Datenstrukturen und Algorithmen

Vorlesung 5

Überblick

- Vorige Woche:
  - ADT Prioritätsschlange
  - ADT Stack
  - ADT MultiMap & SortedMultiMap
  - ADT IndexedList & IteratedList
  - Einfach verkettete Listen (SLL) - intro
- Heute betrachten wir:
  - Einfach verkettete Listen - Iterator
  - Doppelt verkettete Listen (DLL)
  - Sortierte Listen
  - Arrays und Listen – Zusammenfassung
  - Zirkuläre Listen

Einfache Verkettete Listen - Repräsentierung

- Für die Repräsentierung einer einfach verketteten Liste (SLL) braucht man zwei Datenstrukturen:
  - Eine Datenstruktur für die Knoten
  - Eine Datenstruktur für die Liste

SLLNode:
    info: TElem //die eigentliche Daten
    next: ↑ SLLNode //die Adresse des Nachfolgers

SLL:
    head: ↑ SLLNode //die Adresse des ersten Knotens

SLL – Iterator

- Wie kann man einen Iterator für ein SLL definieren?
- Ein Iterator braucht eine Referenz zu dem *aktuellen* Element aus dem Datenstruktur, die er iteriert
- Wie kann man das *aktuelle* Element einer Liste repräsentieren?

SLL – Iterator

- Für ein SLL ist das aktuelle Element aus dem Iterator ein Zeiger auf einem Knoten der Liste:

SLLIterator:
    list: SLL
    currentElement: ↑SLLNode

SLL – Iterator – init Operation

- Was sollte die *init* Operation tun?

```
subalgorithm init(it, sll) is:
//pre: sll ist ein SLL
//post: it ist ein SLLIterator für sll
        it.sll ← sll
        it.currentElement ← sll.head
end-subalgorithm
```

- Komplexität: $\Theta(1)$

SLL – Iterator – getCurrent Operation

- Was sollte die *getCurrent* Operation tun?

```
function getCurrent(it) is:
//pre: it ist ein SLLIterator, it ist gültig
//post: e ist ein TElem, e ist das aktuelle Element aus it
        if it.currentElement = NIL then
            @throw ein Exception
        end-if
        e ← [it.currentElement].info
        getCurrent ← e
end-function
```

- Komplexität: $\Theta(1)$

SLL – Iterator – next Operation

- Was sollte die *next* Operation tun?

```
subalgorithm next(it) is:
//pre: it ist ein SLLIterator, it ist gültig
//post: it' ist ein SLLIterator, das aktuelle Element aus it' verweist
//auf das nächste Element
//throws: Exception falls it nicht gültig
    if it.currentElement = NIL then
            @throw ein Exception
    end-if
    it.currentElement ← [it.currentElement].next
end-subalgorithm
```

- Komplexität: $\Theta(1)$

SLL – Iterator – valid Operation

- Was sollte die *valid* Operation tun?

```
function valid(it) is:
//pre: it ist ein SLLIterator
//post: wahr falls it gültig ist, falsch ansonsten
    if it.currentElement ≠ NIL then
            valid ← True
    else
            valid ← False
    end-if
end-function
```

- Komplexität: $\Theta(1)$

Denk darüber nach

- Wie würdet ihr einen bidirektionalen Iterator für ein SLL definieren?

  Welche wäre die Komplexität der *previous* Operation?

- Wie würdet ihr einen bidirektionalen Iterator für ein SLL definieren, wenn man zusätzlich weiß, dass die *previous* Operation nicht zweimal aufeinander aufgerufen wird (zwischen zwei *previous* Operationen gibt es wenigstens eine *next* Operation)?

  Welche wäre die Komplexität der *previous* Operation?

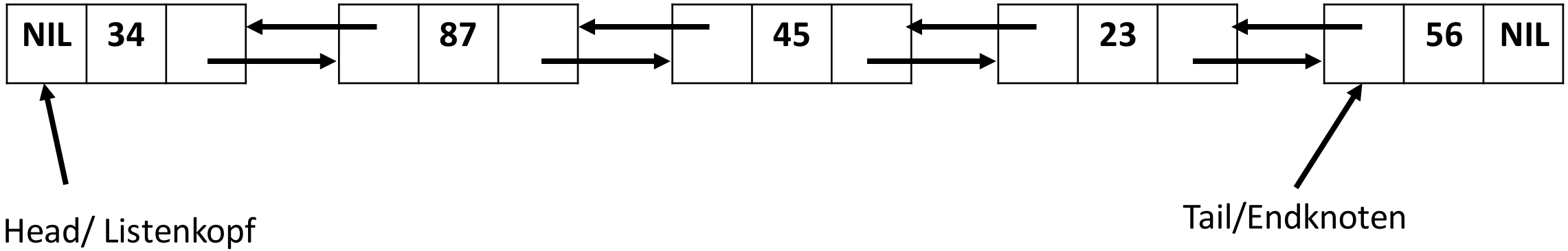
Doppelt verkettete Listen - DLL

- Eine doppelt verkettete Liste ähnelt der einfach verketteten Liste, aber die Knoten enthalten zusätzlich auch die Adresse des Vorgängers (außer dem *next* Link, gibt es auch einen *prev* Link)

- Aus einem Knoten der DLL kann man zu dem nächsten oder zu dem vorigen Knoten gehen (man kann die Elemente der DLL in beiden Richtungen durchlaufen)

- Der *prev* Link des ersten Knotens hat den Wert *NIL* (sowie der *next* Link des letzten Knotens)

Doppelt verkettete Listen – Beispiel



Doppelt verkettete Listen - Repräsentierung

- Für die Repräsentierung einer doppelt verketteten Liste (DLL) braucht man zwei Datenstrukturen:
  - Eine Datenstruktur für die Knoten
  - Eine Datenstruktur für die Liste

DLLNode:

```
info: TElem          //die eigentliche Daten
next: ↑ DLLNode      //die Adresse des Nachfolgers
prev: ↑ DLLNode      //die Adresse des Vorgängers
```

DLL:

```
head: ↑ DLLNode    //die Adresse des ersten Knotens
tail: ↑ DLLNode    //die Adresse des letzten Knotens
```

DLL – Operationen

- Ein DLL kann dieselben Operationen haben wie ein SLL:
  - Suche ein Element mit einem gegebenen Wert
  - Füge ein Element ein: am Anfang der Liste, am Ende der Liste, auf einer gegebenen Position, 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
- Die meisten Operationen haben dieselbe Implementierung wie bei SLL (search, getElement) oder eine ähnliche Implementierung
- Bei dem DLL muss man meistens mehrere Links ändern und man muss auch auf den Tail Knoten aufpassen

DLL – eine leere Liste erstellen

- Ein leere Liste hat keine Knoten, also die Adresse der ersten Knotens ist NIL

```
subalgorithm init(dll) is:
//pre: wahr
//post: dll ist ein DLL
    dll.head ← NIL
    dll.tail ← NIL
end-subalgorithm
```

- Komplexität: $\Theta(1)$

DLL – am Ende der Liste einfügen

- In einem DLL ist es einfach ein neues Element am Ende der Liste einzufügen:
  - da es den *tail* gibt, muss man die Liste nicht mehr durchlaufen

- Welche Sonderfälle müssen separat behandelt werden?

