Datenstrukturen und Algorithmen

Vorlesung 6

Überblick

- Vorige Woche:
  - Einfach verkettete Listen - Iterator
  - Doppelt verkettete Listen (DLL)
  - Sortierte Listen
  - Arrays und Listen – Zusammenfassung
  - Zirkuläre Listen
- Heute betrachten wir:
  - Repräsentierungen für schwachbesetzte Matrix, Stack, Queue, Prioritätsschlange, Deque
  - Einfach verkettete Listen auf Arrays – Einfügen, Löschen, Suchen

Schwachbesetzte Matrix – Wiederholung

- Falls die Matrix viele 0-Werte (oder 0_{TElem}) enthält, dann redet man von **schwachbesetzte** oder **dünnbesetzte** Matrix (**sparse**) → in diesem Fall ist es effizienter nur die Elemente unterschiedlich von 0 zu speichern

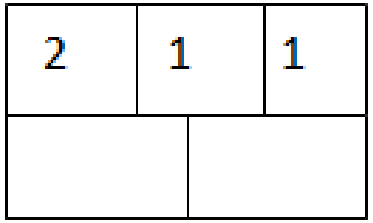
Schwachbesetzte Matrix – Repräsentierung D

- **Verkettete Repräsentierung mit Hilfe von zirkulären Listen**
- Jeder Knoten enthält die Zeile, die Spalte, den Wert (unterschiedlich von 0) und zwei Pointers: zu dem nächsten Element von derselben Zeile, und zu dem nächsten Element von derselben Spalte
- Die letzten Knoten haben ein Pointer zu dem ersten Knoten (zirkuläre Listen)
- Jede Zeile und Spalte hat einen speziellen Knoten, der den Anfang der entsprechenden Liste bezeichnet

Schwachbesetzte Matrix – Repräsentierung D Beispiel

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 7 \\ 0 & 6 & 0 & 0 \\ 0 & 0 & 9 & 1 \end{bmatrix}$$

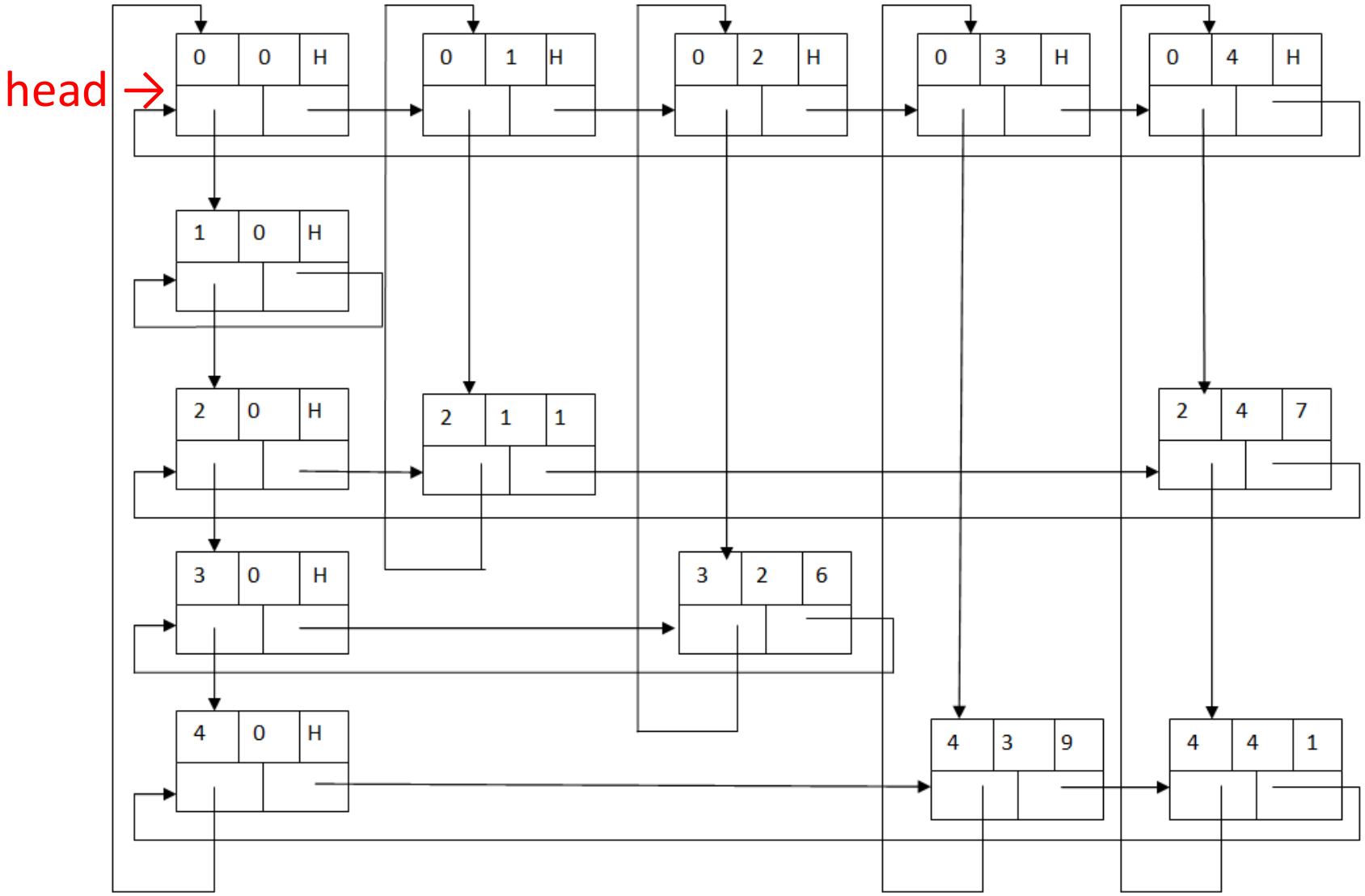
- Jeder Knoten enthält:
  - die Zeile,
  - die Spalte,
  - den Wert (unterschiedlich von 0),
  - einen Pointer zu dem nächsten Element von derselben Zeile, und
  - einen Pointer zu dem nächsten Element von derselben Spalte
- Element von Zeile 2, Spalte 1 mit dem Wert 1:



