Programmiertechnik II

Ralf Herbrich

Balancierte Bäume

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	$1.39 \cdot \log_2(n)$	$1.39 \cdot \log_2(n)$	$\sqrt{n}$
Ziel	$\log_2(n)$	$\log_2(n)$	$\log_2(n)$	$\log_2(n)$	$\log_2(n)$	$\log_2(n)$

- **Herausforderung**: Garantierte Laufzeit ist $O(\log_2(n))$!

Überblick

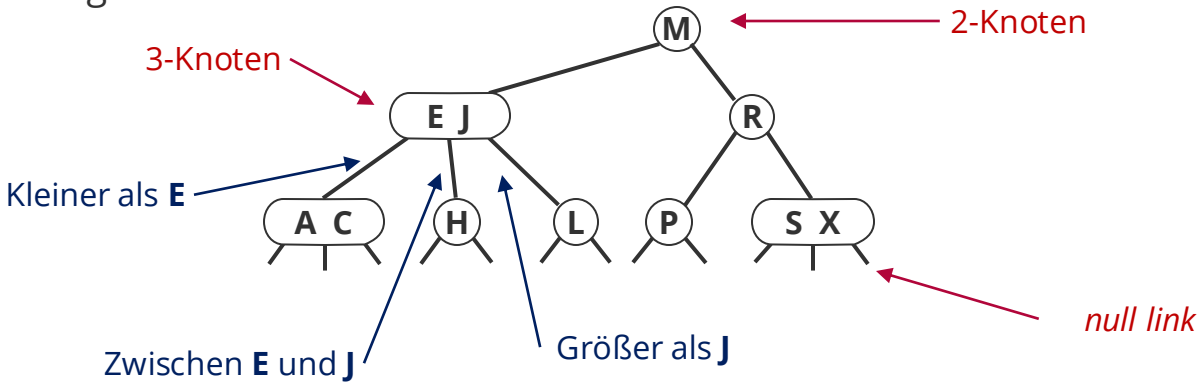
1. 2-3 Bäume
2. (Links-Neigende) Rot-Schwarz Bäume (*left-leaning red-black trees*)
   - Rotationen und Farbwechsel
   - Einfügen

Überblick

1. **2-3 Bäume**
2. (Links-Neigende) Rot-Schwarz Bäume (*left-leaning red-black trees*)
   - Rotationen und Farbwechsel
   - Einfügen

2-3 Bäume

- **Idee**: Wir erlauben ein oder zwei Suchschlüssel!
  - **2-Knoten**: Einen Suchschlüssel a und zwei Kinder (Links: alle Knoten kleiner als a; Rechts: alle Knoten größer als a)
  - **3-Knoten**: Zwei Suchschlüssel a und b und drei Kinder (Links: alle Knoten kleiner als a; Mitte: alle Knoten größer als a aber kleiner als b; Rechts: alle Knoten größer als b)
- **Symmetrische Ordnung** ergibt weiterhin alle Knoten in richtiger Reihenfolge
- **Satz (Perfekte Balance)**: Jeder Pfad von der Wurzel zu den *null links* hat die gleiche Länge!



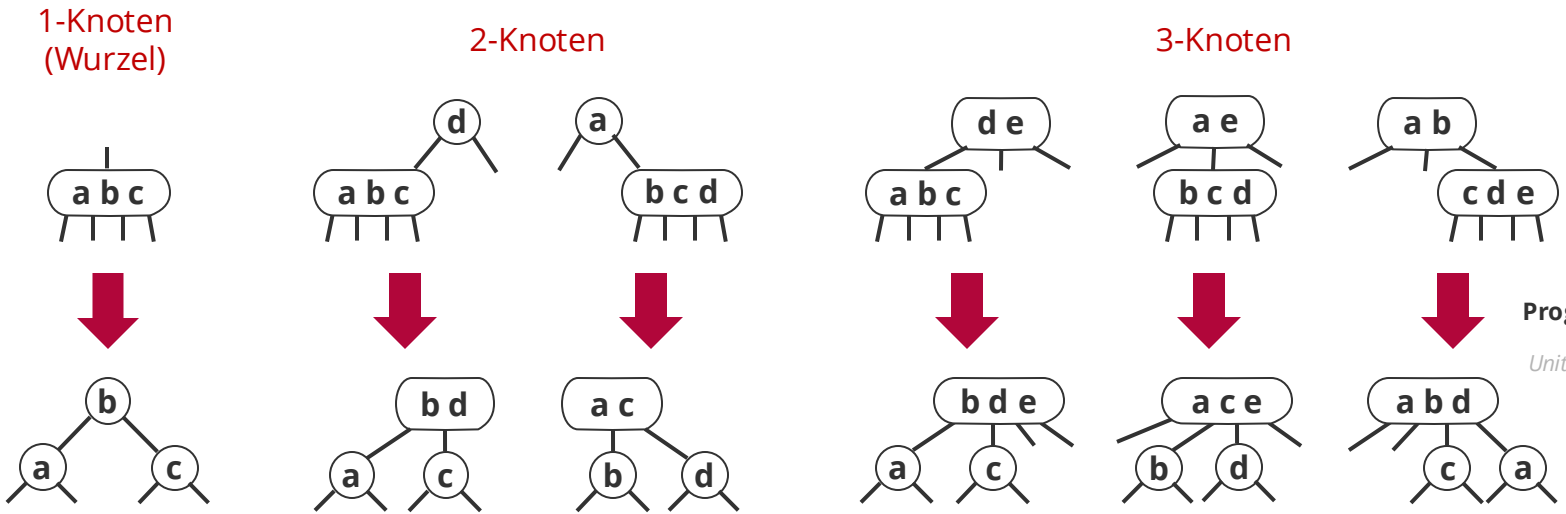
John Hopcroft
(1939 –)

