Master Theorem Beispiele

Für positive Konstanten a, b, c, d, sei $n = b^k$ für ein $k \in \mathbb{N}$.

$$r(n) = \begin{cases} a & \text{falls } n = 1 \text{ Basisfall} \\ cn + dr(n/b) & \text{falls } n > 1 \text{ teile und herrsche.} \end{cases}$$

schon gesehen, kommt noch, allgemeinerer Fall

$d < b$: Median bestimmen

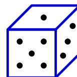
$d = b$: mergesort, quicksort

$d > b$: Schulmultiplikation, Karatsuba-Ofman-Multiplikation

Analyse im Mittel

später an Beispielen

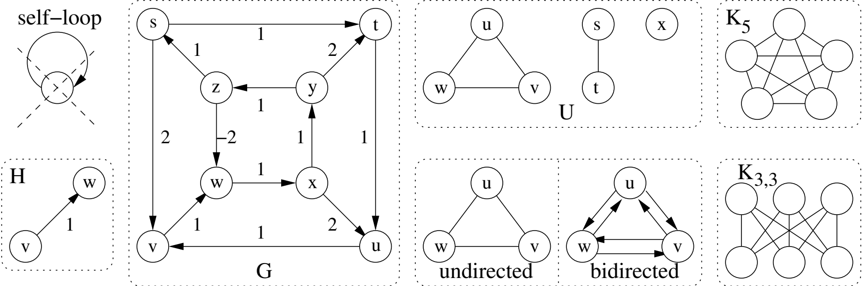
Randomisierte Algorithmen



später an Beispielen

Graphen

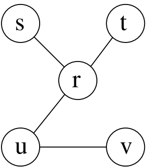
Sie kennen schon (?): Relationen, Knoten, Kanten, (un)gerichtete Graphen, Kantengewichte, Knotengrade, knoteninduzierte Teilgraphen. Pfade (einfach, Hamilton-), Kreise, DAGs

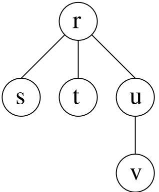


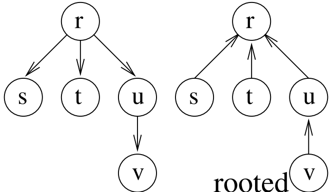
Bäume

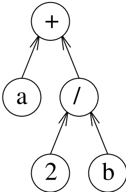
Zusammenhang, Bäume, Wurzeln, Wälder, Kinder, Eltern, . . .

undirected undirected rooted directed expression









Ein erster Graphalgorithmus

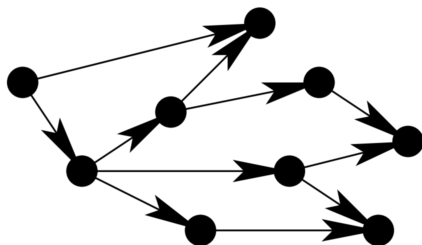
Ein DAG (directed acyclic graph, gerichteter azyklischer Graph) ist ein gerichteter Graph, der keine Kreise enthält.

Function isDAG($G = (V, E)$)
  while $\exists v \in V : \text{outdegree}(v) = 0$ **do**
    invariant G is a DAG iff the input graph is a DAG
    $V := V \setminus \{v\}$
    $E := E \setminus (\{v\} \times V \cup V \times \{v\})$
  return $|V| = 0$

Analyse: kommt auf Repräsentation an (Kapitel 8), geht aber in $\mathrm{O}(|V| + |E|)$.

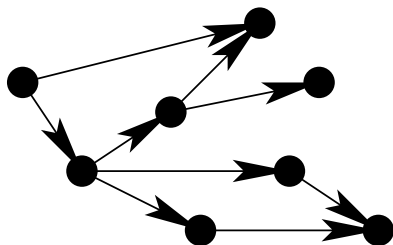
Beispiel

Function isDAG$(G = (V, E))$
  while $\exists v \in V : \text{outdegree}(v) = 0$ **do**
    $V := V \setminus \{v\}$
    $E := E \setminus (\{v\} \times V \cup V \times \{v\})$
  return $|V| = 0$



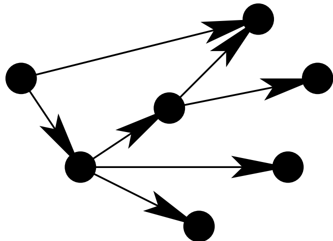
Beispiel

Function isDAG$(G = (V, E))$
  while $\exists v \in V : \text{outdegree}(v) = 0$ **do**
    $V := V \setminus \{v\}$
    $E := E \setminus (\{v\} \times V \cup V \times \{v\})$
  return $|V| = 0$



