Algorithmen und Datenstrukturen

Suchbäume

Prof. Justus Piater, Ph.D.

31. Mai 2025

Dieser Kurs folgt in weiten Teilen dem sehr empfehlenswerten Lehrbuch *Data Structures and Algorithms in Java* [Goodrich u. a. 2014].

Inhaltsverzeichnis

Einführung

Wir haben bereits den abstrakten Datentyp Zuordnungstabelle kennengelernt. Die Zuordnungstabelle formalisiert den Zugriff auf Datenelemente v über Schlüssel k mittels der Methoden `get()`, `put()` und `remove()`. Dies ist jedoch alles. Insbesondere ist über den Schlüsseln *keine Reihenfolge* definiert.

Oft sind wir jedoch nicht nur am schnellen Zugriff auf einzelne Einträge interessiert, sondern auch an sortierten Listen von Einträgen, wie z.B. in der Kontaktliste auf Ihrem Smartphone.

Hierzu dient der abstrakte Datentyp ***Sortierte Zuordnungstabelle***. Diese arbeitet mit Schlüsseln, über denen eine Totalordnung $\leq$ definiert ist, und erweitert die Zuordnungstabelle um Methoden, die Zugriff auf die Schlüssel gemäß ihrer Reihenfolge erlauben.

Die beliebtesten Datenstrukturen für sortierte Zuordnungstabellen sind sogenannte ***Suchbäume***. Die Kernidee von Suchbäumen ist, dass Suchoperationen linear in der Höhe des Baumes gehalten werden. Damit das effizient ist, müssen wir garantieren, dass die Höhe des Baumes logarithmisch in der Anzahl der gespeicherten Elemente bleibt, über Einfüge- und Entfernungsoperationen hinweg. Verschiedene Suchbaum-Algorithmen unterscheiden sich in der Art und Weise, wie sie dies erreichen.

In diesem Kapitel schauen wir uns drei wichtige Typen von Suchbäumen an, nämlich AVL-Bäume, B-Bäume und Rot-Schwarz-Bäume. AVL-Bäume sind einfach zu verstehen und zu implementieren. B-Bäume werden insbesondere in Datenbanken und Dateisystemen

verwendet. Ein Spezialfall von B-Bäumen sind die sogenannten (2,4)-Bäume. Diese sind mathematisch äquivalent zu Rot-Schwarz-Bäumen. Rot-Schwarz-Bäume gehören zu den effizientesten generischen Suchbäumen und sind von großer praktischer Bedeutung. Unter anderem verwendet sie der Linux-Kernel für zahlreiche interne Datenstrukturen.

Video 1 beginnt hier.

1 Sortierte Zuordnungstabelle

ADT: Sortierte Zuordnungstabelle [Slide 1]

Wichtigste Methoden: [Goodrich u. a. 2014; Foto von Maksym Kaharlytskyi auf Unsplash]

```
// Map:
get(k)          // Returns the value v associated with key k,
                // if it exists; otherwise returns null.
put(k, v)       // If M does not have an entry with key k, then
                // adds entry (k, v) to M and returns null.
                // Otherwise, replaces with v the value of the
                // existing entry, and returns the old value.
remove(k)       // Removes from M the existing entry with key k,
                // and returns its value; otherwise returns null.
// Sorted Map:
lowerEntry(k)   // Returns the entry with the greatest key < k (or null)
higherEntry(k)  // Returns the entry with the least key > k (or null)
subMap(k1, k2)  // Returns an iterable collection of all entries with k1 ≤ key < k2
```

Definition

Eine ***Sortierte Zuordnungstabelle*** ist eine Zuordnungstabelle, über deren Schlüsseln eine Totalordnung $\leq$ definiert ist. Sie eignet sich für Anwendungen, bei denen man nicht nur effizient auf Werte über ihre Schlüssel zugreifen möchte, sondern bei denen auch die Reihenfolge der Schlüssel eine Rolle spielt.

Der abstrakte Datentyp Sortierte Zuordnungstabelle übernimmt die Methoden `get()`, `put()` und `remove()` von der Zuordnungstabelle, und ergänzt diese um Methoden, die die Totalordnung über den Schlüsseln nutzen. Die Methode `lowerEntry(k)` liefert den Vorgänger von k, und `higherEntry()` liefert den Nachfolger von k. Die Methode `subMap()` liefert einen iterierbaren Container mit den Schlüsseln im angegebenen Intervall. Sein Iterator liefert die Schlüssel in ihrer definierten Reihenfolge gemäß der Totalordnung.

