# Kapitel ADS:IV

## IV. Datenstrukturen

- Record
- Container
- List
- Linked List
- Stack
- Queue
- Priority Queue
- Hash Table
- Hash Function
- Bäume

# List

## Definition

Eine Liste ist eine Datenstruktur, die Elemente in einer bestimmten Reihenfolge anordnet.

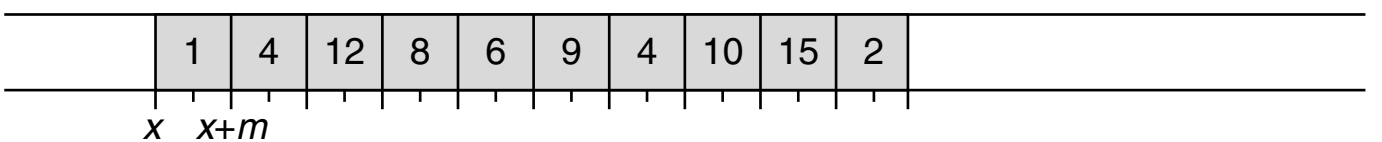
# List

## Definition

Eine Liste ist eine Datenstruktur, die Elemente in einer bestimmten Reihenfolge anordnet.

## Implementierung

- Sequentielle Allokation
  Ablegen der Elemente in der gewünschten Reihenfolge an konsekutiven Speicherstellen.



# List

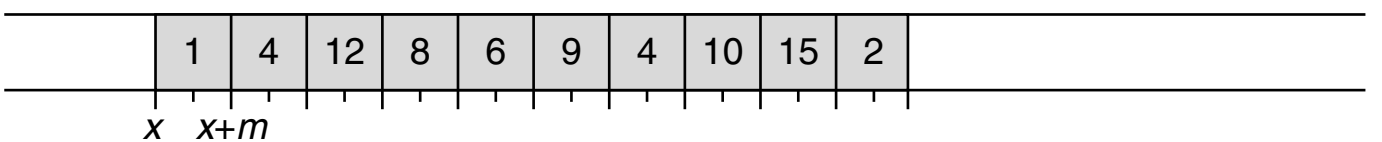
## Definition

Eine Liste ist eine Datenstruktur, die Elemente in einer bestimmten Reihenfolge anordnet.

## Implementierung

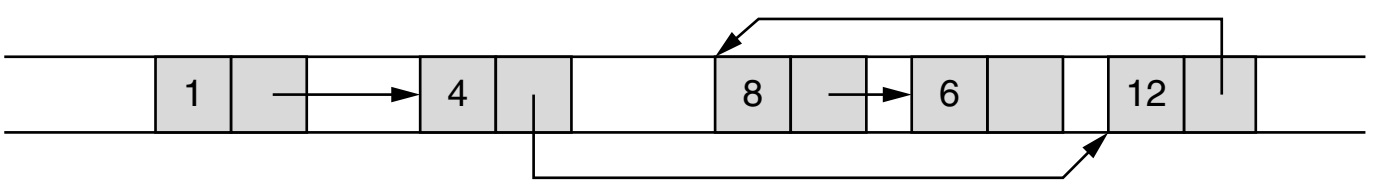
- Sequentielle Allokation

  Ablegen der Elemente in der gewünschten Reihenfolge an konsekutiven Speicherstellen.




- Verkettete Allokation

  Erzeugen eines Records für jedes Element, das je ein Element sowie je einen Pointer auf das Record des jeweils in der gewünschten Reihenfolge nachfolgenden Elements enthält. Records können an beliebigen Speicherstellen abgelegt werden. Es genügt, einen Pointer auf das erste Record zu verwalten.



# List

## Manipulation

| Operation | Listenimplementierung | | | |
|---|---|---|---|---|
| | Array-basiert | | Link-basiert | |
| | statisch | dynamisch | Rand | Mitte |
| Element lesen / modifizieren | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ | $O(n)$ |
| Element einfügen / löschen | $\Theta(n)$ | $\Theta(n)$* | $\Theta(1)$ | $\Theta(1)$** |
| Listen konkatenieren | $\Theta(n)$ | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ |
| Listen aufteilen | $\Theta(n)$ | $\Theta(n)$ | – | $\Theta(1)$ |
| Listenlänge ermitteln | $\Theta(1)$ | $\Theta(1)$ | $\Theta(n)$*** | |
| Liste sortieren | $\Omega(n \lg n)$**** | | | |
| Größtes / kleinstes Element ermitteln | $\Theta(n)$ | | | |
| Element per Index suchen | $\Theta(1)$ | | $\Theta(1)$ | $O(n)$ |
| Element per Schlüssel suchen | $\Theta(n)$ | | | |

* Am rechten Rand des Arrays: $\Theta(1)$ (amortisiert).

