Imperative Programmierung (IPR)

Kapitel 4: Keller

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 4.1

Einführung

Der Keller: Definition

Definition 1 (Keller (engl. stack))

Ein **Keller** *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* **Last In First Out (LIFO)**.

- Ein Keller ist mit einem Stapel von Umzugskisten vergleichbar
  - Es kann eine Kiste oben auf den Stapel gepackt (Operation `push`) oder eine Kiste heruntergenommen werden (Operation `pop`).
  - Aber es kann immer nur auf die oberste Kiste zugegriffen werden (Operation `top`).
- Wenn Elemente nacheinander in den Keller eingefügt werden, so können diese *nur* in *umgekehrter Reihenfolge* entnommen werden.
- Keller werden z. B. für rekursive Problemlösungsverfahren verwendet.

Beispiel: Positiven Integer als Binärzahl ausgeben

Beispiel 1 (Rekursive Lösung)

```
// prints number n as binary number using recursion
void bit(int n) {
  if (n > 1)
    bit(n/2);
  printf("%i", n % 2);
} // 23 --> 10111

bit(23) calls bit(11) and prints 1
bit(11) calls bit( 5) and prints 1
bit( 5) calls bit( 2) and prints 1
bit( 2) calls bit( 2) and prints 0
bit( 1) prints 1
```

Beispiel: Positiven Integer als Binärzahl ausgeben

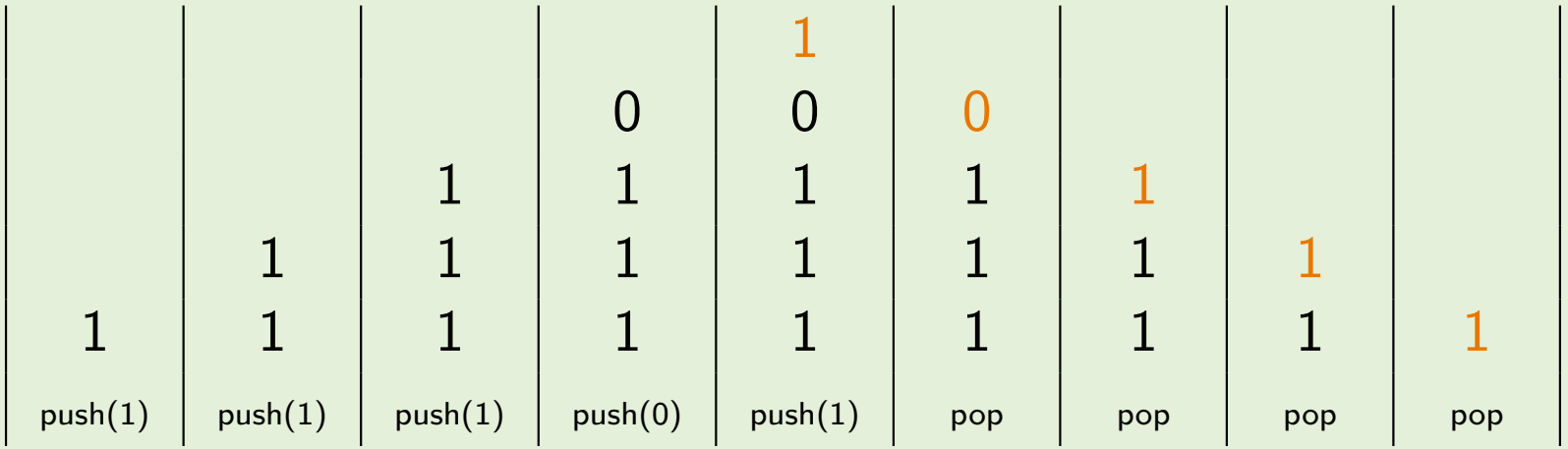
Beispiel 2 (Iterative Lösung mit Stack)

```
// prints number n as binary number using a stack
void bit2(int n) {
  stack* s = stack_init();
  while (n > 0) {
    stack_push(n % 2, s);
    n /= 2;
  }
  while (!(stack_empty(s))) {
    printf("%i", stack_top(s));
    stack_pop(s);
  }
}
```