Genau wie die einfache Zuordnungstabelle darf eine sortierte Zuordnungstabelle einen gegebenen Schlüssel maximal einmal enthalten.

2 Binäre Suchbäume

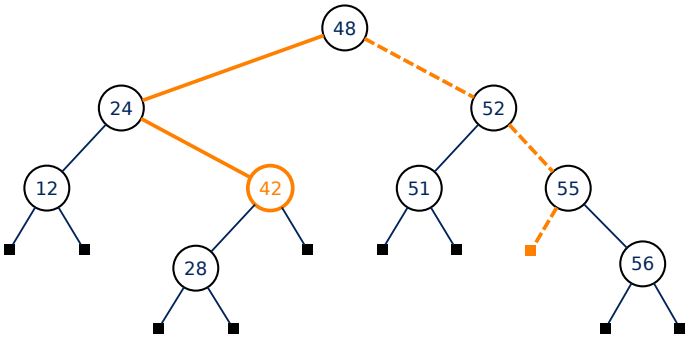
Binärer Suchbaum [Slide 2]

Definition: Sei S eine Menge mit einer Ordnungsrelation. Ein ***binärer Suchbaum*** ist ein voller Binärbaum, für dessen sämtliche internen Knoten p gilt:

- p speichert ein Element $e(p) \in S$.
- Alle Elemente im linken Unterbaum von p sind kleiner als $e(p)$.
- Alle Elemente im rechten Unterbaum von p sind größer als $e(p)$.

Anmerkung

Eine Inorder-Traversierung besucht die Knoten in aufsteigender Reihenfolge.



Anmerkung

Andere Autoren verwenden keine leeren Blätter. Wir verwenden sie hier für alle Suchbäume, da sie bei Rot-Schwarz-Bäumen eine entscheidende Rolle spielen werden und sich damit die Darstellung vereinheitlicht. Außerdem vereinfacht sich die Beschreibung mancher Algorithmen, genau wie bei den Sentinel-Knoten der doppelt verketteten Liste.

Erklärung

Die am weitesten verbreiteten Datenstrukturen für sortierte Zuordnungstabellen sind binäre Suchbäume. Wir definieren einen binären Suchbaum als vollen Binärbaum, dessen Blätter leer sind und dessen interne Knoten Schlüssel-Wert-Paare enthalten, wobei alle Schlüssel im *linken* Unterbaum eines gegebenen Knotens p *kleiner* sind als der Schlüssel in p, und alle Schlüssel im *rechten* Unterbaum von p *größer* sind als der in p.

Da diese Ordnung für jeden Knoten p gilt, folgt unmittelbar, dass eine Inorder-Traversierung die Schlüssel in aufsteigender Reihenfolge besucht.

Aus dieser Ordnung ergibt sich ein einfacher Suchalgorithmus: Wir vergleichen den gesuchten Schlüssel k mit dem des aktuellen Knotens. Ist k kleiner, suchen wir rekursiv im linken Kind weiter; ist er größer, im rechten Kind, und ist er gleich, dann haben wir ihn gefunden, so wie hier im durchgezogenen roten Beispiel bei der Suche nach dem Schlüssel $k = 42$.

Ist der Schlüssel nicht im Baum vorhanden, dann landen wir in einem Blatt, das kein Element enthält. In diesem gestrichelten Beispiel müssen wir nach dem Schlüssel 53 oder 54 gesucht haben, falls alle Schlüssel ganzzahlig sind.

Suche in einem binären Suchbaum [Slide 3]

```
Algorithm treeSearch(p, k):
  Require: A search key k, and a position p of a binary search tree T.
  Ensure:  Return the position q of the subtree T(p) of T rooted at p,
           such that either q's entry has a key equal to k or
           q is an external position at the place where an entry
           with key k would belong if it existed.
  if T.isExternal(p) then
    return p
  else if k < key(p) then
    return treeSearch(T.left(p), k)
  else if k > key(p) then
    return treeSearch(T.right(p), k)
  else
    return p
```

Zur Laufzeit siehe Kapitel *Baumstrukturen*.

Erklärung

Hier sehen wir unseren Suchalgorithmus formal ausformuliert. Falls der gesuchte Schlüssel k kleiner bzw. größer ist als der Schlüssel des aktuellen Knotens p, sucht der Algorithmus im linken bzw. rechten Kind von p rekursiv weiter. Die Rekursion endet, wenn p entweder ein leeres Blatt ist oder den Schlüssel k enthält.

Dieser `treeSearch()`-Algorithmus wird bei der Implementierung verschiedener Methoden der sortierten Zuordnungstabelle eine zentrale Rolle spielen.

Was ist seine Laufzeit?

