4 Fortgeschrittene Datenstrukturen und Algorithmen

LEDA: Library of Efficient Data Types and Algorithms

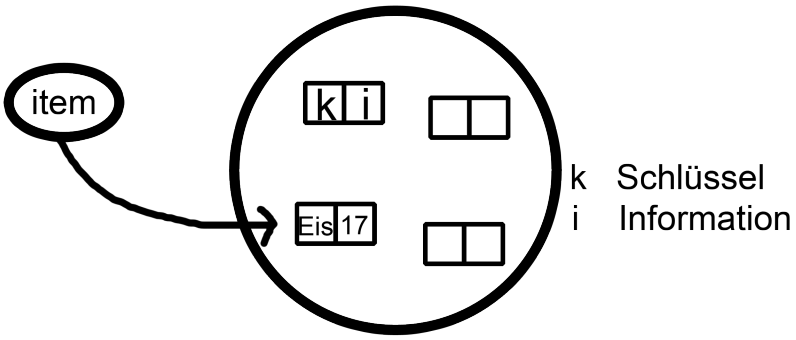
4.1 Spezifikation von Datentypen in LEDA

4.1.1 Item-Konzept

Viele Datenstrukturen werden dabei definiert als eine Menge von sogenannten Items.

Beispiel 4.1 Dictionary
`dictionary<string, int> D` speichert Paare aus Schlüseln (strings) und Informationen (int).

D ist eine Menge von Items (dic_item).



Operationen

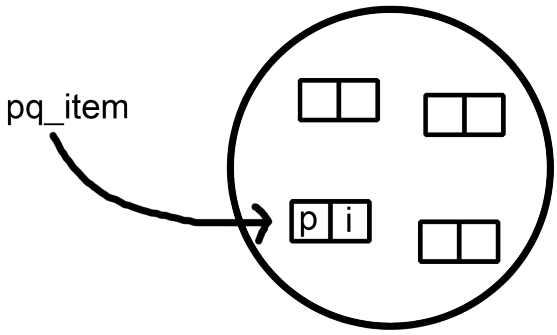
```
dic_item D.insert(string s, int i)\\
   if( D enthält kein Item mit Schlüssel s){
      Füge Item (s,i) ein;
      return (s,i);
   }
   else{
      Ersetze die Information durch i;
      return (s,i);
   }
}
dic_item D.lookup(string s){
   Liefert das Item mit Schlüssel s
   if(kein solches existiert)
      return NULL;
}
void D.change_inf(dic_item it, int x){
   ersetze die Information von it durch x
}
void delete_item(dic_item it){
   Item: Zugriff über Position (Abstraktion von den
      Begriffen Pointer, Referenz, Index)
}
```

Vergleich verschiedener Spezifikationen
Dictionary als Funktion $D : K \to I$ $\leftrightarrow$ Item-Konzept

Beispiel 4.2 Priority Queues
Da bei Priority Queues immer ein Extremwert gesucht wird, eigenen sich für die Implementation Fibonacci-Heaps besonders gut, da eine Abfrage des Extremwertes in $\mathcal{O}(1)$ möglich ist.

`priority_queue<P,I> PQ;`
P: Priorität z.B. Zahlen (int)
I: Information z.B. Knoten eines Graphen (node)
`priority_queue<int, node> PQ;`

PQ ist eine Menge von Items (pq_item)



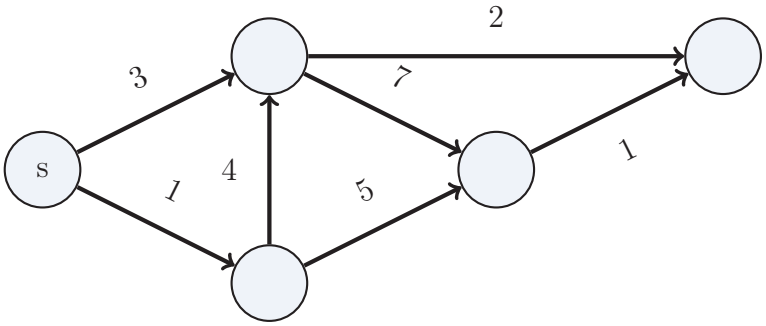
Operationen

```
pq_item PQ.insert(P prio, I inf)\\
      Fügt Item (prio, inf) ein; // Im Gegensatz zum
         Dictionary dürfen bei der Priority-Queue
         Wiederholungen auftreten.
      return (prio, inf);
}
PQ.findmin(){
   Liefert ein Item mit minimaler Priorität.
   Falls die Queue leer ist, wird NULL zurück gegeben.
}
P PQ.prio(pq_item it)
I PQ.inf(pq_item it)
void PQ.delmin(){
   Löscht ein minimales Item.
}
void PQ.decrease_p(pq_item it, P q){
   Vermindert die Priorität von it auf q.
   (q ist kleiner gleich der alten Priorität von it)
}
```

