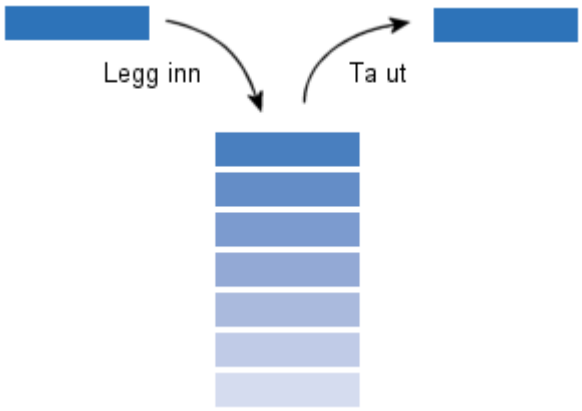
4.1 En stakk

4.1.1 Hva er en stakk?

Ordet *stakk* (eng: stack, oldnordisk: stakkr) brukes i norsk - for eksempel i ordet høystakk. Det engelske ordet *stack* har skandinavisk opprinnelse. Det fulgte med da vikingene invaderte de britiske øyene for over 1000 år siden. Ordboken «Webster's Dictionary» setter opp «*A large, usually conical, circular, or rectangular pile of hay, straw, or like*» som første betydning av ordet. Symbolet < *Scand* forteller at det opprinnelig kommer fra Skandinavia.



Figur 4.1.1 a) : En stakk (eller en stabel)

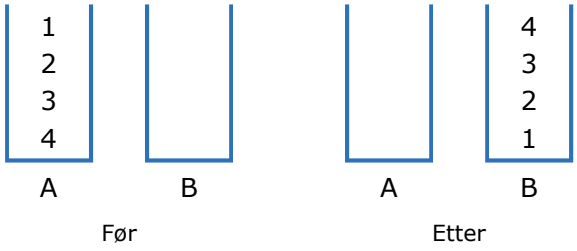
Begrepet høystakk dekker ikke begrepet stakk slik det brukes i databehandling. Ordet *stabel* hadde vært bedre. Ta f.eks. en tallerkenstabel i kjøkkenskapet. En nyvasket tallerken legges normalt øverst og vi tar normalt den øverste når vi trenger en ny tallerken. Med andre ord er en tallerkenstabel en lagringsmåte der siste tallerken som ble lagt inn er den første som vil bli tatt ut. *Figur* 4.1.1 *a)* til venstre viser prinspippet.

Vi velger likevel ordet *stakk* som navn på dette begrepet siden det er så innarbeidet i databehandling.

Definisjon *En stakk er en datastruktur der det alltid er den verdien som ble lagt på sist som står for tur til å bli tatt ut. Vi kan tenke på en stakk som en stabel av verdier der en ny verdi alltid legges øverst (på toppen) og der vi alltid (så sant stakken ikke er tom) tar den øverste hver gang en verdi skal tas ut.*

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

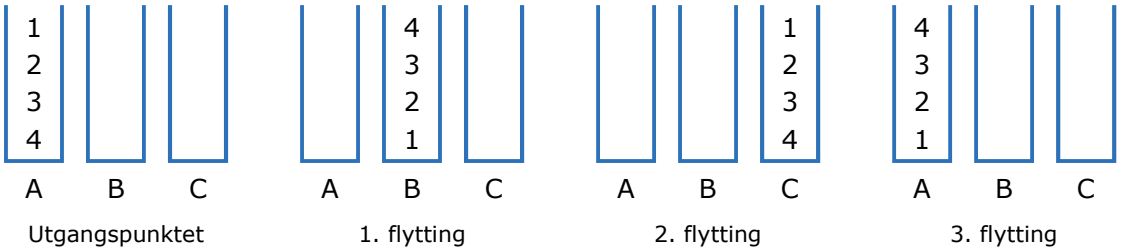
De fleste har flere tallerkener enn de bruker daglig. En ulempe med å ha dem i en stabel er at de øverste blir brukt mer enn de nederste. Da kan man eventuelt legge en og en tallerken over i en ny stabel. Dermed blir rekkefølgen snudd og de minst brukte havner på toppen.



Figur 4.1.1 b): Fra stakk A til stakk B

I *Figur* 4.1.1 *b)* har vi til venstre stakken *A* med fire nummererte verdier (f.eks. fire tallerkener). Flytter vi en og en verdi over til stakken *B* ved siden av blir rekkefølgen snudd.

