



Wintersemester 2006/07

**Einführung in die Informatik für
Naturwissenschaftler und Ingenieure**
(alias **Einführung in die Programmierung**)
(Vorlesung)

Prof. Dr. Günter Rudolph
Fachbereich Informatik
Lehrstuhl für Algorithm Engineering



Kapitel 8: Abstrakte Datentypen



Inhalt

- Definition
- ADT Keller
- ADT Schlange
- ADT Liste
- ADT Binärbaum
- ...



Kapitel 8: Abstrakte Datentypen



Definition:

Abstrakter Datentyp (ADT) ist ein Tripel (T, F, A), wobei

- T eine nicht leere Menge von Datenobjekten
- F eine Menge von Operationen,
- A eine nicht leere Menge von Axiomen, die die Bedeutung der Operationen erklären.

Abstrakt?

- Datenobjekte brauchen keine konkrete Darstellung (Verallgemeinerung).
- Die Wirkung der Operationen wird beschrieben, nicht deren algorithmische Ausprägung.
→ „**WAS, nicht WIE!**“

Kapitel 8: Abstrakte Datentypen



Beispiel: ADT bool

F: Operationen

true	:		→ bool	}	Festlegung, welche Funktionen es gibt
false	:		→ bool		
not	:	bool	→ bool		
and	:	bool x bool	→ bool		
or	:	bool x bool	→ bool		

A: Axiome

not(false)	=	true	}	Festlegung, was die Funktionen bewirken
not(true)	=	false		
and(false, false)	=	false		
and(false, true)	=	false		
and(true, false)	=	false		
and(true, true)	=	true		
or(x, y)	=	not(and(not(x), not(y)))		

Eigenschaften

- Wenn man ADT kennt, dann kann man ihn überall verwenden.
- Implementierung der Funktionen für Benutzer nicht von Bedeutung.
- Trennung von Spezifikation und Implementierung.
- Ermöglicht späteren Austausch der Implementierung, ohne dass sich der Ablauf anderer Programme, die ihn benutzen, ändert!

Nur Operationen geben Zugriff auf Daten!

→ Stichwort: Information Hiding!

Lineare Datenstrukturen: Keller bzw. Stapel (*engl. stack*)

```

create   :           → Keller
push    : Keller x T → Keller
pop     : Keller     → Keller
top     : Keller     → T
empty   : Keller     → bool
    
```

Aufräumen:
Kiste in Keller,
oben auf Haufen.

Etwas aus Keller holen:
Zuerst Kiste, weil oben
auf Haufen.

```

empty(create) = true
empty(push(k, x)) = false
pop(push(k, x)) = k
top(push(k, x)) = x
    
```



LIFO:
Last in, first out.

Implementierung: (Version 1)

```

const int maxSize = 100;
struct Keller {
    T data[maxSize]; // data array
    int sp;          // stack pointer
};
    
```

```

void create(Keller &k) {
    k.sp = -1;
}
    
```

```

T top(Keller k) {
    return k.data[k.sp];
}
    
```

```

bool empty(Keller k) {
    return k.sp == -1;
}
    
```

```

void push(Keller &k, T x) {
    k.data[++k.sp] = x;
}
    
```

```

void pop(Keller &k) {
    k.sp--;
}
    
```

**Probleme:
Arraygrenzen!**

Wann können Probleme auftreten?

Bei **pop**, falls Keller leer ist:

→ Stackpointer wird -2, anschließendes **push** versucht auf **data[-1]** zu schreiben

Bei **top**, falls Keller leer ist:

→ es wird undefinierter Wert von **data[-1]** zurückgegeben

Bei **push**, falls Keller voll ist:

→ es wird versucht auf **data[maxsize]** zu schreiben

⇒ diese Fälle müssen abgefangen werden! **Fehlermeldung!**

```

void error(char *info) {
    cerr << info << endl;
    exit(1);
}
    
```

gibt Fehlermeldung **info** aus und bricht das Programm durch **exit(1)** **sofort** ab und liefert den Wert des Arguments (hier: 1) an das Betriebssystem zurück