Bis auf die rekursiven Aufrufe können wir alle Ausdrücke und Anweisungen in konstanter Laufzeit implementieren. In jedem Aufruf findet maximal ein rekursiver Aufruf statt. Bestimmend für die Laufzeit ist also einzig die Rekursionstiefe, und diese ist offesichtlich begrenzt durch die Höhe des Baums.

Die Laufzeit von `treeSearch()` ist also $O(h)$ für einen Baum der Höhe h.

DS: Suchbaum für ADT Sortierte Zuordnungstabelle [Slide 4]

- `get()`, `put()` und `remove()` rufen `treeSearch()` auf.
- `lowerEntry()` etc. durchlaufen den Baum nach einem `treeSearch()`-Aufruf.
- Da die Laufzeit von `treeSearch()` $O(h)$ ist, sollten wir $h \in O(\log n)$ halten.

Übung: Schreiben Sie einen Algorithmus für `higherEntry()`.

Erklärung

Alle nichttrivialen Methoden einer Suchbaum-basierten sortierten Zuordnungstabelle rufen `treeSearch()` auf. `get()` und `remove()` müssen das gesuchte Element finden. Die Methode `put()` muss ebenfalls den Knoten finden, in dem das neue Element eingefügt bzw. das Existierende ersetzt werden muss.

Da `treeSearch()` eine Laufzeit von $O(h)$ hat, ist es also wichtig, die Höhe h des Suchbaums so klein wie möglich zu halten. Im Idealfall ist diese Laufzeit dann also $O(\log n)$ für einen Suchbaum mit n Knoten.

Einfügen [Slide 5]

Grundgedanke:

- `treeSearch()` liefert den (externen) Knoten, wo sich k befände, falls es existierte.
- Genau dort muss (k, v) also eingefügt werden.

Ein voller Binärbaum unterstütze die folgende Methode:

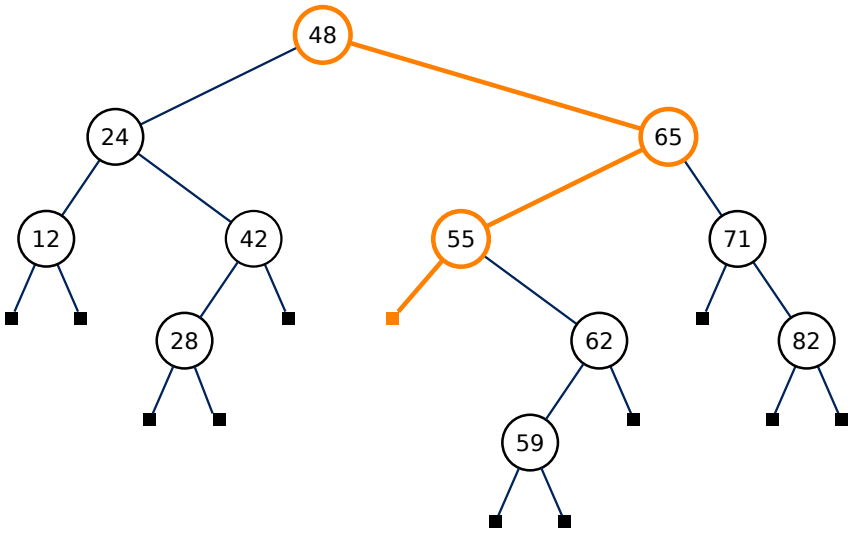
```
expandExternal(p, e) // Stores entry e at the external position p, and
                     // expands p to be internal with sentinel leaves.
```

```
Algorithm treeInsert(k, v):
  Require: A search key k to be associated with value v.
  Ensure: Return the position containing k.
  p ←treeSearch(root(), k)
  if k = key(p) then
    p.setValue(v)
  else
    expandExternal(p, (k, v))
  return p
```

Erklärung

Die Methode `put()` einer Suchbaum-basierten sortierten Zuordnungstabelle nutzt den hier gezeigten Algorithmus `treeInsert()`. Dieser ruft als erstes `treeSearch()` auf, um die Position p zu ermitteln, an der sich der einzufügende Schlüssel k befindet bzw. befinden wird. Existiert dieser Schlüssel bereits, wird sein existierender Wert durch den neuen Wert v ersetzt. Andernfalls ist p ein externer Knoten. In diesem Fall wird mittels der Methode `expandExternal()` das neue Schlüssel-Wert-Paar (k, v) in p gespeichert, und p wird durch Anhängen zweier leerer Blätter zu einem internen Knoten gemacht.

