
  G = G_;
  for (int i=0; i<G.length; i++)
    {
      // Nullstiller merkene i hver node
      for (int j=0; j<G.length; j++) G[j].merke=false;
      antSett=0;

      reksok(G[i]);
      if (antsett == G.length)
        System.out.println(G[i].navn);
    }
}

public void reksok(Node n)
{
  int k;

  /* Setter merket, øker antallteller og fortsetter søket
   * i alle denne nodens utkanter dersom denne er ukjent. */
  if (!n.merke)
    {
      n.merke=true;
      antsett++;
      for (int i=0; i<n.etterf.length; i++)
        reksok(n.etterf[i]);
    }
}
```

Tidsforbruket til denne prosedyren kan beregnes slik: For hver node går vi igjennom og setter alle merker, *O(n)* tid, og gjør deretter ett dybdeførst søk fra denne noden, *O(k)*. Dette gjentatt *n* ganger. Mest korrekt vil det da bli å skrive det som *O(n(k+n))*. Vanligvis vil imidlerid *k* være større enn *n* og vi kan skrive det på formen: *O(n*k)*.

Oppgave 1c

At en node A er med i både TA-kjernen og FA-kjernen betyr at det finnes en vei fra enhver node til A og fra A en vei til hver av de andre nodenen. Dermed er det også slik at det (via A) finnes en vei mellom alle par av noder i grafen. Det er da lett å se at både TA-kjernen og FA-kjernen vil omfatte alle nodene i grafen.

Oppgave 1d

Som man vil se under har denne oppgaven også en svært enkel løsning, men en løsning rett frem etter oppgaven kan være:

```
Node G[];

public boolean inFAK(Node G_[], node n)
{
  G = G_;

  // Nullstiller merkene i samtlige noder
  for (int i=0; i<G.length; i++) G[i].merke = false;

  // Gjør et rekursivt søk fra samtlige noder
  for (int i=0; i<G.length; i++)
    if( !reksok(G[i], n) )
      return false;

  return true;
}

/* Går ut ifra at det ikke finnes noen vei fra denne noden
 * til den vi ønsker å nå, n, men returnerer true dersom
 * vi har funnet den. Hvis ikke fortsetter dybde først søket
 * som vil sitte igjen med true dersom noen av de videre
 * rekursive kallene ledet frem. Setter merket hvis det
 * finnes en vei fra denne til noden n slik at vi kan
 * terminerer senere rekursive kall på samme noden.
 */
public boolean reksok(Node nd, Node n)
{
  boolean vei = false;

  if ( nd==n || nd.merke ) return true;

  for (int i=0; i<nd.etterf.length; i++)
    vei |= reksok(nd.etterf[i]);

  if (vei) nd.merke=true;

  return vei;
}
```

Som nevnt over finnes imidlertid en enklere løsning. Ser man nøyere på oppgaven viser det seg at det er nok å sjekke at (1) noden ’n’ ikke har etterfølgere og (2) alle andre noder har minst en etterfølger (og egentlig følger (1) av (2) siden grafen er løkkefri). At (2) er et nødvendig krav for at ’n’ er i FA-kjernen er opplagt. A det er tilstrekkelig ser vi slik:

Anta at vi starter i en tilfeldig node x!=n. Vi velger så en tilfeldig kant ut av x og kommer til en ny node. Herfra velger vi igjen en tilfeldig kant ut osv. Denne prosessen må endo opp i ’n’ (den eneste uten ut kanter), ellers ville vi få en løkke senest etter å ha gått `G.length` steg. Altså finnes en vei fra ’x’ til ’n’, og siden ’x’ var tilfeldig valgt går det vei fra alle til ’n’.

Programmet her blir bare en enkel løkke, og det er nok å teste krav (2). Et svar langs disse linjer, med en rimelig forklaring er selvfølgelig et fullgodt svar (vel også med litt ekstra pluss, om forklaringen, etc er

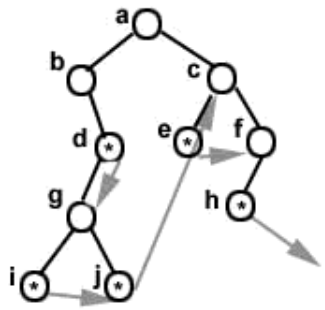
bra).

Oppgave 1e

For en løkkefri graf vil TA-kjernen inneholde maksimalt ett element (men kan også være tom). For anta at det var to noder x og y i TA-kjernen. Da ville det opplagt gå en rettet vei fra x til y og en fra y til x, og dermed ville vi ha en sykel i grafen. Dette blir altså umulig.

Et helt tilsvarende argument kan føres for at FA-kjernen kan maksimalt holde ett element i en rettet løkkefri graf.

Oppgave 2a



Oppgave 2b

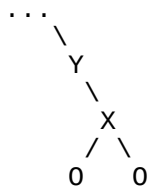
Svaret kan her gjøres svæt enkelt, og man behøver ikke engang teste på ’hpref’. Flere varianter vil imidlertid virke.

