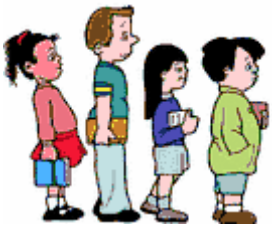
Algoritmer og datastrukturer
Kapittel 4 – Delkapittel 4.2

4.2 En kø

4.2.1 Grensesnittet Kø



Studenter i kø

Begrepet kø (engelsk: queue) bruker vi i mange sammenhenger. Vi snakker om en busskø, en drosjekø, en kø på postkontoret, en polkø, en kø foran kassen i butikken, osv. På en del steder (f.eks. mange postkontorer) brukes kølapper og da «står» vi ikke lenger i kø. Men prinsippet er fortsatt det samme. Et køsystem kjennetegnes ved at den som kom først skal betjenes først. Eller sagt på en annen måte: Den som har ventet lengst er den som står for tur. Med kølappsystem er det den med lavest nummer som ropes opp først. Men den som har lavest nummer er den som trakk nummer (kom) først.

> **Definisjon** *En kø er en datastruktur der verdier legges inn bakerst og tas ut forrest. Med andre ord er den verdien som ble lagt inn først (den som har ventet lengst) som tas ut først.*

Et annet navn som også brukes på dette begrepet, er en «Først-inn-først-ut»-kø. På engelsk heter det «First-In-First-Out»-queue eller FIFO-queue.

En kø trenger metoder av samme type som i en Stakk. I en kø må imidlertid *leggInn*() legge inn bakerst i køen, *kikk*() må returnere (uten å fjerne) den som er først og *taUt*() må ta ut (fjerne) den som er først i køen. Disse kravene setter vi opp i flg. grensesnitt:

```
public interface Kø<T>                // eng: Queue
{
  public boolean leggInn(T verdi); // eng: offer/add/enqueue    inn bakerst
  public T kikk();                 // eng: peek/element/front   den første
  public T taUt();                 // eng: poll/remove/dequeue  tar ut den første
  public int antall();             // eng: size                 køens antall
  public boolean tom();            // eng: isEmpty              er køen tom?
  public void nullstill();         // eng: clear                tømmer køen

} // interface Kø
```

Programkode 4.2.1 a)

I en stakk brukes normalt (på engelsk) navnene *push*, *peek* og *pop*. Det er mange som også bruker disse navnene i en kø. Men der er nok navn som *enqueue*, *front* og *dequeue* mer vanlig. I grensesnittet Queue i `java.util` er det imidlertid valgt andre navn. Der er det til og med to metoder for hver operasjonstype. Både *add* og *offer* legger en verdi forrest i køen. Forskjellen er at den første (*add*) kaster et unntak, mens den andre (*offer*) returnerer *false* hvis innleggingen ikke kan gjennomføres. Metodene *element* og *peek* «ser på» den første i køen. Den første (*element*) kaster et unntak, mens den andre (*peek*) returnerer *null* hvis køen er tom. Tilsvarende er det for metodene *remove* og *poll*. Det betyr at hvis vi f.eks bruker *peek* og køen har en nullverdi, så blir det tvetydig. Mer om dette i *Avsnitt* 4.2.5.

Oppgaver til Avsnitt 4.2.1

1. Wikipedia har en artikkel om *Queue*. Les hva som står der.
2. Det er enkelt å snu innholdet av en kø hvis en kan bruke en stakk som hjelpemiddel. Da er det bare å legge alle verdiene fra køen over på stakken og så ta dem fra stakken og legge dem tilbake i køen. Dvs. slik:

   ```
   while (!kø.tom()) stakk.leggInn(kø.taUt());     // fra kø til stakk
   while (!stakk.tom()) kø.leggInn(stakk.taUt());  // fra stakk til kø
   ```

   Men er det mulig å snu innholdet i en kø:

   a) Ved å bruke to hjelpekøer? Se også Oppgave 11 i *Avsnitt* 4.2.2.

   b) Ved å bruke én hjelpekø og noen hjelpevariabler? Se også Oppgave 12 i *Avsnitt* 4.2.2.

   c) Ved å bruke kun hjelpevariabler? Se også Oppgave 13 i *Avsnitt* 4.2.2.
3. Hvis en kø inneholder verdier som kan sammenlignes og ordnes i rekkefølge, bør det i prinsippet være mulig å finne minste verdi og å sortere køen. Med andre ord bør det være mulig å lage flg. generiske metoder:

   ```
   public static <T> T min(Kø<T> kø, Comparator<? super T> c)
   public static <T> T fjernMin(Kø<T> kø, Comparator<? super T> c)
   public static <T> int sorter(Kø<T> kø, Comparator<? super T> c)
   ```

   a) Har du forslag til hvordan metoden *min()* kunne lages? Den skal returnere den minste verdien (bestemt av komparatoren). Køen skal etterpå være som den var.

   b) Har du forslag til hvordan metoden *fjernMin()* kunne lages? Den skal returnere (og fjerne) den minste verdien (bestemt av komparatoren). Resten av køen skal etterpå ha samme rekkefølge som før.

   c) Kan en kø sorteres ved kun å bruke en hjelpekø (med eventuelle hjelpevariabler)?

   d) Kan en kø sorteres uten bruk av hjelpestrukturer (men med noen hjelpevariabler)?
4. I en kø kan vi si at verdiene har en rekkefølge eller indeksering (slik som i en liste). Den første i køen har indeks lik 0, den neste i køen indeks lik 1, osv. Hvis det er *n* verdier i køen, er indeks til den siste lik *n* - 1.

   a) Lag `public static <T> T hent(Kø<T> kø, int indeks)`. Den skal returnere verdien på plass *indeks* i køen *kø*. Etterpå skal alle verdiene ha samme indeks som de hadde før. Det skal løses kun ved hjelp av metodene i Kø (og eventuelle hjelpevariabler) og uten hjelpestrukturer. Metoden kan testes vha. klassen TabellKø. Se *Avsnitt* 4.2.2.

   b) Lag `public static <T> boolean leggInn(Kø<T> kø, int indeks, T verdi)`. Den skal legge *verdi* i køen slik at den får plass *indeks*. Alle de opprinnelige verdiene i køen skal etterpå ha samme innbyrdes rekkefølge som før. Det skal løses kun ved hjelp av metodene i Kø (og eventuelle hjelpevariabler) og uten hjelpestrukturer. Metoden kan testes vha. klassen TabellKø. Se *Avsnitt* 4.2.2.

