HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Programmiertechnik II

Ralf Herbrich

Bäume (Suchbäume)

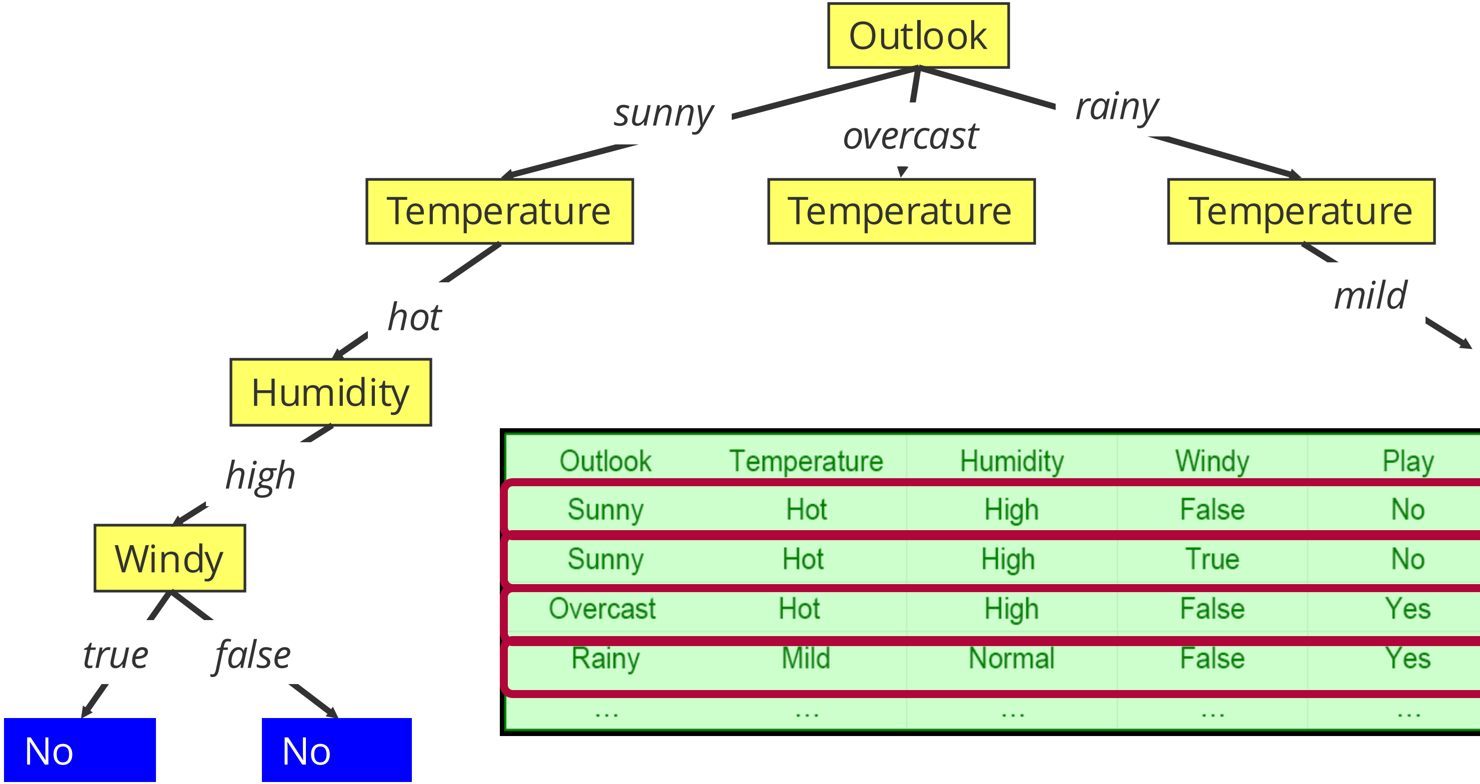
Überblick

1. Begriffe
   - Bäume als spezielle Graphen
   - Eigenschaften
2. Binäre Suchbäume
   - Einfügen
   - Entfernen (*Hibbard deletion*)

Überblick

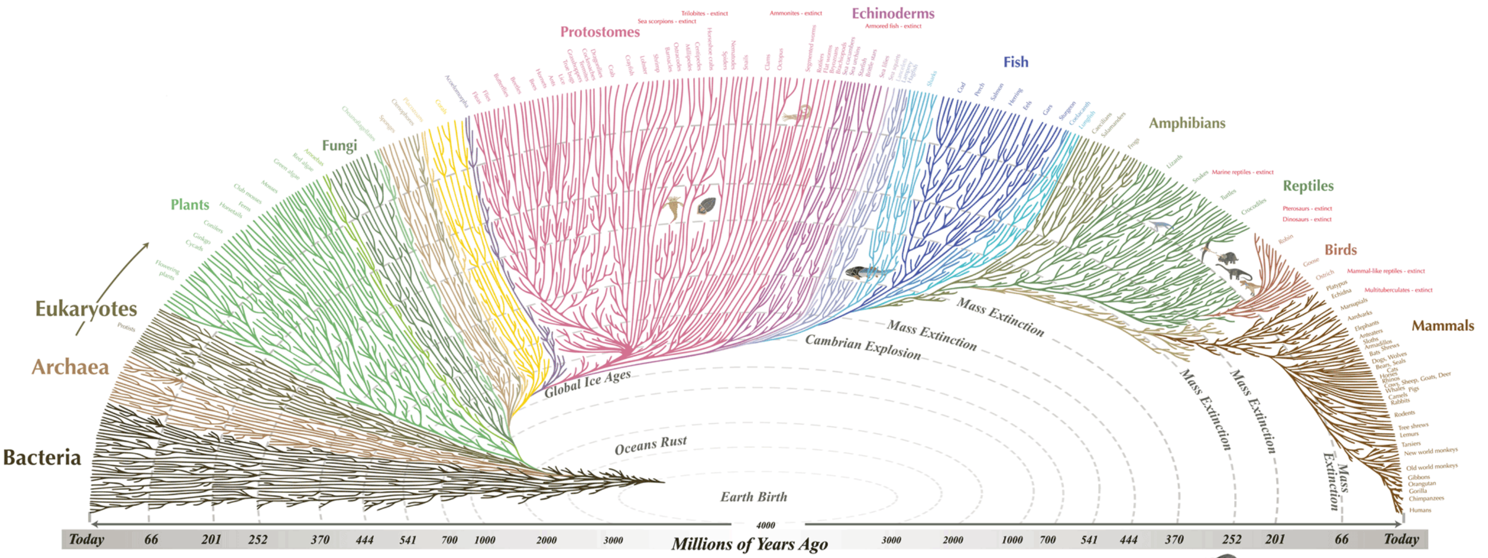
1. **Begriffe**
   - Bäume als spezielle Graphen
   - Eigenschaften
2. Binäre Suchbäume
   - Einfügen
   - Entfernen (*Hibbard deletion*)

