Høgskolen i Gjøvik
Avdeling for informatikk og medieteknikk

E K S A M E N

EMNENAVN: **Algoritmiske metoder**

EMNENUMMER: **IMT2021**

EKSAMENSDATO: **18. desember 2012**

KLASSE(R): **11HB - IDATA / PUA / DRA / ISA / SPA**

TID: **14.00-19.00**

EMNEANSVARLIG: **Frode Haug**

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

TILLATTE HJELPEMIDLER: **Alle trykte og skrevne**
(kalkulator er *ikke* tillatt)

- Kontroller at alle oppgavearkene er til stede.
- Innføring med penn, eventuelt blyant som gir gjennomslag. Pass på så du ikke skriver på mer enn ett innføringsark om gangen (da 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.
- Husk kandidatnummer på alle ark.

Oppgave 1 (teori, 25%)

Denne oppgaven inneholder tre uavhengige oppgaver fra kap.8, 11, 14, 15 og 9 i læreboka.

a) Vi har koden (s.112) for distribuert telling. Teksten som skal sorteres (blanke regnes ikke med): ”A B C A A B C B C C A A” (dvs. M = 3, N = 12). **Skriv opp innholdet i arrayen `count`:**

- **etter at 2.for-løkke er ferdig utført**
- **etter at 3.for-løkke er ferdig utført**
- ***pluss* innholdet i arrayen `b` i det 4.for-løkke har gått halvveis**

b) I de følgende deloppgaver er det key'ene ”P E Y R E S O U R D E”
(i denne rekkefølge fra venstre mot, og blanke regnes ikke med) som du skal bruke.
For alle deloppgavene gjelder det at den initielle heap/tre er *tom* før første innlegging (”Insert”) utføres. **Tegn (skriv) den *resulterende* datastruktur når key'ene legges inn i:**

1) **en heap**
2) **et binært søketre**
3) **et 2-3-4 tre**
4) **et Red-Black tre**

c) Du skal utføre Quicksort på teksten ”B A T T A J U S” (blanke regnes ikke med).
Lag en figur tilsvarende fig. 9.3 side 119 i læreboka, der du for hver rekursive sortering skriver de involverte bokstavene og markerer/uthever hva som er partisjonselementet.

Oppgave 2 (teori, 25%)

Denne oppgaven inneholder tre uavhengige oppgaver fra kap.22, 30 og 32.

a) Vis **konstruksjonsprosessen** når bokas metode **for Huffman-koding** (s.324-330) brukes på teksten ”TDF ER SYKKELTOUREN SOM STORT SETT SYKLES RUNDT OMKRING I FRANKRIKE” (inkludert de blanke – husk den ifm. linjeskiftet).
Hvor mange bits trengs for å kode denne teksten? Dvs. skriv/tegn opp:

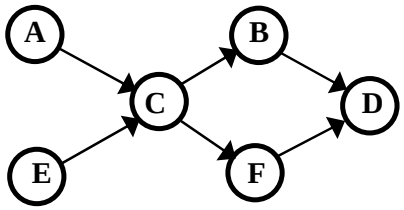
- tabellen for bokstavfrekvensen (jfr. fig.22.3).
- tabellen for ”dad”en (jfr. fig.22.6).
- Huffmans kodingstreet/-trien (jfr. fig.22.5 og koden øverst s.328).
- bokstavenes bitmønster og ”len” (jfr. fig.22.7 og koden øverst s.329).
- totalt antall bits som brukes for å kode teksten.

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

AE BE CD EC BC DA

Vi skal nå utføre *union-find m/weight balancing og path compression* 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 rettede asykliske (ikke-vektede) graf («dag») er gitt:



Angi ***alle* mulige topologiske sorteringssekvenser** av nodene i denne grafen.

