Datenstrukturen und Algorithmen

Vorlesung 13: Zusammenfassung der Vorlesungen 9-12

Prof. Dr. Erika Ábrahám

Theorie Hybrider Systeme
Informatik 2

http://ths.rwth-aachen.de/teaching/ss-14/
datenstrukturen-und-algorithmen/

Diese Präsentation verwendet in Teilen Folien von Joost-Pieter Katoen.

20. Juni 2014

RWTH AACHEN UNIVERSITY

Übersicht

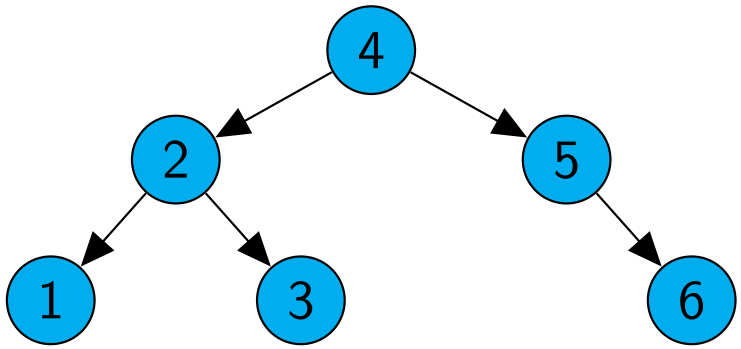
Übersicht

Binäre Suchbäume (BST)

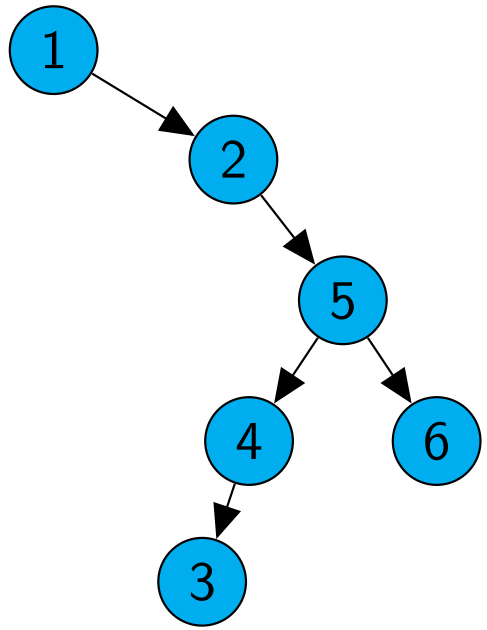
Binärer Suchbaum

Ein binärer Suchbaum (binary search tree BST) ist ein Binärbaum, der Elemente mit Schlüsseln als Knoten enthält, wobei der Schlüssel jedes Knotens

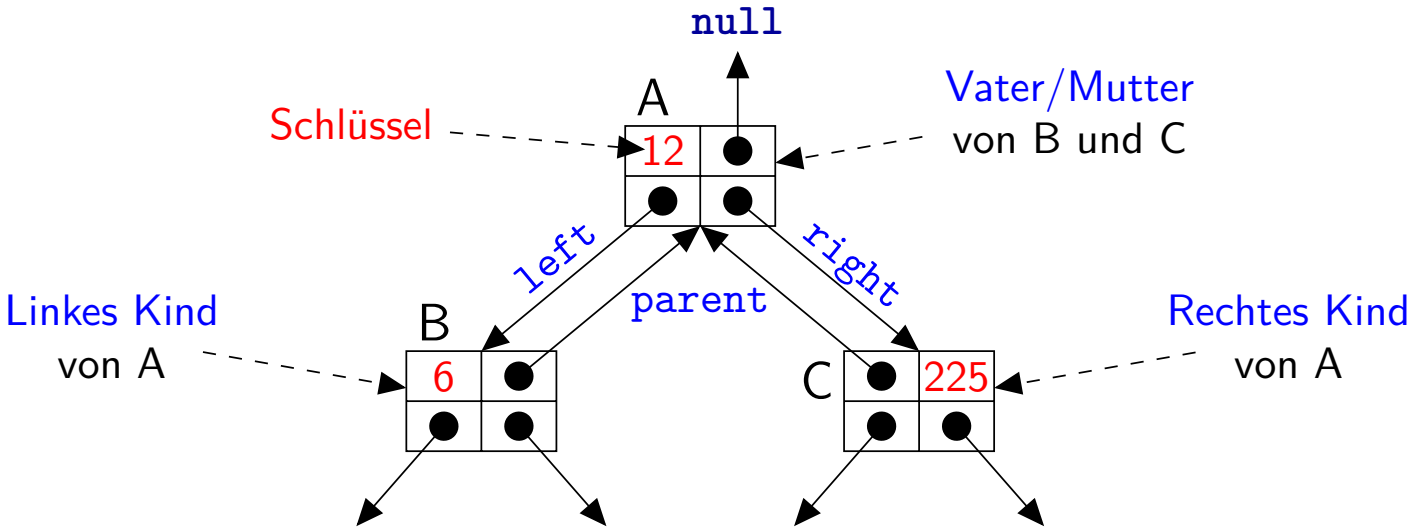
- mindestens so groß ist, wie jeder Schlüssel im linken Teilbaum und
- höchstens so groß ist, wie jeder Schlüssel im rechten Teilbaum.



Zwei binäre Suchbäume, die jeweils die Schlüssel 1, 2, 3, 4, 5, 6 enthalten.



BST: Implementierung

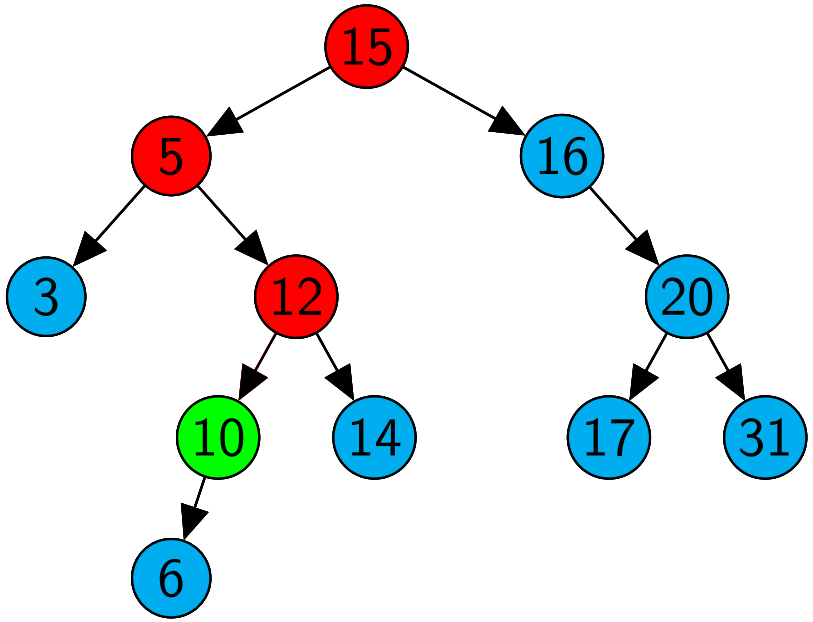



```
1  class Node {
2    int key;
3    Node left, right;
4    Node parent;
5    // ... evtl. eigene Datenfelder
6  };

8  class Tree { Node root; };
```

BST: Suche nach Schlüssel k

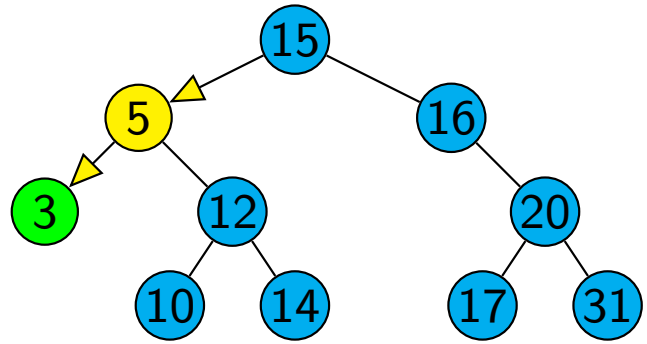
1. Sei `node` die Wurzel
2. Wenn `node==null`, fertig (k ist nicht im Baum).
3. Sei k' der Schlüssel des besuchten Knotens.
4. Wenn $k' = k$, fertig (Schlüssel gefunden).
5. Wenn $k' > k$, steige zum linken Kind herunter.
6. Wenn $k' < k$, steige zum rechten Kind herunter.
7. Gehe zu 2.



- Worst-Case Komplexität: $\Theta(h)$

BST: Suche nach Minimum

Steige im Baum immer entlang des linken Kindes herunter, bis ein Knoten ohne linkes Kind erreicht wird.



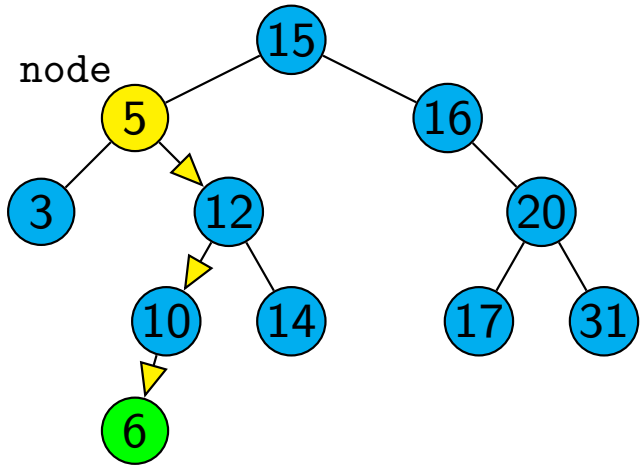
- Komplexität: $\Theta(h)$ bei Baumhöhe h.
- Analog kann das Maximum gefunden werden.

BST: Nachfolger suchen

BST: Nachfolger suchen

Der rechte Teilbaum existiert:

Der Nachfolger ist der kleinste Knoten im rechten Teilbaum.



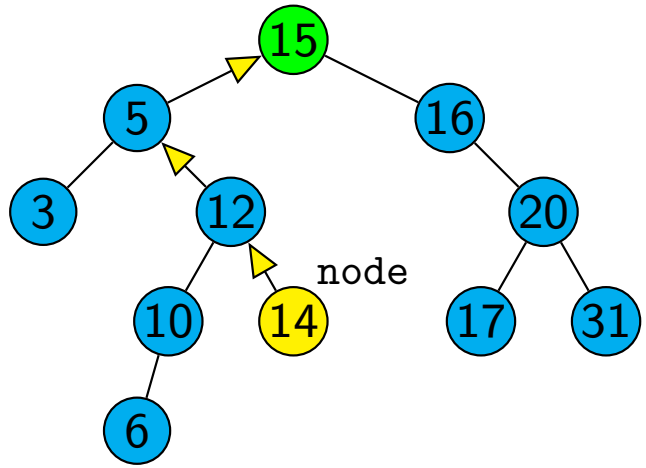
BST: Nachfolger suchen

Der rechte Teilbaum existiert:

Der Nachfolger ist der kleinste Knoten im rechten Teilbaum.