Anwendung auf kürzeste Wege Algorithmen

Speziell: Dijkstra's Algorithmus

Beispiel 4.3



Eingabe

- Graph $G = (V, E)$
- Kostenfunktion $cost : E \to int^*$
- Startknoten $s \in V$

Ausgabe

- Distanzfunktion $dist : V \to int^*$
  $dist(v)$ = Kosten eines billigsten Pfades von s nach v.
  Kosten eines Pfades ist die Summe des billigsten Pfades.

Idee von Dijkstra

1. Überschätze die Dist-Funktion

$$dist(v) := \begin{cases} \infty, & \text{falls } v \neq s \\ 0, & \text{falls } v = s \end{cases}$$

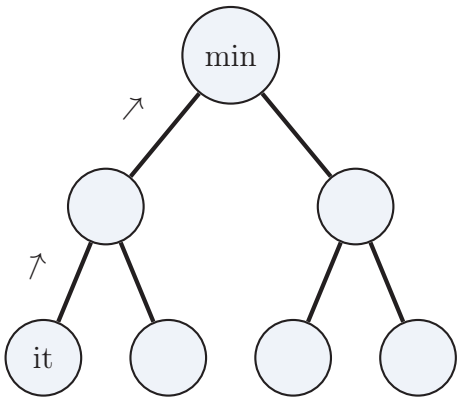
2. Kandidatenliste $\mathcal{U}$ = Menge aller Knoten aus denen Kanten ausgehen können, die eine Abkürzung darstellen. ($\to$ Verletzung Dreiecks-Ungleichung)
   Am Anfang ist nur der Startknoten s in der Liste, da $dist(s) = 0$

3. Hauptschleife über $\mathcal{U}$
   Idee von Dijkstra:
   i) Wähle jeweils $u \in \mathcal{U}$ mit $dist(u)$ minimal. Entferne den gewählten Knoten u aus der Menge $\mathcal{U}$.
   ii) Beobachtung: $dist(u)$ ist korrekt.
   iii) Durchlaufe alle aus u ausgehenden Kanten und überprüfe Dreiecks-Ungleichung.

```
forall_out_edges(e,u){
    v = target(e)
    d = dist(u) + cost(e)
    if(d < dist(v)) {
        dist(v) = d
        U = U + v
    }
}
```

`PQ.decrease_p(it, q)` kann effizient realisiert werden.
Fibonacci-Heap $\rightarrow \mathcal{O}(1)$ (amortisiert)

Binärer Heap:



Iteration:
In C++ eignen sich dafür Makros. Makros sind kleine Programmstücke, die über ihren Namen an beliebigen Stellen im Code aufgerufen werden können. Wie bei Funktionen auch, können ihnen Parameter übergeben werden. Datentypen werden dabei ignoriert, da die Variablen einfach nur ersetzt werden durch die übergebenen Werte.

```
# 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.

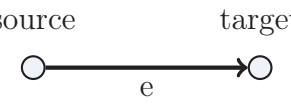
4.2 Graphen und Graphalgorithmen (in LEDA)

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.)

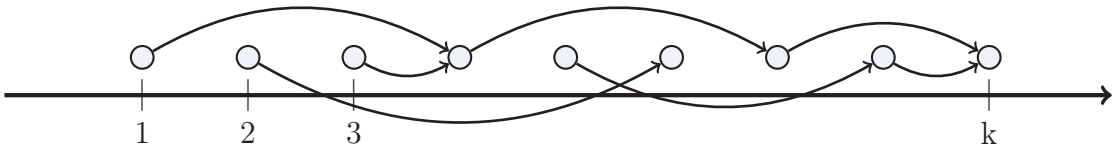
```
bool isAcyclic(graph G){ // Übergabe by value (Kopie)
   stack<node> ZERO;
   node v;
   forall_nodes(v,G){
      if(G.indeg(v) == 0)
         ZERO.push(v);
   }
   while(!ZERO.empty()){
      node u = ZERO.pop();
      edge e;
      forall_out_edges(e,u){
         node w = G.target(e);
        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