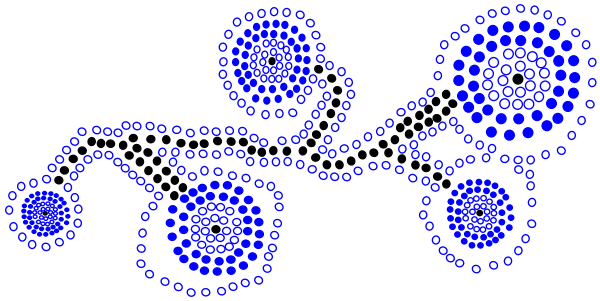
PRIORITETSKØ

- Applikasjon: aksjehandel
- ADT Prioritetskø (eng: Priority Queue - PQ)
- Implementering av PQ med sekvenser
- Sortering vha PQ
- Mer om sortering



Aksjehandel

- Vi ser på en aksje som kjøpes og selges på børsen.
- Investorer gir ***ordre*** som består av tre deler (***aksjon***, ***pris***, ***antall***), hvor ***aksjon*** er ***kjøp*** eller ***salg***, ***pris*** er laveste prisen han/hun er villig til å betale/motta, og ***antall*** er antall aksjer involvert.
- Ekvilibrium eksisterer om alle kjøp-ordre har lavere pris enn alle salgs-ordre.
- Spekulanter kan be om å få vite høyeste kjøpspris og laveste salgspris.
- Et ***salg*** forekommer om en ny ordre kan imøtekommes av en eller flere eksisterende ordre, noe som fører til fjerning av endel ordre.
- En ordre kan når som helst ***kanselleres.***

Datastruktur for aksjehandel

- For hver aksje har vi to datastrukturer, en for kjøpsordre, og en for salgsordre.
- Operasjoner som kan utføres:

Aksjon	***Salg Datastruktur***	***Kjøp Datastruktur***
ny ordre	***insert***(price, size)	***insert***(price, size)
Finn lav/høy	***min***()	***max***()
salg	***removeMin***()	***removeMax***()
kanseller	***remove***(order)	***remove***(order)

- Disse datastrukturene heter ***prioritetskøer***.
- På NASDAQ utføres daglig ca 1 milliard salg.

Nøkler og Totalorden-relasjonen

- En Prioritetskø (eng: Priority Queue - PQ) rangerer elementer vha en ***nøkkel*** som har en ***totalorden*** relasjon.
- Nøkler:
  - Hvert element har en nøkkel
  - Flere element kan ha samme nøkkel
- Totalorden-Relasjon
  - Angitt ved $\leq$
  - Total: for ethvert par k_1,k_2 så har vi enten $k_1 \leq k_2$ eller $k_2 \leq k_1$
  - **Refleksiv:** $k \leq k$
  - **Antisymmetrisk:** om $k_1 \leq k_2$ og $k_2 \leq k_1$, så har vi $k_1 = k_2$
  - **Transitiv:** om $k_1 \leq k_2$ og $k_2 \leq k_3$, så har vi $k_1 \leq k_3$
- En Prioritetskø kan utføre følgende operasjoner på nøkkel-element par (k,e):
  - min()
  - insertItem(k, e)
  - removeMin()

Sortering vha PQ

- En Prioritetskø *P* kan brukes til å sortere en sekvens S som følger:
- Fase 1: sett inn elementene fra S i P vha en rekke insertItem(*e*, *e*) operasjoner
- Fase 2: fjern elementene fra P i stigende rekkefølge og sett dem tilbake i S vha en rekke removeMin() operasjoner

Algorithm PriorityQueueSort(*S*, *P*):
Input: A sequence *S* storing *n* elements, on which a total order relation is defined, and a Priority Queue *P* that compares keys with the same relation
Output: The Sequence *S* sorted by the total order relation

while !*S*.isEmpty() **do**
e ← *S*.removeFirst()
P.insertItem(*e*, *e*)
while *P* is not empty **do**
e ← *P*.removeMin()
S.insertLast(*e*)

ADT Proritetskø

- En PQ *P* har følgende operasjoner:

- size():
  Returner antall elementer i *P*
- isEmpty():
  Test om *P* er tom
- insertItem(*k*,*e*):
  Sett inn nytt element *e* med nøkkel *k* i *P*
- minElement():
  Returner (uten å fjerne) et element i *P* som har den minste nøkkel; feilmelding om *P* tom.
- minKey():
  Returner minste nøkkel i *P*; feilmelding om *P* er tom
- removeMin():
  Returner og fjern et element i P som har den minste nøkkel; feilmelding om P tom.

