BÄUME UND REKURSION

Inhalt

- Bäume: Definition und Terminologie
- Eigenschaften
- Verschiedene Darstellungen
- Traversierungsmethoden
- Rekursion

Einführung

- Bäume sind Datenstrukturen, die in der Informatik sehr oft verwendet werden
- Ihre Eigenschaften wurden auch sehr intensiv studiert
- Bäume treffen auch im alltäglichen Leben sehr oft auf, wie zum Beispiel
  - der Familienstammbaum, der Vorfahren / Nachkommen darstellt
  - die Organisation eines Sportturniers (Fussballcup, Tennismeisterschaft, ...)
  - die Organisation eines grossen Unternehmens
- In der Informatik werden Syntaxbäume verwendet um Sätze einer Computersprache darzustellen
- Bäume sind mit dem Konzept Rekursion sehr eng verbunden

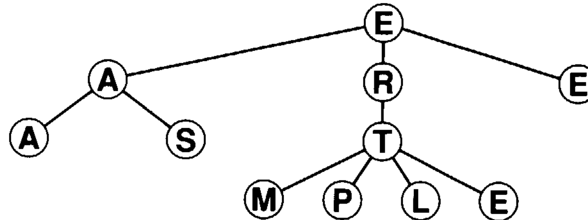
Mathematische Definition

- Ein Graph besteht aus einer nicht leeren Menge von Knoten und Kanten
  - ein *Knote* ist ein Objekt, das einen Namen und andere Informationen enthält
  - eine *Kante* ist eine Verbindung zwischen zwei Knoten
- Ein *Pfad* ist eine Liste von Unterschiedlichen Knoten, in welcher aufeinanderfolgende Knoten durch eine Kante verbunden sind
- Ein *Baum* ist ein Graph, in dem es zwischen zwei beliebigen Knoten immer einen einzigen Pfad gibt

Standpunkt der Informatik

- Einer der Knoten im Baum wird als *Wurzel* bezeichnet
- Graphisch wird der Baum so dargestellt, dass die Wurzel ganz oben ist (unnatürlich!)
- Die anderen Knoten werden so plaziert, dass ein Knote *x* unter *y* liegt, wenn *y* auf dem Pfad liegt zwischen *x* und der Wurzel
- Jeder Knoten (ausser der Wurzel) besitzt somit genau einen Knoten der unmittelbar über ihm liegt, der als seinen *direkten Vorgänger* bezeichnet wird
- Knoten unmittelbar unter einem Knoten werden als seine *direkten Nachfolger* bezeichnet
- Knoten, welche keine Nachfolger haben werden als *Blätter*, als *Endknoten* oder als *äussere Knoten,* bezeichnet
- Knoten die keine Blätter sind, werden auch *innere Knoten* genannt
- Jeder Knote ist die Wurzel eines *Unterbaumes*, welcher aus ihm und den Knoten unter ihm besteht

Beispiel eines Baumes



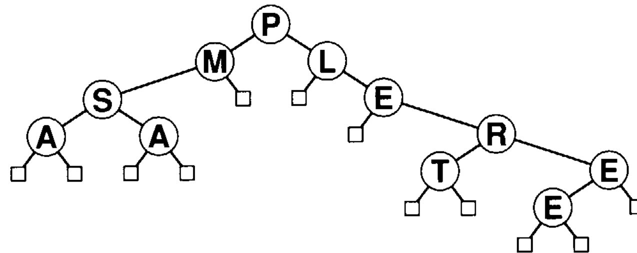
- Dieser Baum besteht aus 11 Knoten
- E ist die Wurzel
- Der besitzt 7 Endknoten und 3 weitere Unterbäume