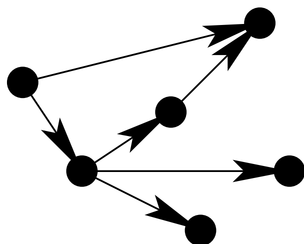
Beispiel

Function isDAG$(G = (V, E))$
  while $\exists v \in V : \text{outdegree}(v) = 0$ **do**
    $V := V \setminus \{v\}$
    $E := E \setminus (\{v\} \times V \cup V \times \{v\})$
  return $|V| = 0$



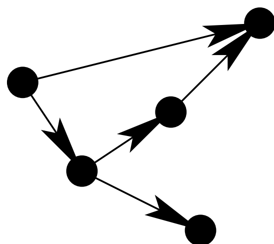
Beispiel

Function isDAG$(G = (V, E))$
  while $\exists v \in V : \text{outdegree}(v) = 0$ **do**
    $V := V \setminus \{v\}$
    $E := E \setminus (\{v\} \times V \cup V \times \{v\})$
  return $|V| = 0$



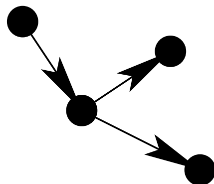
Beispiel

Function isDAG($G = (V, E)$)
  while $\exists v \in V : \text{outdegree}(v) = 0$ **do**
    $V := V \setminus \{v\}$
    $E := E \setminus (\{v\} \times V \cup V \times \{v\})$
  return $|V| = 0$



Beispiel

Function isDAG$(G = (V, E))$
  while $\exists v \in V$: outdegree$(v) = 0$ **do**
    $V := V \setminus \{v\}$
    $E := E \setminus (\{v\} \times V \cup V \times \{v\})$
  return $|V| = 0$



Beispiel

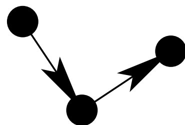
Function isDAG($G = (V, E)$)

while $\exists v \in V$: outdegree$(v) = 0$ **do**

$V := V \setminus \{v\}$

$E := E \setminus (\{v\} \times V \cup V \times \{v\})$

return $|V| = 0$



Beispiel

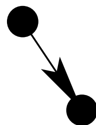
Function isDAG($G = (V, E)$)

  while $\exists v \in V : \text{outdegree}(v) = 0$ **do**

    $V := V \setminus \{v\}$

    $E := E \setminus (\{v\} \times V \cup V \times \{v\})$

  return $|V| = 0$



Beispiel

Function isDAG($G = (V, E)$)

  while $\exists v \in V : \text{outdegree}(v) = 0$ **do**

    $V := V \setminus \{v\}$

    $E := E \setminus (\{v\} \times V \cup V \times \{v\})$

  return $|V| = 0$

Beispiel

Function isDAG($G = (V, E)$)

  while $\exists v \in V : \text{outdegree}(v) = 0$ **do**

    $V := V \setminus \{v\}$

    $E := E \setminus (\{v\} \times V \cup V \times \{v\})$

  return $|V| = 0$

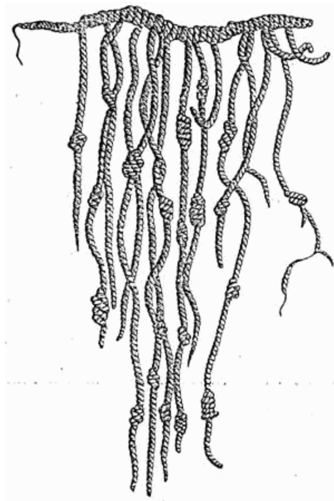
Leerer Graph.

P und NP

das kommt in „Theoretische Grundlagen der Informatik“
Ganz kurz:

- Es gibt einigermaßen gute Gründe, „effizient“ mit „polynomiell“ gleichzusetzen (d. h. Laufzeit $n^{O(1)}$).
- Es gibt viele algorithmische Probleme (NP-vollständig/-schwer), bei denen es SEHR überraschend wäre, wenn sie in Polynomialzeit lösbar wären.

Folgen als Felder und Listen



Folgen

spielen in der Informatik eine überragende Rolle.
Das sieht man schon an der Vielzahl von Begriffen:
Folge, Feld, Schlange, Liste, Datei, Stapel, Zeichenkette, Log...
(sequence, array, queue, list, file, stack, string, log...).
Wir unterscheiden:

- abstrakter Begriff $\langle 2,3,5,7,9,11,\ldots \rangle$ — Mathe
- Funktionalität (stack, ...) — Softwaretechnik
- Repräsentation und Implementierung — Algorithmik

Anwendungen

- Ablegen und Bearbeiten von Daten aller Art
- Konkrete Repräsentation abstrakterer Konzepte wie Menge, Graph (Kapitel 8),. . .

