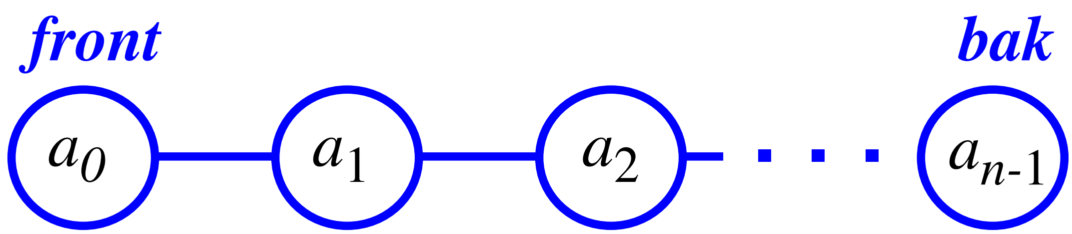
ADTer: Stabel, Kø og 2-sidig Kø

- Data strukturer, dvs konkrete implementasjoner: Tabell, lenket liste, 2-veis lenket liste.
- Tidligere har vi sett: Stabel implementert vha tabell. Idag: ADT stabel implementert ved å ‘adaptere’ ADT 2-sidig kø.
- Kø implementert vha tabell og vha lenket liste, samt ved å adaptere ADT 2-sidig kø.
- 2-sidig kø implementert vha 2-veis lenket liste.

ADT Kø (eng: queue)

- En kø følger first-in-first-out (FIFO) prinsippet.
- Elementer kan innsettes når som helst, men kun elementet som har befunnet seg lengst i køen kan taes ut.
- Elementene settes inn bakerst (eng:rear) (enqueue) og fjernes fra fronten (dequeue)

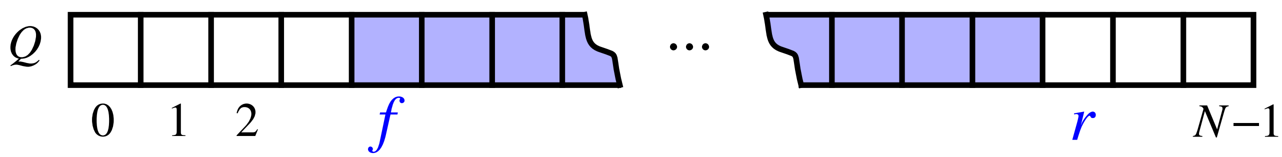


Abstrakt Data Type Kø

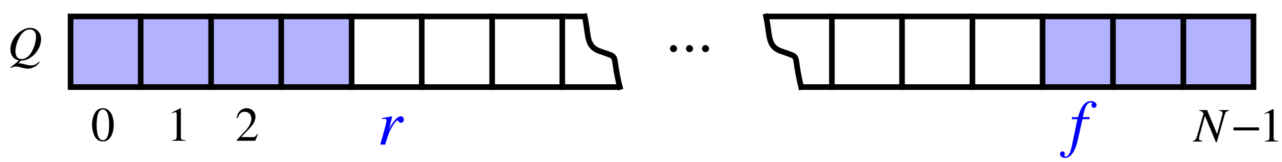
- To metoder:
  - enqueue(*o*): Insert object *o* at the rear of the queue
  - dequeue(): Remove the object from the front of the queue and return it; an error occurs if the queue is empty
- Følgende støtte-metoder kan også defineres:
  - size(): Return the number of objects in the queue
  - isEmpty(): Return a boolean value that indicates whether the queue is empty
  - front(): Return, but do not remove, the front object in the queue; an error occurs if the queue is empty

En tabell-basert kø

- Implementer en kø vha array som brukes sirkulært
- Max.størrelse N, f.eks. $N = 1{,}000$.
- Køen består av en N-element array Q og to integer variable:
  - f, index til front element
  - r, index til element rett etter det bakerste
- "normal konfigurasjon"



- "sirkulær" konfigurasjon



- Hva innebærer $f=r$?

