



Wintersemester 2007/08

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

Prof. Dr. Günter Rudolph

Fakultät für Informatik

Lehrstuhl für Algorithm Engineering





Kapitel 8: Elementare Datenstrukturen

Inhalt

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



Wiederholung: ADT Schlange

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 statischen Speicher ersetzen durch dynamischen Speicher



Wiederholung: 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!



Kapitel 8: Elementare Datenstrukturen

Vorüberlegungen für ADT Schlange mit dynamischen Speicher:

Wir können bei der Realisierung der Schlange statt statischen (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

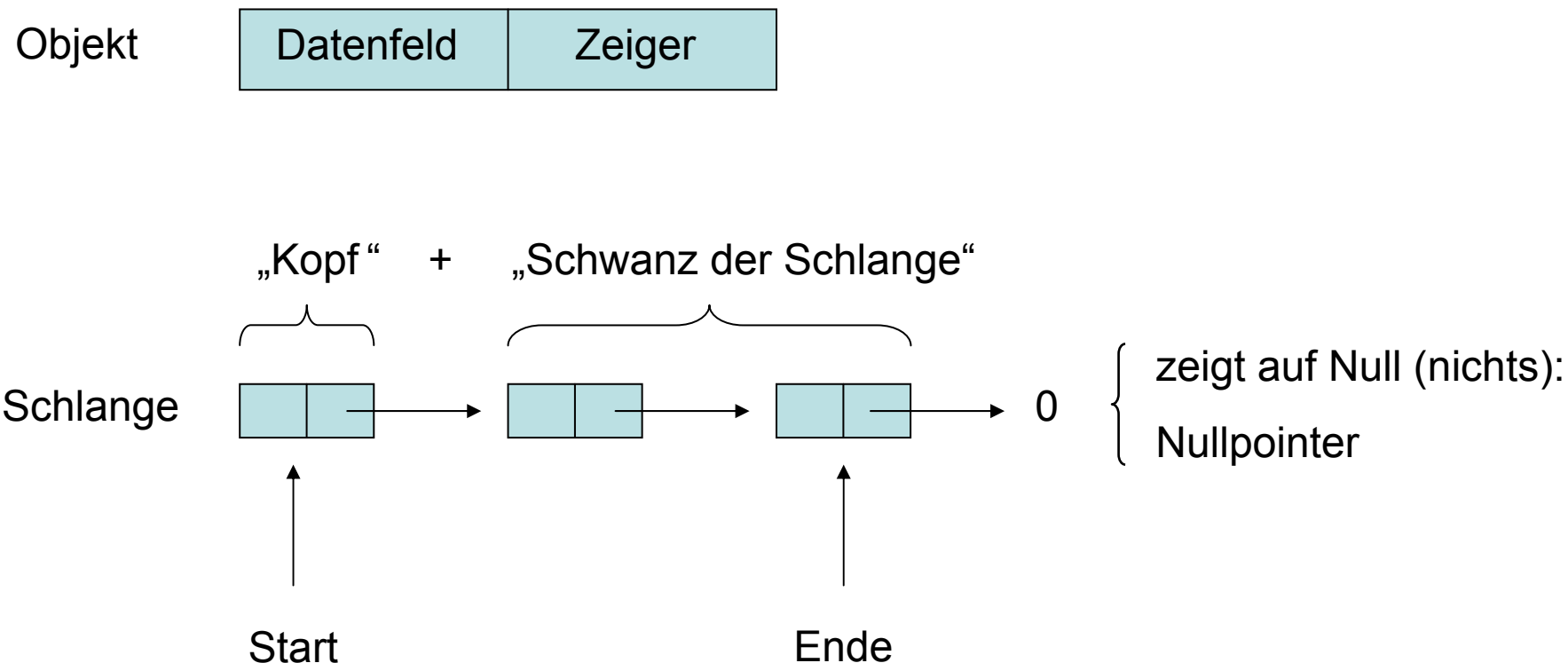
1. größeres Array anlegen
2. Arraywerte ins größere Array **kopieren** und
3. kleineres Array löschen.

} **ineffizient!**



Kapitel 8: Elementare Datenstrukturen

Vorüberlegungen für ADT Schlange mit dynamischen Speicher:





Kapitel 8: Elementare Datenstrukturen

Implementierung: (Version 4)

```
struct Objekt {
    T data;           // Datenfeld
    Objekt *tail;    // Zeiger auf Schwanz der Schlange
};
struct Schlange {
    Objekt *sp;      // Start
    Objekt *ep;      // Ende
};
```

```
Schlange *create() {
    Schlange *s = new Schlange; ← Speicher für Schlange allokieren
    s->ep = 0;                 ← Initialisierung!
    return s;                  ← Rückgabe des Zeigers auf Schlange
}
```

```
bool empty(Schlange *s) {
    return s->ep == 0;        ← true falls s->ep == 0, sonst false
}
```