Schwachbesetzte Matrix – Repräsentierung D

- Die Knoten mit dem Wert *H* sind *Header* Knoten. Diese sind keine echten Elemente, sondern bezeichnen nur den Anfang der entsprechenden Zeile oder Spalte
- Die Knoten werden irgendwo im Speicherplatz gespeichert, nicht unbedingt auf aufeinanderfolgenden Speicherplätzen, aber die graphische Repräsentierung wurde intuitiv in Form einer „Matrix“ dargestellt
- Da die Repräsentierung zirkuläre Listen benutzt, verweist der letzte Knoten von jeder Zeile und Spalte auf den entsprechenden Header Knoten
- Es genügt die Adresse des (0,0) Header-Knotens zu speichern

Schwachbesetzte Matrix – Repräsentierung D



$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 7 \\ 0 & 6 & 0 & 0 \\ 0 & 0 & 9 & 1 \end{bmatrix}$$

ADT Stack - Wiederholung

- ADT Stack ist ein Container, wo der Zugriff zu den Elementen auf ein Ende des Behälters (*Top des Stacks)* beschränkt ist:
  - In einem Stack können Elemente nur “von oben” hinzugefügt (eingekellert) und entnommen (ausgekellert) werden
  - Man kann nur auf das „oberste“ Element zugreifen

- Es gilt also das LIFO Prinzip (Last-in-First-Out-Prinzip) – das letzte Element, das eingefügt wurde, wird als erstes gelöscht werden

ADT Stack - Repräsentierung

- Um ADT Stack zu implementieren kann man folgende Datenstrukturen für die Repräsentierung benutzen:
  - Arrays:
    - Statische Arrays (für konstante Kapazität)
    - Dynamische Arrays
  - Verkettete Liste
    - Einfach verkettete Listen
    - Doppelt verkettete Listen

ADT Stack – Repräsentierung auf Arrays

- Wo sollte man den Top des Stacks speichern, um eine gute Effizienz zu erreichen?
- Es gibt zwei Möglichkeiten:
  - Den Top des Stacks am Anfang des Arrays zu speichern – bei jeder *push* und *pop* Operation müssen alle Elemente des Arrays nach rechts, bzw. nach links, verschoben werden
  - Den Top des Stacks am Ende des Arrays zu speichern – bei den *push* und *pop* Operation müssen die Elemente des Arrays nicht mehr verschoben werden
- Also man speichert den Top des Stacks am Ende des Arrays

ADT Stack – Repräsentierung auf einfach verketteten Listen (SLL)

- Wo sollte man den Top des Stacks speichern, um eine gute Effizienz zu erreichen?
- Es gibt zwei Möglichkeiten:
  - Den Top des Stacks am Ende der Liste zu speichern (so wie bei der Repräsentierung auf Arrays) – bei jeder *push* und *pop* Operation muss man die ganze Liste durchlaufen um das letzte Element zu finden
  - Den Top des Stacks am Anfang der Liste zu speichern – bei den *push* und *pop* Operation muss man die Liste nicht durchlaufen
- Also man speichert den Top des Stacks am Anfang der Liste

