Datenstrukturen und Algorithmen

Vorlesung 4

Überblick

- Vorige Woche:
  - ADT Bag & SortedBag
  - ADT Set & SortedSet
  - ADT Map & SortedMap
  - ADT Matrix
  - ADT Queue

- Heute betrachten wir:
  - ADT Prioritätsschlange
  - ADT Stack
  - ADT MultiMap & SortedMultiMap
  - ADT IndexedList & IteratedList

  - Einfach verkettete Listen (SLL) - Intro

- Wie funktioniert eine Schlange in der Notaufnahme in einem Krankenhaus?

ADT Prioritätsschlange/Vorrangwarteschlange (Priority Queue)

- 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

- Generell, kann man die Relation *R* auf die Menge der Prioritäten definieren: $R : TPriority \times TPriority$

- Das Element mit der höchsten Priorität wird von der Relation *R* bestimmt

- Zum Beispiel, falls die Relation *R* = ”$\geq$”, dann ist das Element mit der höchsten Priorität das größte Element (Maximum)

ADT Prioritätsschlange

- Domäne:

$\mathcal{PQ}$ = { *pq* | *pq* ist eine Prioritätsschlange mit Elementen (*e, p*), *e* ∈ *TElem, p* ∈ *TPriority* }

ADT Prioritätsschlange - Interface

- init(pq, R)
  - **descr:** erstellt eine leere Prioritätsschlange
  - **pre:** R ist eine Relation auf die Menge der Prioritäten, $R : TPriority \times TPriority$
  - **post**: $pq \in \mathcal{PQ}$, pq ist eine leere Prioritätsschlange

- destroy(*pq*)
  - **descr**: zerstört eine Prioritätsschlange
  - **pre**: $pq \in \mathcal{PQ}$
  - **post**: *pq* wurde zerstört

ADT Prioritätsschlange - Interface

- push(pq, e, p)
  - **descr:** fügt ein neues Element in die Prioritätsschlange ein
  - **pre:** $pq \in \mathcal{PQ}$, $e \in TElem$, $p \in TPriority$
  - **post**: $pq' \in \mathcal{PQ}$, $pq' = pq \oplus (e,p)$

- pop(pq)
  - **descr:** liefert das Element mit der höchsten Priorität aus der Prioritätsschlange und entfernt es von der Schlange. Die Priorität des Elementes wird auch zurückgegeben.
  - **pre:** $pq \in \mathcal{PQ}$
  - **post:** $pop \leftarrow$ (e, p), $e \in TElem$, $p \in TPriority$, e ist das Element mit der höchsten Priorität aus pq und p ist seine Priorität, $pq' \in \mathcal{PQ}$, $pq' = pq \ominus (e,p)$
  - **throws**: ein Underflow Error, falls die Prioritätsschlange leer ist

ADT Prioritätsschlange - Interface

- top(pq)
  - **descr:** liefert das Element mit der höchsten Priorität aus der Prioritätsschlange zusammen mit seiner Priorität. Die Prioritätsschlange wird nicht geändert.
  - **pre:** *pq* ∈ $\mathcal{PQ}$
  - **post:** *top* ← (e, p), *e* ∈ *TElem, p* ∈ *TPriority, e* ist das Element mit der höchsten Priorität aus *pq* und *p* ist seine Priorität
  - **throws**: ein Underflow Error, falls die Prioritätsschlange leer ist

- isEmpty(pq)
  - **descr:** überprüft ob die Prioritätsschlange leer ist
  - **pre:** *pq* ∈ $\mathcal{PQ}$
  - **post**: *isEmpty* ← $\begin{cases} wahr, & falls\ pq\ keine\ Elemente\ enthält \\ falsch, & ansonsten \end{cases}$

ADT Prioritätsschlange - Interface

- isFull(pq)
  - **descr:** überprüft ob die Schlange voll ist
  - **pre:** *pq* ∈ $\mathcal{PQ}$
  - **post**: *isFull* $\leftarrow \begin{cases} wahr, & falls\ pq\ voll\ ist \\ falsch, & ansonsten \end{cases}$

- **Bemerkung!**

  Die Prioritätsschlange kann nicht iteriert werden!

  Dafür gibt es auch keine *iterator* Operation.

ADT Stack (Stapel/Keller/Kellerspeicher)

ADT Stack (Stapel/Keller/Kellerspeicher)

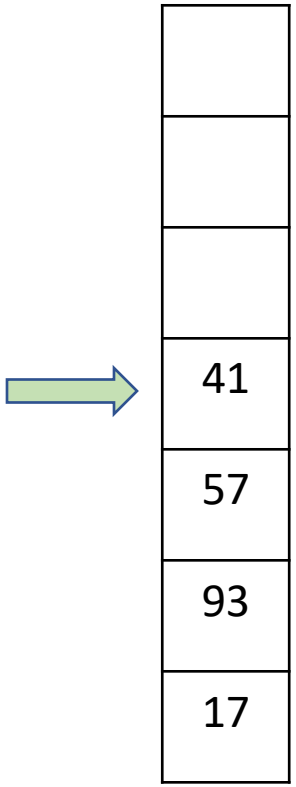
- ADT Stack ist ein Container, wo der Zugriff zu den Elementen auf ein Ende des Behälters (*Top des Stacks)* beschränkt ist:
  - In einem Stack können Elemente nur “von oben” hinzugefügt (eingekellert) und von oben entnommen (ausgekellert) werden
  - Man kann nur auf das „oberste“ Element zugreifen

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

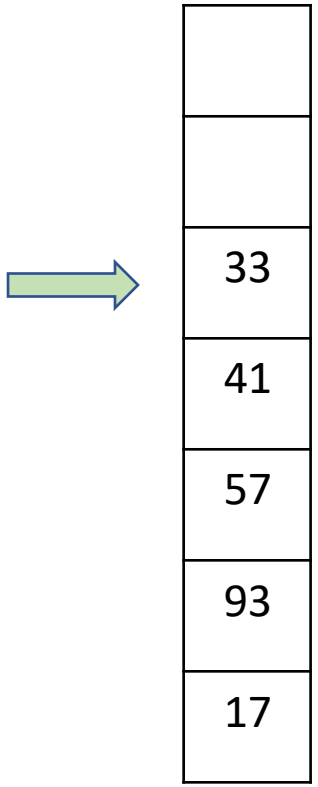
- Statt Stack verwendet man auch die Bezeichnungen *Stapel, Keller/Kellerspeicher* oder ***LIFO****-Liste*

Stack – Beispiel

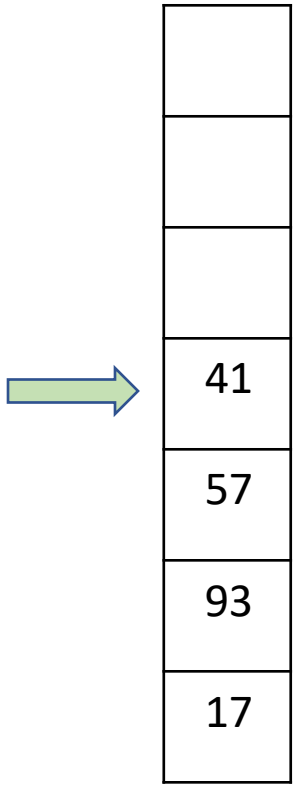
- Man fängt vom folgenden Stack an (grüner Pfeil zeigt den Top)



- Man fügt den Wert 33 ein (push)

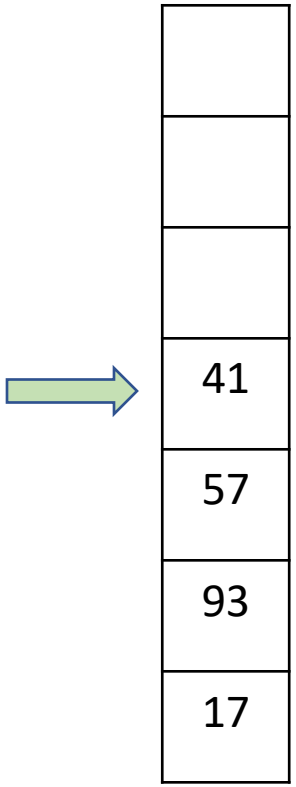


- Man löscht ein Element (pop)

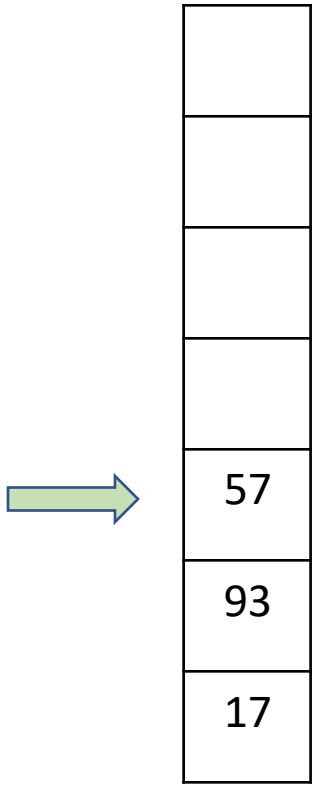


Stack – Beispiel

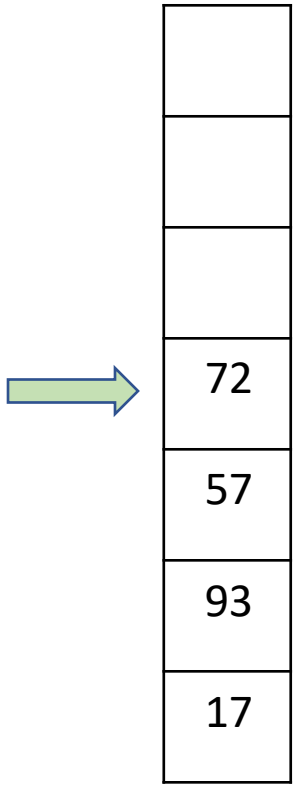
- Man fängt vom folgenden Stack an (grüner Pfeil zeigt den Top)