DLL – am Ende der Liste einfügen

```
subalgorithm insertLast(dll, elem) is:
//pre: dll ist ein DLL, elem ist ein TElem
//post: elem wir am Ende von dll eingefügt
        newNode ← allocate()   //einen neuen DLLNode allokieren
        [newNode].info ← elem
        [newNode].next ← NIL
        [newNode].prev ← dll.tail
        if dll.head = NIL then     //die Liste ist leer
                dll.head ← newNode
                dll.tail ← newNode
        else
                [dll.tail].next ← newNode
                dll.tail ← newNode
        end-if
end-subalgorithm
```

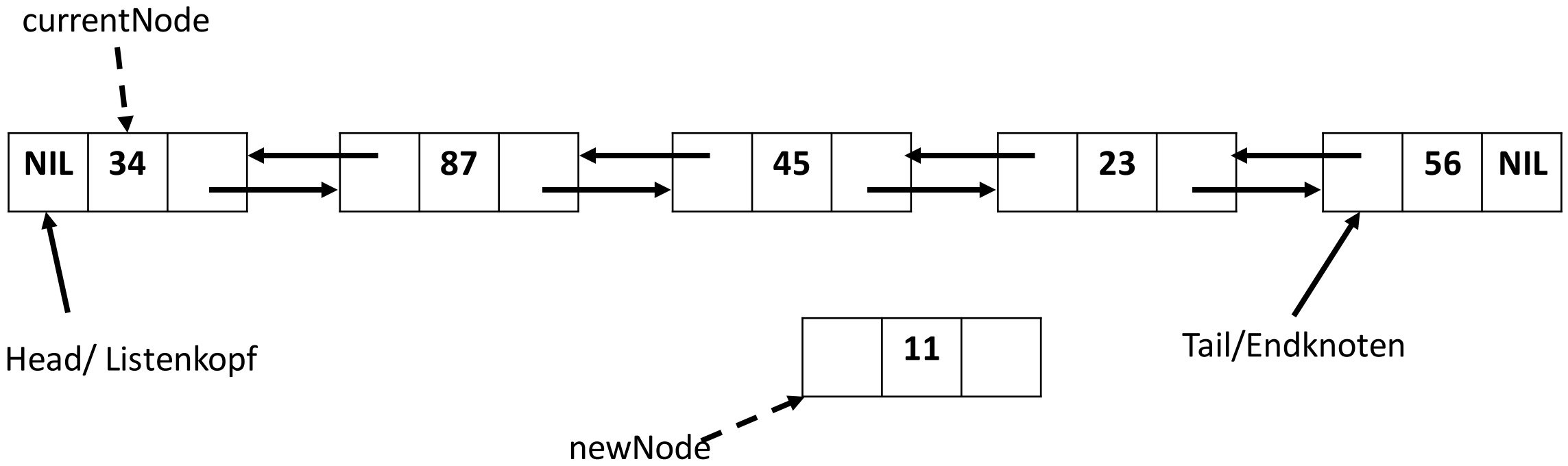
- Komplexität: $\Theta(1)$

DLL - an eine bestimmte Position einfügen

- Die Idee bei dem Einfügen an einer bestimmten Position ist die gleiche wie bei SLL
- Der Unterschied besteht darin, dass man mehrere Links festlegen muss und dass man zusätzlich überprüfen muss ob sich der Tail der Liste ändert
- Für ein SLL musste man bei dem Knoten aufhalten, nach welchem die Einfügung stattfinden muss
- Für ein DLL kann man entweder bei dem Knoten davor oder bei dem Knoten danach aufhalten, aber man muss das vorher entscheiden (die Sonderfälle, die man betrachten muss, unterscheiden sich)

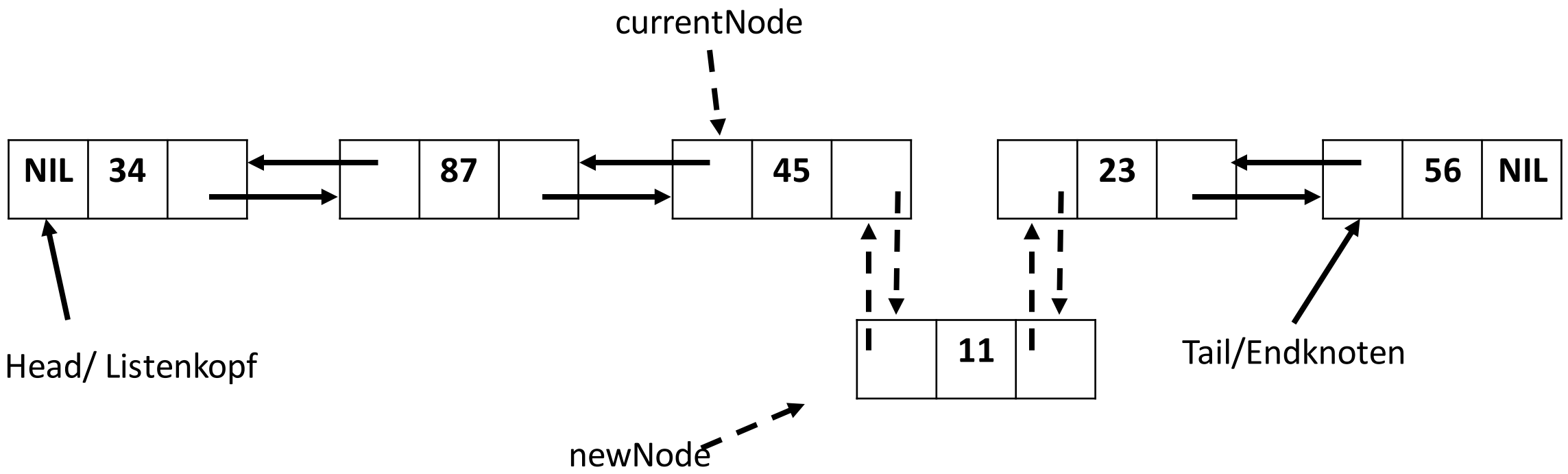
DLL - an eine bestimmte Position einfügen

- Man fügt den Wert 11 an die vierte Position ein:



DLL - an eine bestimmte Position einfügen

- Man fügt den Wert 11 an die vierte Position ein:



DLL - an eine bestimmte Position einfügen

```
subalgorithm insertPosition(dll, pos, elem) is:
//pre: dll ist ein DLL; pos ist eine ganze Zahl; elem ist ein TElem
//post: elem wird an die Position pos in dll eingefügt
    if pos < 1 then
        @ error, invalid position
    else if pos = 1 then
        insertFirst(dll, elem)
    else
        currentNode ← dll.head
        currentPos ← 1
        while currentNode ≠ NIL and currentPos < pos - 1 execute
            currentNode ← [currentNode].next
            currentPos ← currentPos + 1
        end-while
//Fortsetzung auf der nächsten Folie ...
```

DLL - an eine bestimmte Position einfügen

```
        if currentNode = NIL then
            @error, invalid position
        else if currentNode = dll.tail then
            insertLast(dll, elem)
        else
            newNode ← alocate()
            [newNode].info ← elem
            [newNode].next ← [currentNode].next
            [newNode].prev ← currentNode
            [[currentNode].next].prev ← newNode
            [currentNode].next ← newNode
        end-if
    end-if
end-subalgorithm
```