Men hva skal vi gjøre hvis vi ønsker at verdiene skal havne der de opprinnelig lå, men i snudd rekkefølge? Hvis det er snakk om en tallerkenstabel, kan vi rett og slett flytte hele stabelen. Men hva hvis vi kun kan bruke de to operasjonene «legg inn» og «ta ut»? La oss ta det rent praktisk, f.eks. en tallerkenstabel. La stabelen stå på plass *A*. Se *Figur* 4.1.1 *c)* under. Så flytter vi en og en tallerken (ta ut og legg inn) til plass *B*. Deretter fra plass *B* til plass *C*. Dermed vil tallerkenene på plass *C* ha samme rekkefølge som de opprinnelig hadde på plass *A*. Men så avslutter vi ved flytte dem tilbake til plass *A*. Da har de både blitt snudd og kommet tilbake til sin opprinnelige plass.



Figur 4.1.1 c) : Verdiene på stakk *A* har blitt snudd ved hjelp av stakkene *B* og *C*

Tidligere, blant annet i forbindelse med rekursjon, tok vi opp begrepet programstakk eller kjørestakk (eng: call stack, run-time stack). Det er en stakk av samme type som den vi tar opp i dette kapittelet. Når et program utføres er det alltid det som ligger øverst på programstakken som utføres.

En stakk må ha en *leggInn*-metode og en *taUt*-metode. I tillegg bør vi kunne få vite antallet som ligger på stakken. Hvis antallet er 0 er stakken tom. Derfor trengs det egentlig ikke en egen metode som forteller oss om stakken er tom eller ikke, men det er likevel gunstig å ha det. I noen situasjoner er det ønskelig å «se på» det som ligger øverst på stakken. En *kikk*-metode som viser verdien på toppen (uten at den tas ut) oppfyller det ønsket. Dette kan oppsummeres i flg. grensesnitt:

```
public interface Stakk<T>          // eng: Stack
{
  public void leggInn(T verdi);    // eng: push
  public T kikk();                 // eng: peek
  public T taUt();                 // eng: pop
  public int antall();             // eng: size
  public boolean tom();            // eng: isEmpty
  public void nullstill();         // eng: clear

} // interface Stakk
```

Programkode* 4.1.1 *a)

Kravet er at metoden *leggInn*() skal legge *verdi* øverst på stakken. Videre skal metoden *kikk*() returnere en referanse til det øverste objektet, mens *taUt*() skal ta ut (fjerne) den øverste verdien. Metoden *tom*() skal returnere sann (true) hvis stakken er tom, og usann (false) ellers og *antall*() skal returnere antallet verdier på stakken, og dermed 0 hvis stakken er tom. Metoden *nullstill*() skal «tømme» stakken slik at den kan brukes på nytt.

Eksempel 1 La *A* og *B* være to stakker. Anta at *A* inneholder verdier og at *B* er tom. Da vil flg. kode flytte alle verdiene fra *A* over på *B*:

```
while (!A.tom()) B.leggInn(A.taUt());  // verdiene flyttes fra A til B
```

Programkode* 4.1.1 *b)

Eksempel 2 *Figur 4.1.1 c)* viser hvordan vi kan få snudd rekkefølgen til verdiene på en stakk *A* ved først å flytte dem over til en hjelpestakk *B*, så fra *B* til en hjelpestakk *C* og til slutt fra *C* tilbake til *A*. La nå *A*, *B* og *C* være tre stakker. Anta at *A* inneholder verdier og at *B* og *C* er tomme. Da vil flg. kode sørge for at stakken *A* får de samme verdiene som den opprinnelig hadde, men i omvendt rekkefølge:

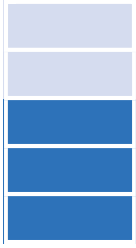
```
while (!A.tom()) B.leggInn(A.taUt());  // verdiene flyttes fra A til B
while (!B.tom()) C.leggInn(B.taUt());  // verdiene flyttes fra B til C
while (!C.tom()) A.leggInn(C.taUt());  // verdiene flyttes fra C til A
```

Programkode 4.1.1 c)