Kapitel 8: Abstrakte Datentypen

Implementierung: (Version 2, nur Änderungen und Zusätze bei Funktionen)

```
const int maxSize = 100;
struct Keller {
    T data[maxSize]; // data array
    int sp;          // stack pointer
};
```

unverändert

```
T top(Keller k) {
    if (!empty(k))
        return k.data[k.sp];
    else error("leer");
}
```

```
void push(Keller &k, T x) {
    if (!full(k))
        k.data[++k.sp] = x;
    else error("voll");
}
```

```
void pop(Keller &k) {
    if (!empty(k)) k.sp--;
    else error("leer");
}
```

```
bool full(Keller k) {
    return k.sp == maxSize-1;
}
```

Kapitel 8: Abstrakte Datentypen

Lineare Datenstrukturen: Schlange (engl. queue)

FIFO:
First in, first out.

```
create :           → Schlange
enq  : Schlange x T → Schlange
deq  : Schlange    → Schlange
front: Schlange    → T
empty: Schlange    → bool
```

Schlange an der Supermarktkasse:
Wenn Einkauf fertig, dann **hinten** anstellen. Der nächste Kunde an der Kasse steht ganz **vorne** in der Schlange.

```
empty(create) = true
empty(enq(s, x)) = false
deq(enq(s, x)) = empty(s) ? s : enq(deq(s), x)
front(enq(s, x)) = empty(s) ? x : front(s)
```

Eingehende Aufträge werden „geparkt“, und dann nach und nach in der Reihenfolge des Eingangs abgearbeitet.

Kapitel 8: Abstrakte Datentypen

Implementierung: (Version 1)

```
const int maxSize = 100;
struct Schlange {
    T data[maxSize]; // data array
    int ep;          // end pointer
};
```

```
void create(Schlange &s) {
    s.ep = -1;
}
```

```
void enq(Schlange &s, T x) {
    s.data[++s.ep] = x;
}
```

```
bool empty(Schlange s) {
    return s.ep == -1;
}
```

```
void deq(Schlange &s) {
    for (int i=0; i<s.ep; i++)
        s.data[i] = s.data[i+1];
    s.ep--;
}
```

```
T front(Schlange s) {
    return s.data[0];
}
```

```
void enq(Schlange &s, T x) {
    if (!full(s))
        s.data[++s.ep] = x;
    else error("full");
}
```

```
bool full(Schlange s) {
    return
        s.ep == maxSize-1;
}
```

Probleme: Arraygrenzen!

Kapitel 8: Abstrakte Datentypen

Implementierung: (Version 2, nur Änderungen und Zusätze bei Funktionen)

```
const int maxSize = 100;
struct Schlange {
    T data[maxSize]; // data array
    int ep;          // end pointer
};
```

unverändert

```
T front(Schlange s) {
    if (!empty(s))
        return s.data[0];
    error("leer");
}
```

```
void enq(Schlange &s, T x) {
    if (!full(s))
        s.data[++s.ep] = x;
    else error("full");
}
```

```
void deq(Schlange &s) {
    if (empty(s)) error("leer");
    for (int i=0; i<s.ep; i++)
        s.data[i] = s.data[i+1];
    s.ep--;
}
```

```
bool full(Schlange s) {
    return
        s.ep == maxSize-1;
}
```

Kapitel 8: Abstrakte Datentypen

Benutzer des (abstrakten) Datentyps `Schlange` wird feststellen, dass

1. fast alle Operationen schnell sind, aber
2. die Operation `deq` vergleichsweise langsam ist.

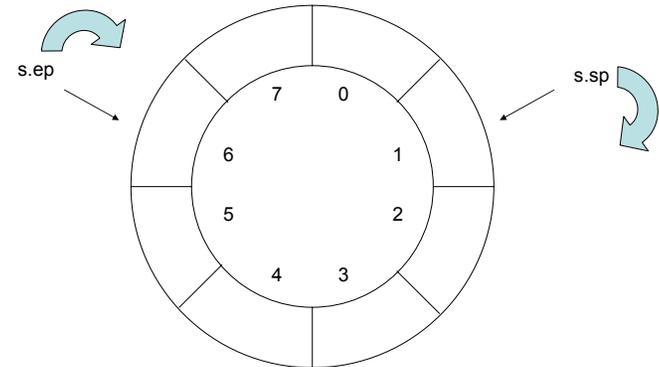
Laufzeit / Effizienz der Operation `deq`

```
void deq(Schlange &s) {
    if (empty(s)) error("leer");
    for (int i=0; i < s.ep; i++)
        s.data[i] = s.data[i+1];
    s.ep--;
}
```