- Komplexität: *O(n)*

DLL - an eine bestimmte Position einfügen

- Bemerkungen:
  - Die Operation *insertFirst* haben wir für DLL nicht implementiert (man vermutet aber, dass sie implementiert ist)
  - Die **Reihenfolge**, in der man die **Links festlegt** ist wichtig: wenn man die letzten zwei Zuweisungen vertauscht, führt das zu einem Problem in der Liste
  - Eine Möglichkeit wäre auch zwei *currentNodes* zu benutzen:

```
nodeBefore ← currentNode
nodeAfter ← [currentNode].next
//man fügt das neue Element zwischen nodeBefore und nodeAfter
[newNode].next ← nodeAfter
[newNode].prev ← nodeBefore
[nodeAfter].prev ← newNode
[nodeBefore].next ← newNode
```

DLL - vom Anfang der Liste löschen

```
function deleteFirst(dll) is:
//pre: dll ist ein DLL
//post: der erste Knoten aus dll wurde gelöscht und zurückgegeben
    deletedNode ← NIL
    if dll.head ≠ NIL then
        deletedNode ← dll.head
        if dll.head = dll.tail then
            dll.head ← NIL
            dll.tail ← NIL
        else
            dll.head ← [dll.head].next
            [dll.head].prev ← NIL
        end-if
        @man stellt die Links von deletedNode auf NIL
    end-if
    deleteFirst ← deletedNode
end-function
```

DLL - vom Anfang der Liste löschen

- Komplexität für die Funktion *deleteFirst:*

  $\Theta(1)$

DLL – einen gegebenen Wert löschen

- Wenn man einen Knoten mit einem gegebenen Wert löschen muss, dann muss man erstmal den Knoten finden:
  - man muss die Liste durchlaufen bis man den Knoten findet
  - wenn man den Knoten findet dann löscht man den Knoten und man ändert die Links
  - Sonderfälle:
    - Element nicht in der Liste
    - Lösche Head
    - Lösche Tail
    - Lösche Head, das auch Tail ist (d.h. die Liste hat nur ein Element)

```
function deleteElement(dll, elem) is:
//pre: dll ist ein DLL, elem ist ein TElem
//post: der Knoten mit dem Wert elem wird gelöscht und zurückgegeben
    currentNode ← dll.head
    while currentNode ≠ NIL and [currentNode].info ≠ elem execute
        currentNode ← [currentNode].next
    end-while
    deletedNode ← currentNode
    if currentNode ≠ NIL then
        if currentNode = dll.head then  //man löscht den ersten Knoten
            if currentNode = dll.tail then  //man löscht den letzten Knoten
                dll.head ← NIL
                dll.tail ← NIL
            else  //die Liste hat mehrere Elemente
                dll.head ← [dll.head].next
                [dll.head].prev ←  NIL
            end-if
        else if currentNode = dll.tail then
     //Fortsetzung auf der nächsten Folie ...
```

```
            dll.tail ← [dll.tail].prev
            [dll.tail].next ← NIL
        else
            [[currentNode].next].prev ← [currentNode].prev
            [[currentNode].prev].next ← [currentNode].next
            @ man stellt die Links von deletedNode auf NIL
        end-if
    end-if
    deleteElement ← deletedNode
end-function
```

- Komplexität: *O(n)*
- Wenn man anstatt die *search* Funktion benutzt, dann ist die Komplexität auch *O(n):* das Löschen wäre Θ(1), aber das Suchen ist *O(n)*

DLL -Iterator

- Der Iterator für DLL ist identisch mit dem Iterator für SLL (aber *currentNode* ist ein *DLLNode* und nicht ein *SLLNode*)

- Für DLL ist es einfach einen bidirektionalen Iterator zu definieren:
  - Man braucht zusätzlich die Operation *previous*

  - Manchmal ist es nützlich zwei zusätzliche Operationen zu definieren, *first* und *last*, um den Iterator mit dem Head oder Tail initialisieren zu können

Dynamische Arrays vs verkettete Listen

- Ähnlich wie bei dem Dynamischen Array gibt es zwei Möglichkeiten die Elemente einer verketteten Liste zu durchlaufen:
  - Mit einem Iterator
  - Mit einer Schleife wo getElement abgerufen wird

- Welche ist die Komplexität für die zwei Ansätze?

Dynamische Arrays vs. Verkettete Listen

- Dynamische Arrays und Verkettete Listen haben im Allgemeinen dieselben Operationen, aber diese können unterschiedliche Zeitkomplexitäten haben

Algorithmus	**DA**	**SLL**	**DLL**
suchen	O(n)	O(n)	O(n)
gebe Element an einer Position zurück	$\Theta(1)$	O(n)*	O(n)*
an der ersten Position einfügen	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
an der letzten Position einfügen	$\Theta(1)$	O(n)**	$\Theta(1)$
an einer bestimmten Position einfügen	O(n)	O(n)	O(n)
Element an der ersten Position löschen	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Element an der letzten Position löschen	$\Theta(1)$	$\Theta(n)$***	$\Theta(1)$
Element an einer bestimmten Position löschen	O(n)	O(n)	O(n)

Dynamische Arrays vs. Verkettete Listen

- Bemerkungen:
  - * – um das Element an der Position *i* in einer verketteten Liste zurückzugeben, braucht man eigentlich genau *i* Schritte (Komplexität Θ(i)); da aber $i \leq n$, benutzen wir meistens O(n)
  - ** – ein Element an der letzten Position in einem SLL einfügen kann die Komplexität $\Theta(1)$ haben, falls man auch die Adresse des Tails speichert
  - *** – falls man auch die Adresse des Tails in dem SLL speichert, hilft uns das für das Löschen des letzten Elementes?

Dynamische Arrays vs. Verkettete Listen

- Vorteile der verketteten Listen:
  - Kein Speicherplatz wird umsonst benutzt (für Elemente, die noch nicht existieren)
  - Für Operationen am Anfang der Liste wird konstante Zeit benötigt
  - Elemente werden nie *kopiert* (das Kopieren eines Elementes kann viel Zeit dauern)

- Nachteile der verketteten Listen:
  - Man hat keinen direkten Zugriff auf das Element an einer bestimmten Position (aber die Liste durchlaufen mit einem Iterator hat die Komplexität $\Theta(n)$)
  - Man braucht zusätzlichen Speicherplatz für die Adressen in den Knoten
  - Die Knoten besetzen nicht aufeinanderfolgende Speicherplätze (man kann nicht von modernen CPU Caching Methoden profitieren)

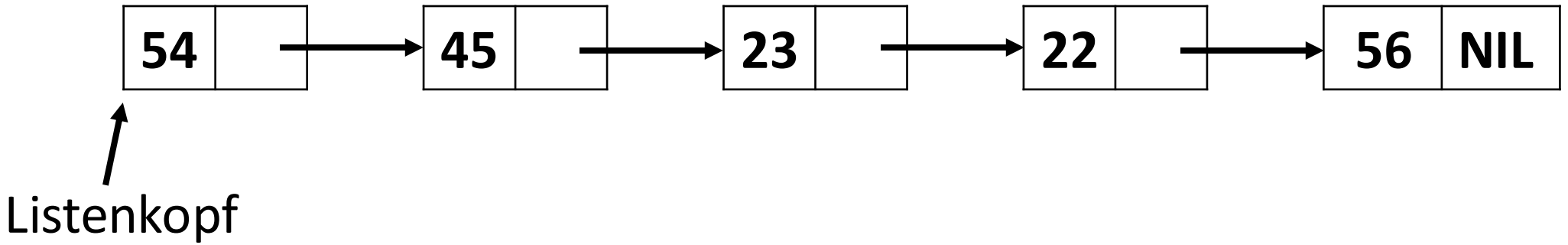
Algorithmische Aufgaben mit verketteten Listen

