Høgskolen i Gjøvik

Avdeling for Teknologi

# E K S A M E N

**FAGNAVN:** **Algoritmiske metoder I**

**FAGNUMMER:** **L 189 A**

**EKSAMENSDATO:** **11. desember 2000**

**KLASSE:** **99HINDA / 99HINDB / 99HINEA / 00HDESY ( 2DA / 2DB / 2EA / DESY )**

**TID:** **09.00-14.00**

**FAGLÆRER:** **Frode Haug**

**ANT. SIDER UTLEVERT:** **5 (inkludert denne forside)**

**TILLATTE HJELPEMIDLER:** **Alle trykte og skrevne.**

- Kontroller at alle oppgavearkene er tilstede.
- Innføring med penn, evt. trykkblyant som gir gjennomslag. Pass på at du ikke skriver på mer enn ett innføringsark om gangen (da det blir uleselige gjennomslag om flere ark ligger oppå hverandre når du skriver).
- Ved innlevering skilles hvit og gul besvarelse og legges i hvert sitt omslag. Oppgavetekst, kladd og blå kopi beholder kandidaten.
- Ikke skriv noe av din besvarelse på oppgavearkene.
- Husk kandidatnummer på alle ark.

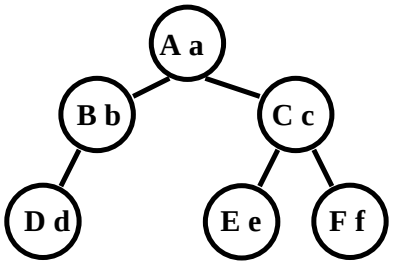
# Oppgave 1 (teori, 25%)

Denne oppgaven inneholder tre uavhengige oppgaver fra kap. 4, 11 og 16 i læreboka.

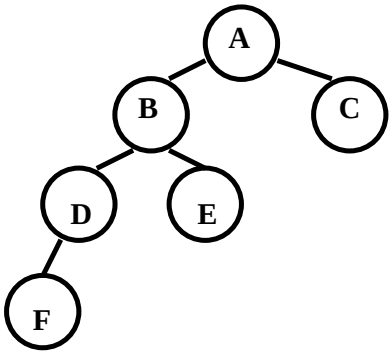
**a)** Såkalt «dobbelorder traversering» av binære trær er en kombinasjon av preorder og postorder traversering. Anta at vi i hver node har to tegn: `ch1` (stor bokstav) og `ch2` (liten bokstav). Vi kan skrive ut disse verdiene i dobbelorder rekkefølge på følgende måte:

```
void dobbelorder (node* t)  {
  if (t != z)  {
        cout << t->ch1;
        dobbelorder(t->l);
        dobbelorder(t->r);
        cout << t->ch2;
  }
}
```

**Skriv resultatet om vi anvender denne algoritmen på følgende tre:**




En spesiell type traverseringsmetode for binære trær produserer sekvensen «ABDFFFDDBEEEBACCCA» ved å anvendes på treet:




**Beskriv kort algoritmen som utfører denne traverseringen.**

**b)** Teksten «P U L T O S T K R E M» (i denne rekkefølge fra venstre til høyre, blanke regnes ikke med) skal heap-sorteres vha. bottom-up heap konstruksjon.
(Se fig.11.8, koden s.156 og fig.11.9 i læreboka.)
**Tegn opp heapens innhold etterhvert som heapen konstrueres og deretter sorteres etter denne metoden** (Dvs. lag en figur etter samme prinsipp som fig.11.9.)

**c)** Koden s.237 og teksten s.239 i læreboka gir oss følgende kode ved dobbel hashing:

```
const int M = 11;
int hash1(int k)   {  return  (k % M);        }
int hash2(int k)   {  return  (3 - (k % 3));  }

void insert(itemType v, infoType info)  {
   int x = hash1(v);
   int u = hash2(v);
   while (a[x].info != infoNIL)  x = (x+u) % M;
   a[x].key = v;  a[x].info = info;
}
```

«k» står for bokstavens nummer i alfabetet (1-29). Vi har en array som er 11 lang (indeks 0-10). Keyene «PULTOSTKREM» skal legges inn i denne arrayen. **Skriv opp hver enkelt key's returverdi fra både hash1 og hash2. Tegn også opp arrayen for hver gang en ny key legges inn.** (Innholdet i «info» trenger du ikke å ta hensyn til.)