Beispiel: Positiven Integer als Binärzahl ausgeben

Beispiel 3 (Ausgabe von 23 als Binärzahl)

$$23/2 = 11 \; \textit{Rest} \; 1 \qquad \textit{LSB}\,(1er)$$
$$11/2 = 5 \; \textit{Rest} \; 1$$
$$5/2 = 2 \; \textit{Rest} \; 1$$
$$2/2 = 1 \; \textit{Rest} \; 0$$
$$1/2 = 0 \; \textit{Rest} \; 1 \qquad \textit{MSB}\,(16er)$$



Kapitel 4.2

Spezifikation

Grundlegende Operationen eines Kellers

Operation	Beschreibung der Operation
`init`	Neuer Keller
`empty(s)`	Prüfen, ob Keller leer
`push(e,s)`	Element oben auf Keller legen („kellern“)
`top(s)`	Oberstes Element des Kellers
`pop(s)`	Oberstes Element des Kellers entfernen („entkellern“)
`depth(s)`	Tiefe des Kellers

Ansatz

- Festlegung der *von außen sichtbaren Eigenschaften* der Operationen.
- Für jede Kellerfunktion, die einen „sichtbaren“ Wert liefert, muss präzise festlegt werden, *welchen* Wert sie liefert.
- Von außen sichtbar bei einem Keller sind
  - Boolesche Werte (Ergebnisse von `empty`)
  - Kellerelemente (Ergebnisse von `top`)
  - Natürliche Zahlen (Ergebnisse von `depth`)
- *Nicht* sichtbar ist der interne Aufbau der Werte vom Typ `Stack` (Ergebnisse von `init`, `push`, und `pop`).

Schrittweise Entwicklung der Spezifikation

- Die Beschreibung des Datentyps `Stack` greift zurück auf Boolesche Werte und natürliche Zahlen sowie auf die Elemente, die in einem Keller vorliegen können:

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

- Innerhalb von *Element* soll es außerdem ein Fehlerelement geben (z. B. für den Fall, dass wir mit `top` auf den leeren Keller zugreifen).

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

- Zur einfacheren Darstellung der Eigenschaften definieren wir noch:

  $$\mid \quad \mathit{Element}_V = \mathit{Element} \setminus \{\mathit{errorelement}\}$$

Schrittweise Entwicklung der Spezifikation

- Definition der Menge von Kellern

$$[Stack]$$

- Keller können mit folgenden Funktionen erzeugt werden

$$\begin{array}{|l} init : Stack \\ push : Element \times Stack \longrightarrow Stack \end{array}$$

- Das Hinzufügen des Fehlerelements soll den Keller nicht verändern

$$\begin{array}{|l} \forall\, s : Stack \bullet \\ \qquad push(errorelement, s) = s \end{array}$$

Kelleroperationen: `empty`

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

$$\forall\, e : Element_V;\ s : Stack \bullet$$

$$empty(init) = True$$
$$empty(push(e, s)) = False$$

Beispiel 4 (`empty`)

$$empty(push(\underbrace{3}_{e}, \underbrace{push(4, push(5, init))}_{s})) = False$$

Kelleroperationen: top

$$top : Stack \longrightarrow Element$$

$$\forall\, e : Element_V;\ s : Stack \bullet$$

$$top(init) = errorelement$$
$$top(push(e, s)) = e$$

Beispiel 5 (top)

$$top(push(\underbrace{3}_{e}, \underbrace{push(4, push(5, init))}_{s})) = 3$$

Kelleroperationen: pop

$$pop : Stack \longrightarrow Stack$$

$$\forall\, e : Element_V;\ s : Stack \bullet$$

$$pop(init) = init$$
$$pop(push(e, s)) = s$$

- Einfache Fehlerbehandlung!

