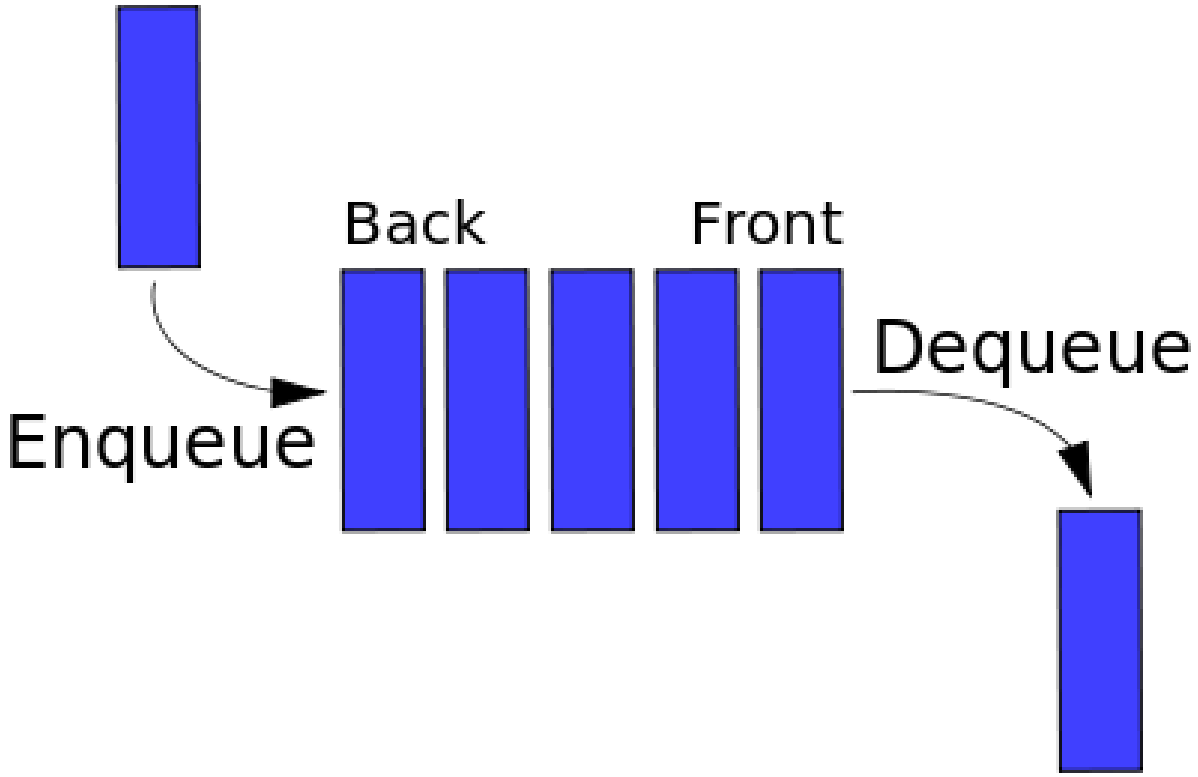
Køer

Hva er en kø?

- En lineær datastruktur der vi til enhver tid kun har tilgang til elementet som ble lagt inn *først*
- Et nytt element legges alltid til *sist* i køen
- Skal vi ta ut et element, tar vi alltid det *første*
- Sammenlikning: Kø med kunder som venter på betjening
- Elementet som har ventet *lengst* er det som behandles *først:* FIFO – First In, First Out

Innsetting og fjerning av data: “enqueue” og “dequeue”



Anvendelser av kø

- Kryptering/dekryptering (avsnitt 5.3 i læreboka)
- Simulering av systemer med en eller flere køer av objekter som må vente på betjening (første obligatoriske oppgave)
- I operativsystemer, der brukere eller prosesser må vente i kø for å få tilgang til en delt ressurs (RAM, CPU, printer, disk, nettverksport...)
- Algoritmer som krever at data legges systematisk i kø for å kunne behandles i riktig rekkefølge (f.eks. radix-sortering og bredde-først traversering av trær og grafer)

