Datastrukturer & Algoritmer, Datalogi C
Forelæsning 22/10-2002
Henning Christiansen

Stakke, køer og lidt om hægtede lister

Hvorfor?

- Fundamentale datastrukturer man får brug for igen og igen
- Et indblik i Java Collections API

Dagens program:

Stakke

- eksempler på anvendelse
- implementation i Java vha. arrays

Køer (nej, ikke sådan nogen, der siger muuuuuuuuuh)

- eksempler på anvendelse
- implementation i Java vha. ”cirkulært” array

Hægtede lister — ”tungere” end arrays men ultimativ fleksibilitet

- enkelthægtet
- dobbelthægtet

Datastrukturen en stak:

Implementerer først-ind–sidst-ud samling af objekter:

Mindst følgende metoder

void push(Object x) — lægger x på stakken

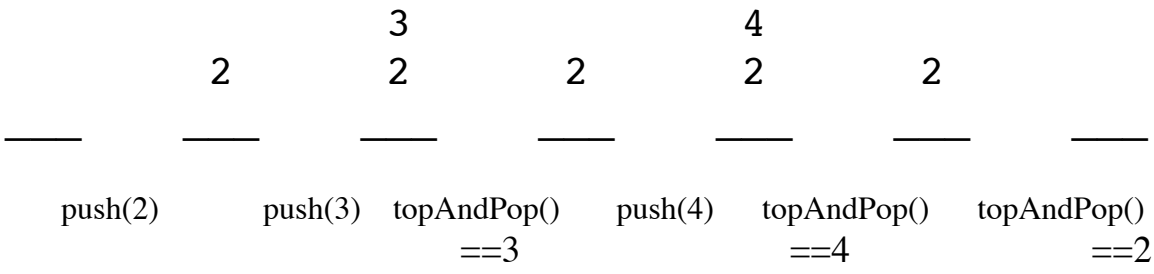
Object topAndPop() — returnerer ”øverste” element på stak.

(kaldes andre steder ofte blot ”pop”)

Forståelse af stak ud fra operationerne:

push(a); push(b); topAndPop() == b; topAndPop() == a;

Forståelse ud fra model:



Velegnet til ting som er indlejret i hinanden:

Rekursive metoder implementeres ved stak:

- metodekald: ”push” plads til parametre og lokale variable
- returhop: ”pop”, dvs. glemme alt om kald og **gå tilbage til gammel tilstand**

Håndoptimering af rekursive metode:
Ofte relevant hvis compiler spilder unødigt meget overhead på rekursive procedurekald:

Vedligehold en lokal stak svarende til lokale variable og parametre

God opgave: Afskaf rekursionen i quicksort mod at indføre en stak

Anvendelse af stak til syntaksgenkendelse af programmeringssprog:

Eksempel på grammatik med matchende paranteser (ingen operatorer):

```
<udtryk> ::=      1   |   2   |   3   |  . . .  |   9
           |      ’(’ udtryk ’)’
           |      ’{’ udtryk ’}’
           |      <udtryk> <udtryk>
```

Følgende er eksempel på udtryk (1 2) {4(5 6) {7} }

Algoritme for genkendelse:

```
t = læs_et_tegn();
while(true) {
   if(er_et_ciffer(t)) ;  //skip
   else if(t==’)’) if(topAndPop()!=’(’) fejl;
   else if(t==’}’) if(topAndPop()!=’{’) fejl;
   else push(t);
   if (!flere-tegn-at-læse()) break;
   t = læs_et_tegn();};
if(stakken-er-tom) det_gik_godt; else det_gik_skidt;
```

NB: kan også implementeres ved rekursion

Anvendelse af stak: Enhver compiler benytter en regnestak

At udregne 1 + 2*3*(4 + 5):

```
push 1;
push 2;
push 3;
mult;    // ≈ push(   topAndPop() * topAndPop()   )
push 4;
push 5;
add;     // ≈ push(   topAndPop() + topAndPop()   )
mult;    // ≈ push(   topAndPop() * topAndPop()   )
add;     // ≈ push(   topAndPop() + topAndPop()   )
// resultatet er på stakken
```

I praksis:

- De øverste dele af stakken opbevares i regneregistre og resten i RAM
- Push, add, mult osv. findes som maskininstruktioner
- ... men compilere producerer ofte mellemkode som ser ud som overfor

Implementation af en generel stak i Java (jvf. bogen s. 514 ff)