Oppgave 3 (koding, 30%)

Vi har et *binært søketre* og den ferdige koden:

```
struct Node  {
  int   ID;       //  Nodens ID/key/nøkkel/navn (et tall).
  Node* left;     //  Peker til venstre subtre, evt. z når tomt.
  Node* right;    //  Peker til høyre subtre, evt. z når tomt.
  Node (int id, Node* l, Node* r)  //  Constructor:
      { ID = id; left = l; right = r; }
};
```

Globale variablene (som i læreboka peker `head->right` til treets reelle rot):

```
Node* z    = new Node(0, NULL, NULL);  // z-noden (ID = 0).
Node* head = new Node(0, NULL, z);  // Head->right peker til rota.
```

Funksjonen:

```
                                    //  Høyrerotasjon omkring 'p'
Node* hoyreRotasjon(Node* p) {      //   (jfr. kode s.225 i læreboka):
  Node* q = p->left;                //  Peker venstre barn.
  p->left = q->right;               //  Overfører subtre.
  q->right = p;                     //  Høyrerotasjon fullføres.
  return q;                         //  Returnerer ny "opprotert".
}
```

I *hele* denne oppgaven skal det *ikke* innføres flere globale variable eller struct-medlemmer enn det gitt ovenfor. **Hint:** Tegn opp et litt større tilfeldig binært søketre, så er det lettere å studere/tenke på hvordan de ulike funksjonene skal operere. Treet *kan* inneholde duplikate noder (noder med lik ID).

a) **Lag den *ikke-rekursive* funksjonen** `int nestMinst(Node* p)`
Funksjonen skal returnere `ID`en til den *nest minste* noden i (sub)treet der `p` er rota.
Vi forutsetter at treet under `p` *alltid* inneholder *minst* to noder (inkludert `p`).

b) **Lag den *ikke-rekursive* funksjonen** `bool erFullstendigHoyreskjevt(Node* p)`
Funksjonen returnerer `true/false` til om *alle* noder i (sub)treet under `p` *bare* har høyrebarn (dvs. er høyreskjevt). Det tomme tre er også definert som fullstendig høyreskjevt.

c) **Lag den *ikke-rekursive* funksjonen** `void gjorFullstendigHoyreskjevt()`
Funksjonen skal gjøre *hele* treet (`head->right` peker til rota) høyreskjevt vha. gjentagende høyrerotasjoner (til *alle* nodene *kun* har høyrebarn). Du *skal* bruke den ferdiglagde `hoyreRotasjon(…)`. Det skal *ikke* brukes ekstra stack/queue/liste/o.l. for å løse dette.

Oppgave 4 (koding, 20%)

Et *anagram* er et ord, navn eller et fast uttrykk som er blitt satt sammen ved å stokke rundt på tegnene/bokstavene i et annet ord eller uttrykk. Moa. permutere alle de opprinnelige tegnene, og så evt. sette inn/ta vekk blanke innimellom for å danne et utrykk/setning. Eksempler på slike kan være:

- rose - eros
- somlepave - av eplemos
- erling jevne - jeg vil renne

I denne oppgaven endrer/utvider vi denne definisjonen litt: ord2 er et anagram av ord1 om *alle* tegnene i ord1 er med i ord2, men at ord2 i tillegg *gjerne* må inneholde *enda* flere tegn/bokstaver.
Dessuten inneholder *ingen* av ordene (verken ord1 eller ord2) blanke tegn. Dvs. vi ser *ikke* på uttrykk/setninger, men *kun* enkeltord.
Eksempler kan være: who (ord1) – how (ord2) eller kanon (ord1) – nonnekappe (ord2)
I det siste eksemplet er altså ord2 et anagram av ord1, mens det motsatte *ikke* er tilfelle/sant.

Lag funksjonen `bool anagram(char* ord1, char* ord2)`
som svarer `true/false` til om `ord2` er et anagram av `ord1` etter den nye definisjonen ovenfor.

Begge ordene er max.80 tegn lange (inkl. '\0'), og *alle* tegnene i begge ordene er skrevet med store bokstaver. Du kan *ikke* bruke den ferdig definerte string-klassen. Men, du kan forutsette at `cstring` er includet, og du kan derfor fritt bruke funksjoner (strcpy, strlen, strcmp, …..) i denne.
Tips: *en* nyttig funksjon kan være: `char* strchr(char* t, char c)`
Denne returnerer en peker inni teksten (tilpekt av) `t` der *første forekomst* av `c` er å finne, evt. NULL om `c` *ikke* finnes.

Legg vekk på effektivitet, ved å avskjære (returnere med `false`) med en gang det oppdages at `ord2` ikke kan være et anagram av `ord1`. For det er fortsatt sant at: "Korrekt og effektiv kode, er kort og elegant" (løsningsforslaget er på ni linjer, inkludert en linje med bare en krøllparentes).

I *hele* dette oppgavesettet skal det *ikke* brukes ferdig kode fra (standard)biblioteker (STL).

Løkke tæll!

frode@haugianerne.no