Operasjonene på en kø

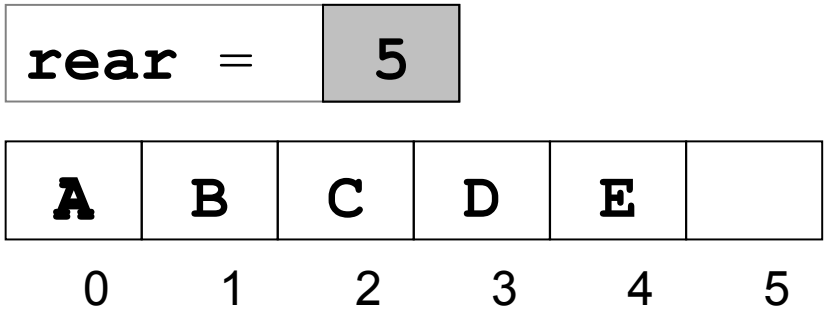
- **enqueue**: legge til et element bakerst i køen
- **dequeue**: ta ut første element i køen
- **first**: se på første element i køen
- **isEmpty**: sjekk om køen er tom
- **size**: antall elementer som er lagret i køen

Kø implementert med enkel array

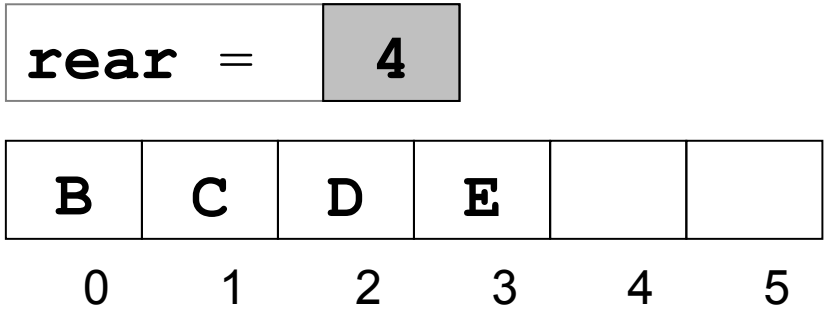
- Mulig løsning:
  - La fronten av køen alltid ligge på indeks 0
  - Hold rede på lengden av køen med en indeksvariabel `rear`
  - Sett inn nye elementer på indeks `rear` og øk denne med 1 for hver innsetting – enqueue blir $O(1)$
- Problem:
  - Hver gang vi tar ut et element, må hele resten av køen flyttes "et hakk" fremover i arrayen
  - Dequeue blir $O(n)$ for en kø med n elementer
  - Ønsker en løsning der både enqueue og dequeue er $O(1)$

Eksempel: Kø implementert med array

- Etter enqueue av verdiene A B C D E :

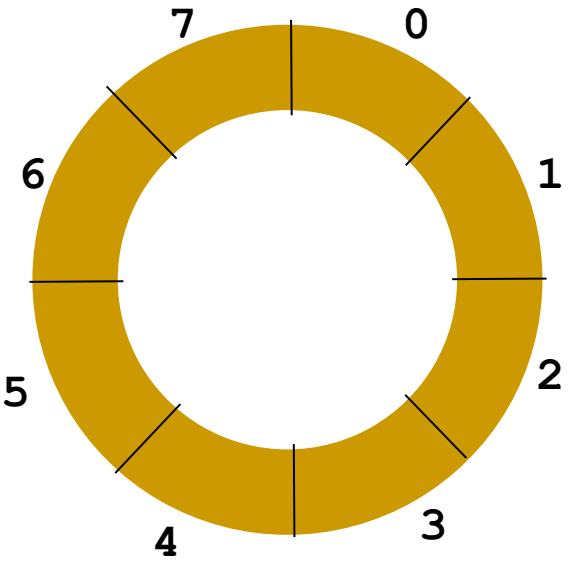


- Etter en dequeue, alle gjenværende verdier må flyttes én indeks mot venstre:



Kø implementert med “sirkulær array”

- Lagrer elementene i køen i en array
- Holder rede på indeksene til starten (`front`) og slutten (`rear`) på køen
- Lagrer antall elementer i `count`
- Innsetting og fjerning blir *O*(1) fordi:
  - Vi lar køen *sirkulere* (med wrap-around) i arrayen
  - Slipper da å flytte alle elementer et “hakk” frem eller tilbake ved innsetting/fjerning
  - Gjøres enkelt med *modulusaritmetikk*



