Imperative Programmierung (IPR)

Kapitel 5: Warteschlangen

Univ.-Prof. Dr.-Ing. habil. Gero Mühl

Lehrstuhl für Architektur von Anwendungssystemen (AVA)
Fakultät für Informatik und Elektrotechnik (IEF)
Universität Rostock

Universität Rostock

Traditio et Innovatio

AVA

Inhalte

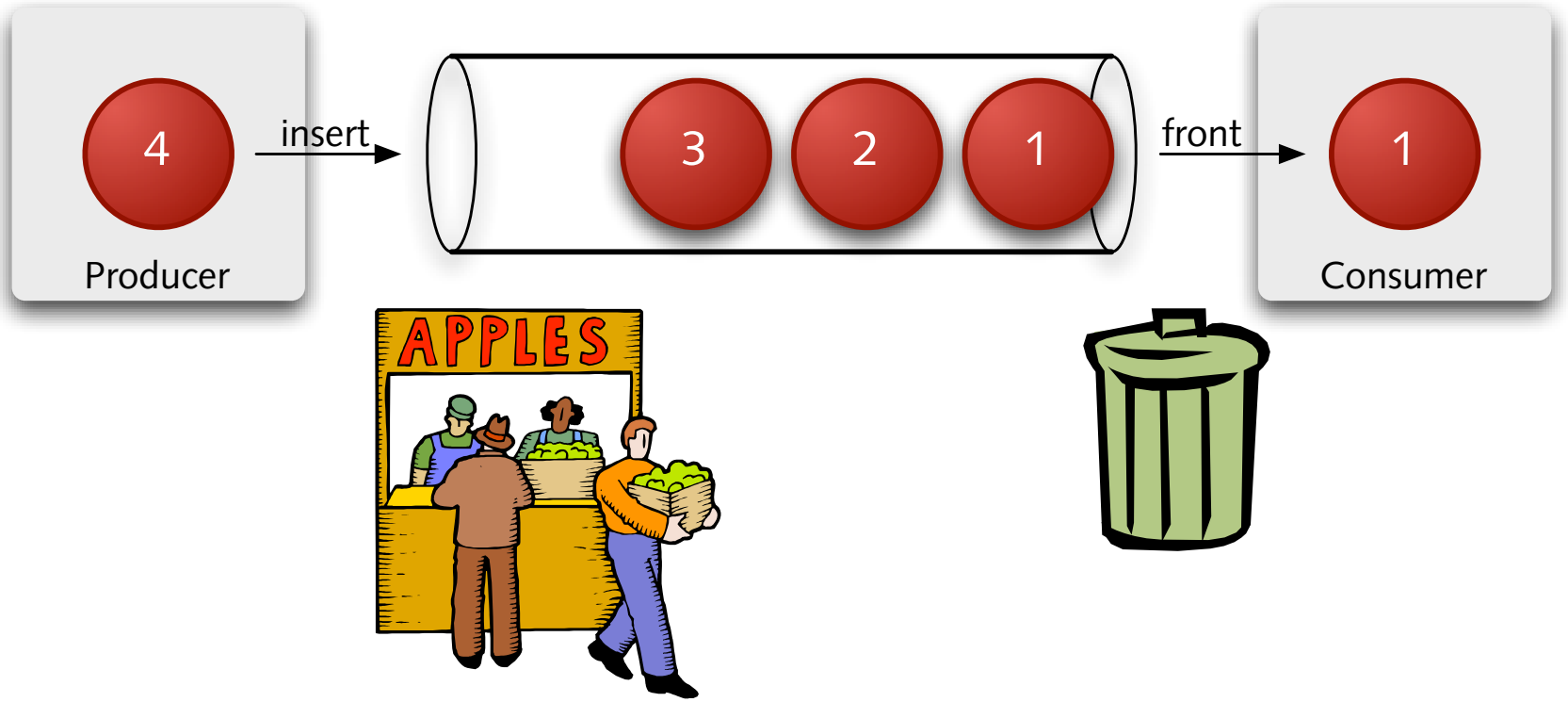
Kapitel 5.1

Definition

Die Warteschlange

Definition 1 (Warteschlange (Queue))

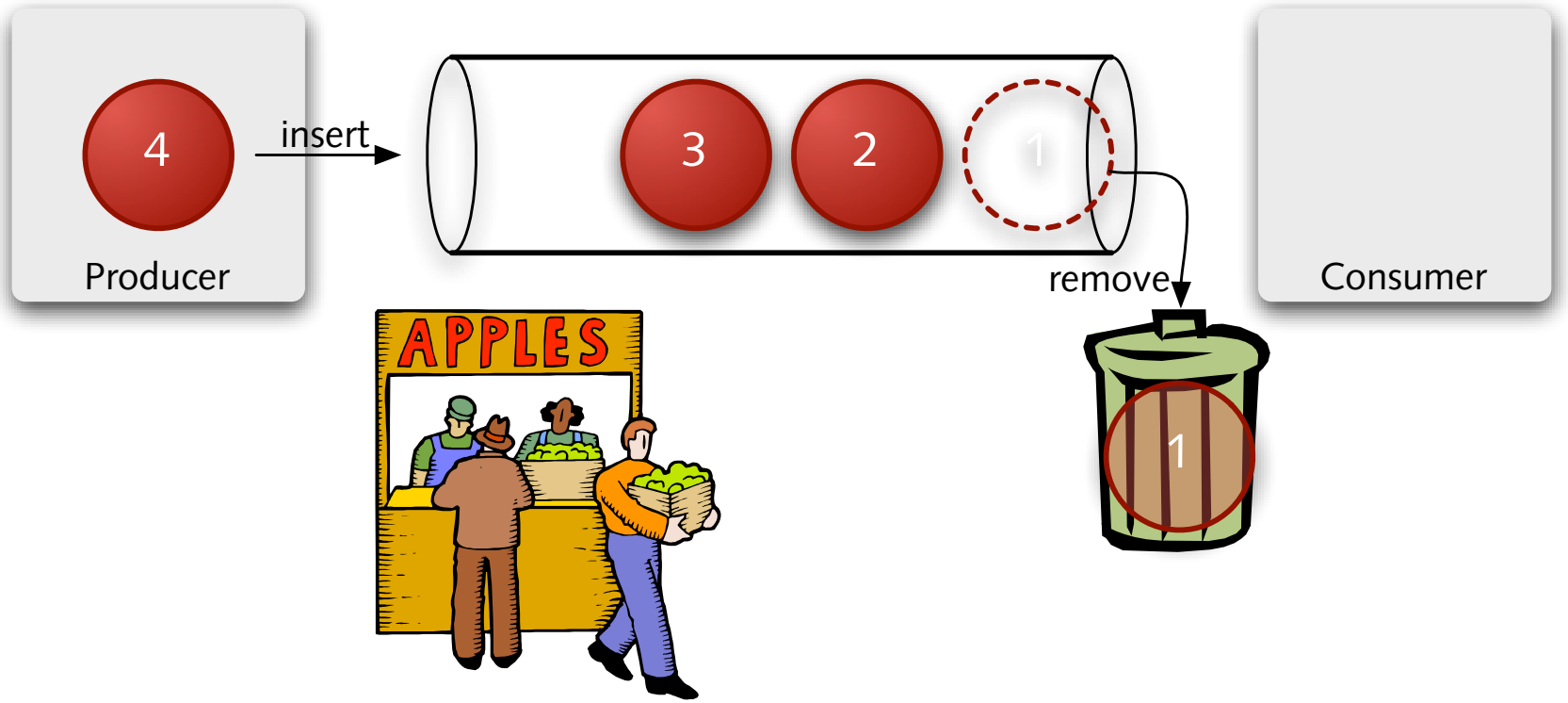
Eine **Warteschlange** *ist ein grundlegender Datentyp, der eine Menge von Elementen verwaltet, wobei immer nur auf ein Element zugegriffen werden kann. Der Zugriff erfolgt nach dem Prinzip* **First In First Out (FIFO)**,



Die Warteschlange

Definition 1 (Warteschlange (Queue))

Eine **Warteschlange** *ist ein grundlegender Datentyp, der eine Menge von Elementen verwaltet, wobei immer nur auf ein Element zugegriffen werden kann. Der Zugriff erfolgt nach dem Prinzip* **First In First Out (FIFO)**,



Kapitel 5.2

Spezifikation der begrenzten Schlange mit erweiterter Fehlerbehandlung

Grundlegende Operationen einer Schlange

Operation	Beschreibung der Operation
`init`	Neue Schlange
`insert(e,q)`	Element hinten an Schlange anfügen
`empty(q)`	Prüfen, ob Schlange leer
`front(q)`	Vorderstes Element der Schlange
`remove(q)`	Löschen des vordersten Elements der Schlange
`length(q)`	Länge der Schlange

Schrittweise Entwicklung der Spezifikation

- Eingebrachte Mengen und ausgezeichneter Fehlerwert:

$$[\mathbb{B}, \mathbb{Z}, \mathit{Element}]$$

$$\mid \quad \mathit{errorelement} \in \mathit{Element}$$

- Es wird die Menge von Schlangen mit zwei Fehlerelementen definiert:

$$[\mathit{Queue}]$$

$$\mid \quad \mathit{underflow}, \mathit{overflow} \in \mathit{Queue}$$

Schrittweise Entwicklung der Spezifikation

- Schlangen können mit folgenden Funktionen erzeugt werden:

$$\begin{aligned} &\mathit{init} : \mathit{Queue} \\ &\mathit{insert} : \mathit{Element} \times \mathit{Queue} \longrightarrow \mathit{Queue} \end{aligned}$$

- Schließlich gibt es eine Konstante für die maximale Anzahl von Elementen in einer Schlange:

$$\mathit{maxelements} \geq 1$$

- Zur einfacheren Spezifikation der Eigenschaften definieren wir noch:

$$\begin{aligned} \mathit{Element}_V &= \mathit{Element} \setminus \{\mathit{errorelement}\} \\ \mathit{Queue}_V &= \mathit{Queue} \setminus \{\mathit{underflow}, \mathit{overflow}\} \\ \mathit{Queue}_X &= \{q \in \mathit{Queue} \mid \mathit{length}(q) = \mathit{maxelements}\} \\ \mathit{Queue}_{X2} &= \mathit{Queue}_X \cup \\ &\qquad \{q \in \mathit{Queue} \mid \mathit{length}(q) = \mathit{maxelements} - 1\} \end{aligned}$$

Schlangenoperationen: `insert`

$$insert : Element \times Queue \longrightarrow Queue$$

$$\forall\, e : Element_V;\ q : Queue \bullet$$

$$insert(errorelement, q) = q$$
$$insert(e, underflow) = underflow$$
$$insert(e, overflow) = overflow$$

$$\forall\, e : Element_V;\ q : Queue_X \bullet$$

$$insert(e, q) = overflow$$

- Diese Gesetze spezifizieren die Fehlerfälle zum Konstruktor *insert*.

Schlangenoperationen: `empty`

$$empty : Queue \longrightarrow \mathbb{B}$$

$$\forall\, e : Element_V, q : Queue_V \setminus Queue_X \bullet$$

$$empty(underflow) = True$$
$$empty(overflow) = True$$

$$empty(init) = True$$
$$empty(insert(e, q)) = False$$

Schlangenoperationen: `empty`

- Betrachten wir nochmal das Gesetz:

$$empty(insert(e, q)) = False$$

  mit $e \in element_V$ und $q \in Queue_V \setminus Queue_X$.

- Wären für e und q die Fehlerwerte nicht ausgeschlossen, so wären folgende Ableitungen mit diesem Gesetz möglich:

$$empty(insert(errorelement, init)) = False$$
$$empty(insert(e, overflow)) = False$$

- Diese ständen dann im Widerspruch zu folgenden Ableitungen:

$$empty(insert(errorelement, init)) = empty(init) = True$$
$$empty(insert(e, overflow)) = empty(overflow) = True$$

Schlangenoperationen: `empty`

- Betrachten wir nochmal das Gesetz:

$$empty(insert(e, q)) = False$$

  mit $e \in element_V$ und $q \in Queue_V \setminus Queue_X$.
- Könnte q auch eine volle Queue sein, dann wäre für ein solches $q_x \in Queue_X$ die folgende Ableitung mit diesem Gesetz möglich:

$$empty(insert(e, q_x)) = False$$

- Diese ständen dann im Widerspruch zur folgenden Ableitung:

$$empty(insert(e, q_x)) = empty(overflow) = True$$

- In ähnlicher Weise müssen entsprechende Sonderfälle für alle Operationen beachtet werden.