Beispiel 6 (pop)

$$pop(push(\underbrace{3}_{e}, \underbrace{push(4, push(5, init))}_{s}))$$
$$= push(4, push(5, init))$$

Kelleroperationen: depth

$$depth : Stack \longrightarrow \mathbb{N}$$

$$\forall\, e : Element_V;\ s : Stack \bullet$$

$$depth(init) = 0$$
$$depth(push(e, s)) = 1 + depth(s)$$

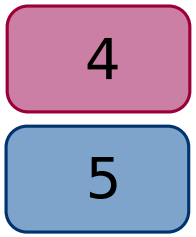
Beispiel 7 (depth)

$$\begin{aligned}
&depth(push(3, push(4, push(5, init))))\\
&= 1 + depth(push(4, push(5, init)))\\
&= 1 + 1 + depth(push(5, init))\\
&= 1 + 1 + 1 + depth(init)\\
&= 1 + 1 + 1 + 0\\
&= 3
\end{aligned}$$

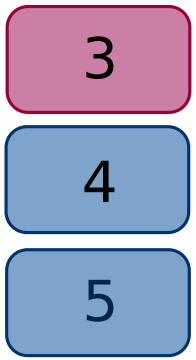
Grafische Repräsentation: `push` und `top`

- Beschreibung des Kellers als Term: $push(3, push(4, push(5, init)))$
- Keller nach $push(5, init)$

- Keller nach $push(4, push(5, init))$

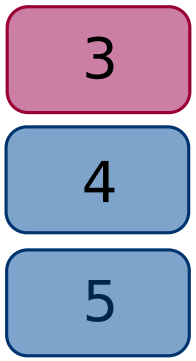


- Keller nach $push(3, push(4, push(5, init)))$

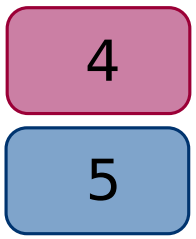


Grafische Repräsentation: `pop` und `top`

- Keller vor pop als Term: $push(3, push(4, push(5, init)))$
- Keller vor pop



- Keller nach $pop(push(3, push(4, push(5, init))))$



- Keller nach $pop(push(4, push(5, init)))$

Ausführbare Spezifikation des Kellers

```
module Stack where
import Prelude hiding (init)
type Element = Int
errorelement = -1

data Stack = Empty | App(Element,Stack)
     deriving Show

init  :: Stack
push  :: (Element,Stack) -> Stack

init = Empty

push(e,s) = if e == errorelement then s
            else App(e,s)
```

Ausführbare Spezifikation des Kellers

```
empty :: Stack -> Bool
top   :: Stack -> Element
pop   :: Stack -> Stack

empty(Empty)     = True
empty(App(e,s)) = False

top(Empty)     = errorelement
top(App(e,s)) = e

pop(Empty)     = Empty
pop(App(e,s)) = s
```

Ausführbare Spezifikation des Kellers

```
depth :: Stack -> Element

depth(Empty)     = 0
depth(App(e,s)) = 1 + depth(s)

tuplestack(Empty)          = []
tuplestack(App(e,Empty)) = [e]
tuplestack(App(e,s))       = [e] ++ tuplestack(s)
```

Kapitel 4.3

Implementierung als Wrapper

Implementierung als Wrapper

- Diese Implementierung verwendet eine der Implementierungen der Liste, greift aber nur über deren Schnittstelle auf diese zu.
- Hierbei nutzt der Keller eine Liste zur Erbringung seiner Funktionalität $\rightarrow$ **Entwurfsmuster der Delegation**
- Die Funktionalität von Listen und Kellern ist sehr ähnlich.
- Daher reicht das einfache Durchschleifen der Methodenaufrufe des Kellers auf die entsprechende Methode der Liste aus.
  $\rightarrow$ **Adapter mit Delegation**

Implementierung

```
#define STACK_ERROR_ELEMENT INT_MIN

typedef int element;

#include "../arraylist/arraylist.h"

struct _stack {
  list* l;
};

typedef struct _stack stack;
```