2-3 Bäume: `Get`

- **Suche** (`get`) kann genauso ausgeführt werden wir bei binären Suchbäumen
  - **2-Knoten**: Vergleiche den Schlüssel mit dem Knotenschlüssel
    - Wenn Schlüssel und Knotenschlüssel gleich sind: fertig!
    - Suche links weiter wenn der Schlüssel kleiner als der Knotenschlüssel ist
    - Suche rechts weiter wenn der Schlüssel größer als der Knotenschlüssel ist
  - **3-Knoten**: Vergleiche den Schlüssel mit den Knotenschlüsseln
    - Wenn Schlüssel und einer der Knotenschlüssel gleich sind: fertig!
    - Suche links weiter wenn der Schlüssel kleiner als der kleine Knotenschlüssel ist
    - Suche rechts weiter wenn der Schlüssel größer als der große Knotenschlüssel ist
    - Ansonsten suche in der Mitte weiter
- **Satz**: Bei Suchoperationen in einem 2-3 Baum mit n Schlüsseln werden garantiert nicht mehr als $\log_2(n)$ Knoten besucht!
  - **Beweis**: Die Höhe des 2-3 Baums liegt zwischen $\log_3(n) = \frac{\log_2(n)}{\log_2 3}$ (wenn alle Knoten 3-Knoten sind) und $\log_2(n)$ (wenn alle Knoten 2-Knoten sind).

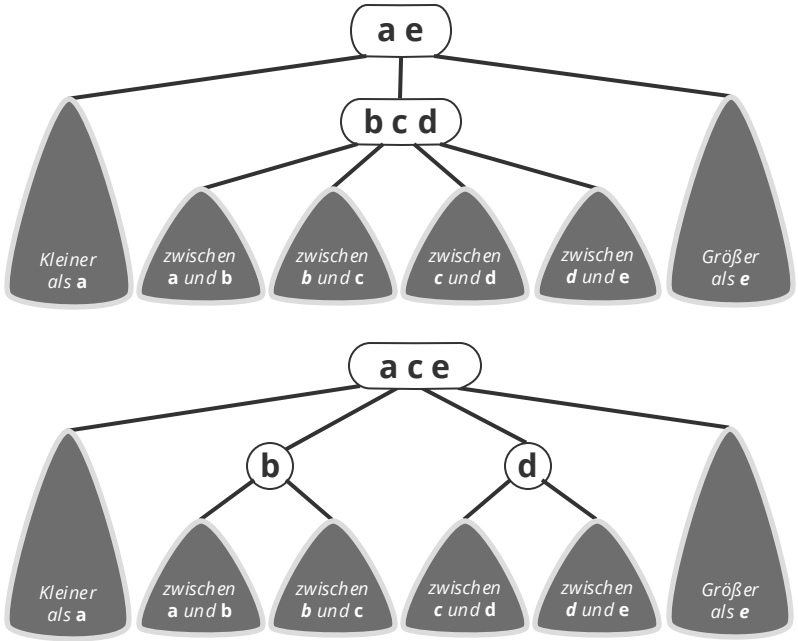
2-3 Bäume: `Put`

- **Einfügen** (`put`) geschieht in zwei Schritten:
  1. Einfügen in den Blattknoten
  2. Mögliche (temporäre) 4-Knoten auf dem Weg zurück zur Wurzel auflösen
- 6 Regeln zur Auflösung von 4-Knoten



Lokale Transformationen von 2-3 Bäume

- Einen temporären 4-Knoten aufzulösen ist eine **lokale** Transformation!



