Teil I
Modellierung und Analyse

- Kap. 2 Modellierung und Simulation diskreter Systeme
  - Einführung in die klassische ereignisdiskrete Simulation
- Kap. 3 Analytische und numerische Analyse diskreter Systeme
  - Ein kurzer Überblick über Warteschlangentheorie
- Kap. 4 Modellierung und Analyse kontinuierlicher Systeme
  - Ein sehr kurzer Einblick in die kontinuierliche Simulation

2. Modellierung und Analyse diskreter Systeme

Gliederung

2.1 Konzepte ereignisdiskreter Simulation

2.2 Spezifikation von Simulatoren

2.3 Generierung und Bewertung von Zufallszahlen

2.4 Modellierung von Eingabedaten

2.5 Auswertung von Simulationsläufen

2.6 Simulationssoftware

2.7 Möglichkeiten und Grenzen der Simulation

2.8 Validierung von Modellen

Literatur:

Primär verwendet

- A. M. Law, W. D. Kelton. Simulation Modeling and Analysis. McGraw Hill 2000.
  Kapitel 1, 3-1.5, 3, 6.1-6.5, 7.1, 7.2, 8.1, 8.2, 9.1-9.5 oder
- Banks et al. Discrete-Event System Simulation. Prentice Hall 200.
  Kap. 2-4, 7, 8, 9.1-9.4, 11.

Auch zu empfehlen (für Teilbereiche):

- W. D. Kelton, R. P. Sadowski, D. T. Sturrock. Simulation with Arena. Mc Graw Hill 2004.
- U.v.a.

In diesem Kapitel werden Systeme untersucht,

- die zu diskreten Zeitpunkten ihren Zustand ändern
- bei denen zwischen Zustandsänderungen eine durch eine Zufallsvariable festgesetzte Zeit vergeht

Simulation dieser Systeme bedeutet Nachbilden und Beobachten des dynamischen Ablaufs

- programmtechnische Darstellung des Systemzustands
- zeitgerechte Ausführung der Ereignisse zur Änderung des Systemzustands
- Auswertung der Beobachtungen zur Ermittlung von Resultatgrößen

Ziele:

- Erkennen des grundsätzlichen Ablaufs diskreter Simulationen
- Kennen lernen unterschiedlicher Realisierungskonzepte für ereignisdiskrete Simulatoren
- Anforderungen an Programmiersprachen zur Erstellung von Simulatoren formulieren können
- Übersicht über einige gängige Simulationswerkzeuge erhalten
- Wissen über die Realisierung von Zufallszahlen erlangen
- Methoden zur Darstellung gemessener Verteilungen durch Zufallszahlen kennen lernen
- Resultatschätzer und Konfidenzintervalle aus Simulationsdaten gewinnen können

2.1 Konzepte ereignisdiskreter Simulation

Simulieren ≈ Imitieren des Realsystems, d.h. (für jeden Zeitpunkt):

- Systemzustand kennen
- (nächste) Zustandsänderung kennen

Zustand im Simulator (imperative Sicht):

- Werte aller Programmvariablen

Nächste Zustandsänderung ⇒ Programmzeiger

Damit natürlich und naheliegend:

- Zustandsraum Realsystem → Datenstruktur im Simulator

In Simulationsterminologie:

Menge der Zustandsvariablen bildet die **statische Struktur** des Modells

Einfacher Abbildungsvorgang:

- Gedankenmodell strukturiert Objekte mit Attributen
- Erfassen von Attributgruppen durch Zustandsvariablengruppen

⇒ minimale Simulator-Datenstruktur

Zustandsänderungen im Simulator:

- Werteänderungen von Zustandsvariablen
- d.h. per Wertzuweisung
- d.h. per Ausführung von Simulator-Code

$\Rightarrow$ zu jedem **Ereigniszeitpunkt** muss ein Stück Simulator-Code abgearbeitet werden

Im Simulator müssen

- **Ereignistypen** festgelegt werden
  (wobei jeder Ereignistyp durch eine Menge von Zustandsänderungen charakterisiert ist)
- Code-Segmente für jeden Ereignistyp programmiert werden (**Ereignisroutinen**)
- die **zeitgerechte Ausführung** der Ereignisroutinen organisiert werden

Zum Zeitablauf:

- Ablauf der Ereignisroutinen im Modell zeitlos
  - Zustandsänderungen sind atomar, finden in Nullzeit statt
  - Abarbeitung der zugehörigen Routine dauert in der Realität Zeit
- zwischen zwei Ereignissen vergeht in der Realität Zeit
  - Erstes Ereignis tritt zum Zeitpunkt t, nächstes zum Zeitpunkt $t+\Delta$ ($\Delta \geq 0$) ein (Δ in Abhängigkeit vom Zustand-Ereignis)
  - Übergang von einer Ereignisroutine zur nächsten erfordert im Rechner (fast) keine Zeit

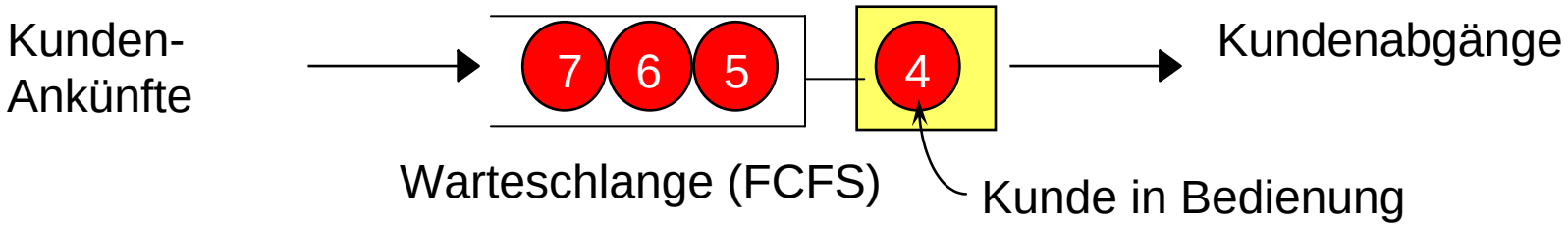
Damit besteht ein Simulator (sehr abstrakt):

- **Abbildung des Zustands → statische Struktur**
- **Abbildung der Zustandsänderungen → dynamische Struktur**

Erläuterung des prinzipiellen Vorgehens der ereignisdiskreten Simulation an einem einfachen Beispiel:

- Ein Bediener, der immer dann arbeitet, wenn Kunden warten
  - der Bediener kann genau einen Kunden zu einem Zeitpunkt bedienen
- Kunden kommen im System an und werden vom Bediener bedient
  - falls der Bediener belegt ist, warten die Kunden in einem Warteraum potenziell unendlicher Kapazität
  - Kunden im Warteraum werden nach First Come First Served (FCFS) abgearbeitet
  - nach Ende der Bedienung verlassen Kunden das System
- Ankunftszeitpunkte und Bedienzeiten resultieren aus Messungen oder werden aus vorgegebenen Verteilungsfunktionen generiert (zweite Variante wird üblicherweise in der Simulation verwendet)

Graphische Darstellung des Systems:



Statische Struktur: (es gibt unterschiedliche Möglichkeiten

- Anzahl Kunden im Warteraum Q (integer)
- Bediener arbeitet B (Boolean, hier mit Werten 0 und 1)
- Zusätzlich vorhanden Variable t zur Beschreibung der aktuellen Simulationszeit
- Annahme: Zustand zu Beginn der Simulation t=0 bekannt,
  Simulation bis zum Zeitpunkt T (> 0)
- B(t) und Q(t) Werte der Variablen zum Zeitpunkt t
- B(0)=false und Q(0)=0

Dynamische Struktur: Zwei Ereignisse Ankunft und Bedienende

- Bei einer Ankunft:
  - Wird B auf 1 gesetzt, falls B=0
    (und startet damit implizit die Bedienung)
  - Wird Q auf Q+1 falls B=1
- Bei einem Bedienende
  - Wird B auf 0 gesetzt, falls Q=0
  - Wird Q auf Q-1 gesetzt, falls Q größer 0 ist

Zusätzliche Aufgaben in der Simulation:

- Zeitgerechte Ausführung der Ereignisse (dazu später mehr)
- Beobachtung des Systemverhaltens
  z.B. durch Definition und Auswertung von Resultatgrößen

Definition von Resultatgrößen:

- Anzahl der Kunden, die ihre Bedienung im Intervall [0,T] beendet haben P
- Mittlere Population in der Warteschlange $\int_0^T Q(t)dt \Big/ T$
- Maximal Population in der Warteschlange $\max_{0 \le t \le T} Q(t)$
- Auslastung des Bedieners $\int_0^T B(t)dt \Big/ T$
- Mittlere Verweilzeit eines Kunden $\sum_{p=1}^{P} D_p \Big/ P$ wobei D_p Verweilzeit des p-ten Kunden ist
- Durchsatz P/T
- u.v.a.

Resultatgrößen sind während der Simulation neu zu berechnen/auszuwerten

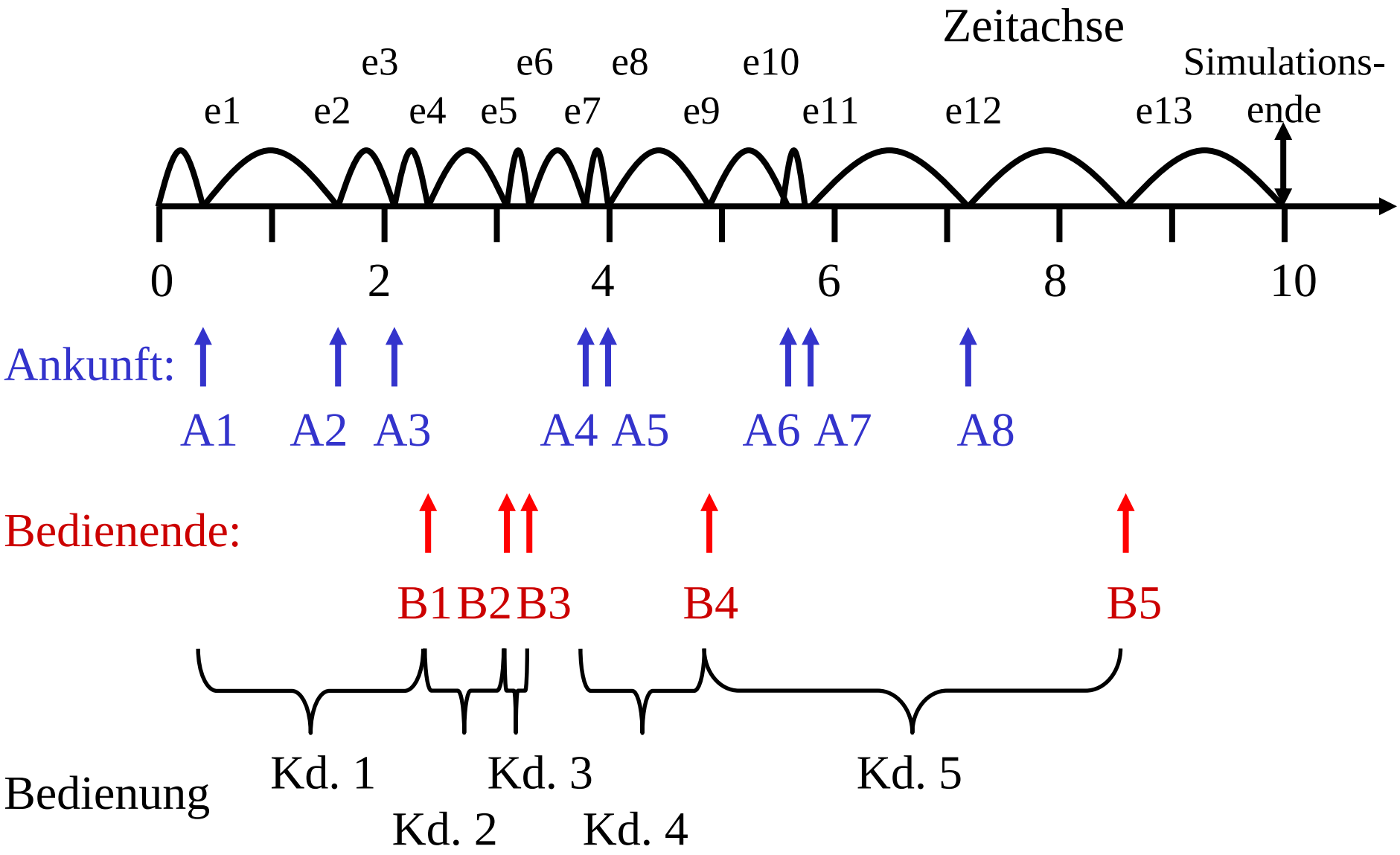
Ablauf (Zeitangaben in Minuten, Simulationsdauer 10 Min.):

Kunde	Ankunftszeit	Zwischenankunftszeit	Bedienzeit
1	0.4	0.4	2.0
2	1.6	1.2	0.7
3	2.1	0.5	0.2
4	3.8	1.7	1.1
5	4.0	0.2	3.7
6	5.6	1.6	1.5
7	5.8	0.2	0.4
8	7.2	1.4	1.1

Es gilt für Kunden i:

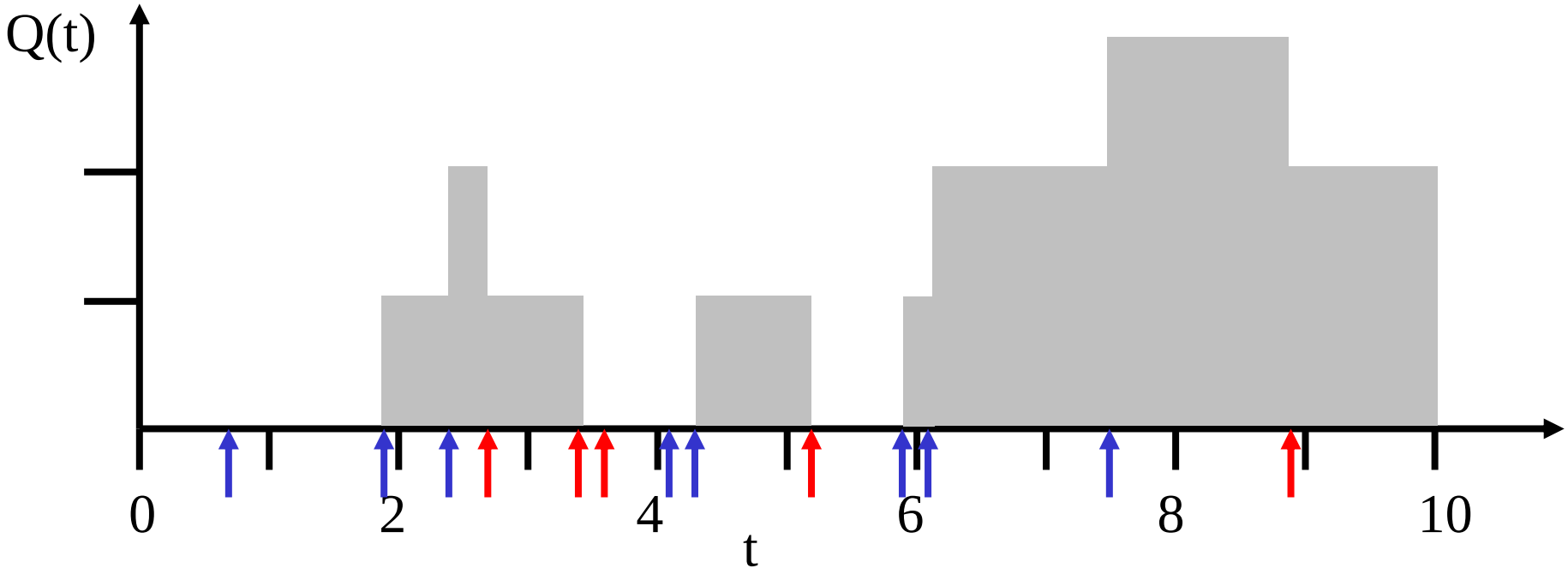
- Bedienende i = Bedienanfang i + Bedienzeit i
- Bedienanfang i = max(Ankunftszeit i, Bedienende i-1)

Ziel: Zeitgerechte Ausführung der Ereignisse!