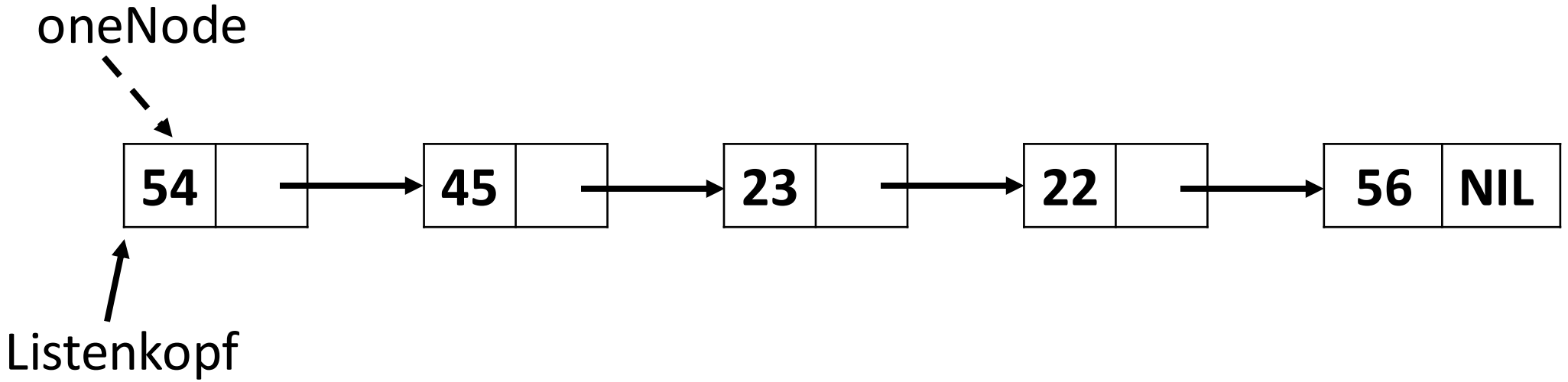
- Finde den n-ten Knoten am Ende einer SLL.

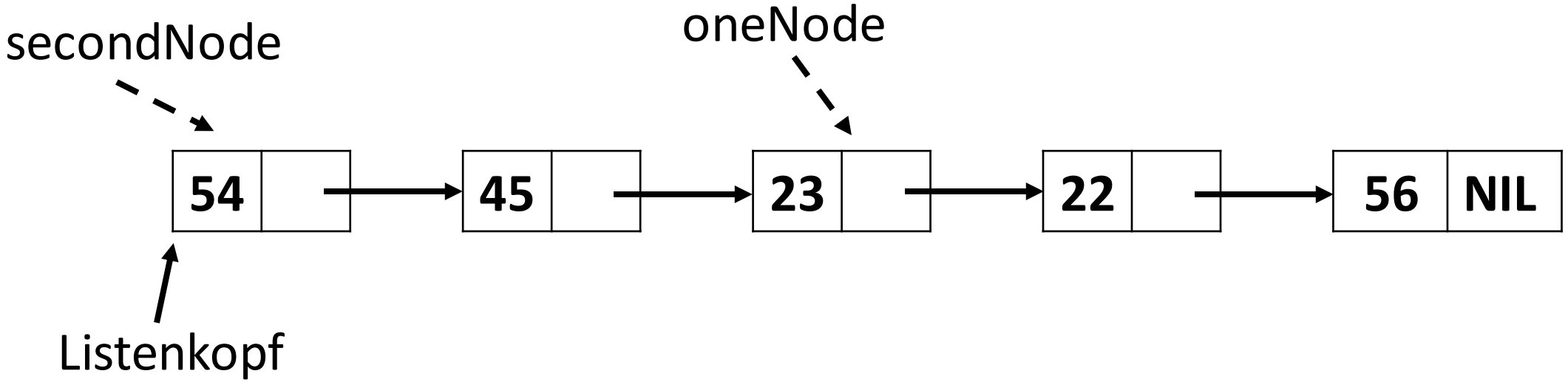
- Einfacher Ansatz:
  - Iteriere durch die Elemente um diese zu zählen
  - Nachdem man die Anzahl der Elemente kennt, kann man die Position (vom Anfang der Liste) des Knotens berechnen
  - Iteriere durch die Liste bis zu dem gesuchten Knoten
- Kann man den Knoten finden indem die Liste nur einmal iteriert wird?
  - Man kann zwei Hilfsvariablen benutzen (zwei Knoten), welche auf dem Head gesetzt werden. Dann geht man mit einer der Variablen n - 1 Positionen vorwärts.
  - Jetzt fängt man damit an, beide Knoten auf einmal mit einer Position nach vorwärts zu verschieben.
  - Wenn der erste Knoten am Ende der Liste ankommt, dann ist der zweite Knoten auf dem n-ten Knoten am Ende der Liste.
- Ist der zweite Ansatz effizienter?

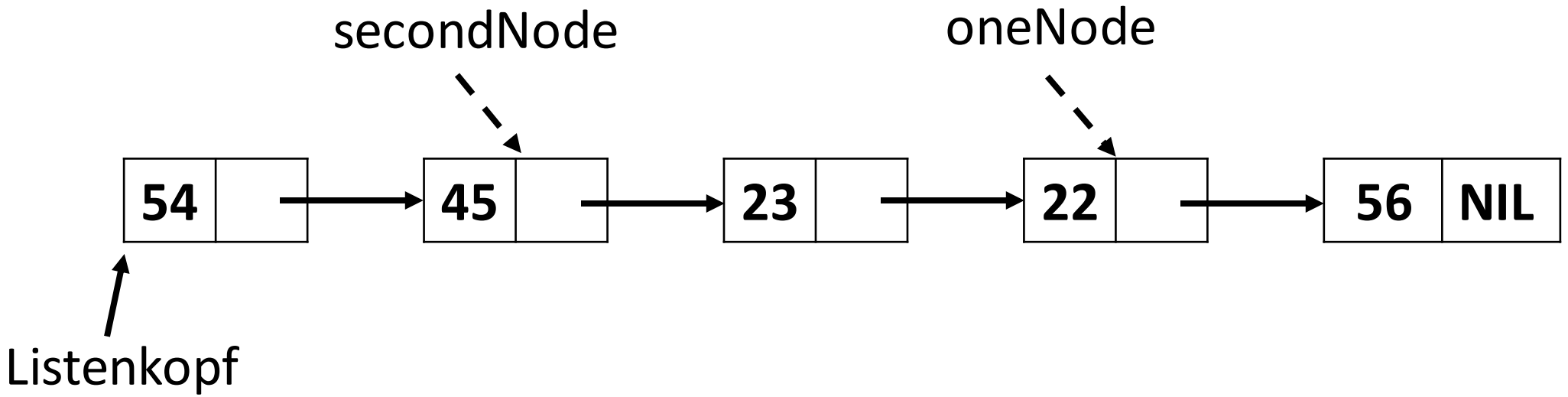
Finde den n-ten Knoten am Ende einer SLL.