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	$1.39 \cdot \log_2(n)$	$1.39 \cdot \log_2(n)$	$\sqrt{N}$
2-3 Bäume	$c \cdot \log_2(n)$	$c \cdot \log_2(n)$	$c \cdot \log_2(n)$	$c \cdot \log_2(n)$	$c \cdot \log_2(n)$	$c \cdot \log_2(n)$

Konstante c hängt von der Implementierung ab

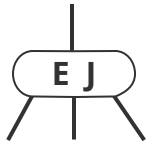
- **Direkte Implementierung** ist kompliziert wegen verschiedenen Knotentypen!

Überblick

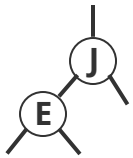
1. 2-3 Bäume
2. **(Links-Neigende) Rot-Schwarz Bäume (*left-leaning red-black trees*)**
   - Rotationen und Farbwechsel
   - Einfügen

Darstellung von 2-3 Bäumen mit Binären Suchbäumen

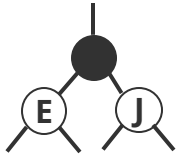
- **Herausforderung**: Wie kann man einen 3-Knoten darstellen?



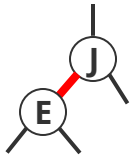
- **Idee 1**: Regulärer Binärsuchbaum
  - Unmöglich einen 2-Knoten und 3-Knoten zu unterscheiden
  - Keine eindeutige Abbildung von BST zu 2-3 Bäumen möglich



- **Idee 2**: Regulärer Binärsuchbaum mit extra Verbindungsknoten
  - Zusätzlicher Knoten und Kante notwendig
  - Code sehr unübersichtlich mit vielen Spezialfällen

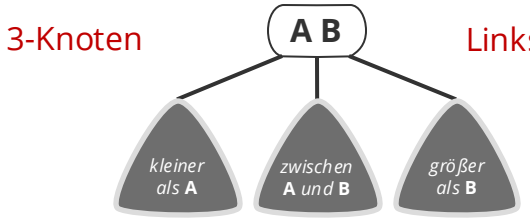


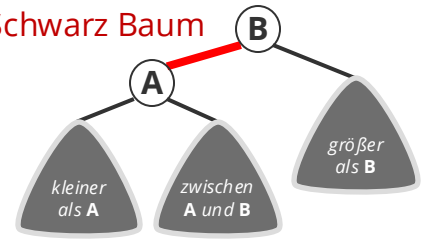
- **Idee 3**: Regulärer Binärsuchbaum mit extra (roter) Verbindung
  - Elegante Lösung ohne zusätzlichen Knoten und Kante
  - Überall in Praxis und Industrie benutzt

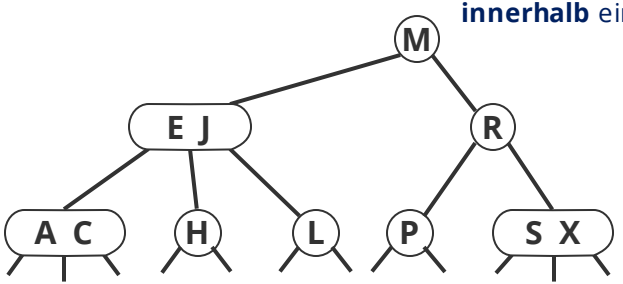


(Links-neigende) Rot-Schwarz Bäume (*left-leaning red-black trees*)

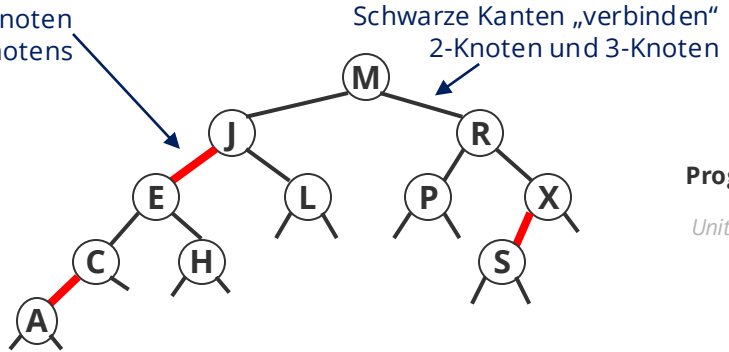
1. Eindeutige Repräsentation von 2-3 Bäume als binäre Suchbäume
2. Eine rote Kante, die einen 3-Knoten darstellt, **muss** die **linke Kante** sein.