Zeitachse
e1
e2
e3
e4
e5
e6
e7
e8
e9
e10
e11
e12
e13
Simulations-
ende
0
2
4
6
8
10
Ankunft:
A1
A2
A3
A4
A5
A6
A7
A8
Bedienende:
B1
B2
B3
B4
B5
Bedienung
Kd. 1
Kd. 2
Kd. 3
Kd. 4
Kd. 5

Bestimmung von Resultaten während der Simulation

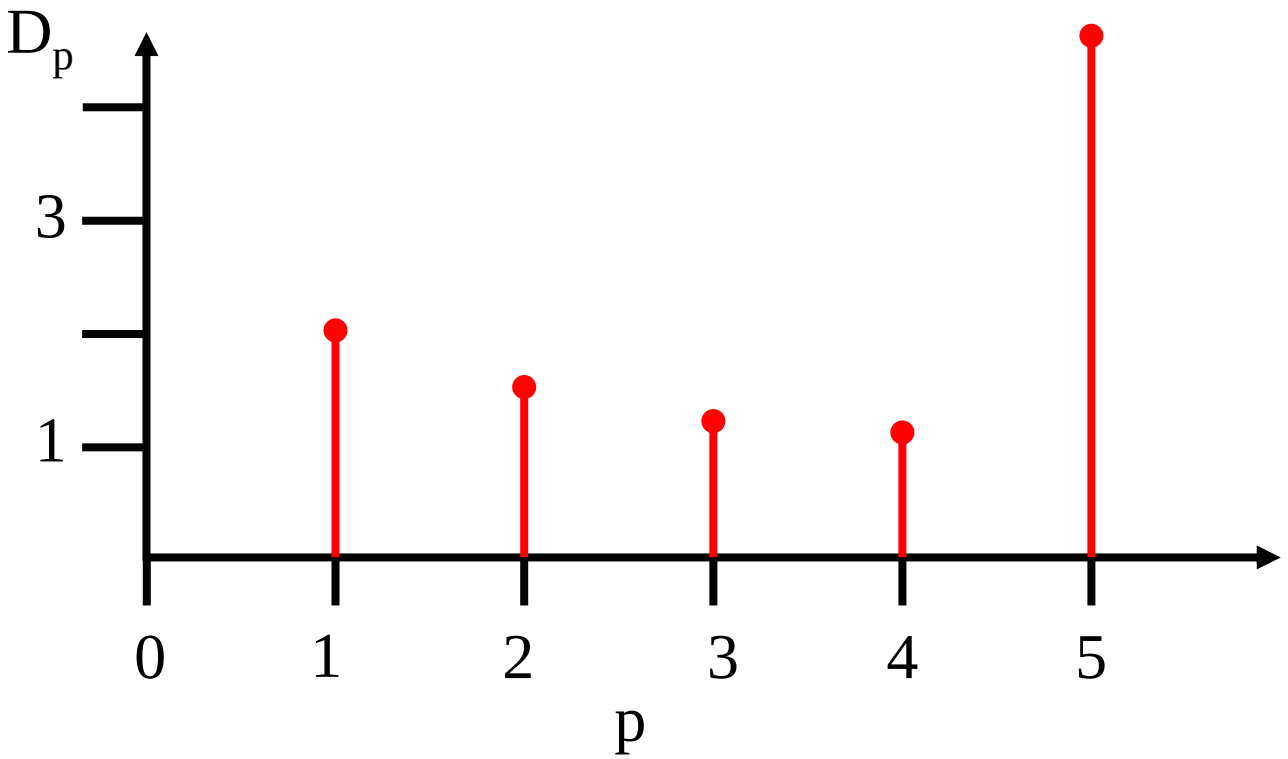
Beispiel: Q(t)



Die mittlere Population in der Warteschlange ergibt sich aus der graue Fläche im Verhältnis zur Länge der Beobachtung

Zur Ermittlung während der Simulation wird bei jedem Ereignis der Flächeninhalt seit dem letzten Ereignis ausgewertet!

Beispiel: D_p Verweilzeiten der Kunden



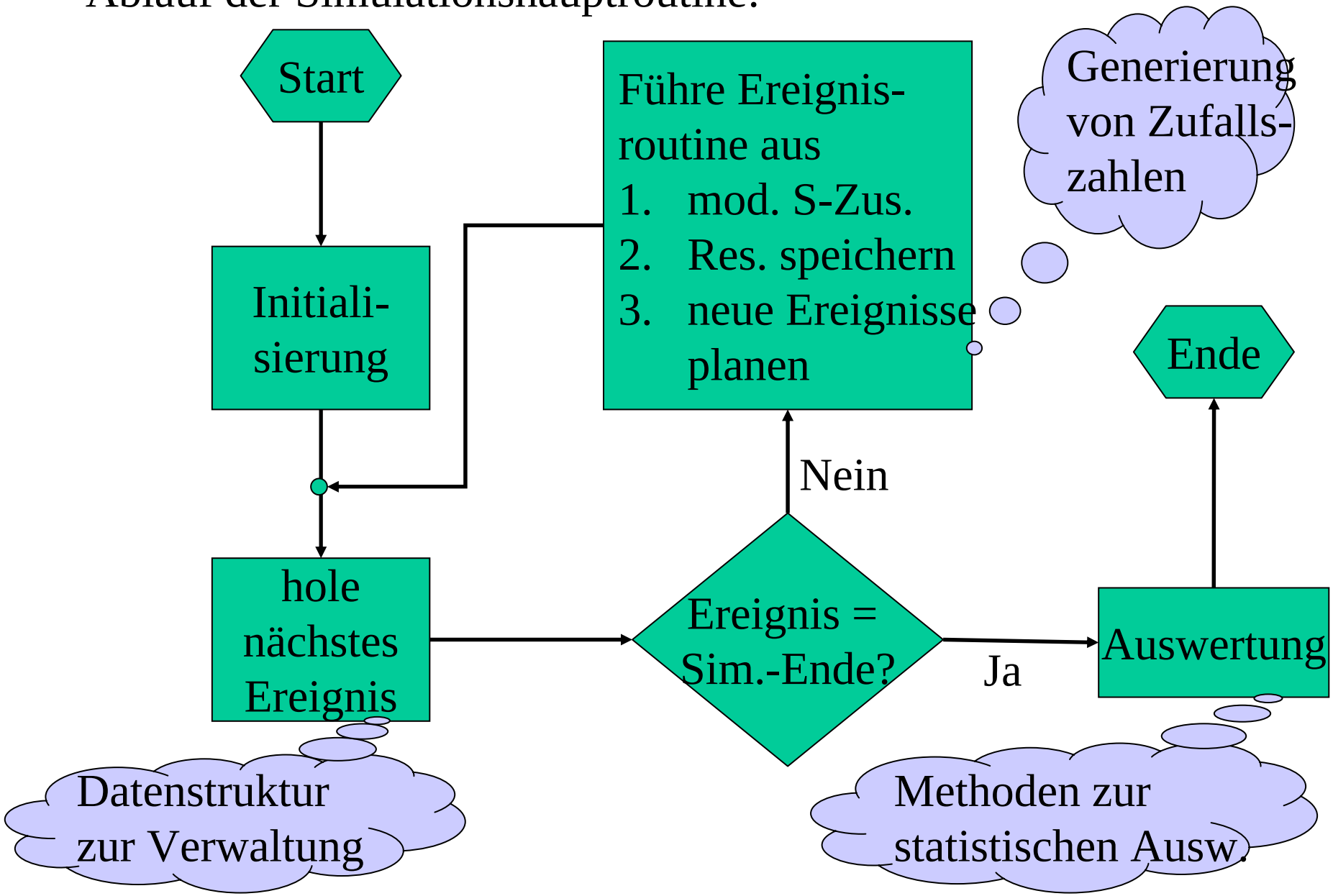
Die mittlere Verweilzeit ergibt sich aus der Summe der Verweilzeiten dividiert durch die Anzahl beobachteter Kunden

Zur Ermittlung während der Simulation wird bei jedem Kunden, der das System verlässt, die Verweilzeit ermittelt!

Einige Beobachtungen:

- Simulationsablauf durch Sprünge von einem Ereignis zum nächsten
- Simulationszeit t ergibt sich aus der Zeit des nächsten Ereignisses
- Ereigniszeitpunkte ergeben sich z. T. während der Simulation
  z.B. Bedienende bestimmt den nächsten Bedienanfang
  - Im Beispiel waren alle Zeitpunkte vorgegeben
    (trace-getriebene Simulation)
  - alternativ werden Zeiten aus vorgegebenen Verteilungen während der Simulation generiert (stochastische Simulation)
- Zu jedem Ereigniszeitpunkt wird die zugehörige Ereignisroutine ausgeführt, um
  1. den Systemzustand zu modifizieren
  2. Resultatwerte speichern
  3. **zukünftige Ereignisse einzuplanen**

Ablauf der Simulationshauptroutine:



Datenstruktur zur Verwaltung von Ereignissen

Anforderungen:

- Jedes Ereignis i ist beschrieben durch den Ereigniszeitpunkt t_i und den Ereignistyp tp_i
- Ereignisse müssen nach Zeit ihres Eintretens t_i geordnet sein
- Zugriffsfunktionen
  - Auslesen des ersten Elements
  - Einfügen eines neuen Elements
  - Löschen eines eingefügten Elements (im einfachen Beispiel nicht verwendet!)

➢ Verwendete Datenstruktur **Ereignisliste**

Ereignisliste:

t_i	t_1	t_2	t_3	t_4	t_5	...
tp_i	tp_1	tp_2	tp_3	tp_4	tp_5	...

- Inhalt der Ereignisliste umfasst die bisher geplante Modellzukunft
- Simulationsende kann als spezielles Ereignis zu Beginn der Simulation eingefügt werden, falls die Abbruchzeit feststeht (es gibt andere Abbruchbedingungen!)
- Ereignisse können nur zu Zeitpunkten $t \geq t_1$ eingefügt werden (Simulationszeit läuft nur in eine Richtung!)
- Für gleichzeitige Ereignisse (d.h. $t_i = t_j$) existiert keine Reihenfolge
  (dies kann zu nicht-deterministischem Verhalten führen!)

Für unser Beispiel:

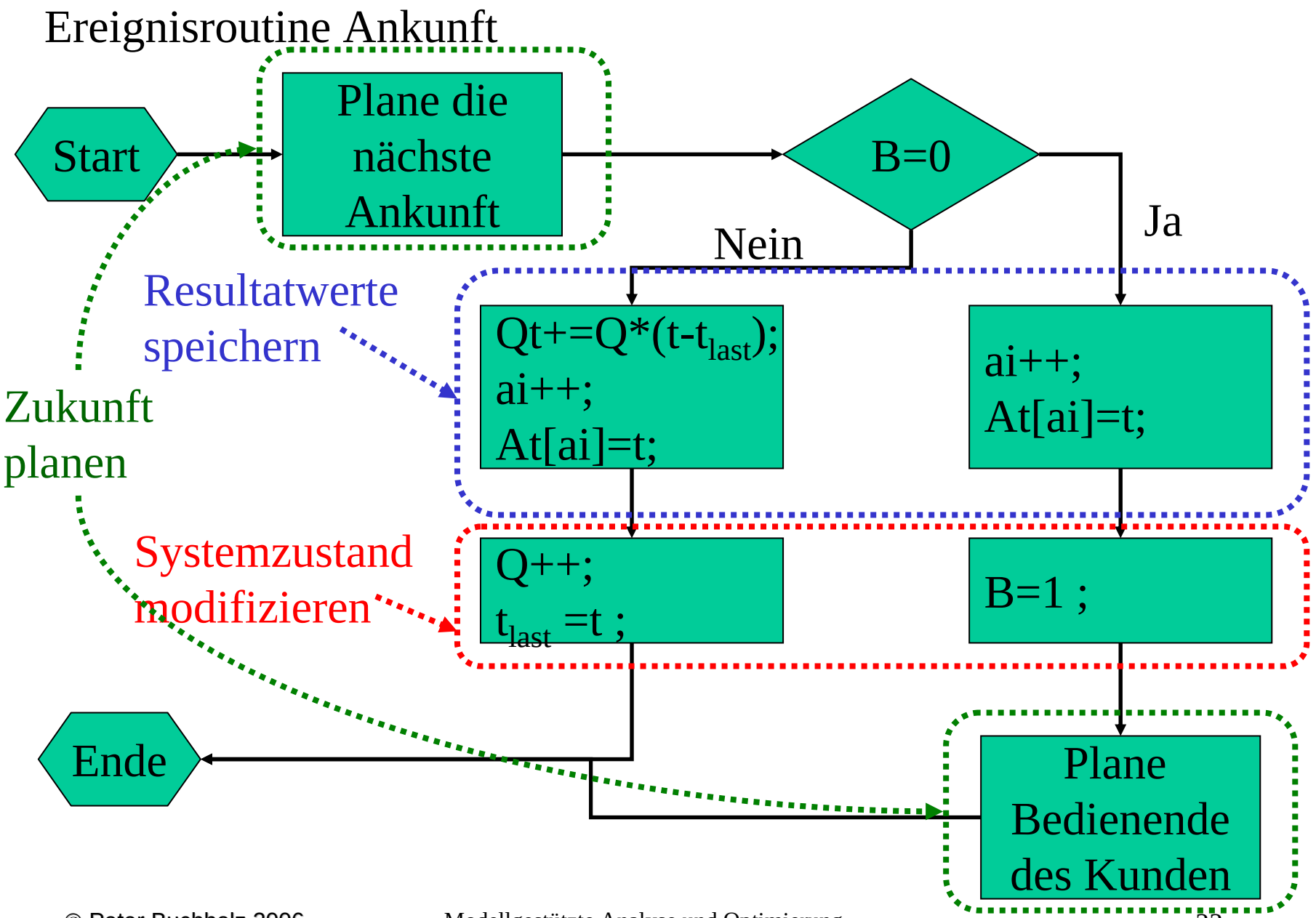
- t aktuelle Simulationszeit

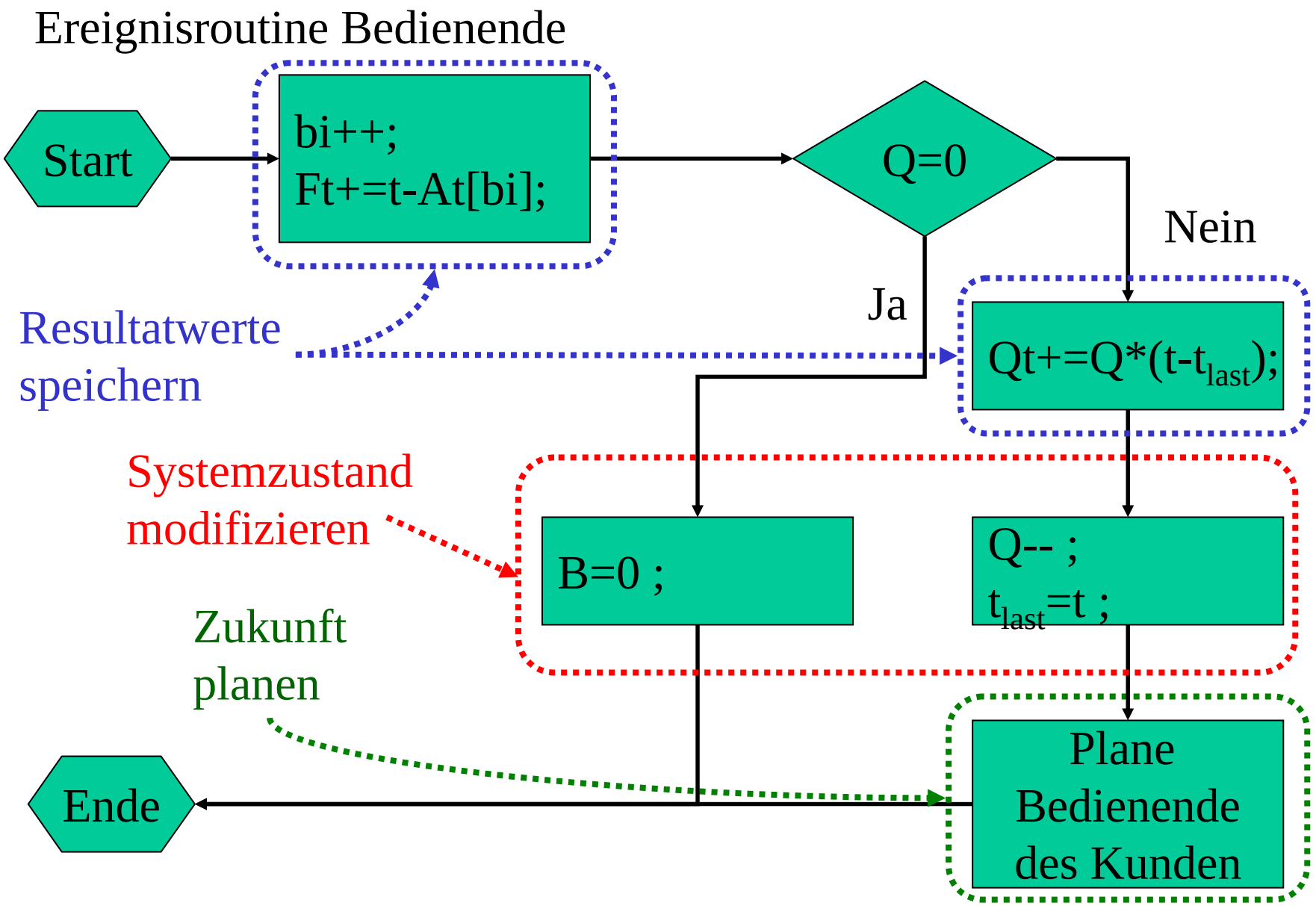
Zustandvariablen

- Q Anzahl Aufträge in der Warteschlange, initialisiert mit 0
- B Status des Bedieners, initialisiert mit 0

