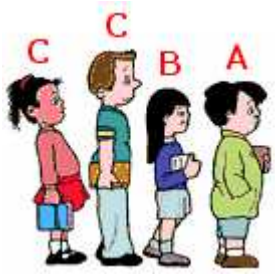
Algoritmer og datastrukturer
Kapittel 4 – Delkapittel 4.4

4.4 En prioritetskø

4.4.1 Grensesnittet PrioritetsKø



En søkerkø

En *prioritetskø* (engelsk: priority queue) er en kø der de som «står» i køen har en prioritet. Da er regelen at den som har «best» eller «høyest» prioritet står først for tur til å bli tatt ut. En konsekvens er at den som har «lav» prioritet kan oppleve å måtte vente lenge.

Vi skal bruke prioritetskøer flere ganger og da som et verktøy i bestemte algoritmer. Men også i «dagliglivet» er det mulig å finne eksempler på prioritetskøer:

En søkerkø: Hvis en student som gjør seg ferdig med en bachelorutdanning, ønsker å søke på en masterutdanning, må han/hun normalt ha minst C som gjennomsnittskarakter. Hvis det er flere søkere enn plasser, vil søkerne normalt bli rangert etter karakterer. Det kan betyr at alle som har A i gjennomsnitt kommer foran de som har B, osv. Med andre ord er her «best» prioritet lik best gjennomsnittskarakter.

En sykehuskø: Behandlingsrekkefølgen på et sykehus kan kanskje ses på som en slik kø. Der er det vel i mange tilfeller slik at man får prioritet etter hvor syk man er. Den som er sykest har «høyest» prioritet. Det høres vel ikke urimelig ut. Risikoen er at noen aldri blir behandlet fordi de har for lav prioritet - de blir hele tiden passert av andre med høyere prioritet.

Kølapper: På mange steder, f.eks. på postkontorer, er det innført et kølappsystem. Da er det den med lavest nummer som står for tur til å bli betjent. Men dette er ikke en prioritetskø, men en vanlig kø fordi en nyankommet trekker alltid et kønummer som kommer etter det som sist ble trukket. Med andre ord kommer alltid en ny kunde bakerst i køen.

Vi trenger minst de samme metodene i en `prioritetskø` som i en vanlig kø, dvs. *`leggInn`*, *`kikk`*, *`taUt`*, *`antall`*, *`tom`* og *`nullstill`*. Forskjellen er at *`taUt`* nå skal ta ut den med «best» eller «høyest» prioritet. Hva som gir «best» eller «høyest» prioritet styrer vi ved hjelp av en komparator. Da det kanskje naturlig at den som er minst med hensyn på ordningen som komparatoren definerer, somhar best/høyest prioritet. Slik er det f.eks. gjort i prioritetskøen som følger med Java (klassen `PriorityQueue` i java.util).

Det kan ofte hende at to eller flere elementer har samme prioritet. Hvem av dem skal da tas ut først? I mange anvendelser spiller dette ingen rolle og dermed kan man lage mer effektive prioritetskøer. I noen tilfeller er det viktig å kunne ha en bestemt rekkefølge på de som har samme prioritet. Et krav kunne f.eks. være at de som har samme prioritet ordnes som i en vanlig kø, dvs. den som har ventet lengst står først for tur til å bli tatt ut. Det kan man få til f.eks. ved å innføre et «kønummer» i tillegg til en prioritet. Den med lavest «kønummer» blant dem med samme prioritet skal da tas ut først.

```
public interface PrioritetsKø<T>          // Java: PriorityQueue
{
  public void leggInn(T verdi);           // Java: add/offer
  public T kikk();                        // Java: element/peek
  public T taUt();                        // Java: remove/poll
  public boolean taUt(T verdi);           // Java: remove
  public int antall();                    // Java: size
  public boolean tom();                   // Java: isEmpty
  public void nullstill();                // Java: clear
}
```