** Bei Singly Linked Lists zuzüglich Suchzeit für das Element vor dem neu einzufügenden.

*** Effektiv $\Theta(1)$, da die Listengröße $n$ wie beim Array als Variable protokolliert werden kann.

**** Abhängig vom Sortieralgorithmus und seiner Tauglichkeit für die Listenimplementierung.

Bemerkungen:

- Programmiersprachen bieten für beide Allokationsformen Hilfestellungen: Für die sequentielle Allokation von Elementen gleichen Typs dienen Arrays und für die verkettete Allokation dienen Pointer und Objekte.
- Die größere Flexibilität der verketteten Allokation kann sich negativ auf die Laufzeit auswirken, da regelmäßig auf verschiedene Positionen im Speicher zugegriffen werden muss, was das Caching von Daten sowie das vorauseilende Nachladen von wahrscheinlich in Zukunft benötigten Daten aufwändiger macht.
- Die besseren Laufzeiten für Operationen auf link-basierten Listen gegenüber array-basierten Listen gehen mit einem größeren Platzverbrauch für die Pointer einher.

# Linked List

## Definition

Eine Linked List (*verkettete Liste*) ist eine Liste, die mit verketteter Allokation implementiert wurde.

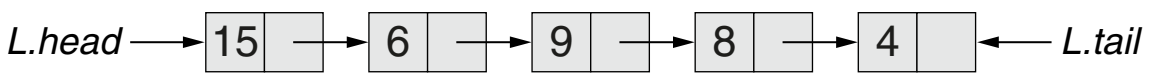
# Linked List

## Definition

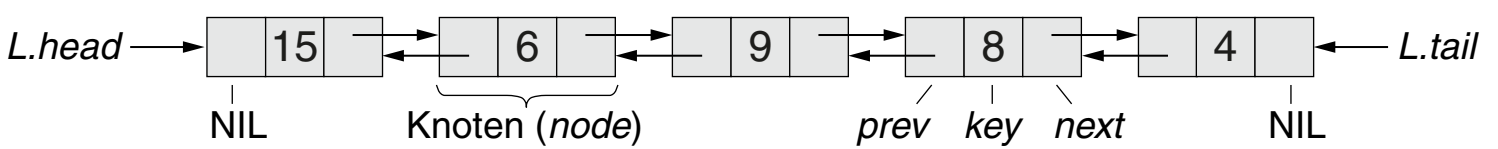
Eine Linked List (*verkettete Liste*) ist eine Liste, die mit verketteter Allokation implementiert wurde.

Varianten:

1. Singly Linked List (*Einfach verkettete Liste*)




2. Doubly Linked List (*Doppelt verkettete Liste*)



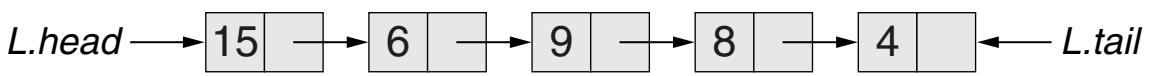
# Linked List

## Definition

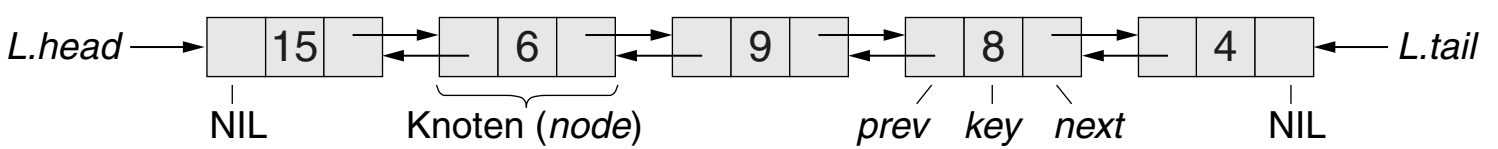
Eine Linked List (*verkettete Liste*) ist eine Liste, die mit verketteter Allokation implementiert wurde.

Varianten:

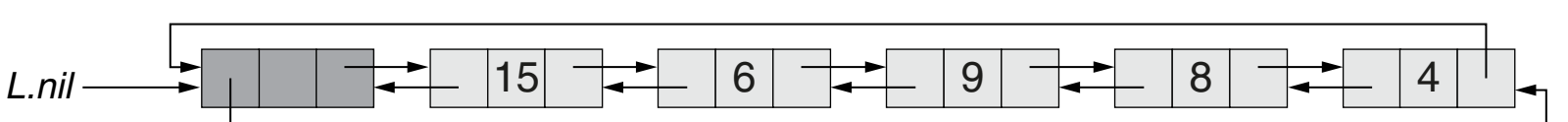
1. Singly Linked List (*Einfach verkettete Liste*)