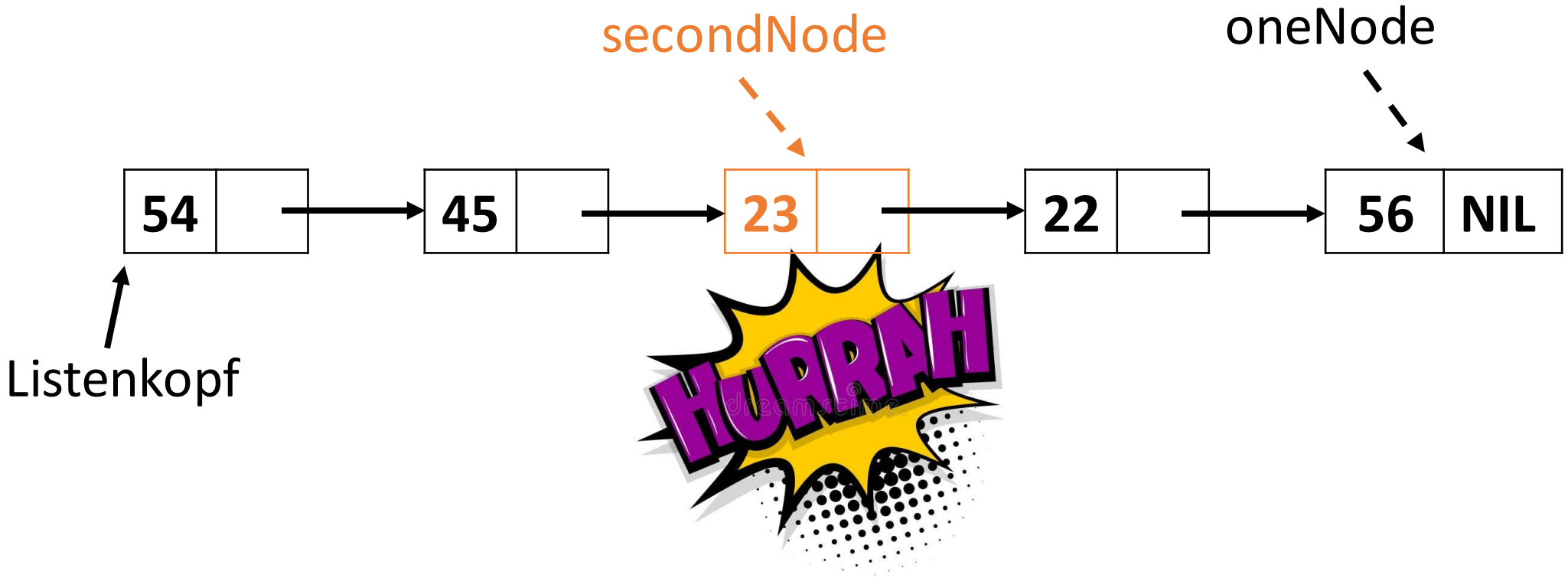
- Finde den dritten Knoten am Ende der Liste




oneNode
54
45
23
22
56
NIL
Listenkopf


secondNode
oneNode
54
45
23
22
56
NIL
Listenkopf


secondNode
oneNode
54
45
23
22
56
NIL
Listenkopf


secondNode
oneNode
54
45
23
22
56
NIL
Listenkopf
HURRAH

N-te Knoten vom Ende der Liste

```
function findNthFromEnd (sll, n) is:
//pre: sll ist ein SLL, n ist eine ganze Zahl
//post: der n-te Knoten vom Ende der Liste oder NIL wird zurückgegeben
    oneNode ← sll.head
    secondNode ← sll.head
    position ← 1
    while position < n and oneNode ≠ NIL execute
            oneNode ← [oneNode].next
            position ← position + 1
    end-while
    if oneNode = NIL then
            findNthFromEnd ← NIL
    else
    //continued on the next slide...
```

N-te Knoten vom Ende der Liste

```
            while [oneNode].next ≠ NIL execute
                    oneNode ← [oneNode].next
                    secondNode ← [secondNode].next
            end-while
            findNthFromEnd ← secondNode
    end-if
end-function
```

Algorithmische Aufgaben mit verketteten Listen

- Schreibe ein Algorithmus der eine einfach verkette Liste um eine Position rotiert (das erste Element wird das letzte Element sein)
  - Man muss den ersten Knoten aus der Liste löschen und am Ende einfügen
  - Sonderfälle:
    - Leere Liste
    - Liste mit einem Knoten

```
subalgorithm rotate (sll) is:
    if not (sll.head = NIL or [sll.head].next = NIL) then
        first ← sll.head                    // man speichert den ersten Knoten
        sll.head ← [sll.head].next          //man entfernt den Knoten aus der Liste
        current ← sll.head
        while [current].next ≠ NIL execute
            current ← [current].next
        end-while
        [current].next ← first
        [first].next ← NIL
    end-if
end-subalgorithm
```

Komplexität: $\Theta(n)$

Denk darüber nach

1. Wenn man den Listenkopf eines SLLs kennt, bestimme ob der Endknoten der Liste NIL in dem *next* Feld enthält, oder ob die Liste einen Zyklus enthält (der letzte Knoten enthält in dem *next* Feld die Adresse eines anderen Knotens).
2. Falls die Liste in dem vorigen Problem einen Zyklus enthält, bestimme die Länge des Zyklus.
3. Bestimme ob ein SLL eine gerade oder ungerade Anzahl von Elementen enthält, ohne die Knoten zu zählen.
4. Drehe eine verkettete Liste in linearen Zeit und $\Theta(1)$ Speicherplatzkomplexität um.

Sortierte Liste

- Eine *sortierte Liste* ist eine Liste deren Elemente aus den Knoten in einer bestimmten Reihenfolge sind, bezüglich einer *Ordnungsrelation*
- *Die Ordnungsrelation* kann $<, \leq, >$, oder $\geq$ sein, aber man kann auch mit einer abstrakten Relation arbeiten
- Eine abstrakte Relation bietet eine höhere Flexibilität an: man kann die Ordnungsrelation einfach ändern (ohne den Code für sortierte Listen zu ändern)
  - Man kann z.B. in derselben Anwendung mehrere Listen haben mit unterschiedlichen Ordnungsrelationen

Die Ordnungsrelation

- Die Relation ist eine Funktion mit zwei Parametern (zwei *TComp* Elemente):

$$\textit{relation}(c_1, c_2) = \begin{cases} true, & falls\ c_1\ vor\ c_2 \\ & oder\ c_1\ gleich\ mit\ c_2\ ist \\ false, & falls\ c_2\ vor\ c_1\ ist \end{cases}$$

- Wenn wir $c_1 \leq c_2$ sagen, dann meinen wir: c_1 ist vor c_2

Sortierte Liste - Repräsentierung

- Wenn man eine sortierte Liste speichern will (oder einen anderen sortierten Container oder Datenstruktur), dann speichert man die Ordnungsrelation als Teil der Datenstruktur (als ein Feld in dem ADT)

- Wir besprechen *sortiere einfach verkettete Listen* (die Repräsentierung und Implementierung für eine sortierte doppelt verkettete Liste sind ähnlich)

Sortierte Liste - Repräsentierung

- Man braucht zwei Datenstrukturen:
  - Knoten – SSLLNode
  - Sortierte einfach verkettete Liste (Sorted Singly Linked List) - SSLL