2-3 Baum



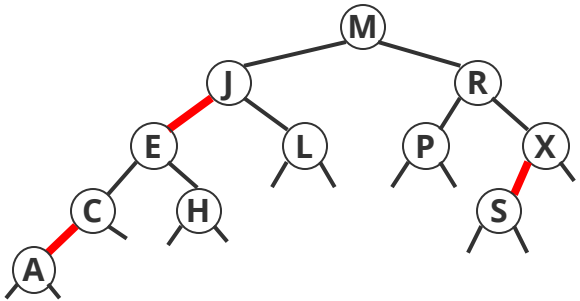
korrespondierender Rot-Schwarz Baum

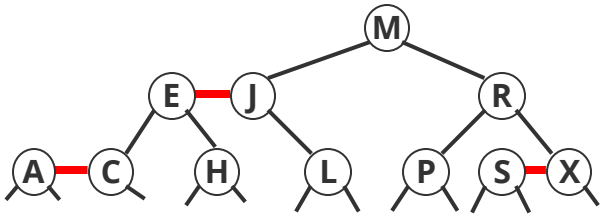
Robert Sedgewick
(1946 –)

Leonidas J. Guibas

Eigenschaften von links-neigenden Rot-Schwarz Bäumen

- Ein *left-leaning red-black* Baum hat folgende **Eigenschaften**:
  1. Kein Knoten hat zwei rote Kanten (das wäre ein temporärer 4-Knoten!)
  2. Rote Kanten neigen nach links.
  3. Jeder Pfad von der Wurzel zu null links hat die gleiche Anzahl schwarzer Kanten!





Links-neigende Rot-Schwarz Bäume: `Get`

- **Beobachtungen**:
  1. Suche ist identisch zu normalen Suchbäumen!
  2. Farbe der Kante kann eindeutig in Farbe des Kindknotens kodiert werden!

```
// a helper binary tree node data type
struct Node {
    Key key;
    Value val;
    Node* left;
    Node* right;
    bool red;
    int size;

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

// returns whether the link from the parent is red
bool is_red(const Node* n) const {
    return (n ? n->red : false);
}
```

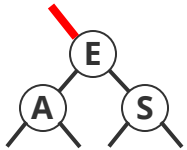
```
// use recursion to find the correct node
const Value* get(const Node* n, const Key& key) const {
    if (n == nullptr) return (nullptr);
    if (key < n->key) return (get(n->left, key));
    if (key > n->key) return (get(n->right, key));
    return &(n->val);
}

public:
// gets a value for a given key
const Value* get(const Key& key) const {
    return (get(root, key));
}
```

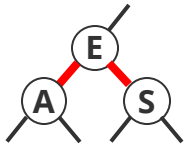
redblack_bst.h

Links-neigende Rot-Schwarz Bäume: `Put`

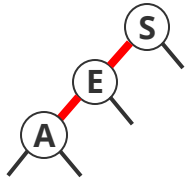
- **Grundlegende Strategie**: Erhalte die 1-1 Korrespondenz mit 2-3 Bäumen!
- **Konkret:**
  - **Symmetrische Ordnung**: Alle Knoten bleiben in der richtigen Reihenfolge
  - **Perfekte Balance**: Jeder Pfad von der Wurzel zu null links hat die gleiche Anzahl schwarzer Kanten
- **Problemfälle**:



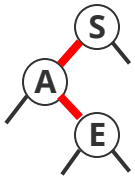
Rechts-neigende rote Kante



Zwei rote Kanten



Links-Links Rote Kanten



Links-Rechts Rote Kanten

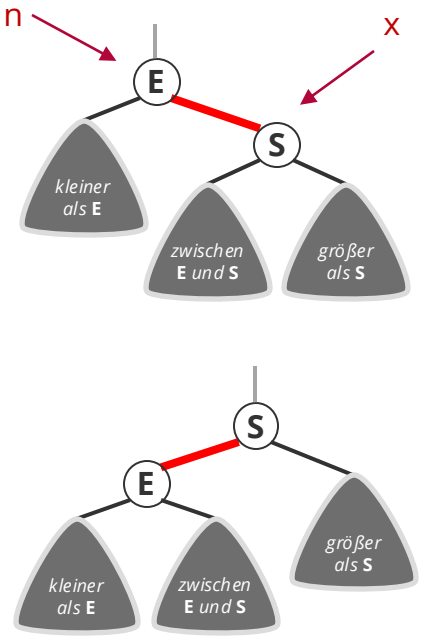
- **Lösung**: Anwendung von elementaren Operationen **Rotation** und **Farbwechsel**

Überblick

1. 2-3 Bäume
2. (Links-Neigende) Rot-Schwarz Bäume (*left-leaning red-black trees*)
   - **Rotationen und Farbwechsel**
   - Einfügen

Elementare Operation: Linksrotation

- **Invarianten**: Symmetrische Ordnung und Perfekte Balance bleiben gewahrt!

