Høgskolen i Gjøvik
Avdeling for teknologi

KONTINUASJONSEKSAMEN

FAGNAVN: **Algoritmiske metoder (I)**

FAGKODE: **L189A og L167E**

EKSAMENSDATO: **8. august 2003**

KLASSE(R): **01HIND* / 01HINE* / div. andre**

TID: **L189A: 09.00-14.00**
L167E: 09.00-13.00

FAGLÆRER: **Frode Haug**

ANTALL SIDER UTLEVERT: **4 (inkludert denne forside)**

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

- Kontroller at alle oppgavearkene er tilstede.

- Innføring med penn, eventuelt trykkblyant som gir gjennomslag. Pass på så du ikke skriver på mer enn ett innføringsark om gangen (det blir uleselige gjennomslag når flere ark ligger oppå hverandre).

- Ved innlevering skilles hvit og gul besvarelse, som legges i hvert sitt omslag.

- Oppgavetekst, kladd og blå kopi beholder kandidaten til klagefristen er over.

- Ikke skriv noe av din besvarelse på oppgavearkene. Men, i oppgavetekst der du skal fylle ut svar i tegning/tabell/kurve, skal selvsagt dette innleveres sammen med hvit besvarelse.

- Husk kandidatnummer på alle ark. Ta vare på dette nummeret til sensuren faller.

NB: De som tar L189A (3 vekttall) skal *løse alle* oppgavene.
Alle oppgavene er da vektlagt likt med 25%
De som tar L167E (2 vekttall) skal *ikke løse* oppgave nr. 2.
Oppgavene er da vektlagt med: 40%, 30% og 30%

Oppgave 1 (teori)

Denne oppgaven inneholder tre uavhengige oppgaver fra kap. 8, 12 og 15 i læreboka.

a) Vi skal utføre Shellsort på key'ene "BILLESUPPE". Jfr. kode s.109 i læreboka.
For hver gang indre for-løkke er ferdig (etter: `a[j] = v;`):
Tegn opp arrayen og skriv verdiene til 'h' og 'i' underveis i sorteringen.
Marker spesielt de key'ene som har vært involvert i sorteringen (jfr. fig.8.7 s.108).

b) Vi skal utføre rekursiv Mergesort på key'ene "BILLESUPPE". Jfr. kode s.166 i læreboka.
For hver gang tredje og siste for-løkke i koden s.166 er ferdig:
Tegn opp arrayen med de key'ene som har vært involvert i sorteringen (jfr. fig.12.1 s.167).

c) Legg key'ene "BRUNBILLESUPPE" (i denne rekkefølge fra venstre til høyre) inn i et 2-3-4 tre. **Tegn opp treet etterhvert som bokstavene legges inn**.
Gjør også om sluttresultatet til et Red-Black tre.

Oppgave 2 (teori) - Skal *ikke* løses av de som tar L167E (2 vekttall)

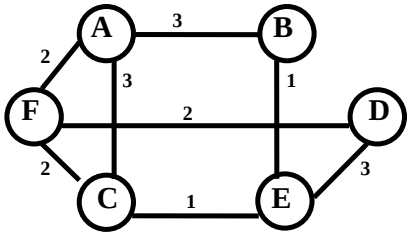
Denne oppgaven inneholder tre uavhengige oppgaver fra kap. 16, 30 og 31 i læreboka.

a) Vi har hash-funksjonen: `hash(k) = k mod 13`
der "k" står for bokstavens nummer i alfabetet (1-29). Vi har også en array med indeksene 0-12. **Tegn opp arrayen hver gang key'ene "R Ø D B I L L E S U P P E" (i denne rekkefølge fra venstre til høyre, blanke regnes ikke med) legges inn i den vha. linear probing.**

b) Følgende kanter i en (ikke-retted, ikke-vekted) graf er gitt:
AF BG FG DC AG BE GD BD FC CE

Vi skal nå utføre union-find på denne grafen. Tegn opp **arrayen "dad"s innhold etterhvert** som skogen bygges opp (jfr. fig.30.6) vha. "find"-funksjonen s.444 i læreboka. Ut fra dette: tegn også opp den resulterende **union-find skogen**
(jfr. nedre høyre skog i fig.30.5).

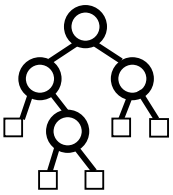
c) Følgende vektede (ikke-rettede) graf er gitt:



Hver nodes naboer er representert i en naboliste. Alle listene er *sortert alfabetisk*.
Tegn opp minimums spenntreet for denne grafen, etter at koden s.455 i læreboka er utført (der "priority" er lik "t->w").
Tegn også opp innholdet i prioritetskøen etterhvert som konstruksjonen av minimums spenntreet pågår (jfr. fig.31.4 i læreboka). **NB:** Husk at ved lik prioritet så vil noden sist oppdatert (nyinnlagt eller endret) havne først i køen.