SSLLNode:
    info: TComp
    next: ↑ SLLNode

SSLL:
    head: ↑ SLLNode
    rel: ↑ Relation

SSLL - Initialisierung

- Für die Initialisierung braucht man auch die Relation als Parameter
- So kann man mehrere SSLLs mit unterschiedlichen Relationen erstellen

```
subalgorithm init (ssll, rel) is:
//pre: rel ist eine Relation
//post: ssll ist eine leere SSLL
        ssll.head ← NIL
        ssll.rel ← rel
end-subalgorithm
```

- Komplexität: $\Theta(1)$

SSLL - Operationen

- Der Hauptunterschied zwischen der Operationen eines SLLs und der Operationen eines SSLLs ist bei der *Einfügeoperation*:
  - Für ein SLL kann man am Anfang der Liste, am Ende der Liste, an einer gegebenen Position, vor/nach einem gegebenen Element einfügen (es gibt also mehrere Einfügeoperationen)
  - Für ein SSLL gibt es eine einzige Einfügeoperation: wir können nicht mehr selber entscheiden wo man ein Element einfügt, sondern das wird von der Ordnungsrelation bestimmt
- Bei den Löschoperationen ist es aber ähnlich: man kann mehrere Löschoperationen haben
- Die anderen Operationen sind auch ähnlich: ein Element suchen, ein Element zurückgeben

SSLL - Einfügeoperation

- In einem einfach verketteten Liste muss man den Knoten davor finden, um ein Element *nach* diesem Knoten einfügen zu können (sonst kann man die Links nicht richtig bestimmen)
- Der Knoten, den wir also finden müssen, ist der erste Knoten deren *Nachfolger größer* als das Element zum Einfügen ist *(größer* heißt, dass die Relation *false* zurückgibt*)*
- Es gibt zwei Ausnahmefälle:
  - Eine leere SSLL
  - Wenn man vor dem ersten Knoten einfügen muss

SSLL - insert

```
subalgorithm insert (ssll, elem) is:
//pre: ssll ist eine SSLL; elem ist ein TComp
//post: das Element elem wurde in ssll an der richtigen Position eingefügt
    newNode ← allocate()
    [newNode].info ← elem
    [newNode].next ← NIL
    if ssll.head = NIL then
    //die Liste ist leer
        ssll.head ← newNode
    else if ssll.rel(elem, [ssll.head].info) = true then
    //elem ist “kleiner als" das info Element aus dem Head
        [newNode].next ← ssll.head
        ssll.head ← newNode
    else
// Fortsetzung auf der nächsten Folie ...
```

SSLL - insert

```
            cn ← ssll.head   //cn - current node
            while [cn].next ≠ NIL and ssll.rel(elem, [[cn].next].info) = false execute
                cn ← [cn].next
            end-while
            //jetzt füge das Element nach cn ein
            [newNode].next ← [cn].next
            [cn].next ← newNode
        end-if
    end-subalgorithm
```

- Komplexität: $O(n)$

SSLL - andere Operationen

- Die Suche ist identisch mit der Suche in einem SLL (mit dem einzigen Unterschied, dass man früher mit dem Suchen aufhören kann: wenn man zu dem ersten Element größer als das gesuchte Element ankommt)
- Die Löschoperationen sind identisch mit den Löschoperationen für ein SLL
- Die Operationen um ein Element zurückzugeben ist identisch wie bei SLL
- Der Iterator für ein SSLL ist identisch mit dem Iterator für ein SLL

SSLL - Beispiel

- Wir definieren eine Funktion, die zwei ganze Zahlen vergleicht:

function compareGreater(e1, e2) **is**:
//pre: e1, e2 ganze Zahlen
//post: compareGreater gibt *true* zurück falls e1 ≤ e2; und *false* falls e1 > e2
    if e1 ≤ e2 **then**
        compareGreater ← true
    else
        compareGreater ← false
    end-if
end-function

SSLL - Beispiel

- Wir definieren eine Funktion, die zwei ganze Zahlen vergleicht in Bezug auf die Summe der Ziffern
- Wir vermuten die Funktion *sumOfDigits* ist schon implementiert

```
function compareGreaterSum(e1, e2) is:
//pre: e1, e2 ganze Zahlen
//post: compareGreaterSum gibt true zurück falls die Summe der Ziffern aus e1 kleiner oder gleich
//ist als die von e2; und false falls die Summe der Ziffern aus e1 größer ist
    sumE1 ← sumOfDigits(e1)
    sumE2 ← sumOfDigits(e2)
    if sumE1 ≤ sumE2 then
            compareGreaterSum ← true
    else
            compareGreaterSum ← false
    end-if
end-function
```

SSLL - Beispiel

- Wir definieren einen Subalgorithmus, der die Elemente des SSLLs mit Hilfe eines Iterators ausdruckt

```
subalgorithm printWithIterator(ssll) is:
//pre: ssll its ein SSLL; post: die Elemente des ssll werden ausgedruckt
    iterator(ssll, it)    //create an iterator for ssll
    while valid(it) execute
        e ← getCurrent(it)
        write e
        next(it)
    end-while
end-subalgorithm
```

SSLL - Beispiel

- Eine kurze Main-Methode, die ein SSLL erstellt, Elemente einfügt und dann ausdruckt

```
subalgorithm main() is:
    init(ssll, compareGreater) //man benutzt compareGreater als Relation
    insert(ssll, 55)
    insert(ssll, 10)
    insert(ssll, 59)
    insert(ssll, 37)
    insert(ssll, 61)
    insert(ssll, 29)
    printWithIterator(ssll)
end-subalgorithm
```

SSLL - Beispiel

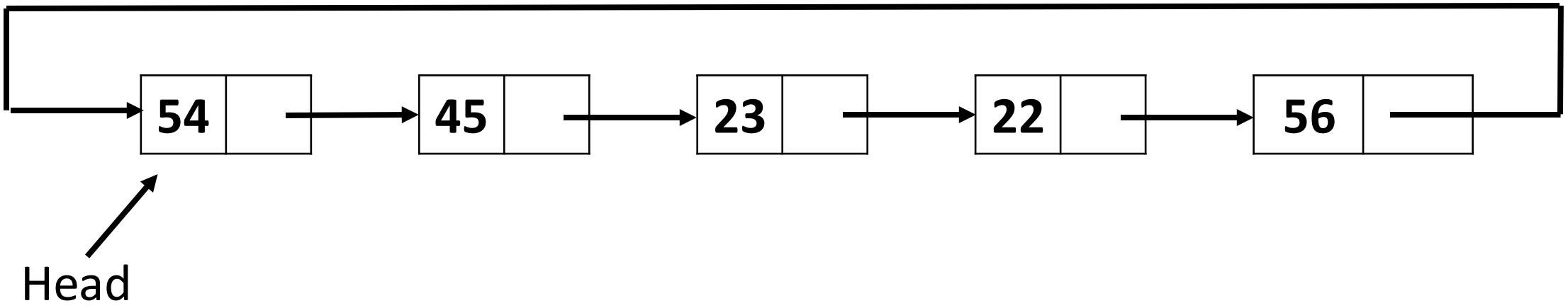
- Die Main-Methode von der vorigen Folie druckt Folgendes aus:
  10, 29, 37, 55, 59, 61

- Wenn wir anstatt *compareGreater* die Ordnungsrelation *compareGreaterSum* benutzen, dann würde die Main-Methode Folgendes ausdrucken: 10, 61, 37, 55, 29, 59