2. Doubly Linked List (*Doppelt verkettete Liste*)



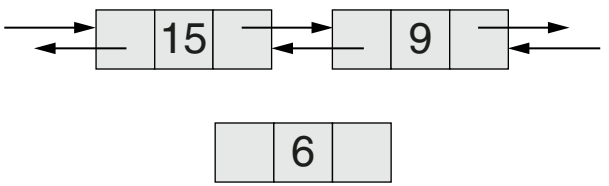

3. Circular Doubly Linked List mit Sentinel



# Linked List

## Manipulation: Beispiel

Einfügen:



# Linked List

## Manipulation: Beispiel

Einfügen:

# Linked List

## Manipulation: Beispiel

Einfügen:


Löschen:


Grenzfälle an den Rändern einer Liste müssen gesondert behandelt werden.

# Linked List

## Manipulation: Variante 2

Eingabe: $L$. Doubly Linked List.
$x$. Pointer zum einzufügenden / zu löschenden Knoten.
$k$. Schlüssel des zu suchenden Knotens.

Ausgabe: $x$. Pointer zum gefundenen Knoten.

*InsertFront*$(L, x)$

```
1.  x.next = L.head
2.  IF L.head ≠ NIL THEN
3.     L.head.prev = x
4.  ENDIF
5.  L.head = x
6.  x.prev = NIL
```

*Delete*$(L, x)$

```
1.  IF x.prev ≠ NIL THEN
2.     x.prev.next = x.next
3.  ELSE
4.     L.head = x.next
5.  ENDIF
6.  IF x.next ≠ NIL THEN
7.     x.next.prev = x.prev
8.  ELSE
9.     L.tail = x.prev
10. ENDIF
```

*SearchKey*$(L, k)$

```
1.  x = L.head
2.  WHILE x ≠ NIL
       AND x.key ≠ k DO
3.     x = x.next
4.  ENDDO
5.  return(x)
```

# Linked List

## Manipulation: Variante 3

Eingabe: $L$. Circular Doubly Linked List mit Sentinel.
$x$. Pointer zum einzufügenden / zu löschenden Knoten.
$k$. Wert des zu suchenden Elements.

Ausgabe: $x$. Pointer zum gefundenen Knoten.

*InsertFront*$(L, x)$

1. $x.\mathit{next} = L.\mathit{nil}.\mathit{next}$
2. $L.\mathit{nil}.\mathit{next}.\mathit{prev} = x$
3. $L.\mathit{nil}.\mathit{next} = x$
4. $x.\mathit{next} = L.\mathit{nil}$

*Delete*$(L, x)$

1. $x.\mathit{prev}.\mathit{next} = x.\mathit{next}$
2. $x.\mathit{next}.\mathit{prev} = x.\mathit{prev}$

*SearchKey*$(L, k)$

1. $x = L.\mathit{nil}.\mathit{next}$
2. **WHILE** $x \neq L.\mathit{nil}$ **AND** $x.\mathit{key} \neq k$ **DO**
3. $x = x.\mathit{next}$
4. **ENDDO**
5. *return*$(x)$

# Stack

## Definition

Eine Datenstruktur heißt Stack (*Stapel*), wenn Elemente in umgekehrter Reihenfolge, in der sie abgelegt wurden, wieder entnommen werden können.

# Stack

## Definition

Eine Datenstruktur heißt Stack (*Stapel*), wenn Elemente in umgekehrter Reihenfolge, in der sie abgelegt wurden, wieder entnommen werden können.

## Manipulation

- Ablegen (*Push*)
  Ein Element auf dem Stack ablegen.
- Entnehmen (*Pop*)
  Ein Element vom Stapel nehmen.
- Vorschau (*Top*)
  Das oberste Element auf dem Stapel betrachten.

# Stack

## Definition

Eine Datenstruktur heißt Stack (*Stapel*), wenn Elemente in umgekehrter Reihenfolge, in der sie abgelegt wurden, wieder entnommen werden können.

## Manipulation

- Ablegen (*Push*)
  Ein Element auf dem Stack ablegen.
- Entnehmen (*Pop*)
  Ein Element vom Stapel nehmen.
- Vorschau (*Top*)
  Das oberste Element auf dem Stapel betrachten.

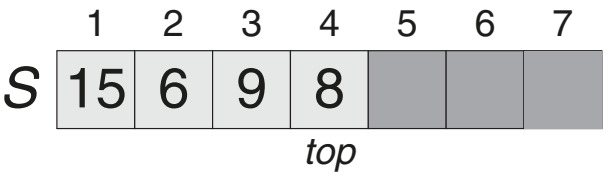
## Implementierung und Konstruktion

- Array-basiert
  Initialisierung eines Arrays; Ablage und Entnahme von Elementen in Reihenfolge im Array.
- Link-basiert
  Initialisierung einer leeren Liste; Anfügen am bzw. Entnahme vom Anfang der Liste.

