Algorithmen und Datenstrukturen

SoSe 2008 in Trier

Henning Fernau
Universität Trier
fernau@uni-trier.de

Algorithmen und Datenstrukturen

Gesamtübersicht

- Organisatorisches / Einführung
- Grundlagen: RAM, $\mathcal{O}$-Notation, Rekursion, Datenstrukturen
- Sortieren
- Wörterbücher und Mengen
- Graphen und Graphalgorithmen

Ebenen der Abstraktion I

Was zu tun ist. . .

Mathematik Funktionen, (logische etc.) Spezifikation der Arbeitsweise

Algorithmik Algorithmus (in Pseudocode)

Programmierung Implementierung als konkrete Prozedur in einer Programmiersprache

Ebenen der Abstraktion II

Womit etwas zu tun ist. . .

Mathematik Algebra (Datentyp): syntaktische und semantische Sicht

Algorithmik Datenstruktur

Programmierung Implementierung als konkreter Typ (Modul, Klasse)

Weitere Datenstrukturen

Elementare Datenstrukturen

Keller (engl.: (meist) Stack)

Schlangen

Listen

Mengen

Bäume

Erinnerung: Kellerspeicher

Kellerspeicher arbeiten nach dem LIFO-Prinzip (Last In, First Out).
Einfüge- und Lösch-Stelle sind identisch.

Statische Keller kann man mit Feldern implementieren.

Dynamische Keller kann man mit einfach verketteten Listen darstellen.

Hinweis: Rekursions-Keller (vom Compiler generiert insbesondere zum Abspeichern von Variableninhalten)

Keller-Operationen und ihre Umsetzung für Keller S über Datentyp D:

$S.clear()$	$top \leftarrow 0$
$S.empty()$	return $(top = 0)$
$S.push(x)$	$top{+}{+}; S.K[top] \leftarrow x;$
$S.pop()$	return $S.K[top]; top{-}{-};$
$S.top()$	return $S.K[top]$

Achtung: Überlauf / Unterlauf (Overflow / Underflow) bislang unbeachtet.
Dafür könnte man zusätzliche (Test-)Operationen einführen.
Beobachte: (1) Ist D elementar (z.B., $D = \mathbb{Z}$), so lassen sich alle Operationen in $\mathcal{O}(1)$ im RAM-Modell umsetzen.
(2) top zeigt immer auf das oberste Kellerzeichen, sofern vorhanden.

Keller-Operationen und ihre Eigenschaften

1. Nach $S.\mathtt{clear}()$ ist $S.\mathtt{empty}()$ stets wahr.

2. Tritt kein Überlauf auf, so gilt: $S.\mathtt{top}(S.\mathtt{push}(x)) = x$.

3. $x \leftarrow S.\mathtt{top}()$ is äquivalent zu: $x \leftarrow S.\mathtt{pop}(); S.\mathtt{push}(x)$.

4. Ist der Keller nicht leer, so gilt: $S.\mathtt{push}(S.\mathtt{pop}())$ verändert den Keller nicht.

Beobachte: Auf die "Implementierung" des Kellers als Feld wird hier nicht mehr Bezug genommen. $\leadsto$ Altenativer (abstrakterer, mathematischerer) Ansatz über *abstrakte Datentypen / Algebren*

Keller-Algebra

(A) Syntaktische Beschreibung:

Ein *Keller* wird beschrieben durch ein Tupel $(D, \chi, \epsilon, \downarrow, \uparrow, \tau)$ mit
D Datentyp (Sorte); daraus abgeleitet: $stack(D) \rightsquigarrow$ mehrsortige Signatur
$\chi :\rightarrow stack(D)$
$\epsilon : stack(D) \rightarrow \mathbb{B}$
$\downarrow: stack(D) \times D \rightarrow stack(D)$
$\uparrow: stack(D) \rightarrow stack(D) \times D$
$\tau : stack(D) \rightarrow D$