Schlangenoperationen: `front`

$$front : Queue \longrightarrow Element$$

$$\forall\, e, f : Element_V, q : Queue_V \setminus Queue_{X2} \bullet$$

$$front(init) = errorelement$$
$$front(underflow) = errorelement$$
$$front(overflow) = errorelement$$

$$front(insert(e, init)) = e$$

$$front(insert(f, insert(e, q))) = front(insert(e, q))$$

- `front` liefert das vorderste Element der Schlange.
- Dieses wurde zuerst eingefügt und befindet sich daher ganz innen im Term, also immer im Teilterm $insert(e, init)$.

Schlangenoperationen: `remove`

$$
\begin{array}{|l}
\hline
\mathit{remove} : \mathit{Queue} \longrightarrow \mathit{Queue} \\
\hline
\forall\, e, f : \mathit{Element}_V, q : \mathit{Queue}_V \setminus \mathit{Queue}_{X2} \bullet \\
\quad \mathit{remove}(\mathit{init}) = \mathit{underflow} \\
\quad \mathit{remove}(\mathit{underflow}) = \mathit{underflow} \\
\quad \mathit{remove}(\mathit{overflow}) = \mathit{overflow} \\
\quad \mathit{remove}(\mathit{insert}(e, \mathit{init})) = \mathit{init} \\
\quad \mathit{remove}(\mathit{insert}(f, \mathit{insert}(e, q))) \\
\qquad = \mathit{insert}(f, \mathit{remove}(\mathit{insert}(e, q)))
\end{array}
$$

- `remove` entfernt das vorderste Element der Schlange.

Schlangenoperationen: `length`

$$length : Queue \longrightarrow \mathbb{Z}$$

$$\forall\, e : Element_V, q : Queue_V \setminus Queue_X \bullet$$

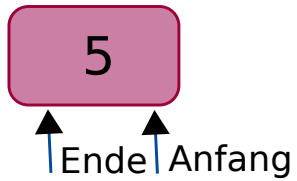
$$length(underflow) = -1$$
$$length(overflow) = maxelements + 1$$

$$length(init) = 0$$
$$length(insert(e, q)) = 1 + length(q)$$

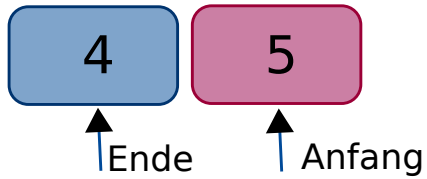
Grafische Repräsentation: `insert` und `front`

- Beschreibung der Schlange als Term: $insert(3, insert(4, insert(5, init)))$
- Erzeugen einer leeren Schlange mittels `init`

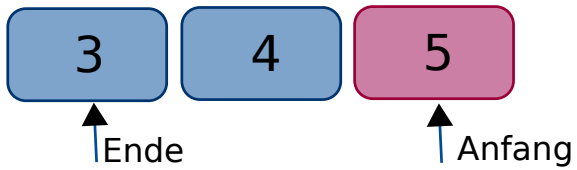
- Schlange nach $insert(5, init)$



- Schlange nach $insert(4, insert(5, init))$

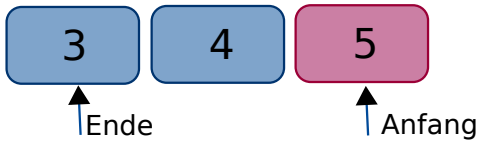


- Schlange nach $insert(3, insert(4, insert(5, init)))$

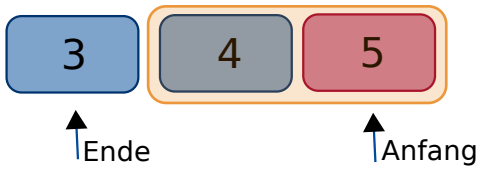


Grafische Repräsentation: `remove`

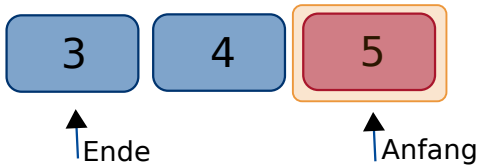
- Schlange als Term: $remove(insert(3, insert(4, insert(5, init))))$
- Schlange vor remove



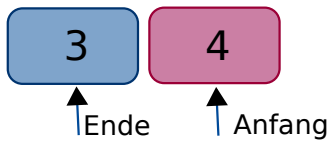
- In $insert(3, remove(insert(4, insert(5, init))))$ betrachteter Schlangenteil



- In $insert(3, insert(4, remove(insert(5, init))))$ betrachteter Schlangenteil



- Schlange nach *remove*



Ausführbare Spezifikation der erweiterten Schlange

```
module Queue where
import Prelude hiding (length)

type Element = Int
errorelement = -1
maxelements = 5

data Queue = Empty | Underflow | Overflow |
             App(Element,Queue) deriving (Show,Eq)

insert :: (Element, Queue) -> Queue

insert(e,q)  =
   if e == errorelement then q
   else if q == Underflow then q
   else if length(q) > maxelements - 1 then Overflow
   else App(e,q)
```

Ausführbare Spezifikation der erweiterten Schlange

```
empty   :: Queue -> Bool
front   :: Queue -> Element

empty(Empty)      = True
empty(Underflow)  = True
empty(Overflow)   = True
empty(App(e,q))   = False

front(Empty)              = errorelement
front(Underflow)          = errorelement
front(Overflow)           = errorelement
front(App(e,Empty))       = e
front(App(f,App(e,q)))    = front(App(e,q))
```

Ausführbare Spezifikation der erweiterten Schlange

```
remove :: Queue -> Queue
length :: Queue -> Element

remove(Empty)             = Underflow
remove(Underflow)         = Underflow
remove(Overflow)          = Overflow
remove(App(e,Empty))      = Empty
remove(App(f,App(e,q))) = App(f,remove(App(e,q)))

length(Empty)      = 0
length(Overflow)   = maxelements + 1
length(Underflow) = -1
length(App(e,q))   = 1 + length(q)

tuplequeue(Empty)          = []
tuplequeue(App(e,Empty)) = [e]
tuplequeue(App(e,q))       = [e] ++ tuplequeue(q)
```

Kapitel 5.3

Implementierung als Ringspeicher

Implementierung als Ringspeicher

- Die Elemente der Schlange werden in einem Array abgelegt.
- Um beim Löschen des ältesten Elements ein Verschieben der neueren Arrayelemente zu umgehen, wird das Array als **Ringpuffer** organisiert.
- Ein neues Element wird hinten in das Array nach dem bisher letzten Element eingefügt.
- Was passiert, wenn das letzte Arrayelement belegt ist und dann ein Element eingefügt werden soll?
- Das neue Element wird stattdessen in das erste Arrayelement eingetragen, sofern dieses bereits wieder frei ist $\rightarrow$ **Wrap-Around**

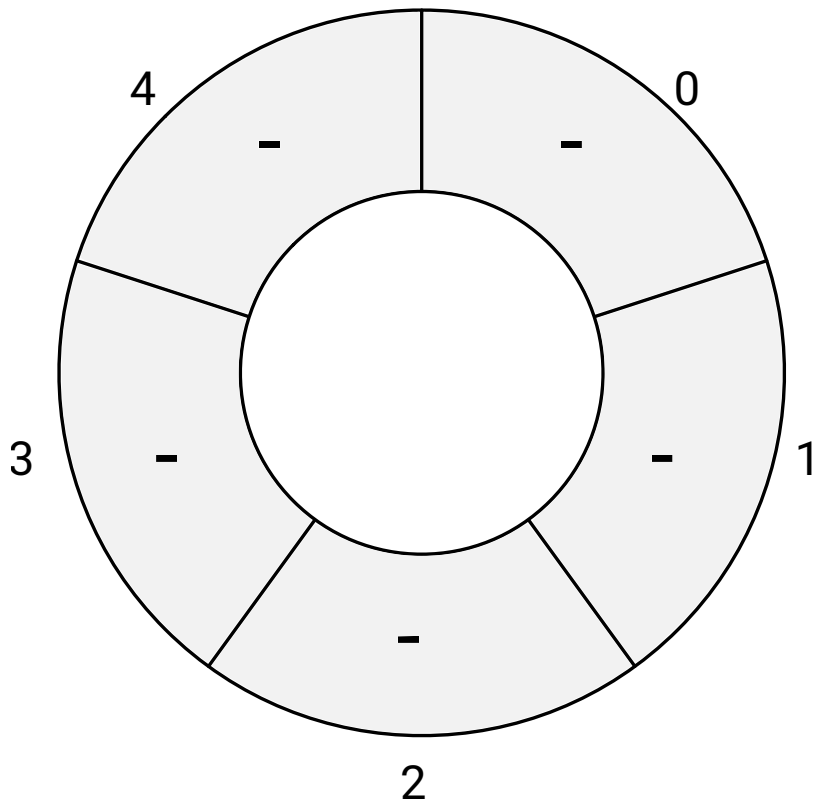
Implementierung als Ringspeicher

- Das Attribut `right` zeigt immer auf das Arrayelement, in das als nächstes ein Element eingefügt werden kann.
- Das Attribut `left` zeigt immer auf das älteste verbleibende Arrayelement, falls der Ring nicht leer ist.
- Das Attribut `elements` enthält die Element der Queue in einem Array.

```
struct _queue {
  int size;
  int left;
  int right;
  element elements[QUEUE_MAX_ELEMENTS];
};
typedef struct _queue queue;
```