Weitere Terminologie

- Eine Menge von Bäume wird *Wald* genannt
- Manchmal ist die Reihenfolge in der die Nachfolger eines jeden Knoten angeordnet sind von Bedeutung; ein solcher Baum wird als *geordneter* Baum bezeichnet
- Die *Ebene* (oder Stufe) eines Knoten ist die Anzahl der Knoten die sich auf dem Pfad zur Wurzel befinden (ohne ihn selbst)
- Die *Höhe* (oder Tiefe ?) eines Baumes ist die höchste Ebene aller Knoten
- Die *Pfadlänge* eines Baumes ist die Summe der Ebenen aller Knoten
  - man unterscheidet manchmal zwischen *innerer* und *äusserer Pfadlänge*
- Wenn jeder Knoten eine bestimmte Anzahl *n* von direkten Nachfolgern haben muss, wird von einem *n-ären Baum* gesprochen
  - es ist zweckmässig, äussere Pseudoknoten einzusetzen, die keine Nachfolger haben

Binären Bäume

- Ein *binärer Baum* ist ein 2-ärer Baum
- Jeder Knoten (ausser den Pseudoknoten) besitzt einen *linken* und einen *rechten* Nachfolger



- Ein *voller* binärer Baum ist ein Baum, in welchem jede Ebene mit inneren Knoten ausgefüllt ist
- Ein *vollständiger* binärer Baum ist ein voller binärer Baum, bei dem alle innere Knoten der letzten Ebene links von den äusseren Knoten dieser Ebene erscheinen

Allgemeine Eigenschaften

- Zwischen zwei beliebigen Knoten eines Baumes existiert genau ein Pfad
- Ein Baum mit N Knoten enthält genau N-1 Kanten

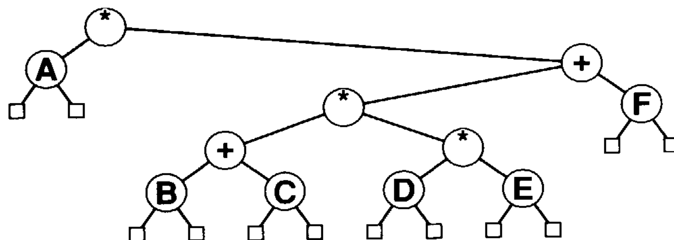
Eigenschaften der binären Bäume

- Ein binärer Baum mit N Knoten enthält genau N+1 äusseren Knoten
- Die äussere Pfadlänge eines binären Baumes mit N inneren Knoten ist um 2N grösser als die innere Pfadlänge
- Die Höhe eines vollen binären Baumes mit N inneren Knoten beträgt etwa $\log_2 N$

© Rolf Ingold, Universität Freiburg

Beispiel: Syntaxbaum eines arithmetischen Ausdrucks

- Syntaxbaum des arithmetischen Ausdrucks `A*(((B+C)*(D*E))+F)`



Darstellung binärer Bäume

- Die gebräuchlichste Darstellung für binäre Bäume besteht aus Knoten mit zwei Verkettungen (mit dem linken (`l`) und dem rechten (`r`) Nachfolger)

```
struct node
  { char info; struct node *l, *r; }
```

- Um die äusseren Knoten darzustellen wird ein Pseudoknoten `z` verwendet

```
struct node *z;
z = (struct node *) malloc(sizeof *z);
z->l = z; z->r = z;
```

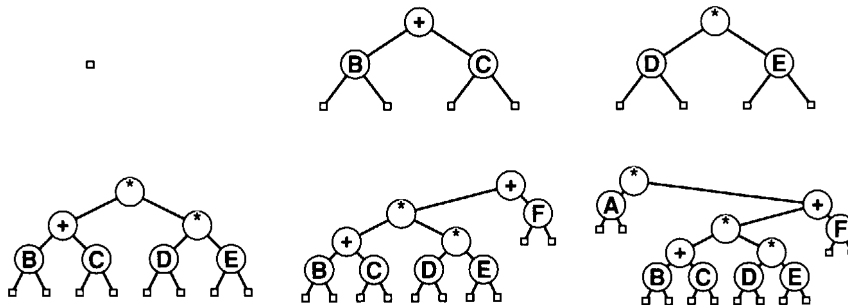
Konstruktion eines Syntaxbaumes

```
struct node *x, *z;
char c;
z = (struct node *) malloc(sizeof *z);
z->l = z; z->r = z;

for (stackinit(); scanf("%1s", &c != EOF); ) {
   x = (struct node *) malloc(sizeof *x);
   x->info = c; x->l = z; x->r = z;
   if (c == '+' || c=='*') {
      x->r = pop(); x->l = pop();
   }
   push(x);
}
```