```
class Node
{
  String navn;
  Node hsub, vsub;
  boolean hpref;
}


/* Dersom noden ikke er null selv, returnerer vi venstre
 * subtre om det finnes og høyre dersom det ikke finnes
 */
public Node neste(Node n)
{
  if ( n!=null )
    if ( n.vsub!=null ) return n.vsub;
    else return n.hsub;
}
```

Oppgave 2c

Vi kan her f.eks ha følgende situasjon:



Anta her at man får en gitt ’x’, og skal finne dens etterfølger i postfiks rekkefølge, som altså er ’y’. Siden ’x’ har et ikke tomt høyre subtre (begge subtrærne er faktisk ikke tomme) er det ikke plass til eksplisitt å lagre peker til neste i postfiks rekkefølge. Uten foreldrepeker i nodene og uten tilgang til treets rot er det altså ikke mulig å få tak i y i det hele tatt.

Altså: Ideen lar set ikke uten videre overføre fra prefiks til postfiks.

Oppgave 2d

Det finnes her mange varianter som kan fungere. F.eks kan vi ha nonetesten i starten av metoden (som under) eller foran hvert kall. Filosofien for forrige pekeren kan også variere. Den kan f.eks. alltid (bortsett fra de første gangen) peke til den forrige noden i prefiks rekkefølge (som under), eller den kan bare peke ut forrige node om denne skal ha satt inn prefiks-peker i ’hsub’, og ellers være nulle.

```
Node forrige;

public void setPrefiks(Node rot)
{
  forrige = null;
  rekpref(rot);

  // Siste node i prefiks har alltid hsub==null
  if( forrige!=null ) forrige.hpref = true;
}

public rekPref(Node n)
{
  if( n!=null ) {
    if( forrige!=null )
      if( forrige.hsub==null ) {
        forrige.hsub = n;
        forrige.hpref = true;
      }
      else forrige = n;
    rekPref(n.vsub);
    rekPref(n.hsub);
  }
}
```

Oppgave 3a

Istedet for å tegne selve heap’en som ett binær tre er den her representert ved hjelp av arrayer, der barn er posistionert i 2n og 2n+1. Tall i fet tekst er de som er ferdig sorterte.

Initsielt

```
   5 3 7 2 1 4 6

Etter buildheap. Største tall, 7 er flyttet til rota
   7 3 6 2 1 4 5

Største tall tas ut, og man gjenoppretter heap strukturen. Tallet
settes så inn bakerst i heapen, og heapstørrelsen krymper med en
   6 3 5 2 1 4 7

og igjen..
   5 3 4 2 1 6 7

og igjen....
   4 3 1 2 5 6 7
```

Oppgave 3b

Løser dette med å legge det inn som en klasse SLU_sort og, den ytre metoden som konstruktør og de indre som objekt metoder.

```
class BitString
{
  public final static int K = ...;
  int bs[] = new bs[K];
}

public class SLU_sort
{
  BitString A[];

  public SLU_sort(BitString A_[])
  {
    A = A_;
    rekSort(0, A.length-1, 0);
  }

  public void bytt(int i, int j)
  {
    BitString buf = A[i];
    A[i] = A[j];
    A[j] = buf;
  }

  public int partition(int i, int j, int p)
  {
    int l=i, h=j;
    while( l<h ) {
      while( l<=j && A[l].bs[p]==0 ) l++;
      while( h>=i && A[h].bs[p]==1 ) h--;
      if( l<h ) bytt( l++, h-- );
    }
    return h;
  }

  public void rekSort(int i, int j, int p)
  {
    int m = partition(i, j, p);

    if( p<BitString.K )
      if( i<m ) rekSort(i, m, p+1);
      else if( m+1&it;i ) rekSort(m+1, j, p+1);
  }
}
```

Oppgave 3c

Om K=1 får vi bare ett kall. Om K=2 får vi 1+2 kall, og om K=3 får vi 1+2+4 kall. Generelt får vi opp til 2^K-1 kall, som er ett helt ok svar.

Man kan også observere at om N er liten vil antallet kall også være begrenset av N*K/2, siden det aldri blir gjort mer enn N/2 kall for hver p-verdi (på hvert nivå. Vi gjør jo aldri kall med mindre et intervall har minst to elementer.

Oppgave 3d

Tiden som brukes i hvert kall (utenom de videre rekursive kallene) er proposjonal med lengden av intervallet kallet skal arbeide på. Det er jo rett og slett arbeidet i `partition`. Dermed cio den totale tiden av kallene på ett bestemt nivå (en bestemt p-verdi) være *O(N)*, summen av intervall-lengdene for en gitt p er N. Maksimalt antall nivåer er K, altså er den totale tiden *O(N*K)*