- Man löscht ein Element (pop)



- Man fügt den Wert 72 ein (push)



ADT Stack – Interface

- Domäne von ADT Stack:

$$\mathcal{S} = \{s \mid s \text{ ist ein Stack mit Elementen vom Typ } \mathit{TElem}\}$$

ADT Stack – Interface

- init(s)
  - **descr:** erstellt einen leeren Stack
  - **pre:** wahr
  - **post**: $s \in \mathcal{S}$, s ist einen leeren Stack

- destroy(s)
  - **descr**: zerstört einen Stack
  - **pre**: $s \in \mathcal{S}$
  - **post**: s wurde zerstört

ADT Stack – Interface

- push(s, e)
  - **descr:** legt ein neues Element oben auf den Stack (Einfüge-Operation)
  - **pre:** $s \in \mathcal{S}$, $e \in TElem$
  - **post**: $s' \in \mathcal{S}$, $s' = s \oplus e$, e ist das oberste Element
  - **throws**: ein Overflow Error, falls s voll ist

- pop(s)
  - **descr:** liefert das oberste Objekt auf dem Stack und entfernt es vom Stack
  - **pre:** $s \in \mathcal{S}$
  - **post**: $pop \leftarrow e$, $e \in TElem$, e ist das oberste Element aus s, $s' \in \mathcal{S}$, $s' = s \ominus e$
  - **throws**: ein Underflow Error, falls s leer ist

ADT Stack – Interface

- top(s)
  - **descr:** gibt das oberste Element auf dem Stack zurück (der Stack wird aber nicht geändert)
  - **pre:** $s \in \mathcal{S}$
  - **post**: *top* ← *e*, *e* ∈ *TElem, e* ist das oberste Element aus *s*
  - **throws:** ein Underflow Error, falls *s* leer ist

- isEmpty(s)
  - **descr:** überprüft ob der Stack leer ist
  - **pre:** $s \in \mathcal{S}$
  - **post**: *isEmpty* ← $\begin{cases} wahr, & falls\ s\ keine\ Elemente\ enthält \\ falsch, & ansonsten \end{cases}$

ADT Stack – Interface

- **Bemerkung!**

  Stacks können nicht iteriert werden!

  Dafür gibt es auch keine *iterator* Operation.

Aufgabe

- Ihr müsst einen Synonymen-Wörterbuch speichern.
- Welcher Container würdet ihr benutzen und mit welchen Eigenschaften?
  - Elemente sind Paare Schlüssel-Wert (Wort – Synonym)
  - Für einen Schlüssel kann es mehrere zugehörige Werte geben
  - Die Reihenfolge der Elemente ist nicht wichtig

⇒ man braucht eine ADT MultiMap

MultiMap

- Welche Unterschiede zu der Map gibt es bei den Operationen der MultiMap?
  - Bei der Einfügeoperation muss man nicht überprüfen ob sich der Schlüssel in der MultiMap befindet
  - Die Löschoperation braucht jetzt auch den Schlüssel und auch den Wert um den Paar löschen zu können
  - Wenn wir nach einem Schlüssel suchen, dann wird eine Liste von Werten zurückgegeben

ADT MultiMap

- Domäne von ADT MultiMap:

  $\mathcal{MM}$ = {*mm* | *mm* ist eine MultiMap mit Elementen *e = (k, v),* wobei *k* ∈ *TKey* und *v* ∈ *TValue;* ein Schlüssel kann mehrere entsprechenden Werte haben}

MultiMap – Interface

- init(mm)
  - **descr:** erstellt eine leere MultiMap
  - **pre:** wahr
  - **post**: *mm* $\in \mathcal{M}\,\mathcal{M}$, *mm* ist eine leere MultiMap

- destroy(mm)
  - **descr**: zerstört eine MultiMap
  - **pre**: *mm* $\in \mathcal{M}\,\mathcal{M}$
  - **post**: *mm* wurde zerstört

MultiMap – Interface

- add(mm, k, v)
  - **descr**: fügt ein neues key-value Paar zu der MultiMap ein
  - **pre**: *mm* ∈ $\mathcal{M}\,\mathcal{M}$, *k* ∈ *TKey, v* ∈ *TValue*
  - **post**: *mm'* ∈ $\mathcal{M}$, *mm'* = *mm* ∪ <k, v>

- remove(mm, k, v)
  - **descr**: löscht ein Paar mit einem gegebenen Schlüssel aus der MultiMap
  - **pre**: *mm* ∈ $\mathcal{M}\,\mathcal{M}$, *k* ∈ *TKey, v* ∈ *TValue*
  - **post**:

$$\text{remove} \leftarrow \begin{cases} true, falls\ \exists < k, v > \in mm\ und\ mm' \in \mathcal{M}\mathcal{M}, mm' = mm \setminus < k, v > \\ false,\ \ ansonsten \end{cases}$$

MultiMap – Interface

- search(mm, k, l)
  - **descr**: gibt die Liste von Werten entsprechend dem gegebenen Schlüssel zurück
  - **pre**: *mm* $\in \mathcal{MM}$, *k* $\in$ *TKey*
  - **post**: *l* $\in \mathcal{L}$, wobei *l* die Liste der entsprechenden Werte für den Schlüssel *k* ist. Falls der Schlüssel *k* nicht in der MultiMap enthalten ist, dann ist *l* eine leere Liste

MultiMap – Interface

- iterator(mm, it)
  - **descr**: gibt ein Iterator für eine MultiMap zurück
  - **pre**: *mm* $\in \mathcal{MM}$
  - **post**: *it* $\in \mathcal{I}$, *it* ist in Iterator für *m*
  - *Bem.* Die *getCurrent* Operation gibt einen Paar <key, value> zurück
- size(mm)
  - **descr**: gibt die Anzahl der Paare in der MultiMap zurück
  - **pre**: *mm* $\in \mathcal{MM}$
  - **post**: *size* $\leftarrow$ Anzahl der Paare in *mm*

MultiMap – Interface

- Andere mögliche Operationen:

- keys(mm, s)
  - **descr**: gibt die Menge der Schlüssel aus der MultiMap zurück
  - **pre**: $mm \in \mathcal{MM}$
  - **post**: $s \in \mathcal{S}$, s ist ein Set, der alle Schlüssel aus mm enthält

MultiMap – Interface

- values(mm, b)
  - **descr**: gibt ein Bag von Werten aus der MultiMap zurück
  - **pre**: $mm \in \mathcal{MM}$
  - **post**: $b \in \mathcal{B}$, b ist ein Bag, der alle Werte aus mm enthält

- pairs(mm, b)
  - **descr**: gibt die Menge der Paare aus der MultiMap zurück
  - **pre**: $mm \in \mathcal{MM}$
  - **post**: $b \in \mathcal{B}$, b ist ein Bag, der alle Paare aus mm enthält
  - *Bem.* Es unterscheidet sich von derselben Operation einer Map, da in einer MultiMap ein Paar mehrmals enthalten sein kann

ADT SortedMultiMap

- Man kann für die Schlüssel in der Map eine Ordnungsrelation definieren, dann benutzt man *TComp* anstatt *TKey*

- Die einzigen Änderungen zu dem Interface sind bei der *init* Operation, wo man auch die Relation als Parameter hat
- Für eine sortierte MultiMap muss der Iterator die Paare in der Reihenfolge gegeben von der Relation durchlaufen. In diesem Fall geben die Operationen *keys* und *pairs* SortedSet und SortedBag zurück.
- Wenn es aber mehrere Werte für denselben Schlüsselwert gibt, dann werden diese nicht sortiert, **nur die Schlüssel werden sortiert**.
- Bei Bedarf kann man auch eine Ordnungsrelation auf Werte definieren

MultiMap/SortedMultiMap - Repräsentierung

- Um ADT MultiMap (oder ADT SortedMultiMap) zu implementieren kann man folgende Datenstrukturen für die Repräsentierung benutzen:
  - (dynamisches) Array
  - Verkettete Liste (s. Beispiel Seminar 3)
  - Hashtabellen
  - (balancierte) Binärbäume – für sortierte MultiMap
  - Skip Listen – für sortierte MultiMaps

MultiMap/SortedMultiMap - Repräsentierung

- Egal ob man ADT MultiMap oder SortedMultiMap repräsentieren will, gibt es im Allgemeinen zwei Möglichkeiten:
  - Man speichert individuelle Paare der Form (Schlüssel, Wert), wobei ein Schlüssel mehrmals vorkommen kann.
  - Man speichert eindeutige Schlüssel und für jeden Schlüssel die entsprechende Liste von Werten

ADT List

- Eine **Liste** ist eine endliche Folge von null oder mehr Elementen eines gegebenen Typs $<l_1, l_2, ..., l_n>$, wobei die Reihenfolge der Elemente bekannt ist, und jedes Element eine *Position* hat

- In einer Liste ist die Reihenfolge der Elemente wichtig (im Gegenteil zu ADT Set zum Beispiel)

- Die Anzahl der Elemente aus der Liste heißt *Länge* der Liste. Eine Liste ohne Elemente heißt *leer.*

- Die Position, an der ein Element *e* steht, wird auch als *Vorkommen* von *e* bezeichnet.