Beispiel: Entscheidungsbäume (*decision trees*)



Outlook	Temperature	Humidity	Windy	Play
Sunny	Hot	High	False	No
Sunny	Hot	High	True	No
Overcast	Hot	High	False	Yes
Rainy	Mild	Normal	False	Yes
…	…	…	…	…

Klassifikation: Phylogenetischer Baum des Lebens



All the major and many of the minor living branches of life are shown on this diagram, but only a few of those that have gone extinct are shown. Example: Dinosaurs - extinct

Überblick

1. Begriffe
   - **Bäume als spezielle Graphen**
   - Eigenschaften
2. Binäre Suchbäume
   - Einfügen
   - Entfernen (*Hibbard deletion*)

Allgemein: Graphen

- **Definition (Graph).** Ein **Graph** $G = (V, E)$ besteht aus einer Menge V von Knoten (*vertices, nodes*) und einer Menge $E \subseteq V \times V$ von Kanten. G ist **ungerichtet** falls $\forall (v, v') \in E \rightarrow (v', v) \in E$. Ansonsten ist G **gerichtet**. Jede Kante $(v, v') \in E$ heißt **ausgehend** für v und **eingehend** für v'.

- **Definition (Pfad).** Eine Folge von Kanten $e_1, e_2, \dots, e_n$ heißt **Pfad der Länge $\boldsymbol{n}$** genau dann wenn für alle i: $e_i = (v', v)$, $e_{i+1} = (v, v'')$ und $v' \neq v''$.

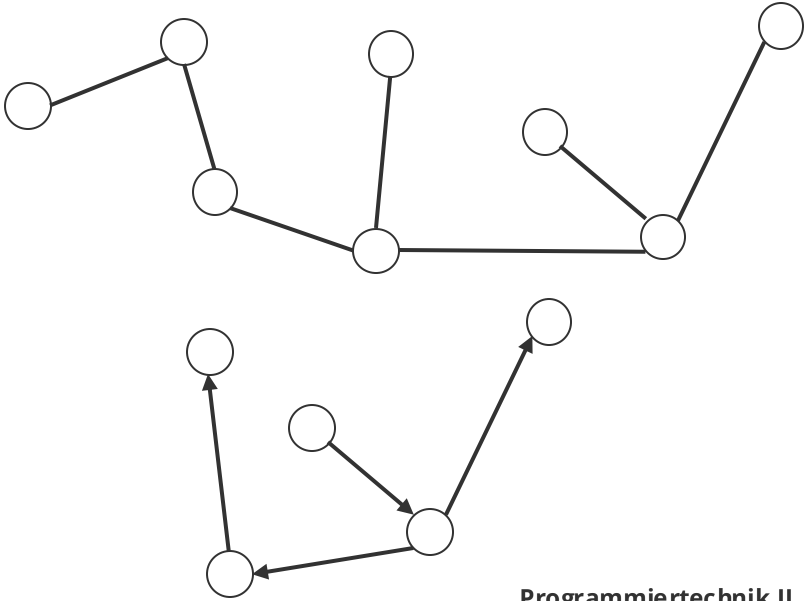
- **Definition (Zusammenhängender Graph).** Ein Graph G ist **zusammenhängend** falls jedes Knotenpaar über mindestens einen Pfad verbunden ist. Ein gerichteter Graph ist **schwach zusammenhängend**, falls der zugehörige ungerichteten Graph zusammenhängend ist.
- **Definition (Azyklischer Graph).** Ein Pfad $(v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n)$ ist azyklisch wenn alle v_i verschieden sind. Ein Graph G ist **azyklisch** falls alle Pfad azyklisch sind.

Bäume als zusammenhängende Graphen

- **Definition (Ungerichteter Baum)**. Ein ungerichteter, zusammenhängender, azyklischer Graph ist ein ungerichteter Baum.

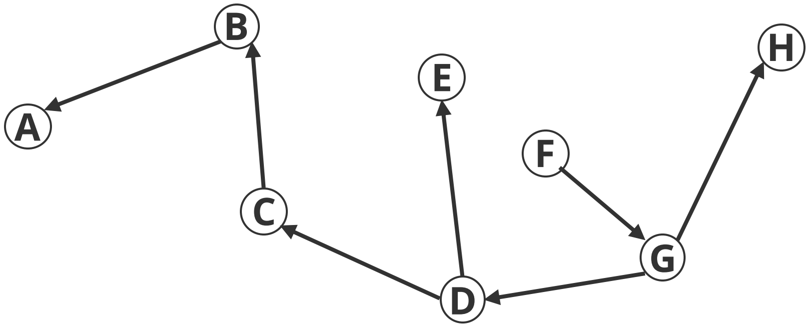
- **Definition (Gerichteter Baum)**. Ein gerichteter, schwach zusammenhängender, azyklischer Graph, in dem jeder Knoten höchstens eine eingehende Kante hat, ist ein gerichteter Baum.