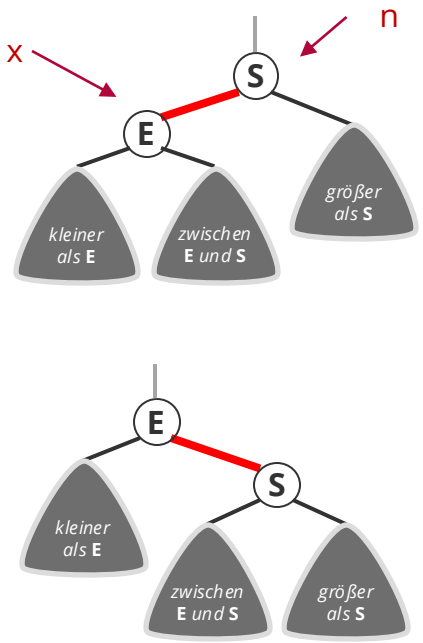



```java
// make a right-leaning link lean to the left
Node* rotate_left(Node* n) const {
    Node* x = n->right;
    n->right = x->left;
    x->left = n;
    x->red = n->red;
    n->red = true;
    x->size = n->size;
    n->size = size(n->left) + size(n->right) + 1;
    return x;
}
```

redblack_bst.h

Elementare Operation: Rechtsrotation

- **Invarianten**: Symmetrische Ordnung und Perfekte Balance bleiben gewahrt!

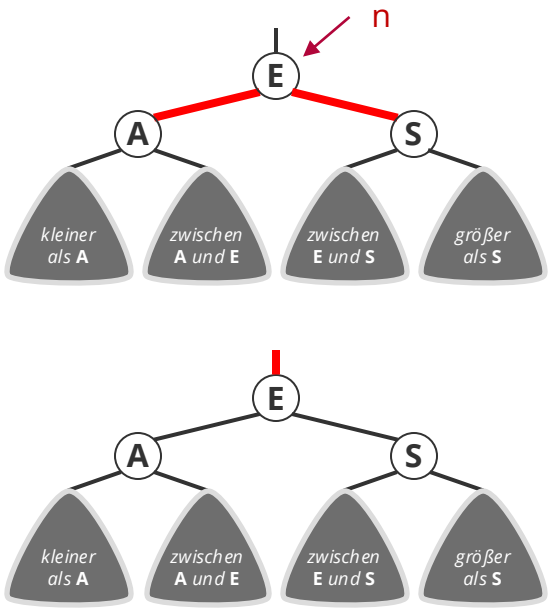



```
// make a left-leaning link lean to the right
Node* rotate_right(Node* n) const {
    Node* x = n->left;
    n->left = x->right;
    x->right = n;
    x->red = n->red;
    n->red = true;
    x->size = n->size;
    n->size = size(n->left) + size(n->right) + 1;
    return x;
}
```

redblack_bst.h

Elementare Operation: Farbwechsel

- **Invarianten**: Symmetrische Ordnung und Perfekte Balance bleiben gewahrt (jeder schwarze Pfad wird um eins länger)!




```
// flip the colors of the link pointing to the node
// and its two children links
void flip_colors(Node* n) const {
    n->red = !n->red;
    if (n->left != nullptr)
        n->left->red = !n->left->red;
    if (n->right != nullptr)
        n->right->red = !n->right->red;
    return;
}
```

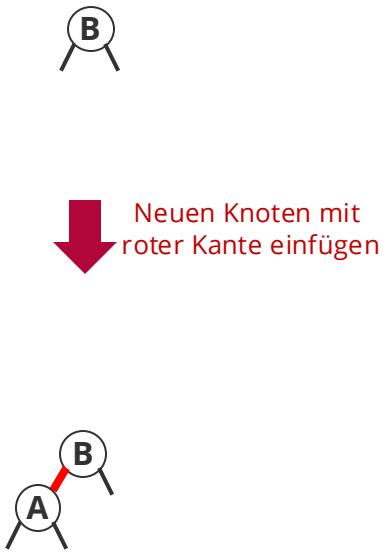
redblack_bst.h

Überblick

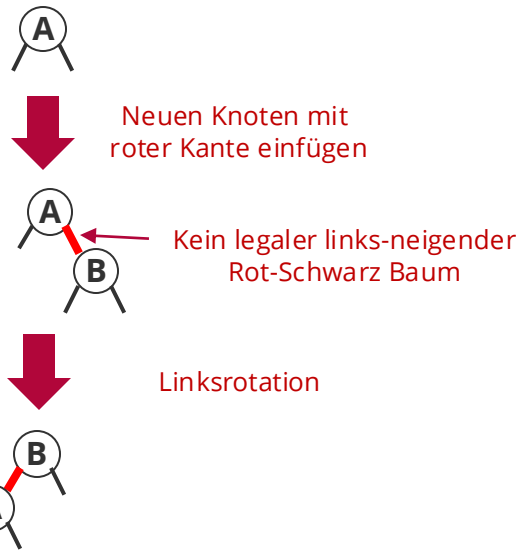
1. 2-3 Bäume
2. (Links-Neigende) Rot-Schwarz Bäume (*left-leaning red-black trees*)
   - Rotationen und Farbwechsel
   - **Einfügen**