ADT List

- Eine Liste ist ein Container, der entweder leer ist oder:
  - Es enthält wenigstens ein Element
  - Für jedes Element, außer des letzten, gibt es einen eindeutigen Nachfolger
  - Für jedes Element, außer des ersten, gibt es einen eindeutigen Vorgänger
- In einer Liste kann man:
  - Elemente auf einer bestimmten Position einfügen,
  - von einer gegebenen Position löschen,
  - auf den Nachfolger und Vorgänger eines gegebenen Elementes zugreifen,
  - auf das Element an einer bestimmten Position zugreifen

ADT List – Positionen

- Jedes Element aus der Liste hat eine eindeutige Position:
  - Positionen sind relativ zur Liste
  - Die Position eines Elementes:
    - bestimmt/identifiziert das Element
    - bestimmt die Position des Nachfolgers und des Vorgängers (falls diese existieren)

ADT List – Positionen

- Die Position eines Elementes kann unterschiedlich betrachtet werden:
  - Als Rang des Elementes in der Liste (erste, zweite, usw.)
    - Ähnlich wie bei Arrays, wo die Position gleich ist mit dem Index des Elementes
  - Als Referenz zu dem Speicherplatz, wo das Element gespeichert wird
    - Z.B. ein Pointer zu der Adresse des Elementes
- Damit wir die Position allgemein betrachten können, benutzen wir den abstrakten Typ *TPosition* für die Position eines Elementes

ADT List – Positionen

- Eine Position *p* ist *gültig* (valid) falls es ein Element in der Liste gibt mit der entsprechenden Position:
  - Falls *p* ein Pointer ist zu einer Adresse im Speicherplatz, dann ist *p* gültig falls diese die Adresse eines Elementes der Liste ist (und nicht NIL oder irgendeine andere Adresse)
  - Falls *p* der Rang eines Elementes ist, dann ist *p* gültig falls es einen Wert zwischen 1 und der Anzahl der Elemente enthält
- Für ungültige Positionen benutzen wir die Notation: $\perp$

ADT List

- Domäne des ADT List:

$\mathcal{L}$ = {*l* | *l* ist eine Liste mit Elementen vom Typ TElem, wobei jedes Element eine eindeutige Position vom Typ TPosition in *l* hat}

ADT List – Interface

- init(l)
  - **descr:** erstellt eine neue, leere Liste
  - **pre:** wahr
  - **post**: *l* ∈ L, *l* ist eine leere Liste

- first(l)
  - **descr**: gibt die TPosition des ersten Elementes zurück
  - **pre**: *l* ∈ L
  - **post**: *first* ← *p* ∈ *TPosition*

$$p \leftarrow \begin{cases} die\ Position\ des\ ersten\ Elementes\ aus\ l, & falls\ l \neq \emptyset \\ \perp, & ansonsten \end{cases}$$

ADT List – Interface

- last(l)
  - **descr**: gibt die TPosition des letzten Elementes zurück
  - **pre**: *l* ∈ *L*
  - **post**: *last* ← *p* ∈ *TPosition*

$$\text{p} \leftarrow \begin{cases} die\ Position\ des\ letzten\ Elementes\ aus\ l, & falls\ l \neq \emptyset \\ \perp, & ansonsten \end{cases}$$

ADT List – Interface

- valid(l, p)
  - **descr**: überprüft ob ein TPosition gültig ist
  - **pre**: *l* ∈ *L*, *p* ∈ *TPosition*
  - **post**: valid $\leftarrow \begin{cases} wahr, & falls\ p\ eine\ g\ddot{u}ltige\ Position\ in\ l\ ist \\ falsch, & ansonsten \end{cases}$

ADT List – Interface

- next (l, p)
  - **descr:** gibt die TPosition des nächsten Elementes aus der Liste zurück
  - **pre**: *l* ∈ *L*, *p* ∈ *TPosition,* valid(*l*, *p*)
  - **post**: next ← *q* ∈ *TPosition*

$$q \leftarrow \begin{cases} die\ Position\ des\ nächsten\ Elementes\ nach\ p, & falls\ p\ nicht\ die\ letzte\ Position\ in\ l\ ist \\ \perp, & ansonsten \end{cases}$$

  - **throws**: ein Exception, falls *p* ungültig ist

ADT List – Interface

- previous (l, p)
  - **descr:** gibt die TPosition des vorigen Elementes aus der Liste zurück
  - **pre**: *l* ∈ *L, p* ∈ *TPosition,* valid(*l*, *p*)
  - **post**: previous ← *q* ∈ *TPosition*

$$q \leftarrow \begin{cases} die\ Position\ des\ Elementes\ vor\ p, & falls\ p\ nicht\ die\ erste\ Position\ in\ l\ ist \\ \perp, & ansonsten \end{cases}$$

  - **throws**: ein Exception, falls *p* ungültig ist

ADT List – Interface

- getElement(l, p)
  - **descr**: gibt das Element von einer gegebenen TPosition zurück
  - **pre**: $l \in L$, $p \in TPosition$, *valid(l, p)*
  - **post**: getElement $\leftarrow e$, $e \in TElem$, e = das Element an der Position p in l
  - **throws**: ein Exception, falls p ungültig ist

ADT List – Interface

- position(l, e)
  - **descr**: gibt die TPosition eines Elementes zurück
  - **pre**: *l* ∈ L, *e* ∈ *TElem*
  - **post**: position ← *p*, *p* ∈ *TPosition*

$$\text{p} \leftarrow \begin{cases} die\ erste\ Position\ des\ Elementes\ e, & falls\ e \in l \\ \perp, & ansonsten \end{cases}$$

ADT List – Interface

- setElement (l, p, e)
  - **descr:** ersetzt das Element an der gegebenen TPosition mit einem neuen Element und gibt den alten Wert zurück
  - **pre**: $l \in L$, $p \in TPosition$, $e \in TElem$, $valid(l, p)$
  - **post**: $l' \in L$, das Element an der Position p aus l' ist e

    setElement $\leftarrow el$, $el \in TElem$, el ist das Element von der Position p aus l
  - **throws**: ein Exception, falls p ungültig ist

ADT List – Interface

- addToBeginning (l, e)
  - **descr:** fügt ein neues Element am Anfang der Liste ein
  - **pre**: $l \in L$, $e \in TElem$
  - **post**: $l' \in L$, das Element e wurde am Anfang der Liste eingefügt

- addToEnd (l, e)
  - **descr:** fügt ein neues Element am Ende der Liste ein
  - **pre**: $l \in L$, $e \in TElem$
  - **post**: $l' \in L$, das Element e wurde am Ende der Liste eingefügt

ADT List – Interface

- addBeforePosition (l, p, e)
  - **descr:** fügt ein neues Element vor der gegebenen Position ein
  - **pre**: $l \in L$, $p \in TPosition$, $e \in TElem$, $valid(l, p)$
  - **post**: $l' \in L$, das Element e wurde vor der Position *p* in *l* eingefügt
  - **throws**: ein Exception, falls *p* ungültig ist

- addAfterPosition (l, p, e)
  - **descr:** fügt ein neues Element nach der gegebenen Position ein
  - **pre**: $l \in L$, $p \in TPosition$, $e \in TElem$, $valid(l, p)$
  - **post**: $l' \in L$, das Element e wurde vor der Position *p* in *l* eingefügt
  - **throws**: ein Exception, falls *p* ungültig ist

ADT List – Interface

- remove (l, p)
  - **descr:** löscht das Element von der gegebenen Position
  - **pre**: *l* ∈ $\mathbb{L}$, *p* ∈ *TPosition, valid(l, p)*
  - **post**: remove ← *e, e* ∈ *TElem, e* ist das Element an der Position *p* aus *l, l'* ∈ $\mathbb{L}$, $l' = l - e$
  - **throws**: ein Exception, falls *p* ungültig ist

- remove (l, e)
  - **descr:** löscht das erste Vorkommen des Elementes e
  - **pre**: *l* ∈ $\mathbb{L}$, *e* ∈ *TElem*
  - **post**:

$$\text{remove} \leftarrow \begin{cases} true, & falls\ e \in l\ und\ e\ gel\ddot{o}scht\ wurde \\ false, & ansonsten \end{cases}$$

ADT List – Interface

- search (l, e)
  - **descr:** sucht ein Element in der Liste
  - **pre**: *l* ∈ *L, e* ∈ *TElem*
  - **post**:

$$\text{search} \leftarrow \begin{cases} true, & falls\ e \in l \\ false, & ansonsten \end{cases}$$

ADT List – Interface

- isEmpty (l)
  - **descr:** überprüft ob eine Liste leer ist
  - **pre**: $l \in L$
  - **post**: isEmpty $\leftarrow \begin{cases} wahr, & falls\ l = \emptyset \\ falsch, & ansonsten \end{cases}$

- size (l)
  - **descr:** gibt die Anzahl der Elemente aus der Liste zurück
  - **pre**: $l \in L$
  - **post**: size $\leftarrow$ Anzahl der Elemente aus *l*

ADT List – Interface

- destroy (l)
  - **descr:** zerstört eine Liste
  - **pre**: $l \in \mathcal{L}$
  - **post**: l wurde zerstört

- iterator (l, it)
  - **descr:** gibt ein Iterator für die Liste zurück
  - **pre**: $l \in \mathcal{L}$
  - **post**: $it \in \mathcal{I}$, *it* ist ein Iterator für *l,* das aktuelle Element in dem Iterator ist das erste Element aus *l*, oder, *it* ist ungültig falls *l* leer ist

TPosition – Java, Python

- In Java und Python, wird TPosition als Index dargestellt
- Das Einfügen, Löschen und Zugreifen auf Elemente werden mit Hilfe von Indexe implementiert
- Es gibt wenigere Operationen in dem Interface der Liste, aber es gibt auch ein Iterator für die Liste

- Zum Beispiel, in Java:
  - void add(int index, E element)
  - E get(int index)
  - E remove(int index)

- Zum Beispiel, in Python:
  - insert (int index, E object)
  - index (E object)