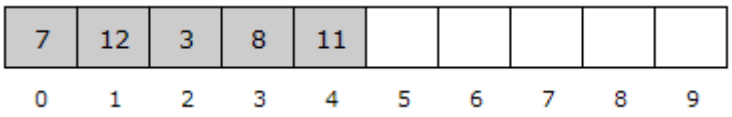
Programkode 4.4.1 a)

- Det er ingen spesielle krav til metoden *leggInn* bortsett fra at null-verdier er ulovlige.
- Metoden *kikk* skal returnere (uten å fjerne) objektet med «best» prioritet. Er det flere med «best» prioritet, er det ingen bestemte krav til hvem av dem som skal returneres.
- Metoden *taUt* skal virke som *kikk*, men i tillegg fjerne objektet fra køen.
- Metoden *taUt(T verdi)* skal fjerne objektet *verdi* fra køen.
- Metoden *antall* skal returnere antallet i køen og dermed 0 hvis køen er tom.
- Metoden *tom* skal returnere sann (true) hvis køen er tom og usann (false) ellers.
- Metoden *nullstill* skal «tømme» køen.

PriorityQueue i Java er en klasse og ikke et grensesnitt. Det kommer nok av at det ikke er aktuelt å ha flere implementasjoner siden det er én som normalt alltid er best. Dette i motsetning til f.eks. grensesnittene Queue og Deque. De kan implementeres på flere aktuelle måter. Vi lager her noen enkle, men lærerike implementasjoner av vårt grensesnitt. Den beste tar vi etter at vi har gått dypere inn i trestrukturer. Se *Avsnitt* 5.3.

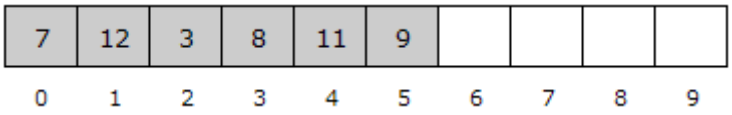
4.4.2 En prioritetskø ved hjelp av en usortert tabell

Prioritetskøen kan ha en usortert dynamisk tabell som intern datastruktur. Vi bestemmer at det objektet som en komparator sier er minst, er det som har best/høyest prioritet. I flg. eksempel inneholder køen heltall og det minste heltallet har dermed høyest prioritet:



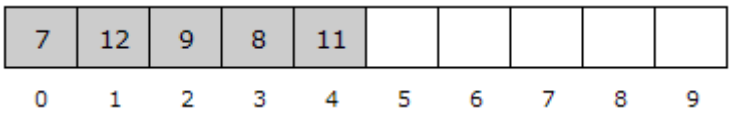
Figur 4.4.2 a) : En prioritetskø med 5 elementer usortert

I *Figur* 4.4.2 *a)* består selve prioritetskøen av den grå delen av tabellen. Som vi ser er den usortert. Når tabellen er usortert kan nye verdier legges bakerst. Innlegging får derfor konstant orden. Hvis f.eks. tallet 9 legges inn, blir dette resultatet:



Figur 4.4.2 b) : Prioritetskøen etter at 9 er lagt inn

Men det å finne den som har best/høyest prioritet, dvs. det minste tallet, krever at vi må lete gjennom hele tabellen. Til det kan vi f.eks. bruke en metode som finner posisjonen til den minste verdien i et tabellintervall. Det betyr at det å ta ut en verdi får lineær orden. Hvis den verdien som her er tallet 3, tas ut av køen, tetter vi igjen «hullet» ved at det bakerste tallet (tallet 9) flyttes dit. Da får vi dette resultatet:



Figur 4.4.2 c) : Prioritetskøen etter at 3 er tatt ut

Det er enkelt å lage en implementasjon som bruker en usortert tabell som idé:

```
public class UsortertTabellPrioritetsKø<T> implements PrioritetsKø<T>
{
  private T[] a;                       // en usortert tabell
  private int antall;                  // antall verdier i køen
  private Comparator<? super T> c;     // en komparator

  // her skal konstruktører og metoder komme
}
```

