Høgskolen i Gjøvik

KONTINUASJONSEKSAMEN

FAGNAVN: **Algoritmiske metoder I**

FAGNUMMER: **L 171 A**

EKSAMENSDATO: **19. august 1999**

KLASSE: **97HINDA / 97HINDB (2DA / 2DB)**

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.

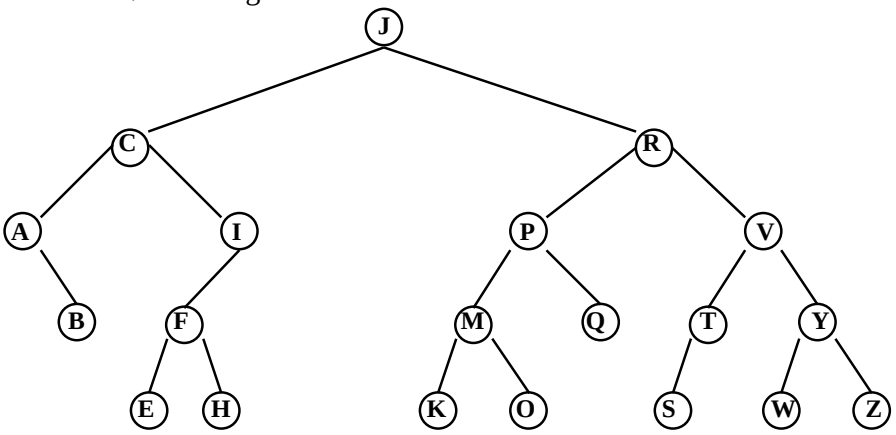
NB: Alle oppgavene teller likt, dvs. 25% vekt på hver av dem !

Oppgave 1 (teori)

Denne oppgaven inneholder tre uavhengige oppgaver fra kap. 9, 14 og 16 i læreboka.

a) Du skal utføre Quicksort på teksten "MEGASTOR". **Lag en figur tilsvarende fig. 9.3 s.119 i læreboka**, der du for hver rekursive sortering skriver de involverte bokstavene og markerer/ uthever hva som er partisjonselementet.

b) Følgende binære søketre er gitt :



Du skal nå fjerne ("remove") noen noder fra dette treet. **Tegn opp treet for hver gang og fortell hvilken av "if ..... else if ..... else"-grenene i koden s.210 som er aktuelle når du etter tur fjerner henholdsvis tegnene 'P', 'I' og 'V'.**

NB: Du skal for hver fjerning på nytt ta utgangspunkt i treet tegnet ovenfor. Dvs. på intet tidspunkt skal du, fra treet, ha fjernet to eller tre av de ovenfor skrevne bokstavene samtidig.