Warmup 1: Einfügen in einen Baum mit 1 Knoten

Links einfügen (**A**)



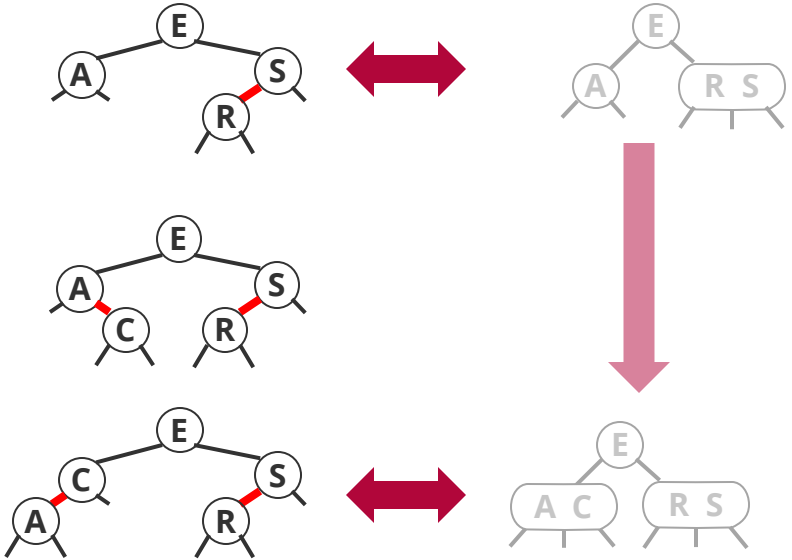
Rechts einfügen (**B**)



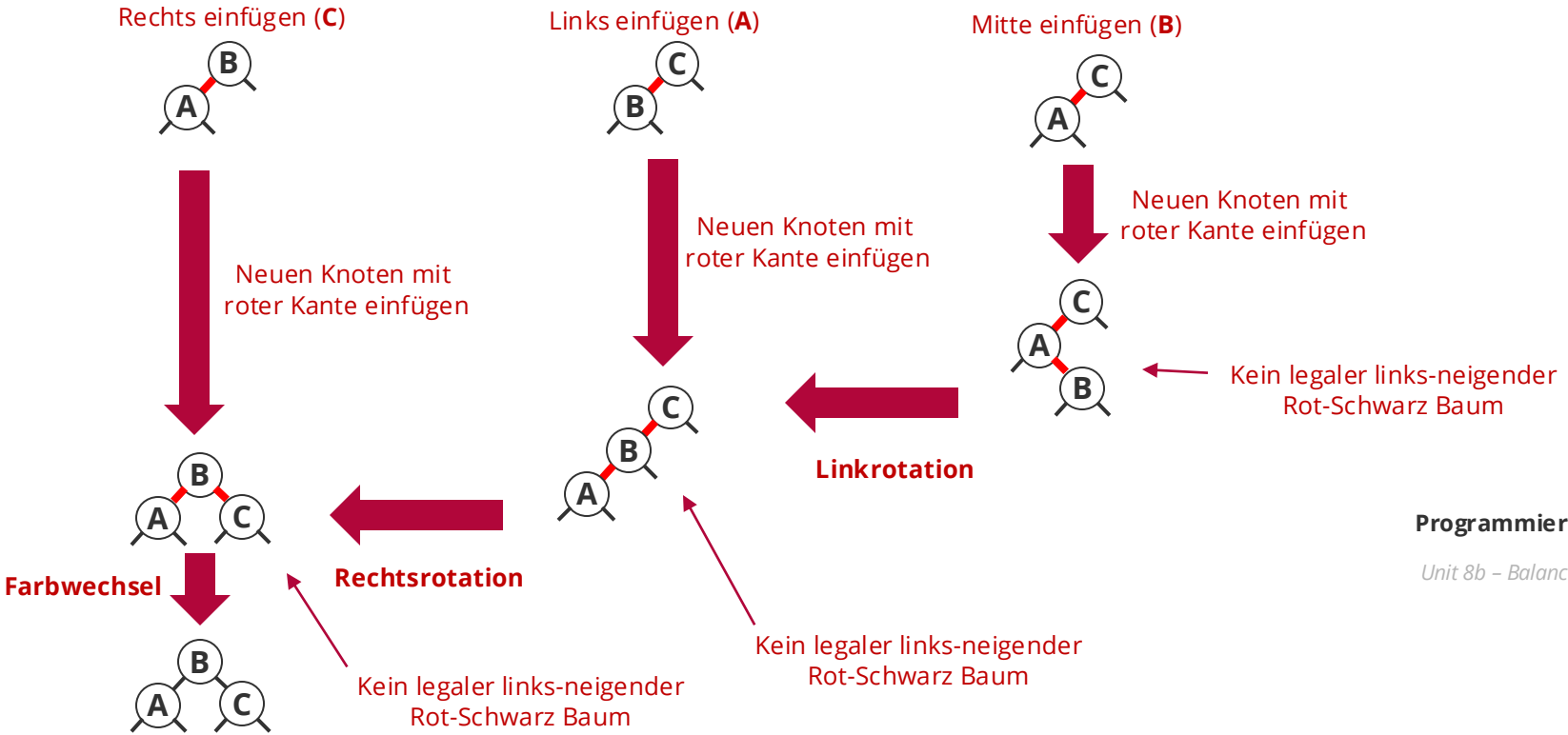
Fall 1: Einfügen in einen Baum mit finalem 2-Knoten