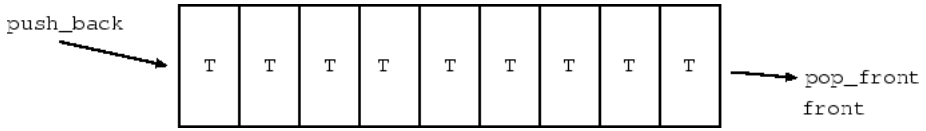
(B) Semantik: Die obigen Symbolfolgen werden als Funktionen gedeutet.
Es gelten insbesondere folgende Gesetze (Axiome):
$\forall S \in stack(D) \forall x \in D(\uparrow (\downarrow (S, x)) = (S, x))$
$\forall S \in stack(D) \forall x \in D(\tau(\downarrow (S, x)) = x)$
$\forall S \in stack(D)(\neg\epsilon(S) \Rightarrow (\downarrow (\uparrow (S)) = S))$

Schlangen-Algebra

(A) Syntaktische Beschreibung:

Eine *Schlange* wird beschrieben durch ein Tupel $(\mathrm{D}, \chi, \epsilon, \mathrm{enq}, \mathrm{deq}, \mathrm{frt})$ mit
D Datentyp (Sorte); daraus abgeleitet: $\mathrm{queue}(\mathrm{D}) \rightsquigarrow$ mehrsortige Signatur
$\chi :\rightarrow \mathrm{queue}(\mathrm{D})$ (create, clear)
$\epsilon : \mathrm{queue}(\mathrm{D}) \rightarrow \mathbb{B}$ (emptiness test)
$\mathrm{enq} : \mathrm{queue}(\mathrm{D}) \times \mathrm{D} \rightarrow \mathrm{queue}(\mathrm{D})$ (enqueue: Einfügen am Ende)
$\mathrm{deq} : \mathrm{queue}(\mathrm{D}) \rightarrow \mathrm{queue}(\mathrm{D}) \times \mathrm{D}$ (dequeue: Entfernen am Anfang)
$\mathrm{frt} : \mathrm{queue}(\mathrm{D}) \rightarrow \mathrm{D}$ (front)

(B) Semantik: Die obigen Symbolfolgen werden als Funktionen gedeutet.
Überlegen Sie sich einmal selbst sinnvolle Regeln.

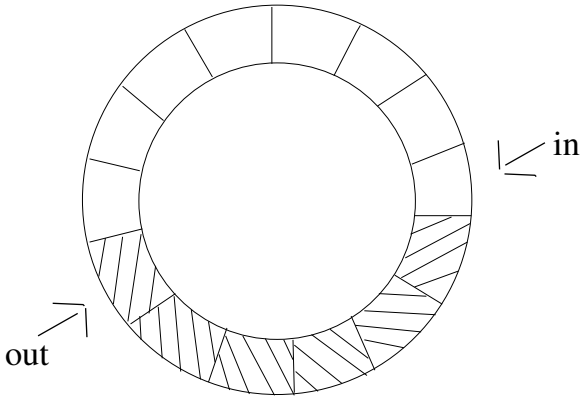


Schlangen: statische Implementierung

Wir verwenden wieder Felder.

Diese werden aber zyklisch interpretiert als *Ringpuffer*.

Hinweis: Anwendung von Ringpuffern bei Erzeuger-Verbraucher-Problemen im Bereich der Betriebssysteme, s. `http://www.informatik.uni-koeln.de/ls_speckenmeyer/teaching/ss06/betriebssysteme/BSprozesseIV06.pdf`



Beachte: Schlangen arbeiten nach dem FIFO-Prinzip (First In, First Out).