Oppgaver til Avsnitt 4.1.1

1. Utfør flg. seks operasjoner på en stakk: legg inn 5, legg inn 7, ta ut, legg inn 3, legg inn 7 og ta ut. Hvilket tall ligger nå øverst på stakken?
2. Hva ligger øverst på stakken *A* etter at flg. kode er utført:
   ```
   A.leggInn(2);
   A.leggInn(5);
   A.leggInn(1);
   A.kikk();
   A.taUt()
   ```
3. *Eksempel 2* over inneholder kode som sørger for at verdiene på en stakk *A* får omvendt rekkefølge ved at de først flyttes fra *A* til *B*, så fra *B* til *C* og til slutt tilbake til *A*. Gjør om koden slik at alle verdiene i *A* blir flyttet til *B* og at de der får samme rekkefølge som de hadde på *A*.
4. Er det mulig å snu rekkefølgen av verdiene på en stakk ved kun å bruke én hjelpestakk og én hjelpevariabel?
5. Er det mulig å flytte verdiene fra en stakk *A* til en stakk *B* slik at rekkefølgen på *B* blir som den var på *A*, ved kun å bruke én hjelpevariabel og ikke noe annet?
6. Er det mulig å få verdiene på en stakk sortert kun ved å bruke én hjelpestakk og noen enkelte hjelpevariabler?
7. Bruker vi krøllparentesene { og } i et Java-program må det være like mange venstre- som høyreparenteser og at de må komme i riktig i forhold til hverandre. F.eks. er dette ukorrekt: { . . . {. . {. .} {. . . }. Hvordan kan en stakk brukes til å evaluere Javakode slik at parentesfeil oppdages?

4.1.2 En tabellbasert stakk



Det er ganske enkelt å implementere en stakk ved hjelp av en tabell. Da tenker vi oss at tabellen står på høykant - med indeks 0 nederst og med indeksene 1, 2, 3 osv. oppover. I figuren til venstre er det en tabell med plass til fem verdier. De tre nederste radene har mørk farge. Det skal indikere at der ligger det verdier. Med andre ord er antall verdier på «stakken» lik 3. De to øverste radene har lys farge. Det indikerer ledige plasser. Øverst på stakken er dermed den øverste av de «mørke».

```
import java.util.*;

public class TabellStakk<T> implements Stakk<T>
{
  private T[] a;                      // en T-tabell
  private int antall;                 // antall verdier på stakken

  public TabellStakk()                // konstruktør - tabellengde 8
  {
    this(8);
  }

  @SuppressWarnings("unchecked")      // pga. konverteringen: Object[] -> T[]
  public TabellStakk(int lengde)      // valgfri tabellengde
  {
    if (lengde < 0)
      throw new IllegalArgumentException("Negativ tabellengde!");

    a = (T[])new Object[lengde];      // oppretter tabellen
    antall = 0;                       // stakken er tom
  }

  // de andre metodene skal inn her!

}  // class TabellStakk
```

Programkode 4.1.2 a)

Obs. Verdien til variabelen *antall* i klassen TabellStakk er antall objekter på stakken. Det betyr at det er *antall* – 1 som er indeksen eller posisjonen til det øverste objektet.

Hvis det ikke er plass i *a* når vi skal legge inn et nytt objekt, kan vi «utvide» tabellen, f.eks. til det dobbelte. Da bruker vi som vanlig metoden *copyOf*() fra klassen Arrays:

```
public void leggInn(T verdi)
{
  if (antall == a.length)
    a = Arrays.copyOf(a, antall == 0 ? 1 : 2*antall);   // dobler

  a[antall++] = verdi;
}
```

Programkode 4.1.2 b)

I metodene *kikk*() og *taUt*() må vi passe på at stakken ikke er tom. Vi kaster et unntak hvis det er tilfellet. Da kan vi enten lage vårt eget «unntak» eller vi kan benytte et som allerede finnes i Java. Kanskje NoSuchElementException fra java.util kunne passe?

```
public T kikk()
{
  if (antall == 0)       // sjekker om stakken er tom
    throw new NoSuchElementException("Stakken er tom!");

  return a[antall-1];    // returnerer den øverste verdien
}

public T taUt()
{
  if (antall == 0)       // sjekker om stakken er tom
    throw new NoSuchElementException("Stakken er tom!");

  antall--;              // reduserer antallet

  T temp = a[antall];    // tar var på det øverste objektet
  a[antall] = null;      // tilrettelegger for resirkulering

  return temp;           // returnerer den øverste verdien
}
```

Programkode 4.1.2 c)

Metodene *tom*() og *antall*() er det rett frem å lage:

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

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

Programkode 4.1.2 d)

Hvor effektivt blir dette? En innlegging starter med en sammenligning. Deretter kommer setningen `a[antall++] = verdi;` som består av en tabelloperasjon, en tilordning og en addisjon. Med andre ord utføres det alltid fire operasjoner. Men hvis tabellen er full, må den «utvides» og det er kanskje en kostbar operasjon?