Tabell-basert kø

- Pseudokode

Algorithm size():
  return $(N - f + r) \bmod N$

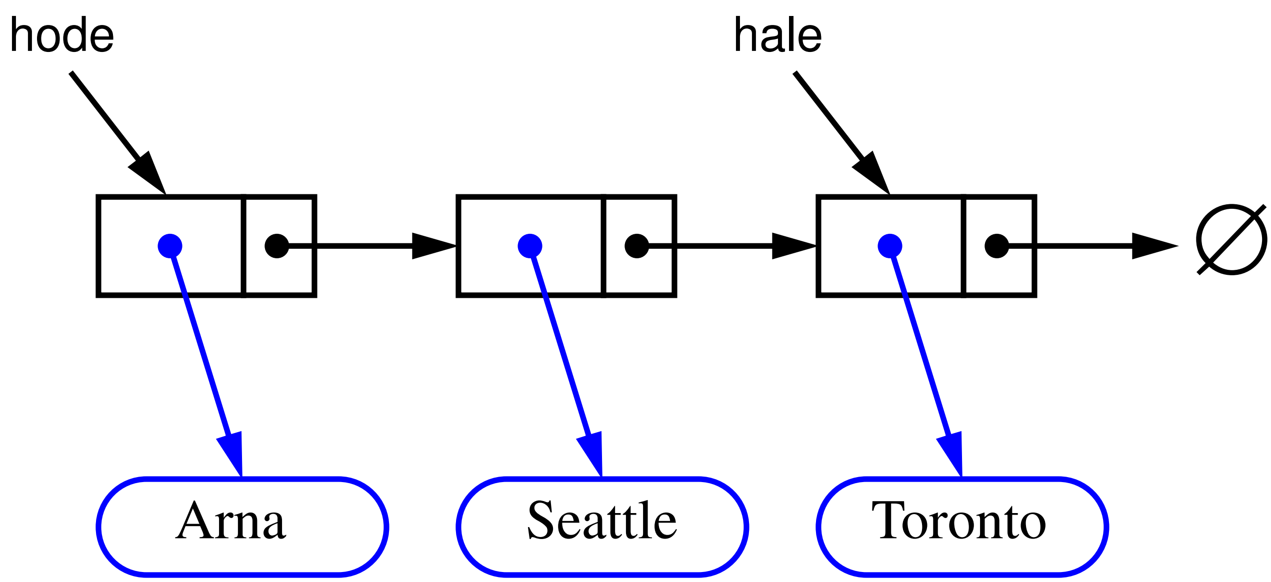
Algorithm isEmpty():
  return $(f = r)$

Algorithm front():
  if isEmpty() **then**
    throw a QueueEmptyException
  return $Q[f]$

Algorithm dequeue():
  if isEmpty() **then**
    throw a QueueEmptyException
  $temp \leftarrow Q[f]$
  $Q[f] \leftarrow$ **null**
  $f \leftarrow (f + 1) \bmod N$
  return *temp*

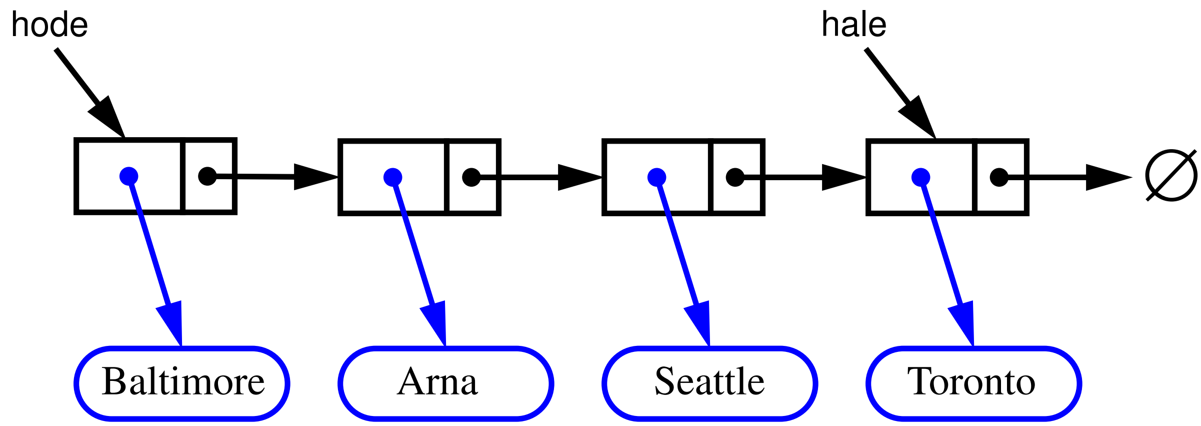
Algorithm enqueue(*o*):
  if size = $N - 1$ **then**
    throw a QueueFullException
  $Q[r] \leftarrow o$
  $r \leftarrow (r + 1) \bmod N$

Kø implementert vha lenket liste

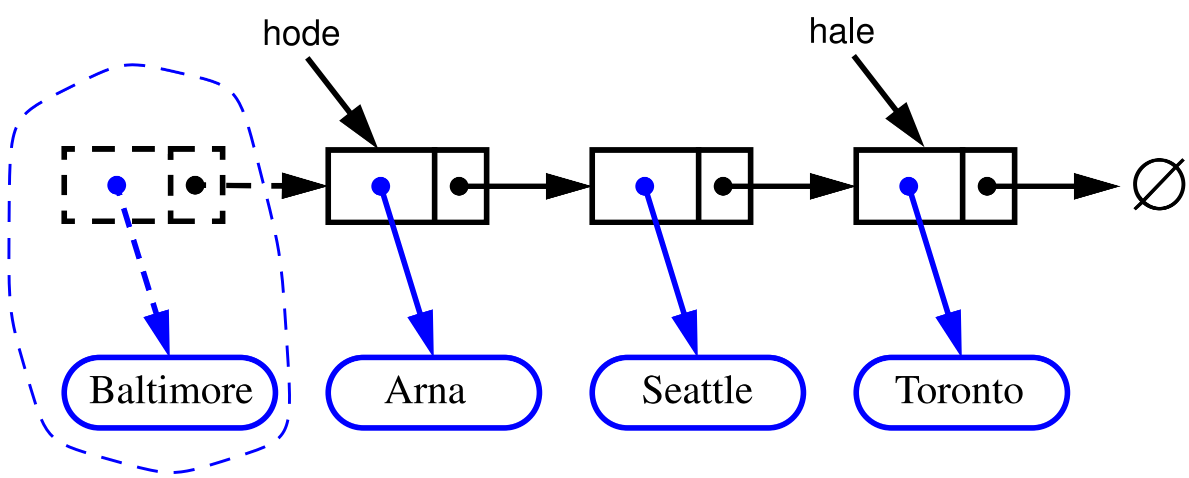
- noder kjedet sammen i en lineær ordning
- 

- hode på listen er foran i køen, og halen på listen er bakerst i køen
- hvorfor ikke motsatt?