Kø og mod-operatoren (%) i Java

- Operatoren % (mod*) beregner *rest* ved heltallsdivisjon:

```
1%5 = 1,  2%5 = 2,  5%5 = 0,  8%5 = 3
```

- Kan brukes til å beregne indeksene `front` og `rear` i en kø implementert som sirkulært array
- Enqueue (`rear` er første ledige indeks):

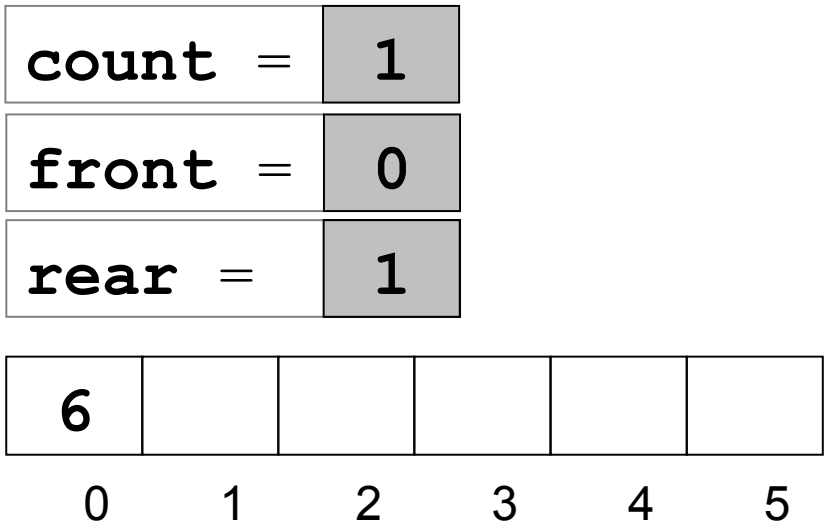
```
rear = (rear + 1) % queue.length;
```

- Dequeue (`front` er indeks til start av køen):

```
front = (front + 1) % queue.length;
```

*: Beregning av rest ved heltallsdivisjon kalles "modulo operation" på engesk:

Eksempel: Kø med sirkulær array



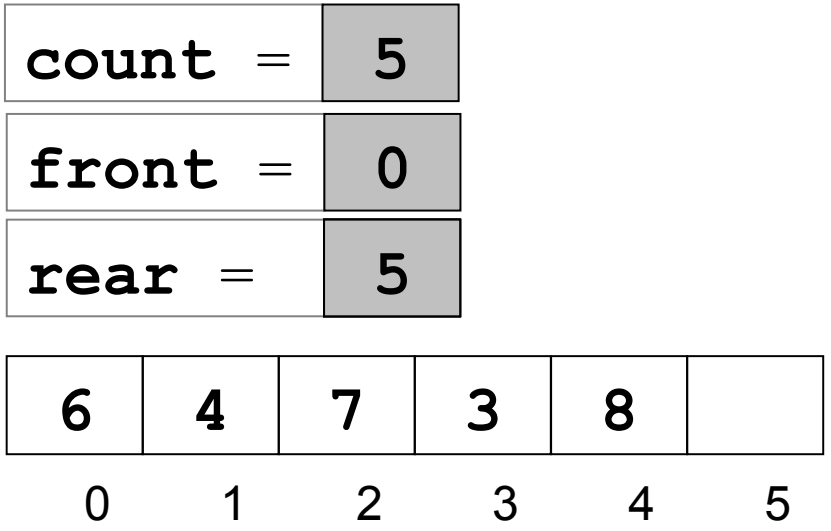

```
//Java Code
Queue q = new Queue();
q.enqueue(6);
```