Implementierung init, empty und top

```
stack* stack_init() {
  stack* s = stack_malloc(sizeof(stack));
  s->l = list_init();
  return s;
}

int stack_empty(stack* s) {
  return list_empty(s->l);
}

element stack_top(stack* s) {
  return list_head(s->l);
}
```

Implementierung push und pop

```
stack* stack_push(element e, stack* s) {
  if (e == STACK_ERROR_ELEMENT) {
    fprintf(stderr,
        "push: trying to insert error element!\n");
    return s;
  }
  s->l = list_insert(e, s->l);
  return s;
}

stack* stack_pop(stack* s) {
  s->l = list_tail(s->l);
  return s;
}
```

Implementierung depth, destroy und copy

```
int stack_depth(stack* s) {
  return list_length(s->l);
}

void stack_destroy(stack* s) {
  list_destroy(s->l);
  stack_free(s);
}

stack* stack_copy(stack* s) {
  stack* ss = stack_malloc(sizeof(stack));
  ss->l = list_copy(s->l);
  return ss;
}
```

Kapitel 4.4

Spezifikation des begrenzten Kellers mit erweiterter Fehlerbehandlung

Fehlerbehandlung

Beispiel 8

- Was halten Sie von folgender „Fehlerbehandlung“?

$$
\begin{aligned}
& top(push(1, pop(init))) \\
& top(push(1, init)) \\
& = 1
\end{aligned}
$$

- Wie wäre es mit folgender Alternative?

$$
\begin{aligned}
& top(push(1, pop(init))) \\
& = top(push(1, underflow)) \\
& = top(underflow) \\
& = errorelement
\end{aligned}
$$

Schrittweise Entwicklung der Spezifikation

- Eingebrachte Mengen und ausgezeichneter Fehlerwert:

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

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

- Definition der Menge von Kellern mit zwei Fehlerelementen:

  $[\mathit{Stack}]$

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

Schrittweise Entwicklung der Spezifikation

- Keller können mit folgenden Funktionen erzeugt werden:

$$\begin{aligned} &\mathit{init} : \mathit{Stack} \\ &\mathit{push} : \mathit{Element} \times \mathit{Stack} \longrightarrow \mathit{Stack} \end{aligned}$$

- Es gibt eine Konstante für die maximale Anzahl von Elementen in einem Keller:

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

- Zur einfacheren Spezifikation der Eigenschaften definieren wir noch:

$$\begin{aligned} &\mathit{Element}_V = \mathit{Element} \setminus \{\mathit{errorelement}\} \\ &\mathit{Stack}_V = \mathit{Stack} \setminus \{\mathit{underflow}, \mathit{overflow}\} \\ &\mathit{Stack}_X = \{s \in \mathit{Stack} \mid \mathit{depth}(s) = \mathit{maxelements}\} \end{aligned}$$

Eigenschaften des begrenzten Kellers mit erweiterter Fehlerbehandlung: `push`

$$push : Element \times Stack \longrightarrow Stack$$

$\forall\, e : Element_V;\ s : Stack \bullet$

$$push(errorelement, s) = s$$
$$push(e, underflow) = underflow$$
$$push(e, overflow) = overflow$$

$\forall\, e : Element_V;\ s : Stack_X \bullet$

$$push(e, s) = overflow$$

Eigenschaften des begrenzten Kellers mit erweiterter Fehlerbehandlung: `empty`

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

$$\forall\, e : Element_V, s : Stack_V \setminus Stack_X \bullet$$

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

$$empty(init) = True$$
$$empty(push(e, s)) = False$$

Eigenschaften des begrenzten Kellers mit erweiterter Fehlerbehandlung: `top`

$$top : Stack \longrightarrow Element$$

$$\forall\, e : Element_V, s : Stack_V \setminus Stack_X \bullet$$

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

$$top(push(e, s)) = e$$

Eigenschaften des begrenzten Kellers mit erweiterter Fehlerbehandlung: pop

$$pop : Stack \longrightarrow Stack$$

$$\forall\, e : Element_V, s : Stack_V \setminus Stack_X \bullet$$

$$pop(init) = underflow$$
$$pop(underflow) = underflow$$
$$pop(overflow) = overflow$$