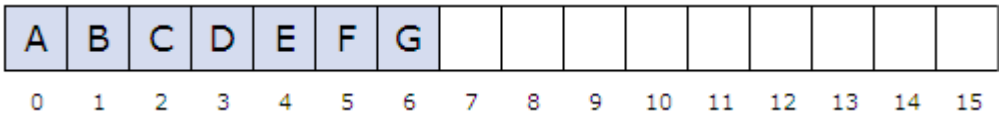
   c) Lag `public static <T> void bytt(Kø<T> kø, int indeks1, int indeks2)`. Den skal bytte verdiene på de to plassene *indeks*1 og *indeks*2. Det skal løses kun ved hjelp av metodene i Kø (og eventuelle hjelpevariabler) og uten hjelpestrukturer. Anta at køen inneholder: Per, Kari, Ole, Åse og Elin, dvs. Per først i køen og Elin sist i køen. Da skal metodekallet `bytt(kø, 0, 4)` føre til at køen blir lik: Elin, Kari, Ole, Åse og Per. Dvs. den første i køen (indeks 0) har byttet plass med den siste i køen (indeks 4). De øvrige står på samme plass som før. Metoden kan testes vha. klassen TabellKø. Se *Avsnitt* 4.2.2.

4.2.2 En sirkulær kø

En vanlig kø (f.eks. køen foran kassen i en butikk) oppfatter vel de fleste av oss som en rekke der man stiller seg opp bakerst og der man betjenes (eller går ut av køen) forrest. Denne «rekken» kan vi godt tenke på som en tabell der posisjon 0 som vanlig er forrest. Det å stille seg opp i køen blir da det samme som å legge inn i tabellen på første ledige plass:

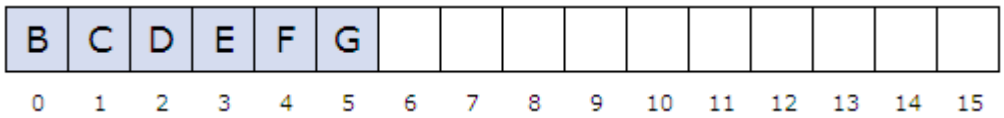
Figur 4.2.2 a) : En tom tabell representerer en tom kø

La f.eks. *A*, *B*, *C*, *D*, *E*, *F* og *G* fortløpende stille seg opp i køen. Det gir flg. tabell:



Figur 4.2.2 b) : A, B, C, D, E, F og G har stilt seg opp i køen

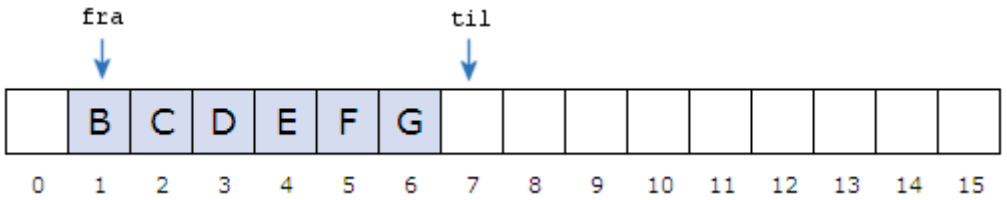
Når den første i køen betjenes (og dermed forlater køen) er det vanlig «køkultur» at de andre beveger seg et skritt eller så fremover. I tabellen betyr det at når den første (her er det *A*) tas ut av køen, blir posisjon 0 ledig. Dermed må de andre fortløpende flytte seg én posisjon mot venstre. Det vil gi følgende resultat:



Figur 4.2.2 c) : Nå står B, C, D, E, F og G i køen

Det er enkelt å lage kode for det som er beskrevet over. Men blir det effektivt nok? Hvis vi kjenner posisjonen til første ledige plass, kan innleggingen skje direkte - dvs. konstant orden. Det å «kikke» på den første innebærer å hente den som ligger i posisjon 0 - også av konstant orden. Men det å ta ut fra køen er «problematisk». Hvis det er n stykker i køen, krever et uttak at de $n - 1$ andre må flyttes én og én mot venstre for å «tette igjen» hullet som oppstod. Det betyr at uttaksoperasjonen blir av *lineær* orden eller av orden n.

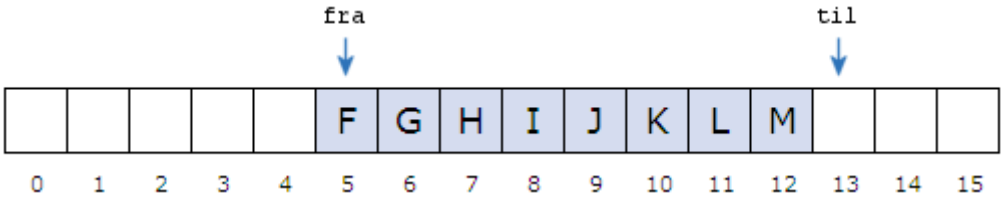
Vi prøver isteden følgende «unaturlige» idé: Innlegging skal være som før, men etter et uttak skal køen forbli i ro. Altså ingen forflytning mot venstre. Hvis vi på nytt tar utgangspunkt i køen i *Figur 4.2.2 b)*, får vi dette resultatet når den første tas ut av køen:



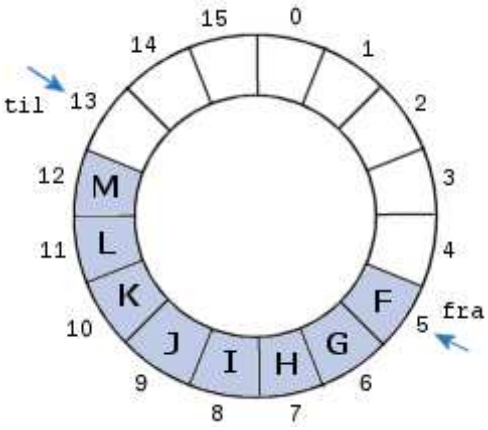
Figur 4.2.2 d) : To piler markerer starten og slutten på køen

Hvis tabellen heter a, utgjør køen nå det halvåpne intervallet $a[fra : til>$. Dette betyr at fra er posisjonen til den første i køen og at til er posisjonen til den første ledige plassen i tabellen, dvs. én posisjon forbi den siste i køen. Videre ser vi at antallet i køen er lik differensen mellom til og fra, dvs. $til - fra$. I *Figur* 4.2.2 *d)* blir det $7 - 1 = 6$.