Setter inn på indeks rear = 0:

rear = (rear + 1) % q.length

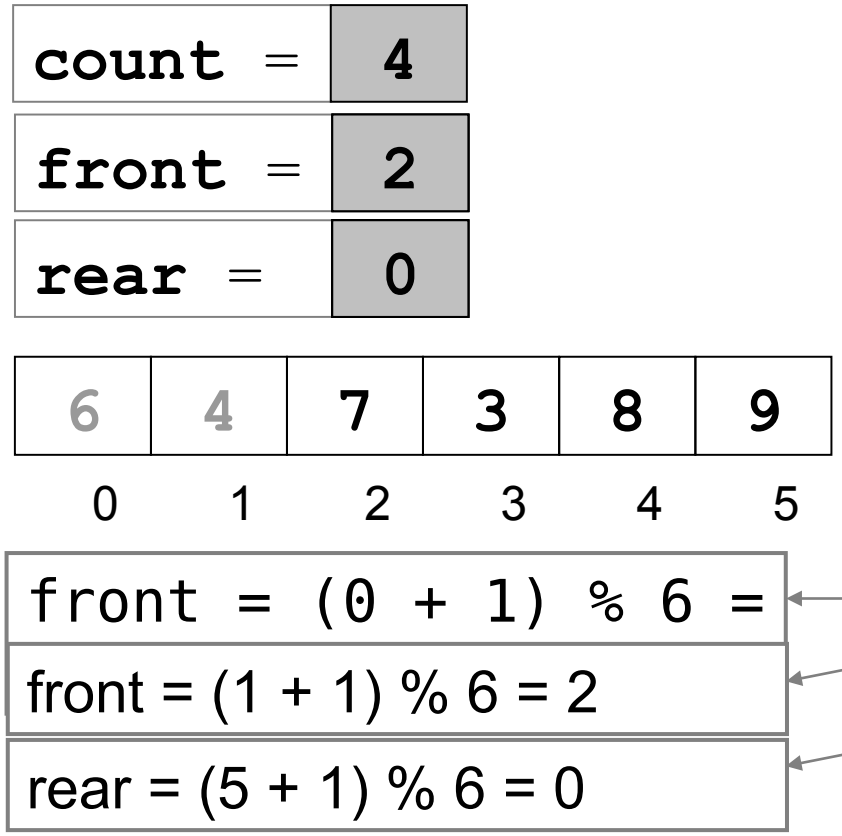
rear = (0 + 1) % 6 = 1

Eksempel: Kø med sirkulær array forts.



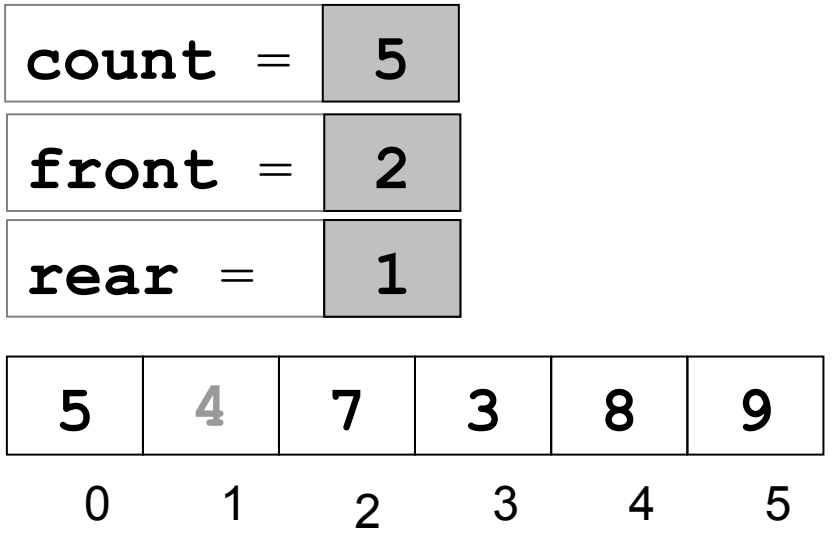

```
//Java Code
Queue q = new Queue();
q.enqueue(6);
q.enqueue(4);
q.enqueue(7);
q.enqueue(3);
q.enqueue(8);
```

Eksempel: Kø med sirkulær array forts.




```java
//Java Code
Queue q = new Queue();
q.enqueue(6);
q.enqueue(4);
q.enqueue(7);
q.enqueue(3);
q.enqueue(8);
q.dequeue();//front = 1
q.dequeue();//front = 2
q.enqueue(9);//rear = 0
```

Eksempel: Kø med sirkulær array forts.




```
//Java Code
Queue q = new Queue();
q.enqueue(6);
q.enqueue(4);
q.enqueue(7);
q.enqueue(3);
q.enqueue(8);
q.dequeue();//front = 1
q.dequeue();//front = 2
q.enqueue(9);//rear = 0
q.enqueue(5);
```

rear = (0 + 1) % 6 = 1

Animasjon av innsetting og fjerning av data:
https://www.cs.usfca.edu/~galles/visualization/QueueArray.html