- benytter et array som fordobles ved overløb

```
public class ArrayStack implements Stack
{ public ArrayStack( )
        {theArray = new Object[ DEFAULT_CAPACITY ]; topOfStack = -1;}

   public boolean isEmpty( )  { return topOfStack == -1;}

   public void makeEmpty( )  {  topOfStack = -1; }

  public Object top( )
     { if( isEmpty( ) ) throw new UnderflowException( "ArrayStack top" );
       return theArray[ topOfStack ]; }

public void pop( )
   { if( isEmpty( ) ) throw new UnderflowException( "ArrayStack pop" );
     topOfStack--; }

public Object topAndPop( )
   { if( isEmpty( ) ) throw new UnderflowException( "ArrayStack topAndPop" );
      return theArray[ topOfStack-- ];}

public void push( Object x )
   { if( topOfStack + 1 == theArray.length ) doubleArray( );
      theArray[ ++topOfStack ] = x;}

 private void doubleArray( )
   { Object [ ] newArray;
      newArray = new Object[ theArray.length * 2 ];
      for( int i = 0; i < theArray.length; i++ ) newArray[ i ] = theArray[ i ];
      theArray = newArray;}

private Object [ ] theArray;
private int        topOfStack;
private static final int DEFAULT_CAPACITY = 10;   }
```

Køer

(simple kø; ikke prioritetskø som er noget andet)

Implementerer først-ind–først-ud samling af objekter:

Mindst følgende metoder

void enqueue(Object x) — stiller x i køen

Object dequeue() — returnerer ”forreste” element i køen

Forståelse af stak ud fra operationerne:

enqueue(a); enqueue(b); dequeue() == a; dequeue() == b;

Forståelse ud fra model:

enqueue(a)	enqueue(b);	dequeue() == a	dequeue() == b
<>	<a>	<b,a>	<b>	<>
tom	forrest=bagest=a	forrest=a bagerst=b	forrest=bagerst=b	forrest: ? bagest. ?

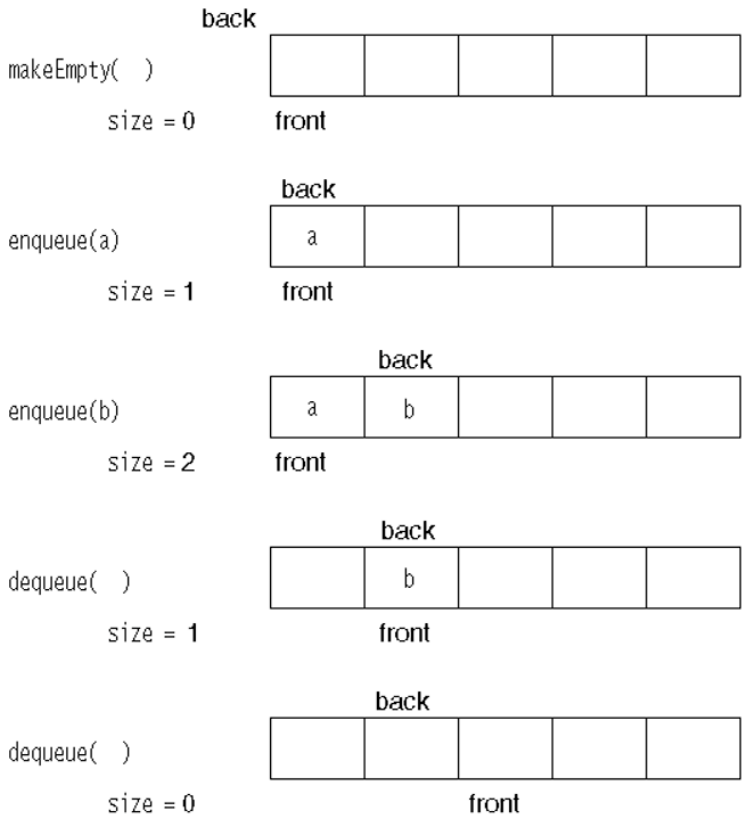
Anvendelser:

- som hjælpere for andre algoritmer, f.eks. korteste vej, topologisk sortering
- simulering af fysisk kø
- afvikling af printerjobs
- datanetværk: ophobning af ikke modtagne pakker
- ....

