Det matematisk-naturvitenskapelige fakultet
UNIVERSITETET I BERGEN
Eksamen i emnet I 120 - Algoritmer, datastrukturer og programmering
Onsdag 13.Desember 2000, kl. 09-15.

Ingen hjelpemidler tillatt. Oppgavesettet består av 5 oppgaver. I vedlegget er du gitt dokumentasjon av utvalgte ADTer og klasser i JDSL, som du fritt kan bruke i dine besvarelser. Skriv ned alle forutsetninger du gjør som ikke står i oppgaveteksten. Lykke til!

Oppgave 1. Stabler og Køer (30%)

Oppgaven dreier seg om ADT Stack og Queue (JDSL-dokumentasjon av disse gitt i vedlegg). Vi skal programmere metode(r) for å sjekke om en stabel og en kø er like. En stabel S er *lik* en kø Q hvis enten:

- både `S.isEmpty()==true` og `Q.isEmpty()==true`, eller
- både `S.isEmpty()==false` og `Q.isEmpty()==false` samtidig som `S.top()==Q.front()` og stabelen som resulterer etter `S.pop()` er *lik* køen som resulterer etter `Q.dequeue()`.

Vi antar at vi har en implementasjon av ADT Stack, dvs en `class Stabel implemements Stack`. I hver av deloppgavene under skal du lage en ny klasse som arver fra `class Stabel` og har en ny metode `public boolean eqQueue(Queue Q)` som returnerer true hvis og bare hvis stabelen metoden kalles fra er *lik* argumentet Q. Du skal bruke kall til metodene fra `interface Stack` og `interface Queue`. I hver av deloppgavene er kravene til implementeringen av `eqQueue` forskjellige, og i hvert tilfelle skal du gi **fullstendig Java-kode**, med bruk av JDSL. La $|Q|$ være antall elementer i argumentet Q.

1.1 Du skal ikke bruke noen ekstra datastrukturer, og utføring av `eqQueue(Q)` kan (og vil nødvendigvis) endre innholdet i både stabelen og køen.

1.2 Du skal nå implementere `eqQueue` slik at når et kall til denne metoden returnerer er både stabelen og køen gjenoppbygget, dvs de er uendret i forhold til før kallet. Du skal kun bruke en ekstra kø og en ekstra stabel til dette, ingen andre datastrukturer, bortsett fra eventuelle løkke-variable av type `int`. (Hint: Bruke while-løkker.) Antall `dequeue`-operasjoner pluss antall `pop`-operasjoner totalt for `eqQueue(Q)` skal være:

- Mindre enn $3|Q|$ dersom eqQueue returnerer false.
- Mindre eller lik $3|Q|$ dersom eqQueue returnerer true.

1.3 Du skal nå implementere `eqQueue` slik at når et kall til denne metoden returnerer er stabelen gjenoppbygget, men køen er **ikke** nødvendigvis gjenoppbygget. Du skal ikke bruke noen ekstra datastrukturer, bortsett fra en objekt-variabel som kan inneholde et enkelt element fra stabelen. (Hint: Bruk rekursjon.)

Oppgave 2. Binære søketrær og heap (15%)

Gitt nøkkelverdiene $S = \{2, 3, 4, 8, 5, 6, 12, 9\}$.

2.1 Tegn 2 forskjellige heap'er som begge representerer S.

2.2 Tegn 2 forskjellige binære søketrær som begge representerer S.

2.3 For hver av de 4 datastrukturene du har tegnet over, tegn datastrukturen som er resultatet av å sette inn et nytt element med nøkkelverdi 7. Bruk algoritmene for innsetting som er gitt i læreboka.