Fjerne element i Hode

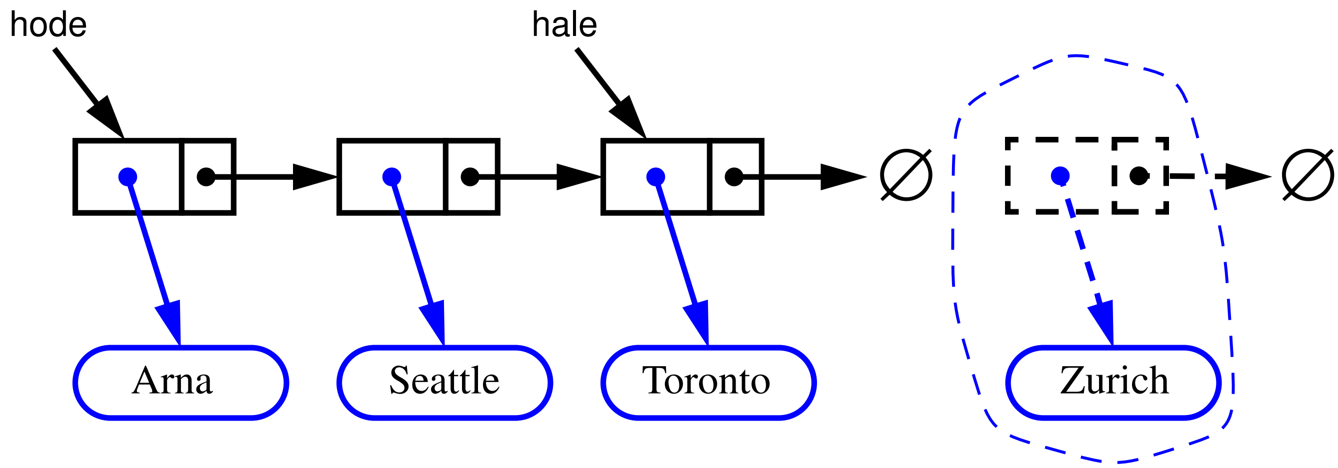


- flytt hodepeker fram

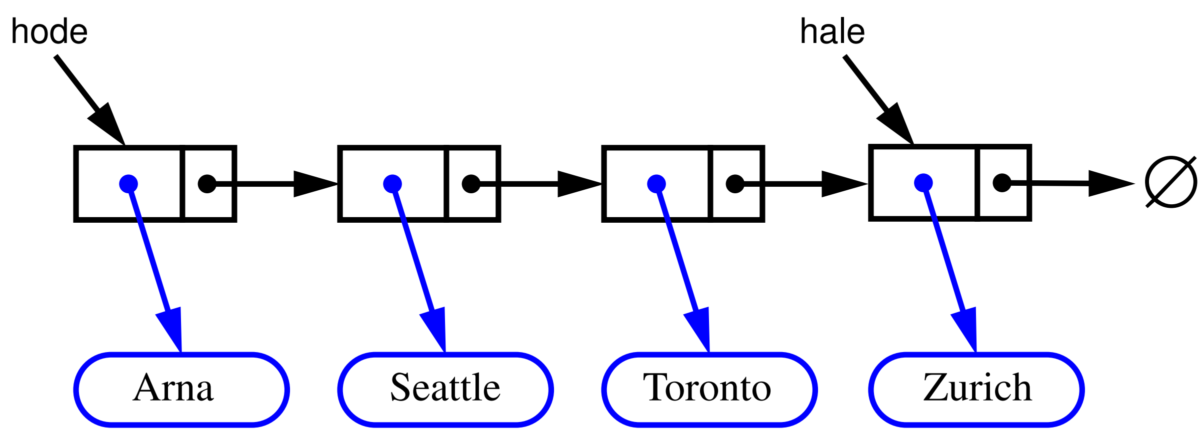


Sette inn i Halen

- lag ny node



- legg noden inn i kjeden og flytt halepeker



- hva med å fjerne element i halen?

To-sidige køer

- En to-sidig kø, (en: double-ended queue = deque), tillater innsetting og fjerning både foran og bak.
- Deque Abstrakt Data Type
  - insertFirst(*e*): Insert e at the deginning of deque.
  - insertLast(*e*): Insert e at end of deque
  - removeFirst(): Removes and returns first element
  - removeLast(): Removes and returns last element
- Støtte-metoder kan være:
  - first()
  - last()
  - size()
  - isEmpty()

Implementering av stabler og køer vha Deques

- Stabler med Deques:

Stack Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
top()	last()
push(e)	insertLast(e)
pop()	removeLast()

- Køer med Deques:

Queue Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue()	insertLast(e)
dequeue()	removeFirst()

Adaptor Pattern

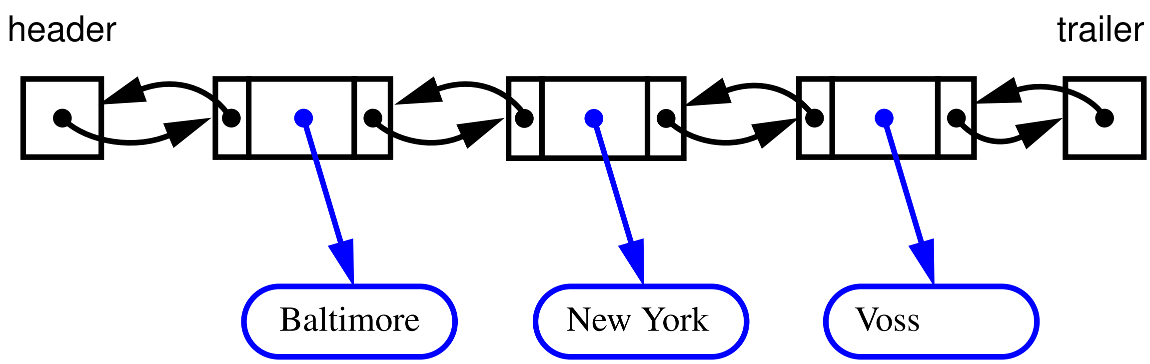
- Det å bruke en deque til å implementere kø eller stabel er eksempel på adaptor pattern. Adaptor pattern innebærer implementasjon av en ADT vha av en annen ADT.
- 2 mulige applikasjoner:
  - Spesialisere en generell ADT ved å endre noen metoder.