$$pop(push(e, s)) = s$$

Eigenschaften des begrenzten Kellers mit erweiterter Fehlerbehandlung: `depth`

$$depth : Stack \longrightarrow \mathbb{Z}$$

$$\forall\, e : Element_V, s : Stack_V \setminus Stack_X \bullet$$

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

$$depth(init) = 0$$
$$depth(push(e, s)) = depth(s) + 1$$

Ausführbare Spezifikation des erweiterten Kellers

```
module Stack where
import Prelude hiding (init)

type Element = Int
errorelement = -1

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

maxelements = 5

init :: Stack

init = Empty
```

Ausführbare Spezifikation des erweiterten Kellers

```
push  :: (Element, Stack) -> Stack
empty :: Stack -> Bool

push(e,s)  =
    if e == errorelement then s
    else if s == Underflow then s
    else if depth(s) > maxelements - 1 then Overflow
    else App(e,s)

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

Ausführbare Spezifikation des erweiterten Kellers

```
top   :: Stack -> Element
pop   :: Stack -> Stack
depth :: Stack -> Element

top(Empty)      = errorelement
top(Underflow)  = errorelement
top(Overflow)   = errorelement
top(App(e,s))   = e

pop(Empty)      = Underflow
pop(Underflow)  = Underflow
pop(Overflow)   = Overflow
pop(App(e,s))   = s
```

Ausführbare Spezifikation des erweiterten Kellers

```
depth :: Stack -> Element

depth(Empty)      = 0
depth(Overflow)   = maxelements + 1
depth(Underflow)  = -1
depth(App(e,s))   = depth(s) + 1

tuplestack(Empty)          = []
tuplestack(App(e,Empty)) = [e]
tuplestack(App(e,s))       = [e] ++ tuplestack(s)
```

Kapitel 4.5

Array-basierte Implementierung

Array-basierte Implementierung

- Implementierung analog zur Array-basierten Liste unter Berücksichtigung von Größenbeschränkung und erweiterter Fehlerbehandlung.

```
#define STACK_MAX_ELEMENTS 100
#define STACK_ERROR_ELEMENT INT_MIN

typedef int element;

struct _stack {
  int size;
  element elements[STACK_MAX_ELEMENTS];
};

typedef struct _stack stack;
```

Implementierung init und destroy

```
stack* stack_init() {
  stack* s = stack_malloc(sizeof(stack));
  s->size = 0;
  return s;
}

void stack_destroy(stack* s) {
  stack_free(s);
}
```

Implementierung depth und overflow, und underflow

```
int stack_depth(stack* s) {
  return s->size;
}

int stack_overflow(stack* s) {
  return stack_depth(s) == STACK_MAX_ELEMENTS + 1;
}

int stack_underflow(stack* s) {
  return stack_depth(s) == -1;
}
```

Implementierung empty und top

```
int stack_empty(stack* s) {
  return stack_underflow(s) ||
      stack_depth(s) == 0 || stack_overflow(s);
}

element stack_top(stack* s) {
  if (stack_underflow(s))
    fprintf(stderr, "top: underflowed stack!\n");
  else if (stack_overflow(s))
    fprintf(stderr, "top: overflowed stack!\n");
  else if (stack_empty(s))
    fprintf(stderr, "top: empty stack!\n");
  else
    return s->elements[stack_depth(s) - 1];

  return STACK_ERROR_ELEMENT;
}
```

Implementierung push

```
stack* stack_push(element e, stack* s) {
  if (e == STACK_ERROR_ELEMENT)
    fprintf(stderr,
        "push: trying to push errorelement!\n");
  else if (stack_underflow(s) || stack_overflow(s)) {
    fprintf(stderr,
        "push: underflowed or overflowed stack!\n");
  } else if (stack_depth(s) == STACK_MAX_ELEMENTS) {
    fprintf(stderr,
        "push: new stack overflow!\n");
    s->size = STACK_MAX_ELEMENTS + 1;
  } else
    s->elements[s->size++] = e;
  return s;
}
```

Implementierung pop