Kø med sirkulær array: Implementasjoner

- Enkel implementasjon: IntQueue.java
  - Kan bare lagre heltall
  - Ingen feilsjekking
- Læreboka: CircularArrayQueue.java
  - Implementerer `public interface Queue<T>`
  - Bruker en sirkulær array med generiske dataelementer
  - Utvider arrayen hvis den blir full av data ved en `enqueue`
  - Sjekker for feil (tom kø) for metodene `dequeue` og `first`

Grensesnitt for en kø-ADT i Java *

```
public interface Queue<T>
{
   // Legge nytt element bakerst i køen
   public void enqueue(T element);

   // Fjerne første element fra køen;
   public T dequeue();

   // Se på første element i køen;
   public T first();

   // Avgjøre om køen er tom
   public boolean isEmpty();

   // Antall elementer i køen
   public int size();
}
```

*: Som i læreboka

Javas innebygde implementasjon av vanlig kø

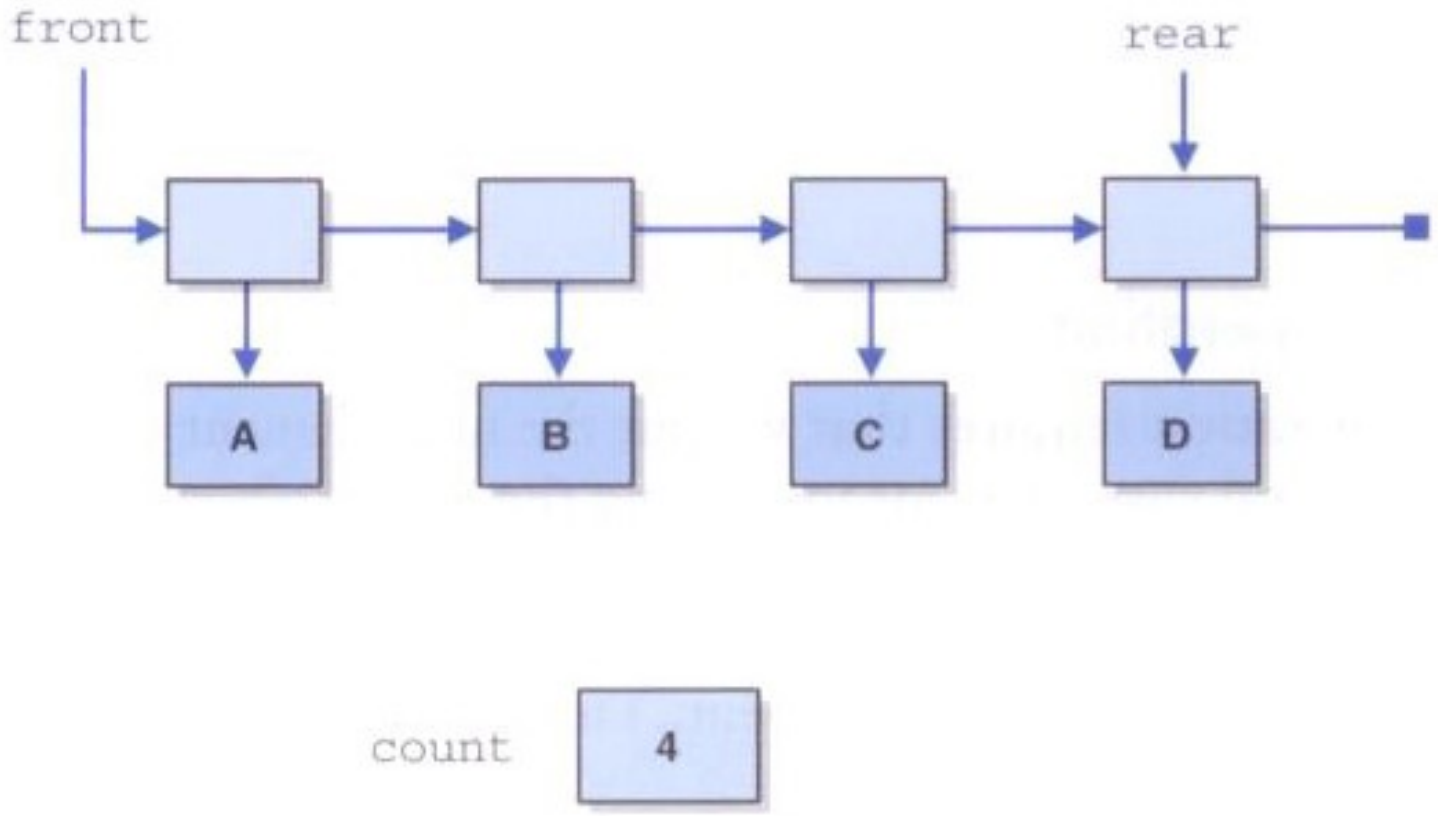
- Bruker en lenket liste i stedet for sirkulær array
- Enqueue-operasjonen kalles "add"
- Dequeue kalles "remove"
- Eksempel på bruk:
  - "Queue Interface In Java"