ADT Stack – Repräsentierung auf DLL

- Wo sollte man den Top des Stacks speichern, um eine gute Effizienz zu erreichen?
- Es gibt zwei Möglichkeiten:
  - Den Top des Stacks am Ende der Liste zu speichern (wie bei der Repräsentierung auf Arrays) – bei der *push* und *pop* Operation muss man die Liste nicht durchlaufen
  - Den Top des Stacks am Anfang der Liste zu speichern – bei der *push* und *pop* Operation muss man die Liste nicht durchlaufen
- Also bei DLL ist es egal an welchem Ende man den Top speichert.

Stack mit fester Kapazität auf einfach verketteten Listen

- Wie kann man einen Stack mit einer festen Kapazität auf einfach verketteten Listen implementieren?

  - Ähnlich wie bei der Implementierung auf statischen Arrays, kann man in der Struktur des Stacks zwei Integer Werte speichern:
    - Die maximale Kapazität
    - Die aktuelle Größe des Stacks (Anzahl der Elemente)

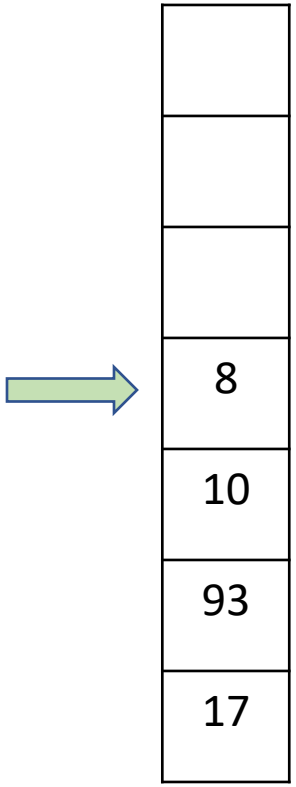
GetMinimum in konstanter Zeit

- Wie kann man einen Stack repräsentieren, sodass die Operation getMinimum $\Theta(1)$ Zeitkomplexität hat (und die anderen Operationen auch $\Theta(1)$ Zeitkomplexität haben) ?

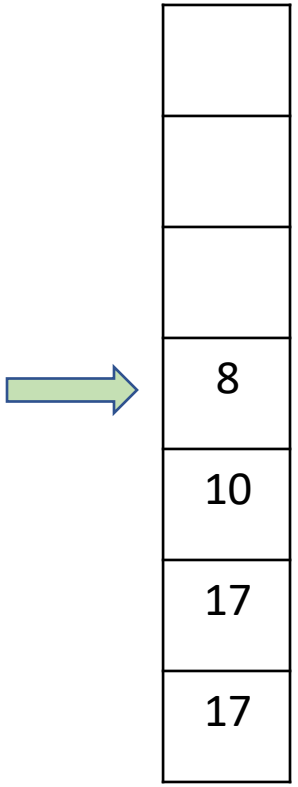
  - Man kann einen Hilfs-Stack benutzen, der die minimale Werte bis zu jedem Element enthält.
  - Wir nennen diesen Stack *min stack* und den ursprünglichen Stack *element stack*

GetMinimum in konstanter Zeit

- *element stack*:

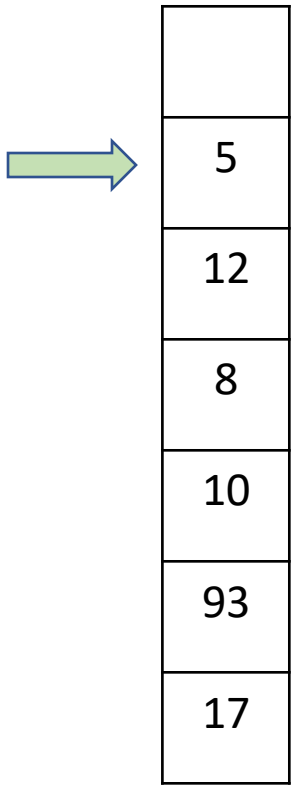


- *min stack*:

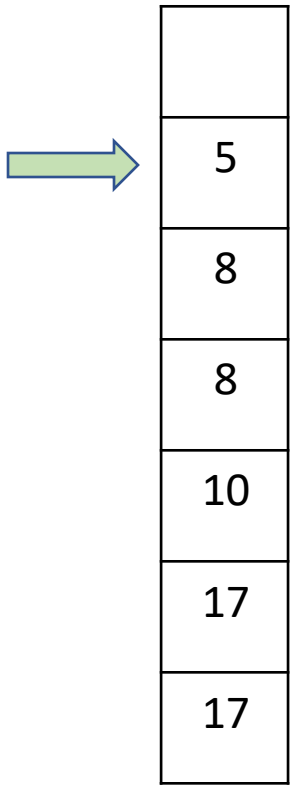


GetMinimum in konstanter Zeit

- Bei der push-Operation muss man in beide Stacks einfügen, aber in den *min stack* fügt man das Minimum zwischen dem Top und dem aktuellen Element
  - *element stack*:



  - *min stack*:



GetMinimum in konstanter Zeit

- Bei der *pop* Operation löscht man ein Element aus beiden Stacks
- Bei der *getMinimum* Operation gibt man den Top der *min stack* zurück
- Die anderen Stack Operationen bleiben unverändert

GetMinimum in konstanter Zeit

- Wie sieht die Implementierung der ***push*** Operation für diesen speziellen Stack aus?

SpecialStack:

elementStack: Stack

minStack: Stack

- Wir benutzen eine der existierenden Implementierungen für Stack und arbeiten mit den Operationen aus der Schnittstelle

```
subalgorithm push(ss, e) is:
    //falls die Stacks voll sein können, dann überprüfe ob sie voll sind
    if isFull(ss.elementStack) then
        @throw overflow (full stack) exception
    end-if
    if isEmpty(ss.elementStack) then  //die Stacks sind leer, push das Element
        push(ss.elementStack, e)
        push(ss.minStack, e)
    else
        push(ss.elementStack, e)
        currentMin ← top(ss.minStack)
        if currentMin < e then  //finde das aktuelle Minimum, das zu minStack eingefügt wird
            push(ss.minStack, currentMin)
        else
            push(ss.minStack, e)
        end-if
    end-if
end-subalgorithm
```

- Komplexität: $\Theta(1)$

SpecialStack – Denke nach

- SpecialStack wurde in solcher Weise entworfen, dass alle Operationen $\Theta(1)$ Zeitkomplexität haben
- Nachteil: man braucht doppelt so viel Platz, wie bei dem normalen Stack

- Wie könnte man den Speicherplatz für *min stack* reduzieren (vor allem, wenn das minimale Element des Stacks selten geändert wird)?
- Hinweis: wenn das Minimum dasselbe bleibt, dann muss man kein neues Element hinzufügen
- Wie werden push und pop implementiert?
- Was passiert wenn das minimale Element mehrmals in dem Stack vorkommt?

ADT Queue (Schlange) - Wiederholung

- ADT Schlange ist ein Container, in welchem die Elemente nur an einem Ende (dem Ende der Schlange) eingefügt (push/enqueue) und nur am anderen Ende (dem Kopf der Schlange) entfernt (pop/dequeue) werden können.

- Es gilt also das FIFO Prinzip (First-in-First-out)

ADT Queue – Repräsentierung

- Um ADT Schlange zu implementieren kann man folgende Datenstrukturen für die Repräsentierung benutzen:
  - Statische Arrays
  - Dynamische Arrays – zirkuläre Arrays (schon besprochen)
  - Einfach verkettete Listen
  - Doppelt verkettete Listen

- Wo sollte man den Anfang und das Ende der Schlange für jede Repräsentierung speichern?

ADT Queue – Repräsentierung auf SLL

- Wo sollte man den Anfang und das Ende der Schlange in einer einfach verketteten Liste speichern?

- Theoretisch gibt es zwei Möglichkeiten:
  - Der Anfang der Schlange am Anfang der Liste und das Ende der Schlange am Ende der Liste speichern
  - Der Anfang der Schlange am Ende der Liste und das Ende der Schlange am Anfang der Liste speichern
- In beiden Fällen hat eine der Operationen *push* oder *pop* Komplexität $\Theta(n)$

ADT Queue – Repräsentierung auf SLL

- Man kann die Komplexität der Operationen verbessern, wenn man sowohl den Head als auch den Tail der Liste speichert (auch für ein SLL)

- Wo sollte man *front* und *rear* der Schlange speichern: im Head oder im Tail?

ADT Queue – Repräsentierung auf DLL

- Wo sollte man den Anfang und das Ende der Schlange in dem DLL speichern?