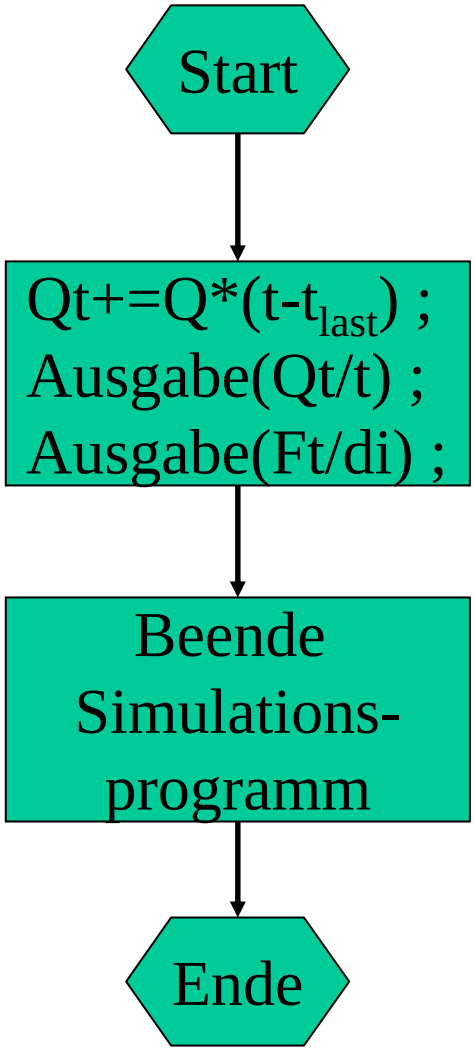
Variablen zur Auswertung der Resultate
(alle Variablen werden mit 0 initialisiert)

- Qt kumulierter Wert der Auftragszahl in der Warteschlange
- t_{last} Zeitpunkt der letzten Änderung von Q
- ai Nummer des letzten Auftrags, der angekommen ist
- bi Nummer des letzten Auftrags, der das System verlassen hat
- At[i] Ankunftszeit des i-ten Auftrags
- Ft kumulierter Wert der Verweilzeiten


Ereignisroutine Ankunft
Start
Plane die nächste Ankunft
B=0
Nein
Ja
Resultatwerte speichern
Zukunft planen
Systemzustand modifizieren
Qt+=Q*(t-t_last);
ai++;
At[ai]=t;
ai++;
At[ai]=t;
Q++;
t_last =t ;
B=1 ;
Ende
Plane Bedienende des Kunden


Ereignisroutine Bedienende
Start
bi++;
Ft+=t-At[bi];
Q=0
Nein
Ja
Resultatwerte speichern
Qt+=Q*(t-t_last);
Systemzustand modifizieren
B=0 ;
Q-- ;
t_last=t ;
Zukunft planen
Plane Bedienende des Kunden
Ende

Simulationsende



- Falls nach T Zeiteinheiten abgebrochen werden soll, wird Ereignis *Simulationsende* zu Beginn der Simulation für den Zeitpunkt T eingeplant!
- Falls bzgl. einer zu erfüllenden Bedingung abgebrochen werden soll, wird sobald die Bedingung erfüllt ist, ein Ereignis *Simulationsende* sofort (d.h. für Zeit t) eingeplant

Zum Start der Simulation

- Initialisierung der Variablen
- Planen der ersten Auftragsankunft

Simulation liefert:

- Mittlere Population in der Warteschlange im Intervall [0,T]
- Mittlere Durchlaufzeit fertig gewordener Aufträge im Intervall [0,T]

Bei stochastischen Simulationen jeweils für einen möglichen Ablauf!

➢ Um Aussagen über alle möglichen Abläufe zu erlangen, sind Methoden der Stochastik notwendig (dazu später mehr)

Ablauf des Beispiels

Ereigniscodierung A Ankunft, B Bedienende, E Simulationsende

Im Beispiel hinzugefügt eine weitere Ankunft zum Zeitpunkt 10.2

Start der Simulation

Zustandsvariablen	Simulatorzustand
Q=0 B=0 ai=0 bi=0 t_{last}=0.0	t=0.0 EL: 0.4 A; 10 E
Resultatvariablen	
Qt=0.0 Ft=0.0 At=()	

Erste Ankunft

Zustandsvariablen	Simulatorzustand
Q=0 B=1 ai=1 bi=0 t_{last}=0.0	t=0.4 EL: 1.6 A; 2.4 B; 10 E
Resultatvariablen	
Qt=0.0 Ft=0.0 At=(0.4)	

Zweite Ankunft

Zustandsvariablen	Simulatorzustand
Q=1 B=1 ai=2 bi=0 t_{last}=1.6	t=1.6 EL: 2.1 A; 2.4 B; 10 E
Resultatvariablen	
Qt=0.0 Ft=0.0 At=(0.4,1.6)	

Dritte Ankunft

Zustandsvariablen	Simulatorzustand
Q=2 B=1 ai=3 bi=0 t_{last}=2.1	t=2.1 EL: 2.4 B; 3.8 A; 10 E
Resultatvariablen	
Qt=0.5 Ft=0.0 At=(0.4,1.6,2.1)	

Erstes Bedienende

Zustandsvariablen	Simulatorzustand
Q=1 B=1 ai=3 bi=1 t_{last}=2.4	t=2.4 EL: 3.1 B; 3.8 A; 10 E
Resultatvariablen	
Qt=1.1 Ft=2.0 At=(0.4,1.6,2.1)	

Zweites Bedienende

Zustandsvariablen	Simulatorzustand
Q=0 B=1 ai=3 bi=2 t_{last}=3.1	t=3.1 EL: 3.3 B; 3.8 A; 10 E
Resultatvariablen	
Qt=1.8 Ft=3.5 At=(0.4,1.6,2.1)	

Drittes Bedienende

Zustandsvariablen	Simulatorzustand
Q=0 B=0 ai=3 bi=3 t_{last}=3.1	t=3.3 EL: 3.8 A; 10 E
Resultatvariablen	
Qt=1.8 Ft=4.7 At=(0.4,1.6,2.1)	

Vierte Ankunft

Zustandsvariablen	Simulatorzustand
Q=0 B=1 ai=4 bi=3 t_{last}=3.8	t=3.8 EL: 4.0 A; 4.9 B; 10 E
Resultatvariablen: At=(0.4,1.6,2.1, 3.8) Qt=1.8 Ft=4.7	

Fünfte Ankunft

Zustandsvariablen	Simulatorzustand
Q=1 B=1 ai=5 bi=3 t_{last}=4.0	t=4.0 EL: 4.9 B; 5.6 A; 10 E
Resultatvariablen: At=(0.4,1.6,2.1, 3.8,4.0) Qt=1.8 Ft=4.7	

Viertes Bedienende

Zustandsvariablen

Q=0 B=1 ai=5 bi=4 t_{last}=4.9

Resultatvariablen

Qt=2.7 Ft=5.8 At=(0.4,1.6,2.1, 3.8,4.0)

Simulatorzustand

t=4.9 EL

5.6	A
8.6	B
10	E

Sechste Ankunft

Zustandsvariablen

Q=1 B=1 ai=6 bi=4 t_{last}=5.6

Resultatvariablen

Qt=2.7 Ft=5.8 At=(0.4,1.6,2.1, 3.8,4.0,5.6)

Simulatorzustand

t=5.6 EL

5.8	A
8.6	B
10	E

Siebte Ankunft

Zustandsvariablen	Simulatorzustand
Q=2 B=1 ai=7 bi=4 t_{last}=5.8	t=5.8 EL: 7.2 A; 8.6 B; 10 E
Resultatvariablen Qt=2.9 Ft=5.8	At=(0.4,1.6,2.1, 5.8,3.8,4.0, 5.6)

Achte Ankunft

Zustandsvariablen	Simulatorzustand
Q=3 B=1 ai=8 bi=4 t_{last}=7.2	t=7.2 EL: 8.6 B; 10 E; 10.2 A
Resultatvariablen Qt=5.7 Ft=5.8	At=(0.4,1.6,2.1, 3.8,4.0,5.6, 5.8,7.2)

Fünftes Bedienende

Zustandsvariablen	Simulatorzustand
Q=2 B=1 ai=8 bi=5 t_{last}=8.6	t=8.6 EL: 10 E; 10.1 B; 10.2 A
Resultatvariablen Qt=9.9 Ft=10.4	At=(0.4,1.6,2.1, 3.8,4.0,5.6, 5.8,7.2)

Simulationsende

Zustandsvariablen	Simulatorzustand
~~Q=3 B=1 ai=8 bi=5 t_{last}=7.2~~	t=10 EL: ~~10.1 B; 10.2 A~~
Ausgabe Qt=12.7/10= **1.27** Ft=10.4/5= **2.08**	

Einige Beobachtungen für dieses Beispiel

- Werte von Q und B werden implizit in ai und bi codiert (Q=max(0,bi-ai-1), B=δ(bi>ai))
- Länge der Ereignisliste auf maximal 3 Elemente beschränkt (ein Element pro Ereignistyp)
- Es werden keine Elemente aus der Ereignisliste gelöscht
- At speichert alle bisherigen Ankunftszeiten
  - Länge wächst mit steigender Simulationszeit/Ankunftszahl
  - Benötigt werden aber nur Ankunftszeiten bi, ..., ai
    andere Speicherstruktur wäre sinnvoll,
    aber Anzahl zu speichernder Elemente ist a priori unbekannt

Grundsätzliches Vorgehen beim einfachen Beispielmodell ist auf komplexere Modelle übertragbar, aber
komplexere Modelle verlangen komplexere Datenstrukturen

Beispiel:

- Kunden haben Eigenschaften, die die Bedienzeit beeinflussen
- Jeder Kunde wird durch eine Datenstruktur Kd zur Codierung der Eigenschaften dargestellt
- Population im System ist eine Menge von Kunden (Liste von Variablen vom Typ Kd)
- Bei Ankunft eines Kunden muss eine Variable vom Typ Kd initialisiert werden
- Verlässt ein Kunde das System, so wird die zugehörige Variable nicht mehr benötigt

auch hier dynamische Belegung und Freigabe von Speicherplatz!

Zeitablauf in der Simulation:

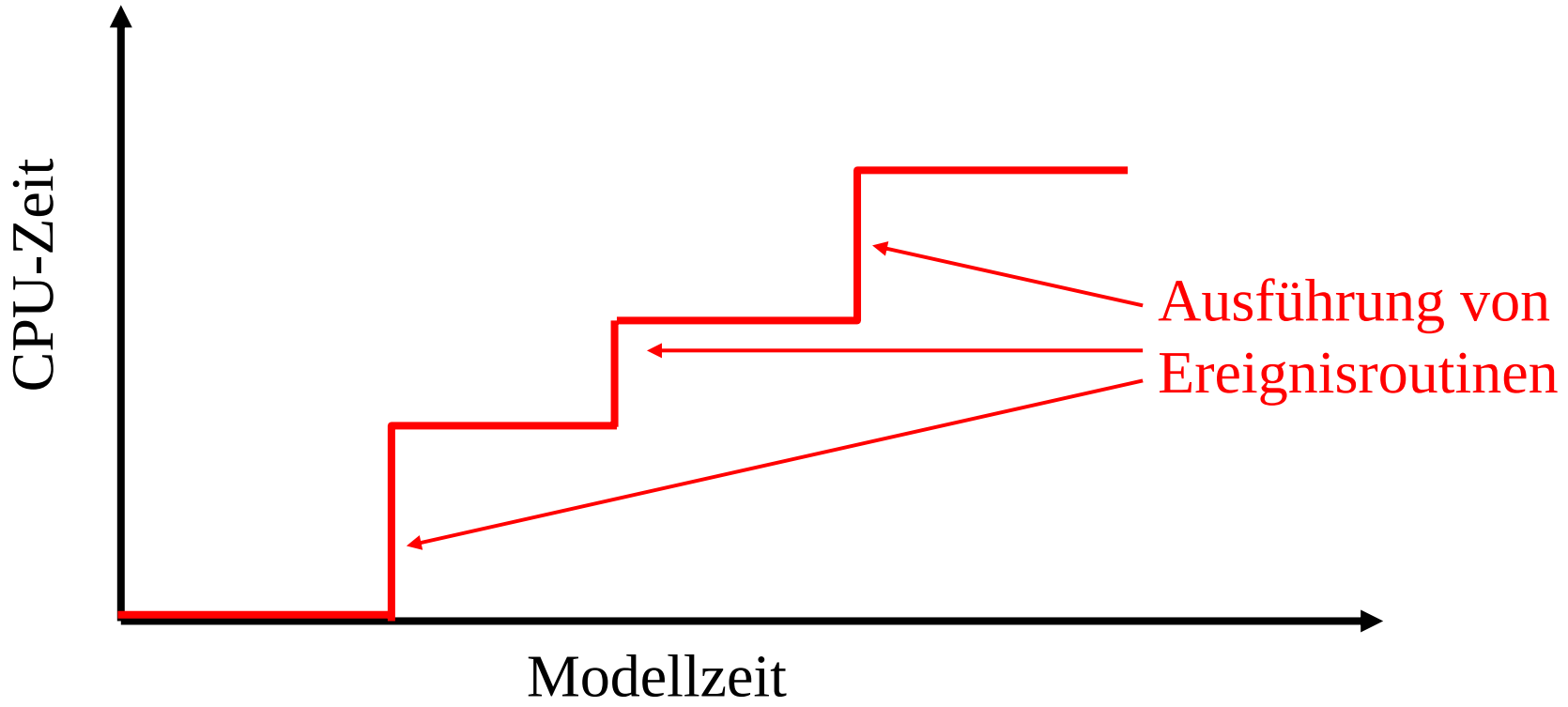
- Zeit läuft diskontinuierlich in Sprüngen ab
  - Zustand ist nur zu Ereigniszeitpunkten definiert (d.h. Zustand direkt nach Eintreten des atomaren Ereignisses)
  - Zwischen Ereigniszeitpunkten ist der Zustand nicht (vollständig) definiert
  - Falls der Zustand zu einem Zeitpunkt t relevant ist, so ist für diesen Zeitpunkt ein Ereignis vorzumerken
  - Da mehrere Ereignisse zu einem Zeitpunkt stattfinden können, kann das System zu einem Zeitpunkt mehrere Zustände haben
- Ein Ablauf eines Simulators über ein Zeitintervall [0,T] wird als **Trajektorie** bezeichnet
  - In stochastischen Simulationen können (unendlich) viele unterschiedliche Trajektorien auftreten

Zeitbegriffe in der Simulation:

Es gibt eine Reihe unterschiedlicher Zeitbegriffe
(oft Quelle von Missverständnissen)

- **Objektzeit**, jene Zeit, in der (zumindest hypothetisch) der Betrieb des realen Systems abläuft
- **Modell-/Simulationszeit**, jene Zeit, die im Programm „Simulator“ manipuliert wird (indem t gesetzt wird);
  - Modellzeit imitiert Objektzeit, d.h. ist identisch zu ihr (bis auf Translation, etwa Start bei „0“)
- **Realzeit** des Simulationsprogramms, (das als Programm real in einem Zeitintervall abläuft) ist für uns belanglos
- **Ausführungszeit/CPU-Zeit** (des Simulationsprogramms), jene Zeit, welche die CPU zur Ausführung benötigt
  - CPU-Zeit ist Teil des Ressourcenbedarfs
  - Simulatoren sind notorische Langläufer

Zusammenhang Modellzeit-CPU-Zeit



Auf den ersten Blick verwirrender Zusammenhang:

- Immer dann, wenn Modellzeit verstreicht, verstreicht keine CPU-Zeit
- und umgekehrt

- Endliche Modellzeitintervalle (zwischen Ereignissen) werden in verschwindender CPU-Zeit nachvollzogen
- Verschwindende Modellzeitintervalle (zu Ereigniszeitpunkten) werden in endlicher CPU-Zeit realisiert

Ausführungszeit eines Simulators ist

- von der Anzahl der simulierten Ereignisse,
- der Komplexität der zugehörigen Ereignisroutinen
- und nicht allein von der überstrichenen Modellzeit abhängig

Erhöhung der Effizienz muss an der Ereigniszahl und -komplexität ansetzen

$\Rightarrow$ am Abstraktionsniveau unseren mentalen Modells

Im allgemeinen gilt:

höhere Abstraktionsniveau $\rightarrow$ weniger Ereignisse

2.2 Spezifikation von Simulatoren

Vorgestelltes Prinzip der ereignisdiskreten Simulation ist strukturiert und verständlich, aber Programmierung von Simulatoren erfordert Abstraktion und ist damit komplex!