La oss nå bruke denne idéen videre: Legg først fortløpende inn *H*, *I*, *J*, *K*, *L* og *M*, og ta deretter ut fortløpende *B*, *C*, *D* og *E*. Det vil gi flg. resultat:



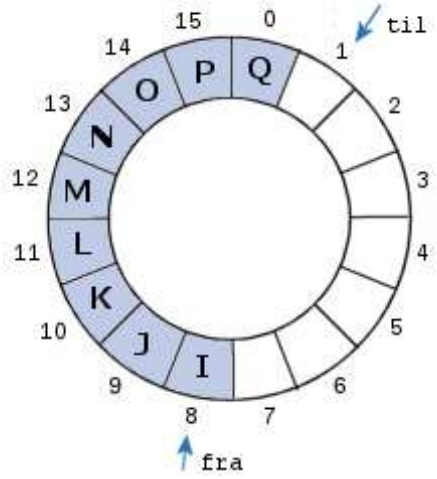
Figur 4.2.2 e) : To piler markerer starten og slutten på køen



Figur 4.2.2 f) : En sirkulær tabell

Det er nå tre ledige plasser lengst til høyre i tabellen over. Det betyr at vi kan legge inn f.eks. *N*, *O* og *P* uten problemer. Men hva så? Vi kan «utvide» tabellen slik vi har gjort før når det ikke er plass i en tabell. Men det er jo dumt når det er ledige plasser lengst til venstre. Vi kan isteden tenke oss at tabellen er «sirkelformet». Hold fast i venstre ende (posisjon 0) og bøy høyre ende nedover på en slik måte at de to endene møtes. Se *Figur* 4.2.2 *f)* til venstre.

Når tabellen er «sirkelformet», får vi et område på 8 fortløpende ledige plasser. Vi legger inn som før i posisjon *til* og *til* økes så med 1. Men hvis den blir 16 (lik *a.length*), må den isteden settes til 0.



Figur 4.2.2 g) : En sirkulær tabell

Vi kan nå legge inn *N*, *O*, *P* og *Q* fortløpende i den «sirkulære» tabellen i *Figur* 4.2.2 *f)*. Deretter tas *F*, *G* og *H* fortløpende ut. Det gir tabellen i *Figur* 4.2.2 *g)*. Legg merke til at vi tar ut fra posisjon *fra* og så økes den med 1. Også her må vi passe på at hvis *fra* blir 16 (eller *a.length*), må den isteden settes til 0.

Normalt har vi en antall-variabel med verdi til enhver tid er lik antallet i køen. Det trengs egentlig ikke her. Antallet kan regnes ut ved hjelp av *fra* og *til*. Hvis *fra* < *til* (slik som i *Figur* 4.2.2 *f*), blir antall lik *til* − *fra* (= 13 − 5 = 8). Men dette vil ikke stemme hvis *fra* > *til* slik som i *Figur* 4.2.2 *g)*. Da blir *til* − *fra* negativ. Men da blir istedet antall lik *a.length* + *til* − *fra* (= 16 + 1 − 8 = 9).

Vi får et spesialtilfelle hvis *fra* og *til* er like. Køen ligger i *a*[*fra* : *til*> og det er tomt hvis *fra* er lik *til*. Men det blir annerledes i vår «sirkulære» tabell. Der kan *a*[*fra* : *til*> utgjøre hele tabellen. Se hva som skjer hvis *R*, *S*, *T*, *U*, *V*, *W* og *X* (7 stykker) legges inn i *Figur* 4.2.2 *g)*. Da ender *til* opp med å bli lik 8. Dette problemet kan vi løse ved å sørge for at tabellen aldri er full. Dvs. hvis en innlegging fører til at den blir full, «utvider» vi den med en gang. Dermed vil *fra* og *til* være like kun når køen er tom. Antallet kan da også alltid regnes ut.

```
public int antall()
{
  return fra <= til ? til - fra : a.length + til - fra;
}
```

Programkode 4.2.2 a)

Følgende punkter oppsummerer diskusjonen over:

- Køen skal ligge i en «sirkulær» tabell *a*.
- En variabel *fra* skal markere den første i køen (starten på køen). Den settes til 0 hvis en økning har gjort at den har blitt lik *a.length*.
- En variabel *til* skal markere første ledige, dvs. én posisjon forbi den siste i køen. Den settes til 0 hvis en økning har gjort at den har blitt lik *a.length*.
- Tabellen må utvides med en gang hvis den har blitt full etter en innlegging.
- Hvis punktene foran er oppfylt, vil køen være tom hvis og bare hvis *fra* og *til* er like.

Vi kaller klassen `TabellKø` siden det er en kø implementert ved hjelp av en tabell:

```
public class TabellKø<T> implements Kø<T>
{
  private T[] a;      // en tabell
  private int fra;    // posisjonen til den første i køen
  private int til;    // posisjonen til første ledige plass

  @SuppressWarnings("unchecked")      // pga. konverteringen: Object[] -> T[]
  public TabellKø(int lengde)
  {
    if (lengde < 1)
      throw new IllegalArgumentException("Må ha positiv lengde!");

    a = (T[])new Object[lengde];

    fra = til = 0;    // a[fra:til> er tom
  }

  public TabellKø()   // standardkonstruktør
  {
    this(8);
  }

  // Her skal resten av metodene settes inn

} // class TabellKø
```

Programkode* 4.2.2 *b)

Metoden *leggInn*() skal legge verdien på første ledige plass, dvs. posisjon *til*, øke *til* med 1 og eventuelt sette den til 0 hvis den har blitt lik *a.length*. Videre skal tabellen utvides hvis den har blitt full etter innleggingen:

```
public boolean leggInn(T verdi)   // null-verdier skal være tillatt
{
  a[til] = verdi;                                 // ny verdi bakerst
  til++;                                          // øker til med 1
  if (til == a.length) til = 0;                   // hopper til 0
  if (fra == til) a = utvidTabell(2*a.length);    // sjekker og dobler
  return true;                                    // vellykket innlegging
}
```

Programkode* 4.2.2 *c)