Schlangen: dynamische Implementierung

Klassendefinition: Queue.cc

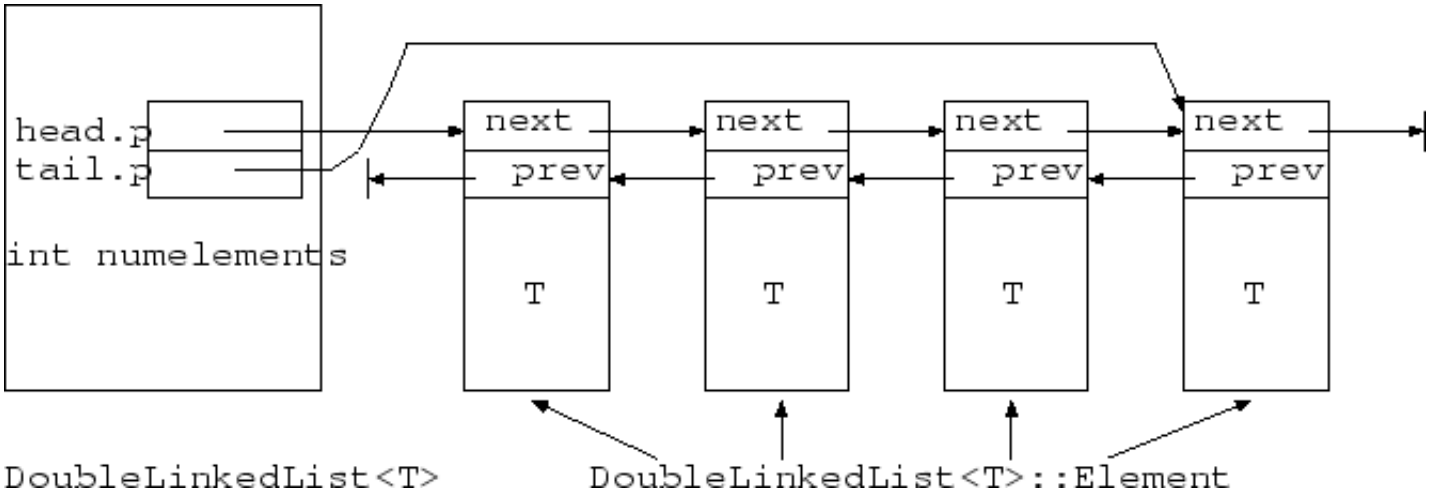
```
template<class T>
class Queue : private DoubleLinkedList<T> {
public :
    Queue ();
    Queue (const Queue<T>&);
    Queue<T>& operator= (const Queue<T>&);

    bool isempty ();
    bool isfull ();
    void push_back (T t);
    T pop_front ();
    T front ();
} ;
```

Die Schlange wird hier mittels der doppelt verketteten Liste und privater Ableitung implementiert.

Schlangen in C++:
Diese Schlange bietet die folgenden Funktionen:

- Konstruktion einer leeren Queue.
- Einfügen eines Elementes vom Typ T am Ende.
- Entfernen des Elementes am Anfang.
- Inspektion des Elementes am Anfang.
- Test ob Queue voll oder leer

Doppelt verkettete Liste



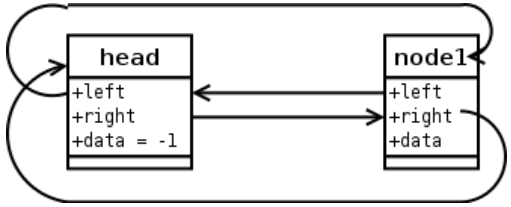
Beachte: Doppelt verkettete Listen implementiert man gerne zyklisch.

Doppelt verkettete Liste Beispiel: Deklaration mit ganzzahligen Elementen

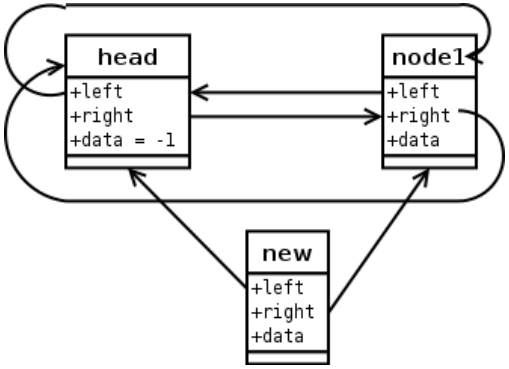
```
struct int_list {
public :
    int_list ();

    bool empty ();
    void push_back (int t);
    int pop_front ();
    ...
private:
    int_list_item* first_it; // Initialisiert mit NULL
    int_list_item* last_it;  // Initialisiert mit NULL
    int size; // Initialisiert mit 0
} ;
struct int_list_item {
  int value;
  int_list_item* pred; //Vorgänger
  int_list_item* succ; //Nachfolger
} ;
```