TPosition – C++

- In STL, wird TPosition durch einen Iterator dargestellt

- Zum Beispiel – vector:
  - iterator insert(iterator position, const value_type& val)
  - iterator erase(iterator position)

- Zum Beispiel – list:
  - iterator insert(iterator position, const value_type& val)
  - iterator erase(iterator position)

ADT IndexedList, ADT IteratedList

- In den Laboraufgaben unterscheiden wir zwischen TPosition dargestellt als Index und TPosition dargestellt durch einen Iterator

- **ADT IndexedList** benutzt TPosition dargestellt als **Index**
- **ADT IteratedList** benutzt TPosition dargestellt als **Iterator**

ADT IndexedList

- Wenn TPosition ein Integer-Wert ist, dann haben wir eine IndexedList
- Die Operationen, die mit Positionen arbeiten haben in diesem Fall eine ganze Zahl als Eingabeparameter für die Position
- Es gibt wenigere Operationen in dem Interface der IndexedList:
  - first, last, next, previous und valid gibt es nicht mehr

ADT IndexedList – Interface

- init(l)
  - **descr:** erstellt eine neue, leere Liste
  - **pre:** wahr
  - **post**: $l \in L$, l ist eine leere Liste

- getElement(l, i)
  - **descr**: gibt das Element von einer gegebenen Position zurück
  - **pre**: $l \in L$, $i \in N$, i ist eine gültige Position
  - **post**: getElement $\leftarrow e$, $e \in TElem$, e = das Element an der Position i in l
  - **throws**: ein Exception, falls i ungültig ist

ADT IndexedList – Interface

- position(l, e)
  - **descr**: gibt die Position eines Elementes zurück
  - **pre**: *l* ∈ *L*, *e* ∈ *TElem*
  - **post**: position ← *i*, *i* ∈ *N*

$$\text{i} \leftarrow \begin{cases} die\ erste\ Position\ des\ Elementes\ e, aus\ l & falls\ e \in l \\ -1, & ansonsten \end{cases}$$

ADT IndexedList – Interface

- setElement (l, i, e)
  - **descr:** ersetzt das Element an der gegebenen Position mit einem neuen Element und gibt den alten Wert zurück
  - **pre**: $l \in L$, $i \in N$, $e \in TElem$, i ist eine gültige Position
  - **post**: $l' \in L$, das Element an der Position i aus l' ist e

    setElement $\leftarrow el$, $el \in TElem$, el ist das Element von der Position i aus l
  - **throws**: ein Exception, falls i ungültig ist

ADT IndexedList – Interface

- addToBeginning (l, e)
  - **descr:** fügt ein neues Element am Anfang der Liste ein
  - **pre**: $l \in L, e \in TElem$
  - **post**: $l' \in L$, das Element e wurde am Anfang der Liste eingefügt

- addToEnd (l, e)
  - **descr:** fügt ein neues Element am Ende der Liste ein
  - **pre**: $l \in L, e \in TElem$
  - **post**: $l' \in L$, das Element e wurde am Ende der Liste eingefügt

ADT IndexedList – Interface

- addToPosition (l, i, e) (oder addBeforePosition)
  - **descr:** fügt ein neues Element vor der gegebenen Position ein
  - **pre**: *l* ∈ *L*, *i* ∈ *N*, *e* ∈ *TElem*, *i* ist eine gültige Position (Anzahl der Elemente + 1 ist gültig beim Einfügen)
  - **post**: *l'* ∈ *L*, das Element e wurde auf der Position *i* in *l* eingefügt
  - **throws**: ein Exception, falls *i* ungültig ist

ADT IndexedList – Interface

- remove (l, i)
  - **descr:** löscht das Element von der gegebenen Position
  - **pre**: *l* ∈ *L, i* ∈ *N, i* gültige Position
  - **post**: remove ← *e, e* ∈ *TElem, e* ist das Element an der Position *i* aus *l, l'* ∈ *L, l'* = *l* − *e*
  - **throws**: ein Exception, falls *i* ungültig ist

- remove (l, e)
  - **descr:** löscht das erste Vorkommen des Elementes e
  - **pre**: *l* ∈ *L, e* ∈ *TElem*
  - **post**:

$$\text{remove} \leftarrow \begin{cases} true, & falls\ e \in l\ und\ e\ gelöscht\ wurde \\ false, & ansonsten \end{cases}$$

ADT IndexedList – Interface

- search (l, e)
  - **descr:** sucht ein Element in der Liste
  - **pre**: *l* ∈ *L, e* ∈ *TElem*
  - **post**:

$$\text{search} \leftarrow \begin{cases} true, & falls\ e \in l \\ false, & ansonsten \end{cases}$$

ADT IndexedList – Interface

- isEmpty (l)
  - **descr:** überprüft ob eine Liste leer ist
  - **pre**: $l \in L$
  - **post**: isEmpty $\leftarrow \begin{cases} wahr, & falls\ l = \emptyset \\ falsch, & ansonsten \end{cases}$

- size (l)
  - **descr:** gibt die Anzahl der Elemente aus der Liste zurück
  - **pre**: $l \in L$
  - **post**: size $\leftarrow$ Anzahl der Elemente aus *l*

ADT IndexedList – Interface

- destroy (l)
  - **descr:** zerstört eine Liste
  - **pre**: $l \in L$
  - **post**: l wurde zerstört

- iterator (l, it)
  - **descr:** gibt ein Iterator für die Liste zurück
  - **pre**: $l \in L$
  - **post**: $it \in I$, *it* ist ein Iterator für *l,* das aktuelle Element in dem Iterator ist das erste Element aus *l*, oder, *it* ist ungültig falls *l* leer ist

ADT IteratedList

- Wenn TPosition ein Iterator ist, dann haben wir eine IteratedList
- Die Operationen, die mit Positionen arbeiten haben in diesem Fall ein Iterator als Eingabeparameter für die Position
- Die Operationen *valid, next, previous* sind eigentlich Operationen des Iterators, diese kommen in dem Interface der IteratedList nicht mehr vor

ADT IteratedList – Interface

- init(l)
  - **descr:** erstellt eine neue, leere Liste
  - **pre:** wahr
  - **post**: *l* ∈ L, *l* ist eine leere Liste

- first(l)
  - **descr**: gibt ein Iterator zurück, der auf das erste Element zeigt
  - **pre**: *l* ∈ L
  - **post**: *first* ← *it* ∈ *Iterator*

$$\text{it} \leftarrow \begin{cases} ein\ Iterator, der\ auf\ \ das\ erste\ Element\ zeigt, & falls\ l \neq \emptyset \\ ein\ ung\ddot{u}ltiger\ Iterator, & ansonsten \end{cases}$$

ADT IteratedList – Interface

- last(l)
  - **descr**: gibt ein Iterator zurück, der auf das letzte Element zeigt
  - **pre**: $l \in L$
  - **post**: *last* ← *it* ∈ *Iterator*

$$\text{it} \leftarrow \begin{cases} ein\ Iterator, der\ auf\ \ das\ letzte\ Element\ zeigt, & falls\ l \neq \emptyset \\ ein\ ung\ddot{u}ltiger\ Iterator, & ansonsten \end{cases}$$

ADT IteratedList – Interface

- getElement(l, it)
  - **descr**: gibt das Element von der Position bezeichnet durch den Iterator zurück
  - **pre**: *l* ∈ *L*, *it* ∈ *Iterator, valid(it)*
  - **post**: getElement ← *e*, *e* ∈ *TElem*, *e* = das Element aus *l* von der aktuellen Position
  - **throws**: ein Exception, falls *it* ungültig ist

ADT IteratedList – Interface

- position(l, e)
  - **descr**: gibt ein Iterator zurück, der auf die erste Position eines Elementes gesetzt wird
  - **pre**: $l \in L$, $e \in TElem$
  - **post**: position $\leftarrow it$, $it \in Iterator$

it $\leftarrow$ $\begin{cases} \text{ein Iterator, der auf die erste Position des Elementes e aus l gesetzt wird ,} & falls\ e \in l \\ ein\ ungültiger\ Iterator, & ansonsten \end{cases}$

ADT IteratedList – Interface

- setElement (l, it, e)
  - **descr:** ersetzt das Element von der Position bezeichnet durch den Iterator mit einm neuen Element und gibt den alten Wert zurück
  - **pre**: $l \in L$, $it \in Iterator$, $e \in TElem$, $valid(it)$
  - **post**: $l' \in L$, das Element an der Position bezeichnet von *it* aus *l'* ist *e*

    setElement $\leftarrow el$, $el \in TElem$, *el* ist das Element von der Position bezeichnet von *it* aus *l*
  - **throws**: ein Exception, falls *it* ungültig ist

ADT IteratedList – Interface

- addToBeginning (l, e)
  - **descr:** fügt ein neues Element am Anfang der Liste ein
  - **pre**: $l \in L, e \in TElem$
  - **post**: $l' \in L$, das Element e wurde am Anfang der Liste eingefügt

- addToEnd (l, e)
  - **descr:** fügt ein neues Element am Ende der Liste ein
  - **pre**: $l \in L, e \in TElem$
  - **post**: $l' \in L$, das Element e wurde am Ende der Liste eingefügt

ADT IteratedList – Interface

- addToPosition (l, it, e) (oder addAfterPosition)
  - **descr:** fügt ein neues Element an der Position angegeben von dem Iterator
  - **pre**: $l \in L$, $it \in Iterator$, $e \in TElem$, $valid(it)$
  - **post**: $l' \in L$, das Element e wurde an der Position angegeben von *it* in *l* eingefügt
  - **throws**: ein Exception, falls *it* ungültig ist