- **Schritt 1**: Färbe die neue Kante rot ← Erhält symmetrische Ordnung und perfekte Balance
- **Schritt 2**: Wenn die neue Kante eine Rechtskante war: Linksrotation ← Erhält Farbinvarianz

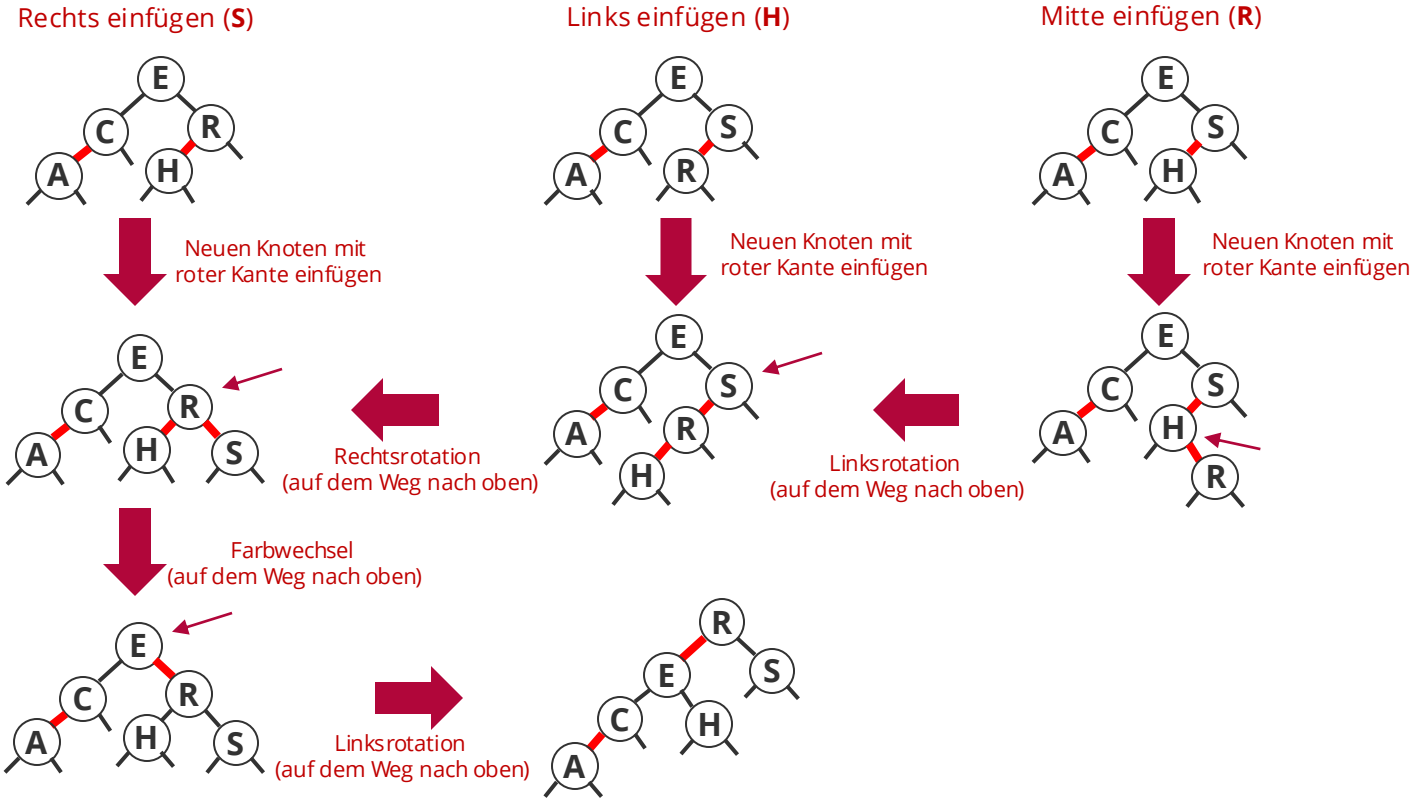
C einfügen



Warmup 2: Einfügen in einen Baum mit 2 Knoten



Fall 2: Einfügen in einen Baum mit finalem 3-Knoten



Links-neigende Rot-Schwarz Bäume: `Put`

```
// uses recursion to put a key-value pair into the tree      You, 3 day
Node* put(Node* n, const Key& key, const Value& val) const {
    if (n == nullptr) {
        auto node = new Node(key, val, true, 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;

    // fix-up any right-leaning links
    if (is_red(n->right) && !is_red(n->left)) n = rotate_left(n);
    if (is_red(n->left) && is_red(n->left->left)) n = rotate_right(n);
    if (is_red(n->left) && is_red(n->right)) flip_colors(n);

    n->size = size(n->left) + size(n->right) + 1;
    return (n);
}

public:
// put a key-value pair into the table
void put(const Key& key, const Value& val) {
    root = put(root, key, val);
    root->red = false;
    return;
}
```

Füge den neuen Knoten mit einer roten Kante ein

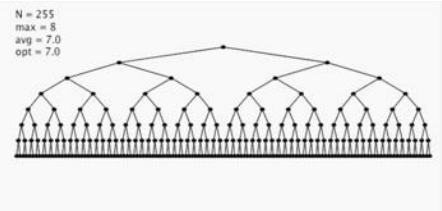
Rekursion bis zum Blatt, wo eingefügt wird und update des neuen Knotens (durch Rotation oder Erzeugen)

Korrektur der 2-3 und links-neigend Eigenschaften auf dem Weg „nach oben“

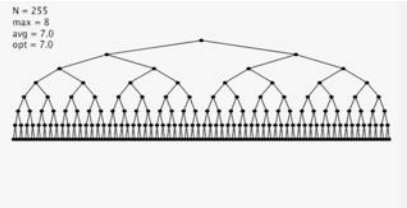
redblack_bst.h