Comparator

- En generisk PQ bruker **comparator** objekter.
- Et comparator objekt sammenlikner to nøkler, men er ekstern til disse.
- Når en PQ skal sammenlikne nøkler, brukes comparator objektet.
- Dermed kan en generisk PQ lagre, og sammenlikne, hvilke objekter som helst.
- ADT comparator har operasjoner:
  - isLessThan(*a*, *b*)
  - isLessThanOrEqualTo(*a*,*b*)
  - isEqualTo(*a*, *b*)
  - isGreaterThan(*a*,*b*)
  - isGreaterThanOrEqualTo(*a*,*b*)
  - isComparable(*a*)

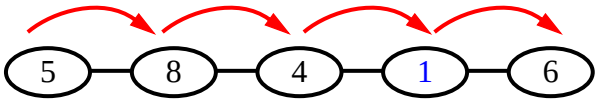
Implementering av PQ vha usortert sekvens S

- Elementene i *S* består av en nøkkel *k*, og et element *e*.
- For PQ-operasjon insertItem() bruker vi Sekvens-operasjon insertLast() . Dette tar ***O***(1) tid.

- Merk at elementene blir lagret usortert.

Implementering av PQ vha usortert sekvens S

- Siden elementene er lagret usortert må PQ-operasjonene minElement(), minKey(), og removeMin() i verste-fall *undersøke alle elementene* i *S*. Verste-fall tidskompleksitet for disse operasjonene blir derfor $O(n)$, om vi har n elementer.



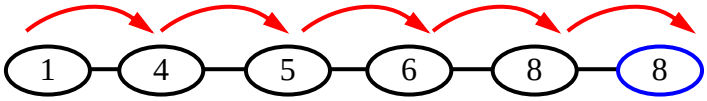
-

insertItem	***O*(1)**
minKey, minElement	***O*(*n*)**
removeMin	***O*(*n*)**

Implementering av PQ vha Sortert sekvens S

- La oss bruke en sekvens *S*, og lagre elementene i stigende nøkkelrekkefølge.
- minElement(), minKey(), og removeMin() tar ***O***(1) tid

- PQ-operasjon insertItem() må nå ***i verste-fall,*** gå gjennom alle elementer i P. Dermed blir insertItem() en $O(n)$ operasjon.



-

insertItem	***O*(*n*)**
minKey, minElement	***O*(1)**
removeMin	***O*(1)**

Implementering av PQ vha sortert sekvens S

```
public class SequenceSimplePriorityQueue
implements SimplePriorityQueue {
  //Implementation of a priority queue
  using a sorted sequence
 protected Sequence seq = new NodeSequence();
 protected Comparator comp;

// auxiliary methods
 protected Object key (Position pos) {
    return ((Item)pos.element()).key();
  } // note casting here

 protected Object element (Position pos) {
    return ((Item)pos.element()).element();
  } // casting here too

  // methods of the SimplePriorityQueue ADT
 public SequenceSimplePriorityQueue (Comparator c) {
    comp = c; }
 public int size () {return seq.size(); }
```

...Continued on next page..

Implementering av PQ vha sortert sekvens S

```
public void insertItem (Object k, Object e) throws
InvalidKeyException {
   if (!comp.isComparable(k)) {
      throw new InvalidKeyException("The key is not valid");
   }
   else {
      if (seq.isEmpty()) {
         //if the sequence is empty, this is the
         seq.insertFirst(new Item(k,e));//first item
      }
      else { //check if it fits right at the end
         if (comp.isGreaterThan(k,key(seq.last()))) {
            seq.insertAfter(seq.last(),new Item(k,e));
         }
         else {
            //we have to find the right place for k.
             Position curr = seq.first();
            while (comp.isGreaterThan(k,key(curr))) {
               curr = seq.after(curr);
            }
            seq.insertBefore(curr,new Item(k,e));
         }
      }
   }
```

...Continued...

Implementering av PQ vha sortert sekvens S

```
public Object minElement () throws
EmptyContainerException {
if (seq.isEmpty()) {
    throw new EmptyContainerException("The priority
    queue is empty");
}
else {
    return element(seq.first());
}


public boolean isEmpty () {
    return seq.isEmpty();
}
}
```


SeleksjonsSortering

- Seleksjonssortering får vi fra PQ-Sortering når PQ er implementert som ***usortert sekvens***.
- **Fase 1**: å sette et enkelt element inn i *P* tar ***O***(1) tid. Med n elementer totalt tar fase 1 dermed ***O***(*n*) tid.
- **Fase 2**, å fjerne et element fra *P* tar tid proporsjonalt med antall gjenværende elementer i *P*.