ADT IteratedList – Interface

- remove (l, it)
  - **descr:** löscht das Element von der Position angegeben von dem Iterator
  - **pre**: *l* ∈ $\mathbb{L}$, *it* ∈ *Iterator, valid(it)*
  - **post**: remove ← *e, e* ∈ *TElem, e* ist das Element an der Position aneggeben von *it* aus *l, l'* ∈ $\mathbb{L}$, $l' = l - e$
  - **throws**: ein Exception, falls *it* ungültig ist

- remove (l, e)
  - **descr:** löscht das erste Vorkommen des Elementes e
  - **pre**: *l* ∈ $\mathbb{L}$, *e* ∈ *TElem*
  - **post**:

$$\text{remove} \leftarrow \begin{cases} true, & falls\ e \in l\ und\ e\ gelöscht\ wurde \\ false, & ansonsten \end{cases}$$

ADT IteratedList – Interface

- search (l, e)
  - **descr:** sucht ein Element in der Liste
  - **pre**: *l* ∈ *L, e* ∈ *TElem*
  - **post**:

$$\text{search} \leftarrow \begin{cases} true, & falls\ e \in l \\ false, & ansonsten \end{cases}$$

ADT IteratedList – Interface

- isEmpty (l)
  - **descr:** überprüft ob eine Liste leer ist
  - **pre**: $l \in L$
  - **post**: isEmpty $\leftarrow \begin{cases} wahr, & falls\ l = \emptyset \\ falsch, & ansonsten \end{cases}$

- size (l)
  - **descr:** gibt die Anzahl der Elemente aus der Liste zurück
  - **pre**: $l \in L$
  - **post**: size $\leftarrow$ Anzahl der Elemente aus *l*

ADT IteratedList – Interface

- destroy (l)
  - **descr:** zerstört eine Liste
  - **pre**: $l \in L$
  - **post**: l wurde zerstört

ADT SortedList

- In einer Liste können die Elemente basierend auf einer Ordnungsrelation sortiert werden → *SortedList*
- Operationen aus ADT List waren:
  - init(l)
  - first(l)
  - last(l)
  - valid(l, p)
  - next(l, p)
  - previous(l, p)
  - getElement(l, p)
  - position(l, e)
  - setElement(l, p, e)
  - addToBeginning(l, e)
  - addToEnd(l, e)
  - addToPosition(l, p, e)
  - remove(l, p)
  - remove(l, e)
  - search(l, e)
  - isEmpty(l)
  - size(l)
  - destroy(l)
  - iterator(l, it)
- Welche Operationen aus ADT List existieren für SortedList nicht mehr?

ADT SortedList

- Unterschiede in dem Interface:
  - *init* hat auch die Relation als Parameter
  - Es gibt nur eine Einfügeoperation, die nur das Element als Eingabeparameter braucht und nicht die Position
  - Es gibt keine setElement Operation (wenn man einen Wert ändert, dann kann es sein, dass dieser die Sortierungsreihenfolge verletzt)

- Auch für sortierte Listen gibt es die zwei Möglichkeiten für TPosition, also es gibt *SortedIndexedList* und *SortedIteratedList*

ADT List - Repräsentierung

- Um ADT List (oder ADT SortedList) zu implementieren kann man folgende Datenstrukturen für die Repräsentierung benutzen:

  - (dynamisches) Array
    - Die Elemente werden auf aufeinanderfolgende Speicherplätze gespeichert
    - Wir haben direkter Zugriff auf die Elemente

  - verkettete Liste
    - Die Elemente werden in Knoten gespeichert
    - Man hat keinen direkten Zugriff auf jedes Element

Dynamisches Array – Wiederholung

- Die Elemente in dem Array sind auf aufeinanderfolgende Positionen in dem Speicherplatz gespeichert
- Vorteile:
  - Zugriff zu einem beliebigen Element in konstanter Zeit
  - Konstante Zeit für Einfüge- oder Löschoperationen am Ende des Arrays

- Nachteile:
  - $\Theta(n)$ Komplexität für Einfüge- oder Löschoperationen am Anfang des Arrays

Verkettete Listen als Datenstruktur

- Eine *verkettete Liste* ist eine lineare Datenstruktur, wo die Reihenfolge der Elemente nicht von Indexen bestimmt wird, sondern von **Zeigern/Pointers** zwischen den Elementen
- Eine verketette Liste besteht aus:
  - **Knoten** (oder Links genannt)
  - Jeder Knoten enthält außer Daten auch ein Zeiger zu der Adresse des nächsten Knotens (und möglicherweise noch ein Zeiger zu der Adresse des vorherigen Knoten)
- Die Knoten aus einer verketteten Liste besetzen nicht unbedingt aufeinanderfolgende Speicherplätze, darum braucht man die Adresse des Nachfolgers in jedem Knoten zu speichern

Verkettete Listen

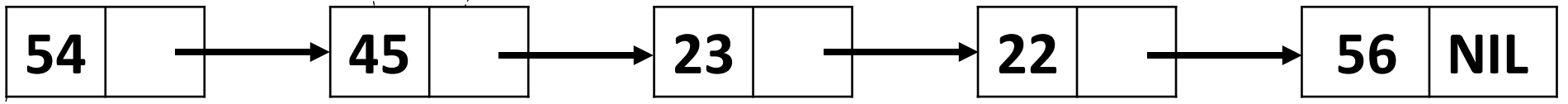
- Auf die Elemente einer verketteten Liste wird mit Hilfe der Zeiger auf die Knoten zugegriffen
- Die Ordnung der Elemente in der Liste ist **unabhängig** von ihrer physischen Position im Speicher

- Man kann **nur auf das erste Element der Liste direkt zugreifen** (und vielleicht auf das letzte Element)

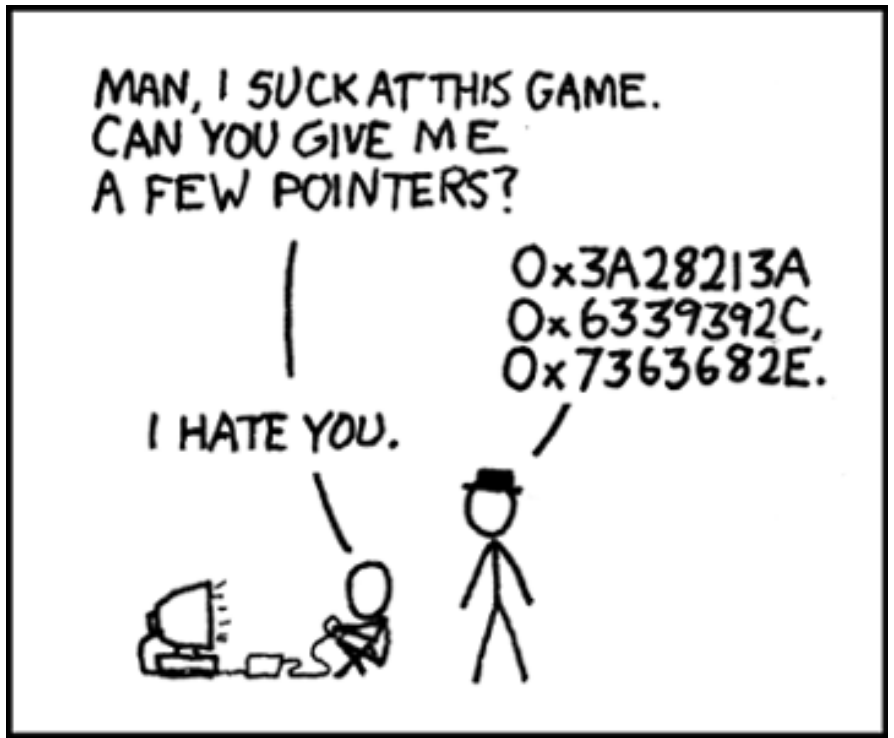
Beispiel – einfach verkettete Liste

Daten

Adresse des nächsten Knotens



Kopf der Liste
(Head of the list)



Endknoten
(Tail of the list)

Einfache Verkettete Listen – SLL (Singly Linked Lists

- In einem SLL enthält jeder Knoten die Daten und die Adresse des Nachfolgers
- Das erste Element in der Liste wird als **Listenkopf** oder **Head** bezeichnet und das letzte als **Listenende** oder **Tail**
- Das Listenende enthält den speziellen Wert NIL für die Adresse des Nachfolgers (diese ist also ungültig, da es kein Nachfolger mehr gibt)
- Falls der Listenkopf den Wert NIL enthält, dann ist die Liste leer

Einfache Verkettete Listen - Repräsentierung

- Für die Repräsentierung einer einfachen 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 - Operationen

- Meistens speichert man für ein SLL nur die Adresse des ersten Elementes (head), aber falls nötig kann man auch die Adresse des letzten Elementes (tail) speichern
- Mögliche Operationen für eine einfach verkettete Liste:
  - Suche ein Element mit einem gegebenen Wert
  - Füge ein Element ein: am Anfang der Liste, am Ende der Liste, auf eine gegebene Position, nach einem bestimmten Wert
  - Lösche ein Element: vom Anfang der Liste, vom Ende der Liste, von einer bestimmten Position, mit einem gegebenen Wert
  - Gib ein Element an einer bestimmten Position zurück
- Meistens braucht man nicht alle Operationen; es hängt davon ab, welches Container man mit der SLL implementiert

SLL - Suche

```
function search (sll, elem) is:
//pre: sll ist eine SLL; elem ist ein TElem
//post: gibt den Knoten zurück, der die gesuchte Daten (elem) enthaltet
//          oder NIL falls es kein Knoten gibt
    current ← sll.head
    while current ≠ NIL and [current].info ≠ elem execute
            current ← [current].next
    end-while
    search ← current
end-function
```

- Komplexität:

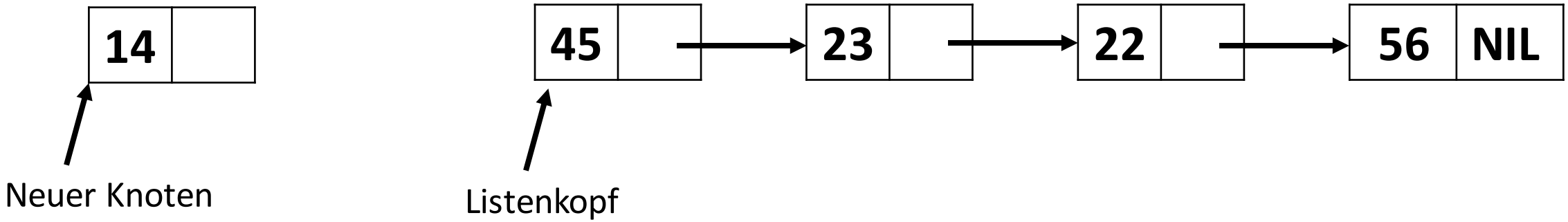
O(n) – man kann das gesuchte Element im ersten Knoten finden oder es kann sein dass man die ganze Liste durchlaufen muss

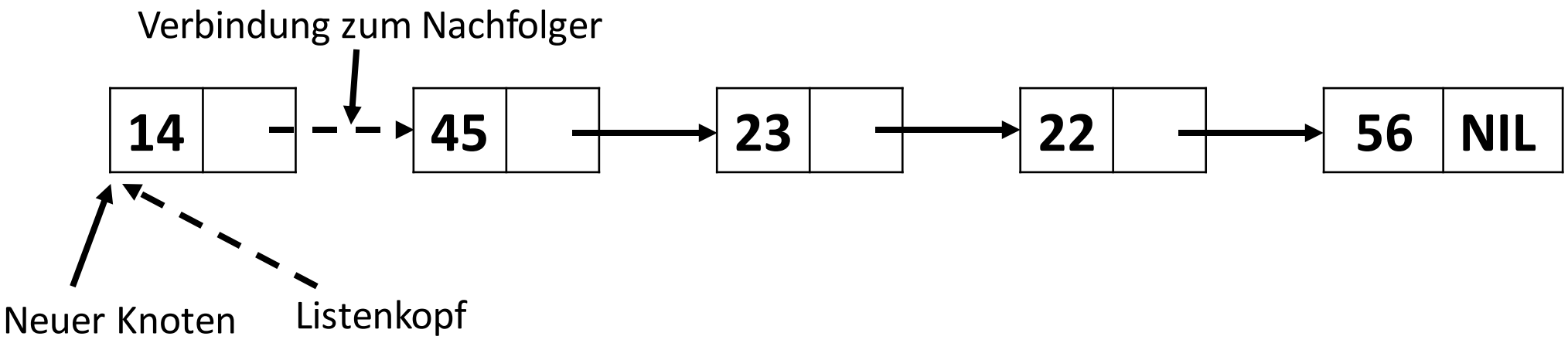
Eine einfache verkettete Liste durchlaufen

- Eine einfache verkettete Liste durchlaufen :
  - Man braucht einen zusätzlichen Hilfsknoten (*current)*, der von dem Listenkopf anfängt
  - Bei jedem Schritt wird dem Hilfsknoten die Adresse des Nachfolgers zugewiesen:

    $current \leftarrow [current].next$
  - man wiederholt den vorigen Schritt bis der Hilfsknoten NIL ist

SLL – am Anfang der Liste einfügen





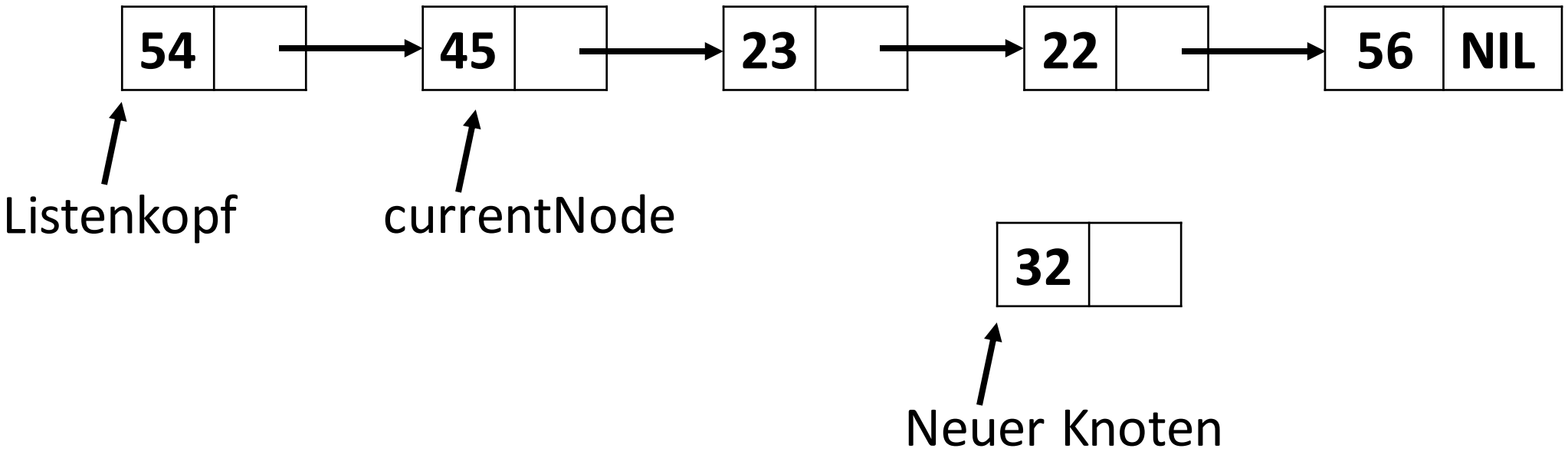
SLL – am Anfang der Liste einfügen

subalgorithm insertFirst (sll, elem) **is**:
//pre: sll ist eine SLL; elem ist ein TElem
//post: das Element *elem* wird am Anfang von *sll* eingefügt
    newNode ← allocate() //einen neuen SLLNode allokieren
    [newNode].info ← elem
    [newNode].next ← sll.head
    sll.head ← newNode
end-subalgorithm

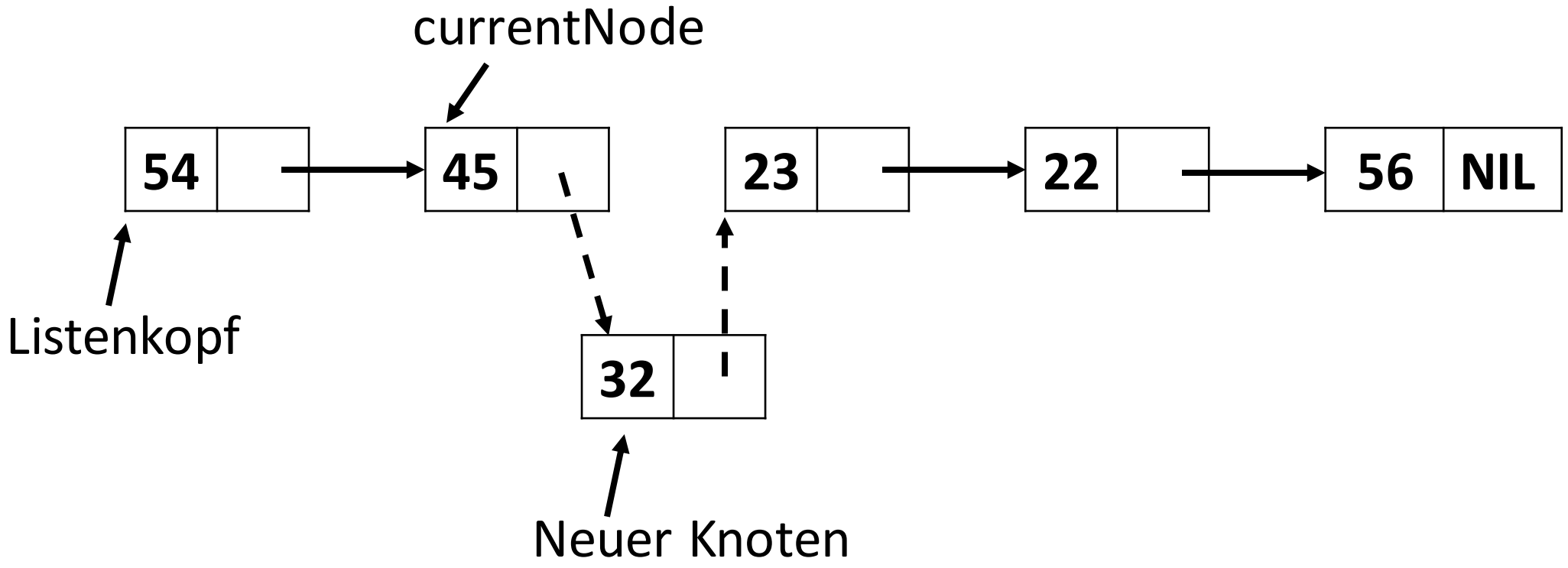
- Komplexität: $\Theta(1)$

SLL – nach einem Knoten einfügen

- Man nimmt an, dass man die Adresse eines Knotens kennt und man will einen neuen Knoten nach dem gegebenen Knoten einfügen



SLL – nach einem Knoten einfügen



SLL – nach einem Knoten einfügen

subalgorithm insertAfter(sll, currentNode, elem) **is**:
//pre: *sll* ist eine SLL; *currentNode* ist ein SLLNode aus *sll*; *elem* ist ein TElem
//post: ein Knoten mit dem Wert *elem* wird nach dem Knoten *currentNode*
// eingefügt
  newNode ← allocate() //einen neuen SLLNode allokieren
  [newNode].info ← elem
  [newNode].next ← [currentNode].next
  [currentNode].next ← newNode
end-subalgorithm

