technische universität
dortmund

Einführung in die Programmierung

Wintersemester 2020/21

Kapitel 9: Elementare Datenstrukturen

M.Sc. Roman Kalkreuth
Lehrstuhl für Algorithm Engineering (LS11)
Fakultät für Informatik

Inhalt

- Definition: Abstrakter Datentyp (ADT)
- ADT Stapel
- ADT Schlange
- ADT Liste
- ADT Binärer Suchbaum
- ADT Graph
- Exkurse:
  - Einfache Dateibehandlung
  - C++-Strings

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!“**

Beispiel: ADT bool

F: **Operationen**

```
true :                  → bool
false      :            → bool
not  : bool             → bool
and : bool  x  bool     → bool
or   : bool  x  bool    → bool
```

Festlegung, **welche Methoden** es gibt

A: **Axiome**

```
not(false)          = true
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)))
```

Festlegung, **was** die Methoden bewirken

Eigenschaften

- Wenn man einen 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**

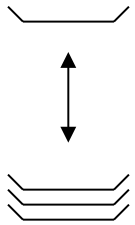
Lineare Datenstrukturen: Keller bzw. **Stapel** (*engl. stack*)

create	:		→ Stapel
push	:	Stapel x T	→ Stapel
pop	:	Stapel	→ Stapel
top	:	Stapel	→ T
empty	:	Stapel	→ bool

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

Aufräumen:
Kiste in den Keller,
oben auf Haufen.

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



Stapel
Teller

LIFO:
Last in, first out.

Klassendefinition: (Version 1)

```
template<typename T>
class Stapel {
public:
  Stapel();          // Konstruktor
  void push(T &x);   // Element auf den Stapel legen
  void pop();        // oberstes Element entfernen
  T top();           // oberstes Element ansehen
  bool empty();      // Stapel leer?
private:
  static unsigned int const maxSize = 100;
  int sz;            // Stapelzeiger
  T data[maxSize];   // Speichervorrat für Nutzdaten
};
```

Alternative: anonymer enum („*the enum trick*“)

```
  enum { maxSize = 100 };
```

Implementierung: (Version 1)

```
template<typename T>
Stapel<T>::Stapel() {
  sz = -1;
}
```

```
template<typename T>
void Stapel<T>::push(T &x) {
  data[++sz] = x;
}
template<typename T>
void Stapel<T>::pop() {
  sz--;
}
template<typename T>
T Stapel<T>::top() {
  return data[sz];
}
template<typename T>
bool Stapel<T>::empty() {
  return (sz == -1);
}
```

Idee:
unzulässiger Arrayindex -1
kennzeichnet leeren Stapel

Problem:
Arraygrenzen!

Wann können Probleme auftreten?

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

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

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

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

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

→ es wird versucht auf **data[maxSize]** zu schreiben (erlaubt: 0 bis maxSize – 1)

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

```
void error(char const *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

Klassendefinition: (Version 2; Ergänzungen in **rot**)

```
template<typename T>
class Stapel {
public:
  Stapel();           // Konstruktor
  void push(T &x);    // Element auf den Stapel legen
  void pop();         // oberstes Element entfernen
  T top();            // oberstes Element ansehen
  bool empty();       // Stapel leer?
  bool full();        // Stapel voll?
private:
  static unsigned int const maxSize = 100;
  int sz;             // Stapelzeiger
  T data[maxSize];    // Speichervorat für Nutzdaten
  void error(char const *info); // Fehlermeldung + Abbruch
};
```

Implementierung: (Version 2, Änderungen und Zusätze in **rot**)

```
template<typename T>
Stapel<T>::Stapel() {
  sz = -1;
}
template<typename T>
void Stapel<T>::push(T &x) {
  if (full()) error("voll");
  data[++sz] = x;
}
template<typename T>
void Stapel<T>::pop() {
  if (empty()) error("leer");
  sz--;
}
```

```
template<typename T>
T Stapel<T>::top() {
  if (empty()) error("leer");
  return data[sz];
}
template<typename T>
bool Stapel<T>::empty() {
  return (sz == -1);
}
template<typename T>
bool Stapel<T>::full() {
  return (sz == maxSize - 1);
}
```

```
template<typename T>
void Stapel<T>::error(char const * info) {
  std::cerr << info << std::endl;
  exit(1);
}
```

← **private Methode**:
kann nur innerhalb der Klasse aufgerufen werden

Erster Test ...

