Algorithm Engineering

Grundlagen

Operatoren/Methoden/Memberfunktionen
Konstruktor: Kein Ergebnistyp: stack();
Destruktor: ~Typname();

```
Class Typname{
        private:
        public:

};
```

Int stack:

```
Class int_stack{
        private:
                int* A
                int sz
                int t
        public:
                stack(int sz)
                ~stack()
                void push(int x)
                int top() const
                int pop()
                bool empty() const
}
```

Variablen Deklaration: c++ (Konstruktor generiert Objekt), Java (Objekt und Referenz darauf erstellen)
Wertzuweisung: c++ (s1 = s2 : Objekt wird kopiert), Java (s1=s2 : Referenzen zeigen auf das gleiche Objekt)

Korrektheit einer Implementierung

Abstrakter Datentyp: int_stack
Konkreter Datentyp: Array
Abstrakter Zustand: Folge von int’s
Konkreter Zustand: Werte A, t, sz

Nur Kombinationen von A,t,sz die gültige Zustände darstellen erlaubt
A ist Feld der Länge z
$-1 \leq t \leq sz - 1$
Z=Menge konkreter Zustände, S=Menge abstrakter Zustände

$$F: Z \to S: (A, sz, t) \to \begin{cases} Folge\ A[0], \dots, A[t] falls\ t \geq 0 \\ Leere\ Folge\ t = -1 \end{cases}$$

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

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

Vererbung

Teile der Daten/Operationen verwenden
neue Daten/Operationen anfügen
Daten/Operationen ändern
-> Vererbung

Typenverträglichkeit:

Wenn A von B erbt: Variable vom Typ B kann auch ein Objekt A zugewiesen werden.

Polymorphe Datenstrukturen:

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

Listen:

```
Class slist_element{
	slist_element* next;
	slist_element(slist_element* p) {next = p}
};
class slist{
	slist_element* first;
	slist() {first = NULL;}
	void push(slist_element* p);
	slist_element* pop();
};
```

Vererbe von slist_element um eigene Liste zu implementieren

Funktionstemplates:

Template<class T>

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

```
Template<class T>
class stack{
	T* A;
	int sz;
	int t;
	public
	void push(T x)
	T pop
}
```

Djikstra 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
Idee:

Überschätze Distanzfunktion: 0, falls s=v, infinity, falls $s \neq v$
Kandidatenliste U: Menge aller Knoten, aus deren Kanten ausgehen könnten, 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

```
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;
                }
        }
```

Laufzeit: O(m+n)log n), m: decrease_p Ausführungen, n: delmin Ausführungen

Maxflow

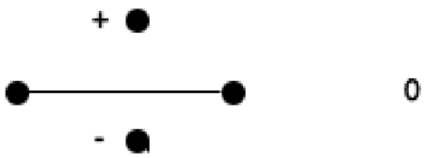
Jede Kante im Transportnetzwerk hat eine Kapazität
Problem: Maximiere Transport von s nach t
Eingabe: $G = (V, E); s, t \in V | s \neq t; Kapazitäten: u \rightarrow \mathbb{R}_0^+$
$u_{ij} = u(i, j)$
Ergebnis: Flussfunktion $x: E \rightarrow \mathbb{R}_0^+$:
1. $\forall (i, j) \in E: 0 \leq x_{ij} \leq u_{ij}$ Kapazitätsbedingung (Fluss ist kleiner gleich Kapazität)
2. $\forall i \in V \setminus \{s, t\}: \sum_{j \in V | (i,j) \in E} x_{ij} = \sum_{k \in V | (k,i) \in E} x_{ki}$ (In i rein = aus i raus)

Gesucht ist Fluss x mit $\sum_{i \in V | (s,i) \in E} x_{si} - \sum_{j \in V | (j,s) \in E} x_{js}$ maximal
$\forall i \in V \setminus \{s, t\}: \delta(i) = 0, maximiere\ \delta(s)$
Beobachtung: $\delta(t) = -\delta(s)$
Restnetzwerk: G(x) Graph wie viel Kapazität auf Kanten verbleibend ist
Das gilt in beide Richtungen, es kann mehr aber auch weniger (Rückfluss) transportiert werden

Erhöhende Pfade:
- Start mit allen $x = 0$ $(\forall (i, j) \in E: x_{ij} = 0)$
- Erhöhe x entlang von Pfaden von s nach t
- Anzahl der Erhöhung: $\delta = \min\{r_{ij} | (i, j) \in P\}$ mit Pfad P im Restnetzwerk
  Weil: Das Minimum ist Bottleneck, um so viel kann maximal erhöht werden

Geometrische Algorithmen

Orientation:



Konvexe Hülle:
Eingabe: Liste von Punkten (im 2D Raum)
Ausgabe: Kleinstes konvexes Polygon P das alle Punkte enthält
Als Liste von Punkten gegen den Urzeigersinn sortiert
Konvex: Alle Winkel von P sind > 180°, damit liegt jede Strecke zwischen zwei Punkten vollständig in P
Algorithmus: Gift-wrapping

```
list <point> convexHull(list<point> &L){
        list<point> P;
        point q0 = L.first();
        point p;
        forall(p in L){ if(p.cmp_xy(q) == -1) q0=p;}
        P.append(q0);L.del(q0);
        while true{
                point q = L.first()
                forall(p in L){
                        if P.last().orientation(q,p) == -1 ||
                                (P.last().orientation(q,p)==0 && P.last().cmp_dist(q,p)==-1)
                                q=p
                }
                L.del(q);
                if q==q0 {break} else {P.append(q)}
        }
return P;
```