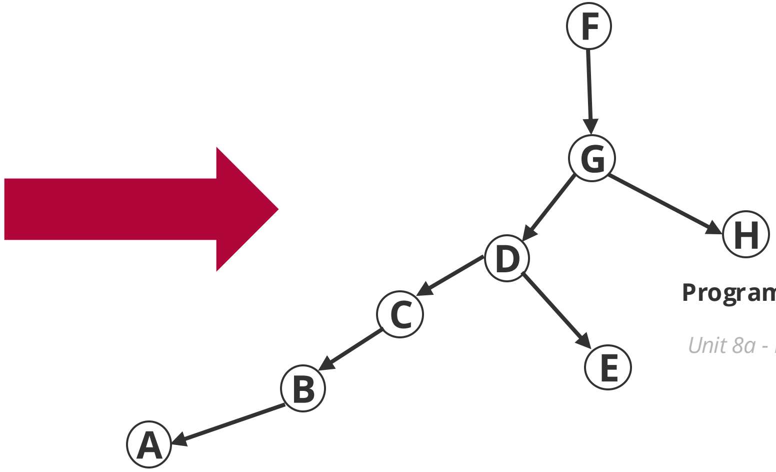
- **Lemma**. In einem ungerichteten Baum gibt es genau einen Pfad zwischen jedem Knotenpaar.



Verwurzelte Bäume

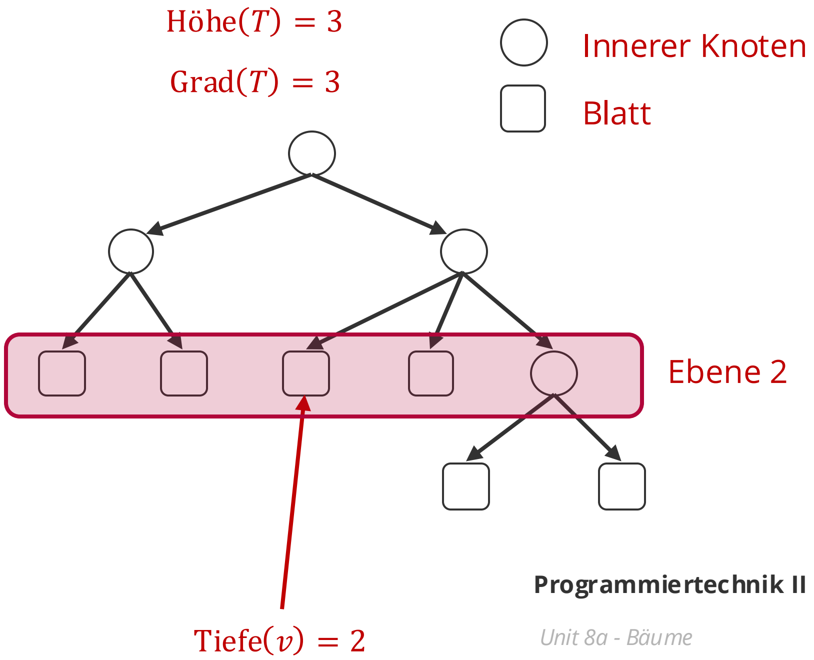
- **Definition (Wurzel)**. Sei der Knoten v in einem gerichteten Baum ohne eingehende Kanten, dann nennen wir v die Wurzel des Baums und den Baum einen **verwurzelten Baum** (auch „gewurzelt").
- **Lemma**. In einem gerichteten, verwurzelten Baum gibt es genau einen Pfad zwischen der Wurzel und jedem anderen Knoten.





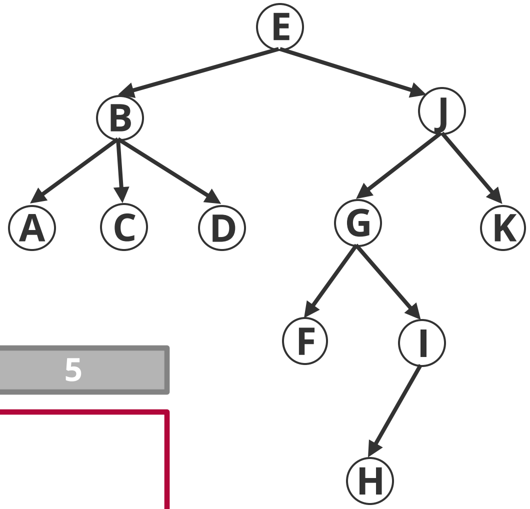
Terminologie

- Ein Knoten ohne ausgehende Kanten ist ein **Blatt**. Alle anderen Knoten sind **innere Knoten**.
- Die **Tiefe** (auch „Niveau“) eines Knotens v ist die Länge des (einzigen) Pfades von der Wurzel zu v.
- Die **Höhe** von T ist die Tiefe des tiefsten Blattes.
- Der **Grad** von T ist die maximale Anzahl von Kindern, die ein Knoten haben darf.
  - Binäre Bäume haben Grad 2.
- **Ebene** i bezeichnet alle Knoten mit Tiefe i.