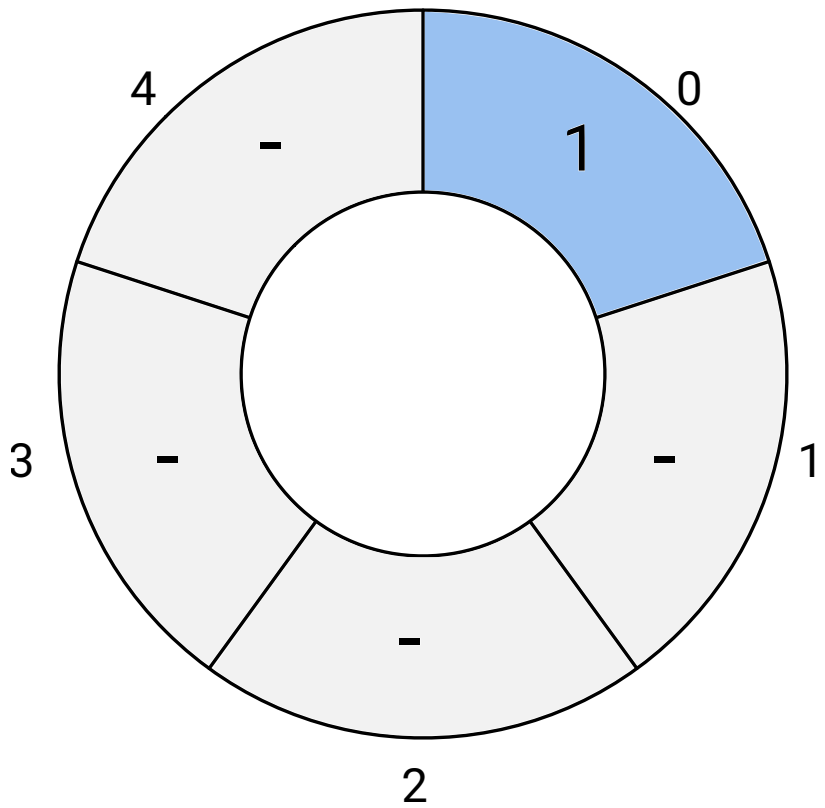
Implementierung als Ringspeicher

- *init*



Implementierung als Ringspeicher

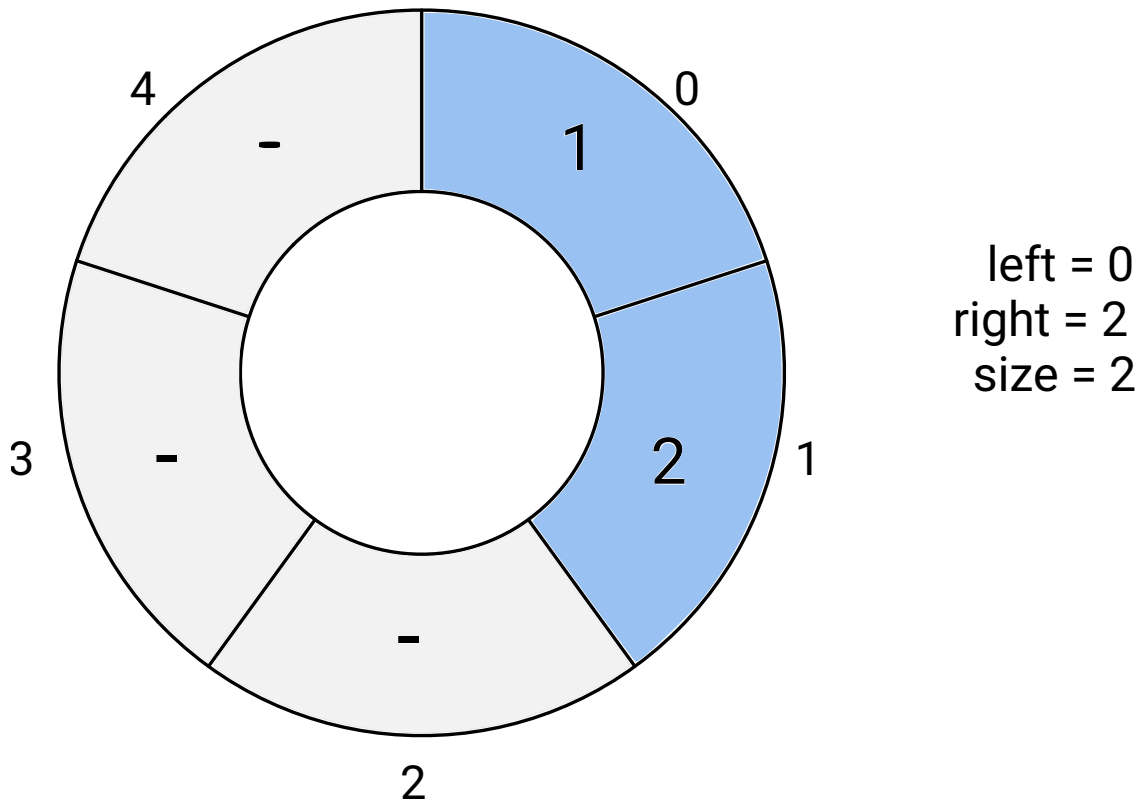
- $insert(1, init)$



left = 0
right = 1
size = 1

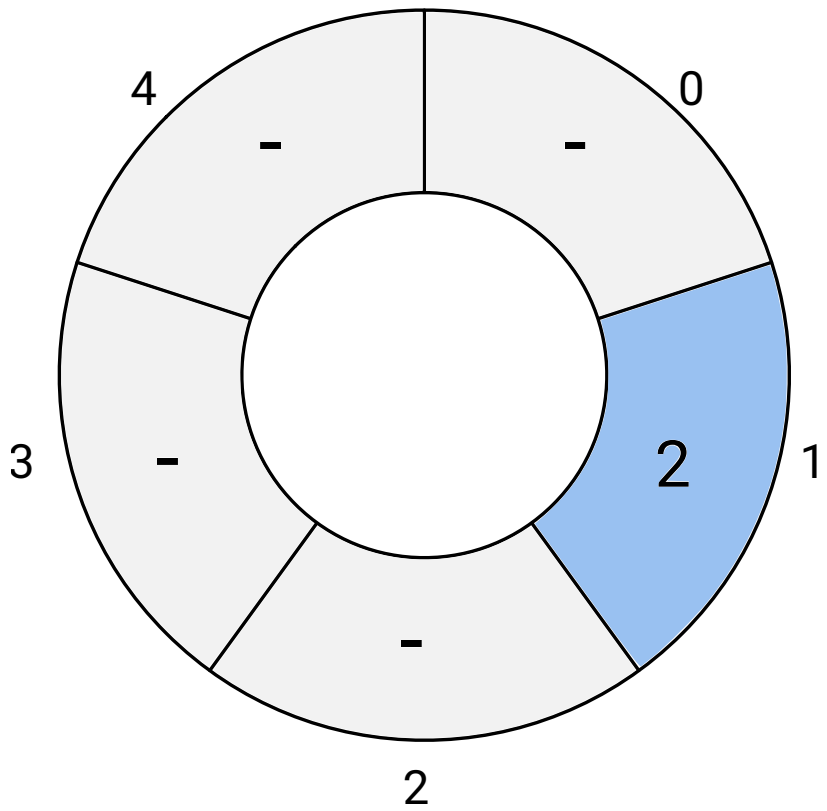
Implementierung als Ringspeicher

- $insert(2, insert(1, init))$



Implementierung als Ringspeicher

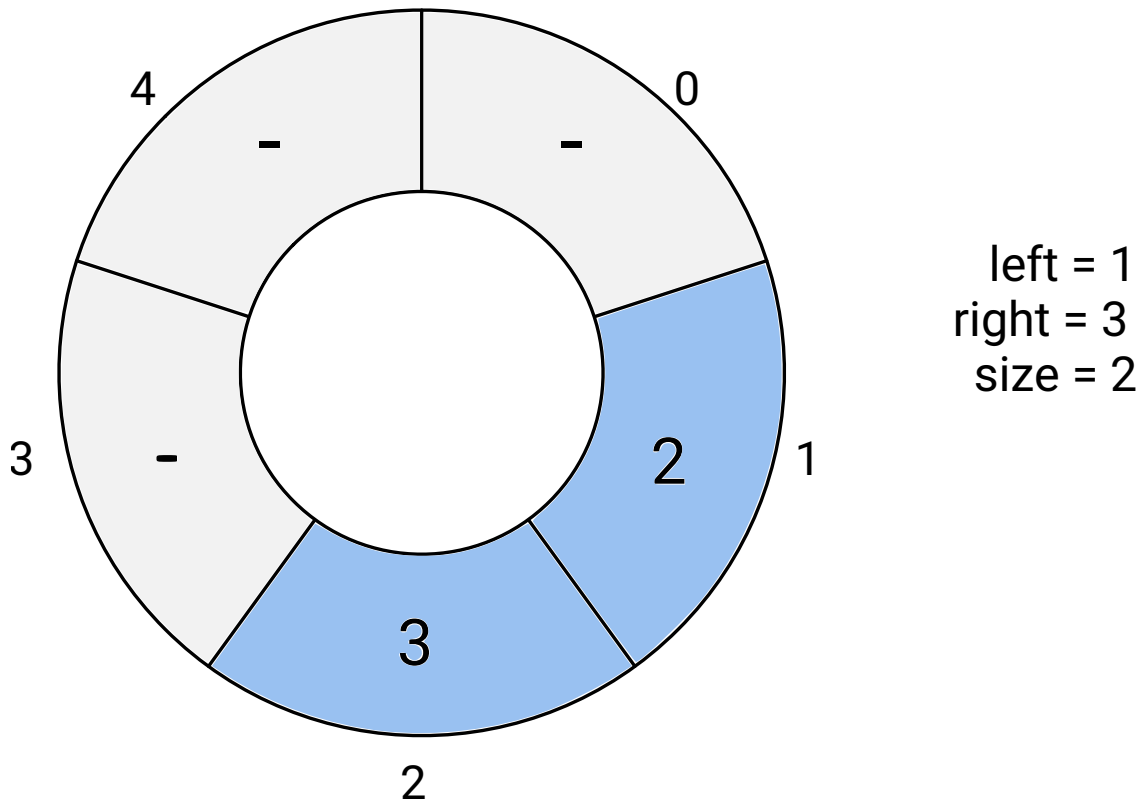
- $remove(insert(2, insert(1, init)))$



left = 1
right = 2
size = 1

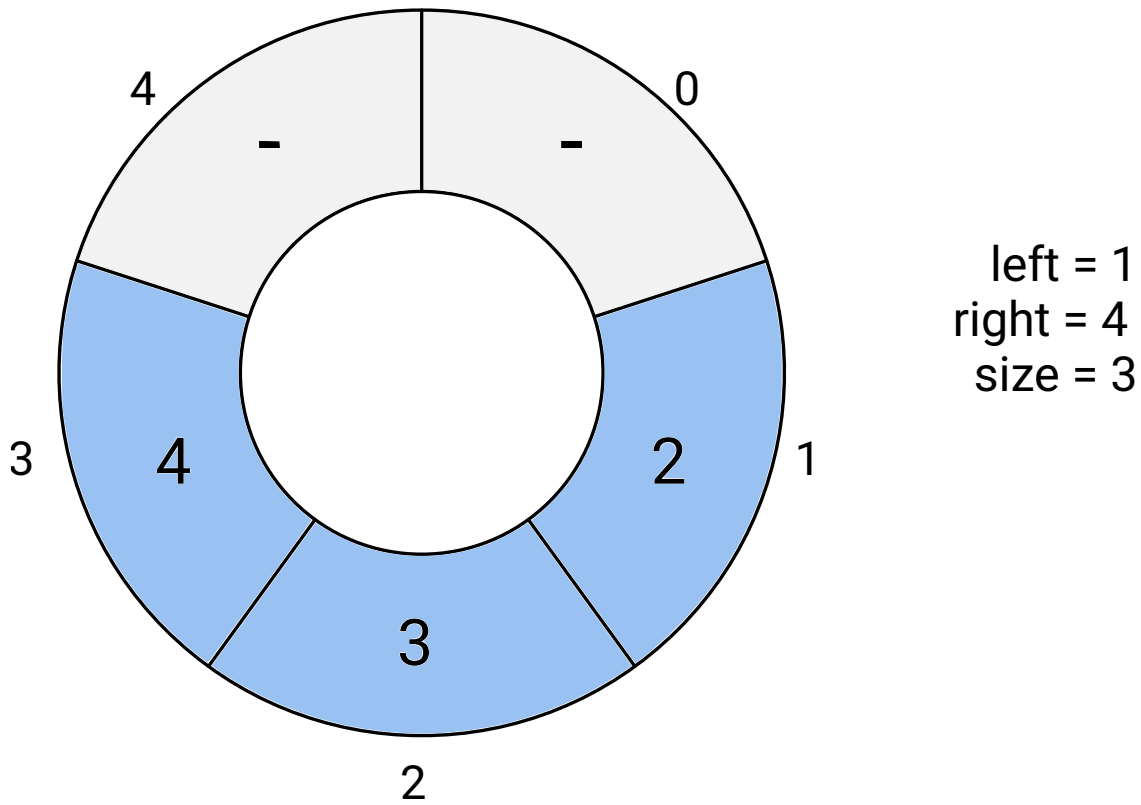
Implementierung als Ringspeicher

- $insert(3, remove(insert(2, insert(1, init))))$



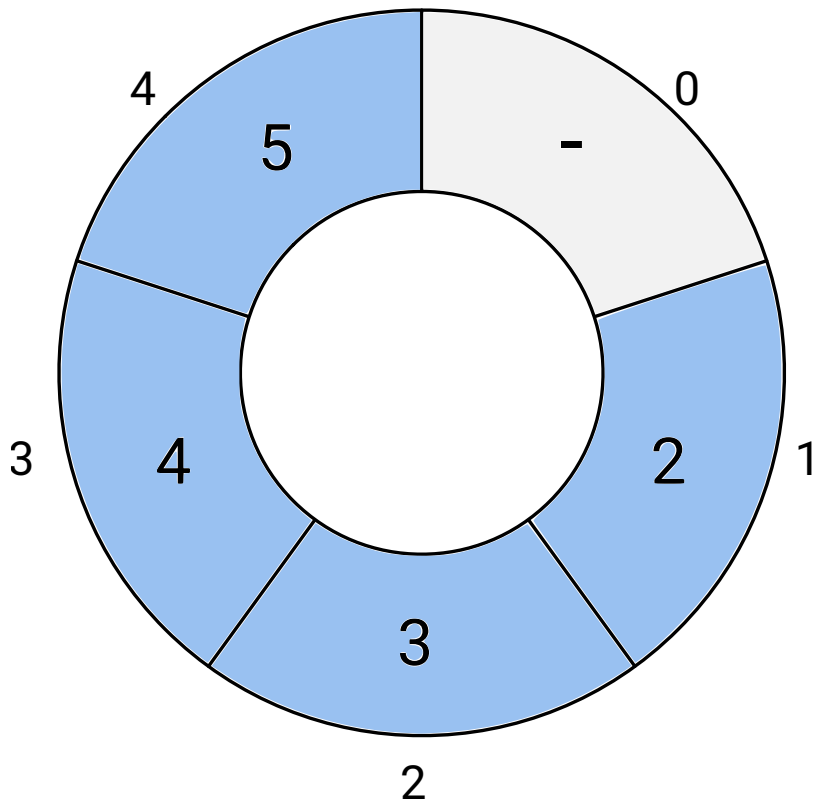
Implementierung als Ringspeicher

- $insert(4, insert(3, remove(insert(2, insert(1, init)))))$



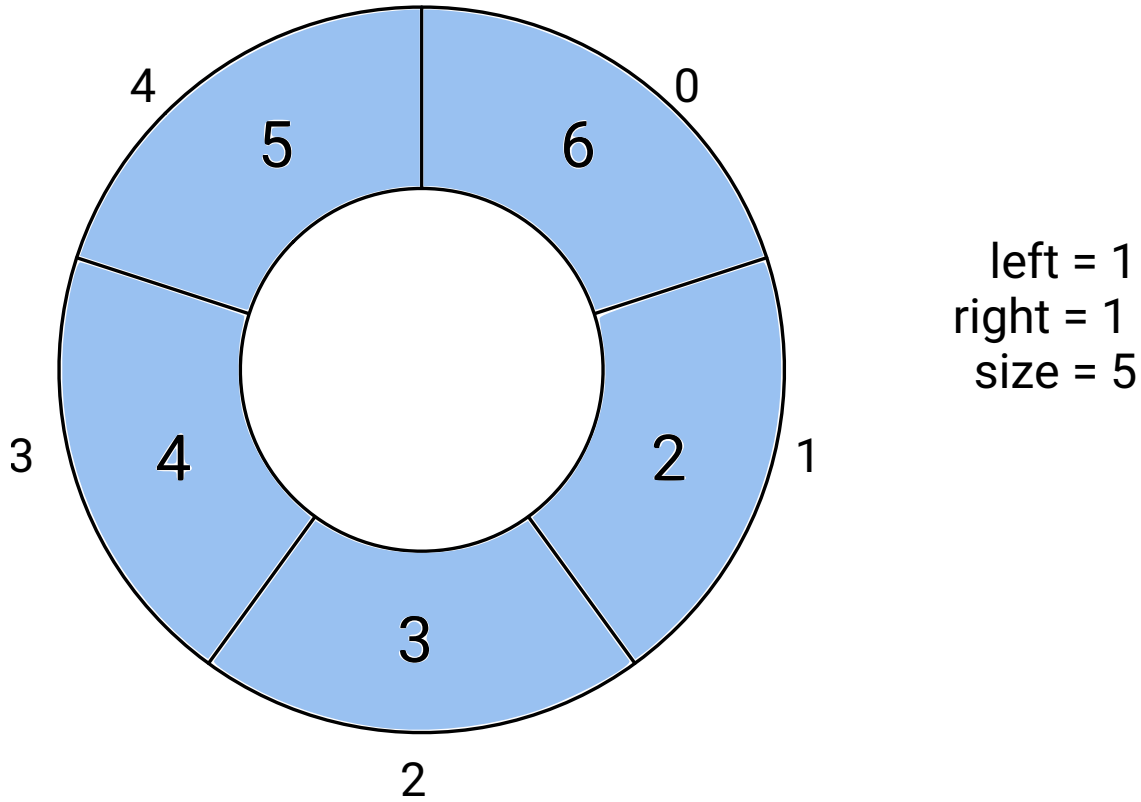
Implementierung als Ringspeicher

- $insert(5, insert(4, insert(3, remove(insert(2, insert(1, init))))))$



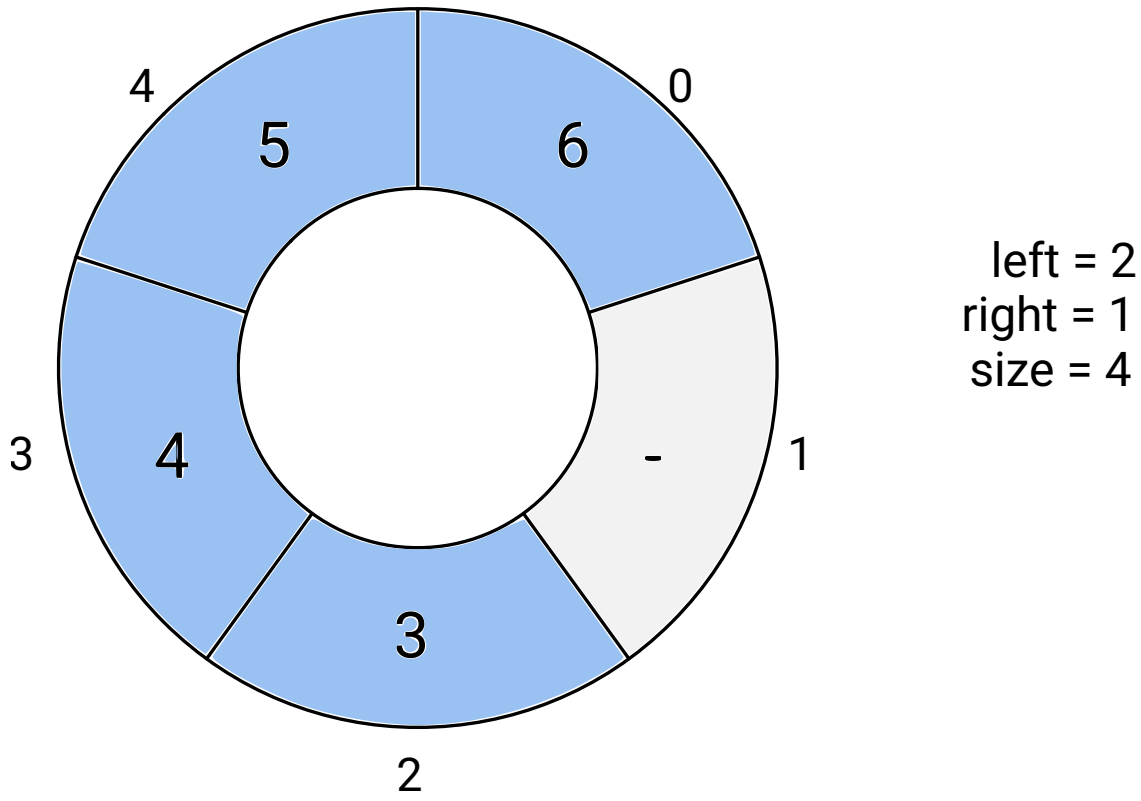
Implementierung als Ringspeicher

- $insert(6, insert(5, insert(4, insert(3, remove(insert(2, insert(1, init)))))))$



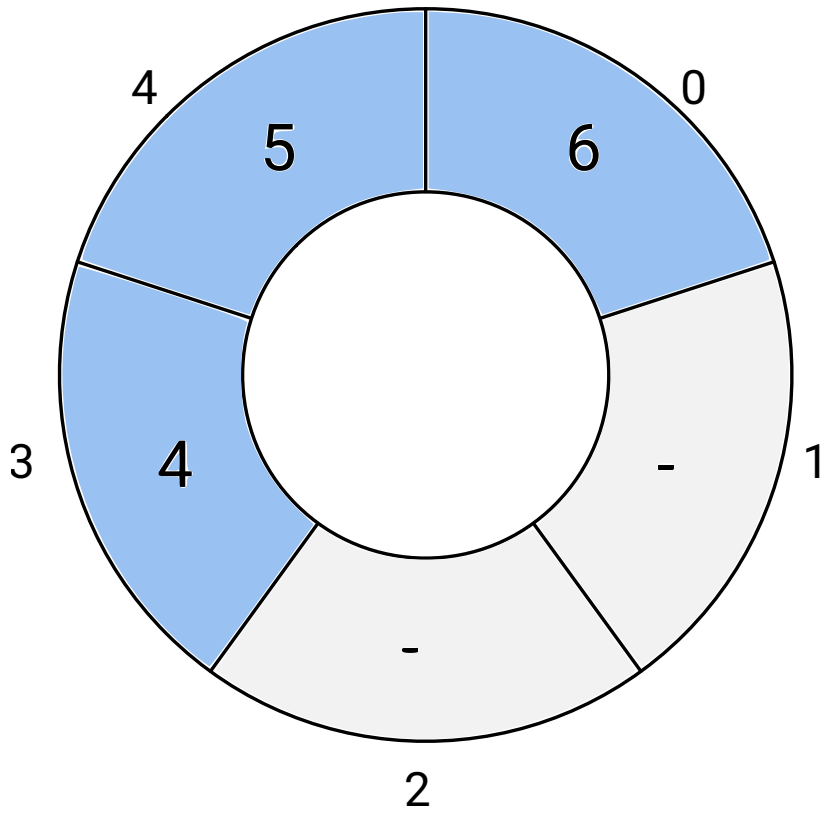
Implementierung als Ringspeicher

- $remove(insert(6, insert(5, insert(4, insert(3, remove(insert(2, insert(1, init)))))))$



Implementierung als Ringspeicher

- $remove(remove(insert(6, insert(5, insert(4, insert(3, remove(insert(2, insert(1, init)))))))))$



left = 3
right = 1
size = 3

Implementierung

```
#define QUEUE_MAX_ELEMENTS 100
#define QUEUE_ERROR_ELEMENT INT_MIN

typedef int element;

struct _queue {
  int size;
  int left;
  int right;
  element elements[QUEUE_MAX_ELEMENTS];
};

typedef struct _queue queue;
```

Implementierung init, destroy und length

```
queue* queue_init() {
  queue* q = queue_malloc(sizeof(queue));
  q->left = 0;
  q->right = 0;
  q->size = 0;
  return q;
}

void queue_destroy(queue* q) {
  queue_free(q);
}
```

Implementierung overflow, underflow und empty

```
int queue_overflow(queue* q) {
  return queue_length(q) == QUEUE_MAX_ELEMENTS + 1;
}

int queue_underflow(queue* q) {
  return queue_length(q) == -1;
}

int queue_empty(queue* q) {
  return queue_underflow(q) ||
              queue_length(q) == 0 || queue_overflow(q);
}
```

Implementierung insert

```
queue* queue_insert(element e, queue* q) {
  if (e == QUEUE_ERROR_ELEMENT) {
    fprintf(stderr,
        "insert: trying to insert errorvalue!\n");