- Theoretisch gibt es zwei Möglichkeiten:
  - Der Anfang der Schlange am Anfang der Liste und das Ende der Schlange am Ende der Liste speichern
  - Der Anfang der Schlange am Ende der Liste und das Ende der Schlange am Anfang der Liste speichern

ADT Prioritätsschlange (Wiederholung)

- ADT Prioritätsschlange ist ein Container, in welchem jedes Element eine zugewiesene Priorität hat

- In einer Prioritätsschlange hat man Zugriff nur auf das Element mit der höchsten Priorität

- Es gilt also das **HPF Prinzip (Highest Priority First)**

ADT Prioritätsschlange - Repräsentierung

- Um ADT Prioritätsschlange zu implementieren kann man folgende Datenstrukturen für die Repräsentierung benutzen:
  - Dynamisches Array
  - Verkettete Liste
  - (Binärer) Heap

ADT Prioritätsschlange - Repräsentierung

- Falls die Repräsentierung ein dynamisches Array oder eine verkettete Liste ist, dann muss man entscheiden, wie die Elemente gespeichert werden:
  - Man kann die Elemente sortiert nach der Priorität speichern
    - Wo sollte man das Element mit der höchsten Priorität speichern?
  - Man kann die Elemente in der Reihenfolge, in der sie eingefügt wurden, speichern

ADT Prioritätsschlange - Repräsentierung

- Komplexität der Operationen für die zwei Repräsentierungs-Möglichkeiten:

Operation	**Sortiert**	**Nicht-sortiert**
push	O(n)	$\Theta(1)$
pop	$\Theta(1)$	$\Theta(n)$
top	$\Theta(1)$	$\Theta(n)$

- Was passiert wenn man in einem separaten Feld das Element mit der höchsten Priorität speichert?

ADT Deque

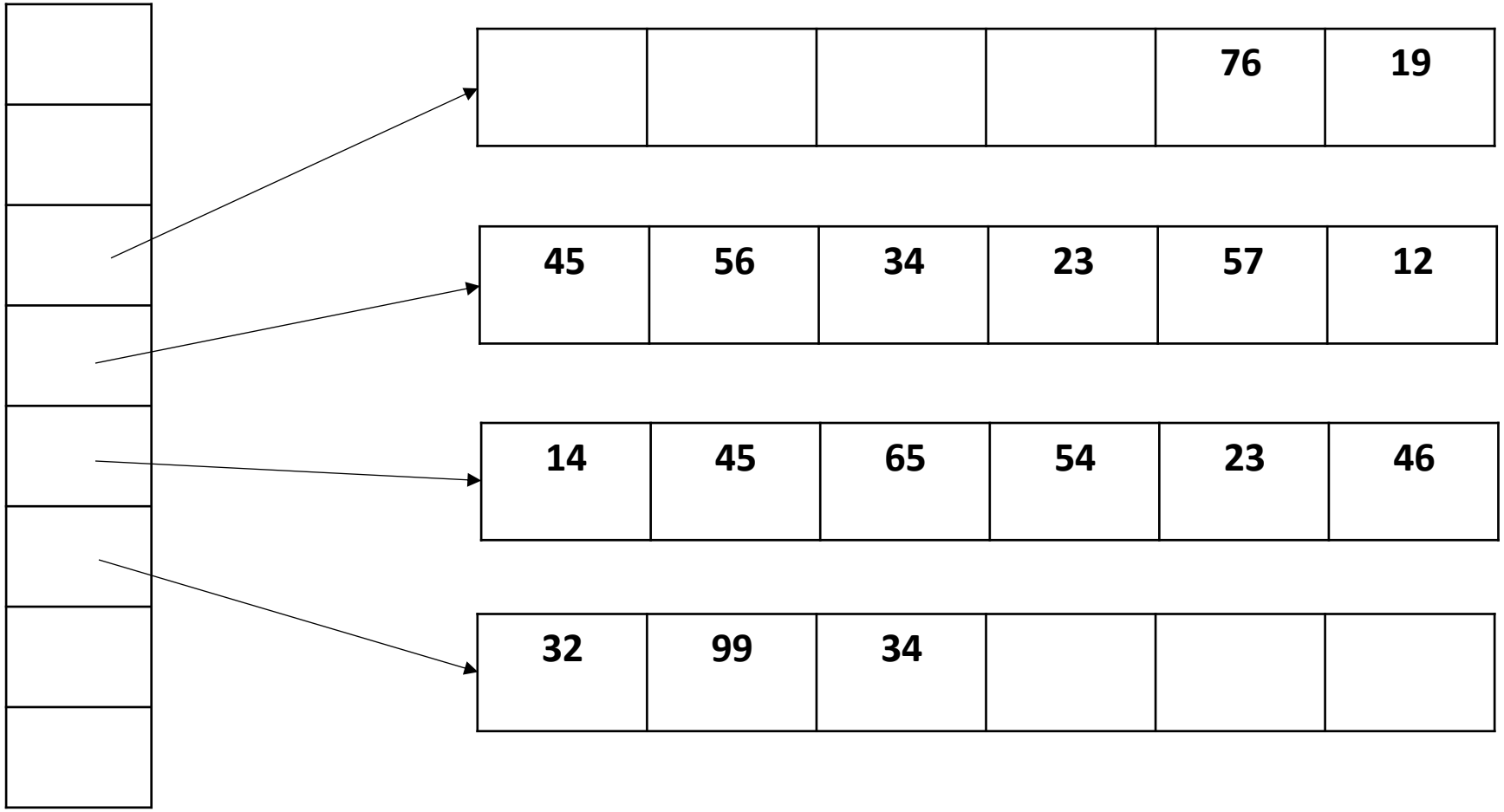
- ADT Deque (Double ended queue) ist ein Container, wo die Daten an beiden Enden gelesen, eingefügt oder entfernt werden können:
  - *push_front* und *push_back*
  - *pop_front* und *pop_back*
  - *top_front* und *top_back*