Andernfalls:

Der Nachfolger ist der jüngste Vorfahre, dessen linker Teilbaum `node` enthält.



BST: Nachfolger suchen

Der rechte Teilbaum existiert:

Der Nachfolger ist der kleinste Knoten im rechten Teilbaum.

Andernfalls:

Der Nachfolger ist der jüngste Vorfahre, dessen linker Teilbaum `node` enthält.

- Komplexität: $\Theta(h)$ bei Baumhöhe h.
- Analog kann der Vorgänger gefunden werden.

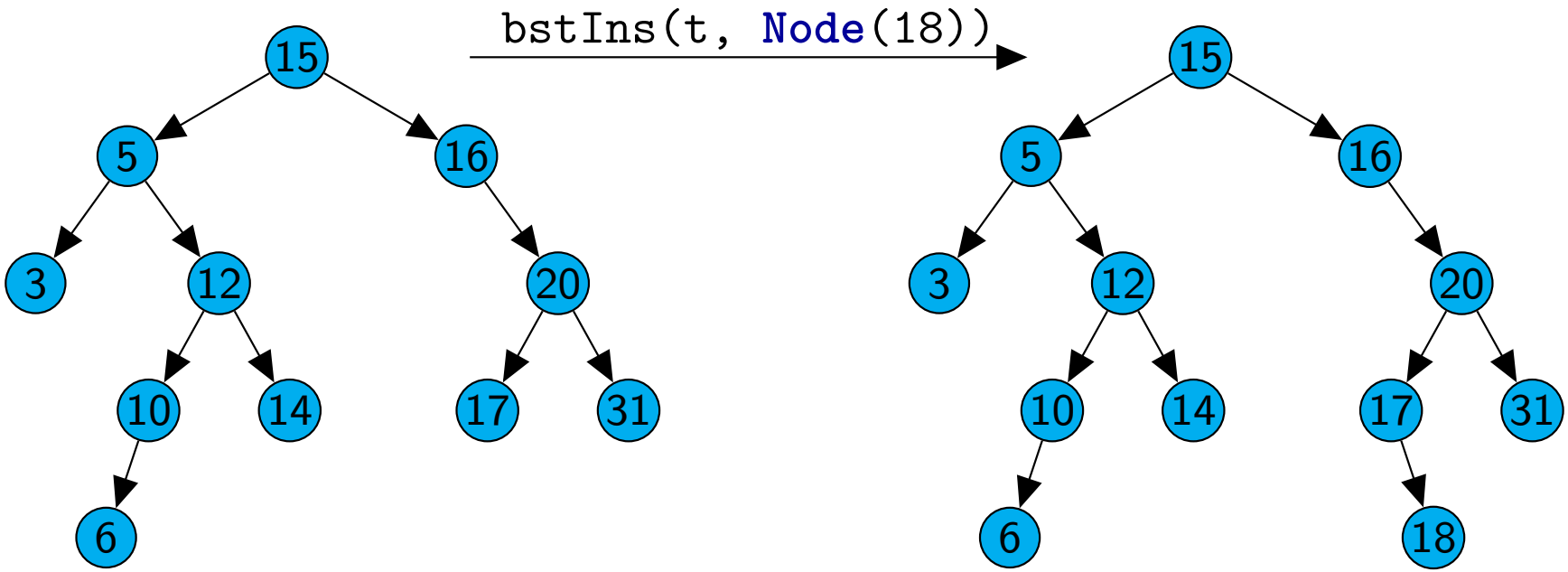
BST: Einfügen eines Knotens

Suche einen geeigneten, freien Platz:

Wie bei der regulären Suche, außer dass, selbst bei gefundenem Schlüssel, weiter abgestiegen wird, bis ein Knoten ohne entsprechendes Kind erreicht ist.

Hänge den neuen Knoten an:

Verbinde den neuen Knoten mit dem gefundenen Vaterknoten.



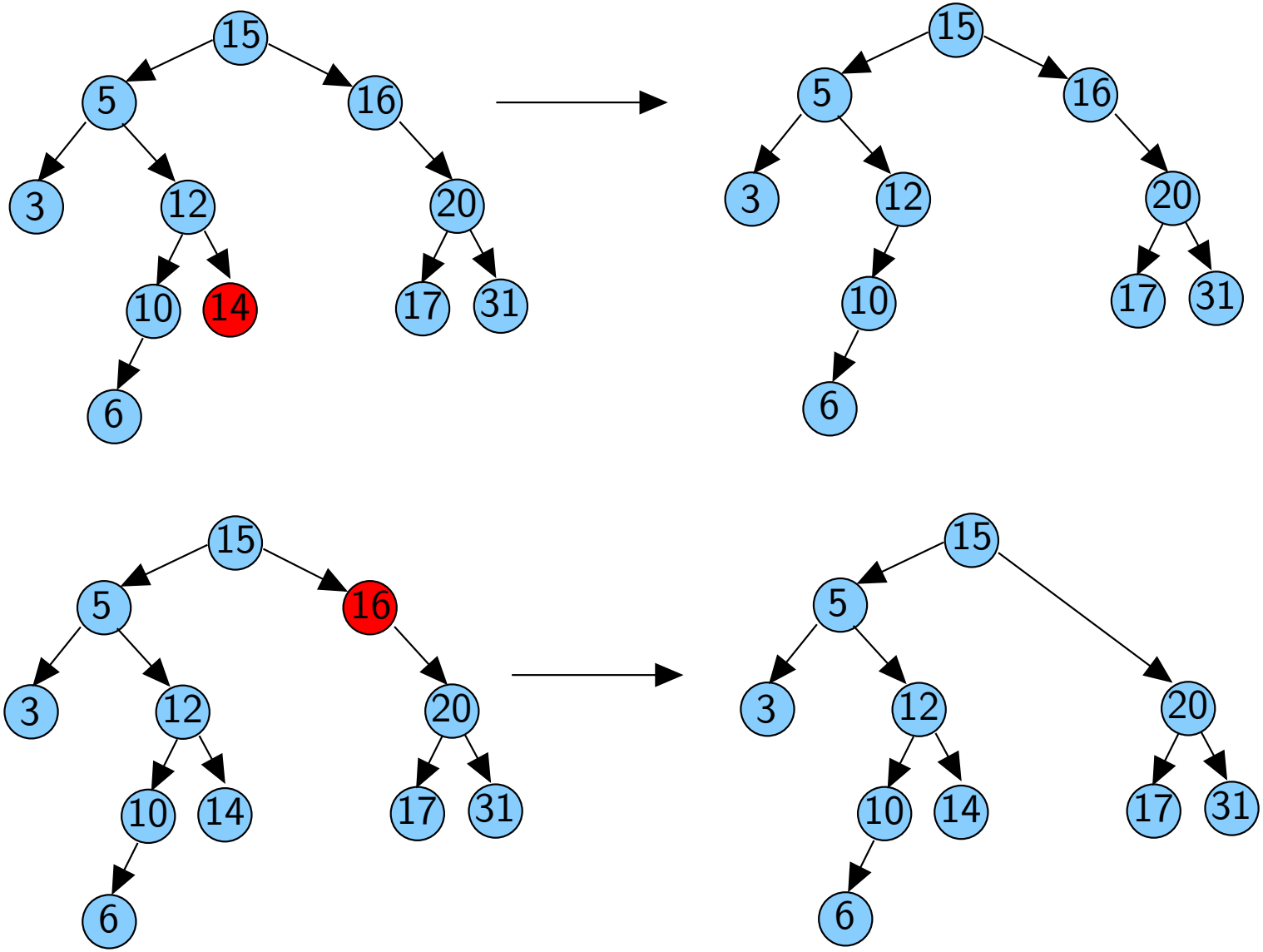
- Komplexität: $\Theta(h)$, wegen der Suche.

BST: Weitere Operationen

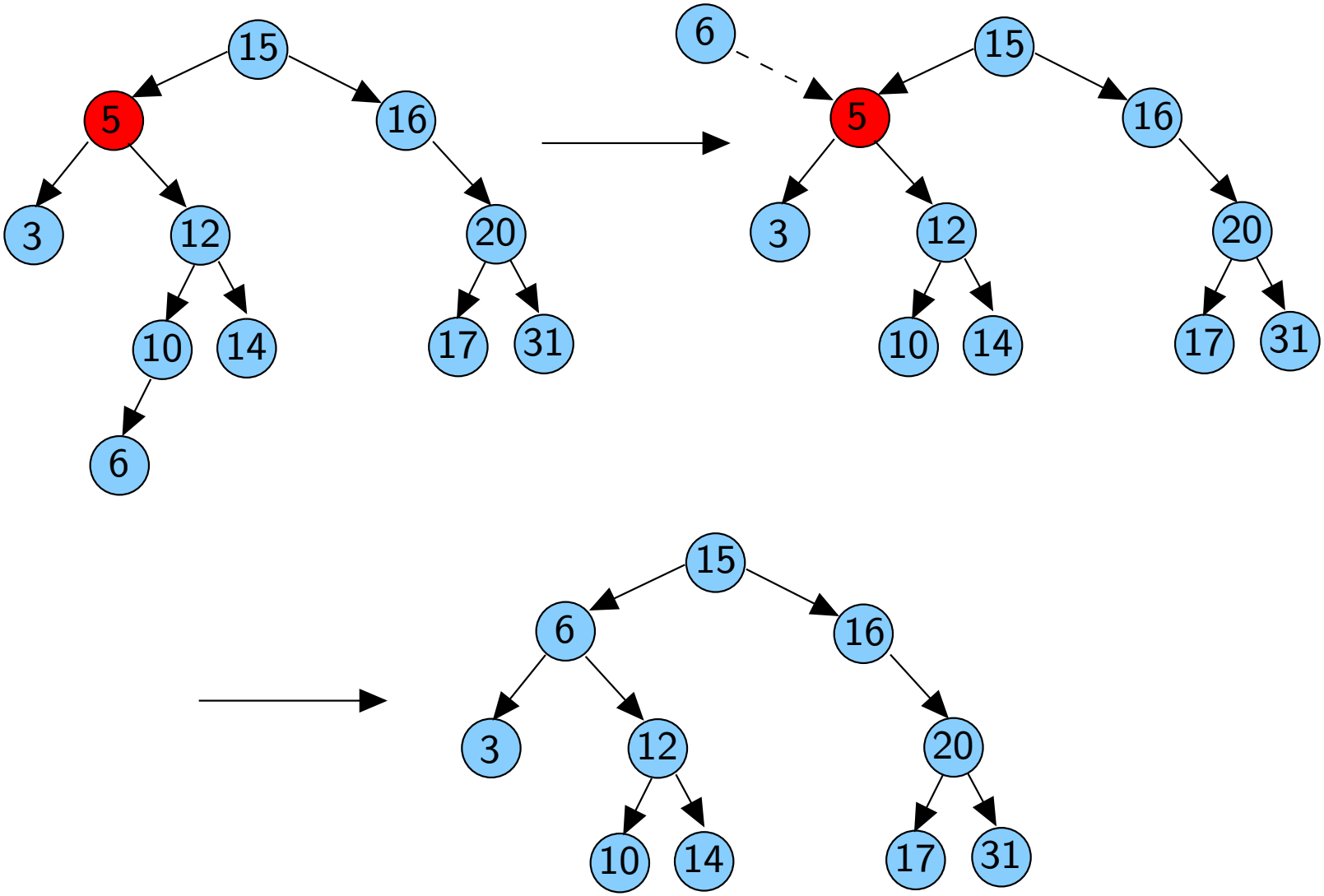
- Ersetzen eines Knotens
- Austauschen eines Teilbaumes

Zeitkomplexität in $\Theta(1)$.