Links-neigende Rot-Schwarz Bäume: Visualisierung

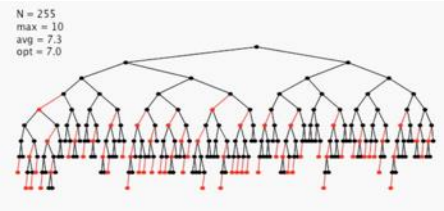
255 Schlüssel in aufsteigender Reihenfolge



255 Schlüssel in absteigender Reihenfolge



255 Schlüssel in zufälliger Reihenfolge



- **Optimalität**: Geht's noch besser?
  - Ja, wenn man die Links-Neigung wegnimmt (aber Code komplizierter)
  - Alle balancierten Bäume basieren auf Rotationen und lokalen Transformationen (AVL)
- Knoten **löschen** sind auch hier komplizierter aber auch durch lokale Transformationen lösbar (Übungsaufgabe?)
- Rot-Schwarz Bäume werden in sehr vielen Standardbibliotheken benutzt (zum Beispiel STL `maps`)

Zusammenfassung

- **2-3 Bäume**
  - 2-3 Bäume haben zwei Arten von Knoten
  - 2-3 Bäume sind *immer* balanciert
  - Balance kann allein durch lokale Operationen hergestellt werden
- **(Links-Neigende) Rot-Schwarz Bäume (*left-leaning red-black trees*)**
  - Links-neigende Rot-Schwarz Bäume kodieren die 2- und 3-Knoten eindeutig
  - Sind ein Spezialfall von Rot-Schwarz Bäumen, die einfachere Algorithmen erlauben
  - Zentrale Operation sind Rotation (links & rechts) und Farbwechsel – alles lokale Operationen
  - Beim Einfügen werden die etwaigen 4-Knoten auf dem Weg „zurück“ zur Wurzel wieder behoben
  - Balancierte Bäume werden in allen Standardimplementationen von Symboltabellen benutzt (z.B. in der STL)

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Viel Spaß bis zur nächsten Vorlesung!