Schnelltest

- Was ist die Tiefe des Knotens F?
  - **3**
- Was ist die Höhe von T?
  - **4**
- Was ist der Grad von T?
  - **3**



1

2

5

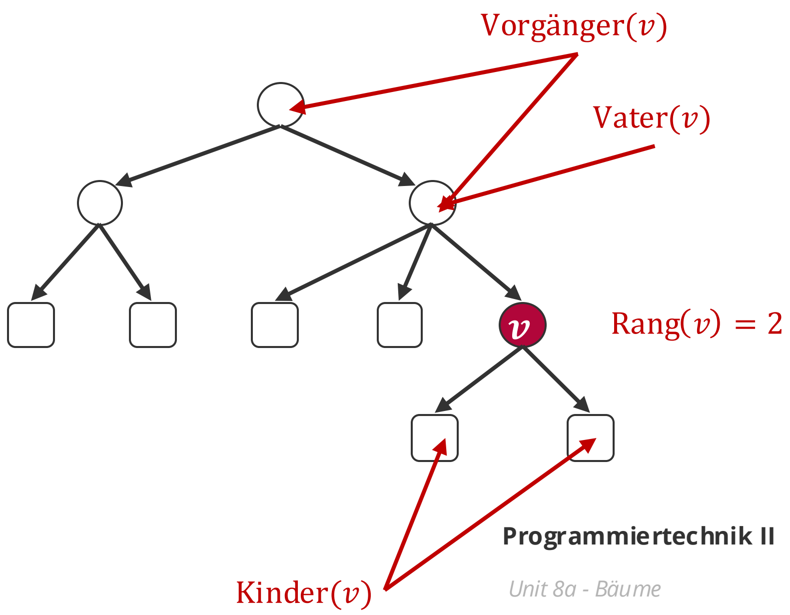
3

4

Keine Ahnung

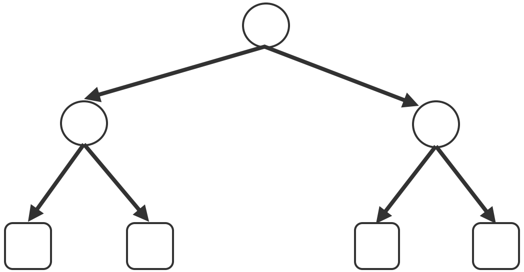
Mehr Terminologie

- Sei T ein Baum und v ein Knoten in T. Dann ist definiert:
  - Alle Knoten an ausgehenden Kanten von v sind dessen **Kinder**.
  - v ist der **Vater** (Elternknoten) aller seiner Kinder.
  - Alle Knoten auf dem Pfad von der Wurzel zu v sind die **Vorgänger** von v.
  - Alle Knoten, die von v erreichbar sind, sind dessen **Nachfolger**.
  - Der **Rang** eines Knotens v ist die Anzahl seiner Kinder.
    - $\mathrm{Rang}(v)$ist stets kleiner-gleich $\mathrm{Grad}(T)$

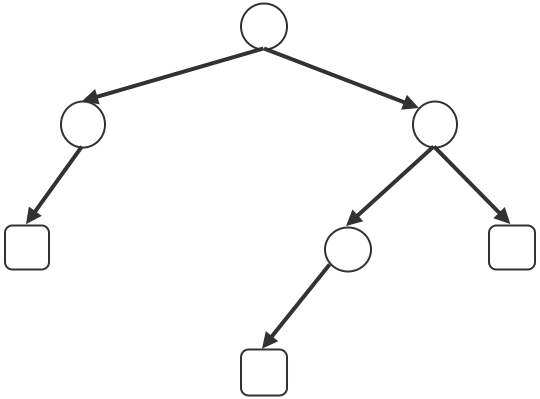


Noch mehr Terminologie

- **Definition (Vollständiger Baum).** Sei T ein gerichteter Baum mit Grad k. T ist **vollständig** falls
  - Alle inneren Knoten den Rang k haben,
  - und alle Blätter die gleiche Tiefe haben.

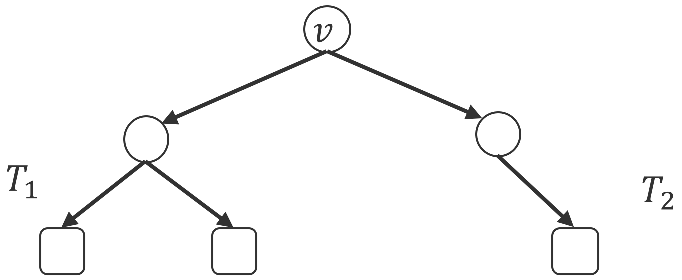


- In der VL zumeist: Verwurzelte, gerichtete, binäre Bäume



Rekursive Definition von Bäumen

- **Bisher**: Bäume als Graph mit bestimmten Bedingungen
- **Aber**: Traversierung oft mit rekursiven Funktionen
- **Rekursive Definition (Baum)**: Ein Baum hat folgende Struktur
  - Ein einzelner Knoten ist ein Baum der Höhe 0.
  - Falls T_1 und T_2 Bäume sind, dann ist die folgende Struktur ein Baum der Höhe $\max\left(\text{Höhe}(T_1), \text{Höhe}(T_2)\right) + 1$ und v ist dessen Wurzel:
    - Neuer Knoten v
    - Neue Kanten von v zu den Wurzeln von T_1 und T_2



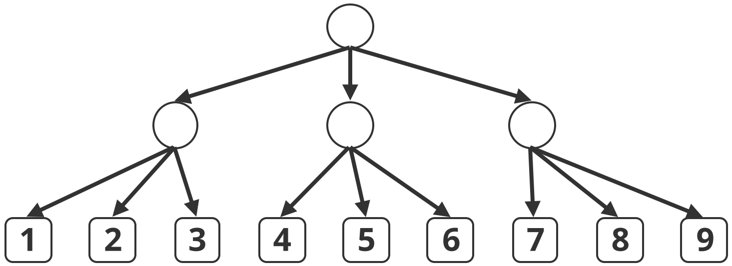
Überblick

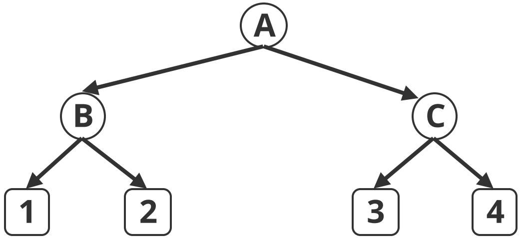
1. Begriffe
   - Bäume als spezielle Graphen
   - **Eigenschaften**
2. Binäre Suchbäume
   - Einfügen
   - Entfernen (*Hibbard deletion*)

