

Algorithm Engineering

June 22, 2015

Datentypen

Getränkeautomat

Automat akzeptiert 1E, ein Getränk kostet 3E

Operatoren:

1. Init(Reset)
2. Akzeptiere1E

Init \rightarrow Zustand

Semantik: Automat geht in Zustand 0

Akzeptiere1E: ZustandX{0,1} \rightarrow ZustandX{tue nichts, gib Getränk}

Semantik: Beschreibung durch einen endlichen Automaten.

Stadtplan

Übung 1

Bemerkungen

- Operatoren können partiell Definiert sein. Man gibt Definitionsbereich oft in einer Vorbedingung an.
- Operatoren, bei denen der Datentyp selbst auf der linken Seite nicht vorkommt, heißen Konstruktoren. Sie erzeugen ein neues Objekt (bzw. versetzen den Typ in einem bestimmten Zustand).
 - Create: $\rightarrow \text{stack}<T>$
 - Create: $\text{int} \rightarrow \text{vector}$ (Vektor bestimmter Dimension)
- Objekt- und Zustandssicht sind beide nützlich. Stack/Getränkeautomat haben internen Zustand, Operatoren können ihn verändern.
 - Integer: Objektsicht besser, Operatoren erzeugen neue Objekte, existierende werden nicht geändert.
- $\text{stack}<T>$ ist ein parametrisierbarer Datentyp: Stack mit Elementen vom Typ T. Hat eventuell besondere Anforderungen an Typ T, z.B. $x \leq y$ in Dictionaries.
- Man kann nun eigentlich schon programmieren, obwohl über die Interpretierung noch nichts bekannt ist.

Anwendung von $\text{stack}<T>$

Auswertung von Postfix-Ausdrücken

Vereinfachungen: alle Operatoren binär (+-*/), Eingabe nur Zahlen 0-9

Bsp.: $(7 - 5) * (3 + 1) \rightarrow 75 - 31 + *$

Defintion eines Datentyps

(In einer Objekt-Orientierten Programmiersprache)

```
class Typname {  
    //Definition der Menge der Objekte bzw Zustaende  
    private: //Deklaration von Variablen zur Darstellung der Objekte/Zustaende  
    public: //Operatoren  
    //Kommentare z.B. ueber Effizienz  
};
```

Operatoren

Methoden/Memberfunktionen

Syntax: Ergebnistyp Name(Argumente...);

Spezielle Methoden:

- Kein Ergebnistyp: stack(); stack(size);
- Destruktor: ~ Typname();

Beispiel

int_stack \rightarrow stack< T >

```
class int_stack {  
    /* Eine Instanz vom Typ int_stack ist eine Folge von ganzen Zahlen (int). Eine Fol  
    private: //Implementierung  
    public: stack(int sz); //Konstruktor  
    //Erzeugt einen Stack mit maximaler Groesse sz  
    ~stack() //Destruktor  
    void push (int x);  
    //fuegt x als letztes Element (top) an die Folge an.  
    int top() const;  
    //liefert das letzte (top) Element  
    //Precondition: Stack nicht leer  
    int pop();  
    //entfernt letztes (top) Element der Folge und gibt es zurueck  
    //Precondition: Stack nicht leer  
    bool empty() const;  
    //true, wenn Stack leer, false sonst.
```

In c++ Spezielle Header Datei, die die Deklarationen ohne Rumpf enthält. Implementierung in .cpp

Implementierung der Klasse int_stack

Mehrere Möglichkeiten: Array, Liste Array Implementierung: *int_stack.h*

```
class int_stack {  
    private  
        int* A; //Feld  
        int sz; //Laenge von A  
        int t;  
};
```

int_stack.cpp

```

#include "int_stack.h"
int_stack::int_stack(int n) {
    sz=2;
    A = new int[sz];
    t = -1; //leer
}
int_stack::~~int_stack(){
    delete[] A;
}
void int_stack::push(int x){
    if (t == sz-1){
        //stack voll
        int* B=new int[2*sz];
        sz ← 2*sz;
        for(int i=0; i ≤ i++){
            B[i] ←
        }
        delete[] A;
    }
    A[++t] ← x; //eigentlich push
}
int int_stack::pop(){
    if (t==0){
        EXCEPTION(" Leerer Stack")
    }
    return A[t--];
}

```

Einschub: Variablen, Konstruktoren, Wertzuweisung

ablen Deklaration c++: Aufruf des Konstruktors generiert ein Objekt.