Oppgave 3 (koding)

Denne oppgaven omhandler binære trær. Vi er bare interessert i trærnes *form*, ikke i hver enkelt nodes innhold (ID/key). Nederst på enhver gren er det (som vanlig) en referanse til z-noden. Vi kan f.eks. ha følgende tre:



Om vi traverserer et slikt tre postorder, og skriver en 'N' for hver reelle node og en 'T' for hver z-node, så får vi treet skrevet på en *utvidet postfix form*. For treet ovenfor ville utskriften blitt: TTTNNTTNN

Vi har følgende deklarasjoner/definisjoner:

```
struct node  {
  node *l, *r;               //  Pekere til venstre og høyre subtre/barn.
  // + andre data (som ID/key), men disse er i denne oppgaven irrelevante.
};
typedef node* itemType;
itemType z;
```

Samt hovedprogrammet:

```
int main()  {
  itemType root;             //  Peker til treets rot.
  char S[] = "TTTNNTTNN";    //  Utvidet postfix form for et tre.
  const int N = 4;           //  Antall reelle noder i dette treet.

  z = new node;              //  Lager/setter z-noden.
  root = (N > 0) ? gentre(N, S) : z;  //  Lar 'root' peke til generert tre.
  return 0;
}
```

Din oppgave er å **lage funksjonen** `itemType gentre(const int NN, char SS[])`

Et binært tre er på utvidet postfix form beskrevet av inn-parameteren 'SS'. **Hele poenget med funksjonen er å generere det aktuelle treet med 'NN' noder.**

Funksjonen *trenger/bør ikke* være rekursiv, samt at du *bør* nok bruke en stakk. Til dette kan du benytte deg av koden øverst på s.26 i læreboka (uten å gjenta/skrive av den i besvarelsen din).

Funksjonen må altså inneholde en referanse til og bruke stakken. Den må gå gjennom hele 'SS' (den er *ikke* 'NN' lang, den er "mye" lengre!) fra venstre mot høyre, og foreta ulike aksjoner ut fra om den finner en 'N' eller en 'T'. Retur fra funksjonen er en peker/referanse til det genererte treet.

NB1: Husk på property 4.3 s.39 i læreboka. Samt at man på ethvert punkt ved lesing, fra venstre mot høyre, av en slik utvidet postfix form *alltid* vil ha lest flere T'er enn N'er.

NB2: Husk at spesialtilfellet med at treet er tomt (N == 0) tar "main" seg av.

Oppgave 4 (koding)

En liste består av noder/elementer definert av:

```
struct node  {          //  Nodene/elementene i lista:
  char ID;              //  Dens ID/key.
  node* neste;          //  Peker til neste noden (etterfølgeren).
  node(char c)  {  ID = c;  neste = NULL;  }    //  Constructor.
};
```

Vi har også de globale pekerne:

```
node *forste, *siste;   //  Listens dummy hode(denne forblir uendret)og dens
                        //    nåværende siste element (denne endrer seg).
```

Den siste noden sin neste-peker refererer til NULL. Det første elementet i listen ('forste') er et "hode", og er ikke med i listen når den skrives ut. De listene vi skal operere på kan vi forutsette at består av "hode" og *minst ett* element til (dvs. de er *ikke* tomme).

<u>Oppgaven går ut på å skrive den *rekursive* funksjonen `void permuter(node* e)` som generere alle mulige permutasjoner av listen.</u>

Dette skal gjøres ved å flytte rundt på elementene i listen. Det er *ikke lov til* å benytte en hjelpe-array med pekere. Dermed blir koden fra EKS_06.CPP uaktuell å direkte gjenbruke, men du kan sikkert hente noen ideer der … Parameteren 'e' definerer den delen av lista som det aktuelle kallet skal arbeide på, dvs. funksjonen skal lage alle mulige permutasjoner av listen f.o.m. 'e->neste' t.o.m. 'siste'.
(For husk *hele* listen har et dummy "hode", og slik kan vi også betrakte alle sublister!)

For hver nye permutasjon kaller du nedenforstående funksjon - *alltid* med 'forste' som parameter:

```
void bruk_liste(node* n)  {   //  "Bruker" (skriver ut) nåværende liste:
  n = n->neste;               //  Hopper over "hodet".
  while (n != NULL) { cout << n->ID << ' ';   n = n->neste; } cout << '\n';
}
```

For å få venstre-rotert en (sub)liste så kan/bør du også benytte deg av funksjonen:

```
                                //  Flytter f->neste bakerst. Dvs. venstre-
void flytt_bakerst(node* f) {//    roterer lista etter den tilpekt av 'f':
                  //  Antar at f->neste aldri peker til NULL eller 'siste'!
  siste->neste = f->neste;   f->neste = f->neste->neste;
  siste = siste->neste;  siste->neste = NULL;
}
```

Hvordan selve lista bygges/oppstår trenger du ikke å tenke så mye på, det viktigste er at det hele i "main" starter med kallet `permuter(forste);`

Lykke til !

frode@haug.com