- Man kann eigentlich zwei Listen haben, eine mit der Ordnungsrelation *compareGreater* und eine mit *compareGreaterSum* ⇒ mit einer abstrakten Ordnungsrelation gibt es eine hohe Flexibilität

Zirkuläre Listen

- Für ein SLL oder DLL enthält der *next* Feld des letzten Knotens den Wert *NIL*
- In einer zirkulären Liste enthält kein Knoten den Wert NIL in dem *next* Feld, sondern der letzte Knoten enthält die Adresse des ersten Knotens



Zirkuläre Listen

- Es gibt einfach verkettete und doppelt verkettete zirkuläre Listen
- Wir besprechen einfach verkettete zirkuläre Listen

- In einer zirkulären Liste gibt es kein „Ende“, also man muss aufpassen wie man die Liste iteriert, sodass man nicht in eine Endlosschleife gelangt
- Kann die Abbruchbedingung gleich bleiben: wenn *currentNode* oder *[currentNode].next* NIL sind?
- Es gibt Aufgaben, wo eine zirkuläre Liste die Lösung einfacher macht (z.B. Josephus-Problem, eine Liste umdrehen)

Josephus-Problem

- Es stehen n Personen in einem Kreis. Beginnend bei Person Nummer *m* wird nun die *m*-te Person aus dem Kreis entfernt und der Kreis danach sofort wieder geschlossen. Nach dem Entfernen fängt man bei der nächsten Person (nach dem entfernten Person) mit dem Zählen wieder an. Das Ganze wird so lange durchgeführt, bis nur noch eine Person übrig bleibt, diese Person überlebt.
- Gegeben sei die Anzahl der Personen, *n*, und die Zahl *m.* Bestimme welche Person überleben wird.
- Z.B., für 5 Personen und m = 3, überlebt die 4-te Person.

Zirkuläre Listen

- Bei den Operationen einer zirkulären Liste muss man folgende wichtige Aspekte beachten:
  - Der *letzte* Knoten der Liste ist der Knoten, dessen *next* Feld die Adresse des Heads enthält
  - Vor dem Listenkopf einfügen oder den Listenkopf löschen sind nicht mehr einfache $\Theta(1)$ Operationen, da jetzt auch der *next* Feld des letzten Knotens geändert werden muss (dafür muss man den letzten Knoten finden)

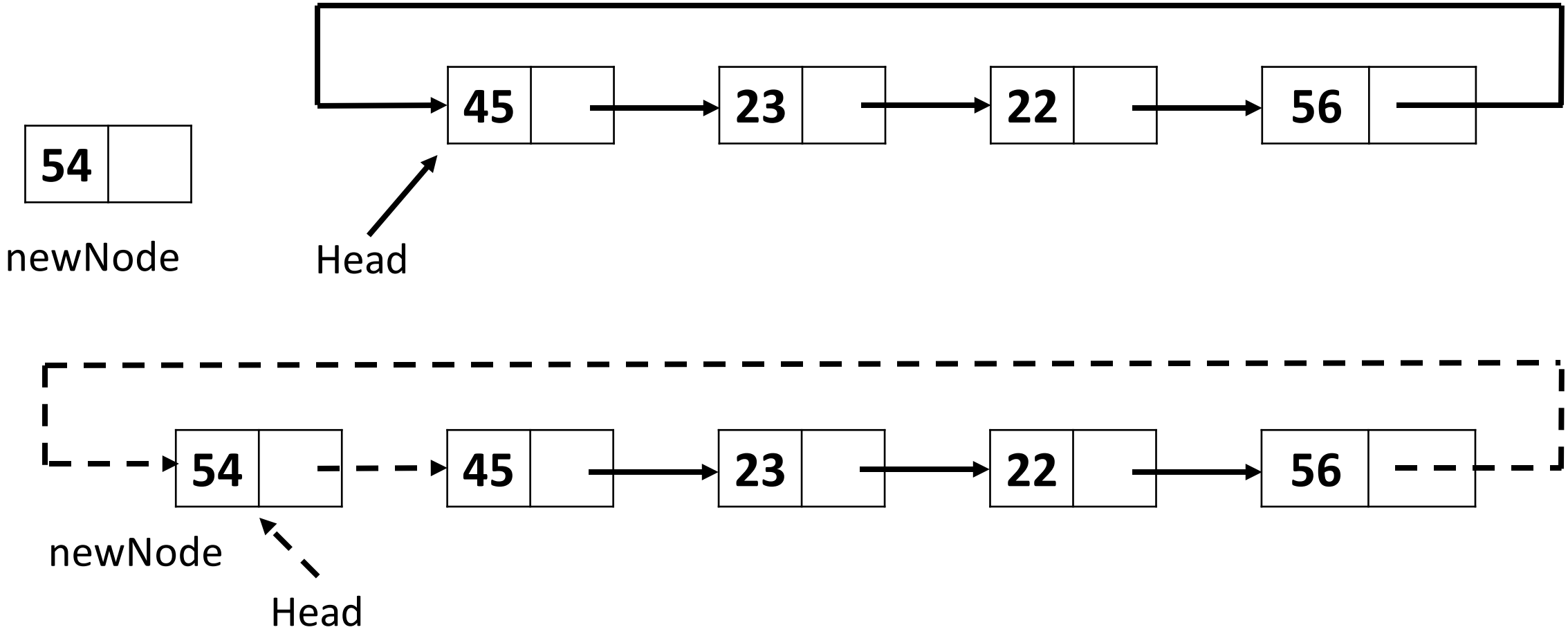
Zirkuläre Listen

- Die Repräsentierung einer zirkulären Liste ist die gleiche wie bei der einfachen SLL
- Es gibt zwei Datenstrukturen, eine für den Knoten und eine für die zirkuläre einfach verkettete Liste (CSLL – Circular Singly Linked List)

