

Institutt for datateknologi og informatikk

Eksamensoppgave i IDATG2102 – Algoritmiske metoder

Faglig kontakt under eksamen:

Frode Haug

Tlf:

950 55 636

Eksamensdato:

8.desember 2021

Eksamenstid (fra-til):

09:00-13:00 (4 timer)

Hjelpemiddelkode/Tillatte hjelpemidler:

F - Alle trykte og skrevne.
(kalkulator er *ikke* tillatt)

Annen informasjon:

Målform/språk:

Bokmål

Antall sider (inkl. forside):

4

Informasjon om trykking av eksamensoppgaven

Originalen er:

1-sidig ☒ 2-sidig ☐

sort/hvit ☒ farger ☐

Skal ha flervalgskjema ☐

Kontrollert av:

Dato

Sign

Oppgave 1 (teori, 25%)

Denne oppgaven inneholder tre totalt uavhengige oppgaver fra pensum.

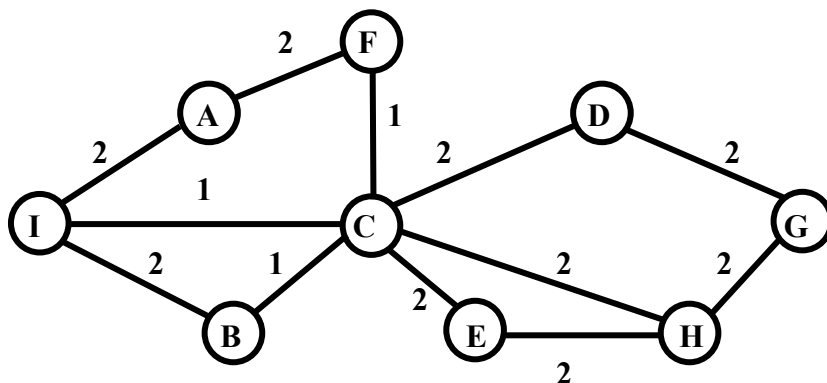
- a) 8 Et av eksemplene i pensum leser et postfix-uttrykk og regner ut svaret.
Vi har postfix-uttrykket: $5\ 2\ 4\ +\ 4\ 2\ *\ 3\ 2\ 4\ *\ +\ +\ *\ +$
Hva blir svaret? Skriv/tegn opp stakkens innhold etter hver gang den er endret.
- b) 9 Shellsort skal utføres på bokstavene «BRYLLUPENE». For hver gang indre for-løkke i eksemplet med Shellsort er ferdig (dvs. rett etter: $a[j] = \text{verdi};$) :
Skriv/tegn opp arrayen og skriv verdiene til 'h' (4 og 1) og 'i' underveis i sorteringen. Marker spesielt de key'ene som har vært involvert i sorteringen.
- c) 8 I forbindelse med dobbelt-hashing har vi teksten «BRYLLUPSPLAN» og de to hash-funksjonene $\text{hash1}(k) = k \bmod 13$ og $\text{hash2}(k) = 5 - (k \% 5)$ der k står for bokstavens nummer i alfabetet (1-29). Vi har også en array med indeksene 0 til 12.
Skriv hver enkelt bokstav sin k-verdi og returverdi fra både hash1 og hash2. Skriv også opp arrayen hver gang en bokstav hashes inn i den.

Oppgave 2 (teori, 25%)

Denne oppgaven inneholder tre totalt uavhengige oppgaver fra pensum.

- a) 5 **Skriv/tegn opp Merkle treet som er basert på 11 blokker.**

- b) 10 Følgende vektete (ikke-rettede) graf er gitt:



Vi bruker nabomatrise. Aktuell kode i et at pensumets eksempler utføres/kjøres på denne grafen.

Hvilke kanter er involvert i korteste-sti spenntreet fra noden A til alle de andre nodene?

Skriv også opp innholdet i/på fringen etterhvert som koden utføres.

NB: Husk at ved lik vekt så vil noden sist oppdatert (nyinnlagt eller endret) havne først på fringen ift. andre med den samme vekten.

- c) 10 Følgende kanter i en (ikke-rettet, ikke-vektet) graf er gitt:

DA FE CA FB GA FC BE

Utfør Union-Find *m/weight balancing (WB)* og *path compression (PC)* på denne grafen.

Skriv/tegn opp innholdet i gForeldre etter hvert som unionerOgFinn2

kjøres/utføres. Bemerk hvor WB og PC er brukt.

Skriv/tegn også opp den resulterende union-find skogen.

Oppgave 3 (koding, 32%)

Vi har et binært tre bestående av:

```
struct Node {
    int ID; // Nodens ID/key/nøkkel/navn (et tall).
    Node *left, *right; // Referanse til begge subtrærne (evt. nullptr/NULL).
    Node(int id) { ID = id; left = right = nullptr; }
};
```

Vi har de to globale variablene:

```
Node* gRoot = nullptr; // Rot-peker (har altså ikke at head->right er rota).
const int MAX = 999; // Max.nodehøyde (høyere enn reelt. Brukes kun i 3a).
```

Det skal her lages/kodes to helt uavhengige funksjoner.

Begge funksjonene kalles initielt fra main med bl.a. gRoot som parameter.