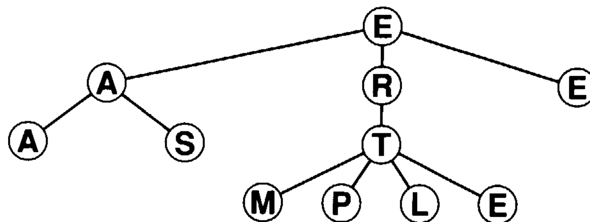
Aufbau des Syntaxbaumes

- Aufbau des Syntaxbaumes des arithmetischen Ausdrucks `A*(((B+C)*(D*E))+F)`



Darstellung allgemeiner Bäume

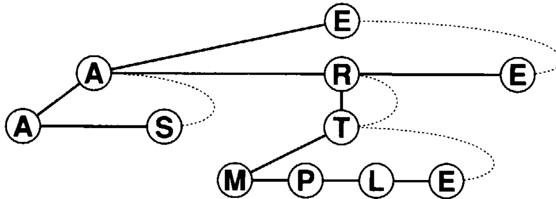
- Erster Ansatz: Verkettung zum direkten Vorgänger
  - kann in parallelen Feldern dargestellt werden

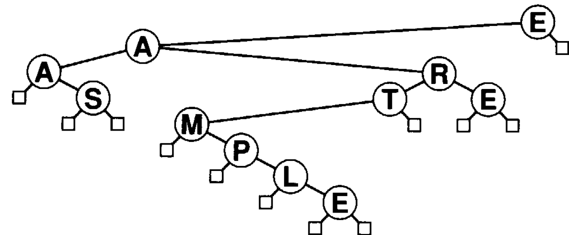


k	1	2	3	4	5	6	7	8	9	10	11
$a[k]$	A	S	A	M	P	L	E	T	R	E	E
$dad[k]$	3	3	10	8	8	8	8	9	10	10	10

Darstellung allgemeiner Bäume

- Zweiter Ansatz: Verwendung einer verketteten Liste der Nachfolger
    - jeder Knoten ist verkettet mit dem ersten Nachfolger sowie mit dem nächsten Knoten in der Liste der Nachfolger Knoten des Vorgängers
    - dabei entsteht ein binärer Baum





Traversierung von Bäumen

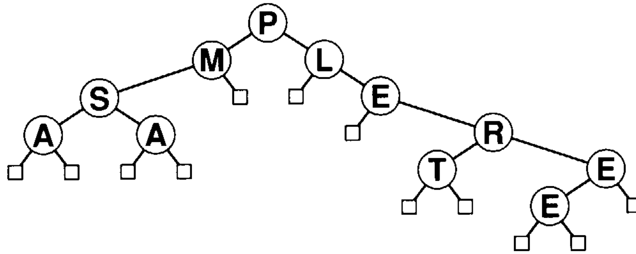
- Für Bäume (im Gegensatz zu Listen) gibt es verschiedene Reihenfolgen in denen die Knoten systematisch besucht werden
- Für binäre Bäume unterscheidet man
  - die *Preorder-Traversierung*
  - die *Inorder-Traversierung*
  - die *Postorder-Traversierung*
  - die *Level-Order-Traversierung*
- Preorder-, Postorder- und Level-Order-Traversierung lassen sich für allgemeine Bäume definieren
- Die Verallgemeinerung auf ein Wald ist ebenfalls einfach: eine Menge von Bäume kann als ein Baum betrachtet werden mit einer imaginären Wurzel.