Java: Erst eine Referenz erstellen, dann ein Objekt generieren und auf dieses verweisen.

Wertzuweisung c++: $int_stack\ s1, s2; s1 = s2;$ Objekt wird kopiert, es gibt 2 Objekte.

Java: $int_stack\ s1, s1; s1 = s2;$ Referenzen zeigen auf ein einziges Objekt.

semantik in c++: Verwendet Pointer auf ein Objekt.

Test auf Gleichheit (==) Operator

Parameterübergabe sind Pointer

semantik in Java: Parameter by Value, gesamtes Objekt kopiert und dann übergeben.

Korrektheit einer Implementierung

(Hier der Array-Implementierung von int_stack)

Eigentlich 2 Datentypen:

1. der abstrakte Datentyp int_stack
2. der konkrete Datentyp Array

Abstrakter Zustand: Folge von int's

Konkreter Zustand: Werte der Variable A,t,sz

Wir garantieren (Invariante), dass nicht die Kombination von A,t,sz möglich sind, sondern nur gültige Zustände mit:

1. A ist ein Feld der Länge sz
2. $-1 \leq t \leq sz - 1$

Sei Z =Menge der konkreten Zustände und S = Menge der abstrakten Zustände

Um die Korrektheit zu zeigen, definieren wir eine Abbildung $F : Z \rightarrow S$

$(A, sz, t) \rightarrow \begin{cases} Folge\ A[0], \dots, A[t] \text{ falls } t \geq 0 \\ Leere\ Folge, t = -1 \end{cases}$ Und zeigen:

1. Konstruktoren erzeugen gültige konkrete Zustände
2. Für jede abstrakte Operation und die dazugehörige konkrete Operation f_{op} zeige $F(f_{op}(Z)) = op(F(Z))$
Bsp.: push: $S \times int \rightarrow S$
 $f_{push} : Z \times int \rightarrow Z$

Kommutatives Diagramm:

$$\begin{array}{ccc} z & \xrightarrow{F} & s \\ \downarrow f_{op} \uparrow op & & \\ z' & \xrightarrow{F} & s' \end{array}$$

Vererbung/Generische Datentypen

Templates/Wiederverwendung von Code

Situation

Man braucht einen Datentyp A, der sehr ähnlich zu einem bereits vorhandenen definierten Typ B ist.

A soll:

- einen Teil der Daten/Operationen verwenden
- andere Daten/Operationen anfügen
- einige verändern (auf andere Weise implementieren)

Beispiel

Es existiert die Klasse Polygon (B), implementiert werden soll eine Klasse Rechteck (A).

A ist eine Spezialisierung von B

Rechteck könnte alle Polygon-Operationen (draw(), translate etc)

Einige Operationen können effizienter implementiert werden (Flächeninhalt etc.)

Andere sind nur für Rechtecke definiert.

Jedes Rechteck ist ein spezielles Polygon.

Allgemein

Wir leiten die Klasse A von der Klasse B ab.

Syntax in c++: A: public B

```
class Rechteck: public Polygon{
    //Konstruktoren -> uebung :(
private:
    //Ueberschreiben
    double b,h;
    double area() {return b*h;}
    double umfang() {return 2*(b+h);}
    //Hinzufuegen
    double ratio() {return b/h;}
};
```

Ableitungen auch als Baum darstellbar (Shape mit Kindern Polygon/Ellipse/Punkt etc)

Typverträglichkeit

Einer Variable vom Typ B* oder B& (alle Variablen in JAVA vom Typ B) kann ein Objekt (Pointer/Referenz) vom Typ A zugewiesen werden.