En tabell «utvides» ved hjelp av metoden `copyOf` fra klassen `Arrays`. I den opprettes det først en ny tabell *b* med dobbelt så stor lengde som den gamle *a*. Deretter kopieres *a* over i *b* ved hjelp av metoden `arraycopy` fra klassen `System`.

Vi kunne dimensjonere tabellen så stor fra starten av at det aldri blir behov for noen «utvidelse». Ulempen er da at man kan komme til å reservere en masse plass i minnet som aldri blir brukt. En mer dynamisk løsning er å starte med en liten tabell og så utvide etter behov. Standardkonstruktøren i klassen bruker 8 som startlengde. Se på flg. eksempel:

```
Stakk<Integer> s = new TabellStakk<>();
for (int k = 1; k <= 1000; k++) s.leggInn(k);
```

Programkode 4.1.2 e)

I *Programkode* 4.1.2 *e)* må den interne tabellen i stakken *s* «utvides» når tallet 9 skal legges inn, deretter når 17 skal legges inn, osv. Med andre ord får vi det «verste tilfellet» når tabellen er full og inneholder 2^k verdier der *k* er et eller annet positivt heltall. Hvis en verdi til skal legges inn, vil alle verdiene bli kopiert over i en annen tabell. I dette tilfellet vil `leggInn` få orden $n = 2^k$. Men denne doblingen skjer ofte i starten, men sjeldnere etter som tabellstørrelsen øker. I gjennomsnitt får derfor likevel *leggInn*() konstant orden.

Hvis en på forhånd vet at stakken minst kommer til å få et bestemt antall verdier, kan den dimensjoneres til det antallet med en gang. De to metodene *kikk*() og *taUt*() er begge av konstant orden siden det som skjer der er helt uavhengig av stakkens størrelse.

java.util En implementasjon ved hjelp av en tabell gir nok den mest effektive stakken. Klassen Stack i javabiblioteket java.util bruker indirekte en tabell siden klassen er en subklasse av Vector. De aktuelle metodene heter *push*(), *peek*() og *pop*(). De brukes slik:

```
Stack<Character> s = new Stack<>();          // oppretter en stakk

s.push('A'); s.push('B'); s.push('C');       // legger inn tre verdier

while (!s.isEmpty()) System.out.print(s.pop() + " ");  // tar ut

// Utskrift: C B A
```

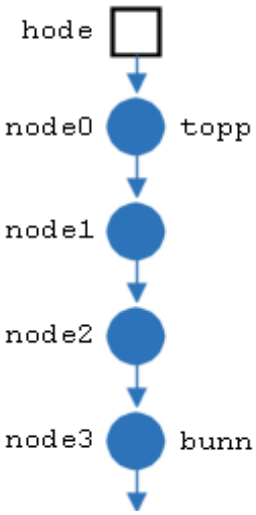
Programkode* 4.1.2 *f)

OBS. Java-ekspertene anbefaler at man bruker en av de andre «stakkene» i java.util istedenfor klassen Stack. Den har noen uheldige sider. Se den innledene teksten i Java API-en Stack. Se også *Avsnitt* 4.1.4.