  } else if (queue_underflow(q) || queue_overflow(q)) {
    fprintf(stderr,
        "insert: underflowed or overflowed queue!\n");
  } else if (queue_length(q) == QUEUE_MAX_ELEMENTS) {
    fprintf(stderr, "insert: new queue overflow!\n");
    q->size = QUEUE_MAX_ELEMENTS + 1;
  } else {
    q->elements[q->right] = e;
    q->right = (q->right + 1) % QUEUE_MAX_ELEMENTS;
    q->size++;
  }
  return q;
}
```

Implementierung front

```
element queue_front(queue* q) {
  if (queue_underflow(q)) {
    fprintf(stderr, "front: underflowed queue!\n");
    return QUEUE_ERROR_ELEMENT;
  } else if (queue_overflow(q)) {
    fprintf(stderr, "front: overflowed queue!\n");
    return QUEUE_ERROR_ELEMENT;
  } else if (queue_empty(q)) {
    fprintf(stderr, "front: empty queue!\n");
    return QUEUE_ERROR_ELEMENT;
  } else
    return q->elements[q->left];
}
```

Implementierung remove

```
queue* queue_remove(queue* q) {
  if (queue_underflow(q) || queue_overflow(q)) {
    fprintf(stderr,
        "remove: underflowed or overflowed queue!\n");
  } else if (queue_empty(q)) {
    fprintf(stderr, "remove: new queue underflow!\n");
    q->size = -1;
  } else {
    q->left = (q->left + 1) % QUEUE_MAX_ELEMENTS;
    q->size--;
  }
  return q;
}
```

Implementierung copy

```
queue* queue_copy(queue* q) {
  queue* c = queue_init();
  c->left = q->left;
  c->right = q->right;
  c->size = q->size;

  int index = q->right - 1;
  int x = q->size;
  while (x > 0) {
    if (index == -1)
      index = QUEUE_MAX_ELEMENTS - 1;
    c->elements[index] = q->elements[index];
    index--;
    x--;
  }
  return c;
}
```

Kapitel 5.4

Prioritätswarteschlangen

Prioritätswarteschlangen (engl. Priority Queues)

- In vielen Fällen ist es sinnvoll, bestimmte Einträge in einer Warteschlange bevorzugt zu behandeln.
- Dies können z. B. wichtige Kunden oder dringende Anfragen sein.
- Diese Einträge sollten bevorzugt geliefert werden, ungeachtet bereits älterer Einträge mit geringerer Dringlichkeit.
- Einträge mit *derselben* Dringlichkeit sollten hingegen nach wie vor nach dem Prinzip *first come, first served* behandelt werden.
- Zur Realisierung dieser Funktionalität werden **Prioritätswarteschlangen** verwendet.
- Diese ordnen jedem Element eine **Priorität** zu, die die Dringlichkeit bzw. Wichtigkeit des jeweiligen Elements beschreibt.