Programkode 4.4.2 a)

Vi trenger i hvert fall to konstruktører - en som har en tabellstørrelse og en komparator som parametere, og en som kun har en komparator:

```
public UsortertTabellPrioritetsKø(int størrelse, Comparator<? super T> c)
{
  a = (T[])new Object[størrelse];   // tabellens startstørrelse
  antall = 0;
  this.c = c;
}

public UsortertTabellPrioritetsKø(Comparator<? super T> c)
{
  this(8,c);  // bruker 8 som startstørrelse
}
```

Programkode 4.4.2 b)

Hvis datatypen *T* er en subtype til `Comparable<T>`, kan vi opprette en «naturlig» prioritetskø ved hjelp av en *konstuksjonsmetode*, dvs. ved hjelp av en statisk metode som returnerer en instans av klassen. Da er det datatypens «naturlige» ordning som brukes:

```
public static <T extends Comparable<? super T>> PrioritetsKø<T> naturligOrdenKø()
{
  return new UsortertTabellPrioritetsKø<>(Comparator.naturalOrder());
}
```

Programkode 4.4.2 c)

leggInn legger nye elementer bakerst, men hvis tabellen er full, må den «utvides»:

```
public void leggInn(T verdi)
{
  if (verdi == null) throw new IllegalArgumentException("Nullverdi!");

  if (antall == a.length) a = Arrays.copyOf(a,2*antall+1);  // utvider

  a[antall++] = verdi;          // verdi legges inn bakesrt
}
```

Programkode 4.4.2 d)

I metodene *kikk* og *taUt* må vi først sjekke om køen er tom og deretter, hvis køen ikke er tom, lete etter verdien med best/høyest prioritet, dvs. den minste verdien. Letingen kan vi gjøre i en egen metode:

```
public int antall()
{
  return antall;
}

public boolean tom()
{
  return antall == 0;
}

private int min()  // finner posisjonen til den minste
{
  int m = 0;
  T minverdi = a[0];

  for (int i = 0; i < antall; i++)
    if (c.compare(a[i],minverdi) < 0)
    {
      m = i;
      minverdi = a[i];
    }

  return m;        // returnerer posisjonen til den minste
}
```

Programkode 4.4.2 e)

Metodene *kikk* og *taUt* blir da slik:

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

  return a[min()];     // returnerer den minste
}

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

  int m = min();       // posisjonen til den minste
  T verdi = a[m];      // tar vare på den minste verdien

  antall--;            // reduserer antallet
  a[m] = a[antall];    // tetter igjen ved å flytte den siste verdien

  a[antall] = null;    // nuller for resirkulering
  return verdi;        // returnerer den minste
}
```

Programkode 4.4.2 f)

Nå mangler kun *nullstill* av metodene fra grenesnittet PrioritetsKø:

```
public void nullstill()
{
  while (antall > 0) a[--antall] = null;
}
```

Programkode 4.4.2 g)

Filen UsortertTabellPrioritetsKø.html inneholder hele klassen som vi nettopp har laget, med både konstruktører og metoder.

Metoden *leggInn* legger et nytt element bakerst. I *taUt*-metoden (og i *kikk*-metoden) må det letes etter den minste verdien. Hvis køen har *n* elementer, trengs *n* - 1 sammenligninger for å finne den. Det betyr at den er av orden *n* eller av *lineær* orden.

UsortertTabellPrioritetsKø		
Metode	Operasjoner	Orden
leggInn	Et nytt element legges alltid bakerst.	*konstant*
taUt	Hvis det er *n* elementer, trengs *n* - 1 sammenligninger for å finne den minste.	*lineær*

Flg. eksempel viser hvordan klassen UsortertTabellPrioritetsKø kan brukes:

```
char[] c = "VMUSXQCJKOATZPLDHIRBNFEGYW".toCharArray();

PrioritetsKø<Character> kø
  = UsortertTabellPrioritetsKø.naturligOrdenKø();

for (int i = 0; i < c.length; i++) kø.leggInn(c[i]);