```
#include <iostream>
#include "Stapel.h"
using namespace std;

int main() {
  Stapel<int> s;
  for (int i = 0; i < 100; i++) s.push(i);
  cout << s.top() << endl;
  for (int i = 0; i <  90; i++) s.pop();
  while (!s.empty()) {
    cout << s.top() << endl;
    s.pop();
  }
  return 0;
}
```

Ausgabe:

```
99
9
8
7
6
5
4
3
2
1
0
```

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.

Klassendefinition: (Version 1; schon mit Fehlerbehandlung)

```
template<typename T>
class Schlange {
public:
  Schlange();             // Konstruktor
  void enq(T &x);         // Element anhängen
  void deq();             // erstes Element entfernen
  T front();              // erstes Element ansehen
  bool empty();           // Schlange leer?
  bool full();            // Schlange voll?
private:
  static unsigned int const maxSize = 100;
  int ez;                 // Endezeiger
  T data[maxSize];        // Array für Nutzdaten
  void error(char const *info);  // Fehlermeldung
};
```

Implementierung: (Version 1; Fehler bei Arraygrenzen werden abgefangen)

```
template<typename T>
Schlange<T>::Schlange() : ez(-1) {
}
template<typename T>
void Schlange<T>::enq(T &x) {
  if (full()) error("voll");
  data[++ez] = x;
}
template<typename T>
void Schlange<T>::deq() {
  if (empty()) error("leer");
  for (int i = 0; i < ez; i++)
    data[i] = data[i+1];
  ez--;
}
```

```
template<typename T>
T Schlange<T>::front() {
  if (empty()) error("leer");
  return data[0];
}
template<typename T>
bool Schlange<T>::empty() {
  return (ez == -1);
}
template<typename T>
bool Schlange<T>::full() {
  return (ez == maxSize - 1);
}
```

```
template<typename T>
void Schlange<T>::error(char const *info) {
  std::cerr << info << std::endl;
  exit(1);
}
```

← **private Methode**:
kann nur innerhalb der Klasse aufgerufen werden

Erster Test ...

```
#include <iostream>
#include "Schlange.h"
using namespace std;

int main() {
  Schlange<int> s;

  for (int i = 0; i < 100; i++) s.enq(i);
  cout << s.front() << endl;
  for (int i = 0; i < 90; i++) s.deq();
  while (!s.empty()) {
    cout << s.front() << endl;
    s.deq();
  }
  return 0;
}
```

Ausgabe:

```
0
90
91
92
93
94
95
96
97
98
99
```

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**

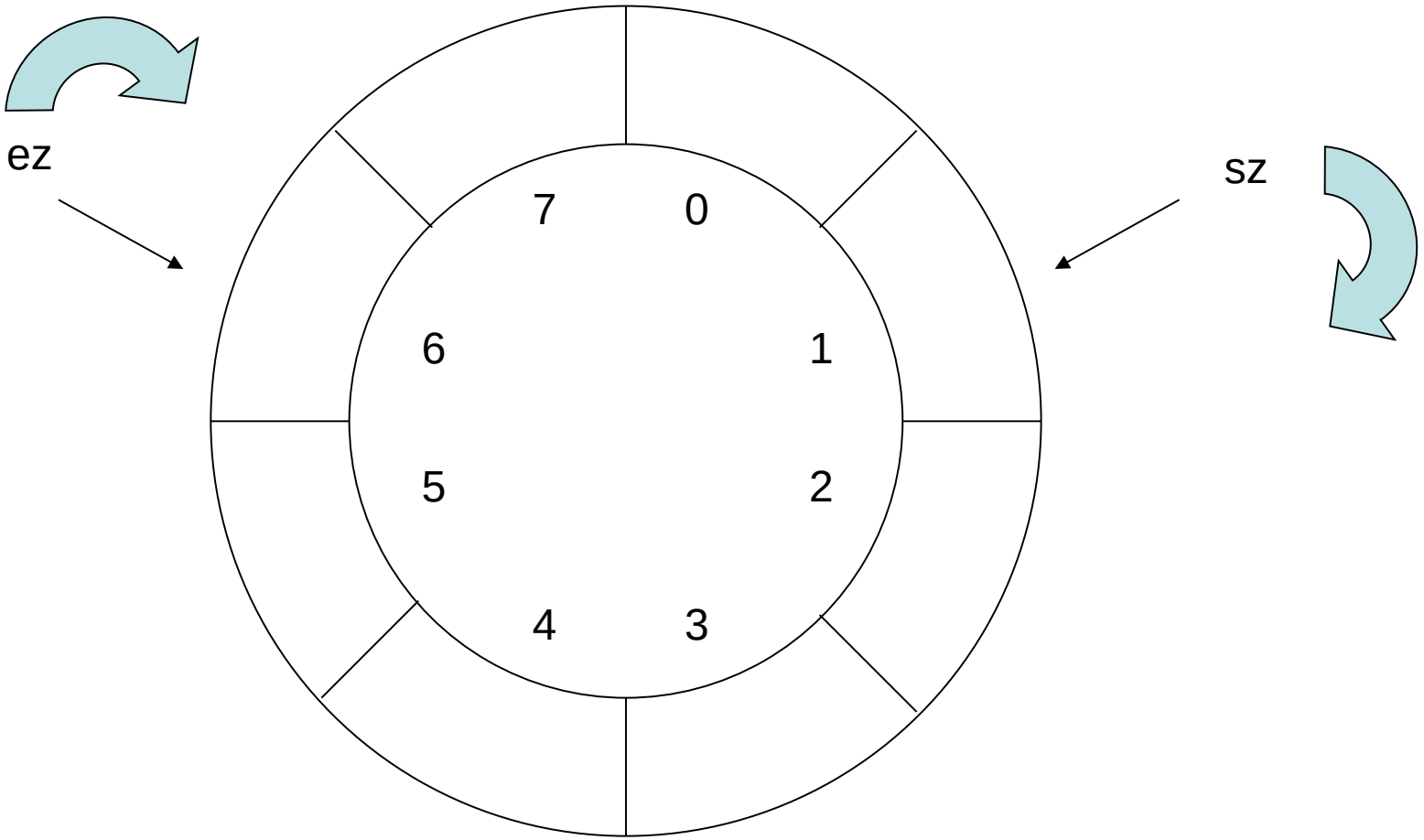
```
template<typename T>
void Schlange<T>::deq() {
  if (empty()) error("leer");
  for (int i = 0; i < ez; i++)
    data[i] = data[i+1];
  ez--;
}
```

`ez` = Anzahl Elemente in Schlange

Insgesamt **`ez`** Datenverschiebungen

Worst case: (**`maxSize`** – 1) mal

Idee: Array zum Kreis machen; zusätzlich Anfang/Start markieren (sz)



Implementierung: (Version 2; mit Ringspeicher)