Grundlegende Funktionen einer Prioritätsschlange

Funktion	Beschreibung der Funktion
`init`	Neue Prioritätsschlange
`empty(q)`	Prüfen, ob Schlange leer
`pinsert(e,p,q)`	Element e mit Priorität p hinten an Schlange q anfügen
`pfront(q)`	Liefert das vorderste der Elemente mit höchster Priorität
`premove(q)`	Entfernt das vorderste der Elemente mit höchster Priorität
`length(q)`	Länge der Schlange
`maxprio(q)`	Maximale Priorität in der Schlange

Spezifikation von Prioritätswarteschlangen

- Für die Spezifikation bieten sich zwei Varianten an:

1. Auswählen beim Auslesen (*RQueue*):
   - Beim Einfügen wird ein neues Element unabhängig von seiner Priorität immer hinten an die Schlange angefügt.
   - Beim Auslesen wird dann das vorderste Element unter denen mit höchster Priorität ermittelt.

2. Sortieren beim Einfügen (*PQueue*):
   - Beim Einfügen eines neuen Elements wird dieses an der richtigen Stelle entsprechend seiner Priorität in die Schlange eingefügt.
   - Sinnvollerweise wird dabei eine Sortierung verwendet, bei der die Wurzel des Terms das älteste Element mit der höchsten Priorität ist.
   - Das heißt, ein neues Element wird direkt vor dem ersten Element mit *niedrigerer* Priorität in den Term eingefügt.

Spezifikation von Prioritätswarteschlangen

- Wir betrachten im Folgenden die *PQueue*.
- Diese hat den Vorteil, dass unterschiedliche Terme auch ein unterschiedliches Verhalten zeigen.
- Bei der *RQueue* können hingegen unterschiedliche Terme das gleiche Verhalten zeigen.

Schrittweise Entwicklung der Spezifikation

- Die Beschreibung des Datentyps `PQueue` greift zurück auf Boolesche Werte, ganze Zahlen und auf die Elemente, die in einer Prioritätswarteschlange sein können, sowie die Menge der Prioritäten:

  $$[\mathbb{B}, \mathbb{Z}, \mathit{Element}, \mathit{Priority}]$$

- Innerhalb von *Element* soll es ein Fehlerelement geben (z. B. für den Fall, dass wir mittels `pfront` auf die leere Schlange zugreifen).

  $$\mid \quad \mathit{errorelement} \in \mathit{Element}$$

- Auf Prioritäten soll eine Ordnung definiert sein

  $$\mid \quad \_ < \_ : \mathit{Priority} \leftrightarrow \mathit{Priority}$$

- Für $p_1, p_2 \in \mathit{Priority}$ mit $p_1 < p_2$ bezeichnen wir p_1 als die **niedrigere Priorität** und p_2 als die **höhere Priorität**.
- Innerhalb der Prioritäten soll es auch ein Fehlerelement geben

  $$\mid \quad \mathit{errorpriority} \in \mathit{Priority}$$

Schrittweise Entwicklung der Spezifikation

- Wir definieren die Menge von Prioritätswarteschlangen:

  $$[PQueue]$$

- Innerhalb der Menge $PQueue$ soll es zwei Fehlerelemente geben:

  $$underflow, overflow \in PQueue$$

- Prioritätswarteschlangen können folgendermaßen erzeugt werden:

  $$init : PQueue$$
  $$pinsert : Element \times Priority \times PQueue \longrightarrow PQueue$$

- Zur einfacheren Darstellung der Eigenschaften definieren wir noch:

  $$Element_V = Element \setminus \{errorelement\}$$
  $$Priority_V = Priority \setminus \{errorpriority\}$$
  $$PQueue_V = PQueue \setminus \{underflow, overflow\}$$
  $$PQueue_X = \{q \in PQueue \mid length(q) = maxelements\}$$
  $$PQueue_{X2} = PQueue_X \cup \{q \in PQueue \mid length(q) = maxelements - 1\}$$

Schrittweise Entwicklung der Spezifikation

- Zur Sortierung nach Priorität werden die Terme auf Basis von *pinsert* auf Terme auf Basis der privaten Funktion *insert* abgebildet:

$$| \quad insert : Element \times Priority \times PQueue \longrightarrow PQueue$$

- Beispielsweise werden die Terme

$$pinsert(3, 1, pinsert(4, 2, init))$$
$$pinsert(4, 2, pinsert(3, 1, init))$$

beide auf den folgenden normalisierten Term abgebildet:

$$insert(4, 2, insert(3, 1, init))$$

- Die weiteren Funktionen werden dann auf Basis der normalisierten Terme, also mittels *insert* definiert.

Schlangenoperationen: `pinsert`

$$pinsert : Element \times Priority \times PQueue \longrightarrow PQueue$$

$\forall\, e : Element_V;\ p : Priority_V;\ q : PQueue \bullet$

$$pinsert(errorelement, p, q) = q$$
$$pinsert(e, errorpriority, q) = q$$
$$pinsert(e, p, underflow) = underflow$$
$$pinsert(e, p, overflow) = overflow$$

$$pinsert(e, p, init) = insert(e, p, init)$$

$\forall\, e : Element_V;\ p : Priority_V;\ q : PQueue_X \bullet$

$$pinsert(e, p, q) = overflow$$

Schlangenoperationen: `pinsert`

$$
\begin{aligned}
&\forall\, e_1, e_2 : Element_V;\ p_1, p_2 : Priority_V;\ q : PQueue_V \setminus PQueue_{X2} \bullet \\
&\quad p_2 > p_1 \Rightarrow \\
&\qquad pinsert(e_2, p_2, insert(e_1, p_1, q)) \\
&\qquad\quad = insert(e_2, p_2, insert(e_1, p_1, q)) \\
&\quad p_2 \leq p_1 \Rightarrow \\
&\qquad pinsert(e_2, p_2, insert(e_1, p_1, q)) \\
&\qquad\quad = insert(e_1, p_1, pinsert(e_2, p_2, q))
\end{aligned}
$$

- Ist p_2 die höhere Priorität, bleibt das Element e_2 an der ersten Stelle.
- Sonst tauscht e_2 den Platz mit dem Element e_1 an der zweiten Stelle und der Rest des Terms wird betrachtet.

Schlangenoperationen: `pinsert`

Beispiel 1 (pinsert)

$$\begin{aligned}
&\mathit{pinsert}(11, 1, \mathit{pinsert}(9, 3, \underbrace{\mathit{pinsert}(15, 2, \mathit{init})})) \\
&= \mathit{pinsert}(11, 1, \underbrace{\mathit{pinsert}(9, 3, \mathit{insert}(15, 2, \mathit{init}))}) \\
&= \underbrace{\mathit{pinsert}(11, 1, \mathit{insert}(9, 3, \mathit{insert}(15, 2, \mathit{init})))} \\
&= \mathit{insert}(9, 3, \underbrace{\mathit{pinsert}(11, 1, \mathit{insert}(15, 2, \mathit{init}))}) \\
&= \mathit{insert}(9, 3, \mathit{insert}(15, 2, \underbrace{\mathit{pinsert}(11, 1, \mathit{init})})) \\
&= \mathit{insert}(9, 3, \mathit{insert}(15, 2, \mathit{insert}(11, 1, \mathit{init})))
\end{aligned}$$

- Es ist jeweils der Teilterm markiert, auf den im nächsten Schritt eines der drei Gesetze angewendet wird.

Schlangenoperationen: `empty`

$$empty : PQueue \longrightarrow \mathbb{B}$$

$$\forall\, e : Element_V;\ p : Priority_V;\ q : PQueue_V \setminus PQueue_X \bullet$$

$$empty(underflow) = True$$
$$empty(overflow) = True$$

$$empty(init) = True$$
$$empty(insert(e, p, q)) = False$$

Schlangenoperationen: `pfront`

$$pfront : PQueue \longrightarrow Element$$

$$\forall\, e : Element_V;\ p : Priority_V;\ q : PQueue_V \setminus PQueue_X \bullet$$

$$pfront(init) = errorelement$$
$$pfront(underflow) = errorelement$$
$$pfront(overflow) = errorelement$$

$$pfront(insert(e, p, q)) = e$$

Schlangenoperationen: `premove`

$$premove : PQueue \longrightarrow PQueue$$

$$\forall\, e : Element_V;\ p : Priority_V;\ q : PQueue_V \setminus PQueue_X \bullet$$

$$premove(init) = underflow$$
$$premove(underflow) = underflow$$
$$premove(overflow) = overflow$$

$$premove(insert(e, p, q)) = q$$

Schlangenoperationen: `length`

$$length : PQueue \longrightarrow \mathbb{Z}$$

$$\forall\, e : Element_V;\ p : Priority_V;\ q : PQueue_V \setminus PQueue_X \bullet$$

$$length(underflow) = -1$$
$$length(overflow) = maxelements + 1$$

$$length(init) = 0$$
$$length(insert(e, p, q)) = 1 + length(q)$$

Schlangenoperationen: `maxprio`

$$maxprio : PQueue \longrightarrow Priority$$

$$\forall\, e : Element_V;\ p : Priority_V;\ q : PQueue_V \setminus PQueue_X \bullet$$

$$maxprio(init) = errorpriority$$
$$maxprio(underflow) = errorpriority$$
$$maxprio(overflow) = errorpriority$$

$$maxprio(insert(e, p, q)) = p$$

Ausführbare Spezifikation

```
module PQueue where
import Prelude hiding (length)

type Element = Int
errorelement = -1

type Priority = Int
errorpriority = -1

data PQueue = Empty | Underflow | Overflow |
              App(Element,Priority,PQueue)
              deriving (Show,Eq)

maxelements = 5
```

Ausführbare Spezifikation

```
pinsert :: (Element, Priority, PQueue) -> PQueue

pinsert(e,p,Empty) =
    if e == errorelement then Empty
    else if p == errorpriority then Empty
         else App(e,p,Empty)

pinsert(e,p,Overflow)  = Overflow
pinsert(e,p,Underflow) = Underflow

pinsert(e2,p2,App(e1,p1,q)) =
    if e2 == errorelement then App(e1,p1,q)
    else if p2 == errorpriority then App(e1,p1,q)
    else if length(q) > maxelements - 1 then Overflow
    else if p1 < p2 then App(e2,p2,App(e1,p1,q))
    else App(e1,p1,pinsert(e2,p2,q))
```

Ausführbare Spezifikation

```
pfront   :: PQueue -> Element
premove  :: PQueue -> PQueue

pfront(Empty)       = errorelement
pfront(Underflow)   = errorelement
pfront(Overflow)    = errorelement
pfront(App(e,p,q)) = e

premove(Empty)       = Underflow
premove(Underflow)   = Underflow
premove(Overflow)    = Overflow
premove(App(e,p,q)) = q
```

Ausführbare Spezifikation

```
empty    :: PQueue -> Bool
maxprio  :: PQueue -> Priority

empty(Empty)       = True
empty(Underflow)   = True
empty(Overflow)    = True
empty(App(e,p,q)) = False

maxprio(Empty)       = errorpriority
maxprio(Underflow)   = errorpriority
maxprio(Overflow)    = errorpriority
maxprio(App(e,p,q)) = p
```

Ausführbare Spezifikation

```
length  :: PQueue -> Element

length(Empty)       = 0
length(Overflow)    = maxelements + 1
length(Underflow)   = -1
length(App(e,p,q)) = 1 + length(q)

tuplequeue(Empty)             = []
tuplequeue(App(e,p,Empty)) = [(e,p)]
tuplequeue(App(e,p,q))        = [(e,p)] ++ tuplequeue(q)
```

Implementierung von Prioritätswarteschlangen

- Wir haben n Elemente in einer Prioritätswarteschlange und wollen ein neues Element eintragen bzw. das wichtigste auslesen.
- Welche Realisierungsalternativen gibt es und welchen Aufwand verursache diese für das Eintragen bzw. das Auslesen?