Kapitel 8: Elementare Datenstrukturen

Implementierung: (Version 4)

```
struct Objekt {
    T data;           // Datenfeld
    Objekt *tail;   // Zeiger auf Schwanz der Schlange
};
struct Schlange {
    Objekt *sp;     // Start
    Objekt *ep;     // Ende
};
```

```
T front(Schlange *s) {
    if (!empty(s))
        return (s->sp)->data;
    error("leer");
}
```

```
void clear(Schlange *s) {
    while (!empty(s)) deq(s);
}
```

Was bedeutet `s->sp->data` ?
identisch zu `(*(*s).sp).data`

vorderstes Element entfernen
bis Schlange leer



Kapitel 8: Elementare Datenstrukturen

Implementierung: (Version 4)

```
struct Objekt {
    T data;           // Datenfeld
    Objekt *tail;   // Zeiger auf Schwanz der Schlange
};
struct Schlange {
    Objekt *sp;     // Start
    Objekt *ep;     // Ende
};
```

```
void enq(Schlange *s, T x) {
    Objekt *obj = new Objekt;
    obj->data = x;
    obj->tail = 0;
    if (empty(s)) s->sp = obj;
    else s->ep->tail = obj;
    s->ep = obj;
}
```

neues Objekt anlegen (dyn. Speicher)
neues Objekt initialisieren
neues Objekt vorne, falls Schlange leer
sonst hinten anhängen
Aktualisierung des Endezeigers



Kapitel 8: Elementare Datenstrukturen

Implementierung: (Version 4)

```
struct Objekt {
    T data;           // Datenfeld
    Objekt *tail;    // Zeiger auf Schwanz der Schlange
};
struct Schlange {
    Objekt *sp;      // Start
    Objekt *ep;      // Ende
};
```

```
void deq(Schlange *s) {
    if (empty(s)) error("leer");
    Objekt *obj = s->sp;
    s->sp = s->sp->tail;
    if (s->sp == 0) s->ep = 0;
    delete obj;
}
```

Zeiger auf zu löschendes Objekt retten
Startzeiger auf 2. Element setzen
falls kein 2. Element, dann Schlange leer
ehemals 1. Element löschen



Kapitel 8: Elementare Datenstrukturen

```
int main() {
    Schlange *s = create();
    if (empty(s)) cout << "Schlange leer" << endl;
    for (int i = 0; i < 10; i++) enq(s, i);
    if (!empty(s)) cout << "Schlange nicht mehr leer" << endl;
    cout << "vorderstes Element: " << front(s) << endl;
    while (!empty(s)) {
        cout << front(s) << " ";
        deq(s);
    }
    cout << endl;
    if (empty(s)) cout << "Schlange jetzt leer" << endl;

    for (i = 0; i < 100; i++) enq(s,i);
    if (!empty(s)) cout << "Schlange nicht mehr leer" << endl;
    clear(s);
    if (empty(s)) cout << "Schlange wieder leer" << endl;
}
```

Testprogramm!



Kapitel 8: Elementare Datenstrukturen

```
e:\visual studio\ini\ini\queue\debug\queue.exe  
Schlange leer  
Schlange nicht mehr leer  
vorderstes Element: 0  
0 1 2 3 4 5 6 7 8 9  
Schlange jetzt leer  
Schlange nicht mehr leer  
Schlange wieder leer  
-
```



Kapitel 8: Elementare Datenstrukturen

ADT Liste (1. Version)

```
struct Liste {  
    T data;  
    Liste *next;  
};
```

Liste wird nur durch einen Zeiger auf ihren Listenkopf repräsentiert

Operationen:

create	:	→ Liste	
empty	:	Liste → bool	
append	:	T x Liste → Liste	hängt am Ende an
prepend	:	T x Liste → Liste	vor Kopf einfügen
clear	:	→ Liste	
is_elem	:	T x Liste → bool	ist Element enthalten?



Kapitel 8: Elementare Datenstrukturen

ADT Liste (1. Version)

```
struct Liste {  
    T data;  
    Liste *next;  
};
```

Liste wird nur durch einen Zeiger auf ihren Listenkopf repräsentiert

```
Liste *create() {  
    return 0;  
}
```

Laufzeit:
unabhängig von Listenlänge

```
bool empty(Liste *liste) {  
    return liste == 0;  
}
```