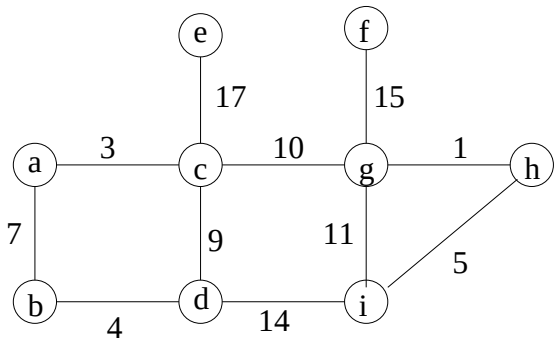
2.4 Skisser (gi pseudokode) en metode `void printSorted(node v)`, slik at kallet `printSorted(root)` for et binært søketre B med rotnode `root` vil skrive ut alle nøkkelverdiene i B i sortert rekkefølge. Treet i deloppgave 2.2 over skal gi utskriften: 2 3 4 5 6 8 9 12. Du kan bruke metodene

- `node left(node v)`: returnerer venstre barn av v dersom denne eksisterer, ellers returnerer den `null`.
- `node right(node v)`: returnerer høyre barn av v dersom denne eksisterer, ellers returnerer den `null`.
- `boolean leaf(node v)`: returnerer true dersom v er en løvnode, dvs v har overhode ingen barn, false ellers.
- `void print(node v)`: printer ut nøkkelverdien inneholdt i node v.

Oppgave 3. Grafalgoritmer (20%)

Gitt grafen G i figuren under. I deloppgavene under skal du simulere forskjellige grafalgoritmer på grafen G, alltid med utgangspunkt i node a. Vi sier at en node *oppdages* idet en algoritme første gang treffer på noden. Dersom det finnes et valg, så anvend alfabetisk ordning, f.eks. betyr dette at naboene til en node hentes fram i alfabetisk rekkefølge f.eks. ved følgende nabolister:

-a: b, c	-d: b, c, i	-g: c, f, h, i
-b: a, d	-e: c	-h: g, i
-c: a, d, e, g	-f: g	-i: d, g, h



Vær nøye med å overholde den alfabetiske ordningen, og dobbeltsjekk dine svar, da et helt korrekt svar vil gi vesentlig flere poeng.

3.1 I hvilken rekkefølge vil DFS(a), dvs dybde-først-søk fra a, oppdage nodene i G?

3.2 I hvilken rekkefølge vil BFS(a), dvs bredde-først-søk fra a, oppdage nodene i G?

3.3 Vi sier at node x *forlates* idet det rekursive kallet DFS(x) terminerer. I hvilken rekkefølge vil DFS(a) forlate nodene i G?

3.4 Du skal nå simulere Dijkstra(a), dvs Dijkstra's algoritme for korteste stier med startnode a, på grafen G, og svare på to spørsmål. I hvilken rekkefølge vil Dijkstra(a) utføre relaxation-operasjon på kanter i G? Angi kantene ved deres endepunkter, og angi også hvorvidt relaxation-operasjonen førte til en endring i distansetabellen, f.eks. er ab den første kanten for hvilken relaxation utføres, og denne fører til en endring. Hva er de suksessive verdiene som blir tildelt hver node i distansetabellen? Angi disse verdiene for hver enkelt node, f.eks. for node b er svaret: $\infty, 7$.

3.5 Du skal nå simulere Prim(a), dvs Prim-Jarnik's algoritme for minimum utspennende tre på grafen G hvor vi antar node a blir vilkårlig valgt i første steg, og svare på to spørsmål. Hva blir det resulterende minste utspennende treet? Tegn dette treet. I hvilken rekkefølge tildelte Prim(a) kantene til dette treet? Sett f.eks. et nummer på hver kant i treet som reflekterer når de ble lagt til treet.

Oppgave 4. Kompleksitetsanalyse (15%)