Oppgaver til Avsnitt 4.1.2

1. TabellStakk<T> skal ha metoden *nullstill*() - se *Programkode* 4.1.2 *a)*. Den skal «tømme» stakken. Dvs. alle referanser skal «nulles» og antall settes til 0. Lag metoden.
2. Lag en *toString*-metode i klassen TabellStakk<T>. Hvis stakken inneholder (fra toppen og nedover) verdiene 1, 2 og 3, skal metoden returnere strengen "[1, 2, 3]".
3. Lag public static <T> void *snu*(Stakk<T> A). Den skal snu rekkefølgen på stakken *A*. Bruk to TabellStakker som hjelpestakker. Se *Figur* 4.1.1 *c)*. Parametertypen til *A* er Stakk<T>. Da er det kun metodene i grensesnitt Stakk<T> som kan brukes.
4. Lag public static <T> void *kopier*(Stakk<T> A, Stakk<T> B). Den skal kopiere innholdet av *A* over i *B*, dvs. *A* skal ha samme innhold som før etter kopieringen og *B* skal inneholde de samme verdiene i samme rekkefølge som *A*. Det antas at *B* er tom før kopieringen starter. Bruk kun én TabellStakk og en enkelt *T*-variabel som hjelpemiddel. Typen til *A* og *B* er Stakk<T>. Da er det kun metodene i grensesnitt Stakk<T> som kan brukes.
5. Som *Oppgave* 3, men én TabellStakk og en enkelt *T*-variabel som hjelpemiddel.
6. Som *Oppgave* 4, men bruk kun én enkelt hjelpevariabel av typen *T*.
7. Metoden public static <T> void *sorter*(Stakk<T> A, Comparator<? super T> c) skal sortere objektene på *A* vha. komparatoren *c*. Lag metoden. Du skal kun bruke én hjelpestakk og noen enkelte hjelpevariabler i kodingen.
8. En tekststreng kan inneholde parenteser og også nøstede parenteser. F.eks. slik "Ordet stakk (eng: stack (se ordboken)) kommer fra begrepet en høystakk". Lag metoden public static boolean *sjekkParenteser*(String s, char v, char h). Der skal *v* og *h* utgjøre et parentespar. F.eks. *v* = '(' og *h* = ')' eller *v* = '{' og *h* = '}'. Metoden skal gå gjennom tegnstrengen *s* og sjekke om det er like mange *v*-parenteser som *h*-parenteser og at de kommer i riktig i forhold til hverandre.
9. Bruk metoden i *Oppgave* 8 til å sjekke om det er syntaksfeil (parentes-feil) i Java-setningen: int n = (1 + 2) * (3) - 4) + (5 / 6);

4.1.3 Stakk ved hjelp av en pekerkjede



Figur 4.1.3 a)

En stakk kan på en enkel måte implementeres ved hjelp av en enkeltlenket liste (pekerkjede.) Hvis listen tegnes vertikalt, vil den ligne på en stakk. Den første noden i listen blir da øverst av nodene (toppen av stakken) og den siste noden blir nederst (bunnen av stakken).

I *Figur* 4.1.3 *a)* til venstre er en liste med fire noder tegnet vertikalt. Når et nytt objekt skal legges på toppen av stakken, representert ved hjelp av denne listen, må den tilhørende noden legges inn først, dvs. som en ny *node* 0. Metoden *leggInn* vil få konstant orden siden arbeidet med å legge inn noe først i en liste er uavhengig av antallet i listen.

Metoden *taUt* (og *kikk*) vil også få konstant orden siden det å fjerne den første noden (*node* 0) i en liste har konstant orden.

I en enkeltlenket liste er det vanlig å ha en halepeker (peker til den siste noden). Da vil det å legge inn et objekt bakerst i listen kunne gjøres effektivt. Men det er det ikke behov for her.

```
import java.util.*;

public class LenketStakk<T> implements Stakk<T>
{
  private static final class Node<T>       // en «nøstet» nodeklasse
  {
    private T verdi;
    private Node<T> neste;

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

  } // class Node

  private Node<T> hode;             // stakkens topp
  private int antall;               // antall på stakken

  public LenketStakk()              // konstruktør
  {
    hode = null;
    antall = 0;
  }

  // Her skal de andre metodene fra grensesnittet Stakk<T> stå

}  // class LenketStakk
```

Programkode* 4.1.3 *a)

Skal en ny verdi legges på stakken må den legges «øverst», dvs. den må legges i en ny node og den nye noden plasseres aller først i pekerkjeden. Med andre ord blir den en ny *node* 0. Se *Figur* 4.1.3 *a)*. Dette kan gjøres med kun én programsetning:

```
public void leggInn(T verdi)
{
  hode = new Node<>(verdi,hode);    // verdi legges først
  antall++;                         // antall økes med 1
}
```

Programkode 4.1.3 c)

Kode for resten av metodene tas opp i *oppgavene* nedenfor.