# Oppgave 2 (teori, 25%)

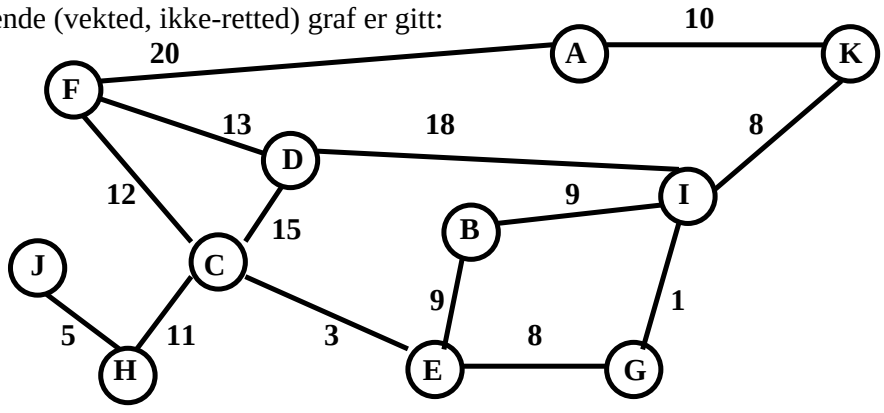
Denne oppgaven inneholder tre uavhengige oppgaver fra kap. 30, 29 & 31 i lærebok og om FSM.

**a)** Følgende kanter i en (ikke-retted, ikke-vekted) graf er gitt:

AB AC DE BD AF

Vi skal nå **utføre union-find m/path compression** (men *ikke* weight balancing) på denne grafen. Det er den første bokstaven som skal legges inn under den siste bokstaven i hver kantangivelse. Dvs. den *eneste* innmaten i if-setningen: `if (doit && (i != j))` blir: { `dad[j] += dad[i]-1; dad[i] = j; }` **Tegn opp innholdet i arrayen «dad» etterhvert** som skogen bygges opp (jfr. fig.30.9) vha. «find»-funksjonen s.447 i læreboka. Ut fra dette: **tegn også opp den resulterende union-find skogen** (dvs. noe lignende til nedre høyre skog i fig.30.8).

**b)** Følgende (vekted, ikke-retted) graf er gitt:




**1. Skriv en lovlig rekkefølge av nodene ut fra et Dybde-Først-Søk (DFS) som starter i node F.**
**2. Skriv en lovlig rekkefølge av nodene ut fra et Bredde-Først-Søk (BFS) som starter i node F.**
**3. Gi en kort begrunnelse for hvorfor kanten EG hører med i grafens minimums spenntre.**
**4. Tegn opp et minimums spenntre for grafen (du trenger ikke å angi «fringen» underveis).**

**c)** Følgende deterministiske endelige tilstandsmaskin (FSM) er gitt:

$M = ( \{ q_0, q_1, q_2, q_3, q_4 \}, \{ a, b, c \}, \delta, q_0, \{ q_3 \} )$

Transisjonstabellen:

| | | |
|---|---|---|
| $\delta( q_0, a ) = q_1$ | $\delta( q_0, b ) = q_4$ | $\delta( q_0, c ) = q_3$ |
| $\delta( q_1, a ) = q_0$ | $\delta( q_1, b ) = q_2$ | $\delta( q_1, c ) = q_3$ |
| $\delta( q_2, a ) = q_1$ | $\delta( q_2, b ) = q_2$ | $\delta( q_2, c ) = q_3$ |
| $\delta( q_3, a ) = q_3$ | $\delta( q_3, b ) = q_3$ | $\delta( q_3, c ) = q_3$ |
| $\delta( q_4, a ) = q_0$ | $\delta( q_4, b ) = q_4$ | $\delta( q_4, c ) = q_3$ |