Eigenschaften von Bäumen

- Sei $T = (V, E)$ ein Baum mit Grad k. Dann gilt
  - $|V| = |E| + 1$ bzw. $|E| = |V| - 1$
  - Falls T vollständig ist, hat er $k^{\text{Höhe}(T)}$ Blätter.
  - Falls T ein vollständiger binärer Baum ist, hat er $2^{\text{Höhe}(T)+1} - 1$ Knoten.
    - Darunter $2^{\text{Höhe}(T)}$ Blätter und $2^{\text{Höhe}(T)} - 1$ innere Knoten
  - Falls T ein binärer Baum ist, gilt

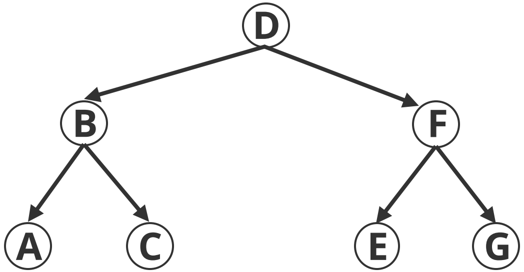
    $$\text{Höhe}(T) \in [\lfloor \log(|V|) \rfloor, |V| - 1]$$





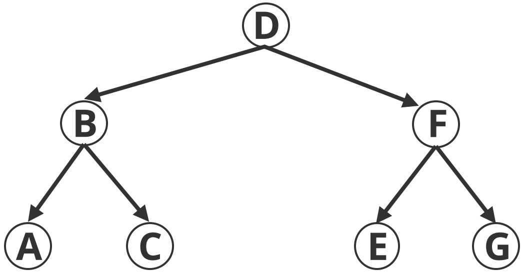
Traversierung: Tiefensuche (*Depth first traversal (DFS)*)

- Systematisches, rekursives Durchlaufen aller Knoten des Baums
- ***In-Order***:
  - 1. linker Teilbaum, 2. Knoten, 3. rechter Teilbaum
  - **A** → **B** → **C** → **D** → **E** → **F** → **G**
  - **Beispiel**: Schlüsselreihenfolge in einem Suchbaum
- ***Pre-Order***:
  - 1. Knoten, 2. linker Teilbaum, 3. rechter Teilbaum
  - **D** → **B** → **A** → **C** → **F** → **E** → **G**
  - **Beispiel**: Ordnerstruktur in Dateisystem
- **Post-Order**:
  - 1. linker Teilbaum, 2. rechter Teilbaum, 3. Knoten
  - **A** → **C** → **B** → **E** → **G** → **F** → **D**



Traversierung: Breitensuche (*Breadth first traversal (BFS)*)

- Auch: „Levelorder“-Durchlauf bzw. „Breitensuche“
  - **D → B → F → A → C → E → G**
  - Wird mit Hilfe einer *queue* implementiert




```
algorithm Levelorder(v)
q = leere Queue;
q.enqueue(k);
while not(is_empty(q)) do
        v = q.dequeue();
        // do something with v->data
        q.enqueue(v->left);
        q.enqueue(v->right);
```

- Tiefensuche als Spezialfall wenn *queue* durch *stack* ausgetauscht wird!

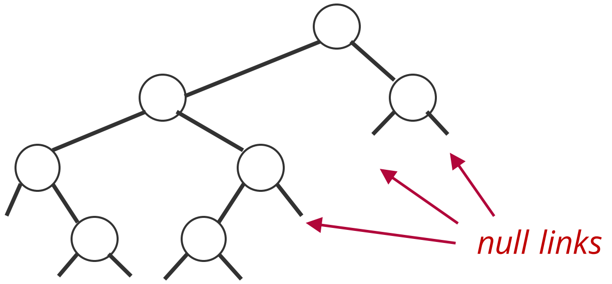
Überblick

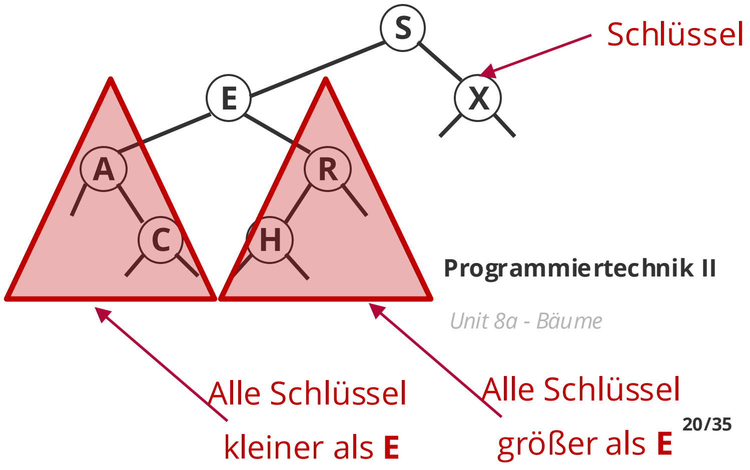
1. Begriffe
   - Bäume als spezielle Graphen
   - Eigenschaften
2. **Binäre Suchbäume**
   - Einfügen
   - Entfernen (*Hibbard deletion*)

- **Definition (Binäre Suchbäume).** Ein binärer Suchbaum (*binary search tree (BST)*) ist ein geordneter binärer Baum.