BST: Löschen eines Knotens



BST: Löschen eines Knotens



BST: Sortieren in linearer Zeit

Sortieren

Eine Inorder Traversierung eines binären Suchbaumes gibt alle Schlüssel im Suchbaum in sortierter Reihenfolge aus.

Zeitkomplexität

Zeitkomplexität in $\Theta(n)$.

BST: Komplexität der Operationen

Operation	Zeit
`bstMin/bstMax`	$\Theta(h)$
`bstSearch`	$\Theta(h)$
`bstSucc/bstPred`	$\Theta(h)$
`bstIns`	$\Theta(h)$
`bstDel`	$\Theta(h)$
`bstSort`	$\Theta(n)$

- Sortieren ist in linearer Zeit.
- Alle anderen Operationen sind linear in der Höhe h des BSTs.
- Die Höhe ist in $\mathcal{O}(\log n)$, wenn der Baum balanciert ist.

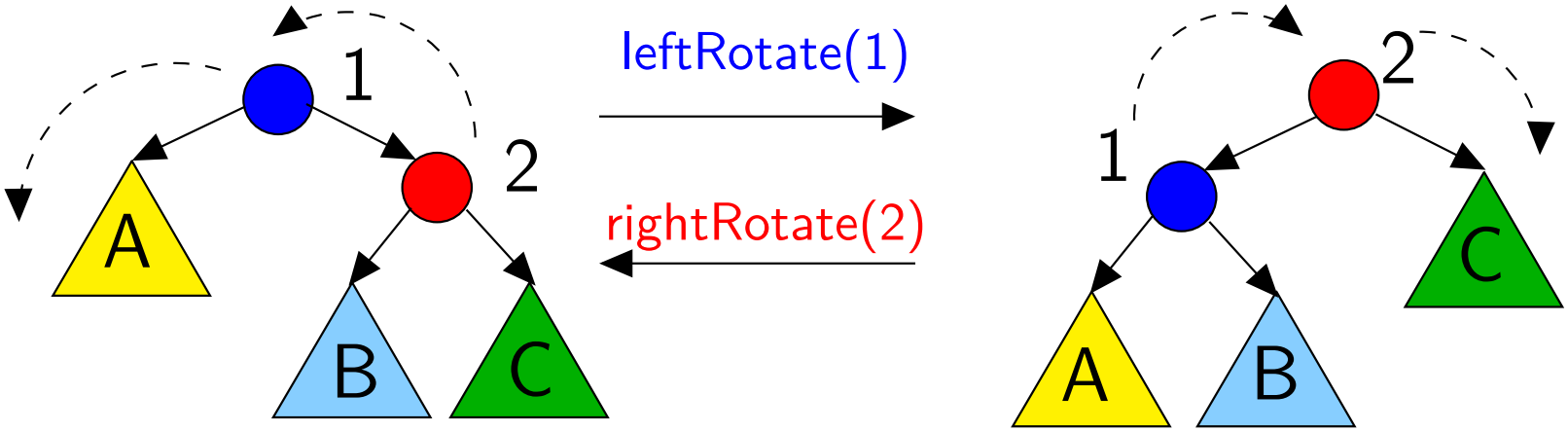
Übersicht

AVL-Bäume

AVL-Baum

- Ein AVL-Baum ist ein balancierter BST, bei dem für jeden Knoten die Höhe der beiden Teilbäume höchstens um 1 differiert.
- Nach jeder (kritischen) Operation wird die Balance durch Rotationen wiederhergestellt. Dies ist in $\Theta(h)$ möglich!

Rotationen: Eigenschaften und Komplexität



Lemma

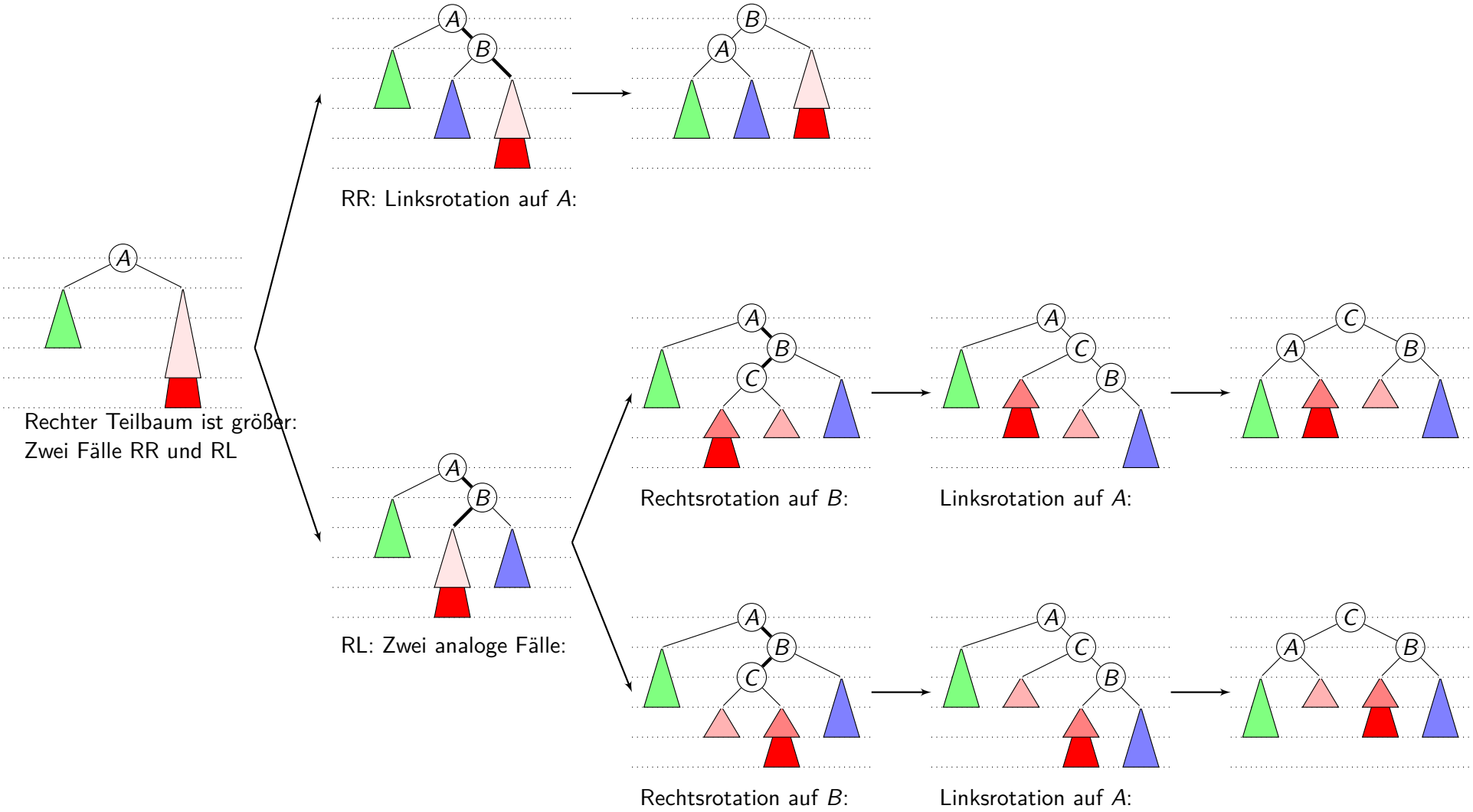
- Ein rotierter BST ist ein BST
- Die Inorder-Traversierung beider Bäume bleibt unverändert.

Zeitkomplexität

Zeitkomplexität von Links- oder Rechtsrotieren ist in $\Theta(1)$.

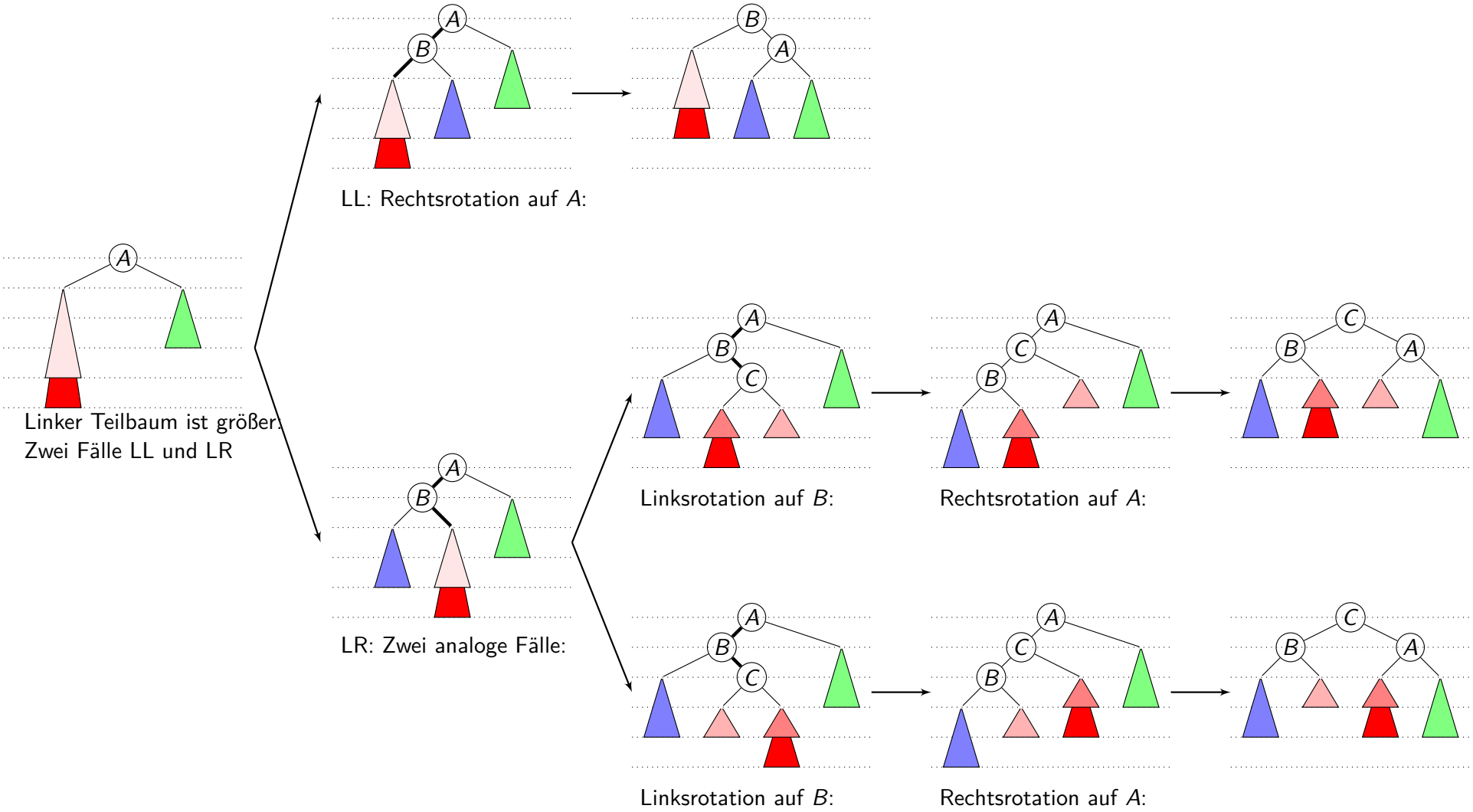
AVL-Bäume: Balancieren nach Einfügen

Sei A der tiefste unbalancierte Knoten auf dem Pfad von der Wurzel zum neu eingefügten Knoten (unbalanciert: *linke Teilbaumhöhe* − *rechte Teilbaumhöhe* = ± 2).