    Eks: implementere stabel ved deque.
  - Spesialisere typen objekter som en ADT bruker.

    Eks: Definer IntegerArrayStack som adapterer ArrayStack til å lagre kun integers.

Implementering av 2-sidige køer vha 2-veis lenkede lister

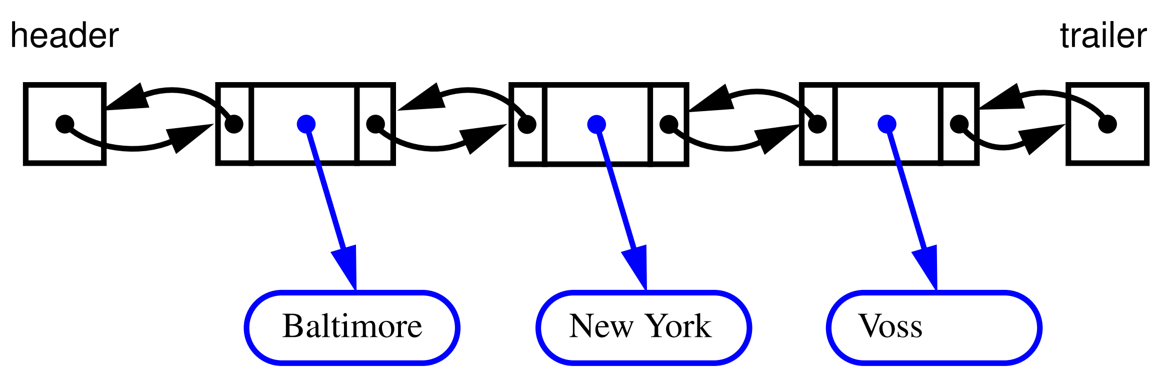
- Fjerning fra halen av 1-veis lenket liste kan ikke gjøres i O(1) tid, dvs konstant tid.
- For implementering av deque, bruker vi 2-veis lenket liste (eng: doubly linked list) med ekstra noder kallt header og trailer.



- Noder har next og prev lenker. Følgende metoder (NB: Ikke en ADT, men en data struktur):
  - setElement(Object e)
  - setNext(Object newNext)
  - setPrev(Object newPrev)
  - getElement()
  - getNext()
  - getPrev()
- Alle deque metoder har nå O(1) kjøretid.

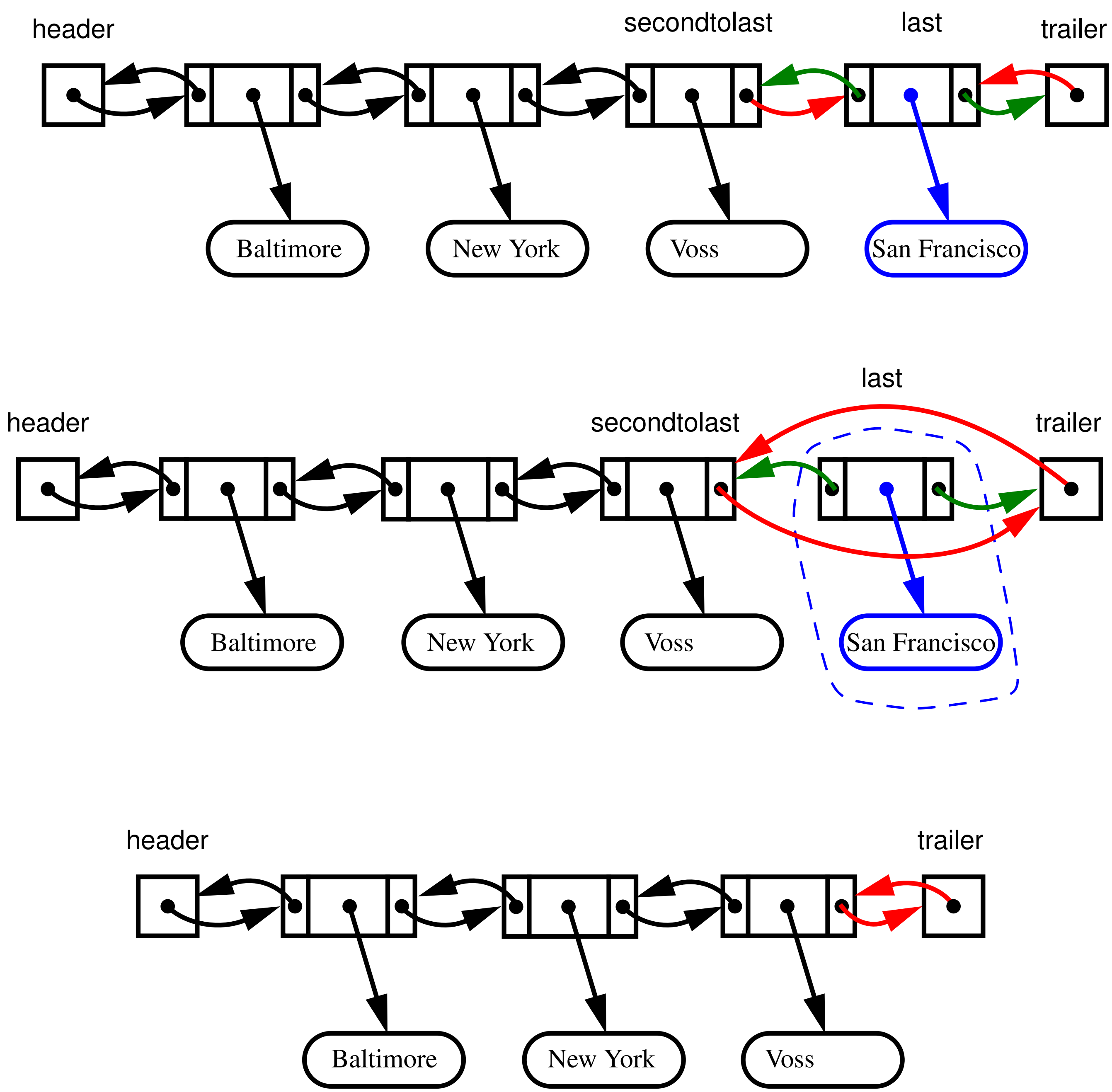
Implementering av 2-sidige køer vha 2-veis lenkede lister