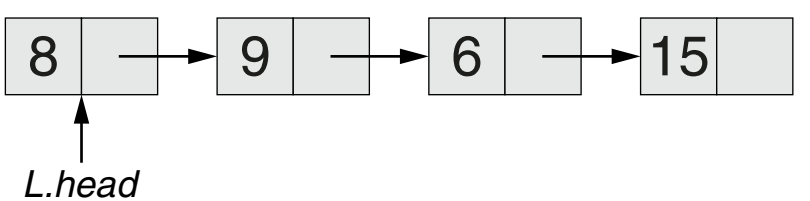
# Stack

## Manipulation: Beispiel

Array-Stack:



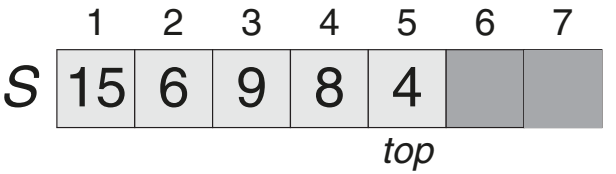

List-Stack:



# Stack

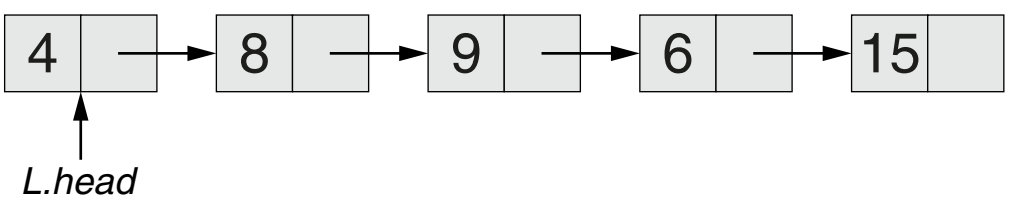
## Manipulation: Beispiel

Array-Stack:




*Push*$(S, 4)$

List-Stack:

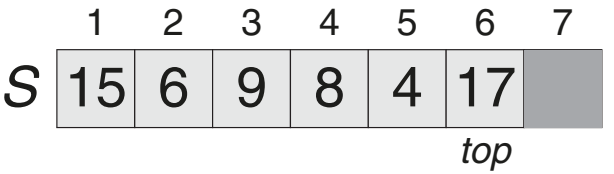



*Push*$(L, 4)$

# Stack

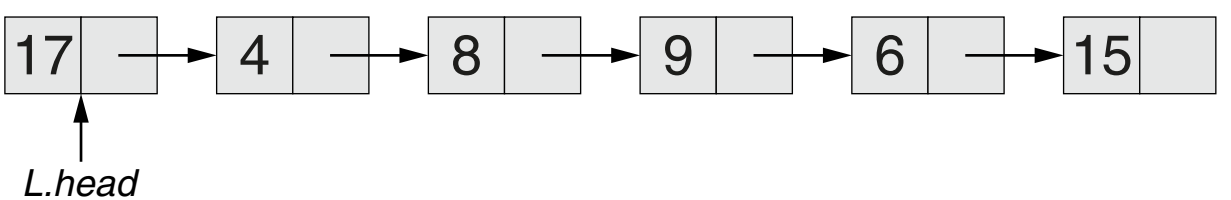
## Manipulation: Beispiel

Array-Stack:




*Push*$(S, 4)$; *Push*$(S, 17)$

List-Stack:

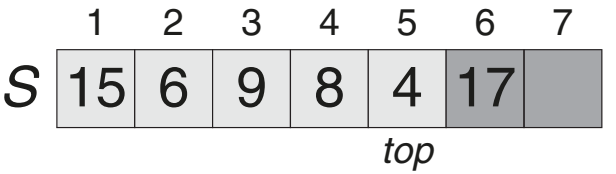



*Push*$(L, 4)$; *Push*$(L, 17)$

# Stack

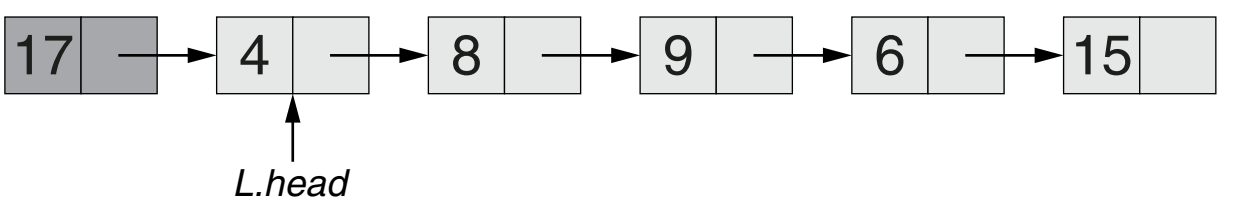
## Manipulation: Beispiel

Array-Stack:




*Push*$(S, 4)$; *Push*$(S, 17)$; *Pop*$(S)$

List-Stack:




*Push*$(L, 4)$; *Push*$(L, 17)$; *Pop*$(L)$

# Stack

## Manipulation: Array-basiert

Eingabe: $S$. Array der Länge $m$ mit zusätzlichem Attribut *top*.
$k$. Einzufügender Schlüssel.