I *Programkode* 4.2.2 *c)* blir tabellen utvidet med en gang hvis en innlegging gjør at den blir full. Her kan vi ikke bruke noen av utvidelsesmetodene fra klassen `Arrays` siden det er

nødvendig at kopieringen av den gamle tabellen over i den nye gir riktige verdier på variablene *fra* og *til*. Vi lager derfor en egen (privat) utvidelsesmetode for vår klasse:

```
private T[] utvidTabell(int lengde)
{
  @SuppressWarnings("unchecked")      // pga. konverteringen: Object[] -> T[]
  T[] b = (T[])new Object[lengde];  // ny tabell

  // kopierer intervallet a[fra:a.length> over i b
  System.arraycopy(a,fra,b,0,a.length - fra);

  // kopierer intervallet a[0:fra> over i b
  System.arraycopy(a,0,b,a.length - fra, fra);

  fra = 0; til = a.length;
  return b;
}
```

Programkode 4.2.2 d)

Metoden *taUt*() er nå rett frem. Metoden *leggInn*() sørger for (utvider) at tabellen aldri er full. Det betyr her at den er tom hvis og bare hvis *fra* og *til* er like:

```
public T taUt()
{
  if (fra == til) throw new          // sjekker om køen er tom
    NoSuchElementException("Køen er tom!");

  T temp = a[fra];                   // tar vare på den første i køen
  a[fra] = null;                     // nuller innholdet
  fra++;                             // øker fra med 1
  if (fra == a.length) fra = 0;      // hopper til 0
  return temp;                       // returnerer den første
}
```

Programkode 4.2.2 e)

Koden til resten av metodene i klassen TabellKø tas opp i *Oppgave* 4 - 8.

Flg. programbit bruker en kø til å snu rekkefølgen på objektene i en stakk:

```
Stakk<Character> s = new TabellStakk<>();
Kø<Character> k = new TabellKø<>();

s.leggInn('A');
s.leggInn('B');
s.leggInn('C');

while (!s.tom()) k.leggInn(s.taUt());
while (!k.tom()) s.leggInn(k.taUt());

System.out.println(s);  // Utskrift: [A, B, C]
```

Effektivitet I starten av avsnittet ble det sagt at vanlig «køkultur» tilsier at når den første går ut av køen, beveger alle de andre seg et skritt fremover. Denne i og for seg fornuftige idéen er det umulig å implementere effektivt. Vi fant at en sirkelformet kø er et smart alternativ. Da beveger køen seg rundt i en sirkel, mens hver enkelt køståer er i ro inntil hun forlater køen. Dette fører til at både innlegging, kikking og uttak får konstant orden. Bedre enn det kan det ikke gjøres. Noen metoder kan kanskje optimaliseres litt. Se *Avsnitt* 4.2.3.

Oppgaver til Avsnitt 4.2.2

1. Tegn en sirkulær tabell med plass til 16 verdier. Sett på indeksene/posisjonene slik som i *Figur 4.2.2 f)*. Fra starten av er både *fra* og *til* lik 0.

   a) Legg inn fortløpende *A*, *B*, *C*, *D*, *E*, *F*, *G*, *H*, *I* og *J* i tabellen du har tegnet. Ta så ut fortløpende fem verdier. Hva blir nå *fra* og *til*? Sjekk ved hjelp av tegningen at koden for metoden `antall()` regner ut rett svar for antallet.

   b) Fortsett fra a) og legg inn fortløpende *K*, *L*, *M*, *N*, *O*, *P*, *Q*, *R*, *S* og *T* i tabellen. Ta så ut fortløpende tre verdier. Hva blir nå *fra* og *til*? Sjekk ved hjelp av tegningen at koden for metoden `antall()` fortsatt regner ut rett svar for antallet.

   c) Fortsett fra b). Ta ut fortløpende alle verdiene. Hva blir nå *fra* og *til*?

2. Tegn en sirkulær tabell med plass til 8 verdier. Sett på indekser fra 0 til 7. Sett både *fra* og *til* lik 0. Legg inn forløpende *A*, *B*, *C*, *D*, *E*, *F*, *G* og *H*. Hva blir *fra* og *til*?

3. Fortsett fra *Oppgave* 2. Ta ut fortløpende tre verdier. Legg så inn *I*, *J* og *K*. Hva blir *fra* og *til*? Ta ut fortløpende alle verdiene. Hva blir *fra* og *til*? Poenget er at *fra* lik *til* kan bety at køen er tom eller at den er full. Men hvis vi sørger for at den aldri er full, så betyr likhet mellom *fra* og *til* at køen er tom.

4. Flytt grensesnittet `Kø` over i ditt prosjekt (f.eks. under `hjelpeklasser`).

5. Flytt klassen `TabellKø` over i ditt prosjekt (f.eks. under `hjelpeklasser`). Legg inn de ferdige metodene `antall()`, `leggInn()`, `utvidTabell()` og `taUt()` inn i klassen.

6. Lag kode for metodene `kikk()`, `tom()` og `nullstill()` i `TabellKø`. Se grensesnittet `Kø`.

7. Lag kode for metoden `toString` i `TabellKø`. Hvis køen er tom skal den returnere tegnstrengen "[]". Hvis den f.eks. inneholder *A*, *B* og *C* skal den returnere "[A, B, C]".

8. Det kan være aktuelt å kjenne til hvor langt ut i køen en bestemt verdi ligger. Legg inn metoden `public int indeksTil(T verdi)` som en ekstra metode i `TabellKø`. Den skal returnere posisjonen til første forekomst av *verdi* i køen. Ligger *verdi* først er posisjonen 0, nest først 1, osv. Hvis *verdi* ikke ligger i køen, skal metoden returnere -1.

9. Lag metoden `public static <T> void snu(Stakk<T> A)`. Den skal snu rekkefølgen av verdiene på stakken *A*. Bruk en kø som hjelpemiddel i kodingen.

10. Lag metoden `public static <T> void snu(Kø<T> A)`. Den skal snu rekkefølgen av verdiene i køen *A*. Bruk en stakk som hjelpemiddel i kodingen.

11. Lag metoden `public static <T> void snu(Kø<T> A)`. Den skal snu rekkefølgen av objektene i køen `A`. Bruk to hjelpekøer. Parametertypen til `A` er `Kø<T>`. Da er det kun metodene i grensesnitt `Kø<T>` som kan brukes.