- Hva gjør header og trailer nodene?
  - Header noden går foran første element i listen, og har en null prev lenke.
  - Trailer noden kommer etter siste element i listen, og har en null next lenke.
- Header og trailer nodene er “dummy” noder som ikke lagrer elementer
- Diagram:



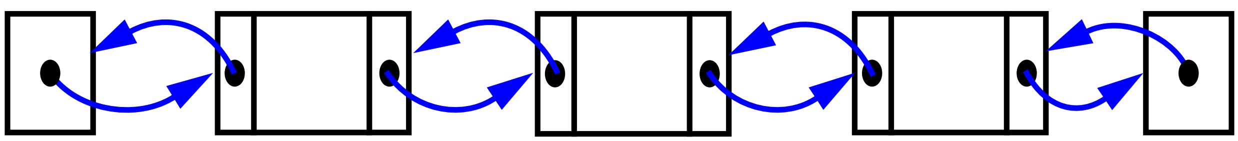
Implementering av 2-sidige køer vha 2-veis lenkede lister

- Visualisering av removeLast().



SEKVENSER

- ADT Vektorer
- ADT Posisjoner
- ADT Lister
- Generell ADT Sekvenser



- Idag: Vektor implementert vha tabell og vha 2-veis lenket liste
-

•

ADT Vektor

- ADT Vektor er en samling av n elementer *S*, ordnet lineært vha rang(rank), med følgende metoder:

  - elemAtRank(*r*):
    Return the element of *S* with rank *r*; an error occurs if $r < 0$ or $r > n - 1$

  - replaceAtRank(*r*,*e*):
    Replace the element at rank *r* with *e* and return the old element; an error condition occurs if $r < 0$ or $r > n - 1$

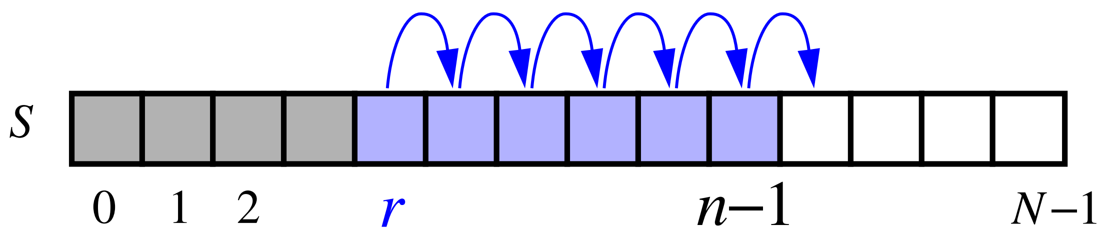
  - insertAtRank(*r*,*e*):
    Insert a new element into *S* which will have rank *r*; an error occurs if $r < 0$ or $r > n$

  - removeAtRank(*r*):
    Remove from *S* the element at rank *r*; an error occurs if $r < 0$ or $r > n - 1$

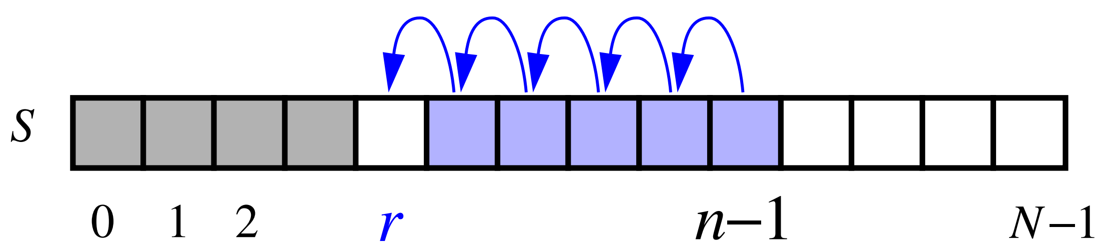
Tabell-basert implementasjon

- Pseudokode:

Algorithm insertAtRank(r,e):
  for $i = n - 1, n - 2, \ldots, r$ **do**
    $S[i+1] \leftarrow s[i]$
  $S[r] \leftarrow e$
  $n \leftarrow n + 1$



Algorithm removeAtRank(r):
  $e \leftarrow S[r]$
  for $i = r, r + 1, \ldots, n - 2$ **do**
    $S[i] \leftarrow S[i + 1]$
  $n \leftarrow n - 1$
  return



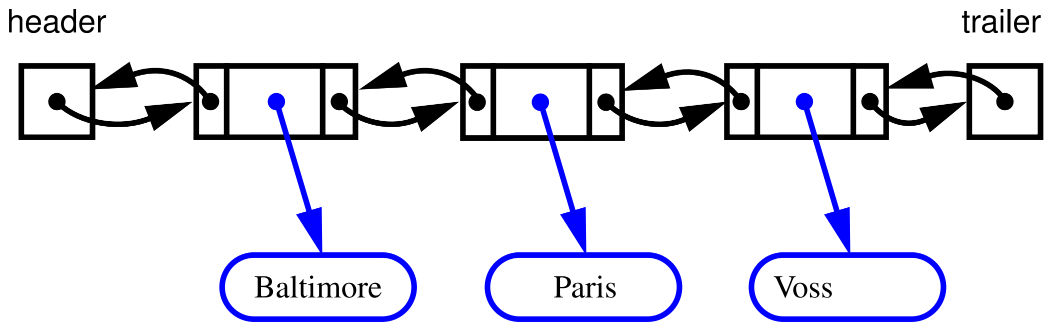
Tabell-basert implementasjon

- Kjøretid (tidskompleksitet) for metodene:

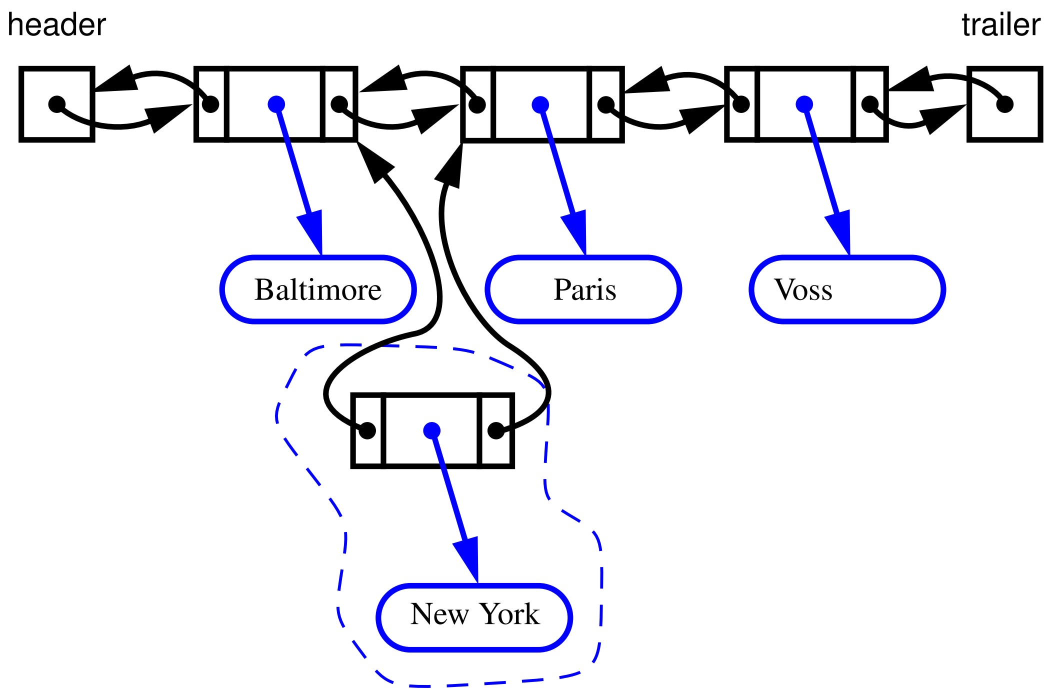
Metode	Tid
size	$\boldsymbol{O}(1)$
isEmpty	$\boldsymbol{O}(1)$
elemAtRank	$\boldsymbol{O}(1)$
replaceAtRank	$\boldsymbol{O}(1)$
insertAtRank	$\boldsymbol{O}(n)$
removeAtRank	$\boldsymbol{O}(n)$

Implementasjon vha 2-veis lenket liste

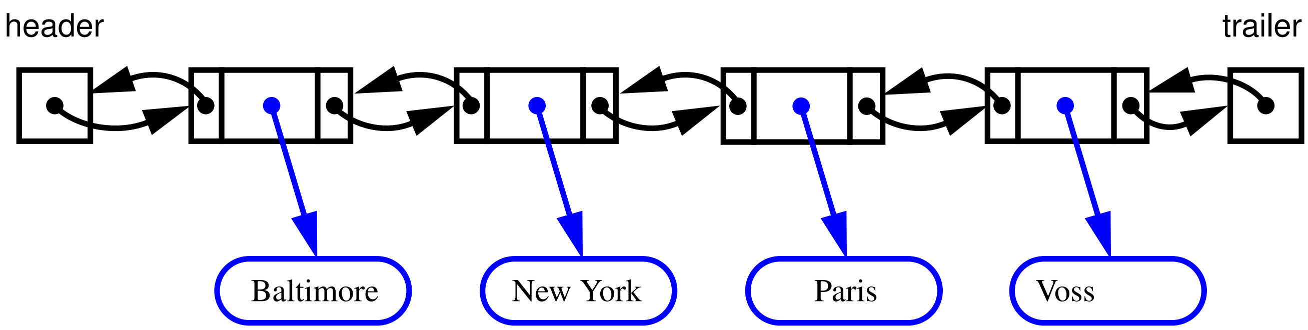
- listen før innsetting:



- lag ny node for innsetting:



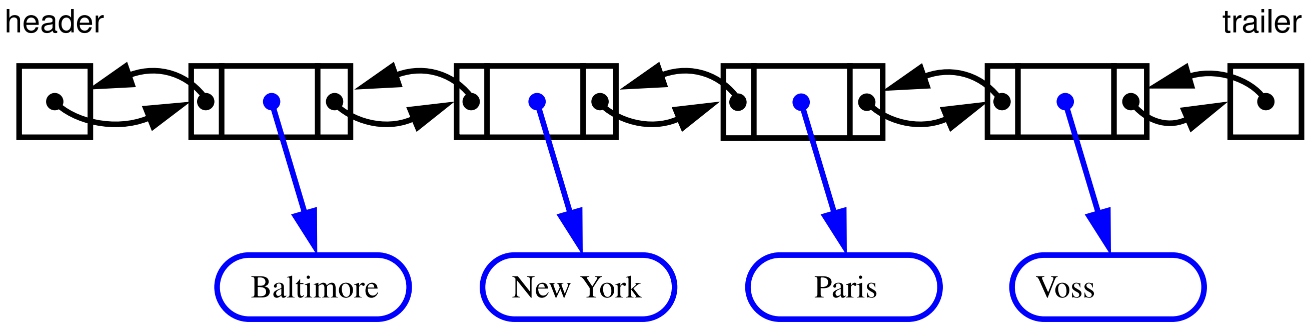
- listen etter innsetting:



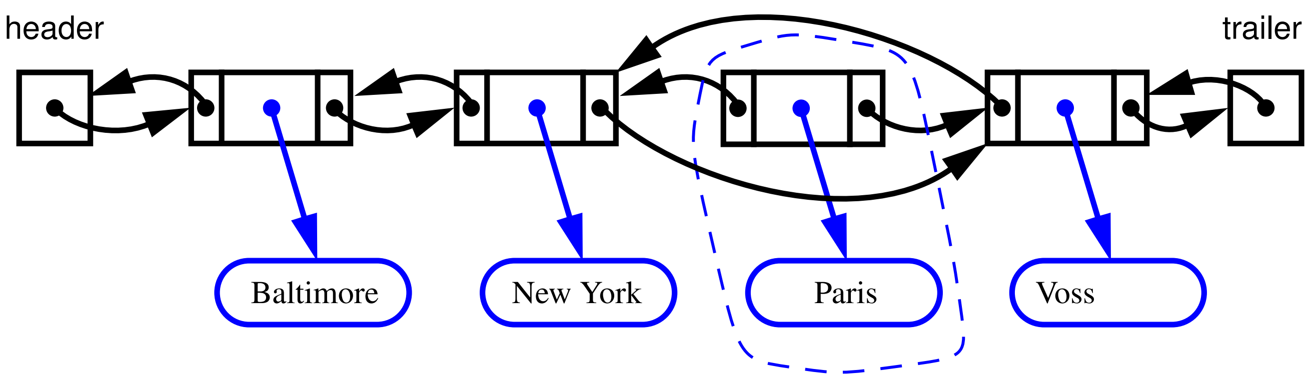

```
public void insertAtRank (int rank, Object element)
  throws BoundaryViolationException {
    if (rank < 0 || rank > size())
      throw new BoundaryViolationException("invalid rank");
    DLNode next = nodeAtRank(rank); // the new node
                          //will be right before this
    DLNode prev = next.getPrev(); // the new node
                          //will be right after this
    DLNode node = new DLNode(element, prev, next);
    // new node knows about its next & prev. Now
    // we tell next & prev about the new node.
    next.setPrev(node);
    prev.setNext(node);
    size++;
  }
```

Implementering vha 2-veis lenket liste

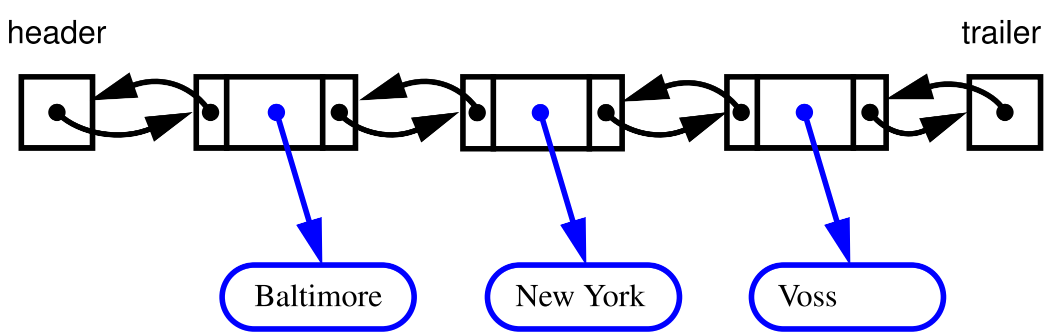
- listen før fjerning av element:



- fjern en node:



- etter fjerning:



Java-Implementasjon

- fjerning av node

```
public Object removeAtRank (int rank)
                throws BoundaryViolationException {
  if (rank < 0 || rank > size()–1)
    throw new BoundaryViolationException(“Invalid
        rank.”);
  DLNode node = nodeAtRank(rank); // node to
                                  // be removed
  DLNode next = node.getNext(); // node before it
  DLNode prev = node.getPrev(); // node after it
  prev.setNext(next);
  next.setPrev(prev);
  size--;
  return node.getElement(); // returns the
                // element of the deleted node
}
```

Java-Implementasjon

- finne node med gitt rank

```
private DLNode nodeAtRank (int rank) {
  // auxiliary method to find the node of the
  // element with the given rank. We make
  // auxiliary methods private or protected.
  DLNode node;
  if (rank <= size()/2) { //scan forward from head
     node = header.getNext();
     for (int i=0; i < rank; i++)
     node = node.getNext();
  }
  else { // scan backward from the tail
     node = trailer.getPrev();
     for (int i=0; i < size()-rank-1 ; i++)
     node = node.getPrev();
  }
  return node;
}
```

Noder

- Lenkede lister kan effektivt utføre ***node-baserte operasjoner***:
  - removeAtNode(Node v) og insertAfterNode(Node v, Object e), tar begge O(1) tid.
- MEN, node-baserte operasjoner kan ikke overføres til tabell-baserte implementasjoner, da tabeller ikke har noder!
- Node er en implementasjonsdetalj.
- **Dilemma**:
  - Hvis vi ikke definerer nodebaserte operasjoner, så gjør vi ikke full nytte av lenkede lister.
  - Hvis vi definerer dem, så bryter vi med grunnideen om ADT, dvs at ADT skal defineres uavhengig av bestemt implementasjon.