Form Follows Function

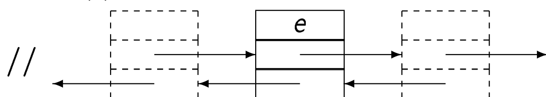
Operation	List	SList	UArray	CArray	explanation '*'
$[\cdot]$	n	n	1	1	
$\|\cdot\|$	1^*	1^*	1	1	not with inter-list splice
first	1	1	1	1	
last	1	1	1	1	
insert	1	1^*	n	n	insertAfter only
remove	1	1^*	n	n	removeAfter only
pushBack	1	1	1^*	1^*	amortized
pushFront	1	1	n	1^*	amortized
popBack	1	n	1^*	1^*	amortized
popFront	1	1	n	1^*	amortized
concat	1	1	n	n	
splice	1	1	n	n	
findNext,...	n	n	n^*	n^*	cache-efficient

Verkettete Listen

Listenglieder (Items)

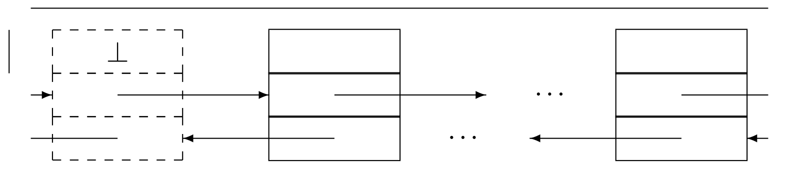
Class Handle = **Pointer to** Item

Class Item **of** Element // one link in a doubly linked list
  e : Element
  next : Handle //
  prev : Handle
  invariant next→prev = prev→next = **this**



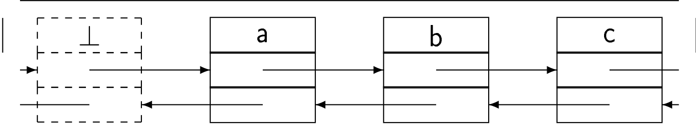
Problem:
Vorgänger des ersten Listenelements?
Nachfolger des letzten Listenelements?

Trick: dummy header



+ Invariante immer erfüllt
+ Vermeidung vieler Sonderfälle
  - ⇝ einfach
  - ⇝ lesbar
  - ⇝ schnell
  - ⇝ testbar
  - ⇝ elegant
− Speicherplatz (irrelevant bei langen Listen)

Dummy header – Beispiel $\langle a, b, c \rangle$



Die Listenklasse

Class List **of** Element

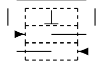
  // Item h is the predecessor of the first element

  // and the successor of the last element.

  Function head : Handle; **return address of** h

  // Pos. before any proper element

  $h = \begin{pmatrix} \bot \\ head \\ head \end{pmatrix}$: Item // init to empty sequence



  // Simple access functions

  Function isEmpty : $\{0, 1\}$; **return** h.next = head // $\langle\rangle$?

  Function first : Handle; **assert** ¬isEmpty; **return** h.next

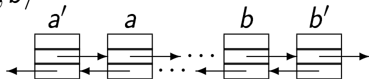
  Function last : Handle; **assert** ¬isEmpty; **return** h.prev

  ⋮

Procedure splice(a, b, t : Handle)// Cut out $\langle a, \ldots, b \rangle$ and insert after t

assert b is not before $a \wedge t \notin \langle a, \ldots, b \rangle$

// Cut out $\langle a, \ldots, b \rangle$

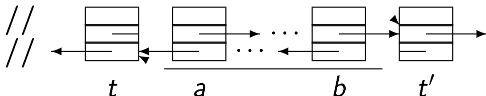


$a' := a \rightarrow \text{prev}$

$b' := b \rightarrow \text{next}$

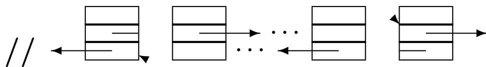
$a' \rightarrow \text{next} := b'$ //

$b' \rightarrow \text{prev} := a'$ //



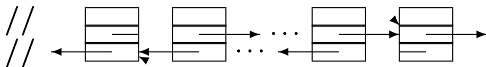
// insert $\langle a, \ldots, b \rangle$ after t

