Algorithm Engineering

Prof. Dr. Näher

Inhaltsverzeichnis

1 Überblick **1**
1.1 Aspekte . . . 1
1.2 Literatur . . . 2

2 Datentyp **3**
2.1 Spezifikation . . . 3
2.2 Definition eines Datentyps in einer objektorientierten Programmiersprache . . . 7
2.3 Operationen → Methoden/Memberfunktionen . . . 7
2.4 Trennung von Spezifikation (Definition) und Implementierung . . . 9
2.5 Integer-Stack: dynamisch . . . 10
2.5.1 Zwei Implementierungen . . . 10
I) Array-Datenstruktur . . . 10
Einschub: Variablen, Konstruktoren, Wertzuweisungen (C++ ↔ Java) . . . 12
II) Einfach verkettete Liste . . . 14
2.5.2 Korrektheit einer Implementierung . . . 14

3 Vererbung ↔ Templates **16**
3.1 Allgemeiner: Ableitungsbaum . . . 18
3.1.1 Typverträglichkeit . . . 18
3.2 Polymorphe Datenstrukturen . . . 19
3.3 Weitere Anwendungen von Vererbung . . . 23
3.3.1 Generische Datenstrukturen . . . 23
Einfach verkettete Listen . . . 23
Templates oder Schablonen . . . 27

4 Fortgeschrittene Datenstrukturen und Algorithmen **30**
4.1 Spezifikation von Datentypen in LEDA . . . 30
4.1.1 Item-Konzept . . . 30
Anwendung auf kürzeste Wege Algorithmen . . . 33
4.2 Graphen und Graphalgorithmen (in LEDA) . . . 35
4.2.1 Operationen auf einem Graphen G . . . 35
Test, ob G azyklisch . . . 36

4.2.2 Basis-Graphtyp graph 38

5 Netzwerkflussprobleme 43
5.1 MaxFlow-Problem 43
5.1.1 Formale Definition 43
5.1.2 Das Restnetzwerk 45
Definition 45
5.1.3 Pfaderhöhung: (Maxflow) s, t 50
Labeling-Algorithmus 50

6 Algorithmische Geometrie 53
6.1 Operationen 53
6.2 Orientierung 54
6.2.1 Lineare Algebra 54
6.2.2 Minimum / Maximum in xy-Ordnung 54
6.2.3 Vergleich von Entfernungen 55
6.2.4 Anwendung von orientation (→ Schnitt-Test) 55
6.2.5 Problem: Konvexe Hülle 56
Einfacher Algorithmus: Gift-Wrapping 57
Inkrementeller Algorithmus 59
Divide & Conquer 62

1 Überblick

Das Gebiet der Algorithmen und Datenstrukturen wird traditionell der theoretischen Informatik zugeordnet. Die hier entworfenen und untersuchten Algorithmen verwenden häufig komplizierte Datenstrukturen und haben selten einen direkten Bezug zur Praxis, wo die asymptotische Komplexität der Probleme und Algorithmen und das Worst-case-Verhalten selten eine Rolle spielt. Hier sind häufig andere Aspekte, wie z. B. die effiziente Nutzung des Caches oder die leichte Implementier- und Wartbarkeit viel wichtiger. Algorithm Engineering versucht die Kluft zwischen Theorie und Praxis zu schließen. Entsprechend werden wir in dieser Vorlesung Algorithmen und Datenstrukturen betrachten, die in der Praxis tatsächlich Verwendung finden.

1.1 Aspekte

- **Korrektheit:** formal $\leftrightarrow$ konkretes Programm
  formal: theoretisch arbeitet man mit Invarianten (bzw. Beweistechniken)
  konkretes Programm: Hier ist eine andere Korrektheit gemeint, als bei der formalen. Es gibt viele Fehlerquellen, die schwierig zu erfassen sind. Durchs Testen des Programms können diese gefunden werden.

  In dieser Vorlesung wird es darum gehen, wie man möglichst korrekt vom Algorithmus zum Programm kommt.

- **Effizienz:** $\mathcal{O}$-Notation $\leftrightarrow$ praktische Laufzeit
  $\mathcal{O}$-Notation: Hier geht es zumeist um den Worst-Case. Konstante Faktoren werden zudem unterschlagen.
  praktische Laufzeit: Der Worst-Case tritt in der Praxis -wenn überhaupt- selten ein.