AVL-Bäume: Balancieren nach Einfügen

Sei A der tiefste unbalancierte Knoten auf dem Pfad von der Wurzel zum neu eingefügten Knoten (unbalanciert: *linke Teilbaumhöhe* − *rechte Teilbaumhöhe* $= \pm 2$).



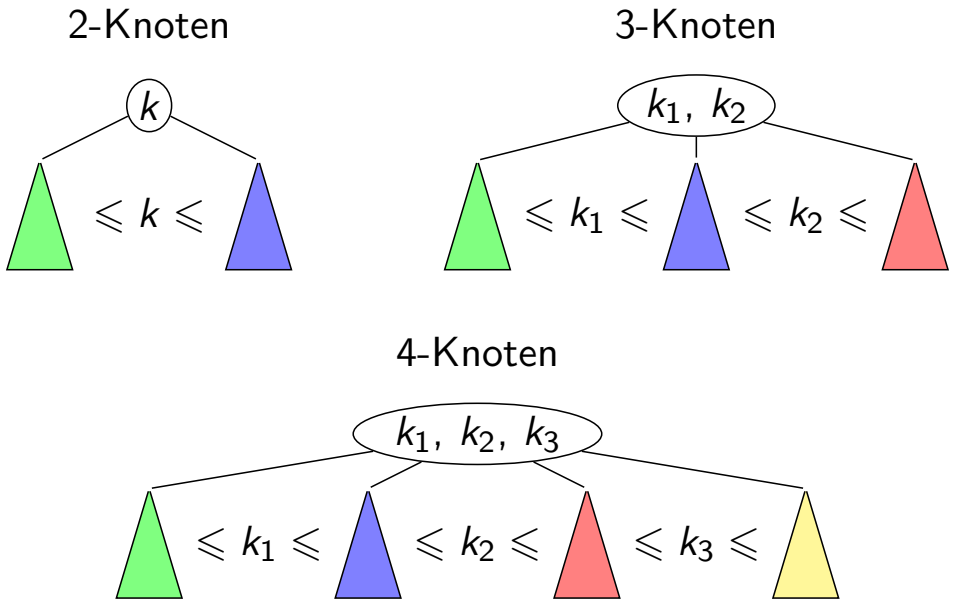
AVL-Bäume: Balancieren nach Löschen

- Einfügen: Baum ist nach einer Einfach- oder Doppelrotation wieder balanciert (da Baumhöhe unter dem balancierten Knoten ist das Gleiche wie vor dem Einfügen).
- Löschen: Nach einmal balancieren kann der Baumhöhe um 1 kleiner sein als vor dem Löschen.

  Im schlimmsten Fall müssen alle unbalancierten Knoten (vom verkleinerten Knoten hoch bis zur Wurzel) einzeln balanciert werden ($\mathcal{O}(\log n)$ Aufwand).

Übersicht

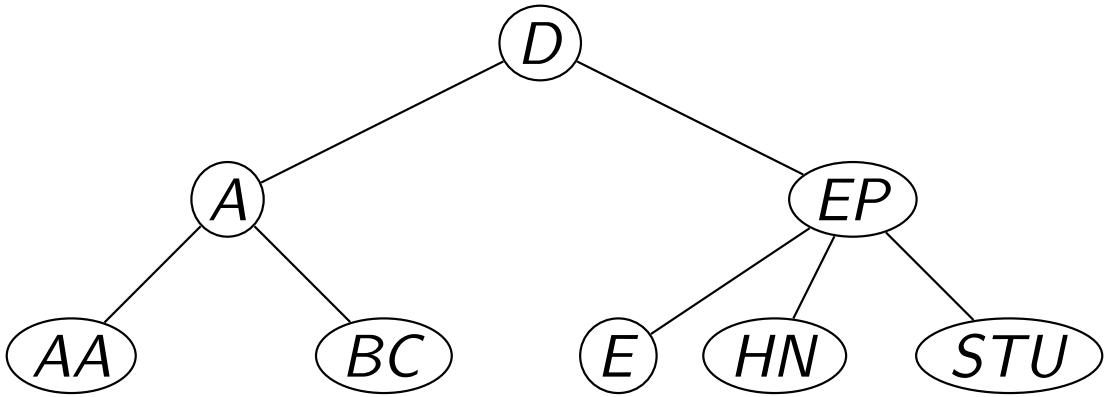
2-3-4-Bäume



- n-Knoten haben $n-1$ Schlüssel.
- Innere n-Knoten haben n Kinder ($\neq$ `null`)
- Blätter n-Knoten haben keine Kinder.
- 2-3-4-Baum: nur 2-, 3- oder 4-Knoten, alle Blätter haben die gleiche Tiefe (vollständiger Baum mit voller letzten Ebene).

Spezialfall von B-Bäumen. Anwendung: Speicherverwaltung.

2-3-4-Bäume: Beispiel



2-3-4-Bäume: Implementierung

```
1  class 234Node{
2    int n; // Anzahl der Schlüssel, <=3
3    int[3] key; // n Schlüssel
4    bool isLeaf; // Ist Blatt?
5    Node[4] child; // n+1 Kinder
6    Node(){ n=0; isLeaf=true; } //Konstruktor
7  };

9  class 234Tree{
10   Node root;
11   234Tree(){ root=null; } //Konstruktor
12 };
```

2-3-4-Bäume: Schlüssel suchen

```
1 // Aufruf auf 234Tree t: 234Search(t.root, k)
2 234Node 234Search(234Node node, int k){
3   if (node==null) return null;
4   int i;
5   for (i=0; i<n && node.key[i]<k; i++) {}
6   if (i<n && node.key[i] == k) return node;
7   else if (node.isLeaf) return null;
8   else return 234Search(node.child[i],k);
9 }
```

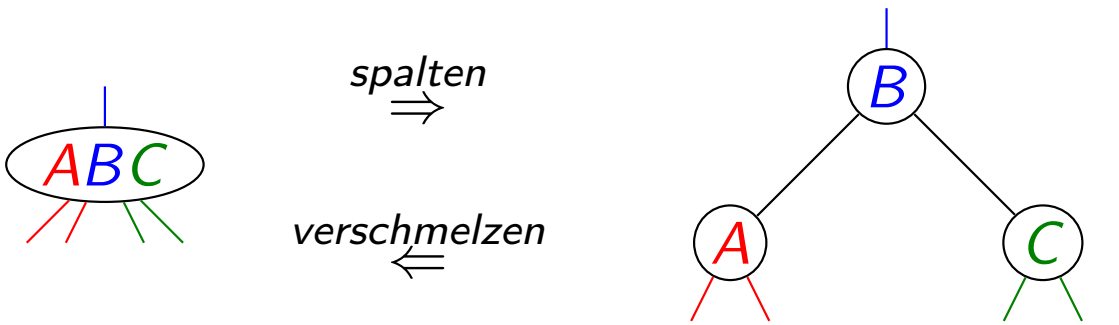
2-3-4-Bäume: Nachfolger/Vorgänger suchen

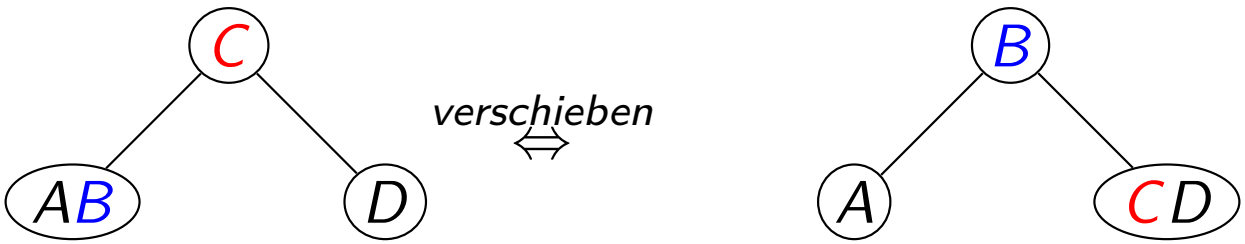
```
1  int 234Succ(234Node node, int i){
2  // Suche den Nachfolgerschlüssel von node.key[i] im Baum von
3  // dem inneren Knoten node!=null
4    node = node.child[i+1];
5    while (!node.isLeaf) node = node.child[0];
6    return node.key[0];
7  }

9  int 234Pred(234Node node, int i){
10 // Suche den Vorgängerschlüssel von node.key[i] im Baum von
11 // dem inneren Knoten node!=null
12   node = node.child[i];
13   while (!node.isLeaf) node = node.child[node.n];
14   return node.key[node.n-1];
15 }
```

2-3-4-Bäume: Operationen

- Darstellung einer Multimenge durch 2-3-4-Bäume ist nicht eindeutig.
- Wir können 4-Knoten spalten.
- Wir können 2-Knoten verschmelzen.
- Wir können Schlüssel verschieben (rotieren).