Grundsätzliche Fragestellungen

- Welche sprachlichen Konstrukte (Syntax + Semantik) sind notwendig, um Simulatoren zu implementieren?
- Wie werden diese Konstrukte umgesetzt oder realisiert?

Wir betrachten hier **programmiersprachliche Realisierungen**, es gibt andere Möglichkeiten

- Menü-Techniken
- Graphische Eingabeschnittstellen
- ...

Etwas mehr dazu in Abschnitt 2.6

Anforderungen der Simulation an eine Programmiersprache:

1. Höhere rekursive Datenstrukturen zur Speicherung von homogenen oder auch inhomogenen (komplexen) Variablen/Objekten mit Funktionen zum
   - Einfügen (nach Schlüssel, am Anfang, am Ende ...)
   - Herauslesen (am Anfang, am Ende, ...)
   - Löschen (nach Schlüssel)
2. Kalender zukünftiger nach Eintretenszeit sortierten Ereignissen ($\rightarrow$ Spezialfall von 1.)
3. Methoden zur Erzeugung und Zerstörung von Variablen/Objekten
   (Dynamische Datenstrukturen und Freispeicherverwaltung)
4. Methoden zur Erzeugung von Zufallszahlen

Höhere Datenstrukturen in heutigen/modernen Programmiersprachen vorhanden als *structure*, *record*, *class*, ...

Beispiel Kunde mit Attributen:

- Name als Zeichenkette
- Ankunftszeit als reelle Zahl
- Priorität als ganze Zahl

In C:

```
typedef struct {
        char *name ;
        float azeit ;
        int    prio ;

} Kunde ;
```

In Java:

```
class Kunde {
        char[] name ;
        float azeit ;
        int    prio ;
        // Methodendefinitionen
        .....
}
```

Dynamische Speicherverwaltung in fast allen modernen Programmiersprachen vorhanden

- Funktionen zur dynamischen Allokation eines Speicherbereichs in Form von *new*, *malloc*, ...
- Freigabe des allokierten Speicherbereichs entweder explizit (per *dispose*, *free*, ..) oder implizit (per *garbage collection*)

Fehleranfälliger Teil in vielen Programmen, durch

- Vergessen der Allokation
- falsche Initialisierung
- fehlende Speicherfreigabe
- zu frühe Speicherfreigabe

Fehler nur sehr eingeschränkt statisch prüfbar/erkennbar

Zur Laufzeit auftretende Fehler oft schwer lokalisierbar

⇒ weitgehende Benutzerunterstützung wird als notwendig angesehen!

In C:

Speicherallokation durch Funktionen:

```
void *malloc(size_t size) ; // Allokiert size Bytes, ohne Initialisierung!
void *calloc(size_t nelem, size_t elsize) ;
                    // Allokiert nelem * elsize Bytes und initialisiert mit 0
void *realloc(void *prt, size_t size) ;
                    // Der Speicherbereich, auf den prt zeigt wird auf Länge
                    // size verlängert/verkürzt
void free (void *prt) ;
                    // Speicherfreigabe des Bereichs, auf den prt zeigt
```

Zuweisung durch Typangabe, also

```
Kunde *meine_kunden = (Kunde *) calloc(10, sizeof(Kunde)) ;
```

Allokiert Speicherplatz für 10 Kunden

```
free(meine_kunden) ; // Freigabe
```

Adressarithmetik ist möglich (aber gefährlich)

In Java:
Allokation durch new

```
int[] size = new int[20] ;
        // allokiert Speicherplatz für 20 integer mit 0 initialisiert
```

Freigabe des Speicherplatzes implizit durch den „Garbage Collector“, wenn keine Referenz auf diesen Bereich im Programm existiert
Für komplexere Datenstrukturen/Klassen explizite Definition eines Konstruktors oder mehrer Konstruktoren mit einem zum Klassennamen identischen Namen

```
Kunde {
        prio = 1 ;
        azeit = 0.0 ;
        name = „Mustermann“;
} // ein Konstruktor
```

```
Kunde (int my_prio, float my_azeit,
        char[] my_name)
{
        prio   = my_prio ;
        azeit  = my_azeit ;
        name = my_name ;
} // ein anderer Konstruktor
```

- Freispeicherverwaltung in C ist sehr flexibel und effizient
- aber auch extrem fehleranfällig
- Freispeicherverwaltung in Java deutlich weniger effizient (z.B. garbage collection ist sehr aufwändig) und auch weniger flexibel
- Durch vorgenommene Einschränkungen werden viele Fehlermöglichkeiten zwar nicht beseitigt, aber doch deutlich eingeschränkt (z.B. Speicherlecks)

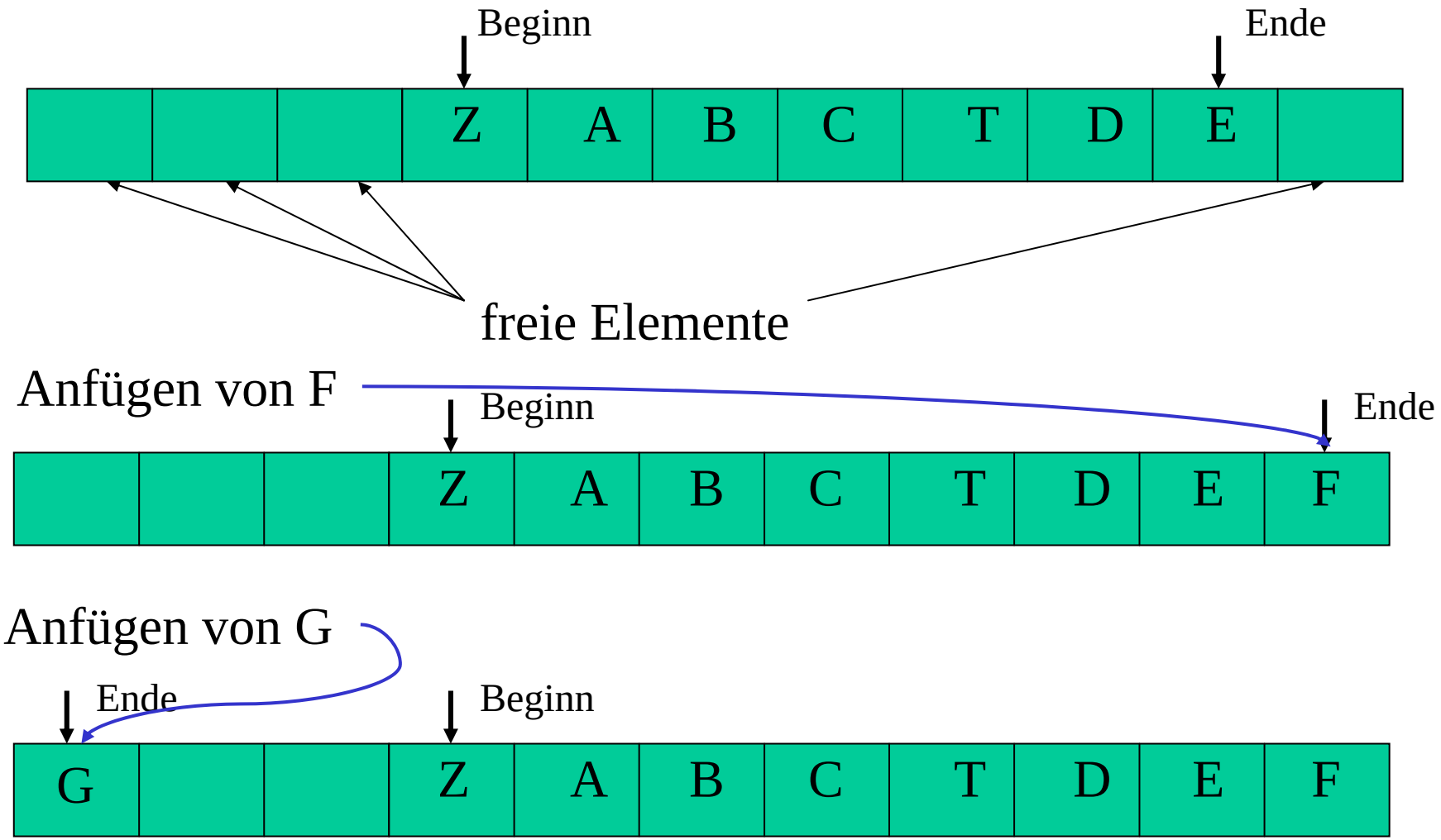
In der Simulation werden i.d.R. sehr viele temporäre Objekte benötigt

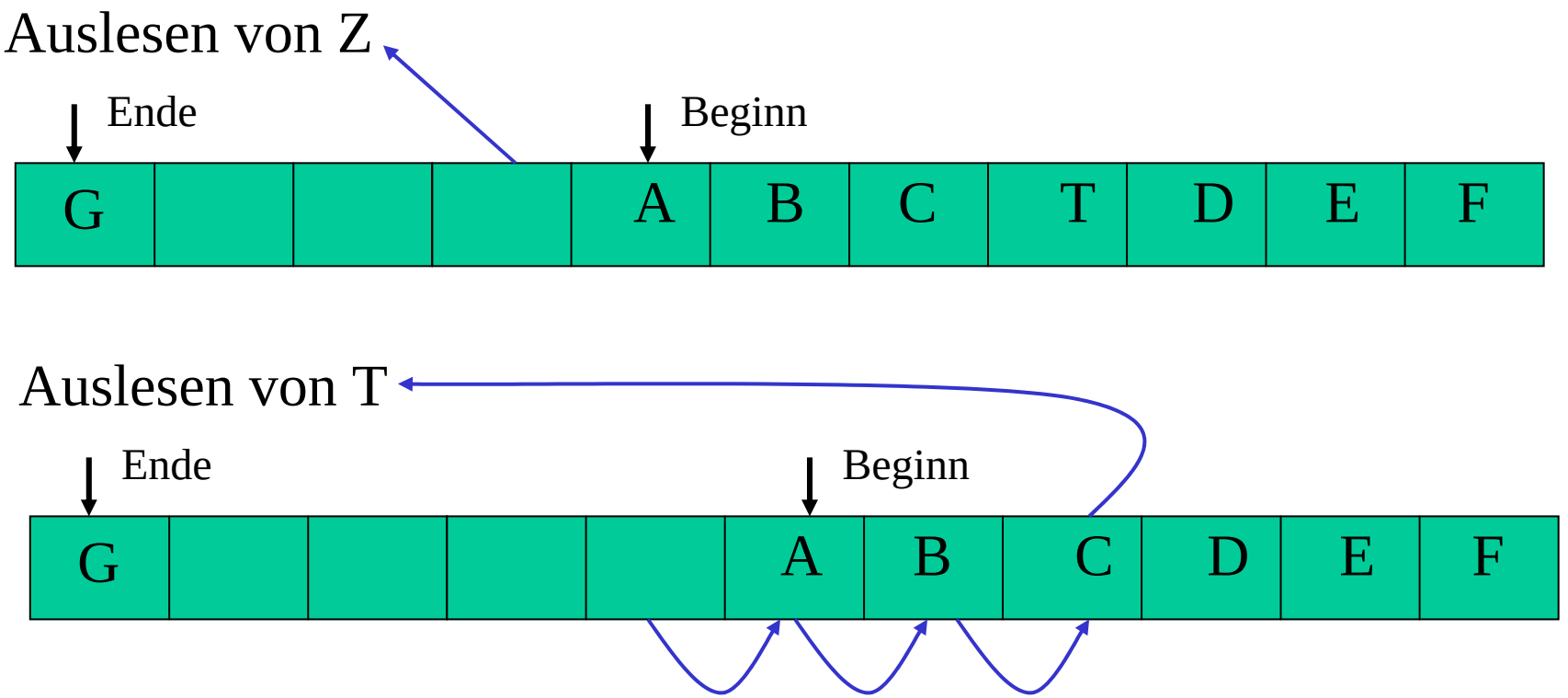
- Jeder ankommende Kunde muss dynamisch zur Laufzeit generiert werden
- Nachdem ein Kunde das System verlassen hat, muss der belegte Speicherplatz wieder freigegeben werden

Speicherallokation und –freigabe sollten automatisch erfolgen

- Simulationssprachen/Simulationswerkzeuge mit entsprechender Unterstützung verwenden

Realisierung von Listen: Einfachste Form Arrays fester Maximallänge





Offensichtliche Nachteile:

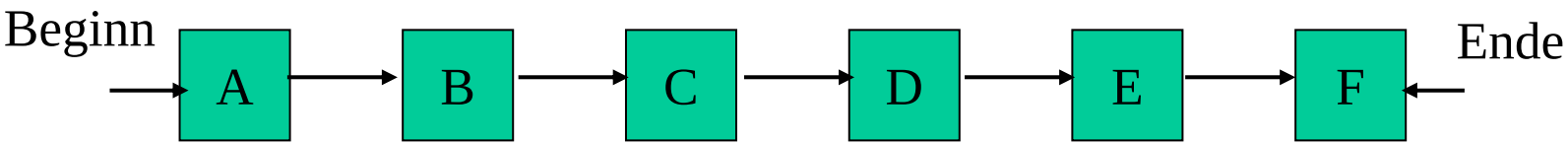
- Feste Maximallänge (d.h. konstanter Speicherbedarf)
- Aufwändiges Entfernen von Elementen aus der Mitte

Aber auch Vorteile

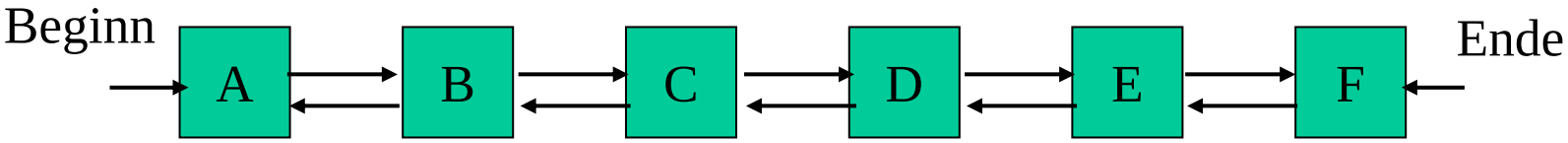
z.B. Möglichkeit der binären Suche, falls Elemente geordnet

Realisierung von Listen mit Pointern:

Einfache Verkettung



Doppelte Verkettung



Darstellung mit offensichtlichen Vorteilen:

- Speicherbedarf proportional zur Länge
- Maximallänge nur durch verfügbaren Speicher beschränkt
- Kein Umspeichern beim Ausfügen aus der Mitte
- Einfache Realisierung von Listen mit inhomogenen Elementen

Ereignisliste als spezielle Liste

Primäre Operationen:

- Herauslesen des ersten Elements
- Einfügen eines Elements nach Schlüssel

Seltener gebraucht

- Löschen eines beliebigen Elements

Datenstruktur für die Listenelemente

- Einfaches Feld mit zwei Elementen
  - t zur Darstellung der Ereigniszeit
    Typ (nicht negative) relle Zahl
  - tp zur Darstellung des Ereignistyps
    i.d.R. als ganze Zahl codiert