1. Anzahl der Prioritäten groß
   - **Liste mit Sortieren beim Einfügen** (*PQueue*):
     Eintragen: $O(n)$, Auslesen: $O(1)$
   - **Liste mit Suche beim Auswählen** (*RQueue*):
     Eintragen: $O(1)$, Auslesen: $O(n)$
   - **Heap-Struktur** (*HQueue*):
     Eintragen: $O(\log n)$, Auslesen: $O(\log n)$

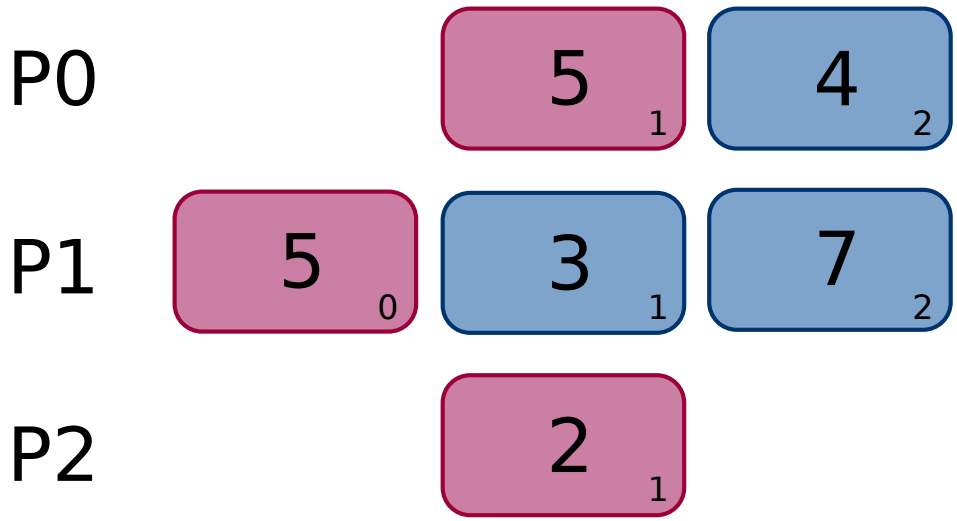
2. Anzahl der Prioritäten klein mit fixem Bereich (z. B. 0-9)
   - **Array von FIFO-Warteschlangen**:
     Eintragen: $O(1)$, Auslesen $O(1)$

Kapitel 5.5

Implementierung als Array von Warteschlangen

Implementierung durch Array von Warteschlangen

- Die Implementierung baut auf der Implementierung einer FIFO-Warteschlange ohne Prioritäten auf.
- Für jede Priorität wird eine eigene Schlange verwendet, in der die Elemente mit dieser Priorität gespeichert werden.
- Die Realisierung ist am einfachsten, wenn ein fixer Bereich von Prioritäten (z. B. 0-9) verwendet wird.
- Die untenstehende Abbildung zeigt eine Realisierung mit drei Ringspeichern und den Prioritäten 0-2.



Implementierung

```
#define PQUEUE_MAX_ELEMENTS 100
#define PQUEUE_MAX_PRIORITY 10
#define PQUEUE_ERROR_ELEMENT INT_MIN

typedef int element;

struct _pqueue {
  int size;
  queue* queues[PQUEUE_MAX_PRIORITY];
};

typedef struct _pqueue pqueue;
```

Implementierung init

```
pqueue* pqueue_init() {
  pqueue* q = pqueue_malloc(sizeof(pqueue));
  q->size = 0;
  for (int i = 0; i < PQUEUE_MAX_PRIORITY; i++)
    q->queues[i] = queue_init();
  return q;
}
```

Implementierung empty, overflow, underflow

```
int pqueue_empty(pqueue* q) {
  return pqueue_underflow(q) ||
              pqueue_length(q) == 0 ||
              pqueue_overflow(q);
}

int pqueue_overflow(pqueue* q) {
  return pqueue_length(q) == PQUEUE_MAX_ELEMENTS + 1;
}

int pqueue_underflow(pqueue* q) {
  return pqueue_length(q) == -1;
}
```

Implementierung pfront

```
element pqueue_front(pqueue* q) {
  if (pqueue_empty(q)) {
    fprintf(stderr, "pfront: empty queue!\n");
    return PQUEUE_ERROR_ELEMENT;
  } else if (pqueue_underflow(q)) {
    fprintf(stderr, "front: underflowed queue!\n");
    return PQUEUE_ERROR_ELEMENT;
  } else if (pqueue_overflow(q)) {
    fprintf(stderr, "front: overflowed queue!\n");
    return PQUEUE_ERROR_ELEMENT;
  }

  // continued on next slide
```

Implementierung pfront

```
  // continued from previous slide

  int i = PQUEUE_MAX_PRIORITY - 1;
  while (i >= 0) {
    if (!queue_empty(q->queues[i]))
      break;
    i--;
  }
  return queue_front(q->queues[i]);
}
```

- Die Schleife wird auf jeden Fall mittels `break` verlassen, da sich mindestens ein Element in der Prioritätswarteschlange befindet.

Implementierung pinsert

```
pqueue* pqueue_insert(element e, int p, pqueue* q) {
  if (pqueue_length(q) == PQUEUE_MAX_ELEMENTS) {
    fprintf(stderr, "pinsert: queue overflow!\n");
    q->size = PQUEUE_MAX_ELEMENTS + 1;
  } else if (p < 0 || p > PQUEUE_MAX_PRIORITY - 1) {
    fprintf(stderr, "pinsert: bad priority");
  } else if (e == PQUEUE_ERROR_ELEMENT) {
    fprintf(stderr,
        "pinsert: trying to insert errorvalue!\n");
  } else if (pqueue_underflow(q) || pqueue_overflow(q))
    fprintf(stderr,
      "pinsert: under-/overflowed queue!\n");
  } else {
    queue_insert(e, q->queues[p]);
    q->size++;
  }
  return q;
}
```

Implementierung premove

```
pqueue* pqueue_remove(pqueue* q) {
  if (pqueue_underflow(q) || pqueue_overflow(q)) {
    fprintf(stderr,
        "premove: under-/overflowed queue!\n");
    return q;
  } else if (pqueue_empty(q)) {
    fprintf(stderr, "premove: queue underflow!\n");
    q->size = -1;
    return q;
  }

  // continued on next slide
```

Implementierung premove

```
  // continued from previous slide

  int i = PQUEUE_MAX_PRIORITY - 1;
  while (i >= 0) {
    if (!queue_empty(q->queues[i]))
      break;
    i--;
  }
  queue_remove(q->queues[i]);
  q->size--;
  return q;
}
```

Implementierung destroy und copy

```
void pqueue_destroy(pqueue* q) {
  for (int i = 0; i < PQUEUE_MAX_PRIORITY; i++)
    queue_destroy(q->queues[i]);
  pqueue_free(q);
}

pqueue* pqueue_copy(pqueue* q) {
  pqueue* c = pqueue_init();
  c->size = q->size;
  for (int i = 0; i < PQUEUE_MAX_PRIORITY; i++) {
    queue_destroy(c->queues[i]);
    c->queues[i] = queue_copy(q->queues[i]);
  }
  return c;
}
```

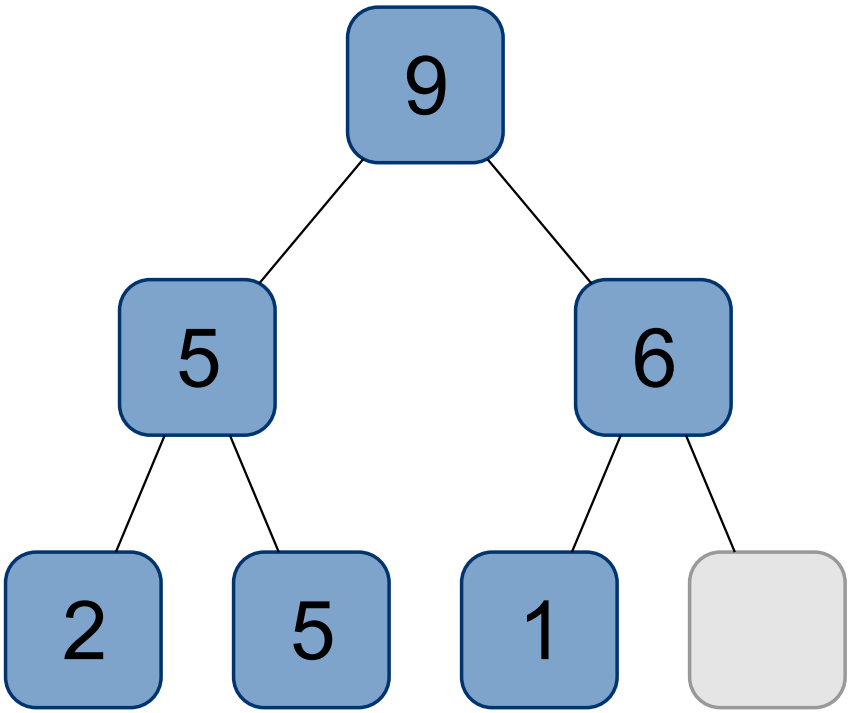
Kapitel 5.6

Implementierung als Heap

Heap

Definition 2 (Heap)

Ein **Heap** *ist ein Binärbaum, bei dem alle Schichten bis auf die unterste vollständig und die Blätter der untersten Schicht linksbündig aufgefüllt sind. Zusätzlich muss der Schlüssel jedes Knotens größergleich den Schlüsseln seiner Kinderknoten sein* → **Max-Heap-Bedingung**.



Heap-Operationen

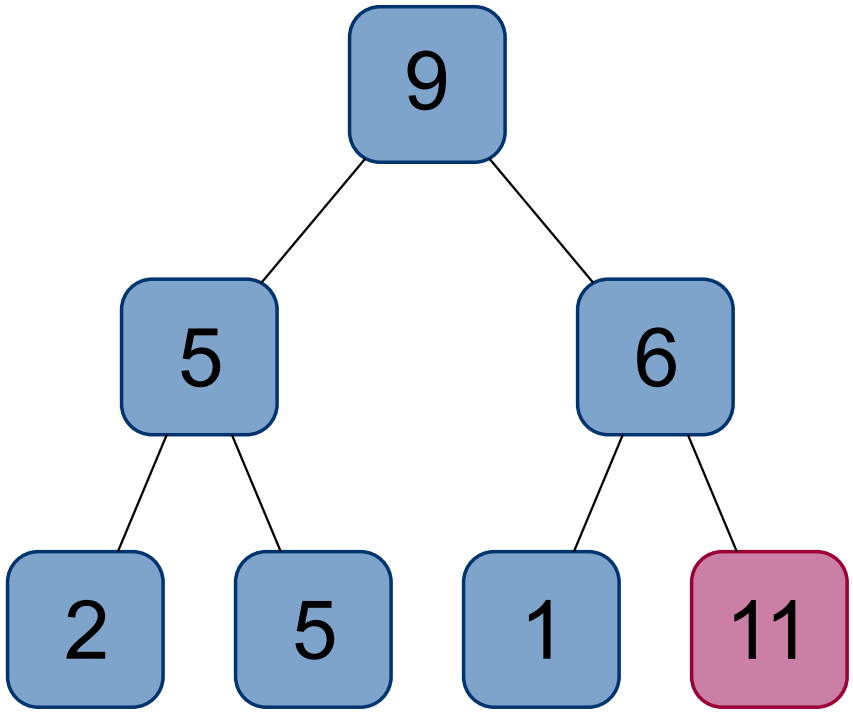
- Heaps unterstützen die für die Implementierung einer Prioritätswarteschlange notwendigen Operationen:
  - `pinsert` Aufwand $O(\log n)$
  - `premove` Aufwand $O(\log n)$
  - `pfront` Aufwand $O(1)$
- Die Realisierung von `pfront` ist einfach, da nur das im Wurzelknoten gespeicherte Element zurückgeliefert werden muss.
- Die Realisierung von `pinsert` und `premove` ist hingegen relativ kompliziert, da die Heap-Bedingung sichergestellt werden muss.
- Für den Anwendungsfall Prioritätswarteschlange muss außerdem die FIFO-Eigenschaft für Elemente mit gleicher Priorität erhalten bleiben.
- Hierfür wird ein Ankunftszähler in den Elementen mitgeführt.