- Dere velger selv hvilken kø-implementasjon som brukes i første obligatoriske oppgave

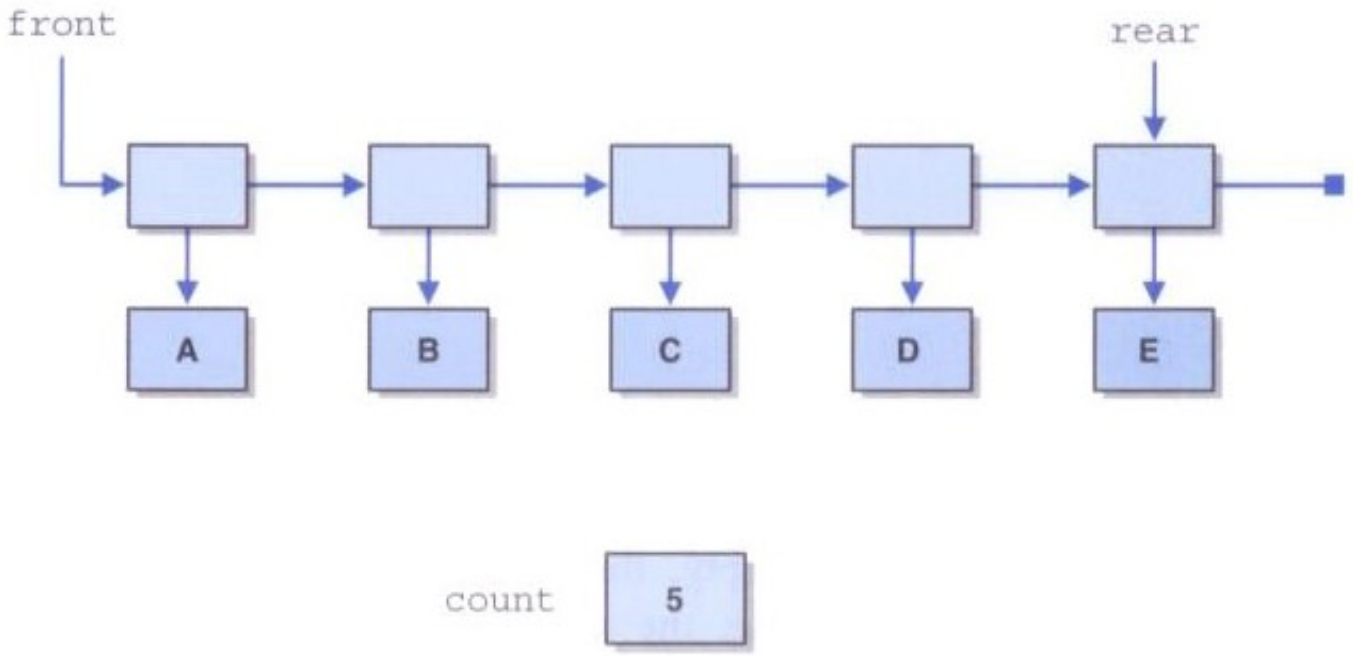
Kø implementert som lenket liste

- Køen har:
  - Referanser til noden først (`front`) og sist (`rear`) i køen
  - En teller (`count`) som inneholder antall elementer i kø
- Hver node i køen har:
  - Referanse/peker til noden som står på neste plass i køen
  - Referanse/peker til et objekt som inneholder dataene
- Alle operasjoner på køen blir *O*(1)
- Se avsnitt 5.6 i læreboka
- Lærebokas Java-kode: LinkedQueue.java