2-3-4-Bäume: 4-Knoten spalten

```
1  void 234Split(234Node p, 234Node node, int i){
2  // node!=null voller Knoten (node.n==3)
3  // node ist i-tes Kind von nicht vollem p!=null
4    234Node b; // neuer rechter Bruder von node
5    b.isLeaf=node.isLeaf; b.n=1; b.key[0]=node.key[2];
6    if (!node.isLeaf){
7      b.child[0]=node.child[2]; b.child[1]=node.child[3];
8    }
9    node.n=1; // Skaliere node, er hat nur noch einen Schlüssel
10   for (int j=p.n; j>=i+1; j--) p.child[j+1] = p.child[j];
11   p.child[i+1] = b; // Bruder wird neues Kind vom Vater
12   for (int j=p.n-1; j>=i; j--) p.key[j+1] = p.key[j];
13   p.key[i] = node.key[1]; // mittlerer Schlüssel von node geht
         zum Vater
14   p.n++;
15 }
```

2-3-4-Bäume: Knoten verschmelzen

```
1  void 234Merge(234Node node, int i){
2  // Vorbed.: node!=null innerer Knoten, 0<=i<node.n,
3  // node.child[i].n==node.child[i+1].n==1.
4  // Verschmelzt node's i-ten Schlüssel und sein rechtes und
5  // linkes Kind in dem linken Kind
6    234Node left = node.child[i];
7    234Node right = node.child[i+1];
8    left.n=3; //verschmelze im linken Kind
9    left.key[1] = node.key[i]; //die Schlüssel
10   left.key[2] = right.key[0];
11   if (!left.isLeaf){// und die Kinder
12     left.child[2]=right.child[0];
13     left.child[3]=right.child[1];
14   }
15   for (; i<node.n-1; i++){//verkleinere node (da 1 Schlüssel
         und 1 Kind weniger)
16     node.key[i] = node.key[i+1];
17     node.child[i+1] = node.child[i+2];
18   }
19   node.n--;
20 }
```

2-3-4-Bäume: Schlüssel verschieben

```
1  void 234ShiftFromRightToLeft(234Node node, int i){
2  // Vorbed.: node!=null innerer Knoten, 0<=i<node.n,
3  // node.child[i].n==1, node.child[i+1].n>1
4    234Node left = node.child[i];
5    234Node right = node.child[i+1];
6    left.key[1] = node.key[i];
7    node.key[i]=right.key[0];
8    for (int j=0; j<right.n-1; j++)
9      right.key[j] = right.key[j+1];
10   left.child[1] = right.child[0];
11   for (int j=0; j<right.n; j++)
12     right.child[j] = right.child[j+1];
13   left.n++;
14   right.n--;
15 }
```

2-3-4-Bäume: Schlüssel verschieben

```
1 void 234ShiftFromLeftToRight(234Node node, int i){
2 // Vorbed.: node!=null innerer Knoten, 0<=i<node.n,
3 // node.child[i].n>1, node.child[i+1].n==1
4   234Node left = node.child[i];
5   234Node right = node.child[i+1];
6   right.key[1] = right.key[0];
7   right.key[0] = node.key[i];
8   node.key[i]=left.key[left.n-1];
9   right.child[1] = right.child[0];
10  right.child[0] = left.child[left.n];
11  left.n--;
12  right.n++;
13 }
```

2-3-4-Bäume: Einfügen eines Schlüssels

- Steige von der Wurzel zum Blatt, in dem eingefügt werden soll, hinunter (ähnlich wie im Binärbaum).
- Spalte auf dem Weg alle maximalen Knoten auf (benötigt nicht-maximalen Vaterknoten, bei der Wurzel wird der Baum nach oben erhöht).
- Füge den Schlüssel im Blatt ein.

2-3-4-Bäume: Knoten einfügen

```
1  void 234Insert(234Tree t, int k){
2    234Node node = t.root;
3    if (node==null){ // wenn Baum leer
4      node = new 234Node(); node.n=1; node.isLeaf=true;
5      node.key[0] = k; t.root=node;
6    } else {
7      if (node.n==3){ // wenn Wurzel voll
8        p = new 234Node(); p.n=0; p.isLeaf=false;
9        p.child[0]=node; t.root=p;
10       234Split(p, node, 0);
11       node = p;
12     }
13     234InsertNonfull(node, k);
14   }
15 }
```

2-3-4-Bäume: Knoten einfügen

```
1  void 234InsertNonfull(234Node node, int k){
2  // node!=null, node ist nicht voll
3    int i;
4    if (node.isLeaf){
5      for (i = node.n-1; i>=0 && k<node.key[i]; i--)
6        node.key[i+1] = node.key[i];
7      node.key[i+1] = k;
8      node.n++;
9    } else {
10     for (i = node.n-1; i>=0 && k<node.key[i]; i--) {};
11     i = i+1;
12     if (node.child[i].n == 3) {
13       234Split(node, node.child[i], i);
14       if (k > node.key[i]) i = i+1;
15     }
16     234InsertNonfull(node.child[i],k);
17   }
18 }
```

2-3-4-Bäume: Löschen eines Schlüssels

1. Steige von der Wurzel zum Knoten (nicht unbedingt Blatt), in dem der Schlüssel k gelöscht werden soll, hinunter (ähnlich zum Einfügen).
2. Vergrößere auf dem Weg alle minimalen Knoten durch
   - 2.1 Verschiebung von Schlüsseln oder
   - 2.2 Verschmelzung von Knoten.
3. Fallunterscheidung Zielknoten:
   - 3.1 Blatt: lösche den Schlüssel k (beachte: Blatt ist nicht minimal!).
   - 3.2 Innerer Knoten:
     - 3.2.1 Wenn ein Kind, das entweder den Nachfolger- oder den Vorgängerschlüssel k' von k enthält, nicht minimal ist, dann lösche k' (rekursiv) und ersetze k durch k'.
     - 3.2.2 Sonst verschmelze die Kinder, die den Vorgänger- und Nachfolgerschlüsseln enthalten, mit k und lösche k rekursiv.

2-3-4-Bäume: Knoten löschen

```
1  //Löscht den Schlüssel k im Baum von node
2  //Vorbedingung: k kommt im Baum vor und node.n>1
3  //Aufruf auf Baum t: 234Delete(t.root,k)
4  //(wenn t.root.n==1, setze statt dessen t.root=null)
5  void 234Delete(234Node node, int k){
6    int i; for (i=0; i<n && node.key[i]<k; i++) {}
7    if (i<n && node.key[i] == k) {//Schlüssel ist in node
8      if (node.isLeaf()){ // Schlüssel ist in einem Blatt
9        for ( ; i<n-1; i++) node.key[i] = node.key[i+1];
10       node.n--; //Schlüssel geloescht
11     } else if (node.child[i].n>1){
12       int r = 234Pred(node,i);
13       234Delete(node.child[i],r); node.key[i] = r;
14     } else if (node.child[i+1].n>1){
15       int r = 234Succ(node,i);
16       234Delete(node.child[i+1],r); node.key[i] = r;
17     } else {
18       234Merge(node,i); return 234Delete(node.child[i],k);
19     }
20   } else {/*Schluessel ist nicht in node: naechste Folie*/}
21 }
```

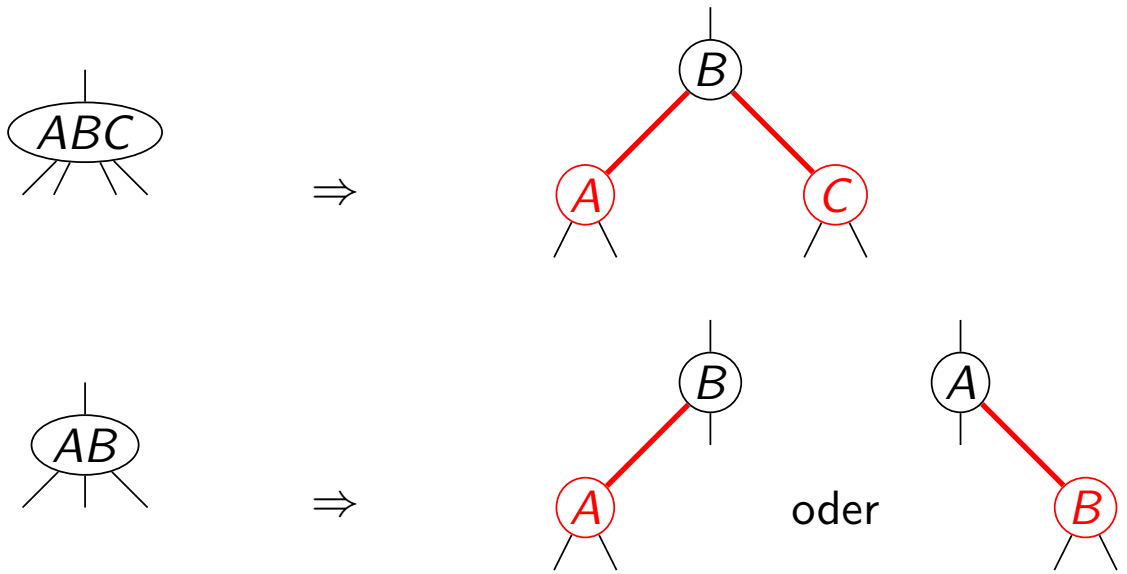
2-3-4-Bäume: Knoten löschen

```
1  void 234Delete(234Node node, int k){
2    int i; for (i=0; i<n && node.key[i]<k; i++) {}
3    if (i<n && node.key[i] == k) {
4      //Schlüssel ist in node: vorige Folie
5    } else {//Schlüssel ist nicht in node
6      if (node.child[i].n==1){
7        if (i<n && node.child[i+1].n>1){
8          234ShiftFromRightToLeft(node,i);
9        } else if (i>0 && node.child[i-1].n>1){
10         234ShiftFromLeftToRight(node,i-1);
11       } else {
12         if (i<n) 234Merge(node,i);
13         else {i = i-1; 234Merge(node,i);}
14       }
15     }
16     return 234Delete(node.child[i],k);
17   }
18 }
```

Übersicht

Motivation

Jeder 2-3-4-Baum kann durch einen (Rot-Schwarz-Baum, red-black-tree, RBT) repräsentiert werden.



- Anzahl schwarzer Knoten auf einem Pfad im RBT entspricht der Anzahl der Knoten auf dem Originalpfad.
- Vollständigkeit des 2-3-4-Baumes impliziert farben-balanciertheit für den RBT: Die Anzahl schwarzer Knoten ist auf allen Pfaden von der Wurzel zu einem Blatt gleich.

Rot-Schwarz-Bäume

Rot-Schwarz-Eigenschaft

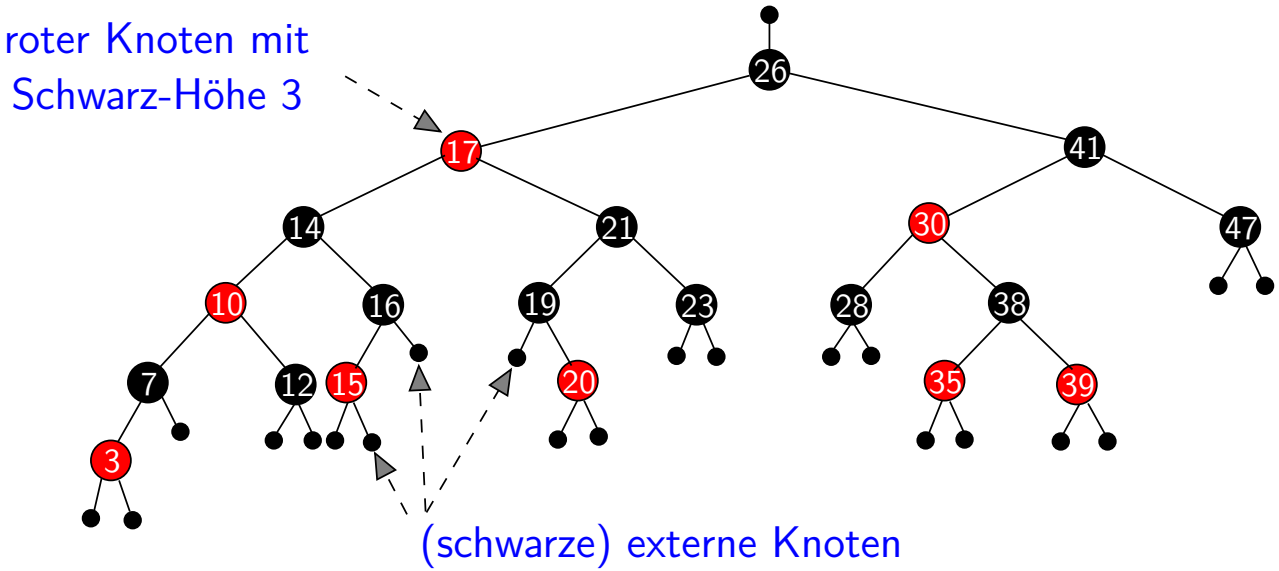
Ein binärer Suchbaum, dessen Knoten jeweils zusätzlich eine Farbe haben, hat die Rot-Schwarz-Eigenschaft, wenn:

1. Jeder Knoten ist entweder rot oder schwarz.
2. Die Wurzel ist schwarz.
3. Ein roter Knoten hat nur schwarze Kinder.
4. `null`-Zeiger (fehlendes Kind, hier enden die Pfade) betrachten wir als externe Knoten mit der Farbe schwarz.
5. Für jeden Knoten enthalten alle Pfade, die an diesem Knoten starten und in einem externen Knoten enden, die gleiche Anzahl schwarzer Knoten.

