```
# define
   forall_items(it,D) // D ist die Datenstruktur
      for(it = D.first_item()){
         if(it != NULL)
            it = D.next_item(it)
      }
```

Eine mögliche Anwendung wäre das Dictionary, bei dem alle Schlüssel (dic_it) ausgegeben werden sollen.
In der Graphtheorie sind Makros zum Beispiel interessant, da sehr oft über Adjazenzen, Kanten oder ähnliches iteriert wird.

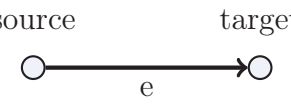
4.2 Graphen und Graphalgorithmen (in LEDA)

Der allgemeine Graph-Datentyp in LEDA: „*graph*“
Der Datentyp repräsentiert gerichtete Graphen.
Ein Graph G besteht aus zwei Typen von Objekten ($\rightarrow$ Items):

Knoten vom Typ „*node*“

Kanten vom Typ „*edge*“

Mit jedem Knoten v sind zwei Listen vom Typ $list < edge >$ assoziiert:
out_edges und in_edges.
Eine Kante verläuft zwischen genau 2 Knoten.



4.2.1 Operationen auf einem Graphen G

- Access

  $node\ G.source(edge\ e);$
  $node\ G.target(edge\ e);$
  $int\ G.outdeg(node\ v);$
  $int\ G.indeg(node\ v);$
  $list < edge >\ G.out\_edges(node\ v);$
  ...

- Update

  $node\ G.new\_node()$;
  Erzeugt einen neuen Knoten in G und gibt ihn (seine Referenz) zurück.
  $edge\ G.new\_edge(node\ v, node\ w)$
  $void\ G.del\_edge(edge\ e)$
  $void\ G.del\_node(nodev)$
  Entfernt v und alle inzidenten Kanten.
  ...

- Iteration (Makros)

  $forall\_nodes(v, G)\{\dots\}$
  $forall\_edges(e, G)\{\dots\}$
  $forall\_out\_edges(e, v)$
  $forall\_in\_edges(e, v)$

Iteration über Nachbarknoten von v

```
forall_out_edges(e,v){
   node w = G.target(e);
   // tue etwas mit w
}
```

Test, ob G azyklisch

Ein azyklischer Graph enthält keine Kreise.

Idee: Topologische Sortierung
Entferne jeweils einen Knoten v mit $indeg(v) = \emptyset$ bis der Graph G leer ist.
Falls wir keinen solchen Knoten finden, ist G zyklisch.
Falls G am Ende leer ist, ist G azyklisch.

```
ZERO ← {v ∈ V: indeg(v) = 0}
   while(ZERO ≠ ∅) {
      u ← bel. Knoten aus ZERO
      ZERO ← ZERO \ {w}
      forall(v ∈ V mit (u,v) ∈ E){
         entferne (u,v) aus G
         if(indeg(v) = 0){
            ZERO ← ZERO ∪ {v}
         }
      }
      entferne u aus G
   }
```

(Realisiere die Menge ZERO durch einen Stack.)

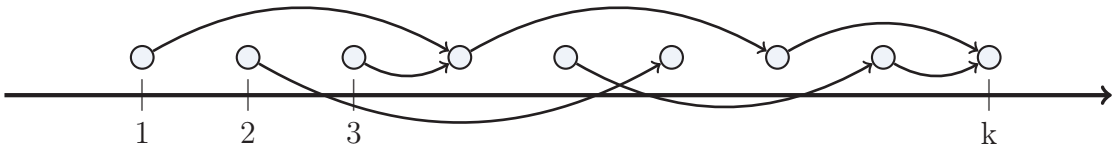
```
bool isAcyclic(graph G){ // Übergabe by value (Kopie)
   stack<node> ZERO;
   node v;
   forall_nodes(v,G){
      if(G.indeg(v) == 0)
         ZERO.push(v);
   }
   while(!ZERO.empty()){
      node u = ZERO.pop();
      edge e;
      forall_out_edges(e,u){
         node w = G.target(e);
         G.del_edge(e);
         if(G.indeg(w) == 0)
            ZERO.push(w);
      }
      G.del_node(u);
   }
   return G.empty();
}
```

Erklärung:
Zuerst werden alle Knoten, die keine eingehenden Kanten besitzen in die Menge *ZERO* aufgenommen. Danach wird solange noch Knoten in der Menge *ZERO* ent-

halten sind, jeweils einer entfernt und für diesen Knoten v überprüft, ob einer der Knoten w, die direkt über eine von v ausgehende Kante mit v verbunden sind, keine eingehenden Kanten mehr hat, wenn diese eine Kante gelöscht würde. Ist dies der Fall, wird w in die Menge $ZERO$ aufgenommen.

Varianten

1. Topologisches Sortieren
   Abb. $ord : V \to \{1, \ldots, n\}$
   (d.h. Anordnung der Knoten)
   1) ord ist injektiv
   2) $\forall (v, w) \in E : ord(v) < ord(w)$



2. G soll nicht zerstört werden
   d.h. Löschen der Kanten wird nur simuliert.
   Idee:
   i) Speichere die Indeg-Werte in einem Feld `indeg[v]`
   ii) Statt `G.del_edge(e)` → `indeg[G.target(e)]-`
   Dadurch können Referenz-Parameter (`graph& G`) genutzt werden und es wird Speicherplatz gespart, da keine zusätzliche Kopie angelegt werden muss.

4.2.2 Basis-Graphtyp graph

Die Informationen (Daten) werden mit den Knoten und Kanten gespeichert.