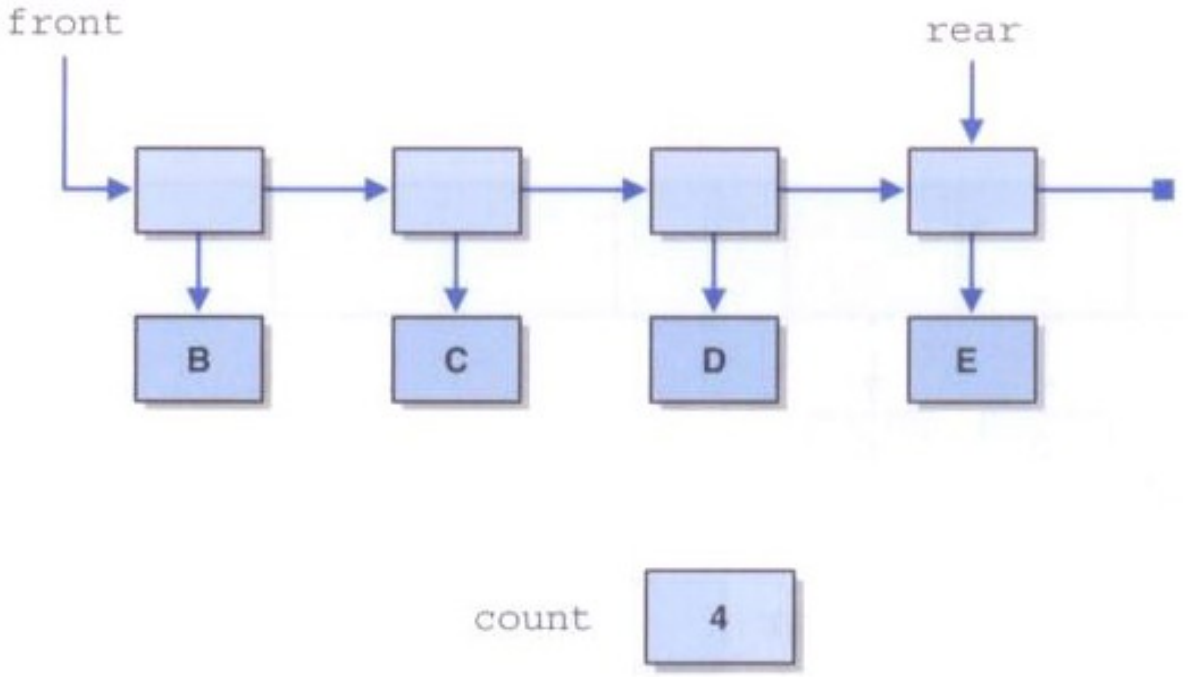
Kø implementert som lenket liste



Etter enqueue(E):



Etter dequeue():



Simulering

- Bruker et system til å etterlige oppførselen til et annet system
- Nyttig hvis det f.eks. er for dyrt, farlig eller tidkrevende å eksperimentere med det virkelige systemet
- Bruksområder:
  - Design og dimensjonering
  - Predikere fremtidig oppførsel
  - Forklare historiske data og fysiske fenomen
- Fysiske, matematiske eller *datamaskinbaserte* simuleringer

Simulering med datamaskiner

- Programutførelsen etterligner systemets oppførsel
- Datastrukturer beskriver systemets objekter
- Handlinger beskrives med operasjoner på dataene
- Reglene i systemet beskrives med algoritmer
- Stor fleksibilitet i forhold til fysiske modeller:
  - Lett å endre modell (kode) og omgivelser (input)
  - Raskt å teste et stort antall ganger
  - Lett å legge inn statistisk usikkerhet (tilfeldige tall)

Et eksempel på *tidsdrevet* køsimulering: Postkontoret *

- Simulerer aktiviteten på et postkontor i løpet av et bestemt antall tidsenheter (f.eks. minutter)
- Det er et fast antall betjente kasser/skranker
- Alle kunder som ankommer stiller seg i én kø
- Betjening av første kunde i køen starter med en gang en kasse er ledig
- Vil observere ventetider, kølengde og effektivitet