`s.ep` = Anzahl Elemente in Schlange
`s.ep` viele Datenverschiebungen
Worst case: $(\text{maxSize} - 1)$ mal

Kapitel 8: Abstrakte Datentypen

Idee: Array zum Kreis machen; zusätzlich Anfang/Start markieren (`s.sp`)



Kapitel 8: Abstrakte Datentypen

Implementierung: (Version 3)

```
const int maxSize = 100;
struct Schlange {
    T data[maxSize]; // data array
    int sp;          // start pointer
    int ep;          // end pointer
};
```

```
void create(Schlange& s) {
    s.sp = 0; s.ep = maxSize;
}
```

```
bool empty(Schlange s) {
    return s.ep == maxSize;
}
```

```
bool full(Schlange s) {
    int d = s.ep - s.sp;
    return (d == -1) || (d == s.ep);
}
```

```
T front(Schlange s) {
    if (!empty(s))
        return s.data[s.sp];
    error("leer");
}
```

Kapitel 8: Abstrakte Datentypen

Implementierung: (Version 3)

```
void enq(Schlange &s, T x) {
    if (!full(s)) {
        s.ep = (s.ep + 1) % maxSize;
        s.data[s.ep] = x;
    }
    else error("full");
}
```

Laufzeit:
unabhängig von Größe
der Schlange

```
void deq(Schlange &s) {
    if (empty(s)) error("leer");
    else if (s.sp == s.ep) create(s);
    else s.sp = (s.sp + 1) % maxSize;
}
```

Laufzeit:
unabhängig von Größe
der Schlange

Kapitel 8: Abstrakte Datentypen

Unbefriedigend bei der Implementierung:

Maximale festgelegte Größe des Kellers bzw. der Schlange!

→ Liegt an der unterliegenden Datenstruktur Array:

Array ist **statisch**, d.h.

Größe wird zur Übersetzungszeit **festgelegt**

und ist **während** der **Laufzeit** des Programms **nicht veränderbar!**

Schön wären **dynamische** Datenstrukturen, d.h.

Größe wird zur Übersetzungszeit **nicht festgelegt**

und ist **während** der **Laufzeit** des Programms **veränderbar!**

Wir wissen:

Lebensdauer benannter Objekte bestimmt durch ihren Gültigkeitsbereich!

Wir wollen:

Objekte erzeugen, die unabhängig von ihrem Gültigkeitsbereich existieren!

Exkurs: Dynamischer Speicher

Erzeugen und löschen eines Objekts zur Laufzeit:

1. Operator **new** erzeugt Objekt
2. Operator **delete** löscht zuvor erzeugtes Objekt

Beispiel: (Erzeugen)

```
int *xp = new int;
double *yp = new double;

struct PunktT {
    int x, y;
};

PunktT *pp = new PunktT;
```

```
int *xap = new int[10];
PunktT *pap = new PunktT[10];
```

Beispiel: (Löschen)

```
delete xp;
delete yp;
```

```
delete pp;
```

```
delete[] xap;
delete[] pap;
```

Exkurs: Dynamischer Speicher

Bauplan:

Datentyp *Variable = **new** Datentyp; (Erzeugen)

delete Variable; (Löschen)

Bauplan für Arrays:

Datentyp *Variable = **new** Datentyp [Anzahl]; (Erzeugen)

delete[] Variable; (Löschen)

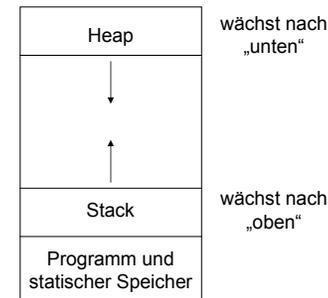
Achtung:

Dynamisch erzeugte Objekte müssen auch wieder gelöscht werden!
Keine automatische Speicherbereinigung!

Exkurs: Dynamischer Speicher

Wo wird Speicher angelegt?

⇒ im **Freispeicher** alias **Heap** alias **dynamischen Speicher**



wenn Heapgrenze
auf Stackgrenze stößt:
Out of Memory Error



Stack bereinigt sich selbst,
für Heap ist Programmierer
verantwortlich!