Aufwand der Operationen bei Listenlänge n

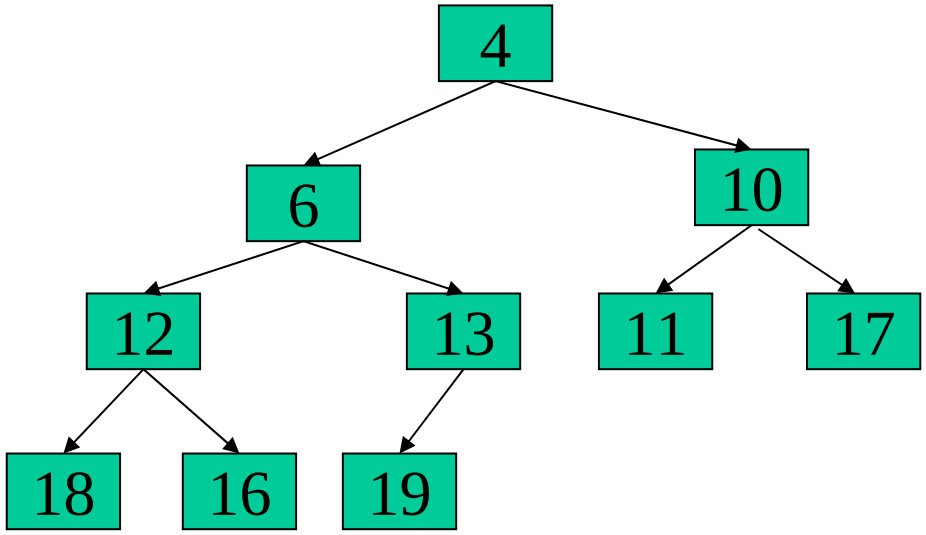
- Herauslesen O(1)
- Einfügen und Löschen O(n)

Andere Möglichkeiten zur Realisierung von Ereignislisten

Darstellung als Heap (Binärbaum):

- Wurzel beinhaltet minimales Element
- Vorgänger kleiner als Nachfolger
- Unterste Ebene von links gefüllt

Bsp. nur Werte von t angegeben



Auslesen des ersten Elements:

- Wurzel entfernen
- Letzten Knoten der untersten Ebene auf die Wurzel setzen
- Wurzel solange mit kleinstem Nachfolger vertauschen, bis kein kleinerer Nachfolger mehr existiert

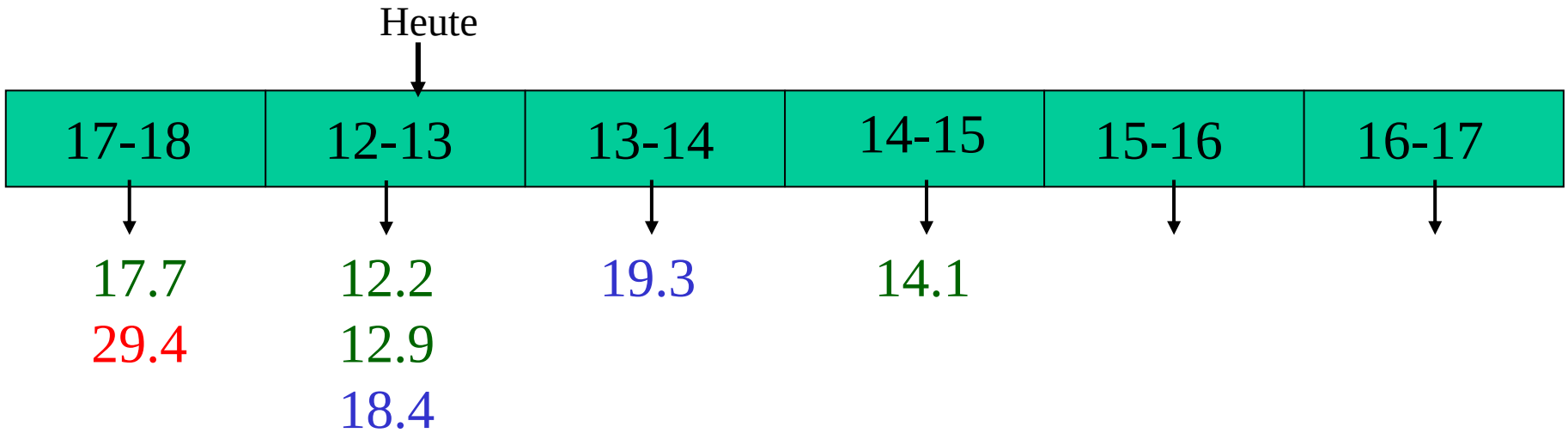
Einfügen eines Elements:

- Einhängen als letztes Element der untersten Ebene
- Solange mit Vorgänger vertauschen, bis dieser kleiner

Aufwand Auslesen und Einfügen jeweils O(log n)!

Darstellung als Kalender: Datenstruktur Array von Listen

- Jede Zelle enthält Ereignisse aus einem festen Zeitraum
- Ereignisse aus jedem Zeitraum werden in geordneter Liste gehalten



- Dynamische Anpassung der Zellbreite, so dass ca. 75% der Einfügeoperationen „in diesem Jahr stattfinden“
- Dynamische Anpassung der Zellzahl
  - Halbieren, falls Anzahl Elemente < 0.5*Anzahl Zellen
  - Verdoppeln, falls Anzahl Elements > 2*Anzahl Zellen

Experimente zeigen konstanten Aufwand für Entfernen und Einfügen!

Generierung von Zufallszahlen:

Methoden zur Generierung von Zufallszahlen vorgegebener Verteilungen, also

```
atime = ziehe_zz(Verteilungstyp, Parameter) ;
```

Später mehr zur Realisierung dieser Funktionen

Weitere Funktionen zur Realisierung von Simulationen:

```
plane(Ereignistyp, Ereigniszeit) ;
```

Zum Einhängen eines Ereignisses in die Ereignisliste

```
ereignis = erstes_ereignis() ;
```

Zum Auslesen des ersten Ereignisses aus der Liste

```
zeit = t ;
```

Zum Auslesen der Zeit des nächsten Ereignisses in der Ereignisliste

Funktionen existieren so oder ähnlich in vielen Simulationssprachen!

Vorgestellte Konstrukte reichen aus, um ereignisdiskrete Simulatoren auf dem bisher beschriebenem Niveau zu realisieren!

Dazu notwendig

- Aufbau der Datenstruktur zur Abbildung der statischen Struktur
- Definition aller Ereignisse und Realisierung der zugehörigen Ereignisroutinen

Ansatz wird als „**event scheduling**“ bezeichnet

Für komplexe Probleme ist das Vorgehen sehr aufwändig und fehleranfällig, da

- sämtliche Ereignisse und deren (globale) Auswirkungen erkannt und kodiert werden müssen
- flache aber stark vernetzte und damit unübersichtliche Modelle entstehen

Geht es einfacher/besser/sicherer ? Wenn ja, wie?

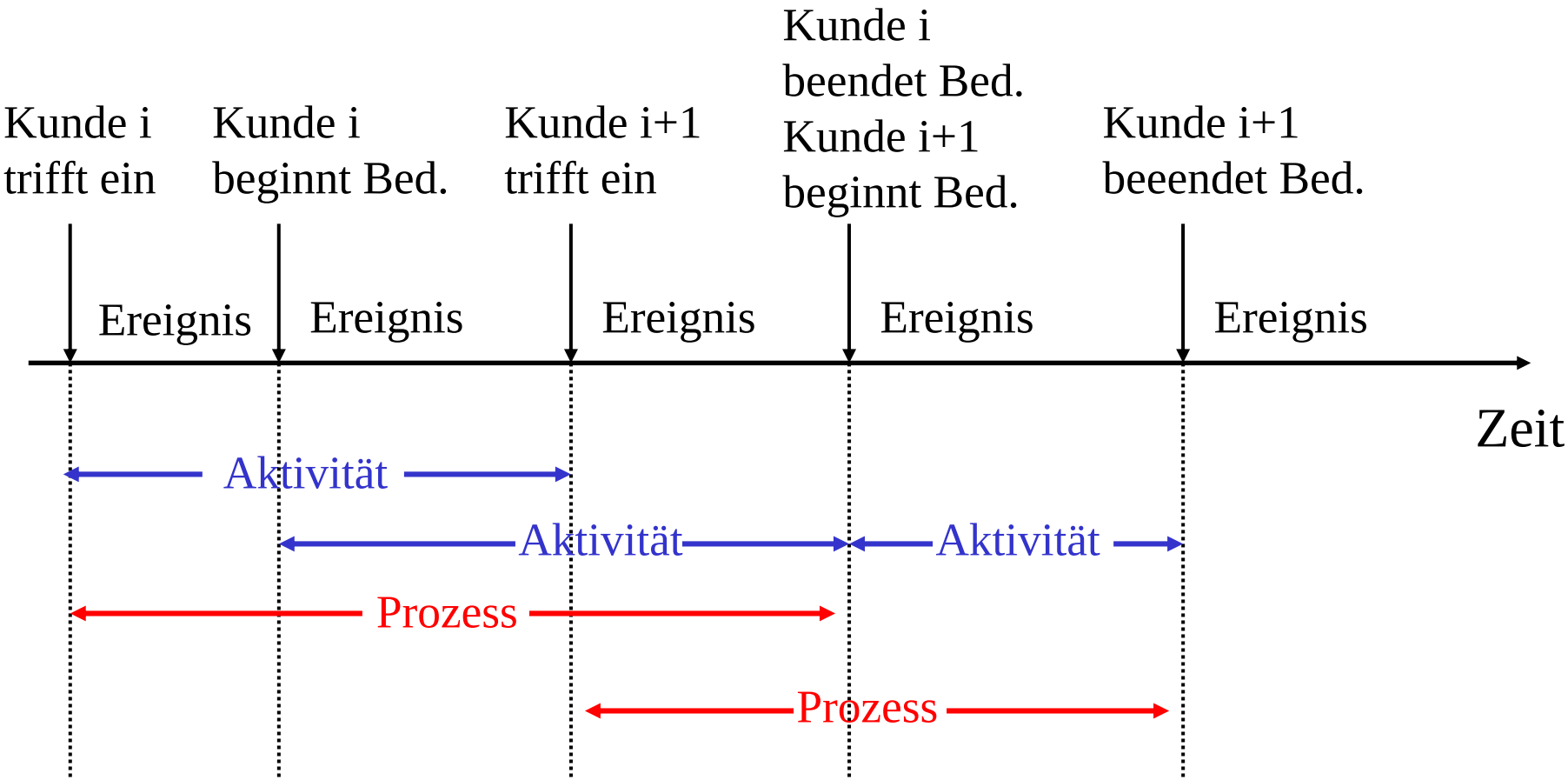
Basis der Modellierung ist unser mentales oder semi-formales Modell!

- Einfachere Simulationsmodelle müssen „näher“ am mentalen Modell sein
- Viele Ereignisse haben „lokale“ Auswirkungen, die sich nur auf einzelne Zustandsvariablen beziehen
- Denken in elementaren Ereignissen ist nicht abstrakt genug
- Ereignisse müssen zusammengefasst/strukturiert werden

Verschiedene Strukturierungsmethoden existieren in der Literatur, weit verbreitet ist die Unterteilung in

- event scheduling
- activity scanning
- process interaction

Unterscheidung am Beispiel des einfachen Schalters:



Entsprechen der Abbildung

- drei „Sichten“ und
- zugehörige konzeptuelle Rahmen der Zeitsteuerung
- mit steigender Abstraktion

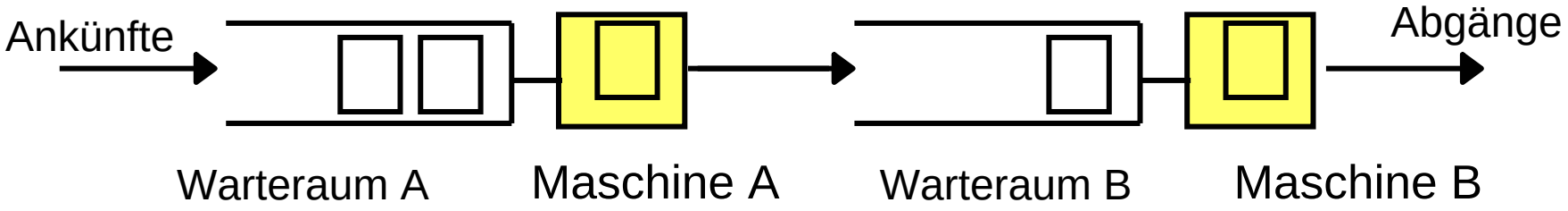
Verknüpfung zwischen Zustand und Zeit durch:

1. **Ereignis**: Veränderung der Zustandsvariablen zu bestimmten Zeitpunkten
2. **Aktivität**: Menge von Operationen während eines Zeitintervalls
3. **Prozess**: Folge von Aktivitäten eines Objekts über eine Zeitspanne

Darüber hinaus weitere Freiheitsgrade für den Modellierer:
Was ist aktiv, was ist passiv?
Im Ergebnis identisch, aber unterschiedliche Sichtweisen des Systemverhaltens

- Kunden als aktive Elemente, Bediener als passive Ressourcen
  Kunde bewegt sich von einem Bediener zum nächsten
- Bediener als aktive Elemente, Kunden nur passiv
  Bediener übergibt Kunden an den nächsten Bediener
- Kombinationen aus beiden Sichtweisen

Vorstellung der Konzepte an einem einfachen Beispiel



Annahmen:

- Aufträge passieren nacheinander Maschine A und B
- Bearbeitungszeiten T_A und T_B (Zufallsvariablen)
- T_I Zeitabstand zwischen zwei Ankünften (ebenfalls ZV)
- Maschinen besitzen unbeschränkte Puffer und bedienen nach FCFS
- nach Bearbeitungsende in B, verlässt ein Auftrag das System

Analyseziele (Beispiele):

- Durchlaufzeiten der Aufträge (kundenorientiert)
- Anzahl Aufträge im Warteraum (ressourcenorientiert)
- Auslastung der Maschinen (kostenorientiert)

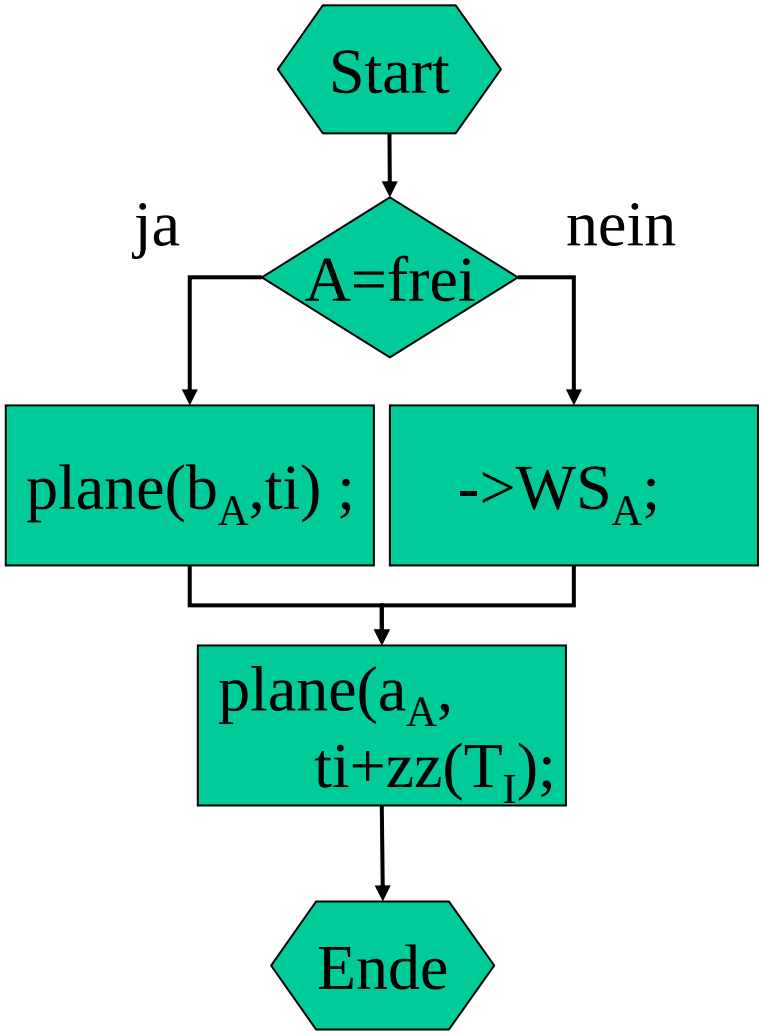
Statische Modellstruktur:

- Zwei Maschinen A und B als permanent existierende Objekte
  - jede Maschine im Zustand frei oder belegt
  - Warteräume vor den Maschinen als FIFO-Queues (von Aufträgen)
- Aufträge als temporär existierende Objekte