- **Software-Bibliotheken** (z.B. LEDA, Library of Efficient Data Types and Algorithms)sorgen für Vereinfachungen des Übergangs Algorithmen $\rightarrow$ Programm

- **Bausteine:** Datentyp
  Der Datentyp beschreibt eher die Funktion und weniger die konkrete Implementierung im Gegensatz zur Datenstruktur.

- **Gebiete:**
  - Grundlegende Datenstrukturen (Queue, Stack, Dictionairy, Priority Queue, ...)
  - Graphen und Netzwerke (kürzeste Wege, Flüsse)
  - Algorithmische Geometrie

1.2 Literatur

- Mehlhorn, Näher: LEDA, a platform for Combinatorial and Geometric Computing
- C++

2 Datentyp

Ein (abstrakter) Datentyp ($\rightarrow$ ADT) besteht aus einer Menge von Objekten (Werte bzw. Zahlen oder Punkte in der Geometrie) oder Zuständen und einer Menge/Familie von Operationen auf dieser Menge.
Eine Operation arbeitet auf einem Objekt/Zustand des Datentyps eventuell in Abhängigkeit weiterer Parameter, verändert es und liefert ein Ergebnis.

Beispiel 2.1

Datentyp	**Objekte**	**Operationen**
Integer	Ganze Zahlen ($\mathbb{Z}$)	$+, -, *, \backslash, +=, --, ++$
Stack<T>	Ein Keller von Werten vom Typ T	push, pop, top, empty
Getränkeautomat	Interne Zustände	Wirf 1€ ein und wähle Getränk ...
Stadtplan	Beschrifteter Graph	Neuer Knoten/Kante, Kürzester Weg, Zeichne

2.1 Spezifikation

Wie definiert man einen Datentyp?

- Name des Typs
- Menge der Objekte/Zustände
- Menge der Operationen
  Eine einzelne Operation wird definiert:
  - Syntax:
    - Name
    - Ein- und Ausgabetypen
  - Semantik:
    - eigentliche Funktion

Beispiel 2.2

1) Datentyp der ganzen Zahlen

Name: Integer

Objekte: $\mathbb{Z}$

Operationen: Add, Sub, Create, Gleich, ...

Add: integer $\times$ integer $\rightarrow$ integer
Semantik: Ordne jedem Paar (x, y) die Summe $x + y$ zu

Sub: integer $\times$ integer $\rightarrow$ integer
Semantik: Ordne jedem Paar (x, y) die Differenz $x - y$ zu

Gleich: integer $\times$ integer $\rightarrow$ integer
Semantik: Liefert true, falls die Parameter gleich sind, ansonsten false

Create: $\rightarrow$ integer (Konstruktor)
Semantik: liefert die ganze Zahl 0

2) Der Datentyp stack<T>

Name: stack<T>

Objekte/Zustände: Folge von Werten vom Typ T (bzw. $w \in T^*$), eventuell leer.

Operationen: create, push, pop, top, empty, size

create: $\rightarrow$ stack<T>
Semantik: Erzeugt den leeren Stack, d.h. die Folge der Länge 0.

push: stack<T> $\times$ T $\rightarrow$ stack<T>
Semantik: Fügt zweites Argument an erstes Argument der Folge an.

pop: stack<T> $\rightsquigarrow$ stack<T> $\times$ T
Semantik: Entfernt letztes (oberstes) Element der Folge und liefert es zurück
Precondition: Stack ist nicht leer (daher partielle Funktion)

3) Getränkeautomat

Automat akzeptiert 1€-Münzen

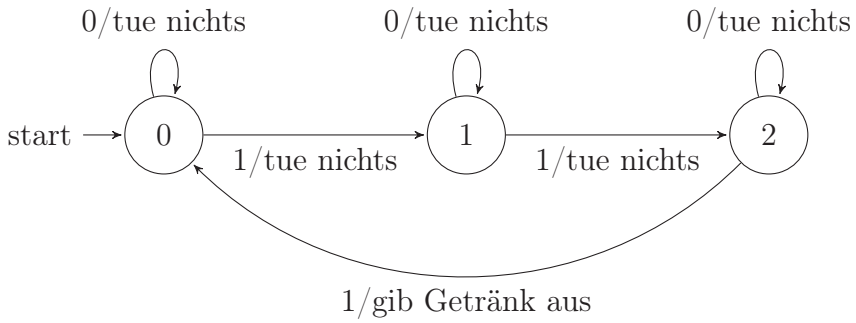
Ein Getränk kostet 3€

Operationen: Init (Reset), Akzeptiere 1€

Init $\rightarrow$ Zustand Semantik: Automat geht in Zustand 0

Akzeptiere_Euro: Zustand $\times\{0,1\} \rightarrow$ Zustand $\times$ {tue nichts, gebe Getränk aus}