Solche Bäume heißen dann Rot-Schwarz-Bäume (red-black tree, RBT).

- In den Algorithmen verwenden wir für `null`-Zeiger (externe Knoten) die Notation `null.color == BLACK`.

Rot-Schwarz-Bäume



Definition

- Die Schwarz-Höhe $bh(x)$ eines Knotens x ist die Anzahl schwarzer Knoten bis zu einem (externen) Blatt, x ausgenommen.
- Die Schwarzhöhe $bh(t)$ eines RBT t ist die Schwarz-Höhe seiner Wurzel.

- Die Schwarz-Höhe eines externen Blattes $bh(\texttt{null}) = 0$.
- Die externen Knoten werden in Zeichnungen oft weggelassen.

Elementare Eigenschaften von Rot-Schwarz-Bäumen

Lemma

Ein Rot-Schwarz-Baum t mit Schwarzhöhe $h = bh(t)$ hat:

- *Mindestens $2^h - 1$ innere Knoten.*
- *Höchstens $4^h - 1$ innere Knoten.*

Theorem

Ein RBT mit n inneren Knoten hat höchstens die Höhe $2 \cdot \log(n+1)$.

$\Rightarrow$ Suchen benötigt also nur $\Theta(\log n)$ statt $\Theta(n)$ Zeit.

- Für `bstMin`, `bstSucc`, etc. gilt dasselbe.

2-3-4-Bäume: Einfügen eines Schlüssels

- Steige von der Wurzel zum Blatt, in dem eingefügt werden soll, hinunter (ähnlich wie im Binärbaum).
- Spalte auf dem Weg alle maximalen Knoten auf (beim Spalten einer maximalen Wurzel wird der Baum nach oben erhöht).
- Füge den Schlüssel im Blatt ein.

2-3-4-Bäume: Einfügen eines Schlüssels

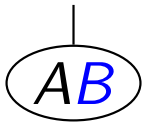
In RBT wird genauso verfahren, aber die 4-Knoten werden nicht auf dem Weg nach unten gespalten, sondern nach dem Einfügen rekursiv nach oben.

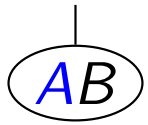
- Steige von der Wurzel zum Blatt, in dem eingefügt werden soll, hinunter.
- Füge den Schlüssel im Blatt ein.
- Spalte rückwärts eventuelle "5-Knoten" auf.

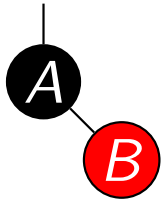
Wir brauchen für die RBT Implementierung...

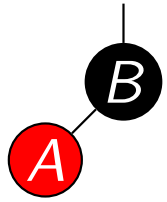
- Einfügen eines Schlüssels in ein Blatt
- Split-Operation für 4-Knoten

Einfügen in Blatt mit 1 Schlüssel

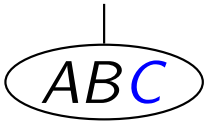


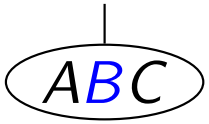


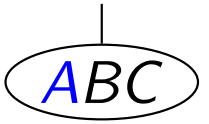


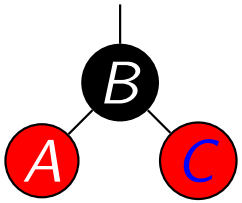


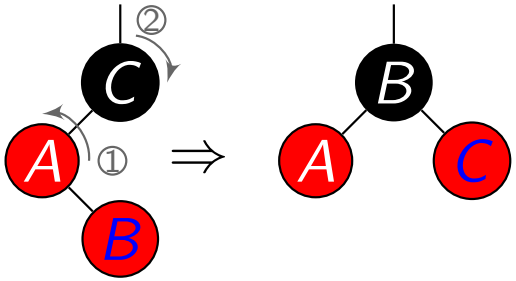
Einfügen in Blatt mit 2 Schlüsseln



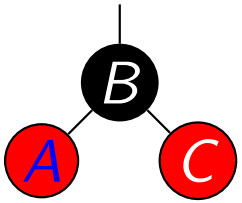


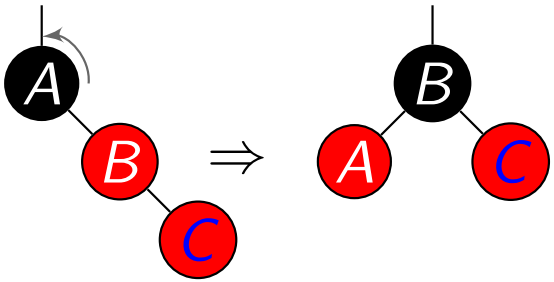




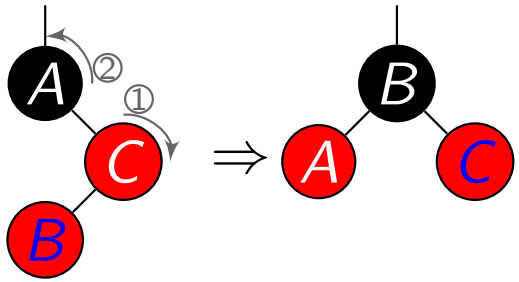


LR-Rotation + umfärben

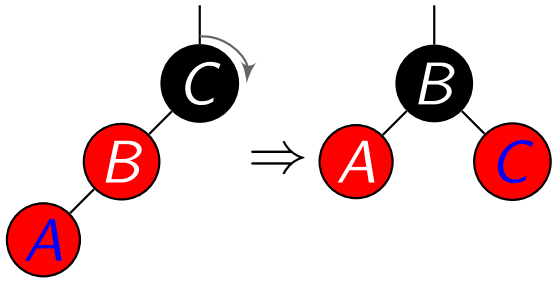




RR-Rotation + umfärben

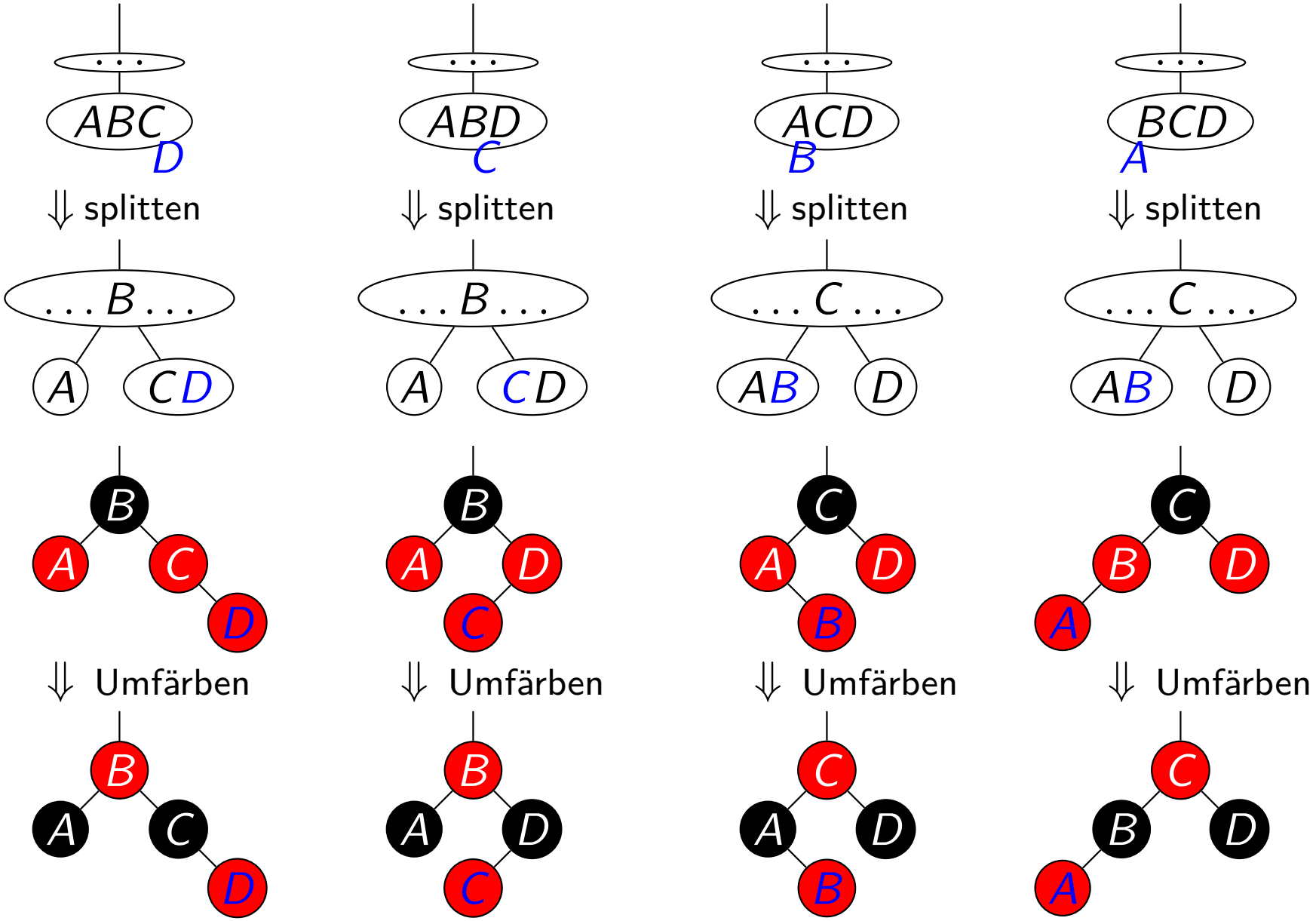


RL-Rotation + umfärben



LL-Rotation + umfärben

Einfügen in Blatt mit 3 Schlüsseln



Einfügen in einen RBT – Algorithmus

```
1 void rbtIns(Tree t, Node node) { // Füge node in den Baum t ein
2   bstIns(t, node); // Einfügen wie beim BST
3   node.left = null;
4   node.right = null;
5   node.color = RED; // eingefügter Knoten immer zunächst rot
6   // stelle Rot-Schwarz-Eigenschaft ggf. wieder her
7   rbtInsFix(t, node);
8 }
```