**Tegn tilstandsmaskinen.**
**Er det mulig å tegne denne FSM'en enklere ? I så fall: gjør det.**
**Hva slags setninger/språk godtar den ?**

# Oppgave 3 (koding, 25%)

I en vekted, retted graf , kan hver node ha maksimalt 10 naboer. Nodene er representert slik:

```
struct node {
      int ant_naboer;       // Antall naboer, 0 <= ant_naboer <= 10
      int nabo[10];         // Nodenummere for alle direkte naboer.
      int vekt[10];         // Kantvekter.
};
```

Arrayene «nabo» og «vekt» inneholder informasjon om de direkte forbindelsene noden har til andre noder i grafen. Bare de «ant_naboer»-1 første plassene i disse arrayene er i bruk i hver node. Hvis f.eks. «ant_naboer» er lik 1, så vil bare indeks nr.0 i de to arrayene være i bruk. Dersom nabo[0] er lik 2 og vekt[0] er lik 5 i en node, betyr det at det går en kant med vekt 5 fra denne noden til node nummer 2. Alle kanter i grafen har vekter større enn 0 (null).

Det finnes i alt N noder i grafen, nummerert fra 0 til N-1. Hver node i grafen er det direkte tilgang til via arrayen «graf»: `node* graf[N];`

**I denne oppgaven skal du lage resten av programmet som legger data om kantene i en graf inn i et binært søketre.** Grafen er representert som beskrevet ovenfor. Det binære søketreet skal inneholde en «tre_node» for hver kant i grafen, med informasjon om kantvekt og om hvilke to noder kanten går fra og til. Nederst på hver gren i treet er det z-noder (som i læreboka). En «tre_node» er definert slik:

```
struct tre_node {
    int vekt;               // Kantvekt.
    int til, fra;           // Nodenummere i grafen for de to nodene som
                            //    er knyttet til hverandre med denne kanten.
    tre_node *l,*r;         // Pekere til venstre og høyre barn i treet.
};
```

Søketreet skal være sortert på kantvektene, slik at kantvekter som er mindre legges til venstre for en tre_node, og kantvekter som er større eller lik vekten i en tre_node legges til høyre for den.
**Løsningen din *skal* bl.a. inneholde funksjonen**: `tre_node* kant_tre();` **som bygger opp et søketre med kant-informasjon som beskrevet ovenfor, og returnerer en peker til roten i dette treet.** (Rekursive) funksjon(er) kan selvsagt benytte de globale variablene/deklarasjonene gitt i teksten ovenfor («node», «N», «graf», «z» og «tre_node»).

**NB:** Du skal *ikke* benytte flere/andre globale variable eller nye verdier inni struct'ene enn det som er angitt ovenfor !

# Oppgave 4 (koding, 25%)

En person mottar et gitt antall kronestykker som kalles INITIELT. Vedkommende har så to muligheter:

1. Be om, og motta ytterligere 53 kronestykker eller
2. Dersom vedkommende har *et partall* kronestykker *kan* (men er ikke nødt) hun/han levere fra seg halvparten av de kronestykkene vedkommende har.

For hver gang vedkommende utfører punkt 1 eller 2 ovenfor, kalles dette et «steg» i spillet.
Målet er å ende opp med *eksakt* 91 kronestykker på N steg eller færre.

F.eks: Dersom INITIELT er 99 og N er 4, vil følgende sekvens av steg nå målet:

99 $\longrightarrow$ 152 $\longrightarrow$ 76 $\longrightarrow$ 38 $\longrightarrow$ 91
Pkt.1 Pkt.2 Pkt.2 Pkt.1

**Skriv en rekursiv funksjon** `bool spill(int ant_kroner, int n)` **som returnerer 'true'/'false' til om det er mulig å nå målet på 91 kronestykker med N eller færre steg. Når den (etter evt. å ha funnet et positivt svar) trekker seg tilbake i rekursjonen, skal den skrive det antall kronestykker som er involvert i/på vedkommende steg.**

**Lykke til !**

**frode@haug.com**