Ausgabe: $k$. Oberster Schlüssel.

*Push*$(S, k)$

```
1. IF S.top == m THEN
2.   error("overflow")
3. ENDIF
4. S.top = S.top + 1
5. S[S.top] = k
```

*Pop*$(S)$

```
1. IF S.top == 0 THEN
2.   error("underflow")
3. ENDIF
4. S.top = S.top − 1
5. return(S[S.top + 1])
```

*Top*$(S)$

```
1. IF S.top == 0 THEN
2.   error("underflow")
3. ENDIF
4. return(S[top])
```

# Stack

## Manipulation: Link-basiert

Eingabe: $L$. Singly Linked List.
$k$. Einzufügender Schlüssel.

Ausgabe: $k$. Oberster Schlüssel.

*Push*$(L, k)$

```
1. x = node(k)
2. x.next = L.head
3. L.head = x
```

*Pop*$(L)$

```
1. IF L.head == NIL THEN
2.   error("underflow")
3. ENDIF
4. x = L.head
5. L.head = L.head.next
6. return(x.key)
```

*Top*$(L)$

```
1. IF L.head == NIL THEN
2.   error("underflow")
3. ENDIF
4. return(L.head.key)
```

Bemerkungen:

- Laufzeiten:
  - Push: $\Theta(1)$
  - Pop: $\Theta(1)$
  - Top: $\Theta(1)$
- Stacks werden in anderen Anwendungskontexten auch „Keller“ genannt.
- Stacks arbeiten nach dem sogenannten LIFO-Prinzip: „Last in, first out“.
- Die Hilfsfunktion *node*$(k)$ erzeugt einen neuen Listenknoten mit Schlüssel $k$.
- Bei Link-basierten Stacks kann immernoch ein Stack Overflow auftreten, und zwar wenn der dem Prozess zur Verfügung stehende Speicher voll ist. Das führt dazu, dass der Aufruf der Hilfsfunktion *node* beim Versuch, Speicherplatz für einen neuen Listenknoten zu allozieren, fehlschlägt.
- Wichtige Anwendungen von Stacks:
  - Berechenbarkeitsmodell Stack Machine (als Alternative zur Registermaschine)
  - Funktionsaufrufe in Programmiersprachen (insbesondere Rekursion)
  - Syntaxprüfung von kontextfreien Sprachen (Compilerbau)
  - Traversierung von Graphen (Tiefensuche)
  - Undo-Redo-Funktionen in Endanwendersoftware

# Queue

## Definition

Eine Datenstruktur heißt Queue (*Warteschlange*), wenn Elemente in der Reihenfolge, in der sie eingereiht wurden, wieder entnommen werden können.

# Queue

## Definition

Eine Datenstruktur heißt Queue (*Warteschlange*), wenn Elemente in der Reihenfolge, in der sie eingereiht wurden, wieder entnommen werden können.

## Manipulation

- Einreihen (*Enqueue*)
  Ein Element ans Ende der Warteschlange einreihen.
- Entnehmen (*Dequeue*)
  Ein Element vom Anfang der Warteschlange entnehmen.

# Queue

## Definition

Eine Datenstruktur heißt Queue (*Warteschlange*), wenn Elemente in der Reihenfolge, in der sie eingereiht wurden, wieder entnommen werden können.

## Manipulation

- Einreihen (*Enqueue*)
  Ein Element ans Ende der Warteschlange einreihen.
- Entnehmen (*Dequeue*)
  Ein Element vom Anfang der Warteschlange entnehmen.

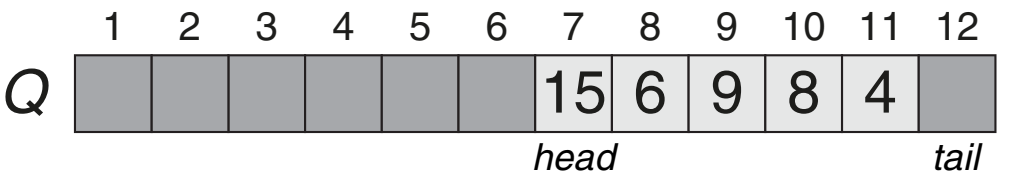
## Implementierung und Konstruktion

- Array-basiert
  Initialisierung eines Arrays; Verwaltung eines umlaufenden Teilarrays als Warteschlange von Elementen.
- Link-basiert
  Initialisierung einer leeren Liste; Einreihen von Elementen am Ende der Liste und Entnehmen von Elementen vom Kopf.