- Kann man die letzten zwei Anweisungen vertauschen?
- Komplexität: $\Theta(1)$

Vor einem Knoten einfügen

Zum Nachdenken

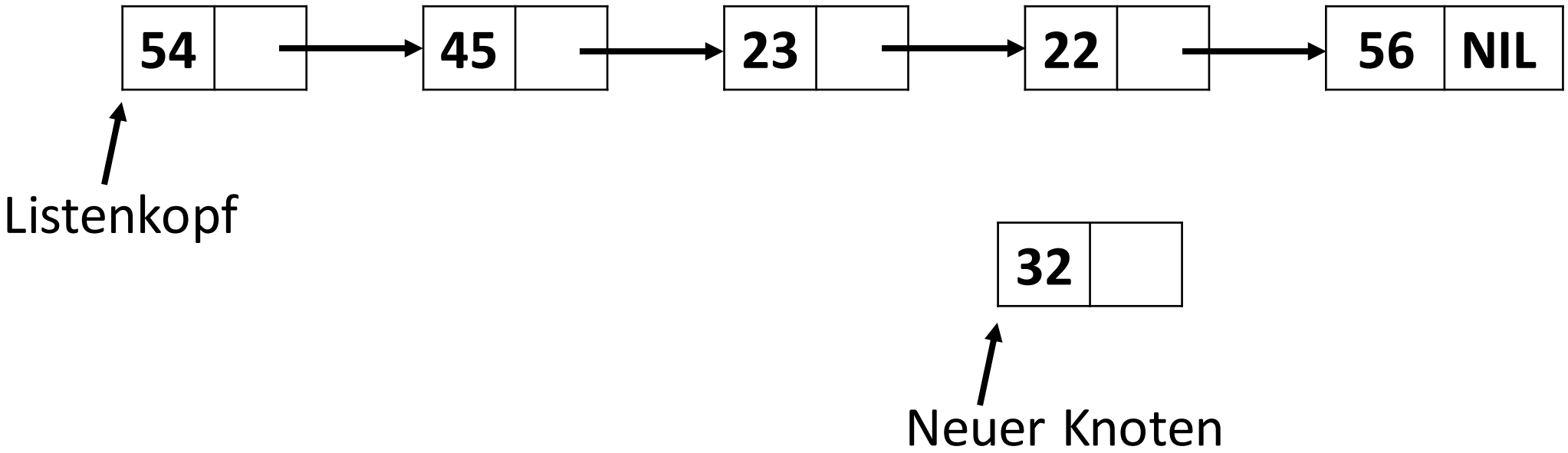
- Wenn ein Knoten angegeben wird, wie kann man ein neues Element vor diesem Knoten in der einfach verketteten Liste einfügen?

SLL - an eine bestimmte Position einfügen

- Meistens kennt man nicht den Knoten, nach dem man das neue Element einfügen will, sondern man kennt entweder:
  - die Position, an der man das neue Element einfügen will, oder
  - den Wert, nach dem man das neue Element einfügen will
- Angenommen, man will ein neues Element an der ganzzahligen Position *p* einfügen (nach dem Einfügen befindet sich das neue Element an der Position *p*):
  - Da man nur Zugriff auf den Listenkopf hat, muss man erstmal die Position finden, nach welcher man das neue Element einfügen will

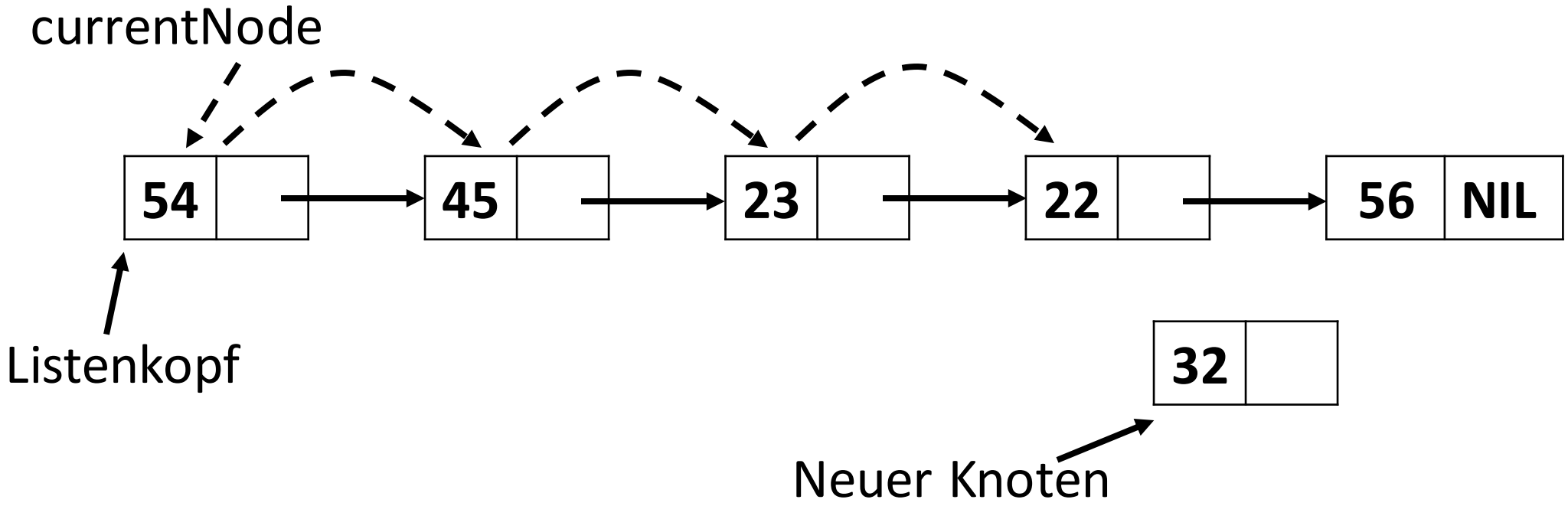
SLL - an eine bestimmte Position einfügen

- Man will das Element 32 an der Position 5 einfügen



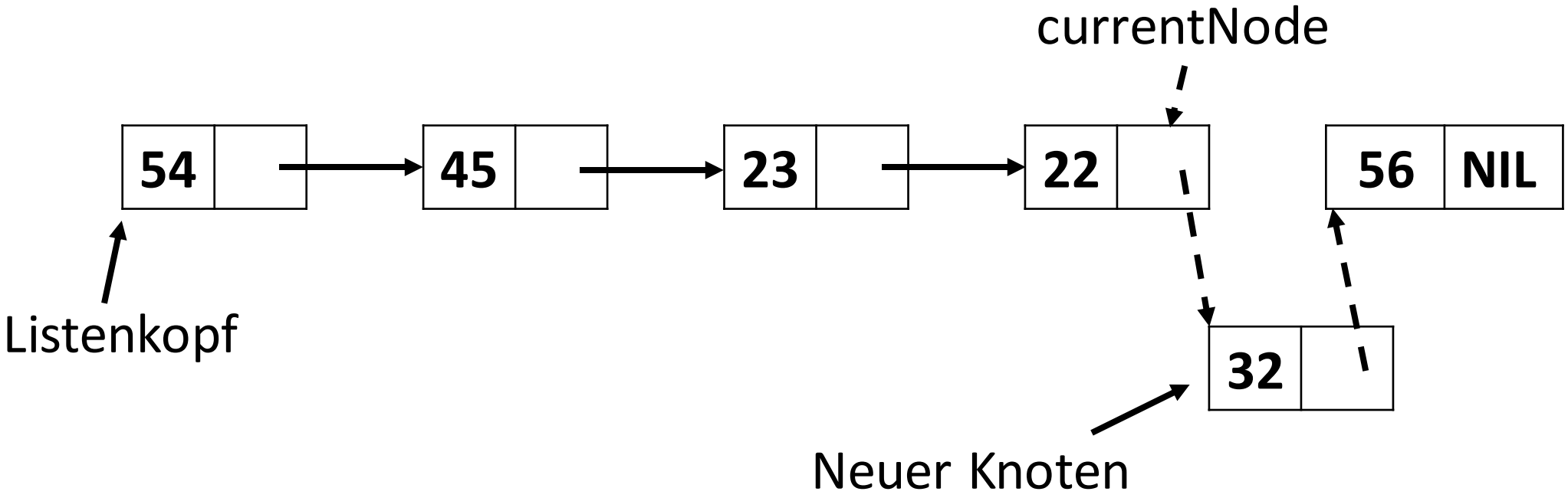
SLL - an eine bestimmte Position einfügen

- Wir suchen den vierten Knoten, um den neuen Knoten danach einzufügen
- Wir brauchen einen Hilfsknoten *(currentNode)* um die Liste durchzulaufen



SLL - an eine bestimmte Position einfügen

- Jetzt fügt man das neue Element nach *currentNode* ein (wie bei der vorigen Einfügeoperation)



SLL - an eine bestimmte Position einfügen

```
subalgorithm insertPosition(sll, pos, elem) is:
//pre: sll ist eine SLL; pos ist eine ganze Zahl; elem ist ein TElem
//post: ein Knoten mit TElem wird an der Position pos eingefügt
    if pos < 1 then
        @error, invalid position
    else if pos = 1 then    //dann will man am Anfang der Liste einfügen
        newNode ← allocate()        //einen neuen SLLNode allokieren
        [newNode].info ← elem
        [newNode].next ← sll.head
        sll.head ← newNode
    else
        currentNode ← sll.head
        currentPos ← 1
//Fortsetzung auf der nächsten Folie
```

```
        while currentPos < pos - 1 and currentNode ≠ NIL execute
            currentNode ← [currentNode].next
            currentPos ← currentPos + 1
        end-while
        if currentNode ≠ NIL then                    //insertAfter
            newNode ← allocate()    //einen neuen SLLNode allokieren
            [newNode].info ← elem
            [newNode].next ← [currentNode].next
            [currentNode].next ← newNode
        else
            @error, invalid position
        end-if
    end-if
end-subalgorithm
```