Einfügen in den Heap

- Ein neues Element wird immer als Blatt in den Heap eingefügt.
- Hierbei wird zunächst die unterste, nicht voll besetzte Ebene von links nach rechts aufgefüllt.
- Wenn die unterste Ebene voll ist, wird eine neue Ebene begonnen und das neue Element als ganz linkes Blatt in diese eingefügt.
- Das neue Element **steigt dann in Richtung der Wurzel auf**, bis der Elternknoten ein größeres oder gleichgroßes Element aufweist.
- Im Gegenzug nimmt im Falle eines Aufstiegs das Element des jeweiligen Elternknotens den vorigen Platz des neuen Elements ein.
- Bei gleicher Priorität hat das Element im Elternknoten immer die kleinere Ankunftszeit.
- Deswegen braucht die Ankunftszeit beim Einfügen nicht berücksichtigt werden.

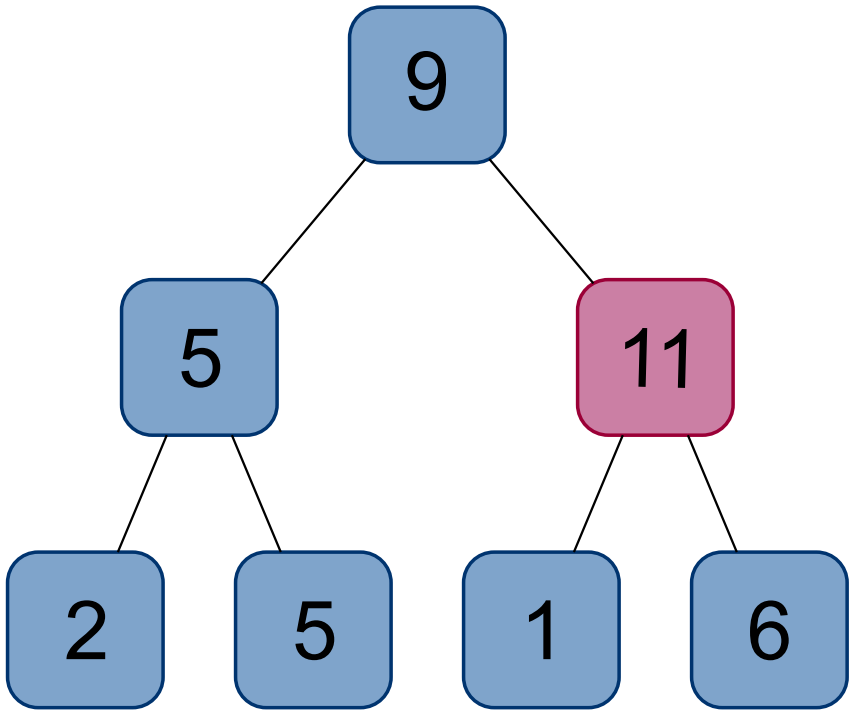
Einfügen in den Heap

- Das Element 11 wird in den Heap eingefügt.
- Da $11 > 6$ ist, ist die Heap-Bedingung verletzt und diese beiden Elementen müssen vertauscht werden.



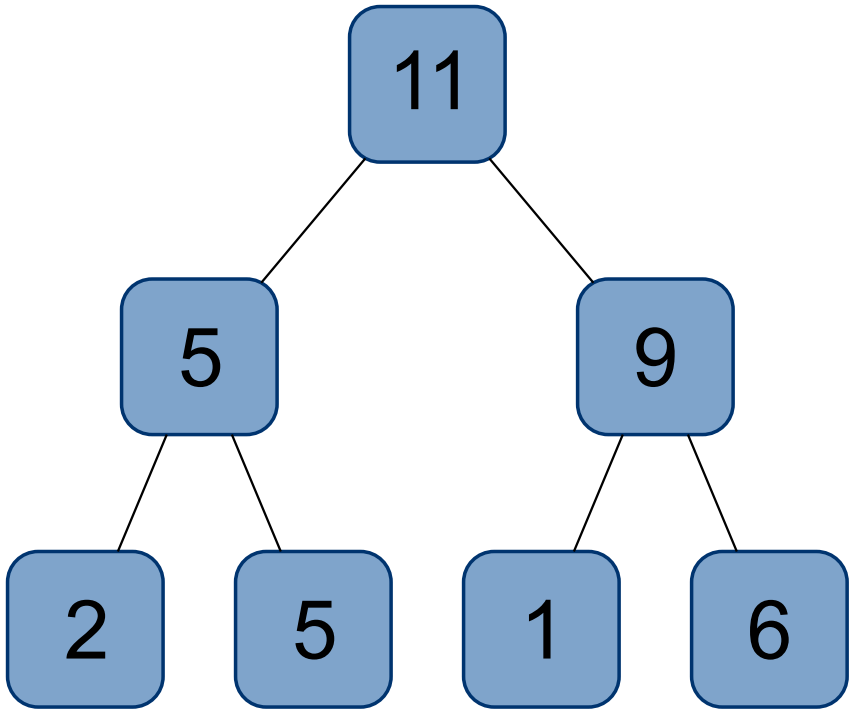
Einfügen in den Heap

- Da $11 > 9$ ist, ist die Heap-Bedingung immer noch verletzt und diese beiden Elementen müssen vertauscht werden.



Einfügen in den Heap

- Jetzt ist die Heap-Bedingung wieder erfüllt.

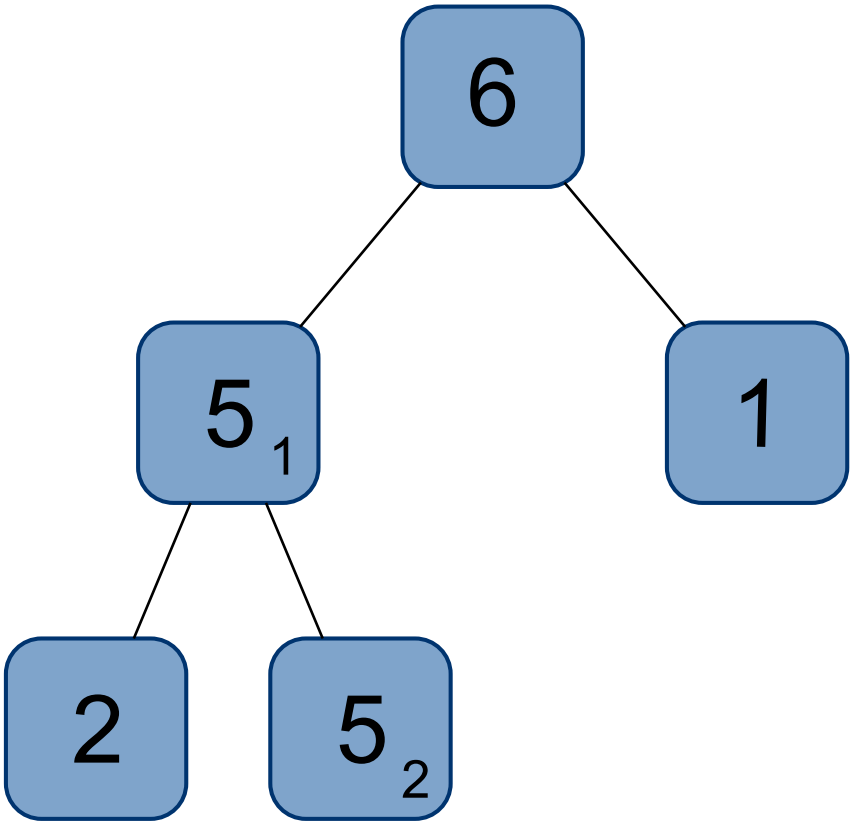


Löschen aus dem Heap

- Beim Löschen aus dem Heap wird das Element der Wurzel entfernt und stattdessen das Element im letzten Blatt als Wurzel eingesetzt.
- Das neue Wurzelelement lässt man nun in den Heap **einsacken**, bis die Heap-Bedingung wieder erfüllt ist.
- Hierfür tauscht dieses wiederholt mit dem größeren Element seiner (neuen) Kinderknoten den Platz, bis beide Kinderknoten kleinere Elemente enthalten.
- Bei gleicher Priorität gilt hierbei ein Element als größer, wenn es eine kleinere Ankunftszeit hat.

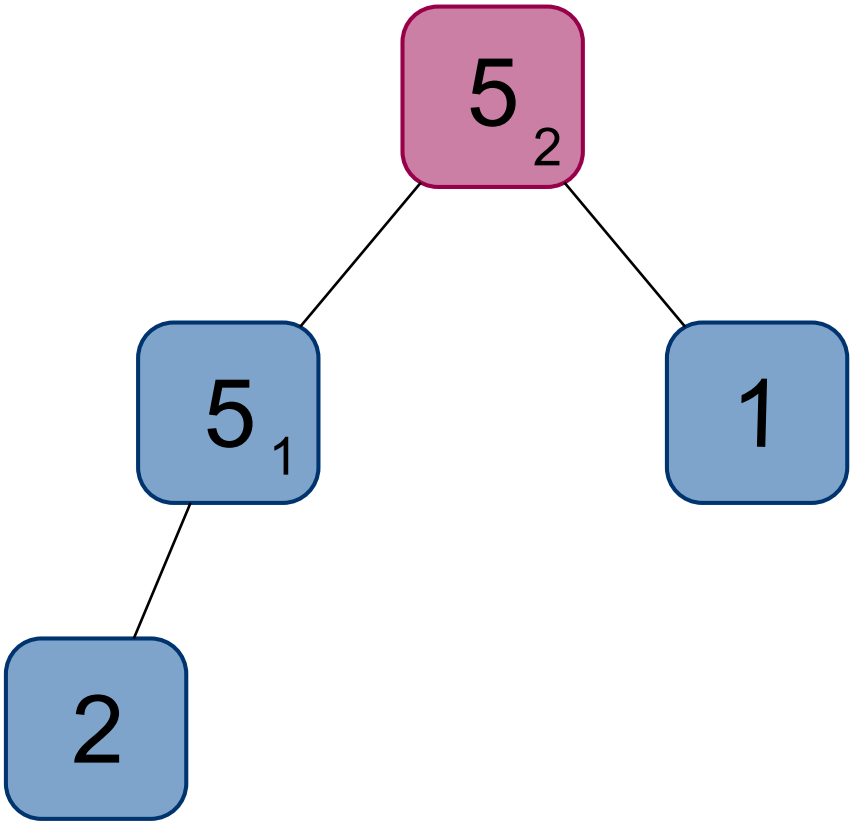
Löschen aus dem Heap

- Aus dem folgenden Heap wird das größte Element entnommen.