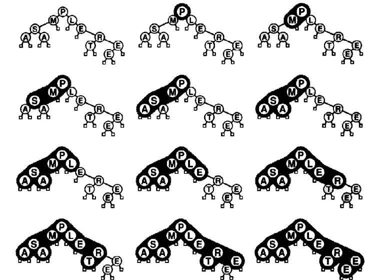
Preorder-Traversierung

- Die rekursive Regel heisst: besuche zuerst die Wurzel dann den linken Unterbaum und schliesslich den rechten Unterbaum
- Für die Preorder-Traversierung kann kann ein Stapel verwendet werden

```
traverse(struct node *t) {
   push(t);
   while (!stackempty) {
      t = pop(); visit (t);
      if (t->r != z) push(t->r);
      if (t->l != z) push(t->l);
   }
}
```

Beispiel der Preorder-Traversierung

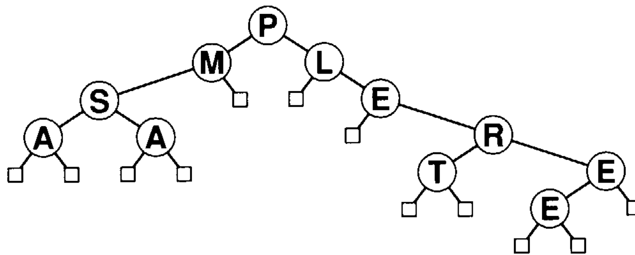


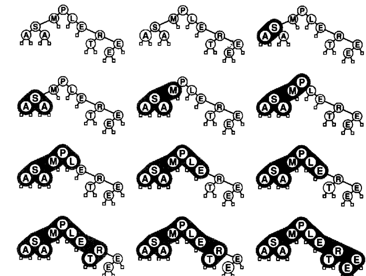


- Die Besuchsreihenfolge lautet `P M S A A L E R T E E`

Inorder-Traversierung

- Die rekursive Regel heisst: besuche zuerst den linken Unterbaum, dann die Wurzel und schliesslich den rechten Unterbaum

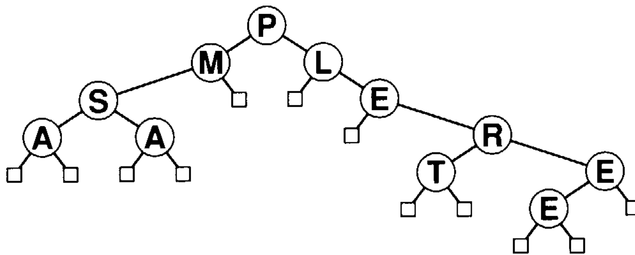


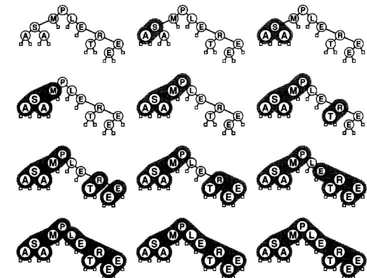


  - Die Besuchsreihenfolge lautet `A S A M P L E T R E E`
- Diese Traversierung wird verwendet um einen arithmetischen Ausdruck in Infix-Notation zu übersetzen

Postorder-Traversierung

- Die rekursive Regel heisst: besuche zuerst den linken Unterbaum, dann den rechten Unterbaum und schliesslich die Wurzel





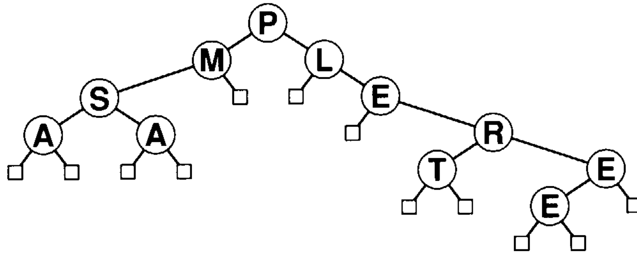
- Die Besuchsreihenfolge lautet `A A S M T E E R E L P`