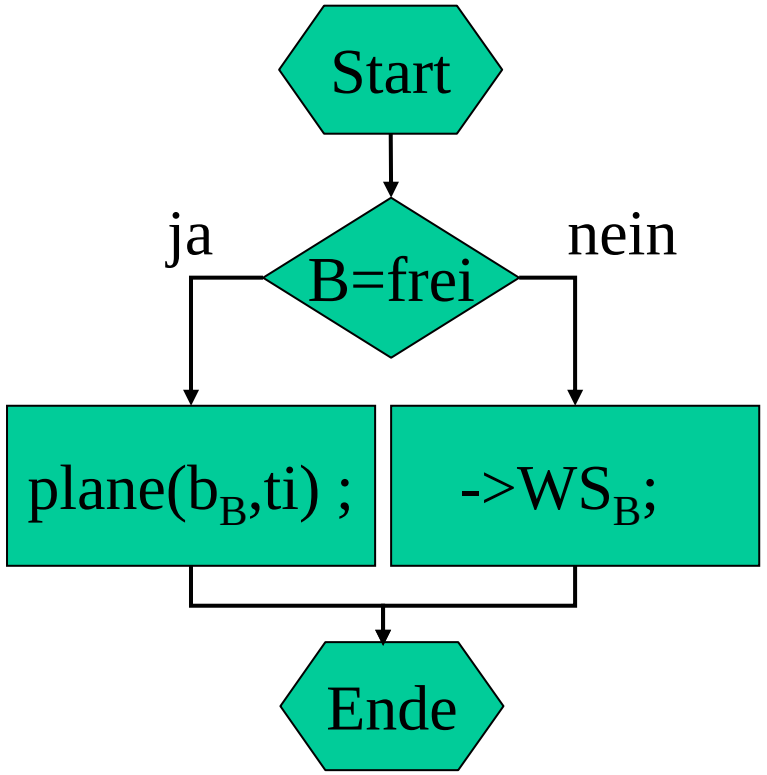
Ereignisse im event scheduling-Ansatz (aktive Aufträge)

1. Ankunft an Maschine A (a_A)
2. Ankunft an Maschine B (a_B)
3. Bearbeitungsbeginn an Maschine A (b_A)
4. Bearbeitungsbeginn an Maschine B (b_B)
5. Bearbeitungsende an Maschine A (e_A)
6. Bearbeitungsende an Maschine B (e_B)

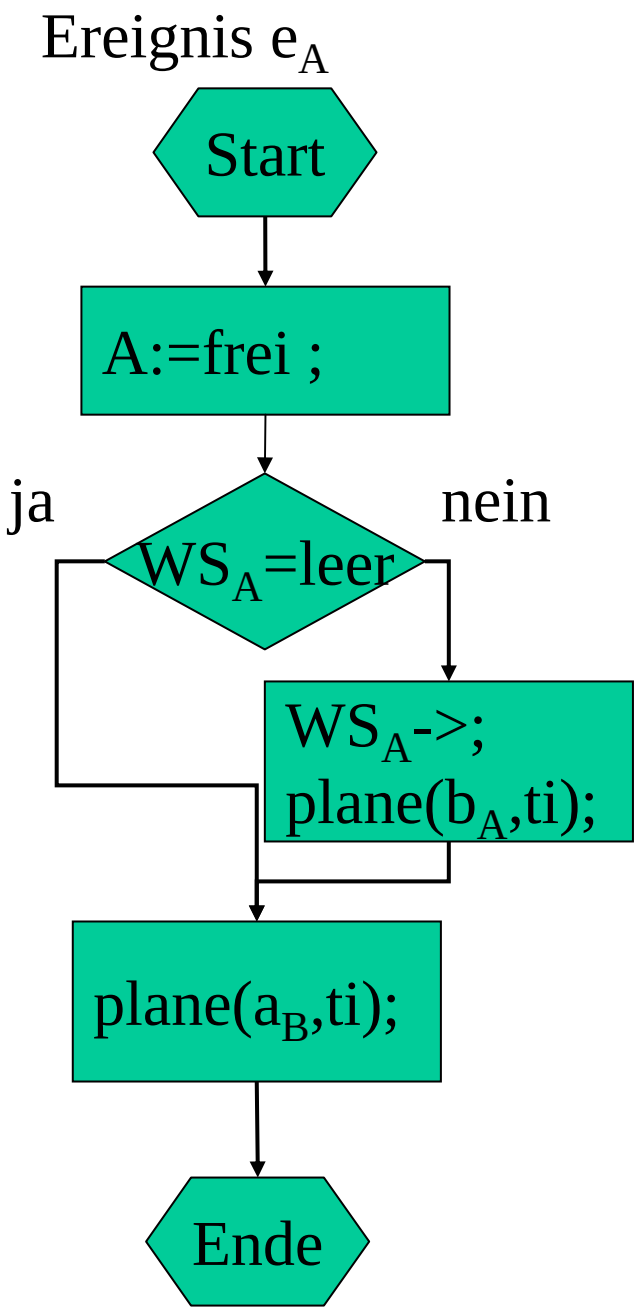
Ereignis a_A

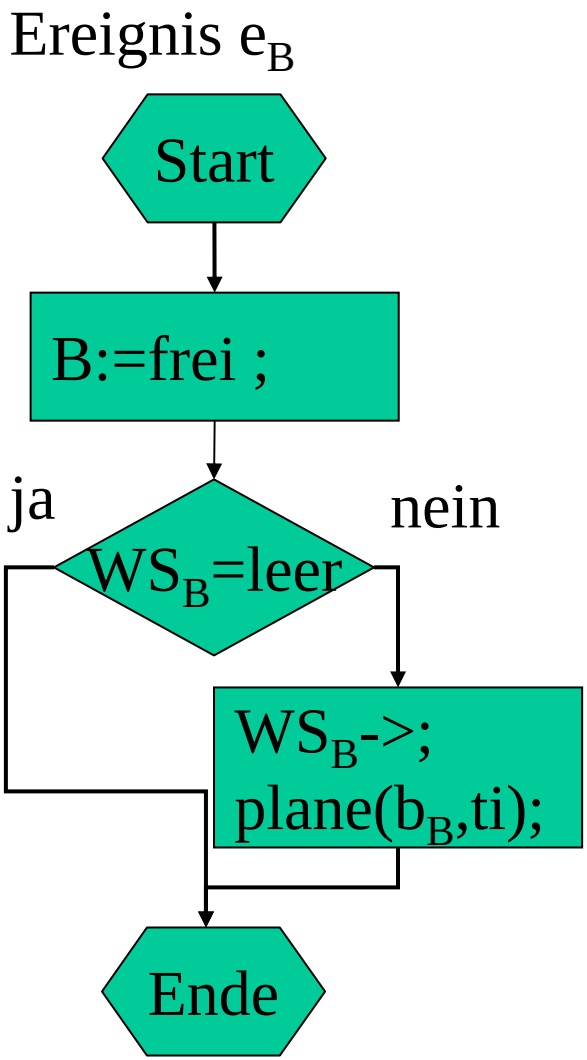


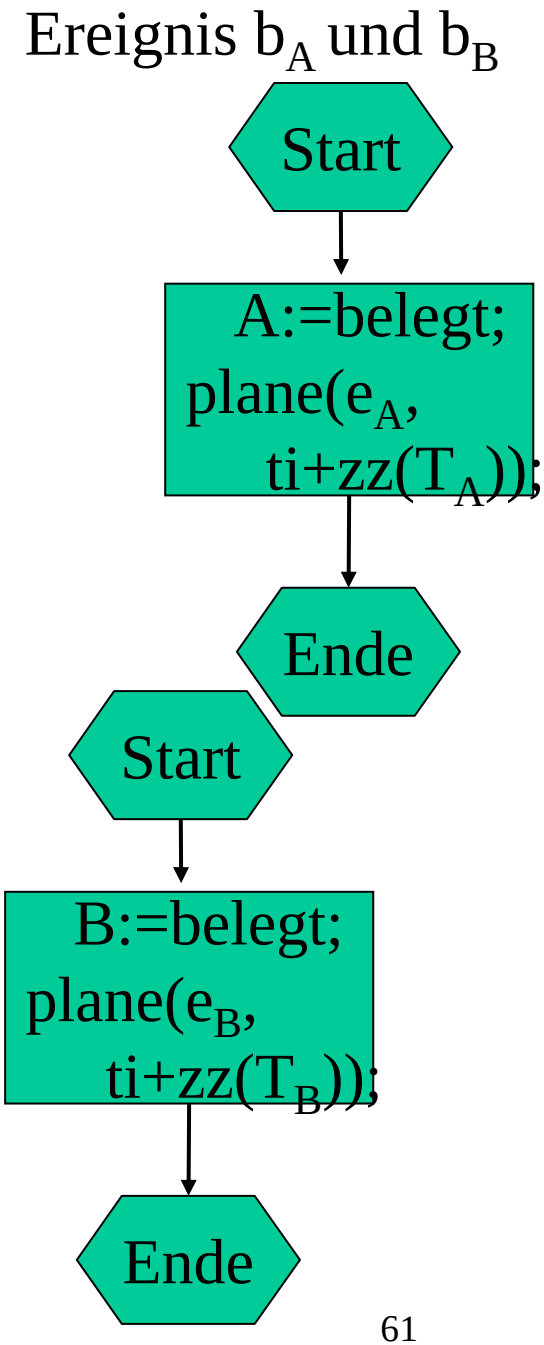
Ereignis a_B



Abkürzende Schreibweisen: ti für simzeit(),->WS_A für Auftrag in Warteschlange Maschine A, $zz(T_A)$ für Generierung einer Zufallszahl gemäß T_A.







Was fehlt noch?

Wie beginnt der Ablauf und wann endet er?

Simulationshauptroutine, die

- Zustandsvariablen initialisiert
  (z.B. ti := 0, Maschinen frei, Warteräume leer)
- erstes Ereignis plant (plane (a_A, zz(T_A)))
- Simulationslauf beendet
  (einfachste Form Einhängen eines Ereignisses *Simulationsende*)

und zu Beginn der Simulation aufgerufen wird.

Beobachtung der Simulation und Resultatermittlung

Aufzeichnung von Beobachtungen in Verbindung mit Ereignissen verbunden mit

- Aufträgen (z.B. Verweilzeiten, Wartezeiten, ...)
  - Auftrag „merkt sich“ in der Ereignisroutine, wann er angekommen ist
  - Bei Abgang des Auftrags wird in der Ereignisroutine dafür gesorgt, dass die verbrauchte Zeit gespeichert wird
- Stationen (z.B. Auslastung, Population, ...)
  - Station hat für ein bestimmtes Zeitintervall einen festen Zustand
  - Zustand ändert sich durch Ereignisse
  - In der Ereignisroutine wird der alte Zustand plus dessen Dauer und der neue Zustand plus dessen Beginn gespeichert

I.a. werden Daten in aggregierter Form (z.B. als Mittelwerte) gespeichert und ausgewertet (später dazu mehr)

Nächst höheres Abstraktionsniveau: **activity scanning**

Denkwelt:

Es gibt eine Menge von Aktivitäten, die „wissen“,

- wann/ob sie beginnen oder enden können
- was anlässlich ihres Beginns/Endes zu tun ist

Implementierungsidee:

- Aktivität besitzt (und setzt) eine eigene „Uhr“
- Aktivität zugeordnet ist eine „Aktivitätsroutine“
- Aktivitäten werden wiederholt „angestoßen“
  - testen, ob sie stattfinden können
  - und führen Zustandsänderungen aus

D.h. nach jeder Zustandsänderung (Aktivität) werden alle Aktivitäten angestoßen

Unterschied event scheduling zu activity scanning:

- Im event scheduling-Ansatz werden alle Folgen eines Ereignisses in der Ereignisroutine behandelt (d.h. insbesondere alle Folgeereignisse generiert)
- Im activity scanning-Ansatz werden nur die „unmittelbaren“ Folgen des Ereignisses beschrieben (Folgeereignisse testen selbst, ob sie aktiviert sind)

Realisierung

- Uhren der Aktivitäten stehen auf lokaler Ereigniszeit
- Nächste lokale Ereigniszeit wird jeweils ausgewählt

Vorteile activity scanning

- Stärkere Modularisierung, kompakte Beschreibung, höheres Abstraktionsniveau, problemnähere Spezifikation

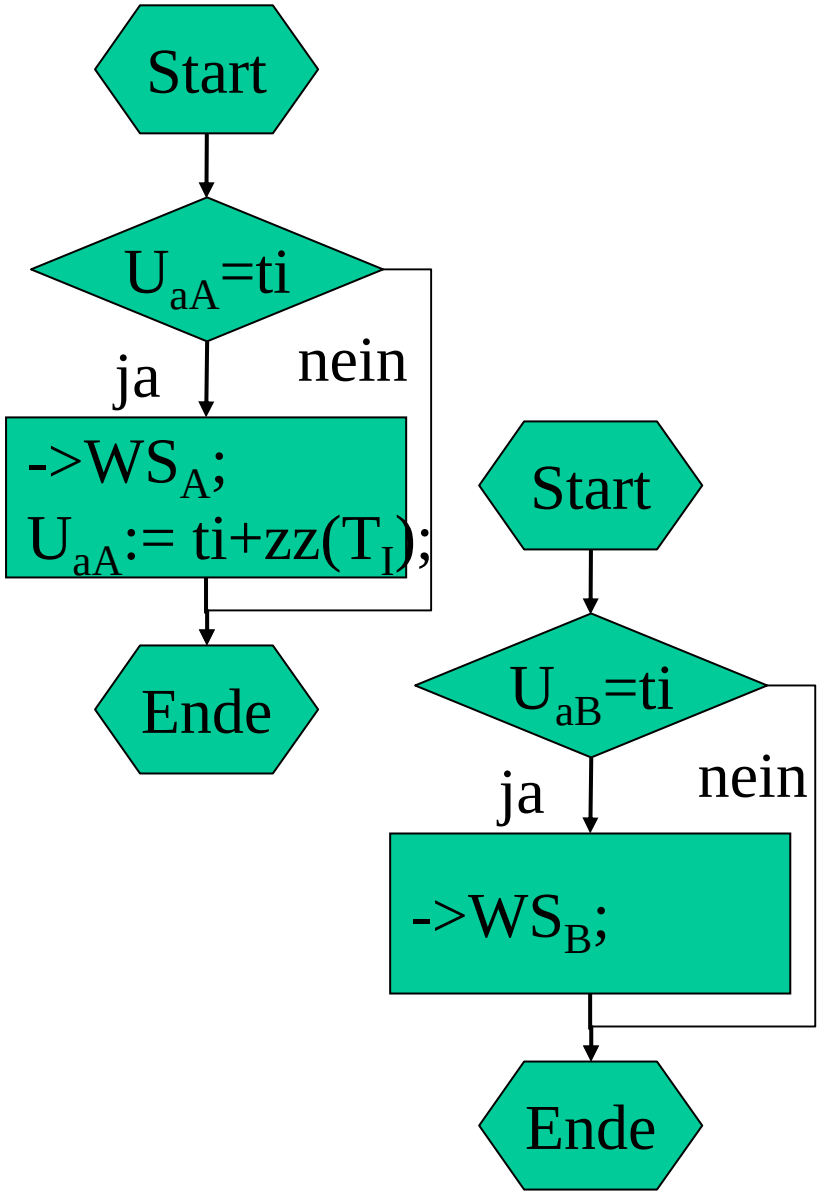
Nachteile

- Effizienz hängt stärker von der Modellstruktur ab

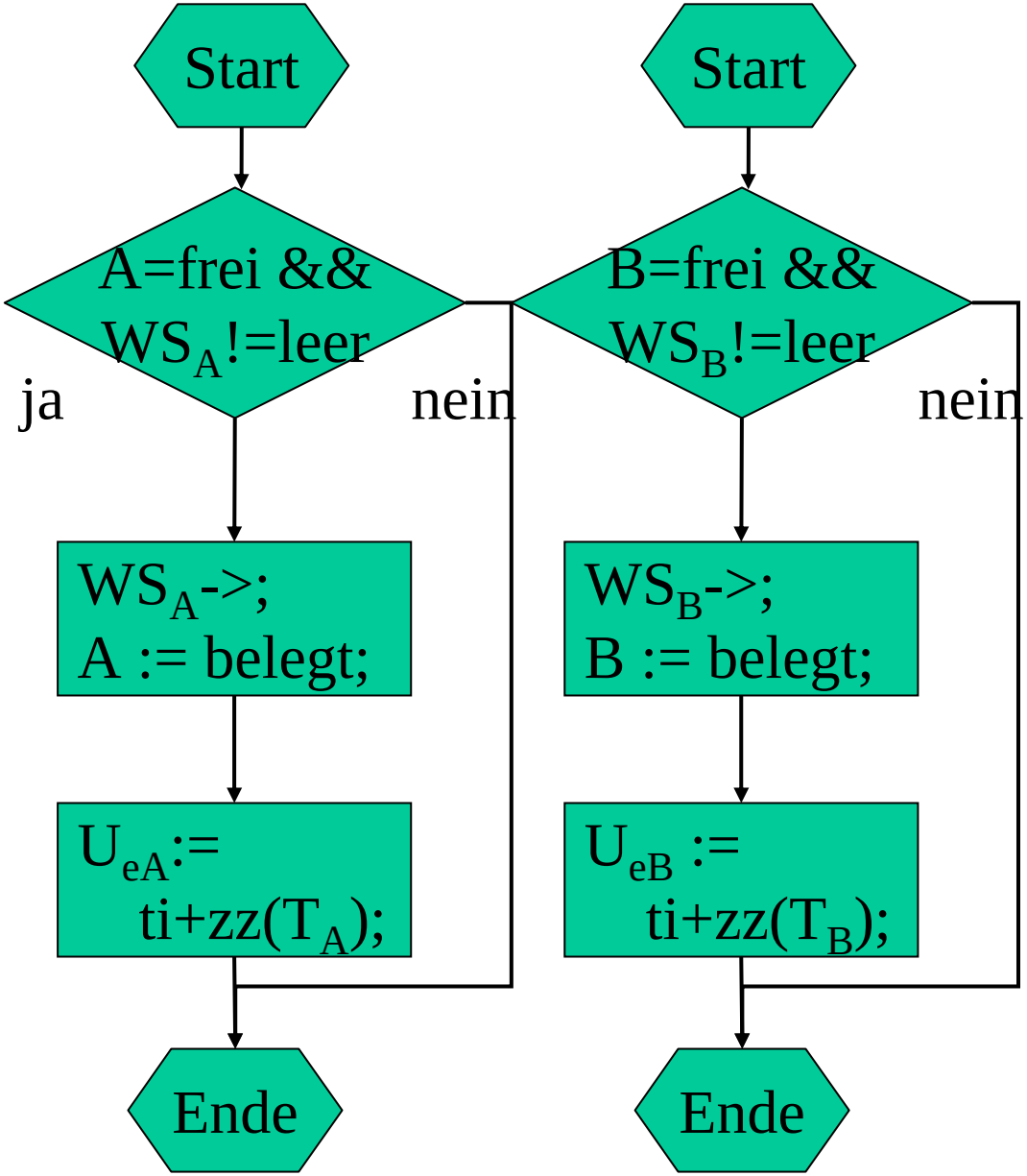
Activity scanning-Ansatz am Beispiel:

- Jede Aktivitätsroutine besitzt eine lokale Uhr (U_{xy}), die jeweils in den Aktivitätsroutinen gesetzt und gelesen wird
- Globaler Prozess springt jeweils zur nächsten „Uhrzeit“
- 6 Aktivitätsroutinen für Ankunft,Bearbeitungsbeginn und Bearbeitungsende (jeweils an jeder Maschine)
- Beobachtung/Speicherung von Resultaten wird hier nicht beschrieben

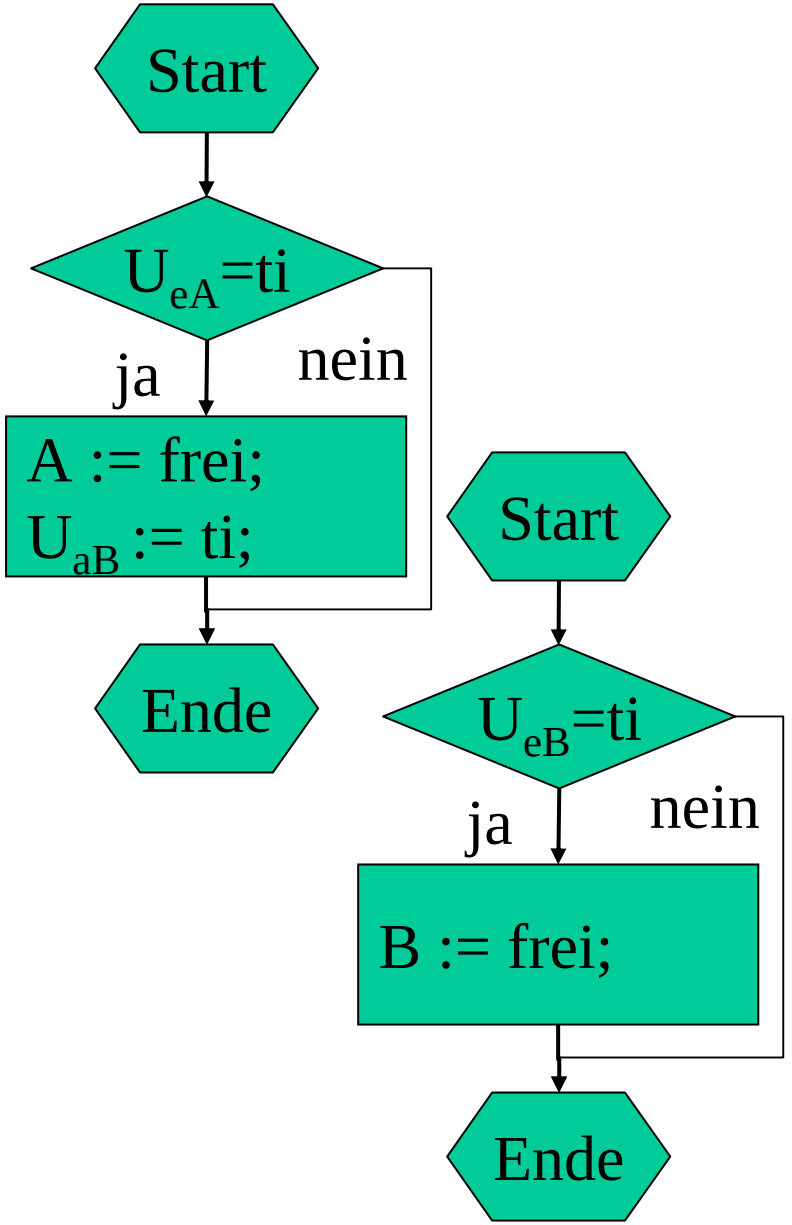
Aktivitätsroutinen Ankunft:



Aktivitätsroutinen Bearbeitungsbeginn



Bearbeitungsende



Effizienzsteigerung von activity scanning durch 3 Phasen Ansatz

1. Zeitfortschreibung
2. Ausführung von B-Aktivitäten
   (d.h. Aktivitäten, deren Vorbedingung immer eintreten müssen; im Beispiel Bearbeitungsende)
3. Ausführung von C-Aktivitäten
   (d.h. Aktivitäten mit Bedingungen)

B-Aktivitäten werden wie Ereignisse behandelt

$\Rightarrow$ Unterscheidung activity scanning und event scheduling verwischt

activity scanning-Ansatz wird nur von wenigen Programmiersprachen/Simulationssprachen unterstützt

Process interaction-Ansatz

Weitere Strukturierung durch Zusammenfassung von Ereignissen

Im Beispiel:

Ein (jeder!) Auftrag

- trifft an Maschine A ein und stellt sich an (Ereignis)
- kommt dran (Ereignis)
- ist fertig und geht weg (Ereignis)
- trifft an Maschine B ein und stellt sich an (Ereignis)
- kommt dran (Ereignis)
- ist fertig und geht weg (Ereignis)

Sechs Ereignisse bestimmen „Leben“ eines Objekts „Auftrag“

Dies ist nur eine Strukturierungsmöglichkeit!

Alternative zur Struktur nach Aufträgen, Struktur nach Maschinen:

- Die Maschine (A oder B)
  - beginnt eine Bedienung (Ereignis)
  - beendet diese später (Ereignis)
  - beginnt die nächste Bedienung (Ereignis)
  - .....
- Die Umwelt
  - schickt einen Auftrag in das System (Ereignis)
  - schickt später einen weitern Auftrag (Ereignis)
  - ...

Eine potenziell unbegrenzte Zahl von Ereignissen bestimmt das Leben der Objekte Maschine und Umwelt

Weiter auf dem eingeschlagenem Weg

$\Rightarrow$ Zusammenfassung größerer Ereignismengen

Lohnenswerter Versuch:

- Verschiedene „Objekt-Leben“ als Strukturierungsmittel nutzen (Einzelereignisse sind „Unterstrukturen“)

Hoffnung:

- Abstraktere, näher an der Vorstellung liegende Modellwelt

Vorstellung:

- Zeitabhängiges Verhalten eines dynamischen Systems ist generiert durch eine Menge dynamischer Objekte (Prozesse), deren jedes (jeder) sich entsprechen festgelegter Vorschriften (Prozessmuster) verhält

$\Rightarrow$ System „paralleler“ Prozesse

Prozesse laufen unabhängig ab und müssen von Zeit zu Zeit interagieren

Im Beispiel:

- „Bearbeitetwerden“ im Auftrags-Prozess und
- „Bearbeiten“ im Maschinen-Prozess

  Vorgänge in beiden Prozessen simultan

⇒ **Prozesse müssen sich synchronisieren**

Anforderungen zur Beschreibung solcher Systeme

1. Prozessmuster notieren
2. Prozesse (bestimmter Muster) starten
3. Prozesse „interagieren“ lassen

Unterscheidung in **temporäre** und **permanente** Prozesse

Ablauf eines Prozessmusters:

- Operationen ausführen
- Warten (d.h. nichts tun, Zeit verstreichen lassen)
- Operationen ausführen
- ....

Am Beispiel Auftrag

- Stell dich an
- Warte bis du dran bist (1.)
- Warte bis Bedienzeit abgelaufen (2.)

(1.) und (2.) beschreiben mögliche Wartebedingungen

1. Warten auf eine Bedingung (weiter, wenn Bedingung erfüllt)
2. Warten für eine Zeit (weiter, wenn Zeit abgelaufen)

Programmiersprachliche Formulierung in der Form:

wait_until (Boolsche-Bedingung)

- Formuliert die 1. Bedingung
- Prozess wird inaktiv, bis Bedingung erfüllt
- Bedingung wird durch Zustandsänderung, d.h. Ereignisse in anderen Prozessen erfüllt

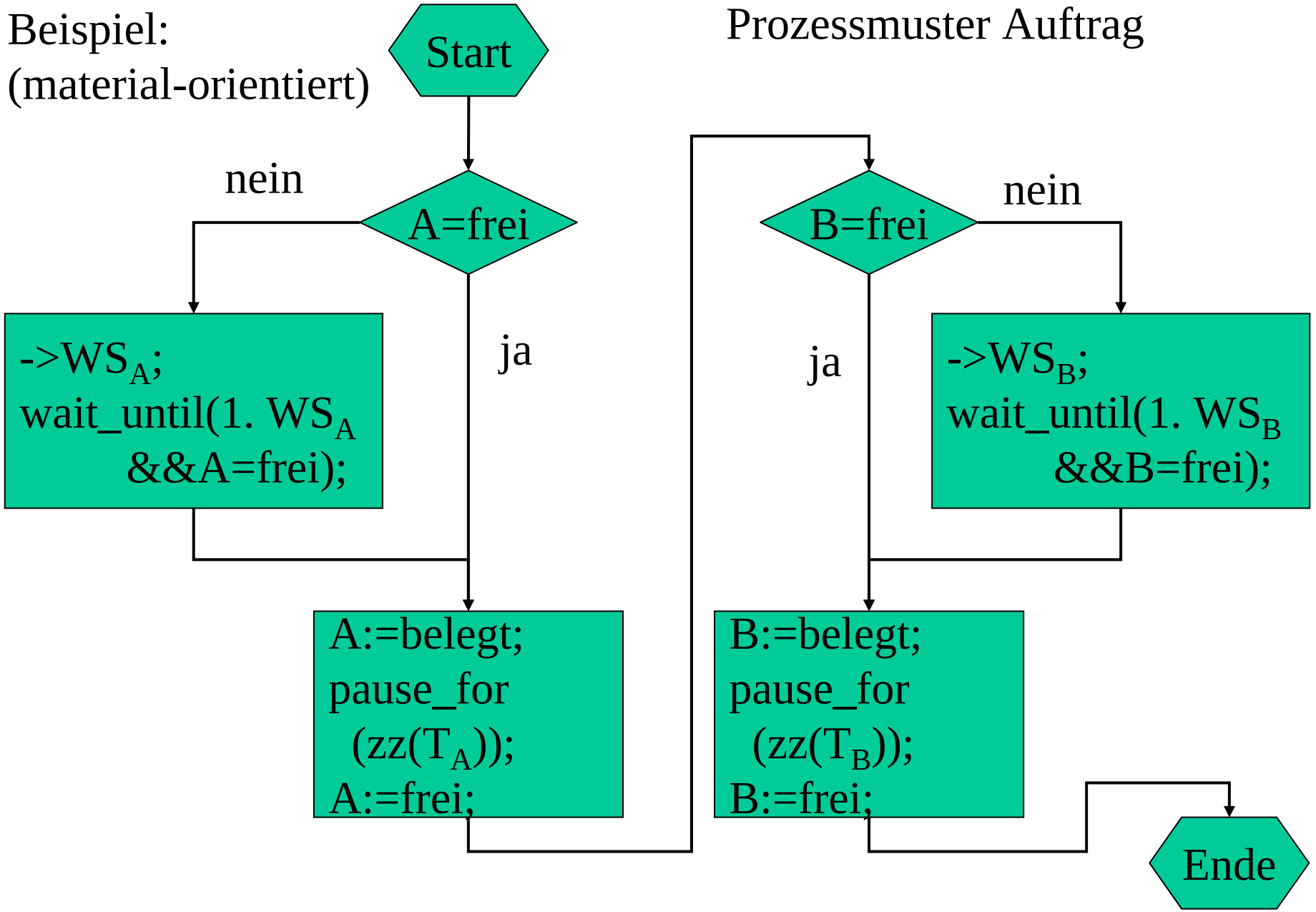
pause_for (t)

- Formuliert die 2. Bedingung
- Prozess wird für die Dauer t inaktiv
- danach wird er von selbst wieder aktiv

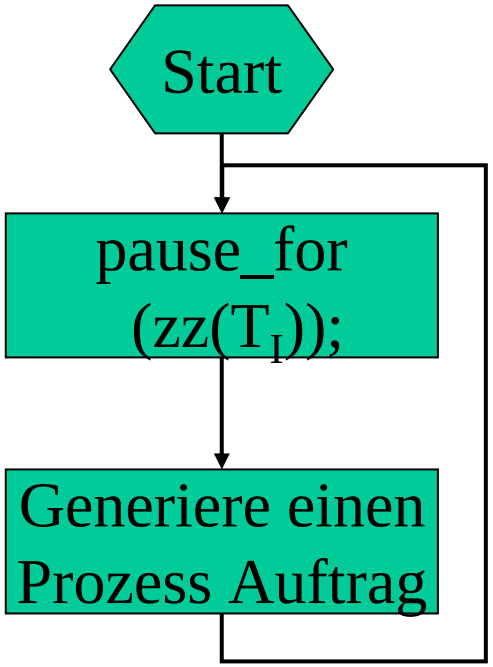
In beiden Fällen fährt der Prozess dort fort, wo er aufgehört hat, also hinter der wait_until/pause_for-Anweisung

Beispiel:
(material-orientiert)