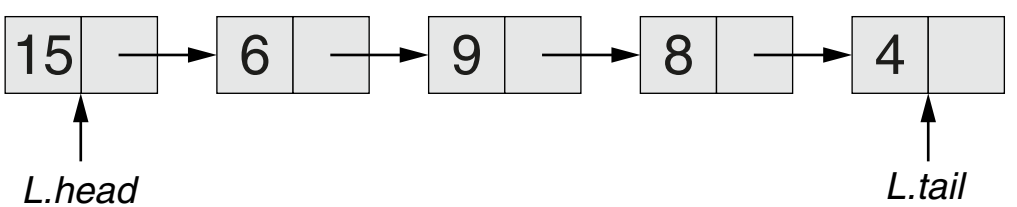
# Queue

## Manipulation: Beispiel

Array-Queue:



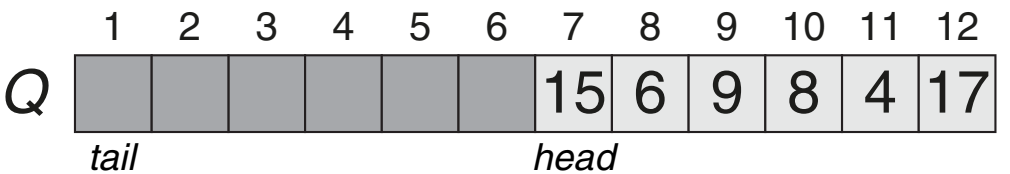

List-Queue:



# Queue

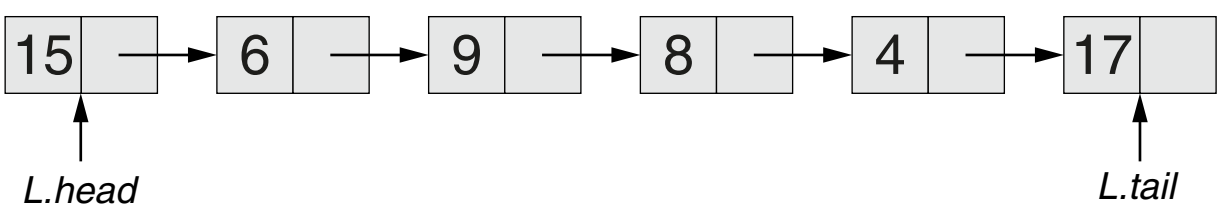
## Manipulation: Beispiel

Array-Queue:




*Enqueue*$(Q, 17)$

List-Queue:

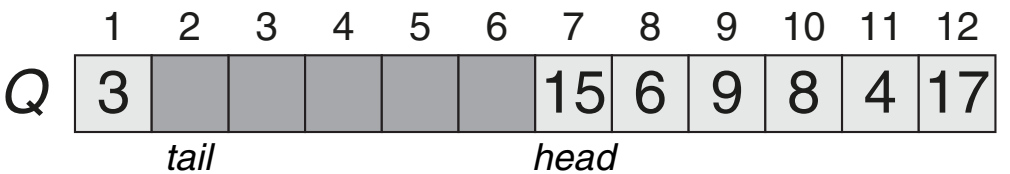



*Enqueue*$(L, 17)$

# Queue

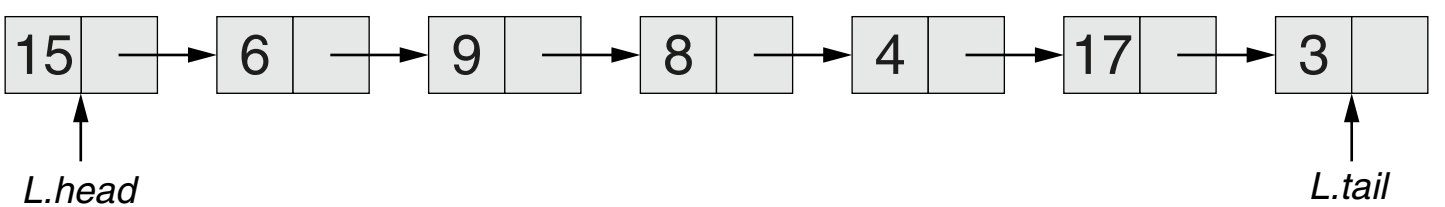
## Manipulation: Beispiel

Array-Queue:




*Enqueue*$(Q, 17)$; *Enqueue*$(Q, 3)$

List-Queue:

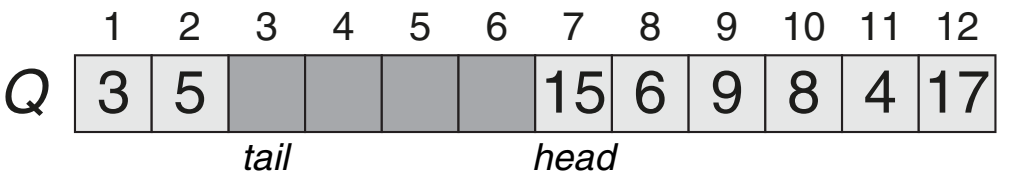



*Enqueue*$(L, 17)$; *Enqueue*$(L, 3)$

# Queue

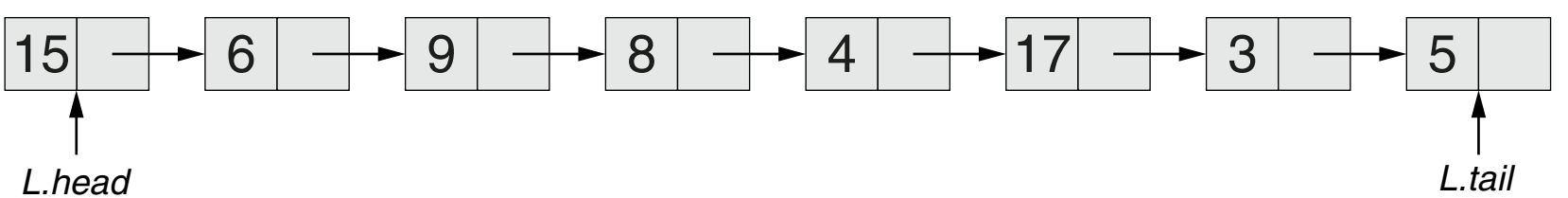
## Manipulation: Beispiel

Array-Queue:




*Enqueue*$(Q, 17)$; *Enqueue*$(Q, 3)$; *Enqueue*$(Q, 5)$

List-Queue:

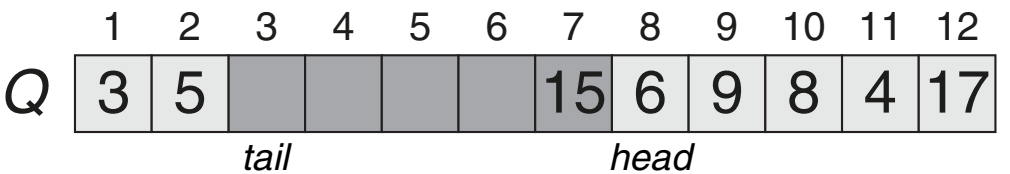



*Enqueue*$(L, 17)$; *Enqueue*$(L, 3)$; *Enqueue*$(L, 5)$

# Queue

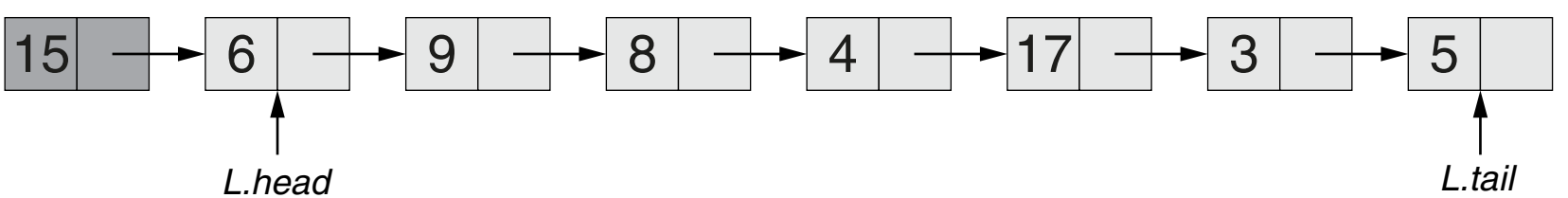
## Manipulation: Beispiel

Array-Queue:




*Enqueue*$(Q, 17)$; *Enqueue*$(Q, 3)$; *Enqueue*$(Q, 5)$; *Dequeue*$(Q)$

List-Queue:




*Enqueue*$(L, 17)$; *Enqueue*$(L, 3)$; *Enqueue*$(L, 5)$; *Dequeue*$(L)$

# Queue

## Manipulation: Array-basiert

Eingabe: $Q$. Array der Länge $m$ mit zusätzlichen Attributen *head* und *tail*.
$k$. Einzufügender Schlüssel.

Ausgabe: $k$. Erster Schlüssel.