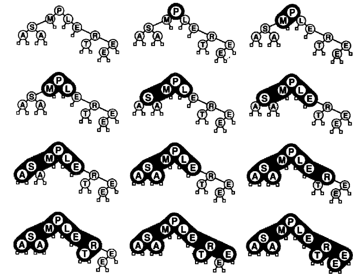
Level-Order-Traversierung

- Die Level-Order-Traversierung ist nicht Rekursiv
- Die Knoten werden von oben nach unten und von links nach rechts besucht
- Für diese Level-Order-Traversierung kann eine Schlange verwendet werden

```
traverse(struct node *t) {
   put(t);
   while (!queueempty) {
      t = get(); visit (t);
      if (t->l != z) put(t->l);
      if (t->r != z) put(t->r);
   }
}
```

Beispiel der Level-Order-Traversierung





- Die Besuchsreihenfolge lautet `P M L S E A A R T E E`

Rekursion

- Die Rekursion ist ein fundamentales Konzept in der Mathematik und in der Informatik
- Ein rekursives Programm ist ein Programm das sich selbst aufruft.
  - wichtig ist dabei die Abbruchbedingung, die bestimmt wann die Rekursion aufhört (sonst würde das Programm nie enden !)
- Viele Algorithmen lassen sich rekursiv sehr elegant beschreiben
- Jeder rekursive Algorithmus kann in einen nicht rekursiven Algorithmus umgewandelt werden

Rekurrente Beziehungen

- Rekursive Definitionen von Funktionen sind in der Mathematik gebräuchlich
  - Beispiel 1: die Fakultät:

    $N! = N(N-1)!$ für $N = 1$

    $0! = 1$

  - Beispiel 2: die Fibonacci-Folge:

    $F_N = F_{N-1} + F_{N-2}$ für $N = 2$

    $F_0 = F_1 = 1$

- Beide Funktionen können mit rekursiven Programmen berechnet werden

```
int factorial(int N) {
  if (N==0) return 1;
  return N*factorial(N-
1);
}
```

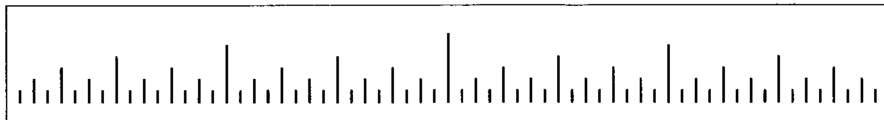
```
int fibonacci(int N) {
  if (N<=1) return 1;
  return fibonacci(N-1)+
         fibonacci(N-2);
}
```

- Rekursive Programme können äusserst uneffizient sein !
  - im Fall der Fibonacci-Folge werden viele Zwischenresultate mehrmals berechnet
  - es werden genau F_N Funktionsaufrufe benötigt !
- Beide Funktionen lassen sich auf eine einfache Schlaufe reduzieren

Teilen und Herrschen

- Viele rekursive Programme verwenden zwei rekursive Aufrufe, die beide die Hälfte des Problems bewältigen
- Im Allgemeinen kann ein solcher Aufruf in 3 Etappen aufgeteilt werden
  - Aufteilung der Daten
  - Rekursive Aufrufe
  - Zusammenführen der Daten
- Das sogenannte Teilen-und-Herrschen-Prinzip kann zu beachtlicher Zeiteinsparung führen

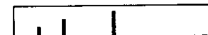
Beispiel: Zeichnen eine Lineals



```
rule (ind l, int r, int h) {
   int m = (l+r)/2;
   if (h>0) {
      mark(m, h);
      rule(l, m, h-1);
      rule(m, r, h-1);
   }
}
```