For hver av metodene `f1, f2, f3, f4`, bestem verste-fall kjøretid i stor-O (big-Oh) notasjon for kallene `f1(a,N), f2(a,N), f3(a,b,c,N), f4(a,N)` som en funksjon av argumentet N. For enkelhets skyld antar vi N er et positivt heltall som er en potens av 2. Den øvre grensen du gir skal være så tett som mulig, og du skal droppe uvesentlige konstanter, f.eks. for lineær kjøretid er O(N) et bedre svar enn O(2N+logN). **Gi en begrunnelse for hvert svar.** (Hint: Det er ikke nødvendig å forstå hva metodene gjør, men det *kan* likevel gi en pekepinn dersom du sitter fast.)

```
void f1(int[] a, int n) {
   int i,j;
   for (j = 0; j < n; j++) {
      i = n;
      while (i > 1) {
         a[j] = a[j]/i;
         i = i/2;
      }
   }
}
```

```
void f2(int[] a, int n) {
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < i; j++) {
         a[i] = a[i]+j;
      }
   }
}
```

```
void f3(int[] a, int[] b, int[] c, int n) {
   int i=0, j=0, k=0;
   while (k < 2*n) {
      if (i<n && (j>n || a[i]<b[j]))
         c[k++]=a[i++];
      else
         c[k++]=b[j++];
   }
}

void f4(int[] a, int n) {
   int[] b = new int[n/2];
   int[] c = new int[n/2];
   if (n < 2) return;
   for (i = 0; i < n/2; i++)
      b[i]=a[i];
   for (i = n/2; i < n; i++)
      c[i]=a[i];
   f4(b, n/2);
   f4(c, n/2);
   f3(b, c, a, n/2);
}
```

Oppgave 5. Programmering (20%)

Du skal gi **fullstendig Java-kode**, med bruk av JDSL, for en implementasjon av interface GeneriskSort:

```
public interface GeneriskSort {
  public void pQSort(RankedSequence S, Comparator C);
     // S inneholder objekter som kan sammenliknes vha C.
     // Objektene i S sorteres i stigende rekkefølge vha
     // prioritetskøsortering, og ender opp sortert i S.
  public void enkelSort(RankedSequence S, Comparator C);
     // S inneholder objekter som kan sammenliknes vha C.
     // Objektene i S sorteres i stigende rekkefølge, og
     // ender opp sortert i S. Det brukes ikke en prioritetskø.
  }
```

I vedlegget finner du dokumentasjon av JDSL-interface Comparator, RankedSequence og SimplePriorityQueue, samt dokumentasjon for JDSL-klasser som implementerer RankedSequence og SimplePriorityQueue. Disse klassene kan du fritt bruke uten å gi implementasjon.

Jan Arne Telle Chunming Rong

Løsningsforslag Eksamen I120-H2000

NB: Kun et forslag, mange andre varianter er korrekte.

Oppgave 1. Stabler og Køer (30%)

1.1

```
public class myStabel extends Stabel {
  public boolean eqQueue (Queue Q) {
    if (size() != Q.size()) return false;
    while (!isEmpty()) {
      if (top() != Q.front()) return false;
      pop();
      Q.dequeue();
    }
    return true;
  }
}
```

1.2

```
public class myStabel extends Stabel {
  public boolean eqQueue (Queue Q) {
    if (size() != Q.size()) return false;
    Stack tempStack = new Stabel();
    while (!isEmpty() && top() == Q.front()) {
      tempStack.push(pop());
      Q.enqueue(Q.dequeue());
    }
    if (isEmpty()) {
      while (!tempStack.isEmpty()) push(tempStack.pop());
      return true;
    }
    else {
      for (int i=0; i<size(); i++) Q.enqueue(Q.dequeue());
      while (!tempStack.isEmpty()) push(tempStack.pop());
      return false;
    }
  }

}
```

1.3

```
public class myStabel extends Stabel {
  public boolean eqQueue (Queue Q) {
    if (size() != Q.size()) return false;
    return eqQ(Q);
  }
  private boolean eqQ (Queue Q) {
    if (isEmpty()) return true;
    if (top() != Q.front()) return false;
    Object temp = pop();
    Q.dequeue();
    boolean ret = eqQ(Q);
    push(temp);
    return ret;
  }
}
```

Oppgave 2. Binære søketrær og heap (15%)

2.1 Det vesentlige er at i) hver heap er et komplett binært tre,dvs har ett element på nivå 4 som ligger lengst til venstre, samt at ii) nøkkel for enhver node er mindre enn nøkkel hos eventuelle barn.