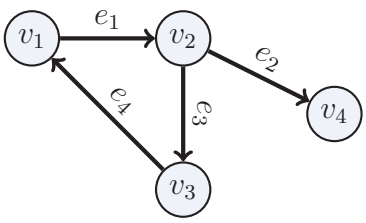
1) Parametrisierte Graphen
   `GRAPH<vtype, etype> G;`
   → Netzwerk: Knoten stehen für Objekte vom Typ `vtype`, Kanten vom Typ `etype`.

   Beispiele:
   ```
   GRAPH<stadt, autobahn> Map;
   GRAPH<transistoren, wire> Schaltkreis;
   GRAPH<person, beziehung> Gruppe;
   ```

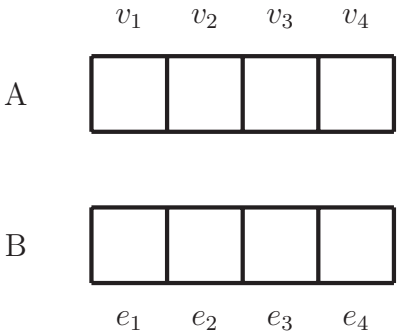
Ableitung: graph → `GRAPH<vtype, etype> G;`

Verwendung der allgemeinen Algorithmen (für den Basistyp)
`bool is_acyclic(graph G)`
`bool is_acyclic(Map)`
Wegen der Typverträglichkeit ist dies möglich.

2) Node/EdgeArray (LEDA)
`node_array<T> A(G)`
`edge_array<T> B(G)`



Getrennt vom Graphen werden die Felder gespeichert.



$x \leftarrow A[v]$
$y \leftarrow B[e]$
`A.operator[]node(v)`

Kommt ein Knoten hinzu können die Felder A und B nicht mehr genutzt werden, da ihre Größe statisch ist. Um das zu umgehen, hat jeder Knoten und jede Kante einen eindeutigen Index. `vec[v → index]`

```
void DIJKSTRA(graph& G, edge_array<int>& cost,
          node s, node_array<int>& dist){...}
```

Eingabe: `graph& G, edge_array<int>& cost, node s`
Ausgabe: `node_array<int>& dist`
Ein- und Ausgabedaten werden getrennt vom Graphen übergeben → generisch

Beispiele:

```
GRAPH<stadt, autobahn> Map,
edge_array<float> km(Map),
edge e,
forall_edges(e,Map) km[e] = Map.get_info(e).km;
node_array<float> dist(Map)
DIJKSTRA(Map,km,trier,dist);
```

3) temporäre Daten
`bool TOPSORT(graph& G, node_array<int>& out)` berechnet eiine topologische Sortierung von G d.h. $forall(v, w) \in Eord[v] < ord[w]$
Der Rückgabewert ist „true“ falls G azyklisch ist, ansonsten „false“

```
// temporäres INDEG-Feld simuliert das Löschen von
   Knoten
node_array<int> INDEG(G);
node v;
list<node> ZERO;
forall_nodes(v,G){
   INDEG[v] = G.indeg(v);
   if(INDEG[v] == 0)
      ZERO.append(v);
}
```

Anwendung DIJKSTRA

```
void Dijkstra(const graph& G, node s, const edge_array<
   int>& cost, node_array<int>& DIST){
   p_queue<node,int> PQ;
   node_array<pq_item> I(G,NULL);
   I[s] = PQ.insert(s,0); // I ist Menge der Indexe der
                          //Knoten
   node v;
   forall_nodes(v,G){
      DIST[v] = MAXINT;
   }
   DIST[s] = 0;

   while(!PQ.empty()){
      node u = PQ.del_min();
      egde e;
      forall_out_edges(e,u){
         node v = G.target(e);
         int d = DIST[u] + cost[e];
         if(d < DIST[v]){ // Dreiecks-Ungleichung verletzt
            if(DIST[v] == MAXINT)
               I[v] = PQ.insert(v,d); // zum ersten Mal
                                      //besucht
            else // v ist in PQ
               PQ.decrease_p(I[v],d)
            DIST[v] = d;
         }
      }
   }
}
```

Erklärung:
Zuerst wird s in die Priority Queue PQ aufgenommen. Die *while*-Schleife läuft nun so lange, bis PQ leer ist. Ist dies der Fall, ist zu jedem Knoten ein kürzester Weg bekannt, falls der Graph azyklisch ist. In der *while*-Schleife wird zuerst aus der PQ ein Knoten u gewählt. Dabei wird darauf geachtet, dass es eine perfekte Wahl ist, d.h. das der Knoten mit dem kleinsten, in der Priority Queue vorkommenden, DIST-Wert gewählt wird. Dann werden alle von u ausgehenden Kanten betrachtet, die jeweilige Δ-Ungleichung überprüft und falls sie verletzt wurde, der erreichte Knoten

entweder in PQ aufgenommen, falls er noch nicht in PQ drin ist oder sonst seine Priorität auf seinen neuen DIST-Wert gesetzt. Danach wird noch sein $DIST$-Wert angepasst.

Analyse
$\left.\begin{array}{l} n \times delmin \\ n \times insert \end{array}\right\} n \times log_2(n)$
$m \times decrease\_p$ ($\mathcal{O}(1)$ amortisiert) $\}\mathcal{O}(m)$

Gesamtlaufzeit: $\mathcal{O}(m + n * log(n))$

Weiterer Aspekt: Korrektheit der Implementierung

Idee: Füge Programmcode hinzu, der für die Konkrete Eingabe testet, ob das Ergebnis korrekt ist.
→ (Programm Checker) Verifying Algorithms

hier: Teste für jede Kante am Ende die Δ-Ungleichung.

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