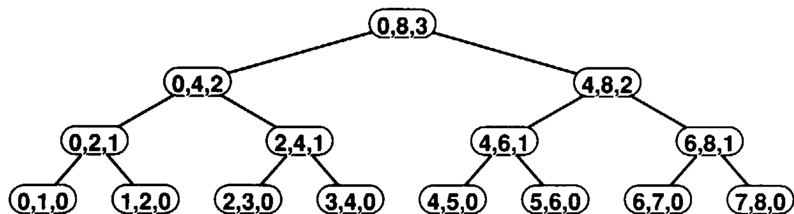
Zeichnen eines Lineals: Aufruffolge

```
rule(0,8,3)
   mark(4,3)
   rule(0,4,2)
      mark(2,2)
      rule(0,2,1)
         mark(1,1)
         rule(0,1,0)
         rule(1,2,0)
      rule(2,4,1)
         mark(3,1)
         rule(2,3,0)
         rule(3,4,0)
rule(4,8,2)
      mark(6,2)
      rule(4,6,1)
         mark(5,1)
         rule(4,5,0)
         rule(5,6,0)
```



```
rule(6,8,1)
   mark(7,1)
   ...
```

Zeichnen eines Lineals: Baum der rekursiven Aufrufe



Rekursive Traversierung von Bäumen

- Traversierungen von Bäumen können sehr einfach rekursiv formuliert werden
  - Beispiel der Inorder-Traversierung

```
traverse (struct node *t) {
   if (t != z) {
      traverse(t->l);
      visit(t);
      traverse(t->r);
   }
}
```

Anwendung: Koordinaten für die Anordnung der Knoten eines binären Baumes

```
int x = 0; int y = 0;

visit (struct node *t) {
   t->x = ++x;
   t->y = y;
}

traverse (struct node *t) {
   y++;
   if (t != z) {
      traverse(t->l);
      visit(t);
      traverse(t->r);
   }
   y--;
}
```

Beseitigung der Rekursion

- Jedes rekursive Programm kann in ein äquivalentes nicht rekursives Programm umgewandelt werden
  - dazu braucht es meistens einen Stapel
  - bei rekursiven Programmiersprachen wird dieses Vorgehen vom Compiler übernommen
- Die Beseitigung der Rekursion in der Preorder-Traversierung kann in 4 Schritten erfolgen
  - Beseitigung des zweiten rekursiven Aufrufes und Einfügen von "goto"-Befehlen
  - Beseitigung des ersten rekursiven Aufrufes unter Anwendung eines Stapels
  - Umstrukturieren und Beseitigung der "goto"-Befehle
  - Vereinfachung

Beseitigung der Rekursion: erster Schritt

```
traverse(struct node *t) {
   if (t != z) {
      visit(t);
      traverse(t->l);
      traverse(t->r);
   }
}
```

- Beseitigung des zweiten rekursiven Aufrufes

```
traverse(struct node *t {
   l: if (t == z) goto x;
      visit(t);
      traverse(t->l);
      t = t->r;
      goto l;
   x: ;
}
```

Beseitigung der Rekursion: zweiter Schritt

- Beseitigung des ersten rekursiven Aufrufes

```
traverse(struct node *t {
   l: if (t == z) goto s;
      visit(t);
      push(t)
      t = t->l;
      goto l;
   r: t = t->r;
      goto l;
   s: if (stackempty()) goto x;
      t = pop();
      goto r;
   x: ;
}
```

Beseitigung der Rekursion: dritter Schritt

- Umstrukturieren

```
traverse(struct node *t {
   l: while (t != z) {
         visit(t); push(t->r); t = t->l;
      }
      if (stackempty()) goto x;
      t = pop(); t = t->r; goto l;
   x: ;
}
```

Beseitigung der Rekursion: vierter Schritt

- Beseitigung der "goto"-Befehle

```
traverse(struct node *t {
   push(t);
   while (!stackempty()) {
      t = pop();
      while (t != z); {
         visit(t); push (t->r); t = t->l;
      }
   }
}
```

Beseitigung der Rekursion: fünfter Schritt

- Vereinfachung

```
traverse(struct node *t {
   push(t);
   while (!stackempty()) {
      t = pop();
      if (t!=z) {
         visit(t);
         push(t->r)
         push(t->l)
      }
   }
}
```

- Es wird empfohlen dieses Programm mit demjenigen zu vergleichen, das beim Vorstellen der Traversierungsalgorithmen gezeigt wurde