$t' := t \rightarrow \text{next}$ //



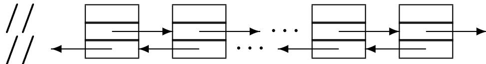
$b \rightarrow \text{next} := t'$ //

$a \rightarrow \text{prev} := t$ //



$t \rightarrow \text{next} := a$ //

$t' \rightarrow \text{prev} := b$ //



Splice Beispiel

$$\langle 1,2,3,4,\overbrace{5,6,7,8}^{a \qquad\qquad b},9,10\rangle,\langle u,v,\overbrace{w}^{t},x,y,z\rangle$$

$$\Downarrow$$

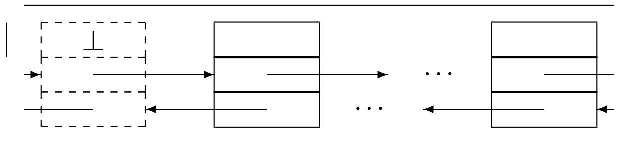
$$\langle 1,2,3,4,9,10\rangle,\langle u,v,w,5,6,7,8,x,y,z\rangle$$

Der Rest sind Einzeiler (?)

// Moving elements around within a sequence.
// $\langle \ldots, a, b, c \ldots, a', c', \ldots \rangle \mapsto \langle \ldots, a, c \ldots, a', b, c', \ldots \rangle$
Procedure moveAfter(b, a' : Handle) splice(b, b, a')

// $\langle x, \ldots, a, b, c, \ldots \rangle \mapsto \langle b, x, \ldots, a, c, \ldots \rangle$
Procedure moveToFront(b : Handle) moveAfter(b, head)

// $\langle \ldots, a, b, c, \ldots, z \rangle \mapsto \langle \ldots, a, c, \ldots, z, b \rangle$
Procedure moveToBack(b : Handle) moveAfter(b, last)



Oder doch nicht? Speicherverwaltung!

naiv / blauäugig /optimistisch:
Speicherverwaltung der Programmiersprache
$\rightsquigarrow$ potentiell sehr langsam
Hier: einmal existierende Variable (z. B. `static` member in Java)
freeList enthält ungenutzte Items.
checkFreeList stellt sicher, dass die nicht leer ist.
Reale Implementierungen:

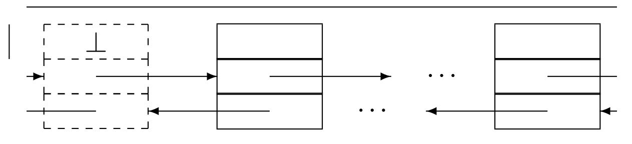
- naiv aber mit guter Speicherverwaltung
- verfeinerte Freelistkonzepte (klassenübergreifend, Freigabe,...)
- anwendungsspezifisch, z. B. wenn man weiß wieviele Items man insgesamt braucht

Items löschen

// $\langle \dots, a, b, c, \dots \rangle \mapsto \langle \dots, a, c, \dots \rangle$
Procedure remove(b : Handle) moveAfter(b, freeList.head)

// $\langle a, b, c, \dots \rangle \mapsto \langle b, c, \dots \rangle$
Procedure popFront remove(first)

// $\langle \dots, a, b, c \rangle \mapsto \langle \dots, a, b \rangle$
Procedure popBack remove(last)



Elemente einfügen

```
// ⟨..., a, b, ...⟩ ↦ ⟨..., a, e, b, ...⟩
Function insertAfter(x : Element; a : Handle) : Handle
    checkFreeList           // make sure freeList is nonempty.
    a' := freeList.first    // Obtain an item a' to hold x,
    moveAfter(a', a)        // put it at the right place.
    a' → e := x             // and fill it with the right content.
    return a'

Function insertBefore(x : Element; b : Handle) : Handle
    return insertAfter(e, b →prev)
Procedure pushFront(x : Element) insertAfter(x, head)
Procedure pushBack(x : Element) insertAfter(x, last)
```

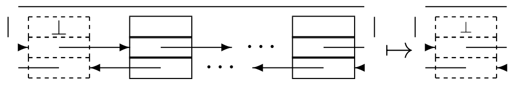
Ganze (Teil)Listen Manipulieren

// $(\langle a,\ldots,b\rangle,\langle c,\ldots,d\rangle)\mapsto(\langle a,\ldots,b,c,\ldots,d\rangle,\langle\rangle)$

Procedure concat(L' : List)
    splice(L'.first, L'.last, last)

// $\langle a,\ldots,b\rangle\mapsto\langle\rangle$

Procedure makeEmpty
    freeList.concat(**this**)//

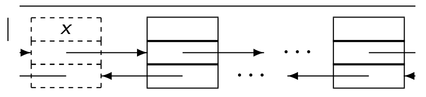


Das geht in konstanter Zeit – unabhängig von der Listenlänge!