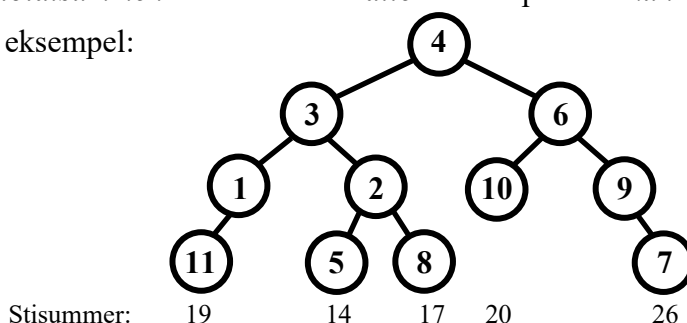
Hvordan et tre har blitt bygd/satt opp, trenger du ikke å tenke på.

NB: I *hele* oppgave 3 skal det *ikke* innføres flere globale data eller struct-medlemmer enn angitt ovenfor. Det skal heller *ikke* brukes andre hjelpestrukturer - som f.eks. array, stakk, kø eller liste.

a) Lag den rekursive funksjonen `int minimumHoyde(const Node* t)`
Funksjonen skal finne *minimumshøyden* i (sub)treet tilpekt av *t*. Altså *nivået for bladnoden på det laveste/minste nivået under t*. Rota er på nivå nr.1, dens barn er på nivå 2, rotas barnebarn er på nivå 3, osv. Et tomt tre (`gRoot == nullptr`) har minimumshøyde lik 0 (null). Funksjonen skal altså rekursivt finne og returnere minimumshøyden for node *t* ift. det laveste subtreet. Bladnoder har høyde lik 1 (ift. sine ikke-eksisterende barn). Er *t* lik `nullptr`, er dets høyde uklart/undefinert. Til dette bruker (og returnerer) vi `MAX`. Husk å spesialbehandle at treet evt. er helt tomt. **NB:** Husk at det *ikke* skal innføres flere globale variable.

b) Lag den rekursive funksjonen `int slettNoderPaaStiMedMinSum(Node* & t, const int sum, const int min)`
Funksjonen skal rekursivt slette (delete) *alle* noder som er på en sti (fra rot til bladnode), der *totalsummen* av ID'ene for *alle* nodene på stien *kun* er *mindre enn* *min*.

For eksempel:



Kallet `slettNoderPaaStiMedMinSum(gRoot, 0, 18)` medfører at nodene 2, 5 og 8 slettes, da disse ligger på stier med *totalsumme* mindre enn 18. Men nodene 3 og 4 slettes *ikke*, da de også ligger på en sti med totalsum høyere eller lik enn 18. Hadde *min*-parameteren i stedet vært 24, ville *alt* på venstre side av noden 4 ha blitt slettet, samt noden 10.

NB1: Funksjonen *skal* returnere *totalsummen på lengste sti* som går gjennom *t*.

NB2: *sum* er totalsummen *hittil* på stien der *t* ligger.

NB3: *t* er referanseoverført! Dermed kan det den peker til evt. slettes (delete), og at den (dvs. «mor sin» *left* eller *right*) kan evt. bli satt til `nullptr/NULL`.

NB4: I Java er det ingen slik måte å direkte oppdatere selve original-parameter-referansen. Men skriver du kode i Java, så *later vi som* at dette er mulig ved å bruke `Node & t`. Dermed er det *ikke* en *lokal kopi-referanse* inni funksjonen som oppdateres, men selve den *originale medsendte referansen* som kan settes til NULL (og urefererte noder slettes jo automatisk i Java).

Oppgave 4 (koding, 18%)

Lag den ikke-rekursive funksjonen

```
void flettToSorterteArrayer(int a[],
                             const int b[],
                             const int aLen, const int bLen)
```

Funksjonen skal *flette sammen* den *sorterte* arrayen `a` med den *sorterte* arrayen `b` inni igjen i en fortsatt sortert array `a`, *uten å bruke ekstra memory/hjelpearray*. Begge arrayene inneholder altså heltall, og disse *kan også være negative*. Verdien 0 (null) i `a` betyr at den aktuelle «skuffen» er ledig. Array `a` er så lang at det er *eksakt* plass til elementene fra `b` i den. Dvs. det er like mange nuller i `a` som antall elementer i `b`. `aLen` og `bLen` er antall elementer i de to arrayene. Disse ligger f.o.m. indeks nr.0 (null) t.o.m indeks nr. `aLen-1/bLen-1`.

For eksempel:

array a:	-8 -4 0 0 -1 3 0 7 0 9 11 0 0 15 19	(15 elementer, derav 6 nuller)
array b:	-10 -3 -1 4 9 24	(6 elementer)
a etterpå:	-10 -8 -4 -3 -1 -1 3 4 7 9 9 11 15 19 24	(15 samsorterte elementer)

NB: I *hele* dette oppgavesettet skal du *ikke* bruke kode fra (standard-)biblioteker (slik som bl.a. STL og Java-biblioteket). Men de vanligste `include/import` du brukte i 1.klasse er tilgjengelig. Koden kan skrives valgfritt i C++ eller Java.

Løkke tæll!
FrodeH