Alle im Ableitungsbaum erreichbaren Typen können zugewiesen werden (Kinder).

Eine Variable hat 2 Typen: Einen Statischen Typ, der zur Compilezeit bekannt ist. Dynamischer Typ, der zur Laufzeit bekannt ist.

Polymorphe Datenstrukturen

```
polygon* p = new rechteck()
double func(polygon& p){
    return p.area();
}
rechteck rect = new rechteck();
func(rect);
```

Es wird per dynamischer Bindung die Funktion func der Klasse Rechteck aufgerufen.

C++ benutzt standardmäßig statische Bindung es sei denn Funktion ist "virtual". Bsp.: Feld von Polygonen

c++: Polygon** (Ein Pointer auf ein Feld von Pointern) A = new Polygon*[100];

A[0] = new Polygon(...);

A[1] = new Rechteck(...);

Abstrakte Klassen werden verwendet um Interfaces zu definieren.

Z.B. Shape als abstrakte Klasse. In C++ werden Methoden als abstrakt markiert, wenn sie bei der Deklaration = 0 gesetzt werden

virtual void draw() = 0;

Anwendung auf Algorithmen

Lineare Ordnungen durch ein Interface umgesetzt.

In C++:

```
class comparable{
    virtual int compare(comparable x)=0;
}
```

Anwendung in generischen Sortieralgorithmen:

Quicksort(comparable* A[])

Zum Vergleich benutzt man x.compare(y)

Anwendung: Sortiere ein Feld von point

```
class point: public comparable{
    int compare(comparable x){
        //x ist tatsaechlich ein point
        double px=((point&)p).x; //Casting
        double py=((point&)p).y;
        //lexikographische Ordnung..
        return ...;
    }
}
```

Aufruf von Quicksort:

point* A[] = new point*[100];

for (i=0; i<100; i++) A[i] = new point(i, i*i);

Quicksort(A, 100);

Datenstrukturen, bei denen Comparable sinnvoll ist:

- Binäre Suchbäume, bei denen die Knoten vergleichbar sind.

Weitere Anwendung von Vererbung

Generische Datenstrukturen wie z.B. Listen von beliebigen Objekten.

Einfach verkettete Liste:

Beobachtung: Implementierung der Operationen (push, pop), ist nicht abhängig vom Typ. Der Wert (int,string,point,...) jedoch schon.

Abstraktion: Liste ohne Werte

1. Basisklasse für allgemeines Listenelement:

```
class slist_element{
    slist_element* next;
    slist_element(slist_element* p) {next = p;}
};
```

2. Basisklasse für allgemeine List:

```
class slist{
    slist_element* first;
    slist() {first = NULL;}

    void push(slist_element* p){
        p->next = first;
        first = p;
    }
    slist_element* pop(){
        if(first == NULL) return first;
        slist_element* p = first;
        first = first->next;
        return p;
    }
}
```

slist funktioniert auch für alle von slist_elem abgeleiteten Klassen.

Besondere Elemente werden als neue Klassen definiert, die von slist_element erben.

Ein "Point" ist ein "slist_element"

```
slist L;
point* p = new point(x,y);
L.push(p);
```

slist ist Polymorph, es kann als Liste verschiedener Datentypen dienen.

Situationen in denen diese Polymorphie vorteilhaft ist: Grafik-Editor:

```
void drawAll() //Iteriere ueber Liste und rufe draw fuer alle auf
forall x in scene //scene ist die Liste
    x -> draw();
```

Falls wir eine Liste von einem bestimmten Objekt-Typ verwenden wollen (z.B. point_list) wird diese von slist abgeleitet.

```
class point_list: public slist {
    //neues Interface das nur points erlaubt
    void push(point* p) {slist::push(p);}
    point* pop(){return (point*) slist::pop();}
```

```

        //Das Casting ist sicher , da durch push sicher nur points in der Liste sind.
    }

```

Aufwändige Datenstruktur: Balancierte Suchbäume (z.B. AVL)