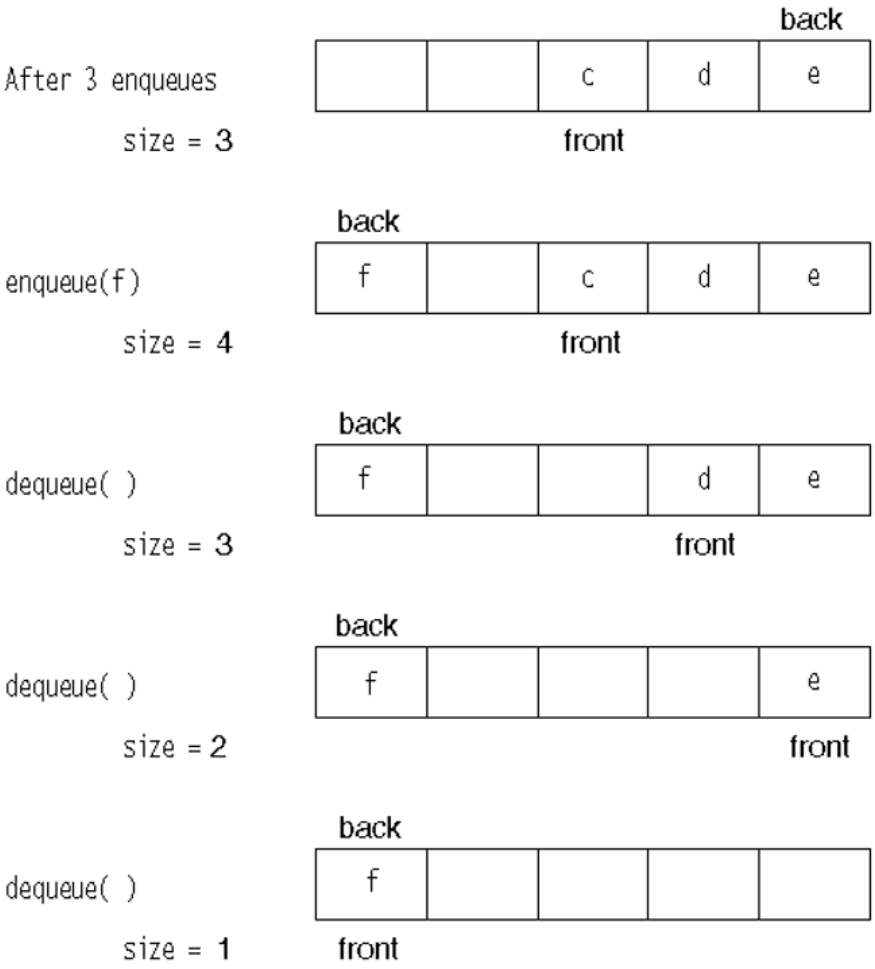
Implementation ved cirkulært array:

”cirkulært” ≈ hvordan vi tænker på det!

```
public class ArrayQueue implements Queue
{  public ArrayQueue( )
    { theArray = new Object[ DEFAULT_CAPACITY ]; makeEmpty( );}
   public void makeEmpty( )
   { currentSize = 0; front = 0; back = -1; }

    public Object dequeue( ) { .... }

   public void enqueue( Object x ) {.....}

   private Object [ ] theArray;
   private int       currentSize;
   private int       front;
   private int       back;

   private static final int DEFAULT_CAPACITY = 10;}
```



”Cirkulært” array



Implementation af metoder i Java

```
public class ArrayQueue implements Queue
{ public ArrayQueue( )
   {theArray = new Object[ DEFAULT_CAPACITY ]; makeEmpty( );}

 public boolean isEmpty( ) { return currentSize == 0;}

 public void makeEmpty( ) { currentSize = 0; front = 0; back = -1; }

public Object dequeue( )
   {if( isEmpty( ) )
        throw new UnderflowException( "ArrayQueue dequeue" );
    currentSize--;
    Object returnValue = theArray[ front ];
     front = increment( front );
     return returnValue; }

 private int increment( int x )
   { if( ++x == theArray.length ) x = 0; return x; }

   public Object getFront( )
   { if( isEmpty( ) )
         throw new UnderflowException( "ArrayQueue getFront" );
     return theArray[ front ];}

  public void enqueue( Object x )
   { if( currentSize == theArray.length )
         doubleQueue( );
      back = increment( back );
      theArray[ back ] = x;
      currentSize++; }
```