12. Som *Oppgave* 11, men bruk kun én hjelpekø og noen enkelte hjelpevariabler.

13. Som *Oppgave* 11, men bruk kun noen enkelte hjelpevariabler.

14. Metoden `public static <T> void sorter(Kø<T> A, Comparator<? super T> c)` skal sortere verdiene i `A` vha. komparatoren `c`. Lag metoden. Du kan bruke to hjelpekøer.

15. Som i *Oppgave* 14, men ved hjelp av én hjelpekø og noen enkelte hjelpevariabler.

16. Som i *Oppgave* 14, men kun ved bruke hjelpevariabler (dvs. ingen hjelpestrukturer).

17. Lag klassen `KøStakk<T>`. Klassen skal implementere grensesnittet `Stakk<T>` og bruke to køer som intern datastruktur.

18. Lag klassen `StakkKø<T>`. Klassen skal implementere grensesnittet `Kø<T>` og bruke to stakker som intern datastruktur.

4.2.3 Binære optimaliseringer i en sirkulær kø

Klassen `TabellKø` i *Avsnitt* 4.2.2 implementerte grensesnittet `Kø` og brukte en «sirkulær» tabell som intern datastruktur. Det er nok ikke mulig å finne en lurere måte å lage en kø på enn det. Men det er noen interessante (og morsomme) binære teknikker som kanskje kan få noen operasjoner til å gå litt raskere. Klassen `ArrayDeque` i *java.util* bruker slike teknikker. De grunnleggende binærteknikkene som vi trenger, er beskrevet i *Delkapittel* 1.7.

I `TabellKø` oppretter standardkonstruktøren en tabell med lengde 8. I utvidelser blir lengden doblet. Det betyr at tabellengden normalt vil være på formen 8, 16, 32, osv. Dette er tall på formen 2^k. La som eksempel tabellen ha lengde $n = 2^8 = 256$. Det betyr at på binærform vil n bestå av et 1-tall med 8 etterfølgende 0-er, dvs. $n = 100000000_2$ og dermed $n - 1 = 255 = 11111111_2$. Hvis tallene er representert på 32 biters format, kommer det i tillegg en serie ledende 0-er. Hvis et ikke-negativt tall m er mindre enn 256, vil tallet ha minst 24 ledende 0-er. De siste 8 sifrene kan være hva som helst. De er markert med en × i *Figur* 4.2.3 *a)*:

m	=	·	·	·	0	0	×	×	×	×	×	×	×	×
n − 1	=	·	·	·	0	0	1	1	1	1	1	1	1	1
m & (n − 1)	=	·	·	·	0	0	×	×	×	×	×	×	×	×

Figur 4.2.3 a) : m < 256, n − 1 = 255, m & (n − 1) = m

Hvis 1 og × er biter, vil alltid uttrykket 1 & × være lik × uansatt om × er lik 0 eller 1. Dermed vil vi, som *Figur* 4.2.3 *a)* viser, alltid få at m & (n − 1) = m så sant 0 <= m < n og n = 256. Men hvis m = n blir det annerledes. Vi har 1 & 0 = 0 & 1 = 0 og dermed, som *Figur* 4.2.3 *b)* under viser, at n & (n − 1) = 0 hvis n = 256.

n	=	·	·	·	0	1	0	0	0	0	0	0	0	0
n − 1	=	·	·	·	0	0	1	1	1	1	1	1	1	1
n & (n − 1)	=	·	·	·	0	0	0	0	0	0	0	0	0	0

Figur 4.2.3 b) : n = 256, n − 1 = 255, n & (n − 1) = 0

Figur 4.2.3 *a)* og *Figur* 4.2.3 *b)* viser hvordan det er for n = 256. Men dette gjelder generelt:

> **Setning 4.2.3 a)** La n være et heltall på formen 2^k (k >= 0) og m et heltall med 0 <= m < n. Da er m & (n − 1) = m og n & (n − 1) = 0.

Vi kan bruke *Setning* 4.2.3 *a)* til å omkode `leggInn`-metoden i *Programkode* 4.2.2 *c)*. I koden ble variabelen `til` økt med 1 (dvs. `til++`) etter en innlegging. Hvis den etter økingen ble lik `a.length`, ble den satt til 0. Hvis vi vet at tabellen `a` har en lengde på formen 2^k, så gjelder *Setning* 4.2.3 *a)*. Dermed vil koden bli noe mer effektiv hvis vi gjør slik:

```
public boolean leggInn(T verdi)
{
  a[til] = verdi;                                  // ny verdi bakerst i køen
  til = (til + 1) & (a.length - 1);                // øker med 1

  if (fra == til) a = utvidTabell(2*a.length);     // sjekker og dobler
  return true;                                     // vellykket innlegging
}
```

Programkode* 4.2.3 *a)

Vi kan også bruke *Setning* 4.2.3 *a)* i *taUt*-metoden i *Programkode 4.2.2 e)*:

```
public T taUt()
{
  if (fra == til)                      // sjekker om køen er tom
    throw new NoSuchElementException("Køen er tom!");

  T temp = a[fra];                     // tar vare på den første i køen
  a[fra] = null;                       // nuller innholdet
  fra = (fra + 1) & (a.length - 1);    // øker fra med 1
  return temp;                         // returnerer den første
}
```

Programkode 4.2.3 b)

Det er også mulig å effektivisere metoden `antall()`. Vi antar som før at lengden på tabellen *a* er lik *n* der *n* er på formen 2^k. Ta som eksempel at $n = 2^8 = 256 = 100000000_2$. La *m* være negativ, dvs. $-n <= m < 0$. Negative tall bruker to-komplement og dermed vil *m* starte med 24 1-ere. De åtte siste bitene i *m* kan være hva som helst. Det er markert med × i *Figur* 4.2.3 *c)*:

m	=	·	·	·	1	1	×	×	×	×	×	×	×	×
n	=	·	·	·	0	1	0	0	0	0	0	0	0	0
n − 1	=	·	·	·	0	0	1	1	1	1	1	1	1	1
m + n	=	·	·	·	0	0	×	×	×	×	×	×	×	×
m & (n − 1)	=	·	·	·	0	0	×	×	×	×	×	×	×	×

Figur 4.2.3 c) : n = 256, -n <= m < 0, m + n = m & (n − 1)

Regnestykket × + 0 blir lik × unasett om × er 0 eller 1. Mens 1 + 1 gir 2, dvs. 1 og 1 i mente. *Figur* 4.2.3 *c)* viser resultatet når *m* og *n* adderes på binærform. Resultatet blir et tall som starter med 24 0-biter. De åtte siste bitene er identisk med de siste åtte bitene i *m*. Men siste rad viser at *m* & (*n* − 1) gir oss det samme. Dette stemmer for *n* = 256. Men det gjelder generelt:

> **Setning 4.2.3 b)** La *n* være et heltall på formen 2^k ($k >= 0$) og *m* et heltall med $-n <= m < 0$. Da er $m + n = m \,\&\, (n - 1)$.