Fra Noder til Posisjoner

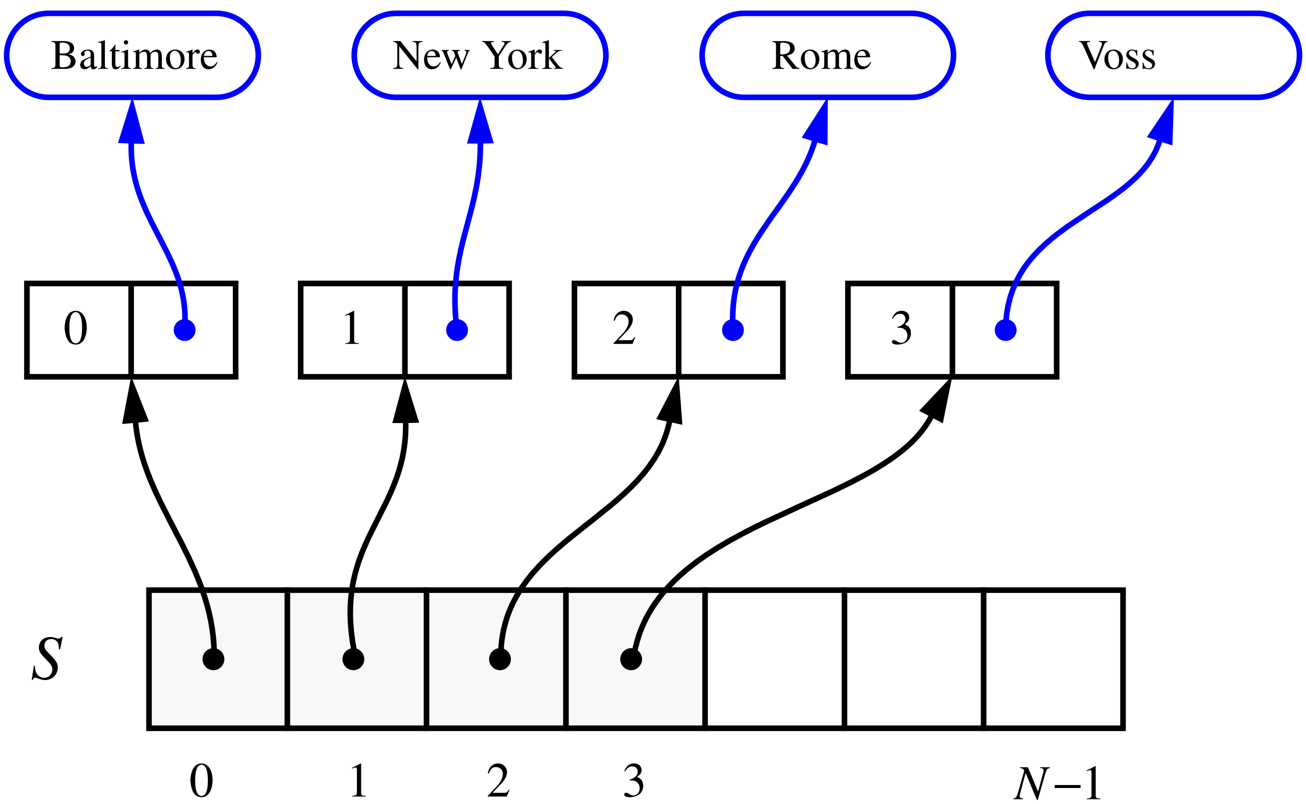
- ADT ***Posisjon*** (eng: position), som intuitivt er en 'celle', dvs et "sted" et element kan befinne seg.
- Posisjoner har kun en metode:
  element(): Return the element at this position .
- En posisjon defineres relativt til andre posisjoner (foran/bak relasjon)
- Posisjoner er ikke knyttet til et element eller en rank

ADT LISTE

- ADT Liste er en ADT med posisjon-baserte metoder
- generiske metoder size(), isEmpty()
- query metoder isFirst(p), isLast(p)
- accessor metoder first(), last(), before(p), after(p)
- update metoder swapElements(p,q), replaceElement(p,e), insertFirst(e), insertLast(e), insertBefore(p,e), insertAfter(p,e). remove(p)
- implementert vha 2-veis lenket liste vil hver av disse metodene ta O(1) tid.

Sekvens ADT

- Kombinerer ADTer Vektor og List (multippel arv av grensesnitt)
- Nye metoder bygger bro mellom rank og posisjon:
  - atRank(r) returns a position
  - rankOf(p) returns an integer rank
- En tabell-basert implementasjon må bruke objekter for å representere posisjoner:



Sammenlikning av sekvens-implementasjoner

Operasjoner	Array	List
size, isEmpty	$O(1)$	$O(1)$
atRank, rankOf, elemAtRank	$O(1)$	$O(n)$
first, last	$O(1)$	$O(1)$
before, after	$O(1)$	$O(1)$
replaceElement, swapElements	$O(1)$	$O(1)$
replaceAtRank	$O(1)$	$O(n)$
insertAtRank, removeAtRank	$O(n)$	$O(n)$
insertFirst, insertLast	$O(1)$	$O(1)$
insertAfter, insertBefore	$O(n)$	$O(1)$
remove	$O(n)$	$O(1)$

Iterator

- Er en abstraksjon av det å ‘gå gjennom alle elementene i en gitt samling elementer’.
- Iterator enkapsulerer begrepene ‘sted’ og ‘neste’.
- Utvider ADT Posisjon
- Generiske og spesialiserte iteratorer.
- ***ObjectIterator***
  - hasNext()
  - nextObject()
  - object()
- ***PositionIterator***
  - nextPosition()
- Nyttige metoder som returnerer iteratorer:
  - elements()
  - positions()