*: Eksemplet har mye til felles med oblig. 1: “Flyplassen”

Parametre i simuleringsmodellen

- Antall tidssteg som simuleringen skal gå
- Antall kasser som er åpne på postkontoret
- Lengste betjeningstid:
  - Trekker tilfeldige tider mellom 1 og en gitt maks.tid
- Hvor ofte kundene ankommer:
  - Maksimalt én kundeankomst per tidsenhet (forenkling)
  - Sannsynligheten P_K for at en kunde kommer er gitt
  - Trekker tilfeldig i hvert tidssteg om det kommer eller ikke kommer en kunde og stiller seg i kø

Algoritme for postkontoret

1 Sett alle kasser til å være ledige

2 Sett køen av kunder til å være tom

3 For hvert tidssteg:

  3.1 Trekk et tilfeldig (uniformt fordelt) tall *r* mellom 0 og 1

  3.2 Hvis $r < P_k$, sett ny kunde i kø

  3.3 For hver kasse som er ledig:

    3.3.1 Ta første kunde ut av køen

    3.3.2 Beregn og lagre tidspunkt for når kassen vil bli ledig igjen

  3.4 Lagre statusinformasjon for tidssteget

4 Beregn og skriv ut resultater fra simulering

Implementasjon

- Egen klasse for kunder, som tar vare på ankomsttiden
- Brukes Javas innebygde køstruktur for kundekøen
- Egen klasse for kassene, som lagrer tidspunktet når kassen blir ledig
- Bruker metoder fra `java.util.Random` til å trekke tider og kundeankomster
- Lagrer og skriver ut informasjon om gjennomsnittlige ventetider for kundene og gjennomsnittlig antall kasser som stod tomme
- Se Javakoden

Prioritetskøer

- I vanlige FIFO-køer, setter vi inn elementer i den ene enden av køen, og tar ut elementer fra den andre enden.
- Av og til ønsker vi å *prioritere* elementene som ligger i køen.
- Noen må behandles med en gang, mens andre kan vente litt.
- En sykehuskø der pasientene blir prioritert etter hvor syke de er, er et eksempel på en prioritetskø.

Eksempel: Timesharing

- En datamaskin har oftest mange aktive programmer/ prosesser som skal kjøres “samtidig”
- En tradisjonell CPU utfører bare én operasjon om gangen
- Får til “samtidighet” med *timesharing*:
  - Prosessene står i en kø og venter på eksekvering
  - “Første” prosess tas ut av køen, kjører en liten stund (en *timeslice*, typisk 10-100ms), og legges så tilbake i køen igjen
  - Med rask CPU og korte timeslices ser det ut som om alle jobbene kjører samtidig
  - Men: Korte jobber kan ta urimelig lang tid på grunn av mye venting hvis det er mange aktive prosesser

Timesharing med en prioritetskø

- Operativsystemer inneholder en *scheduler* som hele tiden velger ut neste prosess som skal få kjøre i CPU
- Scheduleren bruker ofte en *prioritetskø* med prosesser:
  - Prosesser med høy prioritet får mest CPU-tid
  - Scheduler kan f.eks. senke prioriteten til jobber som allerede har kjørt lenge, slik at nye og korte jobber velges oftere og dermed utføres raskere

Operasjoner på prioritetskø

Vi må (minst) kunne utføre følgende to operasjoner på en prioritetskø:

`settInn(x, p)`

Setter inn nytt element `x` med gitt prioritet `p`

`fjernMin()`

Fjerner elementet med laveste verdi (eller høyeste prioritet) fra prioritetskøen.

Mulige implementasjoner av prioritetskø

		`settInn`	`fjernMin`
Lenket liste	Setter inn først i liste. Må gå gjennom hele listen for å finne min.	$O(1)$	$O(n)$
Sortert lenket liste	Må sette inn på riktig plass i liste. Tar ut første element.	$O(n)$	$O(1)$
Binært søketre	Kapittel 11 i læreboka	$O(\log n)$	$O(\log n)$
Binær heap	Kapittel 12 i læreboka	$O(\log n)$	$O(\log n)$