Doppelt verkettete Liste: Einfügen an beliebiger Stelle:

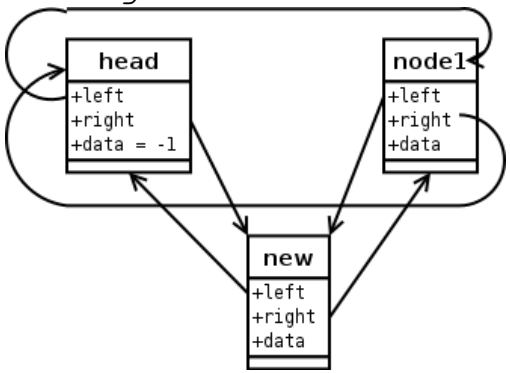
Eine doppelt verkettete Liste mit zwei Elementen.



Ein neues Objekt wird erstellt, der Zeiger `new->left` wird auf `head` gesetzt, der Zeiger `new->right` wird auf `head->right` gesetzt, was gleichbedeutend mit `node1` ist.

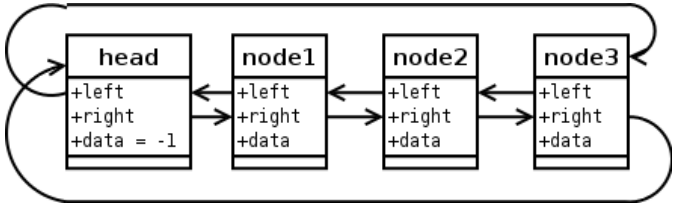


Der Zeiger `head->right` wird auf `new` gesetzt. Jetzt muss noch `node1->left` auf `new` gesetzt werden, aber da wir keinen Zeiger mehr von `head` aus auf `node1` haben, setzt man: `new->right->left = new`.

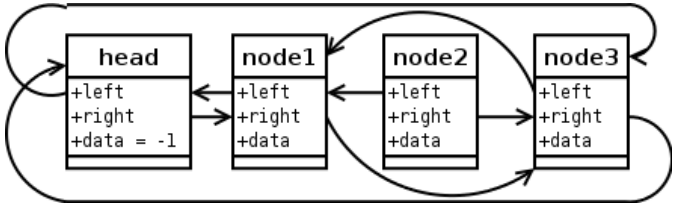


Doppelt verkettete Liste: Löschen

Eine doppelt verkettete Liste mit vier Elementen.
Aus dieser Liste soll `node2` entfernt werden.



`list->right->left`, also `node3->left`, wird auf den linken Nachbarn des zu löschenden Elements gesetzt, also auf `node1`.
Danach kommt die andere Richtung dran, d.h. `list->left->right`, also `node1->right`, wird auf den rechten Nachbarn des zu löschenden Elements gesetzt, also auf `node3`.



Doppelt verkettete Listen: Tipps und Tricks

Wichtig zur Implementierung dynamischer Schlangen (wie genau ?).
Wichtig für die Umsetzung von Backtracking-Algorithmen.
Dabei wichtig für das “Rückverfolgen”: Wie auf der vorigen Folie wird Speicher nach dem Löschen nicht freigegeben, sodass ein einfacheres Wiederherstellen des früheren Zustandes möglich ist. (Dancing Links)

Bäume

Bäume entstammen als Begriff der Graphentheorie.
Dort bezeichnet man ungerichtete Graphen als Bäume, wenn sie kreisfrei (azyklisch) und zusammenhängend sind.

Hinweis: Die Anzahl der *Nachbarn* eines Knoten in einem Baum kann dabei beliebig groß sein.

Es gibt auch das Modell des gerichteten Baumes (*Wurzelbaumes*) in der Graphentheorie: das sind gerichtete (schwach) zusammenhängende Graphen, wo jeder Knoten bis auf einen (die sogenannte *Wurzel*) Eingangsgrad Eins hat.
Die in der Informatik wichtigen Binärbäume sind mit den eben diskutierten Wurzelbäumen verwandt, obliegen aber noch der Einschränkung, dass jeder *innere Knoten* (also jeder Knoten bis auf die *Blätter*, welche Ausgrad Null besitzen) Ausgrad Eins oder Zwei besitzt. Außerdem kann man zwischen den beiden Nachfolgern eines inneren Knotens unterscheiden.