Ved hjelp av *Setning* 4.2.3 *a)* og *Setning* 4.2.3 *b)*, kan metoden `antall()` effektiviseres slik:

```
public int antall()
{
  return (til - fra) & (a.length - 1);
}
```

Programkode 4.2.3 c)

Hvis vi lager en køinstans ved hjelp av standardkonstruktøren i `TabellKø`, vil startlengden på tabellen være 8 og dermed vil den og lengden ved alle senere utvidelser være på formen 8, 16, 32, 64, osv. Men bruker vi den andre konstruktøren kan det gå galt. Vi må derfor gjøre om den. Hvis f.eks. det bes om en lengde på 100, lar vi konstruktøren opprette en tabell med lengde 128. Med andre ord må vi for hver aktuell lengde finne det minste heltallet på formen 2^k som er større enn lengden.

Klassen `Integer` har metoden *`highestOneBit`*`(int n)`. Den returnerer et heltall med en 1-er på den plassen der den første (fra venstre) 1-eren i *n* står og med 0-er ellers. Dette er (hvis *n* > 0) det største tallet på formen 2^k som er mindre enn eller lik *n*. Hvis f.eks. *n* = 100, returnerer metoden 64. Ganger vi dette med 2 (bitforskyvning på 1) får vi ønsket verdi. Hvis det bes om en tabellengde på 0, settes den til 1:

```
public TabellKø(int lengde)
{
  if (lengde < 0) throw new
    IllegalArgumentException("Negativ tabellengde(" + lengde + ")!");

  lengde = lengde <= 1 ? 1 : Integer.highestOneBit(lengde - 1) << 1;

  if (lengde < 0)    // kan få oversvømmelse her
    throw new IllegalArgumentException("For stor tabellengde!");

  a = (T[])new Object[lengde];
  fra = til = 0;
}
```

Programkode* 4.2.3 *d)

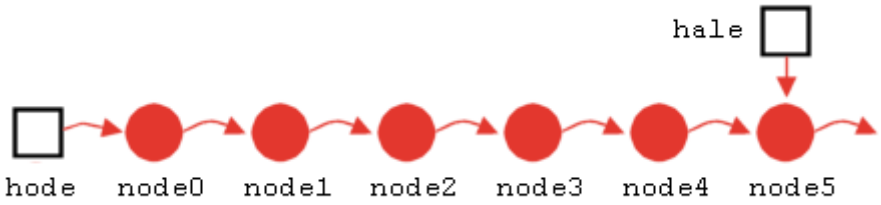
Konklusjon Det vil nok vise seg at den effektivitetsgevinsten vi får ved å kode metodene slik som i dette avsnittet, er helt marginal i forhold til slik det er gjort i *Avsnitt* 4.2.2. Men denne spesielle teknikken som vi bruker får å hoppe over «skjøten» i den «sirkulære» tabellen, er interessant i seg selv.

Oppgaver til Avsnitt 4.2.3

1. Oppgavene 4 - 7 i *Avsnitt* 4.2.2 gikk ut på å fullføre klassen `TabellKø`. I dette avsnittet har vi laget nye versjoner av *`leggInn`*`()`, *`taUt`*`()`, *`antall`*`()` og en *`konstruktør`*. Erstatt de gamle versjonene av disse metodene i klassen `TabellKø` med de nye.
2. Lag nye versjoner av metodene *`nullstill`*`()`, *`toString`*`()` og *`indeksTil`*`()`. Se Oppgave 6, 7 og 8 i *Avsnitt* 4.2.2. Bruk den binære teknikken.

4.2.4 En lenket kø

Det er mulig å implementere en kø ved hjelp av en enkeltlenket liste med hode og hale. Køen starter ved hodet og slutter ved halen. Det å legge inn en verdi på slutten vil være av konstant orden. Det samme for det å ta ut (eller kikke på) den første i køen. Med andre ord vil både *leggInn*(), *kikk*() og *taUt*() da bli effektive operasjoner.



Figur 4.2.4 a) : En lenket liste med hode og hale kan brukes til en kø

Vi har tidligere laget klassen EnkeltLenketListe og dens *leggInn*-metode legger verdier bakerst. Køens *taUt*() og *kikk*() kan kodes ved hjelp av *remove*() og *hent*(). Dermed kunne vi la EnkeltLenketListe implementere Kø (se LinkedList i java.util):

```
public class EnkeltLenketListe<T> implements Liste<T>, Kø<T>
```

Programkode 4.2.4 a)

Det som mangler for at dette skal fungere, er de to metodene *taUt*() og *kikk*(). De øvrige metodene som inngår i Kø, ligger allerede i EnkeltLenketListe og kan brukes som de er. Flg. metoder må inn i EnkeltLenketListe:

```
public T taUt()
{
  if (tom()) throw new NoSuchElementException("Køen er tom!");
  return fjern(0);   // returnerer (og fjerner) den første
}

public T kikk()
{
  if (tom()) throw new NoSuchElementException("Køen er tom!");
  return hent(0);    // henter den første
}
```

Programkode 4.2.4 b)

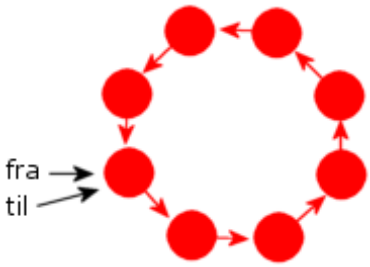
Flg. eksempel viser hvordan dette kan brukes:

```
Kø<Integer> kø = new EnkeltLenketListe<>();

for (int i = 1; i <= 10; i++) kø.leggInn(i);

while (!kø.tom())
{
  System.out.print(kø.taUt() + " ");
}
```

Programkode 4.2.4 c)

Et problem med å bruke en EnkeltLenketListe er at en node som fjernes, forsvinner og når en ny verdi skal inn, må det lages en ny node, dvs. et kall på new. Et kall på new er relativt sett en kostbar operasjon. En idé kunne være å ta vare på de nodene som fjernes og bruke