- **Binäre Suchbäume** sind entweder
  - Leer
  - Zwei disjunkte binäre Teilbäume

- **Geordneter Baum.** Jeder Knoten hat einen Schlüssel. In jedem Knoten gilt für den Schlüssel
  - größer als alle Schlüssel im linken Teilbaum
  - kleiner als alle Schlüssel im rechten Teilbaum





Binäre Suchbäume II

- **Idee von Suchbäumen**: Jeder Knoten speichert „Nutzdaten“
  - Das was wir suchen möchten (Symboltabellen)
- Binäre Suchbäume (oft auch nur Binärbäume genannt) implementieren die folgenden Operationen auf effiziente Weise:
  1. $\mathbf{Get}(x)$: Finde x in der Datenstruktur bzw. stelle fest, dass x nicht enthalten ist.
  2. $\mathbf{Put}(x)$: Füge x in die Datenstruktur ein.
  3. $\mathbf{Remove}(x)$: Lösche x aus der Datenstruktur wenn es darin enthalten ist

- **Probleme**
  - Effizienz nicht trivial erreichbar
  - Änderungen erfordern Reorganisation des Baums (Balancieren)
    - Zunächst aber: Unbalancierte binäre Suchbäume

st.h

Binärer Baum als Abstrakter Datentyp

type BTree(T)

operators

$\text{empty}: \rightarrow \text{BTree}$

$\text{is\_empty}: \text{Btree} \rightarrow \text{Bool}$

$\text{bin}: \text{BTree} \times \text{T} \times \text{BTree} \rightarrow \text{BTree}$

$\text{left}: \text{BTree} \rightarrow \text{BTree}$

$\text{right}: \text{BTree} \rightarrow \text{BTree}$

$\text{value}: \text{BTree} \rightarrow \text{T}$

axioms

$\forall x, y \in \text{BTree}, v \in \text{T}: \quad \text{left}\big(\text{bin}(x, v, y)\big) = x$

$\forall x, y \in \text{BTree}, v \in \text{T}: \quad \text{right}\big(\text{bin}(x, v, y)\big) = y$

$\forall x, y \in \text{BTree}, v \in \text{T}: \quad \text{value}\big(\text{bin}(x, v, y)\big) = v$

$\forall x, y \in \text{BTree}, v \in \text{T}: \quad \text{is\_empty}\big(\text{bin}(x, v, y)\big) = \text{false}$

$\text{is\_empty}(\text{empty}) = \text{true}$

Binäre Suchbäume in C++

- **Klasse**: Ein BST ist ein Zeiger auf den Wurzelknoten (*node*)
- ***Node***: Ein Knoten hat 4 Felder
  - Schlüssel (`key`) und Wert (`val`)
  - Zeiger auf die Wurzel des linken (`left`) und rechten (`right`) Teilbaums

```
struct Node {
    Key key;
    Value val;
    Node* left;
    Node* right;
    int size;

    // constructor with values
    Node(const Key& k, const Value& v, int s) :
        key(k), val(v), size(s),
        left(nullptr), right(nullptr) {}
};
```

Nützlich und zeitsparend für Rang

bst.h

Binäre Suchbäume in C++ (Gerüst)

```
template <typename Key, typename Value>
class BST : public ST<Key, Value> {
    Node* root;

    int size(const Node* n) const {
        return (n ? n->size : 0);
    }
  public:
    void put(const Key& key, const Value& val) { ... }

    const Value* get(const Key& key) const { ... }

    void remove(const Key& key) { ... }

    bool contains(const Key& key) const { return (get(key) != nullptr); }

    bool is_empty() const { return (size() == 0); }

    int size() const { return (size(root)); }
};
```

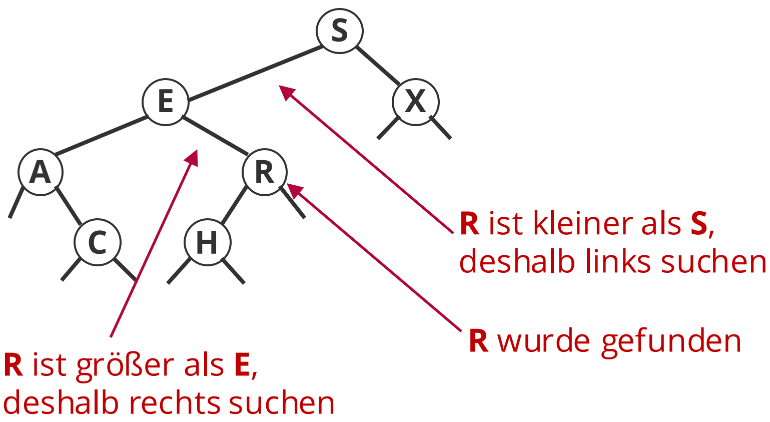
Wurzel des BST

`const` weil `get` nichts verändert im Suchbaum

bst.h

Binäre Suchbäume: `get`

Suche nach **R**




```cpp
// use recursion to find the correct node
const Value* get(const Node* n, const Key& key) const {
    if (n == nullptr) return (nullptr);
    if (key < n->key) return (get(n->left, key));
    if (key > n->key) return (get(n->right, key));
    return &(n->val);
}

public:
const Value* get(const Key& key) const {
    return (get(root, key));
}
```

bst.h

- **Aufwand**: Anzahl Vergleiche = Tiefe des Baums + 1

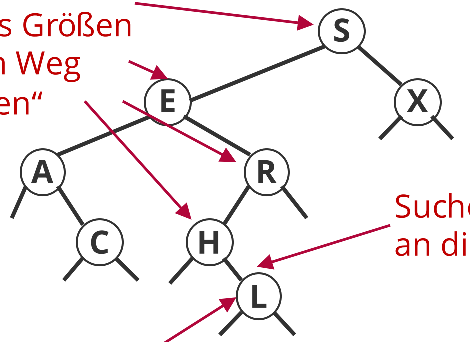
Überblick

1. Begriffe
   - Bäume als spezielle Graphen
   - Eigenschaften
2. Binäre Suchbäume
   - **Einfügen**
   - Entfernen (*Hibbard deletion*)