1. Klasse für die Knoten (benötigt parent, left, right)

```

class bin_tree_node{
    bin_tree_node* left , right , parent;
};

```

2. Klasse für den Baum:

```

class bin_tree{
    virtual int cmp(bin_tree_node* p, bin_tree_node* u) = 0
    //Bsp.: cmp(p,q) = {-1, wenn p<q; 0, wenn p = q; 1, wenn p>q}
    void insert(bin_tree_node* p){
        //fuegt p in den Baum ein , verwendet cmp als Vergleich
    }
    bin_tree_node* lookup(bin_tree_node* p){
        /* in Schleife:
        if cmp(q,p) > 0 q=q->left;
        else q=q->right;
        */
    }
}

```

Anwendung auf Point:

```

class point:public bin_tree_node{
    ...
}
class point_bin_tree:public bin_tree{
    //Definiere cmp Funktion
    int cmp(bin_tree_node* p, bin_tree_node* q){
        point* a = (point*)p;
        point* b = (point*)q;
        if(a->x < b->x) return -1;
        if(a->x > b->x) return 1;
        if(a->y < b->y) return -1;
        if(a->y > b->y) return 1;
        return 0;
    }
    void insert(point* p){
        bin_tree::insert(p);
    }
    point* min() {return (point*)bin_tree::min();}
}

```

Templates

Funktionstemplates

template <class T>

Beispiel: swap(T& x, T& y) Vertauscht den Inhalt der beiden Variablen

```

swap(T& x, T& y){
    T tmp = x;
    x = y;
}

```

```

        y= tmp;
    }

```

Implementierung ist unabhängig von T.

Klassentemplates

```

template<class T>
class stack{
    T* A; //Feld von T's
    int sz;
    int t;

    public
    void push(T x){...}
    T pop {...}
}

```

Beispiel für mehrere Typen:

```

template<class K, class I>
class dictionary{
    //Woerterbuch mit Schluessel vom Typ K und werte vom Typ I
    void insert(K k,I i){...}
    I translate (K key){...}
}

```

Anwendungsbeispiel: Word-Count, zählt wie oft einzelne Wörter in einem Text vorkommen. `dictionary<string,int? >`
D; Speichert Wort als Schlüssel, Häufigkeit als Wert.

Fortgeschrittene Datenstrukturen und Algorithmen

LEDA: Library of Efficient Datatypes and Algorithms

Plattform: Algorithmus → Programm

Datentypen: Listen, Stacks, Dictionaries, Priority Queue

Efficient: Datenstrukturen

Einfache Benutzung (Pseudocode soll leicht in C++ umsetzbar sein)

Korrektheit: Datentypen (Definition), Program Checker

Weitere Themen: Graph-Datenbanken, -Algorithmen, Geometrie

Spezifikationen von Datentypen in LEDA

Item-Konzept: Viele Datentypen sind Definiert als Menge von Items.

Item: Zugriff über Parameter (Abstraktion von den Begriffen Pointer, Referenz, Index)

Bsp.: `dictionary< string,int >` D speichert Paare aus Schlüsseln (string) und Informationen (int)

Definition: D ist eine Menge von Items (dictionary-Items)

Operationen:

- `D.insert(string s, int i)`: Falls D kein Item mit Schlüssel s enthält, füge Item (s,i) ein und gib es zurück.
Sonst: Ändere Datenwert i und liefere es zurück.
- `D.lookup(strin s)`: liefert das Item mit dem Schlüssel s, falls es nicht existiert, Null.

2. Beispiel: Priority Queue

`priority_queue < P,I > PQ`

P: Priorität z.B. Zahl, I: Information (z.B. Knoten eines Graphen)

Definition: Menge von Items

Operationen: insert, findMin (minimale Priorität), prio (setze Priorität), inf (setze Information), delmin, decrease_P

Dijkstra Algorithmus

Eingabe: Graph $G = (V, E)$, Kostenfunktion $cost = E \rightarrow int^+$, Startknoten $s \in V$

Ausgabe: Distanzfunktion $dist : V \rightarrow int^+$, $dist(v)$ = Kosten eines billigsten Pfades von s nach v.