*Enqueue*$(Q, x)$

```
1. IF Q.head == Q.tail + 1 THEN
2.   error("overflow")
3. ENDIF
4. Q[Q.tail] = k
5. IF Q.tail == m THEN
6.   Q.tail = 1
7. ELSE
8.   Q.tail = Q.tail + 1
9. ENDIF
```

*Dequeue*$(Q)$

```
1. IF Q.head == Q.tail THEN
2.   error("underflow")
3. ENDIF
4. k = Q[Q.head]
5. IF Q.head == m THEN
6.   Q.head = 1
7. ELSE
8.   Q.head = Q.head + 1
9. ENDIF
10. return(k)
```

# Queue

## Manipulation: Link-basiert

Eingabe: $L$. Singly Linked List.
$k$. Einzufügender Schlüssel.

Ausgabe: $k$. Erster Schlüssel.

*Enqueue*$(L, x)$

```
1. x = node(k)
2. IF L.tail == NIL THEN
3.    L.head = x
4.    L.tail = x
5. ELSE
6.    L.tail.next = x
7.    L.tail = x
8. ENDIF
```

*Dequeue*$(L)$

```
1. IF L.head == NIL THEN
2.    error("underflow")
3. ENDIF
4. x = L.head
5. L.head = L.head.next
6. IF L.tail == x THEN
7.    L.tail = NIL
8. ENDIF
9. return(x.key)
```

Bemerkungen:

- Laufzeiten:
  - Enqueue: $\Theta(1)$
  - Dequeue: $\Theta(1)$
- Queues arbeiten nach dem sogenannten FIFO-Prinzip: „First in, first out“.
- Eine Variante der Queue ist die Double-ended Queue (auch „Deque“; gesprochen wie „Deck“), bei der sowohl am Anfang als auch am Ende Elemente eingereiht und entnommen werden können.
- Wichtige Anwendungen von Queues:
  - Scheduling von gleichrangigen Verarbeitungsjobs auf CPUs
  - Pufferspeicher für Datenströme oder Datenaustausch zwischen Prozessen
  - Anfrageschlange bei Druckern oder Webservern
  - Traversierung von Graphen (Breitensuche)

# Priority Queue

## Definition

Eine Queue heißt Priority Queue (*Prioritätswarteschlange*), wenn Elemente, unabhängig von der Reihenfolge in der sie hinzugefügt wurden, in der Reihenfolge ihrer “Priorität” (z.B. Größe) entnommen werden können.

# Priority Queue

## Definition

Eine Queue heißt Priority Queue (*Prioritätswarteschlange*), wenn Elemente, unabhängig von der Reihenfolge in der sie hinzugefügt wurden, in der Reihenfolge ihrer “Priorität” (z.B. Größe) entnommen werden können.

## Manipulation

- Einfügen (*Insert*)
  Ein Element in die Warteschlange einreihen.
- Maximum (Minimum)
  Das Element mit höchster Priorität betrachten.
- Maximum entnehmen (*Extract-Max*) (Minimum entnehmen (*Extract-Min*))
  Das Element mit höchster Priorität entnehmen.
- Priorität erhöhen (*Increase-Key*) (Priorität verringern (*Decrease-Key*))
  Die Priorität eines Elements in der Warteschlange erhöhen, so dass es weiter vorrückt.

# Priority Queue

## Definition

Eine Queue heißt Priority Queue (*Prioritätswarteschlange*), wenn Elemente, unabhängig von der Reihenfolge in der sie hinzugefügt wurden, in der Reihenfolge ihrer “Priorität” (z.B. Größe) entnommen werden können.

## Manipulation

- Einfügen (*Insert*)
  Ein Element in die Warteschlange einreihen.
- Maximum (Minimum)
  Das Element mit höchster Priorität betrachten.
- Maximum entnehmen (*Extract-Max*) (Minimum entnehmen (*Extract-Min*))
  Das Element mit höchster Priorität entnehmen.
- Priorität erhöhen (*Increase-Key*) (Priorität verringern (*Decrease-Key*))
  Die Priorität eines Elements in der Warteschlange erhöhen, so dass es weiter vorrückt.

## Implementierung

- Max-Heap (Min-Heap)

Bemerkungen:

- Eine Monotone Priority Queue ist eine spezialisierte Priority Queue für Fälle, in denen (im Falle eines Max-Heaps) kein später eingefügtes Element eine höhere Priorität hat als ein zuvor extrahiertes.
- Wichtige Anwendungen von Priority Queues:
  - Sortieren
  - Scheduling von priorisierten Verarbeitungsjobs
  - Management von Netzwerkverkehr (Quality of Service; z.B. VoIP und IPTV bevorzugt)
  - Simulation diskreter Ereignisse in zeitlicher Abfolge
  - Best-first Suche; kürzeste Wege in Graphen finden
  - Minimale Spannbäume in Graphen finden
  - Datenkompression mit der Huffmann-Kodierung