Binäre Suchbäume: `put`

Einfügen von **L**

Erneuere alle Links uns Größen “auf dem Weg nach oben“



Suche nach **L** endet an diesem *null link*

Erstelle einen neuen Knoten **L**

- **Aufwand**: Anzahl Vergleiche = Tiefe des Baums + 1

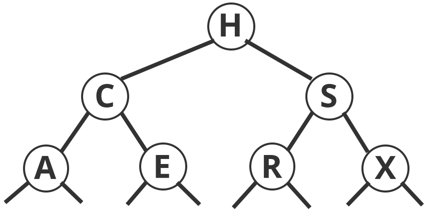
```cpp
// uses recursion to put a key-value pair into the tree
Node* put(Node* n, const Key& key, const Value& val) const {
    if (n == nullptr) {
        auto node = new Node(key, val, 1);
        node->left = nullptr;
        node->right = nullptr;
        return (node);
    }
    if (key < n->key)
        n->left = put(n->left, key, val);
    else if (key > n->key)
        n->right = put(n->right, key, val);
    else
        n->val = val;
    n->size = size(n->left) + size(n->right) + 1;
    return (n);
}

public:
void put(const Key& key, const Value& val) {
    root = put(root, key, val);
}
```

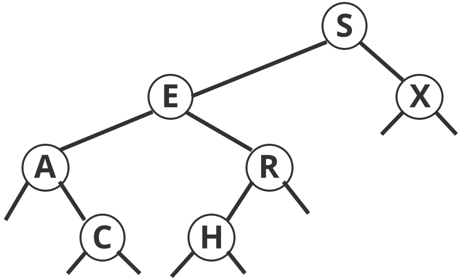
bst.h

Baumform (*tree shape*)

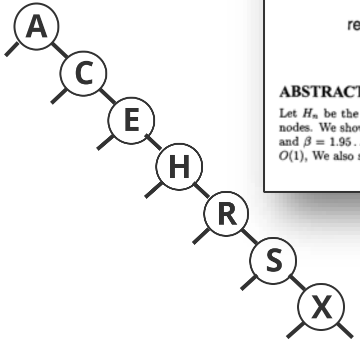
- Für die gleichen Schlüssel, gibt es viele BSTs



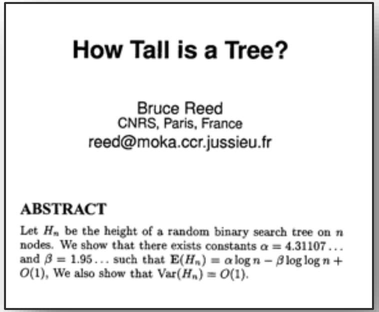
bester Fall



typischer Fall



schlechtester Fall



How Tall is a Tree?

Bruce Reed
CNRS, Paris, France
reed@moka.ccr.jussieu.fr

ABSTRACT

Let H_n be the height of a random binary search tree on n nodes. We show that there exists constants $\alpha = 4.31107\ldots$ and $\beta = 1.95\ldots$ such that $E(H_n) = \alpha \log n - \beta \log\log n + O(1)$, We also show that $Var(H_n) = O(1)$.

- **Satz (Reed 2003).** Wenn n Schlüssel in **zufälliger** Reihenfolge in einen binären Suchbaum eingefügt werden, dann ist die erwartete Anzahl an Vergleichen für Suche/Einfügen $\approx 1.39 \cdot \log_2(n)$ und der Baum hat eine erwartete Tiefe von $\approx 3 \cdot \log_2(n)$.

Symboltabelle: Komplexitäten

Implementierung	Garantiert			*Average*		
	Suche (`get`)	Einfügen (`put`)	Löschen (`remove`)	Suche (`get`)	Einfügen (`put`)	Löschen (`remove`)
Sequentielle Suche	n	n *	n	$\frac{n}{2}$	n	$\frac{n}{2}$
Binäre Suche	$\log_2(n)$	n	n	$\log_2(n)$	$\frac{n}{2}$	$\frac{n}{2}$
Binäre Suchbäume	n	n	n	$1.39 \cdot \log_2(n)$	$1.39 \cdot \log_2(n)$	$?$

$\boldsymbol{n}$

$\boldsymbol{n/2}$

$\boldsymbol{\log_2(n)}$

$\boldsymbol{1.39 \cdot \log_2(n)}$

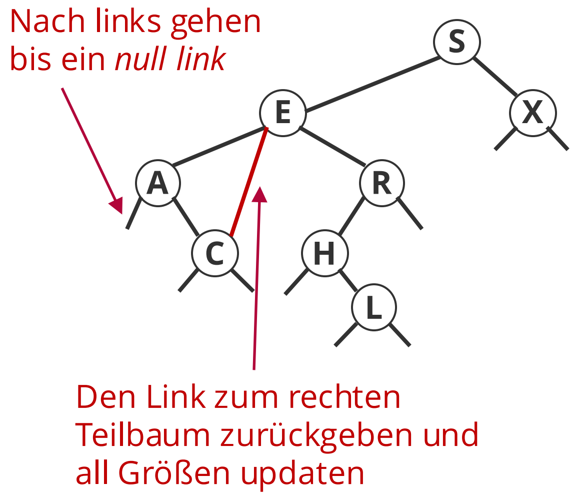
* Bei gleicher Semantik wie beim BST: wenn der Schlüssel schon vorhanden ist, soll nur der Wert überschrieben werden.