Laufzeit:
unabhängig von Listenlänge

```
void clear(Liste *liste) {  
    if (empty(liste)) return;  
    clear(liste->next);  
    delete liste;  
}
```

rekursives Löschen von „hinten“ nach „vorne“

Laufzeit:
proportional zur Listenlänge



Kapitel 8: Elementare Datenstrukturen

ADT Liste (1. Version)

```
struct Liste {  
    T data;  
    Liste *next;  
};
```

```
bool is_elem(T x, Liste *liste) {  
    if (liste == 0) return false;  
    if (liste->data == x) return true;  
    return is_elem(x, liste->next);  
}
```

```
Liste *prepend(T x, Liste *liste) {  
    Liste *L = new Liste;  
    L->data = x;  
    L->next = liste;  
    return L;  
}
```

rekursiver Durchlauf von
„vorne“ nach „hinten“

Laufzeit:
proportional zur Listenlänge

Laufzeit:
unabhängig von Listenlänge



Kapitel 8: Elementare Datenstrukturen

ADT Liste (1. Version)

```
struct Liste {  
    T data;  
    Liste *next;  
};
```

```
Liste *append(T x, Liste *liste) {  
    Liste *tmp = liste;  
    Liste *L = new Liste;  
    L->data = x;  
    L->next = 0;  
    if (liste == 0) return L;  
    while (liste->next != 0)  
        liste = liste->next;  
    liste->next = L;  
    return tmp;  
}
```

Zeiger auf Listenkopf retten

} iterativer Durchlauf von
„vorne“ nach „hinten“

Laufzeit:
proportional zur Listenlänge



Kapitel 8: Elementare Datenstrukturen

ADT Liste (1. Version)

Zusammenfassung:

1. Laufzeit von clear proportional zur Listenlänge
 - kann nicht verbessert werden, weil ja jedes Element gelöscht werden muss
 - unproblematisch, weil nur selten aufgerufen
2. Laufzeit von is_elem proportional zur Listenlänge
 - kann bei dieser **Datenstruktur** nicht verbessert werden
 - später verbessert durch ADT BinärBaum
3. Laufzeit von append proportional zur Listenlänge
 - kann durch Veränderung der **Implementierung** verbessert werden
 - zusätzlicher Zeiger auf das Ende der Liste



Kapitel 8: Elementare Datenstrukturen

ADT Liste (2. Version)

```
struct Liste {
    Element *head;
    Element *foot;
};
struct Element {
    T data;
    Element *next;
};
```

```
Liste *create() {
    Liste *L = new Liste;
    L->head = L->foot = 0;
    return L;
}
```

```
bool empty(Liste *liste) {
    return liste->foot == 0;
}
```

Liste besteht aus 2 Zeigern:
Zeiger auf Listenkopf (Anfang)
Zeiger auf Listenfuß (Ende)

Nutzdaten
Zeiger auf nächstes Element

Laufzeit:
unabhängig von Listenlänge

Laufzeit:
unabhängig von Listenlänge



Kapitel 8: Elementare Datenstrukturen

ADT Liste (2. Version)

```
struct Liste {
    Element *head;
    Element *foot;
};
struct Element {
    T data;
    Element *next;
};
```

```
bool is_elem(T x, Liste *liste) {
    if (empty(liste)) return false;
    Element *elem = liste->head;
    while (elem != 0) {
        if (elem->data == x) return true;
        elem = elem->next;
    }
    return false;
}
```

iterativer Durchlauf von
„vorne“ nach „hinten“

Laufzeit:

proportional zur Listenlänge

→ keine Verbesserung (OK)



Kapitel 8: Elementare Datenstrukturen

ADT Liste (2. Version)

```
struct Liste {
    Element *head;
    Element *foot;
};
struct Element {
    T data;
    Element *next;
};
```

```
void clear(Liste *liste) {
    if (empty(liste)) return;
    Element *elem = liste->head;
    liste->head = elem->next;
    clear(liste);
    delete elem;
}
```

rekursives Löschen von
„hinten“ nach „vorne“

Laufzeit:
proportional zur Listenlänge

→ keine Verbesserung (OK)



Kapitel 8: Elementare Datenstrukturen

ADT Liste (2. Version)

```
struct Liste {
    Element *head;
    Element *foot;
};
struct Element {
    T data;
    Element *next;
};
```

```
Liste *prepend(T x, Liste *liste) {
    Element *elem = new Element;
    elem->data = x;
    elem->next = liste->head;
    if (empty(liste)) liste->foot = elem;
    liste->head = elem;
    return liste;
}
```

Laufzeit:
unabhängig von Listenlänge



Kapitel 8: Elementare Datenstrukturen

ADT Liste (2. Version)

```
struct Liste {
    Element *head;
    Element *foot;
};
struct Element {
    T data;
    Element *next;
};
```

```
Liste *append(T x, Liste *liste) {
    Element *elem = new Element;
    elem->data = x;
    elem->next = 0;
    if (empty(liste)) liste->head = elem;
    else liste->foot->next = elem;
    liste->foot = elem;
}
```

Laufzeit:
unabhängig von Listenlänge

→ Verbesserung!