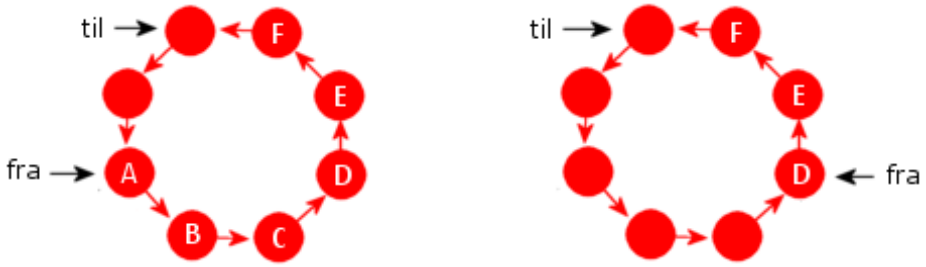
dem på nytt. Vi kunne da starte med et lite antall noder, f.eks. 8 stykker og organisere dem i en sirkel. Det blir omtrent samme idé som i en «sirkulær kø» i *Avsnitt* 4.2.2.



Figur 4.2.4 b) : Sirkulær kø

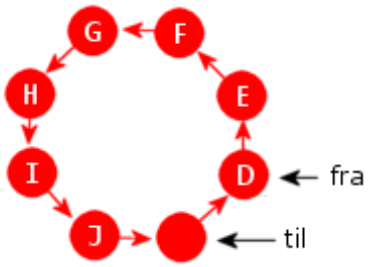
I *Figur* 4.2.4 *b)* til venstre er det 8 noder i sirkel. I tillegg tenker vi oss at vi har to pekere *fra* og *til*. Der skal *fra* peke til den første noden i køen. Pekeren *til* skal gå til den første ledige noden, dvs. den som kommer rett etter den siste i køen regnet mot klokken (pilenes retning). Når køen er tom er *fra* = *til*.

Vi legger først inn verdiene/bokstavene *A*, *B*, *C*, *D*, *E* og *F*. dvs. seks kall på *leggInn()*. Se venstre del av *Figur* 4.2.4 *c)* under. Deretter gjør vi tre kall på *taUt()*. Da forsvinner *A*, *B* og *C*. Se høyre del av *Figur* 4.2.4 *c)* under.



Figur 4.2.4 c) : 1) Seks kall på leggInn(). 2) Tre kall på taUt().

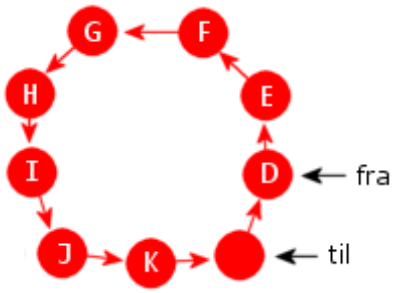
Etter at verdiene *A*, *B*, *C*, *D*, *E* og *F* ble lagt inn, dvs. etter seks kall på *leggInn()*, pekte *fra* til den første i køen (dvs. til *A*) og *til* pekte én forbi den siste i køen (dvs. én forbi *F*). Etter tre kall på *taUt()* flyttet pekeren *fra* seg til den som så er først i køen, dvs. til *D*. Samtidig har verdiene blitt «nullet» i de nodene som er frigjort.



Figur 4.2.4 d) : Én ledig node

Vi legger inn fire nye verdier: *G*, *H*, *I* og *J*. Se *Figur* 4.2.4 *d)* til venstre. Da er det fortsatt en ledig node. Men hvis vi skal legge inn en verdi til, f.eks. *K*, vil listen bli full og *fra* og *til* blir like. Men det skal jo egentlig bety at køen er tom. Vi løser det ved å sette inn en ny og «tom» node mellom *til* og *fra*. Dermed kan *K* legges inn og *til* flyttes til neste ledige node. Dermed bevares «sirkelen».

Figur 4.2.4 *e)* ned til venstre viser hvordan det blir etter at en «tom» node og verdien *K* er satt inn.



Figur 4.2.4 e) : En tom node

Hvis vi nå ønsker å sette inn en verdi til, f.eks. *L*, gjør vi på samme måte som sist. Dvs. vi setter inn en ny og «tom» node mellom *til* og *fra*. Hvis vi isteden skal ta ut en verdi, dvs. et kall på *taUt()*, er det ingen problemer siden det i det tilfellet er *fra* som flyttes.

Hva med antall i køen? Her kan vi ikke finne det på annen måte enn å telle opp hvor mange noder der er i intervallet [*fra*,*til*>. Da er det bedre å bruke en *antall*-variabel. Dermed kan vi avgjøre om køen er tom på to måter: a) *antall* == 0 eller b) *til* == *fra*.

Ideene over kan oppsummeres slik:

- Køen skal organiseres som en sirkelformet pekerkjede med en fast startstørrelse.
- En peker *fra* skal gå til den første i køen.
- En peker *til* skal gå til den første ledige noden, dvs. én forbi den siste i køen.
- Hvis *til* == *fra*, er køen tom.
- Køen skal aldri være full. Ved en innlegging skal det alltid være en ledig node. Hvis det er kun én, utvides pekerekjeden med en ekstra node mellom *til* og *fra*.
- En *antall*-variabel holder orden på antallet verdier i køen.

```
public class LenketKø<T> implements Kø<T>
{
  private static final class Node<T>   // en indre nodeklasse
  {
    private T verdi;         // nodens verdi
    private Node<T> neste;   // peker til neste node

    Node(Node<T> neste)      // nodekonstruktør
    {
      verdi = null; this.neste = neste;
    }
  }   // class Node

  private Node<T> fra, til;  // fra: først i køen, til: etter den siste
  private int antall;        // antall i køen

  private static final int START_STØRRELSE = 8;

  public LenketKø(int størrelse)  // konstruktør
  {
    // kode mangler
  }

  public LenketKø()  // standardkonstruktør
  {
    this(START_STØRRELSE);
  }

  public int antall()
  {
    return antall;
  }

  public boolean tom()
  {
    return fra == til;  // eller antall == 0
  }

  // resten av metodene skal inn her

} // class LenketKø
```

Programkode 4.2.4 d)