Binärbäume—rekursive Definition

Es sei D ein Datentyp; $x : D$ heißt auch *Knoten* vom Typ D.
Der *leere Baum* λ ist ein Binärbaum; er ist der einzige Binärbaum ohne Wurzel.
Gilt $x : D$, so ist (x) ein Binärbaum mit Wurzel x; er ist der kleinste Binärbaum mit Wurzel. In Übereinstimmung mit der im Folgenden eingeführten Notation schreiben wir auch (λ, x, λ) für (x).
Ist $x : D$ eine Knoten und sind T_1, T_2 Binärbäume, so ist auch (T_1, x, T_2) ein Binärbaum (mit Wurzel x).
Nichts weiter sind Binärbäume.
Hierbei heißt T_1 *linker Teilbaum* und T_2 *rechter Teilbaum*; die Wurzel x_1 von T_1 heißt auch *linkes Kind* von x; die Wurzel x_2 von T_2 entsprechend *rechtes Kind*. x heißt auch *Elternteil* von x_1 bzw. x_2, und x_1 und x_2 sind *Geschwister*.
Knoten ohne Kinder heißen *Blätter*, solche mit Kindern *innere Knoten*.

Das lädt zu einem Tafelbild ein. . .

Pfade in Binärbäumen und andere Begriffe

Ein *Pfad* (der *Länge* n) in einem Binärbaum ist eine Folge von Knoten $p_0, \ldots, p_n$, sodass jeweils p_{i+1} Kind von p_i ist.
Die *Höhe* eines Baumes T ist die Länge des längsten Pfades in T, der offensichtlich von der Wurzel zu einem Blatt führen muss.
Für einen Knoten p heißen die Knoten auf einem Pfad von der Wurzel zu p *Vorfahren* von p, und Knoten, die auf einem Pfad von p zu einem Blatt liegen, heißen *Abkömmlinge* oder *Nachfahren* von p.

Satz: Die maximale Höhe eines Binärbaumes mit n Knoten ist $n-1$, also $\mathcal{O}(n)$.
Die minimale Höhe eines Binärbaumes mit n Knoten ist $\lceil \log_2(n+1) \rceil - 1 \in \mathcal{O}(\log(n))$.

Der erste Teil der Aussage ist klar; wenden wir uns dem zweiten Teil zu...

Lemma: Sei $N(h)$ die maximale Knotenanzahl in einem Binärbaum der Höhe h.
Dann gilt $N(h) = 2^{h+1} - 1$.

Beweis: Ein Baum der Höhe h hat maximal 2^i Knoten auf der i*-ten Schicht*, das ist die Menge der Knoten, die i echte Vorfahren besitzen.
Also gilt: $N(h) = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$ (geometrische Reihe)

Solche Bäume heißen auch *vollständig*.
Die Behauptung des Satzes folgt nun leicht aus dem Lemma.

Anwendung: Binäre Suche

Lemma: binsearch benötigt $\mathcal{O}(\log(n))$ Schritte.

Algorithm 1 Suchen in sortiertem Feld: binsearch

Input(s): an array $A : \mathbb{R}[1..n]$, an item $x : \mathbb{R}$

Output(s): message if x is found

$mid \leftarrow \lfloor (n+1)/2 \rfloor$.
if $mid = 1$ **then**
  return $(A[1] = x)$
else if $A[mid] \leq x$ **then**
  return binsearch$(A[1..mid], x)$;
else
  return binsearch$(A[mid+1..n], x)$;

Implementierungen I

Die rekursive Definition von Binärbäumen legt eine listenartige Verweisstruktur zur Umsetzung von Binärbäumen auf Rechnern nahe (dynamische Implementierung).