Kapitel 8: Elementare Datenstrukturen

ADT Liste (2. Version)

Zusammenfassung:

1. Laufzeit von clear proportional zur Listenlänge
 - kann nicht verbessert werden, weil ja jedes Element gelöscht werden muss
 - unproblematisch, weil nur selten aufgerufen
2. Laufzeit von is_elem proportional zur Listenlänge
 - kann bei dieser **Datenstruktur** nicht verbessert werden
 - verbessern wir gleich durch ADT BinärBaum
3. Laufzeit von append unabhängig von Listenlänge
 - war proportional zur Listenlänge in 1. Version
 - Verbesserung erzielt durch Veränderung der **Implementierung**



Kapitel 8: Elementare Datenstrukturen

ADT Binäre Bäume

Vorbemerkungen:

Zahlenfolge (z. B. 17, 4, 36, 2, 8, 19, 40, 6, 7, 37) soll gespeichert werden, um später darin suchen zu können

Man könnte sich eine Menge A vorstellen mit Anfrage: Ist $40 \in A$?

Mögliche Lösung: Zahlen in einer Liste speichern und nach 40 suchen ...

... aber: **nicht effizient**,
weil im schlechtesten Fall alle Elemente überprüft werden müssen!

Bessere Lösungen?



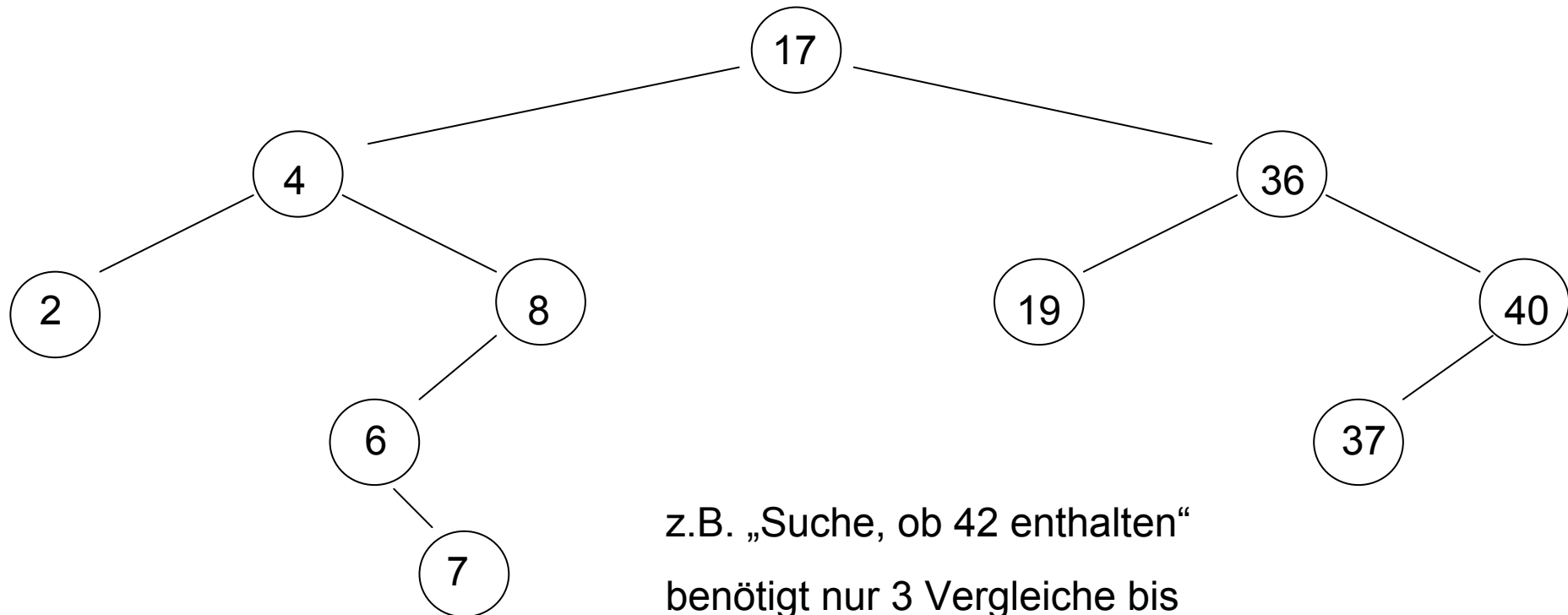
Kapitel 8: Elementare Datenstrukturen

ADT Binäre Bäume

Beispiel:

Zahlenfolge 17, 4, 36, 2, 8, 19, 40, 6, 7, 37

kleiner : nach links
größer : nach rechts



z.B. „Suche, ob 42 enthalten“

benötigt nur 3 Vergleiche bis
zur Entscheidung **false**