

Sequence mit Feldern (1)

▷ Interne Repräsentation einer Sequence

```
var ValueType A[1..maxN]
```

```
var int n
```

▷ Initialisierung

```
n := 0
```

```
procedure INSERT(ValueType x, int p) : int
```

```
  if n = maxN then throw Overflow
```

```
  for i := n downto p do
```

```
    A[i + 1] := A[i]
```

```
  end for
```

```
  A[p] := x
```

```
  n := n + 1
```

```
  return p
```

```
end procedure
```

Sequence mit Feldern (2)

```
procedure DELETE(int p)
  for  $i := p + 1$  to  $n$  do
     $A[i - 1] := A[i]$ 
  end for
   $n := n - 1$ 
end procedure
```

```
function GET(int  $i$ ) : ValueType
  return  $A[i]$ 
end function
```

```
procedure CONCATENATE(Sequence  $S'$ )
  if  $n + S'.n > \text{max}N$  then throw Overflow

  for  $i := 1$  to  $S'.n$  do
     $A[n + i] := S'.A[i]$ 
  end for

   $n := n + S'.n$ 
   $S'.n := 0$ 
end procedure
```

Sequence mit Listen (1)

▷ Repräsentation eines Listenelements

struct ListElement

var *ListElement* prev ▷ Verweis auf Vorgänger

var *ListElement* next ▷ Verweis auf Nachfolger

var *ValueType* value ▷ gespeicherter Wert

end struct

▷ Interne Repräsentation einer Sequence

var *ListElement* head

▷ Initialisierung

head := *nil*

Sequence mit Listen (2)

```
procedure INSERT( ValueType x, ListElement p) :  
ListElement  
  var ListElement q := new ListElement  
  q.value := x  
  
  if head = nil then      ▷ War Liste vorher leer?  
    head := q.next := q.prev := q  
  else  
    if p = head then head := q  
    if p = nil then p := head  
  
    q.next := p  
    q.prev := p.prev  
    q.next.prev := q.prev.next := q  
  end if  
  
  return q  
end procedure
```

Sequence mit Listen (3)

```
procedure DELETE(ListElement p)
  if  $p.next = p$  then
     $head := nil$ 
  else
     $p.prev.next := p.next$ 
     $p.next.prev := p.prev$ 
    if  $p = head$  then  $head := p.next$ 
  end if

  delete  $p$ 
end procedure
```

Sequence mit Listen (4)

procedure CONCATENATE(*Sequence* S')

if $head = nil$ **then**

$head := S'.head$

else if $S'.head \neq nil$ **then**

var *ListElement* $first := S'.head$

var *ListElement* $last := first.prev$

$head.prev.next := first$

$last.next := head$

$first.prev := head.prev$

$head.prev := last$

end if

$S'.head := nil$

 ▷ *Sequence* S' ist jetzt leer!

end procedure

Sequence mit Listen (5)

```
function GET(int i) : ValueType  
  var ListElement p := head  
  while i > 1 do  
    p := p.next  
    i := i - 1  
  end while  
  
  return p.value  
end function
```

Stacks mit Feldern

▷ Interne Repräsentation des Stacks

```
var ValueType A[1..maxN]
```

```
var int n                ▷ Anzahl Elemente im Stack
```

▷ Initialisierung

```
n := 0                    ▷ leerer Stack
```

```
function ISEEMPTY( ) : bool
```

```
    return n = 0
```

```
end function
```

```
procedure PUSH( ValueType x)
```

```
    if n = maxN then throw Overflow
```

```
    n := n + 1
```

```
    A[n] := x
```

```
end procedure
```

```
function POP( ) : ValueType
```

```
    if n = 0 then throw Underflow
```

```
    var ValueType x := A[n]
```

```
    n := n - 1
```

```
    return x
```

```
end function
```

Stacks mit Listen (1)

struct SListElement

var *SListElement* next ▷ Verweis auf Nachfolger

var *ValueType* value ▷ abgespeicherter Wert

end struct

▷ Interne Repräsentation des Stacks

var *SListElement* head ▷ Anfang der Liste

▷ Initialisierung

head := *nil* ▷ leerer Stack

function ISEEMPTY() : **bool**

return *head* = *nil*

end function

Stacks mit Listen (2)

```
procedure PUSH( ValueType x)
  var SListElement p := new SListElement

  p.value := x
  p.next := head
  head := p
end procedure
```

```
function POP( ) : ValueType
  if head = nil then throw Underflow

  var SListElement p := head
  var ValueType x := p.value

  head := p.next
  delete p

  return x
end function
```