Genauer enthält ein Knotenelement dann drei Bestandteile:
Speicher für den eigentlichen Gegenstand
einen Verweis auf das linke Kind
einen Verweis auf das rechte Kind

Möglicherweise ist es (in Analogie zur doppelt verketteten Liste) auch noch sinnvoll, einen Verweis auf den Elternknoten zu speichern.

Implementierungen II

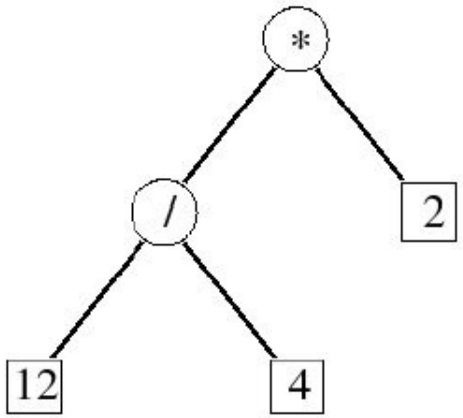
Es gibt auch wieder ein statisches Modell zur Umsetzung von Binärbäumen.
Betrachte einen vollständigen Binärbaum der Höhe h.
Dieser wird in ein Feld der Länge $2^{h+1} - 1$ eingebettet (siehe Lemma).
Für jeden Knoten p des Baumes gilt:
Ist i der Index von p im Feld, so ist $2i$ der Index des linken Kindes und $2i + 1$ der Index des rechten Kindes.

Beispiele für beide Implementierungen: siehe Tafel.

Baumdurchlauf

Da man auf die einzelnen Knoten eines Baumes nur indirekt über die Wurzel und nicht wie bei einem Feld direkt zugreifen kann, benötigt man Algorithmen, die beim Durchlaufen des Baumes jeden Knoten genau einmal besuchen.
Folgende durch Rekursion definierten Baumdurchläufe sind für binäre Bäume wichtig.
—*Inorder*-Durchlauf (linker Teilbaum, Wurzel, rechter Teilbaum),
—*Preorder*-Durchlauf (Wurzel, linker Teilbaum, rechter Teilbaum),
—*Postorder*-Durchlauf (linker Teilbaum, rechter Teilbaum, Wurzel).
Man kann sich die verschiedenen Durchläufe auch als Übersetzungen von Binärbäumen in Listen vorstellen.
Am besten versteht man dies anhand von Beispielen (Tafel).

Baumdurchlauf: Beispiele



inorder → < 12, /, 4, *, 2 >

preorder → < *, /, 12, 4, 2 >

postorder → < 12, 4, /, 2, * >

Bei dem “Operatoren-Beispiel” wird deutlich:
Inorder-Durchlauf liefert *Infix-Notation* (Klammern nötig!).
Preorder bzw. Postorder liefert *Präfix-* bzw. *Postfix-Notation*.

Jenseits der Binärbäume

(Informatische) Bäume mit beliebig vielen Kindern lassen sich entsprechend rekursiv definieren.

Satz: Die maximale Höhe eines Baumes mit maximalem Ausgangsgrad d und n Knoten ist $\mathrm{n}-1$, die minimale Höhe ist $\mathcal{O}(\log_{\mathrm{d}}(\mathrm{n}))$.

Die Speicherung beliebiger Bäume erfolgt entweder statisch oder aber mit Hilfe von Binärbäumen, wobei die Verzeigerung umgedeutet wird: Es gibt statt eines Zeigers auf den linken und auf den rechten Teilbaum einen Zeiger auf das linkeste Kind und auf das linkeste rechtsstehende Geschwisterkind.

Ausblick: Mengen

Es ist oft vonnöten, Mengen (über einem gewissen Grunddatentyp) zu bilden.
Dann möchte man auch die üblichen mengentheoretischen Operationen realisieren.

Damit dies effizient möglich ist, muss man Mengen geeignet speichern.
Für kleine n-elementige Grundmengen (Grunddatentypen) geht dies sehr gut mit *Bitvektor*en.
Jeder Menge wird dabei ein Binärwort der Länge n zugeordnet.
Mengenoperationen entsprechen nun "bitweisen" logischen Operationen.

Für größere Mengen verwendet man meist geordnete Listen; dazu später mehr...