Kosten eines Pfades: Summe der Kanten.

Idee von Dijkstra:

- Überschätze Distanzfunktion: 0, falls $s=v$, inf, falls $s \neq v$.
- Kandidatenlist U: Menge aller Knoten, aus deren Kanten ausgehen können, die eine Abkürzung darstellen.
- Wähle jeweils $u \in U$ mit $dist(u)$ minimal
- Beobachtung: $dist(a)$ ist korrekt.
- Durchlaufe alle aus u ausgehenden Kanten und überprüfe Dreiecksungleichung, reduziere Distanz von v

Kann effizient mit Fibonacci Heap realisiert werden.

Graphalgorithmen in LEDA

Der Datentyp Graph

Dient zur Erstellung von gerichteten Graphen $G=(V,E)$ mit $E \subseteq (V \times V)$

Arten von Objekten

Operationen auf einem Graphen G:

Access Operationen:

node G.source(Edge e): Von welchem Knoten geht die Kante e aus

node G.target(Edge e): Zu welchem Knoten geht die Kante e

int G.outdeg(node v): Ausgangsgrad

int G.indeg(node v): Eingangsgrad

list<edge>G.out_edges(node v)

Update Operationen:

node G.new_node()

edje G.new_edge(node v, node w)

void G.del_edge(edge e)

void G.del_node(node v) (entfernt v und alle Kanten)

Iterationen (Laufvariable wird extern deklariert, weil weil):

forall_nodes(v,G)

forall_edges(e,G)

forall_out_edges(e,v)

forall_in_edges(e,v)

Beispiel: Iteration über alle Nachbarknoten von v:

```
forall_out_edges(e,v){
    node w = G.target(e)
}
```

Beispiel: Teste, ob G azyklisch

Idee: Siehe topologisches Sortieren:

Solange ein Knoten v existiert mit $indeg(v)=0$, entferne ihn und alle ausgehenden Kanten.

Falls G leer, dann ist der Graph azyklisch.

```

zero <- {v ∈ V | indeg(v) = 0}
while zero ≠ ∅ do
    u <- beliebiger Knoten aus zero
    zero <- zero \ {u}
    forall v ∈ V mit (u,v) ∈ E
        entferne (u,v) aus G
        if indeg(v)=0 then
            zero <- zero ∪ {u}
        fi
    od
    entferne u aus G
od

```

Als C++ Programm (oder auch nicht... Näherliegende Gründe):

```

bool isAcyclic (graph G) //Call by Value, Graph wird kopiert
{
    stack<node> zero;
    node v;
    forall_nodes(v,G){
        if (G.indeg(v)==0) zero.push(v);
    }
    while (!zero.empty()){
        node u = zero.pop();
        edge e;
        forall_out_edges(e,u){
            node v = G.target(e);
            G.del_edge(e);
            if (G.indeg(v)==0) zero.push(v);
        }
    }
    return G.number_of_edges()==0;
}

```

Informationen (Daten) mit den Knoten und Kanten speichern.

Parametrisierter Graph

Netzwerk: Knoten stehen für Objekte vom Typ *vtype*, Kanten *etype*.

Unterklasse vom gegebenen graph

GRAPH < *vtype*, *etype* > *G*;

Node/Edge Array

node_array < *T* > *A*(*G*) Mit *G* einem Graph

edge_array < *T* > *B*(*G*)

Z.B. Kürzester Weg: DIJKSTRA(graph *G*, *edge_array* < *int* > *cost*, node *startNode*, *node_array* < *int* > *distance*);

Einige Ausgabedaten werden getrennt vom Graphen übergeben.