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

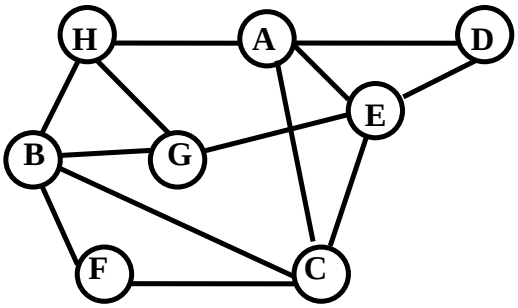
```
const int M = 11;
int hash1(int k)   {  return  (k % M);        }
int hash2(int k)   {  return  (4 - (k % 4));  }

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 "MEGASTOR" 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.** ("Info"s innhold trenger du ikke å ta hensyn til.)

Oppgave 2 (teori)

Denne oppgaven inneholder tre uavhengige oppgaver fra kap. 29, 30 og 31 i læreboka.

a) Vi har grafen :



a1) **Skriv opp nabomatrisen** for dette tilfellet.

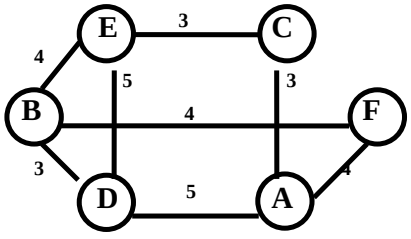
a2) **Tegn dybde-først søketreet** for dette tilfellet (dvs. ved bruk av nabomatrise), når "Search" (s.424) og "Visit" (s.427) fungerer som angitt i læreboka.

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

AH GH FC BG ED AE EG CE FB CA

Vi skal nå utføre union-find m/path compression og weight balancing på denne grafen.
Tegn opp arrayen "dad"s innhold 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).

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

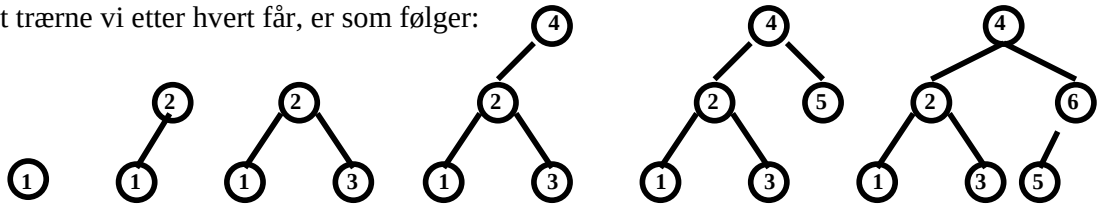


Hver nodes naboer er representert i en naboliste. Alle listene er sortert alfabetisk.
Tegn opp minimums spenntreet for denne grafen, etter at koden s.455 i læreboka er utført (der "priority" er lik "t->w").
Tegn også opp innholdet i prioritetskøen etterhvert som konstruksjonen av minimums spenntreet 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)

Vi skal her arbeide med et binært søketre. Vi skal stadig sette inn nye noder, men aldri ta noen ut. Innimellom innsettingene skal vi gjøre vanlige søk i treet. Poenget her er imidlertid at når en ny node skal settes inn, vil den alltid ha større verdi enn alle tidligere innsatte noder. Om vi hadde satt noder inn på vanlig måte, ville dette gitt et helt skjevt tre (med bare høyrebarn, dvs. en liste). Men, siden vi vet at vi nå vil sette inn nodene i stigende rekkefølge, kan vi utnytte dette til å lage et ganske godt balansert tre. Husk altså at treet hele tiden (mellom innsettinger) må være et binært søketre.

Om vi tenker oss at verdiene i de nodene vi skal sette inn er 1, 2, 3, 4, 5 og 6, så vil vi organisere oss slik at trærne vi etter hvert får, er som følger:



Algoritmen vi hele tiden skal følge ved innsetting, er som følger: Vi sier at en node er "ubalansert" dersom høyden av dens to subtrær er ulike. Ved innsetting av en ny node N (som altså har større verdi enn alle nåværende noder i treet) følger vi først den veien vi ville gått ned gjennom treet om vi lette etter en node med denne verdien, altså hele tiden til høyre. På veien registrerer vi de noder som er ubalanserte, og velger til slutt den siste ubalanserte noden vi fant. Denne kalles U. Vi setter så N inn som ny høyre subnode til U, og lar det gamle høyre subtreet til U bli nytt venstre subtre til N. Hva som skal gjøres dersom man ikke finner noen ubalanserte noder, må du selv finne ut av.

a) 5% Anta at vi fortsetter innsettingen angitt ovenfor ved å sette inn noder med verdiene 7, 8, 9, 10, 11 og 12. **Tegn opp hvordan treet vil se ut etter at verdiene 8, 10 og 12 er satt inn. Du skal altså tegne opp tre fullstendige og ulike trær.**

b) 5% I en implementasjon (se koden for "node" under) ønsker vi i hver node å ha registrert dens høyde i det nåværende treet. Når vi setter inn en ny node, slik som angitt av algoritmen ovenfor, må vi selvfølgelig gi den riktig høyde i treet. **Hvilke andre noder i treet kan få forandret sin høyde ? Forklart kort (max. 0.5 A4-side)**.

Nodene skal ha følgende struct-deklarasjon:

```
struct node  {
    int id;              //  Nodens ID/verdi.
    int hoyde;           //  Nodens høyde.
    node* left;          //  Peker til venstre subtre.
    node* right;         //  Peker til høyre subtre.
    node(int v)   {  id = v;   hoyde = 0;   left = right = NULL;  }
};
```

c) 15% **Lag funksjonen `void settinn(node* & rot, int verdi)`**

Husk at "verdi" er større enn alle andre verdier i treet. Lag en ny node med denne som verdi/ID, og sett den inn i treet gitt av roten "rot". Funksjonen skal basere seg på algoritmen gitt ovenfor, og den trenger/bør ikke å være rekursiv. Grunnen til at "rot" er referanseoverført er at du ved enkelte tilfeller også har behov for å endre roten. Det kan hende at du også får behov for å lage en eller flere andre små hjelpefunksjoner.

Oppgave 4 (koding)

I denne oppgaven skal vi fokusere på noe som kalles et ”beslutningstre”. Hver node i et slikt tre representerer en gitt situasjon og alle de valgmuligheter man har i denne situasjonen. Stien fra rota og ned til en bladnode representerer en løsning på det problemet man ønsker å finne et svar på. For å løse et gitt problem, gjelder det å finne den stien som gir den beste løsningen ift. problemet. I denne eksamensoppgaven skal vi se på det at man kun har to alternative valg i en hver situasjon (node). Dvs. vi har et binært, totalt balansert og fullt tre. To eksempler på en slik problemstilling kan være:

1. En pose diamanter skal fordeles mest mulig likt mellom to personer (A og B). Diamantene er av meget varierende størrelse/vekt. Med ”likt” menes at diamantene skal fordeles slik at A og B hver får en totalvekt av diamanter som er mest mulig lik hverandre.
2. En masse kasser (av meget ulik vekt) med varer skal fordeles på to jernbanevogner. Fordel disse slik at totalvekten i de to vognene er mest mulig lik hverandre. Kassenes volum/størrelse er likegyldig/uinteressant.

(Du kan selv sikkert komme på andre lignende eksempler, også der man har mer enn to valgmuligheter i enhver situasjon …..……)

En logisk representasjon (og evt. en fysisk representasjon i datamaskinens hukommelse) av et slikt binært beslutningstre vil være som følger: Rota representerer den første gjenstanden. Følger man veien ned til dens venstre subtre, så betyr det at gjenstanden (rota) tildeles A. Følger man veien ned til det høyre subtreet, så betyr det at B blir tildelt den. Uansett hvilken vei man følger, så vil man på nivå to treffe på to noder som kun representerer gjenstand nr.2. Hvem som således tildeles denne, avgjøres på samme måte ved å følge veien til et av deres to subtrær. Nivå tre vil deretter inneholde fire noder som kun representerer gjenstand nr.3. Slik blir det hele veien ned til det siste nivået ’N’, som kun vil inneholde noder som representere gjenstander nr. ’N’.
For å gjøre dette helt klart (?): alle gjenstandene/nodene er duplisert flere ganger i treet (unntatt rota). Hvert nivå (nr.’i’) inneholder kun identiske noder, som representerer gjenstand nr.’i’.

a) La oss tenke oss at det totalt er N gjenstander som skal deles i to ulike ”hauger”.
5%
- **Hvor mange noder vil det være på nivå nr.’N’ ?**
- **Hvor mange interne noder vil treet totalt inneholde ?**
- **Hvor mange eksterne noder (z-noder) vil treet totalt inneholde ?**
- **Hvor mange ulike løsninger vil det være på en slik fordeling ?**

b) La oss tenke oss at alle gjenstandenes verdi/vekt er lagt inn i arrayen
20%
```
int verdi[N+1];
```
(Vi bruker ikke indeks nr.0.)

Skriv et program som finner (og skriver ut) den optimale løsningen på hvordan fordele disse gjenstandene mest mulig likt/rettferdig i to ”hauger”.

Hint: Du må sikkert lage et lite hovedprogram, definere noen flere (globale) hjelpevariable og lage en eller flere (rekursive) funksjoner som ”gjør jobben”.

Lykke til !
frode@haug.com