CSLLNode:
    info: TElem
    next: ↑ CLLNode

CSLL:
    head: ↑ CSLLNode

CSLL - InsertFirst



CSLL - InsertFirst

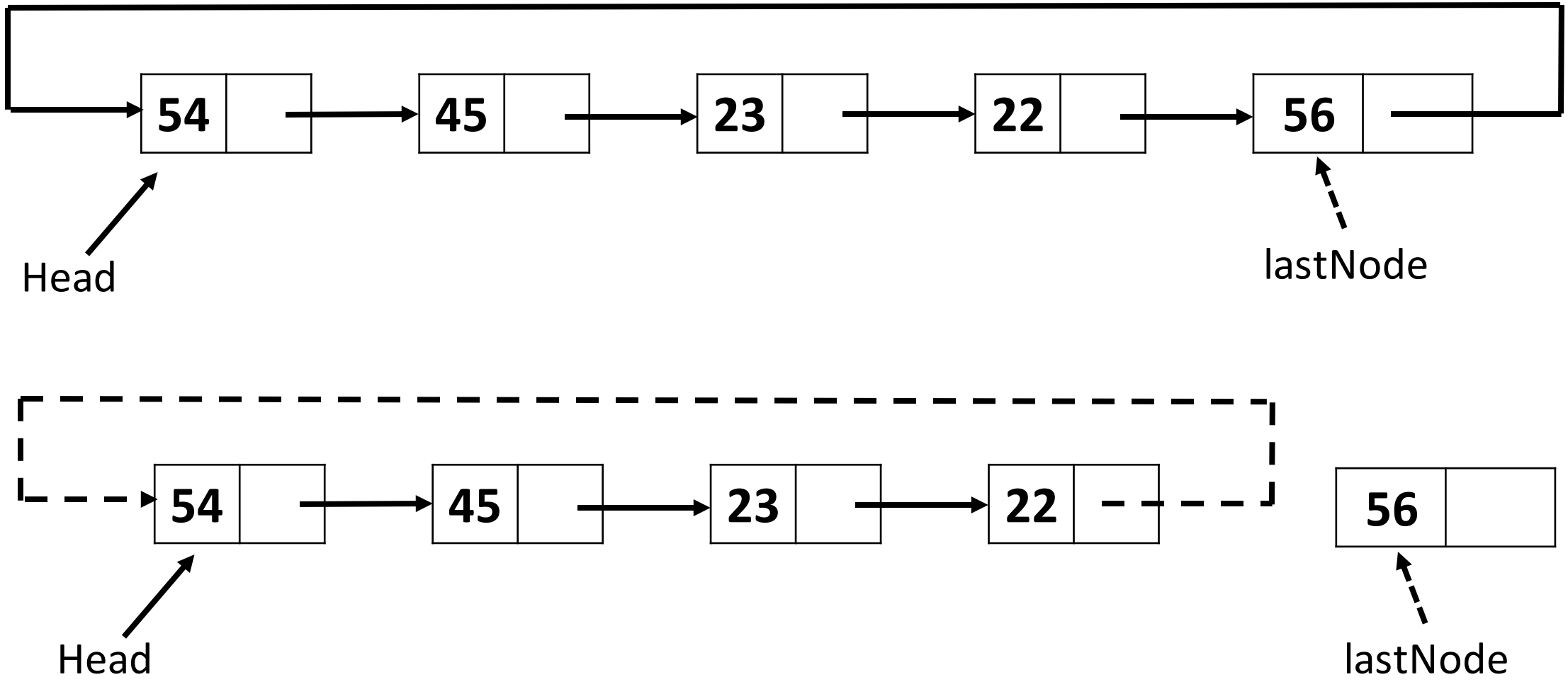
```
subalgorithm insertFirst (csll, elem) is:
//pre: csll ist ein CSLL, elem ist ein TElem
//post: das Element elem wird am Anfang von csll eingefügt
    newNode ← allocate()
    [newNode].info ← elem
    [newNode].next ← newNode
    if csll.head = NIL then
        csll.head ← newNode
    else
        lastNode ← csll.head
        while [lastNode].next ≠ csll.head execute
            lastNode ← [lastNode].next
        end-while
//Fortsetzung auf der nächsten Folie ...
```

CSLL - InsertFirst

```
            [newNode].next ← csll.head
            [lastNode].next ← newNode
            csll.head ← newNode
        end-if
    end-subalgorithm
```

- Komplexität: $\Theta(n)$
- Die Einfüge-Operation am Ende der Liste sieht ähnlich aus, aber man muss den Head nicht ändern (also die letzte Anweisung fehlt)

CSLL - DeleteLast



CSLL - DeleteLast

```
function deleteLast(csll) is:
//pre: csll ist ein CSLL
//post: das letzte Element aus csll wird gelöscht und der Knoten wird zurückgegeben
    deletedNode ← NIL
    if csll.head ≠ NIL then
        if [csll.head].next = csll.head then
            deletedNode ← csll.head
            csll.head ← NIL
        else
            prevNode ← csll.head
            while [[prevNode].next].next ≠csll.head execute
                prevNode ← [prevNode].next
            end-while
//Fortsetzung auf der nächsten Folie ...
```

CSLL - DeleteLast

```
                deletedNode ← [prevNode].next
                [prevNode].next ← csll.head
            end-if
        end-if
        [deletedNode].next ← NIL
        deleteLast ← deletedNode
    end-function
```

- Komplexität: $\Theta(n)$

CSLL - Iterator

- Wie kann man ein Iterator für ein CSLL definieren?
- Das Hauptproblem bei dem *normalen* SLL Iterator ist die *valid* Funktion.
  - Für ein SLL wird die *valid* Funktion falsch zurückgeben, wenn der Wert des *currentElement* NIL ist. Das ist aber nie der Fall in einer zirkulären Liste.
  - In einer zirkulären Liste kommt die Iterierung zu Ende wenn *currentElement* gleich mit dem *Head* ist
  - Wenn aber die *valid* Funktion falsch zurückgibt wenn *currentElement* gleich mit dem *Head* ist, dann ist der Iterator bei dem Erstellen schon ungültig

CSLL – Iterator - Möglichkeiten

- Man kann entscheiden, dass der Iterator ungültig ist, wenn das *next* Element des *currentElement* gleich ist mit dem *Head* der Liste

- In diesem Fall wird der Iterator bei dem letzten Element aufhören
- Wenn man also alle Elemente ausdrucken will, dann muss man die *element* Operation nachdem der Iterator ungültig wird noch einmal aufrufen (oder man kann eine *do-while* Schleife anstatt eine *while* Schleife benutzen – aber das verursacht Probleme im Falle einer leeren Liste)
- Das Problem in diesem Fall ist, dass dies die Vorbedingung verletzt, dass *element* nur aufgerufen werden sollte, wenn der Iterator gültig ist

CSLL – Iterator - Möglichkeiten

- Man kann **ein Boolean Flag** zu dem Iterator außer dem *currentElement* hinzufügen, der dann zeigt ob der Head der Liste nur einmal besucht wurde (also bei dem Erstellen des Iterator) oder ob man ein zweites Mal in dem Head ist (also nachdem man alle Elemente durchlaufen hat)

- Für diese Version ändert sich die Implementierung des Iterators nicht

- Ein andere Möglichkeit wäre anstatt den Flag **einen Zähler** zu speichern, den man mit der Anzahl der Elemente vergleichen kann (falls die Anzahl in der Repräsentierung vorkommt)

CSLL – Iterator - Möglichkeiten

- Für bestimmte Aufgaben kann man einen read-write Iterator brauchen: damit man den Inhalt des CSLL mit Hilfe des Iterators ändern kann

- Man kann die Operationen *insertAfter* (ein neues Element nach dem aktuellen Knoten einfügen) und *deleteAfter* (lösche das Element nach dem aktuellen Knoten) haben

- Man kann sagen, dass der Iterator ungültig wird wenn es keine Elemente in der zirkulären Liste mehr gibt (vor allem wenn man aus der Liste löschen kann)

Zirkuläre Liste - Variationen

- Es gibt unterschiedliche Variationen für zirkuläre Listen, die nützlich sein könnten:
  - Anstatt dem Head kann man den **Tail** speichern. In diesem Fall haben wir Zugriff sowohl zu dem Head als auch zu dem Tail der Liste, und man kann einfach vor dem Head oder nach dem Tail einfügen. Den Listenkopf löschen ist auch einfach, aber um den Tail zu löschen braucht man immer noch $\Theta(n)$ Zeit.

  - Man kann einen **Header oder Sentinel** Knoten benutzen – ein spezieller Knoten, der als Listenkopf betrachtet wird, aber der nie gelöscht werden kann (eine Abtrennung zwischen dem Head und dem Tail). Für diese Version ist es einfacher den Iterator aufzuhalten.