Überblick

1. Begriffe
   - Bäume als spezielle Graphen
   - Eigenschaften
2. Binäre Suchbäume
   - Einfügen
   - **Entfernen (*Hibbard deletion*)**

Binäre Suchbäume: `min` und `remove_min`

Finden und Entfernen des Minimums



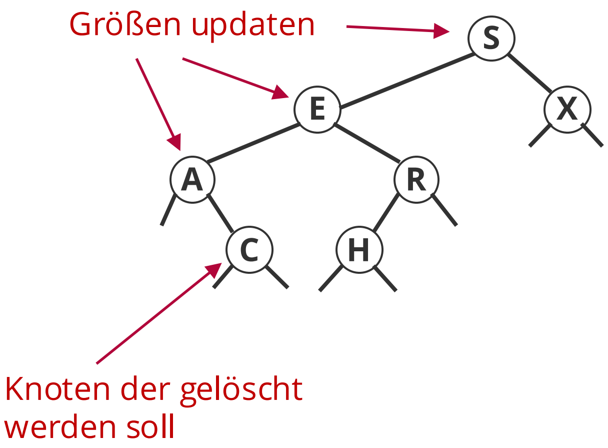

```cpp
// finds the minimum of a tree rooted at n
Node* min(Node* n) const {
    if (n->left == nullptr)
        return (n);
    else
        return (min(n->left));
}
```

```cpp
// removes the minimum node of a tree rooted at n
Node* remove_min(Node* n) const {
    if (n->left == nullptr) {
        return (n->right);
    } else {
        n->left = remove_min(n->left);
        n->size = 1 + size(n->left) + size(n->right);
        return (n);
    }
}
```

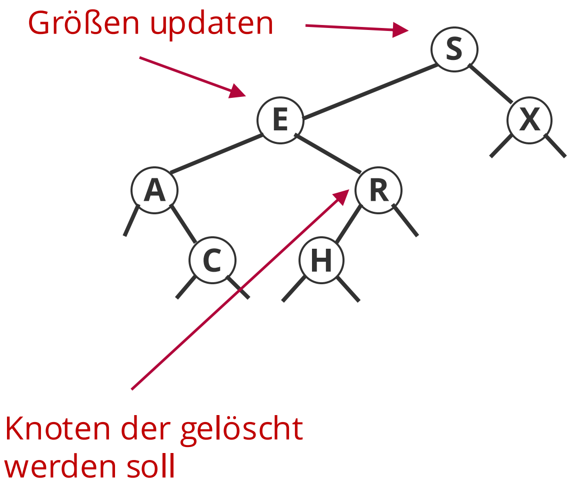
bst.h

Binäre Suchbäume: `remove` (*Hibbard deletion*)

- **Fall 1 (keine Kinder)**: Lösche **C**



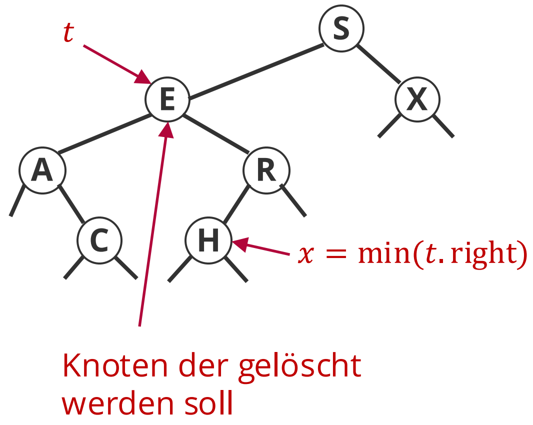
- **Fall 2 (ein Kind)**: Lösche **R**

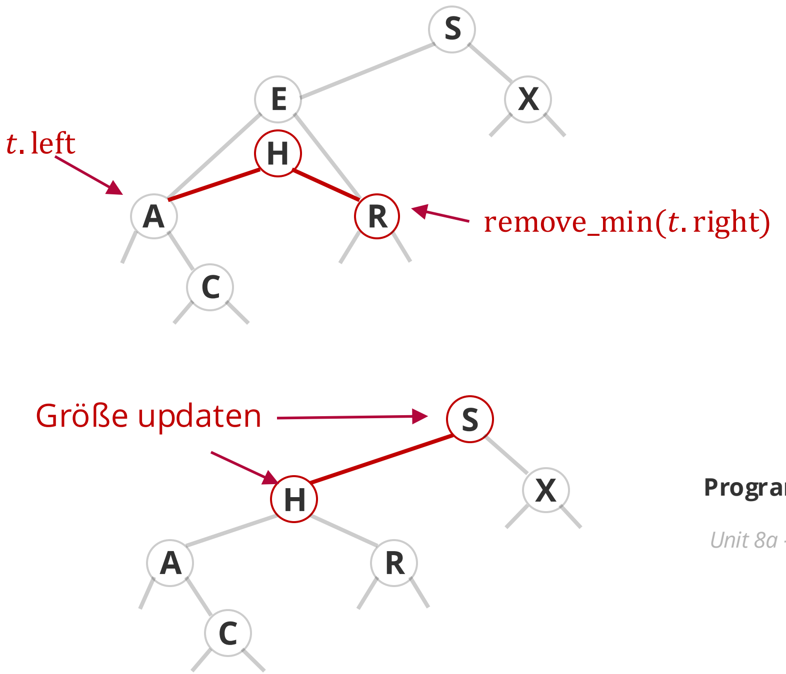


Thomas Hibbard
(1929 – 2016)

Hibbard Deletion: Fall 3 (zwei Kinder)

- Lösche **E**





Programmiertechnik II

Unit 8a - Bäume

Binäre Suchbäume: remove

```
// removes the node with the a key from a tree rooted at n
Node* remove(Node* n, const Key& key) const {
    if (n == nullptr) return (nullptr);
    if (key < n->key) n->left = remove(n->left, key);
    else if (key > n->key) n->right = remove(n->right, key);
    else {
        // one child case: just return the other child
        if (n->right == nullptr || n->left == nullptr) {
            Node* t = (n->right) ? n->right : n->left;
            delete (n);
            return (t);
        }

        // two child chase
        Node* t = n;
        n = min(t->right);
        n->right = remove_min(t->right);
        n->left = t->left;
        delete t;
    }
    n->size = 1 + size(n->left) + size(n->right);
    return (n);
}

public:
 void remove(const Key& key) { root = remove(root, key); }
```

- **Satz**: Die durchschnittliche Anzahl von Schritten von Löschoperationen ist $\sqrt{n}$.

bst.h

Viel Spaß bis zur nächsten Vorlesung!