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

// Utskrift:  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

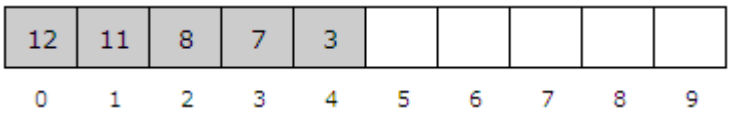
Programkode 4.4.2 h)

Oppgaver til Avsnitt 4.4.2

1. Hva skjer med like verdier i UsortertTabellPrioritetsKø? Kommer de ut i samme rekkefølge som de ble lagt inn? Eller er det uforutsigbart hvilke rekkefølge de kommer i?
2. Metoden *public boolean taUt(T verdi)* mangler kode. Lag kode for den. Hvis verdi ikke ligger i køen, skal den returnere false og true ellers.
3. Lag klassen public class UsortertLenketPrioritetsKø<T>. Klassen skal implementere grensesnittet PrioritetsKø<T> og bruke en usortert enkeltlenket pekerkjede som intern datastruktur. Da kan *leggInn* legge verdien forrest. Men *kikk* og *taUt* må lete etter den største verdien.

4.4.3 En prioritetskø ved hjelp av en sortert tabell

I *Avsnitt* 4.4.2 brukte vi en usortert tabell som intern datastruktur for en prioritetskø. Vi kan isteden bruke er sortert tabell:



Figur 4.4.3 a) : En prioritetskø med 5 elementer sortert

I *Figur* 4.4.3 *a)* består prioritetskøen av den grå delen av tabellen og er, som vi ser, sortert avtagende. Det betyr at den minste verdien ligger bakerst. I metodene *kikk* og *taUt* er det bare å hente denne verdien og til det trengs én operasjon og dermed konstant orden. I metoden *leggInn* blir det derimot mer arbeid. Vi må lete oss frem til rett sortert plass og samtidig rydde plass til den nye verdien. Hvis køen har n verdier trengs i gjennomsnitt ca. $n/2$ sammenligninger og $n/2$ forflyttninger. Antall sammenligninger kan imidlertid reduseres hvis vi bruker binærsøk, men det vil fortsatt bli $n/2$ forflyttninger. Det betyr at *leggInn*-metoden blir av orden n eller av lineær orden.

SortertTabellPrioritetsKø		
Metode	Operasjoner	Orden
leggInn	Det må letes etter rett sortert plass og de til høyre må forskyves.	*lineær*
taUt	Den minste ligger alltid bakerst.	*konstant*

```
public class SortertTabellPrioritetsKø<T> implements PrioritetsKø<T>
{
  private T[] a;                        // en usortert tabell
  private int antall;                   // antall verdier i køen
  private Comparator<? super T> c;      // en komparator

  public SortertTabellPrioritetsKø(int størrelse, Comparator<? super T> c)
  {
    a = (T[])new Object[størrelse];   // tabellens startstørrelse
    antall = 0;
    this.c = c;
  }

  public SortertTabellPrioritetsKø(Comparator<? super T> c)
  {
    this(8,c);  // bruker 8 som startstørrelse
  }

  public static <T extends Comparable<? super T>> PrioritetsKø<T> naturligOrdenKø()
  {
    return new SortertTabellPrioritetsKø<>(Comparator.naturalOrder());
  }

  // De øvrige metodene skal inn her.

}
```

Programkode* 4.4.3 *a)