Semantik: Beschreibung durch endlichen Automaten



Bemerkungen:

- Operationen können als partielle Funktion definiert sein. Man gibt dann den Definitionsbereich oft in einer Vorbedingung (Precondition) an.
  Beispiel: Stack und die Pop-Operation mit der Bedingung der Nicht-Leerheit des Stacks

- Operationen, bei der der Datentyp selbst auf der linken Seite nicht vorkommt, heißen Konstruktoren. Sie erzeugen eine neues Objekt (bzw. versetzen den Typ in einen bestimmten Zustand).
  Beispiele: create: $\rightarrow$ stack<T>
  create: int $\rightarrow$ Vektor (einer bestimmten Dimension)
  create int $\rightarrow$ stack <T> (Maximalgröße)

- Objekt- und Zustandssicht sind beide nützlich.
  - Stack/Getränkeautomat: Haben beide einen internen Zustand. Operationen können diesen ändern.
  - Integer: Hier ist die Objektsicht besser, da hier Operationen (z.B. ADD: integer × integer → integer) neue Objekte erzeugen und dabei existierende nicht geändert werden.
  - Syntax mit Parametern
    stack<T> ist ein sogenannter parametrisierter Datentyp, ein Stack mit Elementen vom Typ T (eventuell gibt es Anforderungen an den Typ T z.B. dictionary<T>: T muss eventuell linear geordnet sein ($x \leq y$)). Ein Stack hingegen braucht z.B. keine speziellen Anforderungen.
- Man kann nun eigentlich schon programmieren, obwohl über die Implementierung noch nichts bekannt ist.

Anwendung von Stack<int>

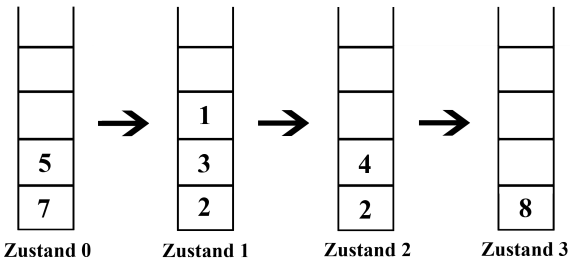
Auswertung von Postfix-Ausdrücken.
Vereinfachung:

- alle Operationen binär $+, -, *, \backslash$ (kein unäres Minus)
- Eingabe nur Zahlen 0,... ,9

Beispiel 2.3

$(7-5)*(3+1) \rightarrow 75-31+*$ (polnische Notation) Bei Stacks ist dies gut machbar, da immer nur die zwei obersten Elemente verwendet werden.

Übung: Schreiben Sie ein Programm, das diese Ausdrücke berechnet mit dem Datentyp stack<int>.



2.2 Definition eines Datentyps in einer objektorientierten Programmiersprache

Im Folgenden wurde C++ als objektorientierten Programmiersprache gewählt.

class Typename{
// Die Definition der Menge der Objekte/Zustände (als Kommentar)

private: // Deklaration von Variablen zur Darstellung der Objekte/Zustände

public: Operation 1
...
Operation k

// Kommentar: Weitere Angaben z.B. über Effizienz

2.3 Operationen → Methoden/Memberfunktionen

Syntax: Ergebnistyp Name(Arg 1, Arg 2, ..., Arg k);
z.B. void push(int x);
int pop();