Erstes konkretes Beispiel: Dijkstra

```

void DIJKSTRA (graph& G, node s, edge_array<int>& cost, node_array<int>& dist)
{
    p_queue<node, int> PQ;
    node_array<pq_item> I(G);
    dist[s] = 0;
}

```

```

I[s] = PQ.insert(s, 0);
node v;
forall_nodes(v, G){
    if (v!=s) dist[s] = MAXINT;
}
While (!PQ.empty()){
    node u = PQ.delmin();
    edge e;
    forall_out_edges(e, u){
        node w = G.target(e);
        int d = dist[u]+cost[e];
        if (d<dist[w]){ //Dreiecksungleichung
            if (dist[w]==MAXINT) //Wurde schon besucht?
                I[w] = PQ.insert(w, d)
            else PQ.decrease_p(I[w], d)
            dist[w] = d;
        }
    }
}

```

Analyse: delmin, insert wird n mal ausgeführt. decrease_p wird m mal (für alle nodes und alle deren Kanten) ausgeführt.

Ergebnis: $O((m+n)\log n)$

Korrektheit:

Idee: Füge Programmcode hinzu, der für eine konkrete Testeingabe überprüft, ob das Ergebnis korrekt ist.

Teste am Ende, ob für jede Kante die Dreiecksungleichung erfüllt ist.

```

edge e;
forall_edges(e, G){
    node v = G.source(e);
    node w = G.target(e);
    ASSERT(dist[v] + cost[e] >= dist[w]);
}

```

Netzwerkflussprobleme: Maxflow

Graph: Transportnetzwerk

Jede Kante besitzt eine Kapazität (wie viel pro Zeiteinheit über diese Kante transportiert werden kann).

Probleme:

1. Maximiere den Transport von einem Knoten s (Source) zu einem Knoten t (Senke)
2. Minimiere die Transportkosten (min cost flow)

Eingabe: Graph $G = (V, E)$, Knoten $s, t \in V | s \neq t$, Kapazitäten $u \rightarrow \mathbb{R}_0^+$

Schreibweise: u_{ij} für $(u : u(i, j))$

Ergebnis: Flussfunktion $x : E \rightarrow \mathbb{R}_0^+$ mit:

1. $\forall (i, j) \in E : 0 \leq x_{i,j} \leq u_{i,j}$ Kapazitätsbedingung
2. $\forall i \in V \setminus \{s, t\} : \sum_{j \in V | (i,j) \in E} x_{i,j} = \sum_{k \in V | (k,i) \in E} x_{k,i}$ (Die Menge, die aus i rausgeht, muss auch in i reinkommen).

Gesucht ist ein Fluss x mit $\sum_{i \in V | (s,i) \in E} x_{s,i} - \sum_{j \in V | (j,s) \in E} x_{j,s}$ maximal.

$\forall i \in V \setminus \{s, t\} \delta(i) = 0$ maximiere $\delta(s)$

Beobachtung: $\delta(t) = -\delta(s)$

Idee für einen Algorithmus: Erhöhende Pfade

- Starte mit allen $x=0$ ($\forall (i, j) \in E : x_{i,j} \leftarrow 0$)

- Erhöhe x entlang von Pfaden von s nach t .

Definition Restnetzwerk $G(x)$ (hängt vom aktuellen Fluss x ab), beschreibt die Möglichkeiten x zu verändern.

Erhöhende Pfade Pfad P von s nach t in $G(x)$. Erhöhung von $\delta = \min\{r_{i,j} \mid (i,j) \in P\}$

Achtung Der reale Pfad in G hat eventuell Kanten in Gegenrichtung. Korrektheit x ist immer ein Fluss.

Terminiert mit Fluss x , sodass in $G(x)$ kein Pfad von s nach t (bitte was?) Implementierung

- 1 Keine explizite Darstellung von $G(x)$, stattdessen Navigiere im Originalgraphen, verwende ausgehende Kanten, wenn $u_{i,j} > x_{i,j}$, eingehende Kanten, wenn $x_{i,j} > 0$.
- 2 Berechnung eines erhöhten Pfades: Verwende DFS oder BFS vom Knoten s aus, und berechne die Kanten wie in 1. beschrieben. Stoppe, wenn t erreicht.
- 3 Darstellung des Pfades durch `pred_Verweise` (Vorfahrverweise).