I metoden *leggInn* bruker vi samme teknikk som i *innsettingssortering*:

```
public void leggInn(T verdi)
{
  if (antall == a.length) a = Arrays.copyOf(a, 2*antall+1);

  int i = antall - 1;   //  vi sammenligner og flytter
  for (; i >= 0 && c.compare(verdi,a[i]) > 0; i--) a[i+1] = a[i];
  a[i+1] = verdi;
  antall++;
}

public int antall()
{
  return antall;
}

public boolean tom()
{
  return antall == 0;
}

public T kikk()
{
  if (tom()) throw new NoSuchElementException("Køen er tom!");
  return a[antall-1];
}

public T taUt()
{
  if (antall == 0) throw new NoSuchElementException("Køen er tom!");

  T minverdi = a[--antall];      // tar vare på den bakerste
  a[antall] = null;              // klargjør for resirkulering
  return minverdi;               // returnerer den største
}

public void nullstill()
{
  while (antall > 0) a[--antall] = null;
}
```

Programkode 4.4.3 b)

Filen `SortertTabellPrioritetsKø.html` inneholder hele klassen med konstruktører og metoder.

Oppgaver til Avsnitt 4.4.3

1. Kjør *Programkode 4.4.2 h)* med `SortertTabellPrioritetsKø`.
2. Hva skjer med like verdier i `SortertTabellPrioritetsKø`? Tas de ut i samme rekkefølge som de ble lagt inn? Eller er uttaksrekkefølgen uforutsigbar? Kan du eventuelt kode det om slik at like verdier tas ut i samme rekkefølge som de ble lagt inn?
3. Metoden *`public boolean taUt(T verdi)`* mangler kode. Lag kode for den. Hvis `verdi` ikke ligger i køen, skal den returnere `false` og `true` ellers.
4. Lag klassen `public class SortertLenketPrioritetsKø<T>`.

4.4.4 Sortering ved hjelp av en prioritetskø

I en prioritetskø tas verdiene ut etter prioritet, dvs. at ved hvert uttak er det den med best eller høyest prioritet som velges. Det er verdienes ordning som bestemmer prioriteten. Den som etter ordningen er «minst», har best eller høyest prioritet.

Dette betyr at hvis vi legger inn en samling verdier i en prioriteskø, får vi verdiene ut i stigende sortert rekkefølge. I *Programkode* 4.4.2 *h)* ble bokstaver, i usortert rekkefølge, lagt inn og utskriften viser at de kommer i sortert rekkefølge.

Bruker vi en `UsortertTabellPrioritetsKø` vil taUt-metoden hver gang finne den minste i tabellen, dvs. at dette er samme teknikk som i `utvalgssortering`. Hvis vi isteden bruker den andre prioritetskøen, dvs. en `SortertTabellPrioritetsKø`, vil leggInn-metoden hver gang legge nye verdier på rett sortert plass i tabellen, mens taUt-metoden tar ut den bakerste. Med andre ord er dette samme teknikk som i *`innsettingssortering`*.

Som nevnt til slutt i *Avsnitt* 4.4.1 finnes det en mye «smartere» teknikk for en prioriteskø enn det å bruke en usortert eller en sortert tabell. Det går ut på å bruke en heap, dvs. et komplett minimumstre - se *Avsnitt* 5.3.3. Ideen der kan også brukes til sortering og da kalles det *`heapsortering`*.

4.4.5 PriorityQueue i java.util

Klassen `PriorityQueue` er deklarert slik:

```
public class PriorityQueue<T> extends AbstractQueue<T>
  implements java.io.Serializable
```

Denne klassen bruker en minimumsheap (et komplett minimumstre) som intern datastruktur. Bruk av en slik datastruktur for en prioritetskø tas opp i *Delkapittel* 5.3. De metodene som svarer til våre metoder *`leggInn`*, *`kikk`* og *`taUt`* er *`add/offer`*, *`element/peek`* og *`remove/poll`*.

Her er et eksempel på bruk av klassen `PriorityQueue`.

```
Comparator<Integer> c = Comparator.naturalOrder();       // en komparator

PriorityQueue<Integer> prkø = new PriorityQueue<>(5,c);  // plass til 5

int[] a = {3,5,2,4,1};
for (int k : a) prkø.offer(k);   // legger inn

while (!prkø.isEmpty())
{
  System.out.print(prkø.poll() + " ");   // tar ut
}

// Utskrift: 1 2 3 4 5
```