Suchen

Trick: gesuchtes Element in Dummy-Item schreiben:

Function findNext(x : Element; from : Handle) : Handle
 $h.e = x$ // Sentinel
 while from $\rightarrow e \neq x$ **do**
  from:= from $\rightarrow$ next
 return from

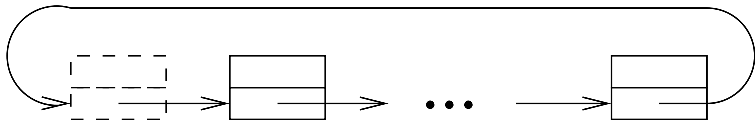


Spart Sonderfallbehandlung.
Allgemein: ein Wächter-Element (engl. Sentinel) fängt Sonderfälle ab.
$\rightsquigarrow$ einfacher, schneller,...

Funktionalität $\leftrightarrow$ Effizienz

Verwalte zusätzliches Member size.
Problem: inter-list splice geht nicht mehr in konstanter Zeit
Die Moral von der Geschicht:
Es gibt nicht DIE Listenimplementierung.

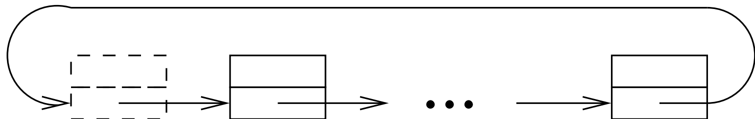
Einfach verkettete Listen



Vergleich mit doppelt verketteten Listen

- weniger Speicherplatz
- Platz ist oft auch Zeit
- eingeschränkter, z. B. kein remove
- merkwürdige Benutzerschnittstelle, z. B. removeAfter

Einfach verkettete Listen – Invariante?



Betrachte den Graphen $G = (\text{Item}, E)$ mit
$E = \{(u, v) : u \in \text{Item}, v = u.\text{next}\}$

- u.next zeigt immer auf ein Item
- $\forall u \in \mathit{Item} : \text{indegree}_G(u) = 1$.
  Wohl definiert obwohl nicht unbedingt leicht zu testen.

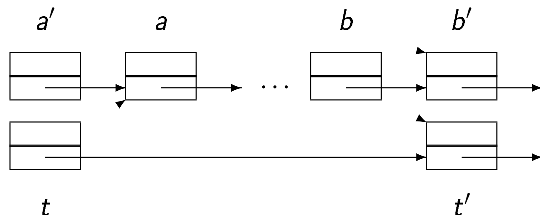
Folge: Items bilden Kollektion von Kreisen

Einfach verkettete Listen – splice

$// (\langle \dots, a', a, \dots, b, b' \dots \rangle, \langle \dots, t, t', \dots \rangle) \mapsto$
$// (\langle \dots, a', b' \dots \rangle, \langle \dots, t, a, \dots, b, t', \dots \rangle)$

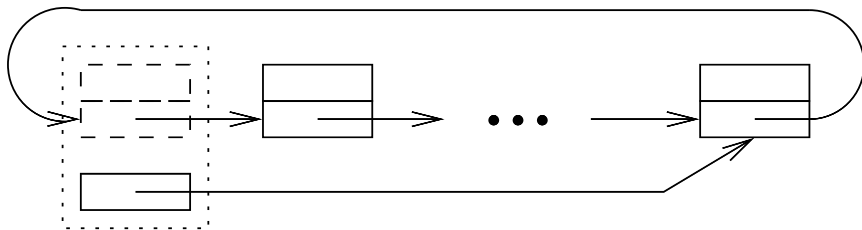
Procedure splice(a',b,t : SHandle)

$$\begin{pmatrix} a' \rightarrow \text{next} \\ t \rightarrow \text{next} \\ b \rightarrow \text{next} \end{pmatrix} := \begin{pmatrix} b \rightarrow \text{next} \\ a' \rightarrow \text{next} \\ t \rightarrow \text{next} \end{pmatrix}$$



Einfach verkettete Listen – pushBack

Zeiger auf letztes Item erlaubt Operation pushBack



Listen: Zusammenfassung, Verallgemeinerungen

- Zeiger zwischen Items ermöglichen flexible, dynamische Datenstrukturen
  später: Bäume, Prioritätslisten
- (einfache) Datenstrukturinvarianten sind Schlüssel zu einfachen, effizienten Datenstrukturen
- Dummy-Elemente, Wächter,... erlauben Einsparung von Sonderfällen
- Einsparung von Sonderfällen machen Programme, einfacher, lesbarer, testbarer und schneller