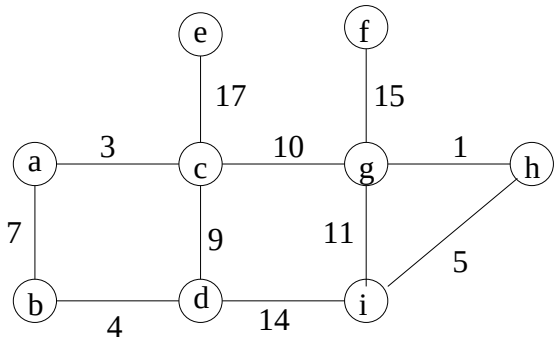
2.2 Det vesentlige er at i) hvert søketre er et binært tre (0, 1 eller 2 barn) og ii) at for enhver node med nøkkel x er alle nøkler i dens venstre deltre mindre enn x og i dens høyre deltre større enn x.

2.3 For heap brukes UpHeap-prosedyren, dvs at den nye nøkkelen starter i neste ledige posisjon, og 'bobler opp' til rett plass.
For binære søketrær skal den nye nøkkelverdien legges til i en ny løvnode på rett plass, og ingen andre endringer skjer i treet.

2.4

```
Algorithm printSorted(node v)
   if left(v) != null printSorted(left(v))
   print(v)
   if right(v) != null printSorted(right(v))
```

Oppgave 3. Grafalgoritmer (20%)



3.1 a, b, d, c, e, g, f, h, i

3.2 a, b, c, d, e, g, i, f, h

3.3 e, f, i, h, g, c, d, b, a

3.4 ab, ac, cd, ce, cg, bd, di, gf, gh, gi, hi (alle fører til endring)

$a : 0$
$b : \infty, 7$
$c : \infty, 3$
$d : \infty, 12, 11$
$e : \infty, 20$
$f : \infty, 28$
$g : \infty, 13$
$h : \infty, 14$
$i : \infty, 25, 24, 19$

3.5 Treet får kantene (i rekkefølge): ac, ab, bd, cg, gh, hi, gf, ce.

Oppgave 4. Kompleksitetsanalyse (15%)

f1 er $O(NlogN)$. En ytre for-løkke som utfører while-løkken N ganger. For hver av disse utføres operasjonene inne i while-løkken x ganger, hvor x er antall ganger vi kan dele N på 2 før vi når ned til 1. Per definisjon er $x = log_2 N$.

f2 er $O(N^2)$. Tildelingen i den indre for-løkken utføres 1 gang når i=1, 2 ganger når i=2, osv inntil n-1 ganger når i=n-1. Kjøretiden totalt blir dermed $O(\Sigma_{i=1}^{i=N-1} i) = O(N^2)$.

f3 er $O(N)$. k starter på 0, øker med 1 hver gang gjennom while-løkken, og vi stanser når $k = 2 * N - 1$.

f4 er $O(NlogN)$. Dette er Flettesorteringsalgoritmen (borsett fra en liten bug: c[i]=a[i] skulle være c[i-n/2]=a[i]). Analysen kan gjøres ved å se at kallet f4(N) leder til to kall f4(N/2) samt O(N) tid for å flytte elementer mellom tabeller og kjøre f3. Vi kan sette opp et (binært) tre av rekursive kall, som vil ha dybde logN, og se at det i hvert nivå vil utføres O(N) arbeid.

Oppgave 5. Programmering (20%)

```
public class GenSort implements GeneriskSort {
  public void pQSort(RankedSequence S, Comparator C) {
    SimplePriorityQueue pq = new HeapSimplePriorityQueue(C);
    while (!S.isEmpty())
      pq.insertItem(S.removeElemAtRank(0), null);
    while (!pq.isEmpty()) {
      S.insert(pq.minKey());
      pq.removeMinElement();
    }
  }
  public void enkelSort(RankedSequence S, Comparator C) {
     for (int j=0; j<S.size(); j++) {
        Object min=S.elemAtRank(j);
        int minRank=j;
        for (int i=j+1; i<S.size(); i++) {
          Object next=S.elemAtRank(i);
          if C.isLessThan(next, min) {
            min=next;
            minrank=i;
          }
        S.replaceElemAtRank(minrank, S.elemAtRank(j));
        S.replaceElemAtRank(j, min);
        }
     }
  }
}
```