```
stack* stack_pop(stack* s) {
  if (stack_underflow(s) || stack_overflow(s)) {
    fprintf(stderr,
        "push: underflowed or overflowed stack!\n");
  } else if (stack_empty(s)) {
    fprintf(stderr, "pop: new stack underflow!\n");
    s->size = -1;
  } else
    s->size--;

  return s;
}
```

Kapitel 4.6

Implementierung als verkettete Datenstruktur mit gekapselten Elementen

Implementierung

- Analog zur verketteten Liste mit gekapselten Elementen.

```
#define STACK_ERROR_ELEMENT INT_MIN
#define STACK_MAX_ELEMENTS 100
typedef int element;

struct _node {
  element value;
  struct _node* next;
};
typedef struct _node node;

struct _stack {
  int size;
  node* top;
};
typedef struct _stack stack;
```

Implementierung init und destroy

```
stack* stack_init() {
  stack* m = stack_malloc(sizeof(stack));
  m->size = 0;
  m->top = NULL;
  return m;
}

void stack_destroy(stack* s) {
  while (!stack_empty(s))
    stack_pop(s);
  stack_free(s);
}
```

Implementierung overflow, underflow und empty

```
int stack_overflow(stack* s) {
  return stack_depth(s) == STACK_MAX_ELEMENTS + 1;
}

int stack_underflow(stack* s) {
  return stack_depth(s) == -1;
}

int stack_empty(stack* s) {
  return (stack_underflow(s) ||
      stack_depth(s) == 0 || stack_overflow(s));
}
```

Implementierung depth und top

```
int stack_depth(stack* s) {
  return s->size;
}

element stack_top(stack* s) {
  if (stack_underflow(s))
    fprintf(stderr, "top: underflowed stack!\n");
  else if (stack_overflow(s))
    fprintf(stderr, "top: overflowed stack!\n");
  else if (stack_empty(s))
    fprintf(stderr, "top: empty stack!\n");
  else
    return s->top->value;

  return STACK_ERROR_ELEMENT;
}
```

Implementierung push

```
stack* stack_push(element e, stack* s) {
  if (e == STACK_ERROR_ELEMENT)
    fprintf(stderr, "push: error element!\n");
  else if (stack_underflow(s) || stack_overflow(s))
    fprintf(stderr,
        "push: underflowed or overflowed stack!\n");
  else if (stack_depth(s) == STACK_MAX_ELEMENTS) {
    fprintf(stderr, "push: new stack overflow!\n");
    s->size = STACK_MAX_ELEMENTS + 1;
  } else {
    node* n = stack_malloc(sizeof(node));
    n->value = e;
    n->next = s->top;
    s->top = n;
    s->size++;
  }
  return s;
}
```

Implementierung pop

```
stack* stack_pop(stack* s) {
  if (stack_underflow(s) || stack_overflow(s))
    fprintf(stderr,
        "pop: underflowed or overflowed stack!\n");
  else if (stack_empty(s)) {
    fprintf(stderr, "pop: new stack underflow!\n");
    s->size = -1;
  } else {
    node* tmp = s->top;
    s->top = s->top->next;
    stack_free(tmp);
    s->size--;
  }
  return s;
}
```

Exemplarische Fragen zur Lernkontrolle

1. Wozu dient ein Keller?
2. Welche Operation bietet ein Keller typischerweise an?
3. Was bedeutet LIFO?
4. Spezifizieren Sie alle grundlegenden Kelleroperationen!
5. Was versteht man unter dem Entwurfsmuster der Delegation?
6. Wie kann ein Keller auf Basis einer Liste implementiert werden?
7. Wozu dient die erweiterte Fehlerbehandlung?
8. Wie kann ein Keller für einen rekursiven Algorithmus genutzt werden?

Vielen Dank für Ihre Aufmerksamkeit!

Univ.-Prof. Dr.-Ing. Gero Mühl

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