Hvor effektiv er denne stakkimplementasjonen? F.eks. sammenlignet med klassen *TabellStakk* i *Avsnitt* 4.1.2 ? Metoden *leggInn* er åpenbart av konstant orden. Det skjer noe arbeid i setningen: `hode = new Node<T>(verdi,hode)`, det skal opprettes en instans av nodeklassen og pekere må oppdateres. Men dette er uavhengig av hvor lang pekerkjeden er, dvs. arbeidet er konstant. Også de to andre viktige metodene (*taUt* og *kikk*) vil ha konstant orden. Se *Oppgave* 1 og 2. I flg. testprogram legges samme verdi en million ganger på stakken. Deretter fjernes de en etter en:

```
long tid = System.currentTimeMillis();

Stakk<Integer> s = new LenketStakk<>();
Integer k = 0;

for (int i = 0; i < 10000000; i++) s.leggInn(k);
while (!s.tom()) s.taUt();

tid = System.currentTimeMillis() - tid;
System.out.println(tid);
```

Programkode 4.1.3 d)

Tidsforbruket er avhengig av hvor rask maskin en bruker. På en 2.80GHz Intel Pentium med Windows XP og Java 1.7, tok det 3,3 sekunder. Spørsmålet er hva det blir hvis en isteden bruker en *TabellStakk*. Se *Oppgave* 6 og 7.

Oppgaver til Avsnitt 4.1.3

1. Lag metoden *kikk*() i klassen LenketStakk<T>. Den skal, hvis stakken ikke er tom, returnere den «øverste» verdien. Hvilken orden får metooden?
2. Lag metoden *taUt*() i klassen LenketStakk<T>. Den skal, hvis stakken ikke er tom, fjerne og returnere den «øverste» verdien. Hvilken orden får metoden?
3. Lag metodene *tom*() og *antall*() i klassen LenketStakk<T>.
4. Lag metoden *nullstill*() i klassen LenketStakk<T>. Den skal «tømme» stakken.
5. Lag en *toString*-metode i klassen LenketStakk<T>. Hvis stakken inneholder (fra toppen og nedover) verdiene 1, 2 og 3, skal metoden returnere strengen "[1, 2, 3]".
6. Sørg for at din klasse LenketStakk er kjørbar og kjør så *Programkode* 4.1.3 *d)* noen ganger. Hvilke tider får du? Gjenta dette, men bruk TabellStakk istedenfor LenketStakk
7. Bruk klassene Stack, ArrayDeque og LinkedList fra java.util i *Programkode* 4.1.3 *d)*. Da må en bytte ut *leggInn*, *tom* og *taUt* med *push*, *isEmpty* og *pop*. Hvilke tider gir det? Hvilken versjon bør en normalt bruke?

4.1.4 Stack i java.util

Java burde hatt et grensesnitt `Stack` med de aktuelle metodene. Men det er ikke mulig siden navnet `Stack` er i bruk til en klasse. Isteden er alle de aktuelle stakk-metodene lagt inn i grensesnittet `Deque` (Deque = Double Ended Queue):

```
public interface Deque<T>
{
  public void push(T t);           // legger inn øverst
  public T peek();                 // ser på den øverste
  public T pop();                  // tar ut den øverste
  public int size();               // antallet
  public boolean isEmpty();        // er det tomt?
  public void clear();             // nullstiller

  // + mange flere metoder

} // interface Deque
```

Programkode 4.1.4 a)

Vi kan derfor bruke en `ArrayDeque` (eller en `LinkedList`) i *Programkode 4.1.2 f)* istedenfor en instans av klassen `Stack` siden begge implementerer grensesnittet `Deque`:

```
Deque<Character> s = new ArrayDeque<>();  // oppretter en stakk

s.push('A'); s.push('B'); s.push('C');        // legger inn tre verdier

while (!s.isEmpty()) System.out.print(s.pop() + " ");  // tar ut

// Utskrift: C B A
```

Programkode 4.1.4 b)

Klassen `ArrayDeque` bruker en tabell, mens klassen `LinkedList` bruker en lenket liste som intern datastruktur. Her vil normalt `ArrayDeque` være det beste valget når en trenger en stakk. Se også *Oppgave 7* i forrige avsnitt. Hva blir tidene sammenlignet med `TabellStakk` og `LenketStakk`?

Hvis en `Deque`-implementasjon (f.eks. `ArrayDeque` eller `LinkedList`) brukes som en stakk, kan metodene `addFirst`, `peekFirst` og `removeFirst` brukes istedenfor `push`, `peek` og `pop`. De virker på samme måte bortsett fra en ting. Metoden `removeFirst` kaster unntaket `NoSuchElementException`, mens `pop` returnerer null hvis stakken er tom.