Einfügen in einen RBT – Algorithmus Teil 2

```
1  // Behebe eventuelle Rot-Rot-Verletzung mit Vater, node ist rot
2  void rbtInsFix(Tree t, Node node) {
3    // solange noch eine Rot-Rot-Verletzung besteht
4    // NB: root.parent.color==black
5    while (node.parent.color == RED) {
6      if (node.parent == node.parent.parent.left) {
7        // der von uns betrachtete Fall
8        node = leftAdjust(t, node); // node jetzt weiter oben?
9        // (node = node.parent.parent im Fall 1 von leftAdjust)
10     } else {
11       // der dazu symmetrischer Fall
12       node = rightAdjust(t, node);
13     }
14   }
15   t.root.color = BLACK; // Wurzel bleibt schwarz
16 }
```

Einfügen in einen RBT – Algorithmus Teil 3

```
1  Node leftAdjust(Tree t, Node node) {
2    Node uncle = node.parent.parent.right;
3    if (uncle.color == RED) { // Fall 1: Split
4      node.parent.parent.color = RED; // Großvater
5      node.parent.color = BLACK; // Vater
6      uncle.color = BLACK; // Onkel
7      return node.parent.parent; // prüfe Rot-Rot weiter oben
8    } else { // Fall 2 und 3
9      if (node == node.parent.right) { // Fall 2
10       // dieser Knoten wird das linke, rote Kind:
11       node = node.parent;
12       leftRotate(t, node);
13     } // Fall 3
14     rightRotate(t, node.parent.parent);
15     node.parent.color = BLACK;
16     node.parent.right.color = RED;
17     return node; // fertig, node.parent.color == BLACK
18   }
19 }
```

Einfügen in einen RBT – Analyse

Zeitkomplexität Einfügen

Die Worst-Case Laufzeit von `rbtIns` für ein RBT mit n inneren Knoten ist $O(\log n)$.

Erinnerung: Löschen im 2-3-4-Baum

1. Steige von der Wurzel zum Knoten (nicht unbedingt Blatt), in dem der Schlüssel k gelöscht werden soll, hinunter (ähnlich zum Einfügen).
2. Vergrößere auf dem Weg alle minimalen Knoten durch
   - 2.1 Verschiebung von Schlüsseln oder
   - 2.2 Verschmelzung von Knoten.
3. Fallunterscheidung Zielknoten:
   - 3.1 Blatt: lösche den Schlüssel k (beachte: Blatt ist nicht minimal!).
   - 3.2 Innerer Knoten:
     - 3.2.1 Wenn ein Kind, das entweder den Nachfolger- oder den Vorgängerschlüssel k' von k enthält, nicht minimal ist, dann lösche k' (rekursiv) und ersetze k durch k'.
     - 3.2.2 Sonst verschmelze die Kinder, die den Vorgänger- und Nachfolgerschlüsseln enthalten, mit k und lösche k rekursiv.

Wie das Splitten während des Einfügens, für RBTs machen wir auch das Vergrößern nicht auf dem Weg nach unten, sondern nach Erreichen des zu löschenden Schlüssels nach oben.

Löschen im RBT – Algorithmus Teil 1

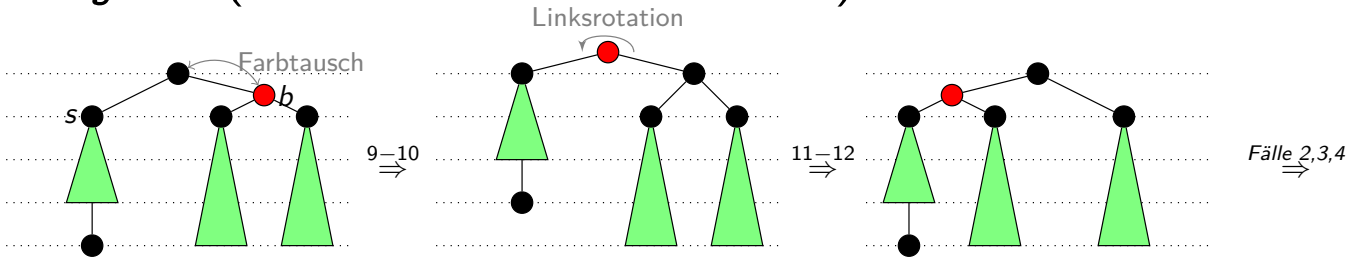
```
1  // Entfernt node aus dem Baum.
2  void rbtDel(Tree t, Node node) {
3    if (node.left && node.right) {
4      // zwei Kinder: ersetze Knoten durch den Nachfolger
5      Node tmp = bstMin(node.right); // finde den Nachfolger
6      rbtDel(t, tmp); // lösche den Nachfolger
7      bstSwap(t, node, tmp); // ersetze node durch Nachfolger
8      tmp.color = node.color; // übernimm die Farbe
9    } else { // zu löschender Schlüssel ist in 2- oder 3-Blatt
10     Node child;
11     if (node.left) child = node.left; // Kind ist links
12     else if (node.right) child = node.right; // Kind ist rechts
13     else child = null; // kein Kind
14     rbtDelFix(t, node, child);
15     bstReplace(t, node, child);
16   }
17 }
```

Löschen im RBT – Algorithmus Teil 2

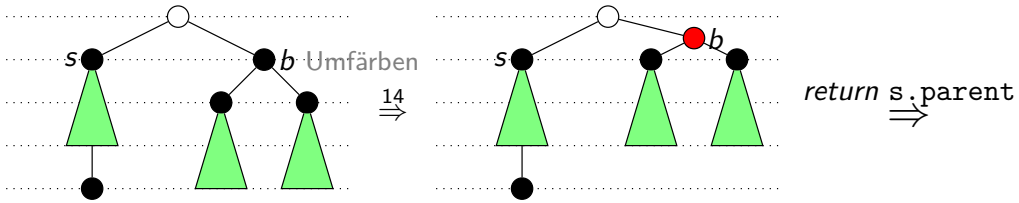
```
1  // node soll gelöscht werden, child ist das einzige Kind
2  // (bzw. node hat keine Kinder, dann ist child == node);
3  // ist node rot, so ist nichts zu tun; sonst suchen wir
4  // einen roten Knoten, der durch Umfärben auf schwarz
5  // die schwarze Farbe von node übernimmt
6  void rbtDelFix(Tree t, Node node, Node child) {
7    if (node.color == RED) return;
8    if (child.color == RED) {
9      child.color = BLACK;
10   } else {
11     Node searchPos = node;
12     // solange der Schwarzwert nicht eingefügt werden kann
13     while (searchPos.parent && searchPos.color == BLACK) {
14       if (searchPos == searchPos.parent.left) //linkes Kind
15         searchPos = delLeftAdjust(t, searchPos);
16       else // rechtes Kind
17         searchPos = delRightAdjust(t, searchPos);
18     }
19     searchPos.color=BLACK;
20   }
21 }
```

Idee von `delLeftAdjust` (s: `searchPos`, b: `bruder`)

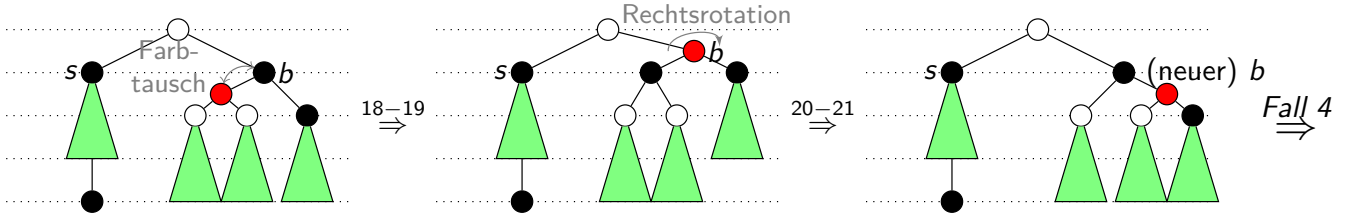
Fall 1: roter Bruder



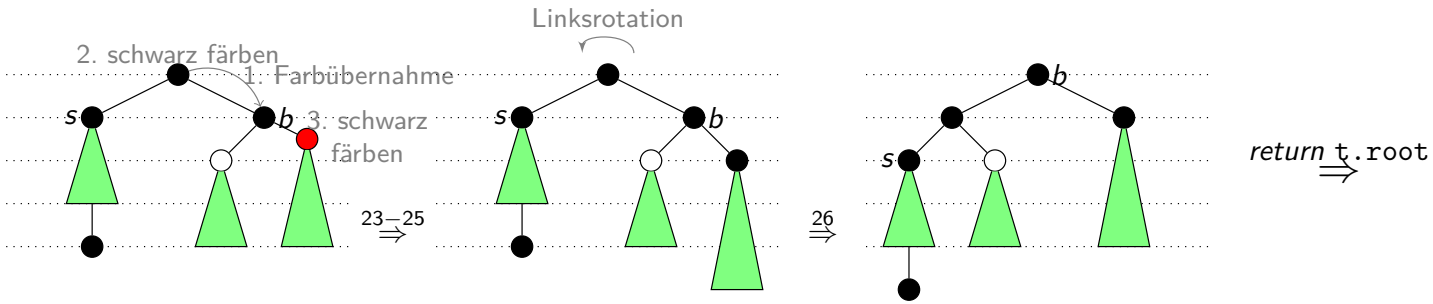
Fall 2: schwarzer Bruder mit schwarzen Kindern



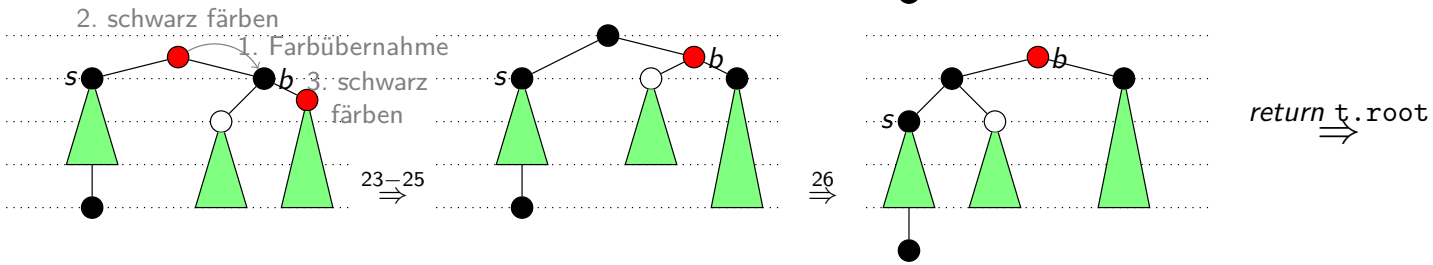
Fall 3: schwarzer Bruder mit schwarzem rechten und rotem linken Kind



Fall 4: schwarzer Bruder mit rotem rechten Kind ...und schwarzem Vater



...und rotem Vater



Löschen im RBT – Analyse

Zeitkomplexität Löschen

Die Worst-Case Laufzeit von `rbtDel` für ein RBT mit n inneren Knoten ist $O(\log n)$.