- Man kann ein Stack oder eine Schlange mit einer Deque simulieren, wenn man nicht alle Operationen braucht

ADT Deque

- Mögliche Repräsentierungen für Deque:
  - Zirkuläres Array
  - Doppelt verkettete Liste
  - **Dynamisches Array von Arrays** mit konstanter Größe

ADT Deque - Beispiel



- Die Elemente der Deque sind: 76, 19, 45, ..., 12, 14, ..., 46, 32, 99, 34

ADT Deque - Repräsentierung

- Eine interessante Repräsentierung für Deque ist **ein dynamisches Array von Arrays mit konstanter Kapazität**:
  - Man speichert die Elemente in **Arrays mit konstanter Kapazität** (Blocks)
  - In **einem dynamischen Array** speichert man **die Adressen der Blocks**
  - Alle Blocks außer dem ersten und dem letzten sind voll
  - Der erste Block wir von rechts nach links ausgefüllt
  - Der letzte Block wird von links nach rechts ausgefüllt
  - Falls der erste oder letzte Block voll ist, dann erstellt man einen neuen Block und man speichert die Adresse in dem dynamischen Array
  - Falls das dynamische Array voll ist, dann wird seine Kapazität vergrößert (man allokiert einen größeren Speicherplatz und die Adressen der Blocks werden **kopiert, aber nicht geändert**)

ADT Deque - Beispiel

- Informationen, die man speichern muss für die Repräsentierung auf ein dynamisches Array von Blocks:
  - Blockgröße
  - Das dynamische Array von Blockadressen
  - Die Kapazität des dynamischen Arrays
  - Die erste besetzte Position in dem dynamischen Array
  - Die erste besetzte Position in dem ersten Block
  - Die letzte besetzte Position in dem dynamischen Array
  - Die letzte besetzte Position in dem letzten Block

  - Die letzten zwei Felder müssen nicht unbedingt gespeichert werden, wenn man die Anzahl der Elemente aus dem Deque kennt

Verkettete Listen auf Arrays

- Was wäre wenn man eine verkettete Liste braucht, aber man arbeitet in einer Programmiersprache, die keine Pointers oder Referenzen hat?

- Man kann verkettete Datenstrukturen implementieren, ohne explizite Pointers oder Referenzen zu benutzen, indem man diese mit Hilfe von Arrays und Indexe simuliert werden.

Verkettete Listen auf Arrays

- Wenn man normalerweise mit einem Array arbeitet, dann werden die Elemente nacheinander gespeichert anfangend von ganz links (keine leeren Plätze in der Mitte sind erlaubt)
- Die Reihenfolge der Elemente in einem Array ist gleich mit der Reihenfolge in dem diese in dem Array gespeichert werden

elems

46	**78**	**11**	**6**	**59**	**19**				

- Reihenfolge der Elemente: 46, 78, 11, 6, 59, 19

Verkettete Listen auf Arrays

- Man kann eine verkettete Datenstruktur auf einem Array definieren, wenn man annimmt, dass die Reihenfolge der Elemente nicht von der relativen Position in dem Array gegeben wird, sondern von einer zugehörigen ganzen Zahl, die den Index des nächsten Elementes angibt

elems	46	78	11	6	59	19				
next	5	6	1	-1	2	4				
	1	2	3	4	5	6	7	8	9	10

head = 3

- Reihenfolge der Elemente: 11, 46, 59, 78, 19, 6

- Wie kann man das Element 46 löschen?