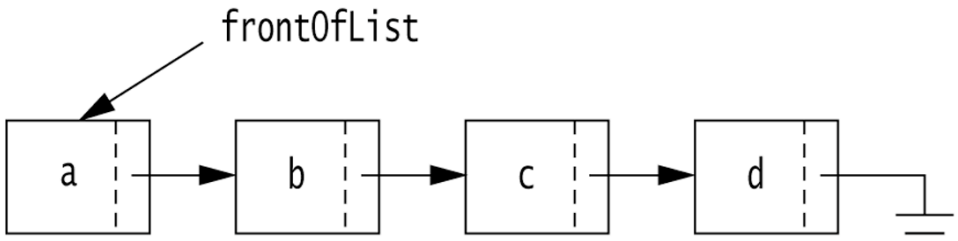
```
 private void doubleQueue( )
   { Object [ ] newArray;
     newArray = new Object[ theArray.length * 2 ];
     // Copy elements that are logically in the queue
     for( int i = 0; i < currentSize; i++, front = increment( front ) )
         newArray[ i ] = theArray[ front ];
     theArray = newArray;
      front = 0;
      back = currentSize - 1; }

   private Object [ ] theArray;
   private int        currentSize;
   private int        front;
   private int        back;

   private static final int DEFAULT_CAPACITY = 10;
}
```

Hægtede lister, den grundlæggende idé:

Fleksibel datastruktur, som kan implementere stakke, køer med videre med udvidede metoder
Vigtigste egenskab: Der kan indsættes og slettes hvor som helst i listen

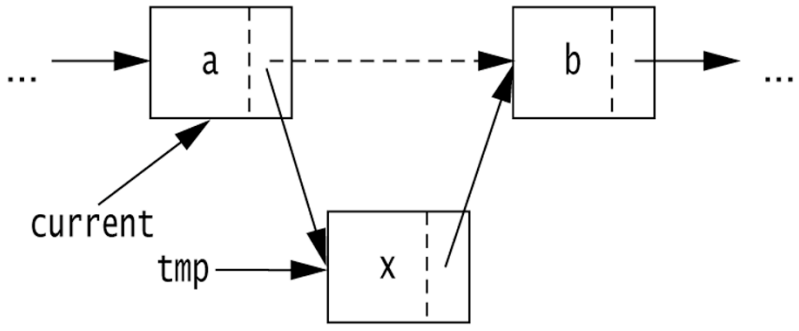
Eksempel:

```
class ListNode {
   Object element;
   ListNode next;

  public void ListNode(Object e, ListNode n)
     {element=e; next=n;} }
```



At indsætte objekt x vilkårligt sted i listen udpeget ved reference ”current”:

```
current.next = new ListNode(x, current.next);
```



At slette element efter ”current”:

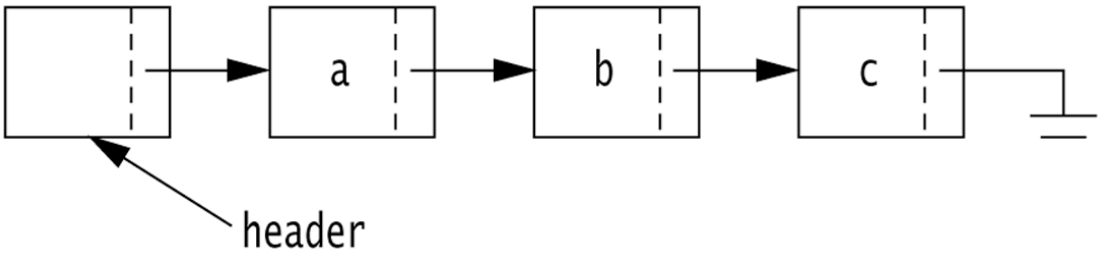
```
current.next = current.next.next;
```

Hægtede lister i Java (Collections API)

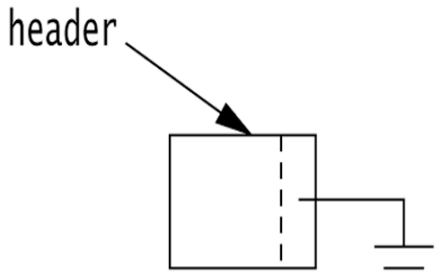
NB: Bogens version, lidt anderledes end Collections!
Kombinerer tre klasser:

- listeelementer: ListNodes som holder Object'er
- LinkedListIterator (som ikke er en udvidelse af Iterator)
  - generaliseret "position i listen"
- LinkedList: En liste som selvstændigt objekt,
  - implementation "List" som er impl. af Collection

```
class ListNode {
   public ListNode( Object theElement ) {this( theElement, null );}

   public ListNode( Object theElement, ListNode n )
    { element = theElement; next    = n;}

    public Object   element;
    public ListNode next;}

public class LinkedList {
  public LinkedList( ) {header = new ListNode( null );}
  // methods
  private ListNode header; }
```



Tom liste:



Formål med header-element:

For at forenkle implementation af metoder:
Den tomme liste ligner en hver anden liste, f.eks. indsættelse.