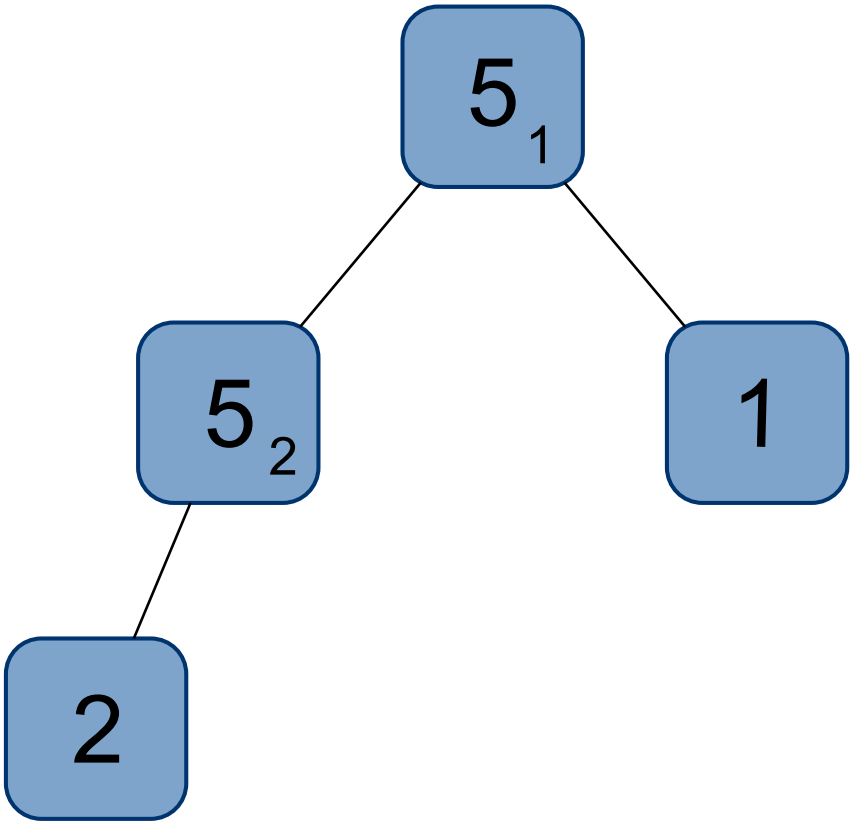
Löschen aus dem Heap

- Die Heap-Bedingung ist zwar schon erfüllt, aber zum Erhalt der FIFO-Eigenschaft muss noch ein Tausch vorgenommen werden.



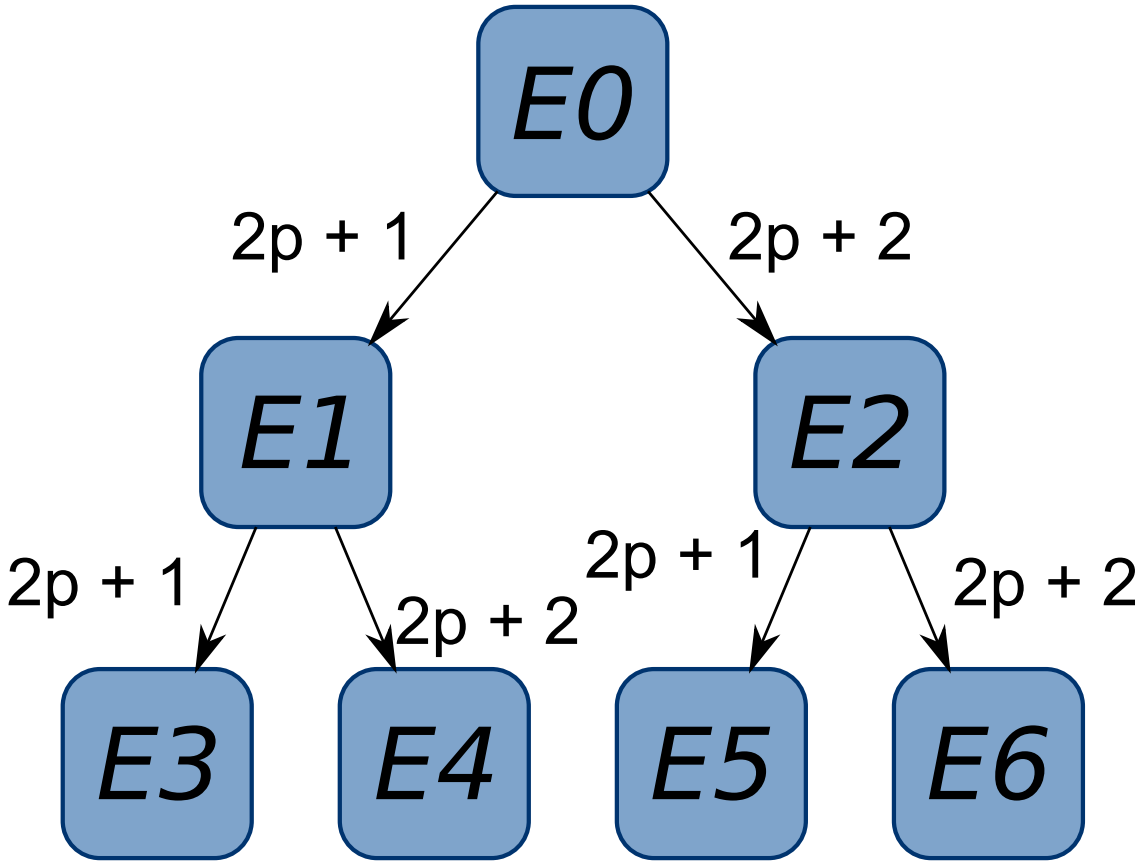
Löschen aus dem Heap

- Jetzt ist die Heap-Bedingung wieder erfüllt und die FIFO-Eigenschaft wird sichergestellt.



Einbettung eines Heaps in ein Array

- Navigation von einem Knoten p zum linken Kindknoten: $2p + 1$.
- Navigation von einem Knoten p zum rechten Kindknoten: $2p + 2$.
- Navigation vom Knoten k zum Elternknoten: $\lfloor (k - 1)/2 \rfloor$.



Implementierung in C

```
#define PQUEUE_MAX_ELEMENTS 100
#define PQUEUE_ERROR_ELEMENT INT_MIN
typedef int element;

struct _node {
  element value;
  int priority;
  long arrival_time;
};
typedef struct _node node;

struct _pqueue {
  int size;
  long time;
  node* nodes[PQUEUE_MAX_ELEMENTS];
};
typedef struct _pqueue pqueue;
```

Implementierung init und destroy

```
pqueue* pqueue_init() {
  pqueue* q = pqueue_malloc(sizeof(pqueue));
  for (int i = 0; i < PQUEUE_MAX_ELEMENTS; i++)
    q->nodes[i] = NULL;
  q->size = 0;
  q->time = 0;
  return q;
}

void pqueue_destroy(pqueue* q) {
  for (int i = 0; i < PQUEUE_MAX_ELEMENTS; i++)
    if (q->nodes[i] != NULL)
      pqueue_free(q->nodes[i]);
  pqueue_free(q);
}
```

Implementierung length, empty, overflow, underflow

```
int pqueue_length(pqueue* q) {
  return q->size;
}

int pqueue_empty(pqueue* q) {
  return pqueue_underflow(q) ||
            pqueue_length(q) == 0 ||
            pqueue_overflow(q);
}

int pqueue_overflow(pqueue* q) {
  return pqueue_length(q) == PQUEUE_MAX_ELEMENTS + 1;
}

int pqueue_underflow(pqueue* q) {
  return pqueue_length(q) == -1;
}
```

Implementierung pfront

```
element pqueue_front(pqueue* q) {
  if (pqueue_empty(q)) {
    fprintf(stderr, "pfront: empty queue!\n");
    return PQUEUE_ERROR_ELEMENT;
  } else if (pqueue_underflow(q)) {
    fprintf(stderr, "front: underflowed queue!\n");
    return PQUEUE_ERROR_ELEMENT;
  } else if (pqueue_overflow(q)) {
    fprintf(stderr, "front: overflowed queue!\n");
    return PQUEUE_ERROR_ELEMENT;
  }

  node* n = q->nodes[0];
  return n->value;
}
```

Implementierung maxprio

```
int pqueue_maxprio(pqueue* q) {
  if (pqueue_empty(q) || pqueue_underflow(q) ||
        pqueue_overflow(q))
    return -1;

  node* n = q->nodes[0];
  return n->priority;
}
```

Implementierung pinsert

```
pqueue* pqueue_insert(element e, int p, pqueue* q) {
  if (pqueue_length(q) == PQUEUE_MAX_ELEMENTS) {
    fprintf(stderr, "pinsert: new queue overflow!\n");
    q->size = PQUEUE_MAX_ELEMENTS + 1;
    return q;
  } else if (e == PQUEUE_ERROR_ELEMENT) {
    fprintf(stderr,
        "pinsert: trying to insert errorvalue!\n");
    return q;
  } else if (pqueue_underflow(q)
                || pqueue_overflow(q)) {
    fprintf(stderr,
        "pinsert: underflowed or overflowed queue!\n");
    return q;
  }

  // continued on next slide
```

Implementierung pinsert

```
  // continued from previous slide

  q->size++;
  q->time++;
  int i = q->size - 1;
  while (i > 0 &&
             q->nodes[(i - 1) / 2]->priority < p) {
    q->nodes[i] = q->nodes[(i - 1) / 2];
    i = (i - 1) / 2;
  }
  node* n = pqueue_malloc(sizeof(node));
  n->value = e;
  n->priority = p;
  n->arrival_time = q->time;
  q->nodes[i] = n;
  return q;
}
```

Implementierung premove

```
pqueue* pqueue_remove(pqueue* q) {
  if (pqueue_underflow(q) || pqueue_overflow(q)) {
    fprintf(stderr,
        "pqueue_remove: under-/overflowed queue!\n");
    return q;
  } else if (pqueue_empty(q)) {
    fprintf(stderr, "premove: new queue underflow!\n");
    q->size = -1;
    return q;
  }
  pqueue_free(q->nodes[0]);
  q->nodes[0] = q->nodes[q->size - 1];
  q->nodes[q->size - 1] = NULL;
  q->size--;
  pqueue_heapify(q, 0);
  return q;
}
```

Implementierung heapify

```
void pqueue_heapify(pqueue* q, int pos) {
  int left = pos * 2 + 1;
  int right = left + 1;
  int largest = pos;
  /* left child exists and has higher priority */
  if (left < pqueue_length(q))
    if (is_greater(q, left, largest))
      largest = left;
  /* right child exists and has higher priority */
  if (right < pqueue_length(q))
    if (is_greater(q, right, largest))
      largest = right;

  // continued on next slide
```

Implementierung heapify

```
  // continued on next slide

  /* swap nodes and continue heapify */
  if (largest != pos) {
    node* n = q->nodes[pos];
    q->nodes[pos] = q->nodes[largest];
    q->nodes[largest] = n;
    pqueue_heapify(q, largest);
  }
}
```

Implementierung isGreater

```
int is_greater(pqueue* q, int first, int second) {
  if (q->nodes[first]->priority >
        q->nodes[second]->priority) {
    return 1;
  } else if (q->nodes[first]->priority ==
                q->nodes[second]->priority) {
    if (q->nodes[first]->arrival_time <
          q->nodes[second]->arrival_time) {
      return 1;
    }
  }
  return 0;
}
```

Implementierung copy

```
pqueue* pqueue_copy(pqueue* q) {
  pqueue* c = pqueue_init();

  for (int i = 0; i < pqueue_length(q); i++)
    pqueue_insert(q->nodes[i]->value,
        q->nodes[i]->priority, c);

  return c;
}
```

Exemplarische Fragen zur Lernkontrolle

1. Wozu dient eine Warteschlange?
2. Was bedeutet FIFO?
3. Welche Operation bietet eine Warteschlange typischerweise an?
4. Spezifizieren Sie alle grundlegenden Operationen der Warteschlange!
5. Erläutern Sie die Implementierung der Warteschlange als Ringpuffer!
6. Worin unterscheidet sich die Prioritätswarteschlange von der Warteschlange ohne Prioritäten?
7. Spezifizieren Sie alle grundlegenden Operationen der Prioritätswarteschlange!
8. Wie funktioniert das Sortieren beim Einfügen?
9. Was ist ein Heap und welche grundlegenden Operationen bietet er an?
10. Wie implementiert man die Priorirätswarteschlange mittels Heap?

Vielen Dank für Ihre Aufmerksamkeit!

Univ.-Prof. Dr.-Ing. Gero Mühl

gero.muehl@uni-rostock.de
https://www.ava.uni-rostock.de