Verkettete Listen auf Arrays

- Wenn man jetzt das Element 46 löschen will (eigentlich das zweite Element aus der Liste), dann muss man nicht alle Elemente aus dem Array nach link verschieben, sondern man muss nur die Links ändern

elems		78	11	6	59	19				
next		6	5	-1	2	4				

head = 3

- Reihenfolge der Elemente: 11, 59, 78, 19, 6

- Wie kann man das Element 44 an die Position 3 einfügen?

Verkettete Listen auf Arrays

- Wenn man jetzt das Element 44 an die Position 3 einfügen will, dann kann man das Element an irgendeiner Position in dem Array speichern und dann die Links richtig ändern

elems		78	11	6	59	19		44		
next		6	5	-1	8	4		2		

head = 3

- Reihenfolge der Elemente: 11, 59, 44, 78, 19, 6

Verkettete Listen auf Arrays

- Bei dem Einfügen eines neuen Elementes, kann dieses an einer leeren Position in dem Array gespeichert werden. Um aber eine leere Position zu finden ist die Komplexität O(n), d.h. die Komplexität für die Einfüge Operation wird auch O(n) sein (egal wo man einfügen will).
- Um das zu vermeiden speichert man auch eine verkettete Liste von leeren Positionen.

elems		78	11	6	59	19		44		
next	7	6	5	-1	8	4	9	2	10	-1
	1	2	3	4	5	6	7	8	9	10

head = 3

firstEmpty = 1

Verkettete Listen auf Arrays

- Um also eine einfach verkettete Liste auf Arrays zu simulieren braucht man:
  - Ein Array wo man die Elemente speichert
  - Ein Array wo man die Links als Indexe zum nächsten Element speichert
  - Die Kapazität der zwei Arrays (diese haben dieselbe Kapazität, also braucht man den Wert nur einmal zu speichern)
  - Ein Index der den Listenkopf angibt
  - Ein Index zur ersten leeren Position in dem Array

SLL auf Arrays – Repräsentierung

- Die Repräsentierung einer einfach verketteten Liste auf einem Array (SLLA) ist:

SLLA:
elems: TElem[]
next: Integer[]
capacity: Integer
head: Integer
firstEmpty: Integer

SLLA - Operationen

- Man kann alle Operationen, die es für SLL gibt, auch für SLLA implementieren:
  - Suche ein Element mit einem gegebenen Wert
  - Füge ein Element ein: am Anfang der Liste, am Ende der Liste, auf eine gegebene Position, vor/nach einem bestimmten Wert
  - Lösche ein Element: vom Anfang der Liste, vom Ende der Liste, an einer bestimmten Position, mit einem gegebenen Wert
  - Gib ein Element an einer bestimmten Position zurück

SLLA - init

```
subalgorithm init(slla) is:
//pre: true; post: slla ist ein leeres SLLA
    slla.cap ← INIT_CAPACITY
    slla.elems ← @an array with slla.cap positions
    slla.next ← @an array with slla.cap positions
    slla.head ← -1
    for i ← 1, slla.cap-1 execute
        slla.next[i] ← i + 1
    end-for
    slla.next[slla.cap] ← -1
    slla.firstEmpty ← 1
end-subalgorithm
```

- Komplexität:

  $\Theta(n)$

SLLA - search

```
function search (slla, elem) is:
//pre: slla ist ein SLLA, elem ist ein TElem
//post: gibt true zurück falls elem in slla ist, false ansonsten
    current ← slla.head
    while current ≠ -1 and slla.elems[current] ≠ elem execute
        current ← slla.next[current]
    end-while
    if current ≠ -1 then
        search ← True
    else
        search ← False
    end-if
end-function
```

- Komplexität:

  $O(n)$

SLLA - search

- In der *search* Funktion sieht man wie eine SLLA durchlaufen werden kann (ähnlich wie ein SLL):
  - Man braucht ein *current* Element, das mit dem *Head* initialisiert wird
  - Man hört auf wenn *current* -1 wird (also es zeigt zu keinem gültigen Element mehr)
  - Man geht zu dem nächsten Element mit der Anweisung:

    current ← *slla.next[current]*

SLLA - insertFirst

```
subalgoritm insertFirst (slla, elem) is:
//pre: slla ist ein SLLA, elem ist ein TElem
//post: das Element elem wird am Anfang von slla eingefügt
    if slla.firstEmpty = -1 then
        newElems ← @an array with slla.cap * 2 positions
        newNext ← @an array with slla.cap * 2 positions
        for i ← 1, slla.cap execute
            newElems[i] ← slla.elems[i]
            newNext[i] ← slla.next[i]
        end-for
        for i ← slla.cap + 1, slla.cap*2 - 1 execute
            newNext[i] ← i + 1
        end-for
        newNext[slla.cap*2] ← -1
//Fortsetzung auf der nächsten Folie ...
```

SLLA - insertFirst

```
        //deallokiere slla.elems und slla.next falls nötig
        slla.elems ← newElems
        slla.next ← newNext
        slla.firstEmpty ← slla.cap+1
        slla.cap ← slla.cap * 2
    end-if
    newPosition ← slla.firstEmpty
    slla.elems[newPosition] ← elem
    slla.firstEmpty ← slla.next[slla.firstEmpty]
    slla.next[newPosition] ← slla.head
    slla.head ← newPosition
end-subalgorithm
```

Komplexität:

$\Theta(1)$ amortisiert

SLLA - insertPosition

```
subalgorithm insertPosition(slla, elem, pos) is:
//pre: slla ist ein SLLA, elem ist ein TElem, pos ist eine ganze Zahl
//post: das Element elem wird in slla an der Position pos eingefügt
    if pos < 1 then
        @error, invalid position
    end-if
    if slla.firstEmpty = -1 then
        @resize
    end-if
    newElem ← slla.firstEmpty
    slla.firstEmpty ← slla.next[firstEmpty]
    slla.elems[newElem] ← elem
    slla.next[newElem] ← -1
//Fortsetzung auf der nächsten Folie ...
```

SLLA - insertPosition

```
if pos = 1 then
    if slla.head = -1 then
        slla.head ← newElem
    else
        slla.next[newElem] ← slla.head
        slla.head ← newElem
    end-if
else
    currentPos ← 1
    currentNode ← slla.head
    while currentNode ≠ -1 and currentPos < pos - 1 execute
      currentPos ← currentPos + 1
      currentNode ← slla.next[currentNode]
    end-while
```

//Fortsetzung auf der nächsten Folie ...

SLLA - insertPosition

```
        if currentNode ≠ -1 then
            slla.next[newElem] ← slla.next[currentNode]
            slla.next[currentNode] ← newElem
        else
            @error, invalid position
        end-if
    end-if
end-subalgorithm
```

Komplexität: $O(n)$

SLLA - insertPosition

- Bemerkungen:
  - Die *resize* Operation ist gleich mit der *resize* Operation von *addToBeginning*
  - Ähnlich wie bei SLL, muss man die Liste iterieren bis man das Element gefunden hat, **nach** welchem man einfügen muss (*currentNode*)
  - Ein Ausnahmefall ist das Einfügen an der ersten Position (es gibt keinen Knoten nach welchem man einfügen muss)

SLLA – deleteElement

```
subalgorithm deleteElement (slla, elem) is:
//pre: slla ist ein SLLA; elem ist ein TElem
//post: das Element elem wird aus SLLA gelöscht
    nodC ← slla.head
    prevNode ← -1
    while nodC ≠ -1 and slla.elems[nodC] ≠ elem execute
        prevNode ← nodC
        nodC ← slla.next[nodC]
    end-while
    if nodC ≠ -1 then
        if nodC = slla.head then
            slla.head ← slla.next[slla.head]
        else
            slla.next[prevNode] ← slla.next[nodC]
        end-if
//Fortsetzung auf der nächsten Folie ...
```

SLLA - deleteElement

```
//nodC Position wird zu der Liste von leeren Positionen hinzugefügt
        slla.next[nodC] ← slla.firstEmpty
        slla.firstEmpty ← nodC
    else
        @the element does not exist
    end-if
end-subalgorithm
```

Komplexität: $O(n)$

SLLA – Iterator

- Der Iterator für SLLA ist eine Mischung aus dem Iterator für ein Array und dem Iterator für eine einfach verkettete Liste
- Da die Elemente in einem Array gespeichert sind, ist das *currentElement* ein Index aus dem Array
- Aber, da es sich um eine verkettete Liste handelt, kann man nicht mit einer Inkrementierung von *currentElement* zu dem nächsten Element gehen, sondern man muss die *next* Links folgen
- Womit wird *currentElement* initialisiert*?*
  - Mit der Position des Heads (nicht mit 1!)