Indsættelse.... indsætte hvor?

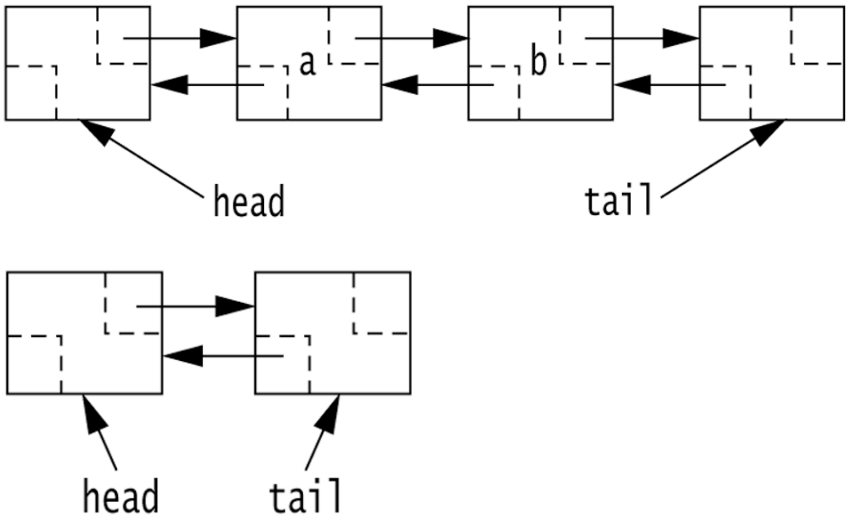
LinkedListIterator, krydsning mellem

- Iterator ~ den generaliserede for-sætning
- Listeposition for indsættelse og sletning m.v.

```
public class LinkedListIterator {
  LinkedListIterator( ListNode theNode ) {current = theNode;}
  public boolean isValid( ) {return current != null;}
  public Object retrieve( ) {return isValid( ) ? current.element : null;}
  public Object advance( ) {if(isValid()) current=current.next;}
  ListNode current; }

public class LinkedList {
  public LinkedList() { ... }
  public boolean isEmpty() { ... }
  public LinkedListIterator zeroth(){return new LinkedListIterator(header);}
  public LinkedListIterator first(){return new LinkedListIterator(header.next);}

  public void insert( Object x, LinkedListIterator p ) {
    if( p != null && p.current != null )
      p.current.next = new ListNode( x, p.current.next );}

  public LinkedListIterator find( Object x ) {
    ListNode itr = header.next;
    while( itr != null && !itr.element.equals( x ) ) itr = itr.next;
    return new LinkedListIterator( itr ); }

  public void remove( Object x ){
    LinkedListIterator p = findPrevious(x);
    if(p.current.next!= null)
      p.current.next = p.current.next.next; }} // Bypass deleted node
```

Variationer:

Dobbelthægtede lister:



Fordel:
Vi kan rykke iterator frem **og tilbage** på konstant tid

Omkostning:
Fylder en reference mere pr. knude;
Indsætte, slette tager ca. 2 x længere tid.

Sorterede lister

SortedLinkedList extends LinkedList

Udvider med metode insert(Comparable x),

- som (ifølge sagens natur) må tælle sig ca. halvvejs gennem listen

Kombinerer listens fleksibilitet med fordelen ved at ting er sorteret;
OBS: Binær søgning virker ikke :(

Se bogens afsnit 17.5 og dukomentation for Javas Collections API for stort udvalg af metoder.

Afslutning:

Tilsyneladende en stor rodebunke af datastrukturer som ligner hinanden!!!

Hjælp, hvad gør vi?

Overvej følgende spørgsmål?

- Hvilken overordnet strategi har vi brug for
- Stak? sidst-ind-først-ud
- Kø? Først-ind-først-ud
- Noget endnu mere fleksibelt, i retning af (dobbelt-hægtet) liste?

Generel i forhold til valg af Collection eller klasser i det hele taget:

- Hvilke metoder har vi brug for?
- Hvilke metoder er tidskritiske?
- ... sidder i ”den inderste løkke” og kaldes igen og igen og igen?
- Hvilke kun en sjælden gang?