Konstruktøren `LenketKø(int størrelse)` skal opprette en sirkulær pekerkjede med så mange noder som *størrelse* sier og *til* og *fra* skal begge peke på samme node:

```
public LenketKø(int størrelse)   // konstruktør
{
  til = fra = new Node<>(null);  // lager den første noden

  Node<T> p = fra;               // en hjelpevariabel
  for (int i = 1; i < størrelse; i++)
  {
    p = new Node<>(p);           // lager resten av nodene
  }
  fra.neste = p;                 // for å få en sirkel

  antall = 0;                    // ingen verdier foreløpig
}
```

Programkode 4.2.4 e)

I *leggInn*() vil *til* peke på første ledige node og der legges *verdi*. Hvis *til.neste* er lik *fra*, oppretter vi en ny node mellom *til* og *fra*:

```
public boolean leggInn(T verdi)   // null-verdier skal være tillatt
{
  til.verdi = verdi;              // legger inn bakerst

  if (til.neste == fra)           // køen vil bli full - må utvides
  {
    til.neste = new Node<>(fra);  // ny node mellom til og fra
  }

  til = til.neste;                // flytter til
  antall++;                       // øker antallet

  return true;                    // vellykket innlegging
}
```

Programkode 4.2.4 f)

Metoden *kikk*() skal, hvis køen ikke er tom, returnere køens første verdi. Mens *taUt*() i tillegg skal fjerne verdien:

```
public T kikk()
{
  if (tom()) throw new NoSuchElementException("Køen er tom!");
  return fra.verdi;           // returnerer verdien
}

public T taUt()
{
  if (tom()) throw new NoSuchElementException("Køen er tom!");

  T tempverdi = fra.verdi;    // tar vare på verdien i fra
  fra.verdi = null;           // nuller innholdet i fra

  fra = fra.neste;            // flytter fra
  antall--;                   // reduserer antallet

  return tempverdi;           // returnerer verdien
}
```

Programkode 4.2.4 g)

Oppgaver til Avsnitt 4.2.4

1. La klassen `EnkeltLenketListe` implementere `Kø`. Se *Programkode 4.2.4 a)*. Legg så inn de to metodene *`kikk()`* og *`taUt()`*. Sjekk at *Programkode 4.2.4 c)* virker.
2. Lag metoden `public String toString()` i klassen `LenketKø`. Den skal returnere en tegnstreng med køens innhold i rekkefølge fra den første til den siste. Hvis køen er tom skal tegnstrengen inneholde [] og hvis den f.eks. inneholder bokstavene A, B og C skal den inneholde [A, B, C].
3. Lag metoden `public void nullstill()` i klassen `LenketKø`. Hvis antallet noder er mindre enne eller lik `START_STØRRELSE`, skal verdiene i alle nodene nulles. Hvis antallet er større, skal vi etterpå ha at antallet noder er lik `START_STØRRELSE`. Men både i de resterende nodene og i de nodene som skal forsvinne, skal verdien nulles.
4. Metoden `public static <T> void sorter(Kø<T> kø, Stakk<T> stakk, Comparator<? super T> c)` skal sortere *`kø`*, mens *`stakk`* kun skal fungere som hjelpestruktur. Metoden skal kodes uten bruk av andre hjelpestrukturer. Sjekk så at metoden virker uansett hva slags kø eller stakk vi bruker: `TabellKø`, `EnkeltLenketListe` (se oppgave 1), `LenketKø`, `TabellStakk` eller `LenketStakk`. F.eks. skal flg. kode virke:

```
Integer[] a = Tabell.randPermInteger(10);

Kø<Integer> kø = new EnkeltLenketListe<>();
for (Integer i : a) kø.leggInn(i);

System.out.println(kø);    // usortert

Stakk<Integer> stakk = new TabellStakk<>();

sorter(kø, stakk, Comparator.naturalOrder());

System.out.println(kø);    // sortert
```

5. (*) Metoden `public static <T> void sorter(Kø<T> kø, Comparator<? super T> c)` skal sortere *`kø`*. Det skal ikke brukes noen hjelpestrukturer. En eller noen enkeltvariabler er tillatt. Metoden skal virke uansett hva slags kø det er.

4.2.5 Queue i java.util

Java har flg. grensesnitt for en kø:

```
public interface Queue<T> extends Collection<T>
{
  public boolean add(T verdi);    // legger inn bakerst
  public boolean offer(T verdi);  // legger inn bakerst
  public T element();             // ser på den første (tom kø: kaster unntak)
  public T peek();                // ser på den første (tom kø: returnerer null)
  public T remove();              // tar ut den første (tom kø: kaster unntak)
  public T poll();                // tar ut den første (tom kø: returnerer null)

  // + metoder som arves fra Collection<T>

} // interface Queue
```

Programkode 4.2.5 a)

Det fremgår av kommentarene i Queue (se over) hva *element*(), *peek*(), *remove*() og *poll*() gjør. Metodene *add*() og *offer*() legger begge inn en verdi bakerst i køen. Hvis innleggingen er vellykket, returnerer begge *true*. Det finnes køtyper der den interne lagringsstrukturen har en maksimal størrelse. Det gjelder f.eks. klassen ArrayBlockingQueue. Hvis det ikke er plass, vil *offer*() gi *false*, mens *add*() kaster unntak. Det er også mulig å ha en kø der like verdier ikke er tillatt. I så fall skal både *offer*() og *add*() gi false hvis en forsøker å legge inn en verdi som finnes fra før.

Både ArrayDeque og LinkedList kan brukes som en kø. En ArrayDeque er normalt mest effektiv, men tillater ikke null-verdier. Hvis en har behov for det, er LinkedList alternativet.

```
Character[] bokstaver = {'A','B','C'};                    // bokstaver
Queue<Character> kø = new ArrayDeque<>();                 // oppretter en kø
for (char c : bokstaver) kø.offer(c);                     // bruker offer
while (!kø.isEmpty()) System.out.print(kø.poll() + " "); // tar ut med poll

// Utskrift: A B C
```

Programkode 4.2.5 b)

Dette kunne også vært kodet med LinkedList, *add*() og *remove*():

```
Character[] bokstaver = {'A','B','C'};                    // bokstaver
Queue<Character> kø = new LinkedList<>();                 // oppretter en kø
for (char c : bokstaver) kø.add(c);                       // bruker add
while (!kø.isEmpty()) System.out.print(kø.remove() + " "); // tar ut med remove

// Utskrift: A B C
```

Programkode 4.2.5 c)