Spezielle Methoden (create → Konstruktoren) haben keinen Ergebnisty.
Name = Typname
z.B. stack();
stack(int size);

in C++: Destruktoren
syntax ∼Typename();
z.B. ∼stack(); → Speicherfreigabe

Beispiel 2.4

int_stack → stack<int>

Eine Instanz vom Typ int_stack ist eine Folge von ganzen Zahlen (int). Eine Folge der Länge 0 heißt der leere Stack.

```
class int_stack{
   private: // Implementierung
   public:
      // Konstruktor
      stack(int sz); // Erzeugt einen Stack mit
            // maximaler Anfangsgröße sz.
      // Destruktor
      ~stack()
      void push(int x); // Fügt x als letztes
         // (Top-)Element an die Folge an.
      int top() const; // liefert das letzte
         //(Top-) Element
         // Precondition: Stack ist nicht leer.
      int pop(); // entfernt das letzte Element der
         // Folge und gibt es zurück.
         // Precondition: Stack ist nicht leer.
      bool empty() const; // liefert true zurück,
         // falls der Stack leer ist, sonst false
};
```

`const` ist ein Schlüsselwort. Steht es hinter einer Funktion, so gibt es an, dass dies eine Funktion ist die den Zustand des Objekts, für das sie aufgerufen wird, nicht ändert.

2.4 Trennung von Spezifikation (Definition) und Implementierung

Im Gegensatz zu Java existiert in C++ eine Trennung zwischen Spezifikation und Implementierung. Hier werden die Deklarationen einer Methode (ohne Rumpf) in einer speziellen Header-Datei (z.B. int_stack.h) angegeben. Die Implementierung (dh. der C++-Code der Methoden) wird in einer anderen Datei (z.B. int_stack.cpp) angegeben. In dieser wird über #include „int_stack.h“ die Header-Datei eingebunden.
In C++ ist es, im Gegensatz zu Java, möglich mehr als eine Klasse pro Datei zu implementieren.
Diese und weitere Unterschiede zwischen Java und C++ entstehen durch die unterschiedlichen Ziele der beiden Programmiersprachen. So ist beispielsweise Effizienz ein wichtiges Ziel von C++ im Gegensatz zur Sicherheit, die ein wichtiges Ziel in Java ist.

Beispiel 2.5

C++ (int_stack.h)

```
class int_stack{
   private int* A; // Feld
   int sz; // Länge von A
   int t; // Pointer (int*) auch möglich
}
```

C++ (int_stack.cpp)

```
#include "int_stack.h"
// Konstruktor
int_stack::int_stack(int n){
   sz = n;
   A = new int[sz];
   t = -1; // Stack leer
}
```

Java

```
private int[] A;
int t;
```

2.5 Integer-Stack: dynamisch

2.5.1 Zwei Implementierungen

Es gibt mehrere Möglichkeiten die Klasse int_stack zu implementieren.

I) Array-Datenstruktur

(int_stack.h)

```
class int_stack{
   public:
      // Konstruktor
      int_stack(); // Erzeugt leeren Stack
      // Destruktor
      ~int_stack();
   private:
      int* A;
      int sz; // Anfangsgröße. Kann später
            //vergrößert werden.
      int t;
      void push(int x);
}
```

Implementierung der Operationen

(int_stack.cpp)

```
int_stack::int_stack(){
   sz = 2; // Am Anfang Feld der Länge 2
   A = new int[sz];
   t = -1;
}
int_stack::~int_stack(){
   delete[] A; // Speicherfreigabe
}
void int_stack::push(int x){
   if(t == sz-1){ // Stack ist voll
      int* B = new int[2*sz];
      sz = 2*sz;
   }
   for(int i=0; i<=t; i++){
```

```
        B[i] = A[i]; // Werte kopieren
    }
    delete[] A;
    A = B;
    A[++t] = x; // Eigentlicher push
}
int int_stack::pop(){
    if(t == -1){
        EXCEPTION 'Pop auf leerem Stack';
    }
    if(t == sz/4){
        int* B = new int[sz/2];
        sz = sz/2;
    }
    for(int i=0; i<=t; i++){
        B[i] = A[i]; // Werte kopieren
    }
    delete[] A;
    A = B;
    }
    return A[t--];
}
```

Bemerkungen

- Der Destruktor wird automatisch aufgerufen, wenn eine Variable nicht mehr gültig ist.
- In Java löscht der Garbage Collector eine Variable automatisch, wenn er merkt, dass sie nicht mehr genutzt wird.
- Push-Operation: Verdopple das Feld, falls es zu klein wird.