Komplexität der RBT-Operationen

Operation	Zeit
`bstSearch`	$\Theta(h)$
`bstSucc`	$\Theta(h)$
`bstMin`	$\Theta(h)$
`bstIns`	$\Theta(h)$
`bstDel`	$\Theta(h)$

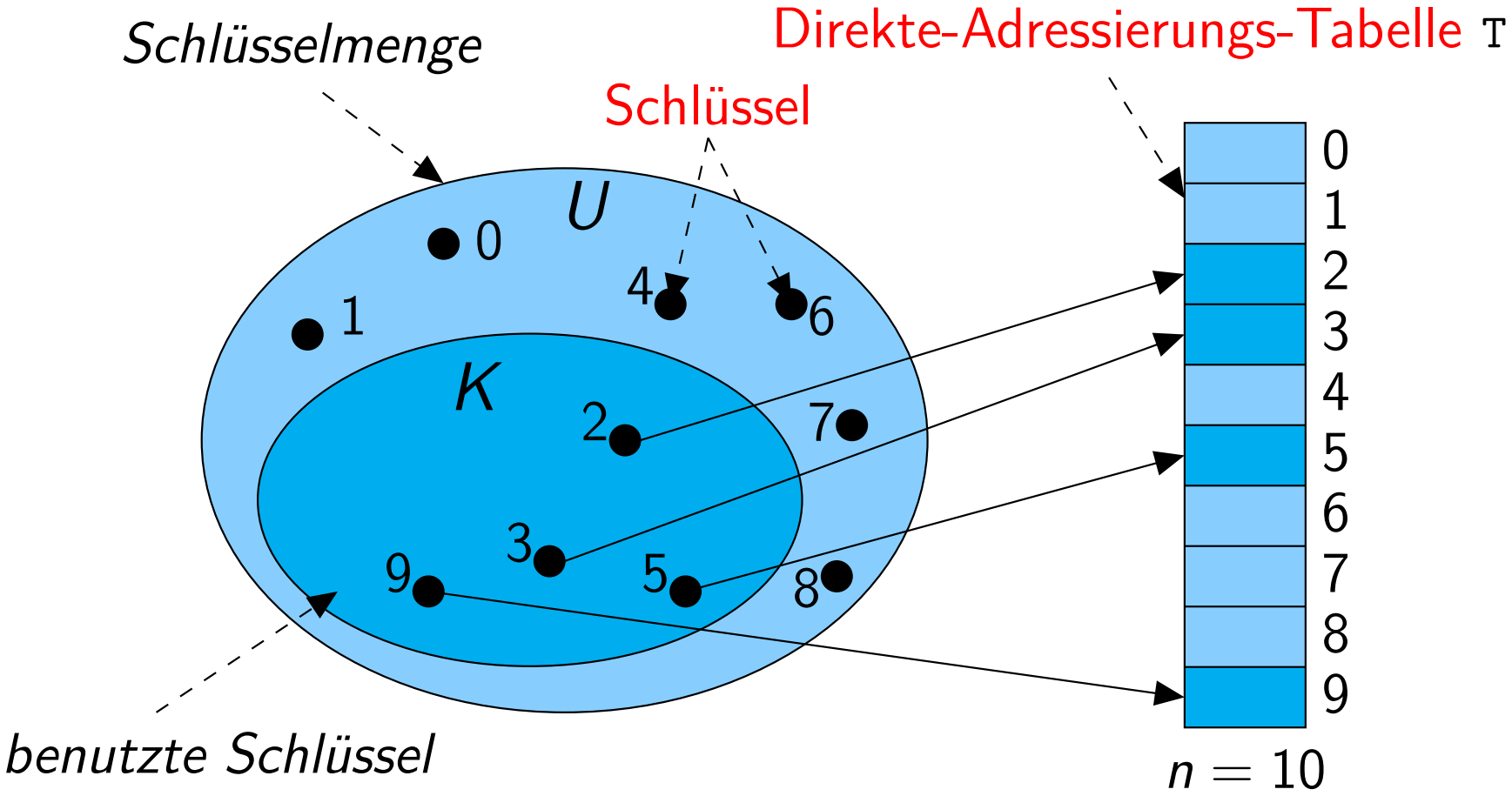
Operation	Zeit
`rbtIns`	$\Theta(\log n)$
`rbtDel`	$\Theta(\log n)$

- Alle anderen Operationen wie beim BST, wobei $h = \log n$.

Alle Operationen sind logarithmisch in der Größe des Rot-Schwarz-Baumes

Übersicht

Direkte Adressierung



Hashing

Hashfunktion, Hashtabelle, Hashkollision

Eine Hashfunktion bildet Schlüssel auf Indices der Hashtabelle `T` ab:

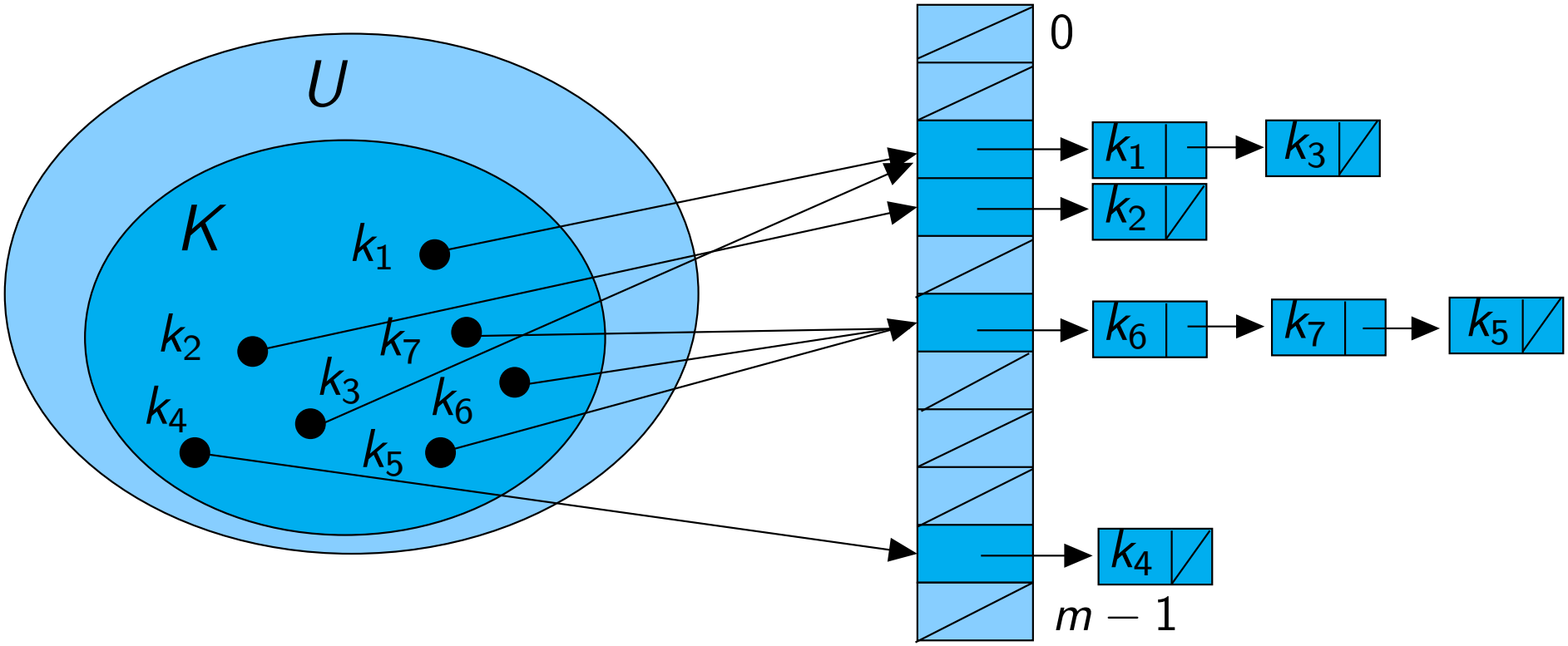
$$h : U \longrightarrow \{0, 1, \ldots, m-1\} \text{ für Tabellengröße } m.$$

Wir sagen, dass $h(k)$ der Hashwert des Schlüssels k ist.
Das Auftreten von $h(k) = h(k')$ für $k \neq k'$ nennt man eine Kollision.

Kollisionsauflösung durch Listen

Idee

Alle Schlüssel, die zum gleichen Hash führen, werden in einer verketteten Liste gespeichert. [Luhn 1953]



Hashfunktionen

Divisionsmethode

Hashfunktion: $h(k) = k \textbf{ mod } m$

Multiplikationsmethode

Hashfunktion: $h(k) = \lfloor m \cdot (k \cdot c \textbf{ mod } 1) \rfloor$ für $0 < c < 1$

- $k \cdot c$ **mod** 1 ist der Nachkommateil von $k \cdot c$, d. h. $k \cdot c - \lfloor k \cdot c \rfloor$.

Universelles Hashing

Wähle zufällig eine Hashfunktion aus einer gegebenen kleinen universellen Menge H, unabhängig von den verwendeten Schlüsseln.

$$h_{a,b}(k) = ((a \cdot k + b) \textbf{ mod } p) \textbf{ mod } m$$

- p sei Primzahl mit $p > m$ und $p >$ größter Schlüssel.

Offene Adressierung

- Alle Elemente werden direkt in der Hashtabelle gespeichert (im Gegensatz zur Verkettung).

$\Rightarrow$ Höchstens m Schlüssel können gespeichert werden, d. h. $\alpha(n, m) = \frac{n}{m} \leqslant 1$.

- Man spart aber den Platz für die Pointer.

Einfügen von Schlüssel k

- Sondiere (überprüfe) die Positionen der Hashtabelle in einer bestimmten Reihenfolge, bis ein leerer Slot gefunden wurde.
- Die Reihenfolge der Positionen sind vom einzufügenden Schlüssel k abgeleitet.
- Die Hashfunktion hängt also vom Schlüssel k und der Nummer der Sondierung ab:

$$h : U \times \{0, 1, \ldots m-1\} \longrightarrow \{0, 1, \ldots m-1\}$$

Wie wählt man die nächste Sondierung?

Wir benötigen eine Sondierungssequenz für einen gegebenen Schlüssel k:

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$$

Sondierungsverfahren

- Wir behandeln Lineares Sondieren, Quadratisches Sondieren und Doppeltes Hashing.

Hashfunktion beim linearen Sondieren

$h(k,i) = (h'(k) + i)$ **mod** m (für $i < m$).

- h' ist eine übliche Hashfunktion.

Hashfunktion beim quadratischen Sondieren

$h(k,i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2)$ **mod** m (für $i < m$).

- h' ist eine übliche Hashfunktion, und
- $c_1 \in \mathbb{N}$, $c_2 \in \mathbb{N} \setminus \{0\}$ geeignete Konstanten.

Hashfunktion beim doppelten Hashing

$h(k,i) = (h_1(k) + i \cdot h_2(k))$ **mod** m (für $i < m$).

- h_1, h_2 sind übliche Hashfunktionen.