```
template<typename T>
class Schlange {
public:
  Schlange();
  void enq(T &x);
  void deq();
  T front();
  bool empty();
  bool full();
private:
  static unsigned int const maxSize = 100;
  int ez;            // Endezeiger
  int sz;            // Startzeiger
  T data[maxSize];
  void error(char const *info);
};
```

Implementierung: (Version 2; mit Ringspeicher)

```
template<typename T> Schlange<T>::Schlange() {
  sz = 0;
  ez = -1;
}
template<typename T> T Schlange<T>::front() {
  if (empty()) error("leer");
  return data[sz];
}
template<typename T> bool Schlange<T>::empty() {
  return (ez == -1);
}
template<typename T> bool Schlange<T>::full() {
  if (empty()) return false;
  return ((ez + 1) % maxSize) == sz;
}
```

Implementierung: (Version 2; mit Ringspeicher)

```
template<typename T>
void Schlange<T>::enq(T &x) {
  if (full()) error("full");
  ez = (ez + 1) % maxSize;
  data[ez] = x;
}
```

Laufzeit:

unabhängig von Größe der Schlange

```
template<typename T>
void Schlange<T>::deq() {
  if (empty()) error("leer");
  if (sz == ez) { sz = 0; ez = -1; }
  else sz = (sz + 1) % maxSize;
}
```

Laufzeit:

unabhängig von Größe der Schlange

Unbefriedigend bei der Implementierung:

Maximale festgelegte Größe des Stapels 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**.

⇒ **Dynamischer Speicher!** (Stichwort: **`new`** / **`delete`**)

Lineare Datenstrukturen: Schlange (*engl. queue*)

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

create : erzeugt leere Schlange

enq : hängt Element ans Ende der Schlange

deq : entfernt Kopf der Schlange

front : gibt im Kopf der Schlange gespeichertes Element zurück

empty : prüft, ob Schlange leer ist

→ Implementierung mit statischem Speicher ersetzen durch dynamischen 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!

Vorüberlegungen für ADT Schlange mit dynamischem Speicher:

Wir können bei der Realisierung der Schlange
statt statischem (Array) nun **dynamischen Speicher** verwenden …

Ansatz: **`new int[oldsize+1]`** … bringt uns das weiter?

→ Größe kann zwar zur Laufzeit angegeben werden, ist aber dann fixiert!

Falls maximale Größe erreicht, könnte man

1. größeres Array anlegen
2. Arraywerte ins größere Array **kopieren** und
3. kleineres Array löschen.

} **ineffizient!**