- Komplexität: $O(n)$

SLL - vom Anfang der Liste löschen

- Um einen Knoten vom Anfang der Liste zu löschen, muss man den Listenkopf zu dem nächsten Element verschieben
- Folgende Funktion gibt den gelöschten Knoten zurück

SLL - vom Anfang der Liste löschen

```
function deleteFirst(sll) is:
//pre: sll ist ein SLL
//post: der erste Knoten aus sll wurde gelöscht und zurückgegeben
        deletedNode ← NIL
        if sll.head ≠ NIL then
                deletedNode ← sll.head
                sll.head ← [sll.head].next
                [deletedNode].next ← NIL
        end-if
        deleteFirst ← deletedNode
end-function
```

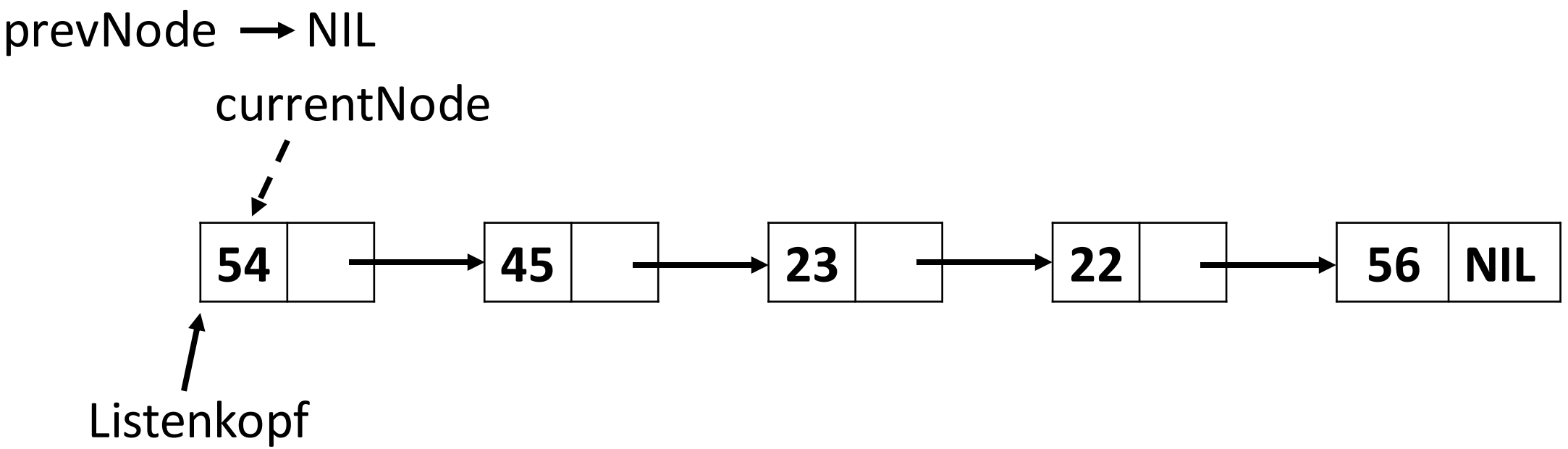
- Komplexität: $\Theta(1)$

SLL – einen gegebenen Wert löschen

- Wenn man einen Knoten irgendwo in der Mitte der Liste löschen will (egal ob man den Wert oder die Position kennt), dann muss man erstmal den Knoten davor finden
- Am einfachsten ist es die Liste mithilfe von zwei Pointers zu durchlaufen: *currentNode* und *prevNode* (für den Knoten bevor *currentNode)*
- Man hört auf wenn *currentNode* auf den Knoten zeigt, den man löschen will

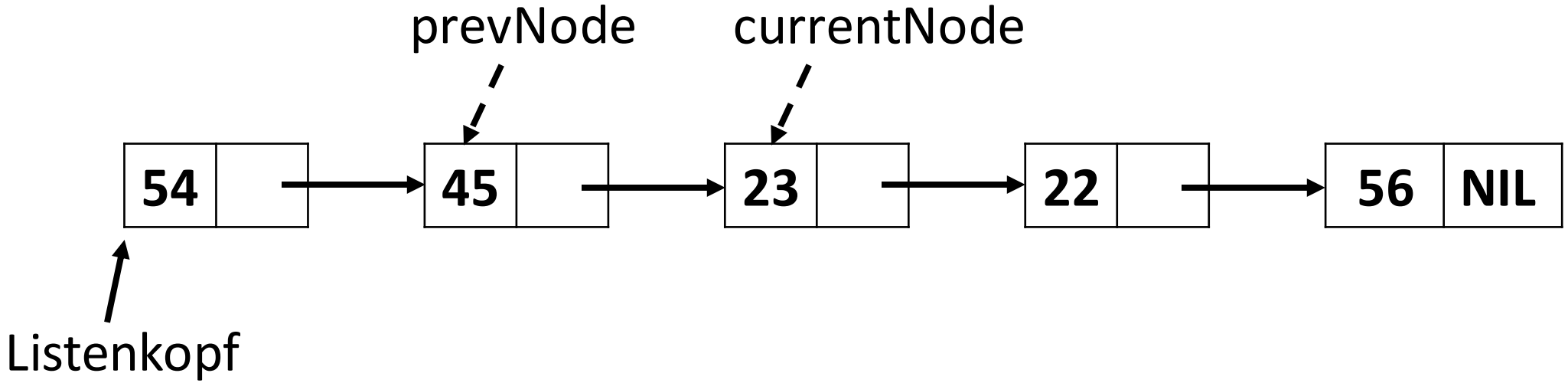
SLL – einen gegebenen Wert löschen

- Man will den Knoten mit dem Wert 23 löschen



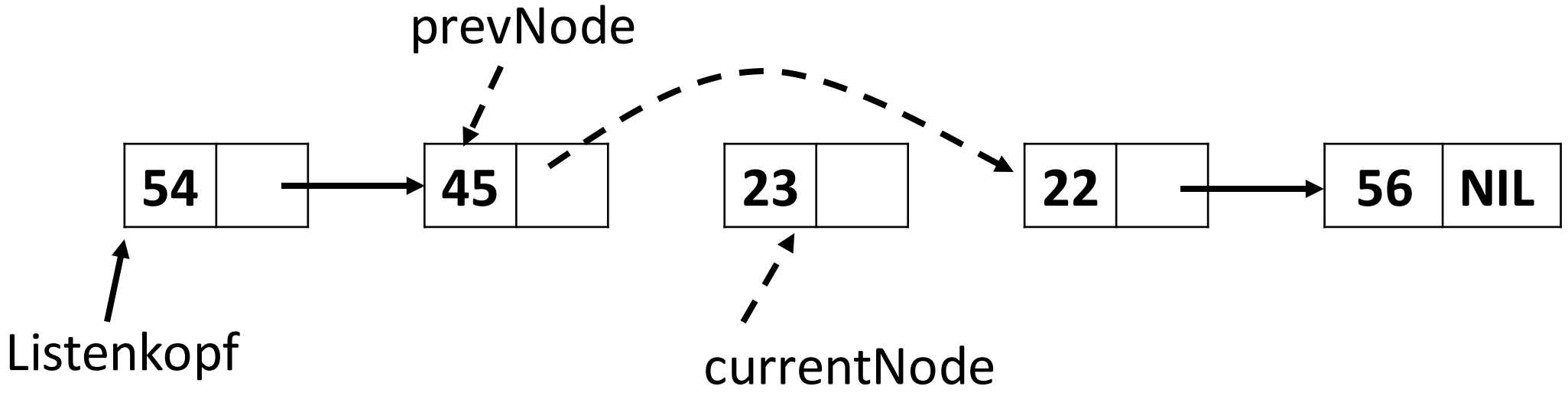
SLL – einen gegebenen Wert löschen

- Man verschiebt die zwei Pointers bis *currentNode* auf den Knoten zeigt, den man löschen muss



SLL – einen gegebenen Wert löschen

- Man löscht *currentNode* indem man den Knoten überspringt



SLL – einen gegebenen Wert löschen

```
function deleteElement(sll, elem) is:
//pre: sll ist ein SLL, elem ist ein TElem
//post: der Knoten mit dem Wert elem wurde gelöscht und zurückgegeben
    currentNode ← sll.head
    prevNode ← NIL
    while currentNode ≠ NIL and [currentNode].info ≠ elem execute
        prevNode ← currentNode
        currentNode ← [currentNode].next
    end-while
    if currentNode ≠ NIL and prevNode = NIL then     //dann löscht man den Listenkopf
        sll.head ← [sll.head].next
    else if currentNode ≠ NIL then
        [prevNode].next ← [currentNode].next
        [currentNode].next ← NIL
    end-if
    deleteElement ← currentNode
end-function
```

SLL – einen gegebenen Wert löschen

- Komplexität für die Funktion *deleteElement:*

  O(n)

SLL – andere Operationen

- Ein Element am Ende der Liste einfügen:
  - Die Liste durchlaufen bis zu dem letzten Knoten, und einen Knoten nach dem letzten Knoten einfügen
- Ein Element vom Ende der Liste löschen:
  - Die Liste mithilfe von zwei Pointers durchlaufen bis zu dem letzten Knoten
  - Den letzten Knoten löschen indem man für den Vorgänger das *next* Feld aus NIL setzt
- Ein Element an einer gegebenen Position zurückgeben:
  - Die Liste durchlaufen bis man an der gewünschten Position ankommt um den Knoten zurückzugeben