Prozessmuster Auftrag



Generierung von Aufträgen durch Prozess Umwelt



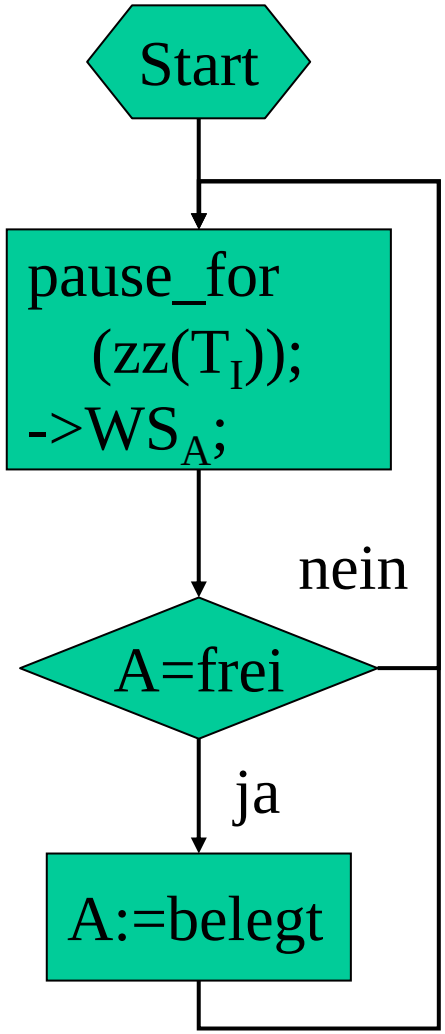
Zum Start der Simulation

- Einen Prozess Umwelt generieren
- Bis zum Simulationsende warten

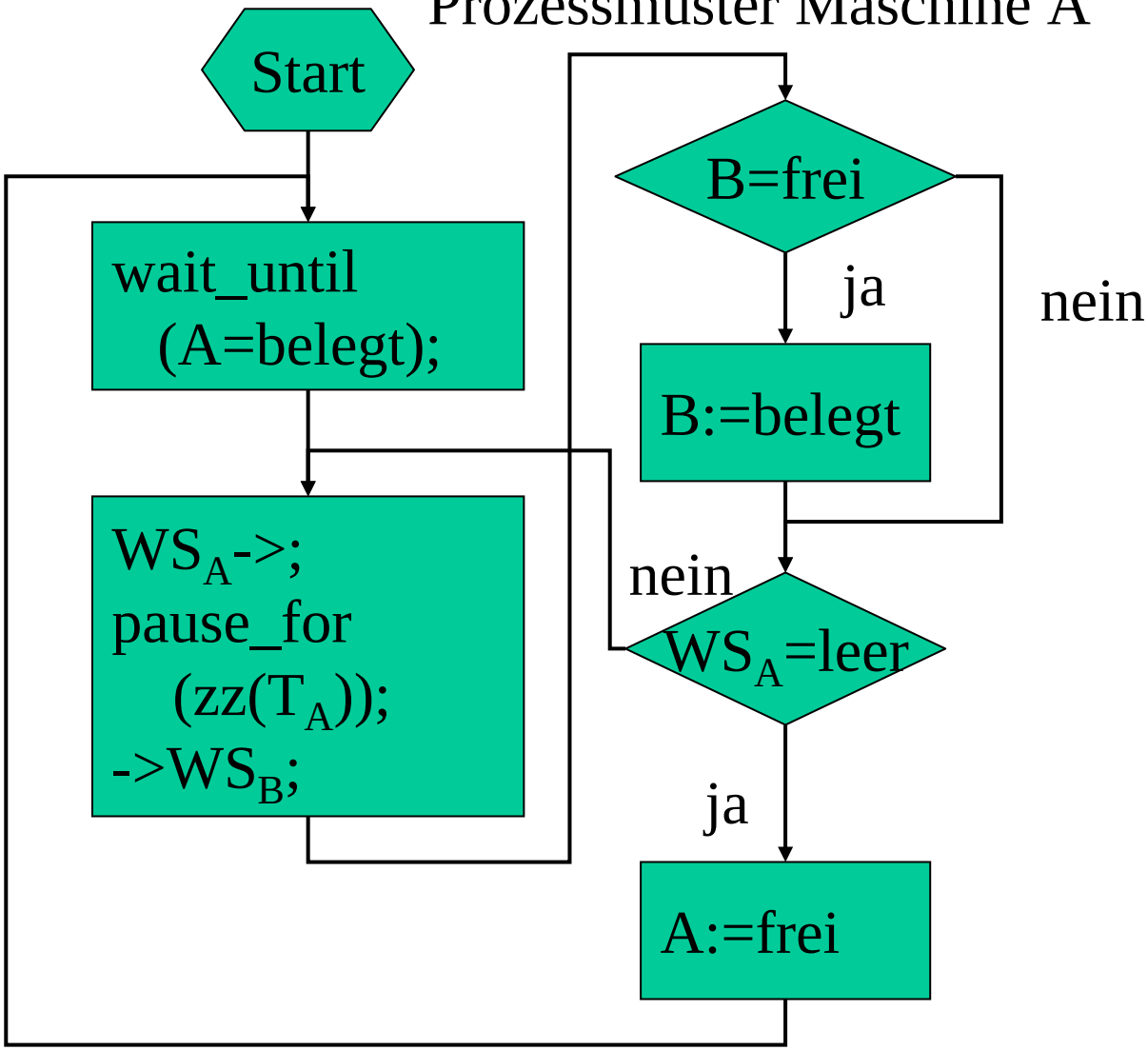
Einige Beobachtungen

- Prozess Umwelt läuft permanent
- Aufträge sind temporäre Prozesse
- Beobachtung und Speicherung von Resultatwerten muss in den Auftragsprozessen ergänzt werden
- Maschinen treten nur implizit durch ihren Zustand und die Warteräume auf

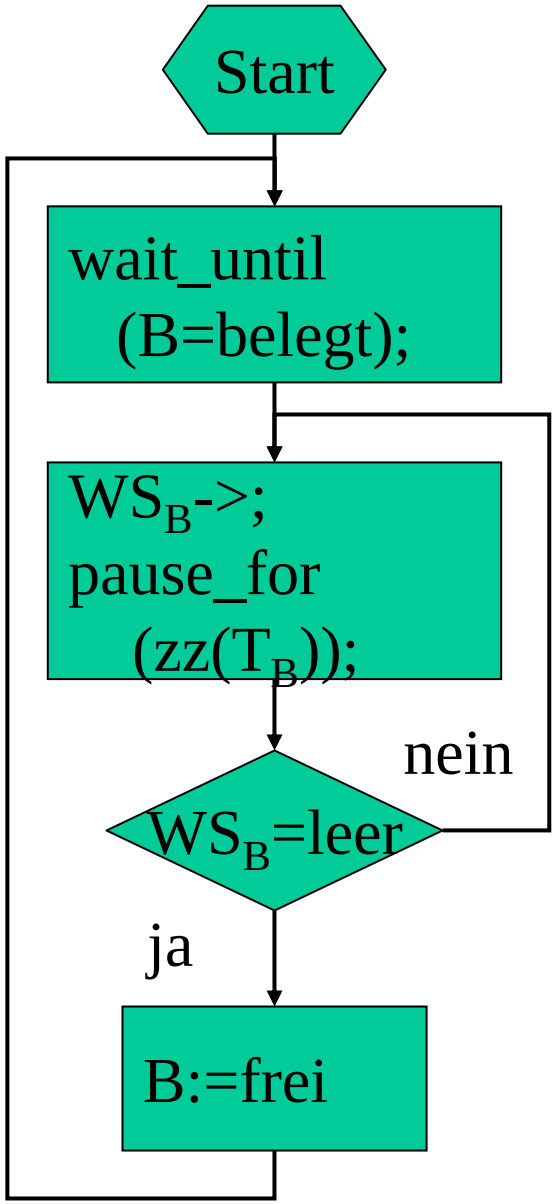
Beispiel:
(maschinen-orientiert)



Prozessmuster Maschine A



Prozessmuster Maschine B



Einige Beobachtungen

- Alle Prozess laufen permanent
- Beobachtung und Speicherung von Resultatwerten muss in den Maschinenprozessen ergänzt werden
- Aufträge treten nur implizit in den Warteräumen auf
- Start des Modells durch Erzeugung der drei Prozesse (Umweltprozess generiert den ersten Auftrag, der die wait_until-Bedingung der ersten Maschine erfüllt, ...)

Unterscheidung in maschinen- und material-orientierte Sicht
Im Beispiel beides möglich aber

maschinen-orientierte Sicht erlaubt

- unterschiedliche Scheduling-Strategien
- maschinen-orientierte Leistungsmaße (z.B. Auslastung)
- Leistungsschwankungen auf Grund von Fehlern, Ausfällen etc.

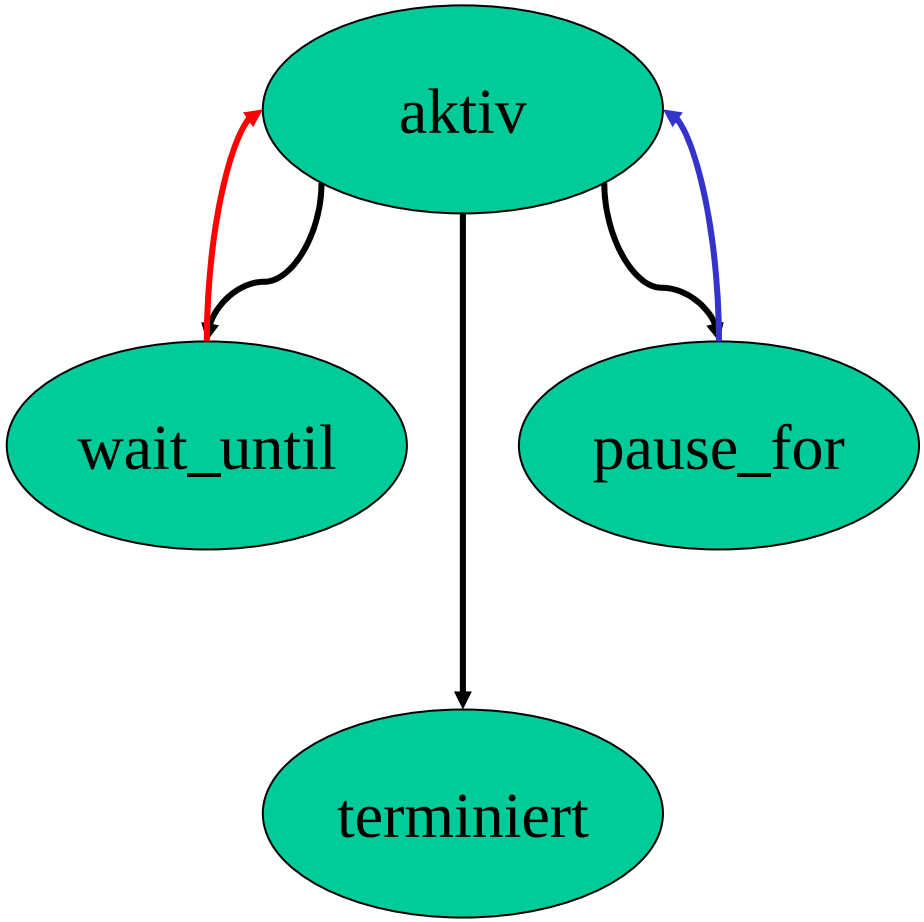
material-orientierte Sicht erlaubt

- simultane Ressourcenbelegung
- material-orientierte Leistungsmaße (z.B. Durchlaufzeiten)
- Synchronisation von Prozessen

In realen Modellen oft Kombination von maschinen- und material-orientierter Sichtweise, um Flexibilität zu erlangen.

Realisierung des process interaction-Ansatzes

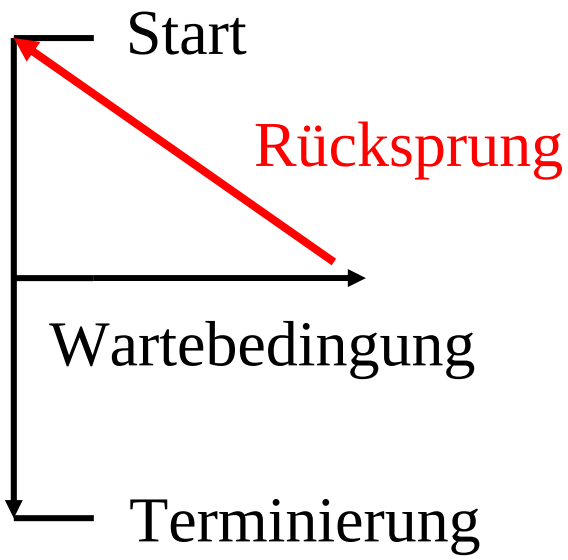
Prozesszustandsdiagramm



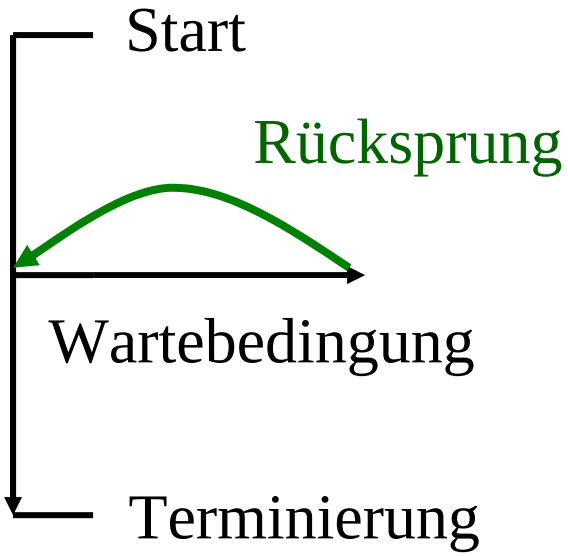
- Mehrere Prozesse laufen „parallel“, d.h. können auch zur gleichen Zeit im Zustand *aktiv* sein
- Realisierung auf Einzelprozessoren bedingt, dass nur jeweils ein Prozess aktiv sein kann (Realisierung auf Multiprozessoren deutlich komplexer, wird hier nicht betrachtet!)
- Parallelität „vorgaukeln“ Ausführungsreihenfolge zeitgleicher Prozesse kann das Verhalten beeinflussen!

Wie werden Prozessmuster beschrieben? Als Prozeduren?

Verhalten einer Prozedur



Benötigt wird aber



Realisierung in Programmiersprachen durch Koroutinen, Threads

Beispiele

- Simula67 mit Koroutinenkonzept,
- Java mit Threadkonzept,
- C ohne solches Konzept, aber Zusatzbibliotheken existieren

Ausführung gleichzeitiger Ereignisse:

Ein einfaches Beispiel mit Prozessen Schalter und Kunde:
Schalter mit Warteschlange WS, Boolscher Variable belegt und Referenz Kd auf einen Kunden
Kunde mit Boolscher Variable dran, Referenz Sc auf den Schalter und Referenz Current auf sich selbst

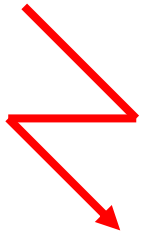
```
Schalter
Repeat
  wait_until(WS nicht leer) ;   (S1)
  Repeat
    Kd := WS.first ;            (S2)
    belegt := true ;            (S3)
    wait_until (not belegt) ;   (S4)
  Until WS leer ;               (S5)
Until (Simulationsende) ;
```

```
Kunde
dran := false ;             (K1)
Current->Sc.WS ;            (K2)
wait_until(dran) ;          (K3)
pause_for(Bedienz.) ;       (K4)
Sc.belegt := false;         (K5)
Sc.WS-> ;                   (K6)
```

Erster Versuch der Festlegung einer Semantik:

- Reihenfolge der Ereignisse eines Prozesses müssen beachtet werden
- Zeitgleiche Ereignisse in unterschiedlichen Prozessen können beliebig angeordnet werden

Wir betrachten das System bei der Ankunft des ersten Kunden K.
Eine mögliche Ereignisreihenfolge:
K1, K2, S1, S2, S3, S4, K3, K4, K5, S5, S2, S3, S4,K6



Schalter ist belegt und referenziert den Kunden, der das System bereits verlassen hat!
$\Rightarrow$ Deadlock-Situation!!

Übliche Vereinbarung in prozessorientierten Simulationssprachen:
Ein Prozess läuft so lange, bis er eine hemmende Bedingung erreicht oder terminiert.
Auswahl des nächsten Prozesses aus einer Menge aktiver Prozesse bleibt willkürlich und kann die Semantik beeinflussen!

Realisierung der Wartebedingungen

pause_for(t):

- Prozess pausiert für t Zeiteinheiten und wird danach wieder aktiv
- Ereignis Aktivierung kann für Zeitpunkt ti+t eingeplant werden
- Realisierung über Ereignisliste, die alle Prozesse enthält, die pausieren

wait_until(b):

- Bedingung b kann über beliebige Zustandsvariablen formuliert werden
- Damit kann jede Wertzuweisung jedes wait_until eines Wartenden ändern
- Damit muss nach jedem Anhalten eines Prozesses jede Wartebedingung überprüft werden

➢ Extrem aufwändig, führt zu ineffizienten Simulatoren, deshalb in kaum einem Simulationssystem in dieser Allgemeinheit verfügbar

Alternativen zum ineffizientem wait_until
Zwei mögliche, „entgegengesetzte“ Richtungen

Näher zur Implementierung
(d.h. weg vom mentalen Modell)

- passivate-Anweisung, die den aufrufenden Prozess anhält
  if not b then passivate;
- Prozess kann nur außen aufgeweckt werden
- active-Anweisung, die einen schlafenden Prozess explizit aufweckt
  activate(P);
- Simulator-Code wird länger/komplexer, aber effizienter

Näher zum mentalen Modell
(d.h. weg von der Implementierung)

- Szenario-Sprachen
  d.h. Simulation in bestimmten Problemklassen
- Warten auf bestimmte Bedingungen
  z.B. warten auf freien Bediener
- Implementierung durch Bildung von lokalen Warteschlangen für Bedingungsmengen