	Sekvens *S*	Prioritetskø *P*
Input	(7, 4, 8, 2, 5, 3, 9)	()
Fase 1:		
(a)	(4, 8, 2, 5, 3, 9)	(7)
(b)	(8, 2, 5, 3, 9)	(7, 4)
...	...	...
(g)	()	(7, 4, 8, 2, 5, 3, ,9)
Fase 2:		
(a)	(2)	(7, 4, 8, 5, 3, 9)
(b)	(2, 3)	(7, 4, 8, 5, 9)
(c)	(2, 3, 4)	(7, 8, 5, 9)
(d)	(2, 3, 4, 5)	(7, 8, 9)
(e)	(2, 3, 4, 5, 7)	(8, 9)
(f)	(2, 3, 4, 5, 7, 8)	(9)
(g)	(2, 3, 4, 5, 7, 8, 9)	()

Seleksjonssortering

- Vi får en flaskehals i fase 2. Den første removeMin()-operasjon tar tid ***O***(*n*), den andre ***O***(*n*–1), etc. inntil den siste som tar ***O***(1) tid.
- Total tid for fase 2 blir:

$$O(n + (n-1) + \ldots + 2 + 1) \equiv O\left(\sum_{i=1}^{n} i\right)$$

- Dette kjenner vi fra før:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

- Tidskompleksitet for fase 2 blir dermed ***O***(n^2). Tidskompleskistet for hele sorteringen blir dermed også ***O***(n^2).

InsettingsSortering (eng: insertion sort)

- Seleksjonssortering får vi fra PQ-Sortering når PQ er implementert som ***sortert sekvens***.

	Sekvens *S*	Prioritetskø *P*
Input	(7, 4, 8, 2, 5, 3, 9)	()
Fase 1:		
(a)	(4, 8, 2, 5, 3, 9)	(7)
(b)	(8, 2, 5, 3, 9)	(4, 7)
(c)	(2, 5, 3, 9)	(4, 7, 8)
(d)	(5, 3, 9)	(2, 4, 7, 8)
(e)	(3, 9)	(2, 4, 5, 7, 8)
(f)	(9)	(2, 3, 4, 5, 7, 8)
(g)	()	(2, 3, 4, 5, 7, 8, 9)
Fase 2:		
(a)	(2)	(3, 4, 5, 7, 8, 9)
(b)	(2, 3)	(4, 5, 7, 8, 9)
...	...	...
(g)	(2, 3, 4, 5, 7, 8, 9)	()

Insettingssortering

- Fase 2 forbedres til $\boldsymbol{O}(n)$.
- Fase 1 blir den nye flaskehalsen. Den første insertItem tar $\boldsymbol{O}(1)$ tid, den andre tar $\boldsymbol{O}(2)$ osv, inntil den siste som tar $\boldsymbol{O}(n)$ tid, i verste-fall. Samme analyse som isted gir totaltid $\boldsymbol{O}(n^2)$.
- Seleksjonssortering og insettingssortering er dermed begge $\boldsymbol{O}(n^2)$ sorteringsalgoritmer.
- Seleksjonssortering vil ***alltid*** utføre et antall operasjoner proporsjonalt med n^2 , uavhengig av input, dvs beste-fall er også $\boldsymbol{O}(n^2)$.
- Kjøretiden for innsettingssortering avhenger av input-sekvensen, i beste-fall er den $\boldsymbol{O}(n)$.
- Ingen av disse er noen god sorteringsalgoritme, om det er mange elementer som skal sorteres.
- Vi skal snart få se en bedre implementasjon av PQ, som vil gi PQ-sorteringsalgoritme med tidskompleksitet $\boldsymbol{O}(n\log n)$....

Litt mer om Sortering

- Sortering er vesentlig fordi ethvert ***søk i database*** krever at dataene er lagret sortert.
- Det antaes at 20% av all maskintid over hele verden brukes til sortering.
- De enkleste sorteringsalgoritmene er ikke de raskeste.
- Vi har nettopp sett to $\boldsymbol{O}(n^2)$ sorteringsalgoritmer, som er ubrukelige for store verdier av n.
- Sammenlikning av nøkler: *brukes hele nøkkelen, eller kun deler av nøkkelen, til sammenlikning?*
- Minnebruk: ***in-place*** *sortering innebærer at vi ikke bruker datastruktur utenom den som alt lagrer dataene ved input.*
- Stabilitet: *en* ***stabil*** *sorteringsalgoritme opprettholder input-rekkefølgen til elementer som har samme nøkkel*