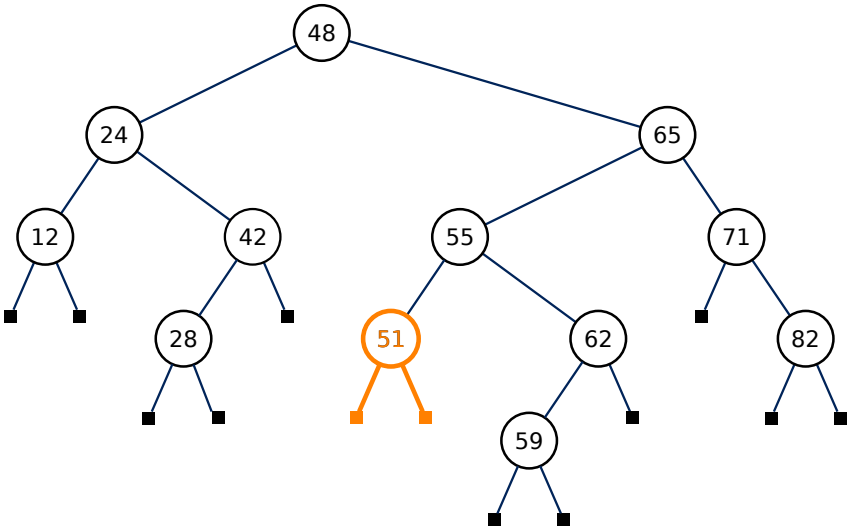
Beispiel: `treeInsert(51, ·)` [Slide 6]



Beispiel

Hier sehen wir einen Beispielaufruf von `treeInsert()` mit Schlüssel 51 und einem nicht weiter bezeichneten Wert. Falls der Schlüssel 51 bereits existierte, befände er sich im gezeigten Blatt.

Beispiel: `treeInsert(51, ·)` [Slide 7]



Beispiel

In diesem Blatt wird der Schlüssel nun also abgelegt, und der Knoten wird durch Anhängen zweier leerer Blätter zu einem internen Knoten.

Entfernen [Slide 8]

`treeRemove(`k`)`

1. $p =$ `treeSearch(root(), `k`)`
2. (a) Ist höchstens ein Kind von p intern, ersetzen wir p durch den Unterbaum, dessen Wurzel das interne Kind ist (falls vorhanden).
   (b) Andernfalls ersetzen wir p durch seinen Inorder-Vorgänger q (dessen rechtes Kind extern sein muss), und wenden Fall 2a auf q an.

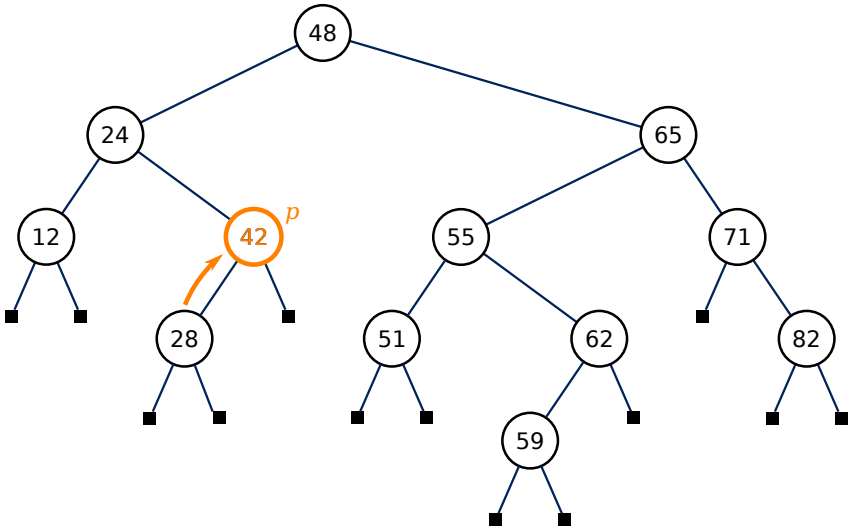
Übung: Schreiben Sie den Algorithmus für `treeRemove()` formal in Pseudocode.

Erklärung

`treeInsert()` ist sehr einfach, da neue Elemente immer in Blättern abgelegt werden. Sein Gegenstück `treeRemove()`, das von der Zuordnungstabellenmethode `remove()` verwendet wird, ist etwas aufwändiger, da sich das zu entfernende Element irgendwo im Baum befinden kann.

Als erstes sucht `treeRemove()` den Knoten p mit dem zu entfernenden Element mittels `treeSearch()`. Wird es nicht gefunden, dann ist der Job bereits getan. Ansonsten unterscheiden wir zwei Fälle:

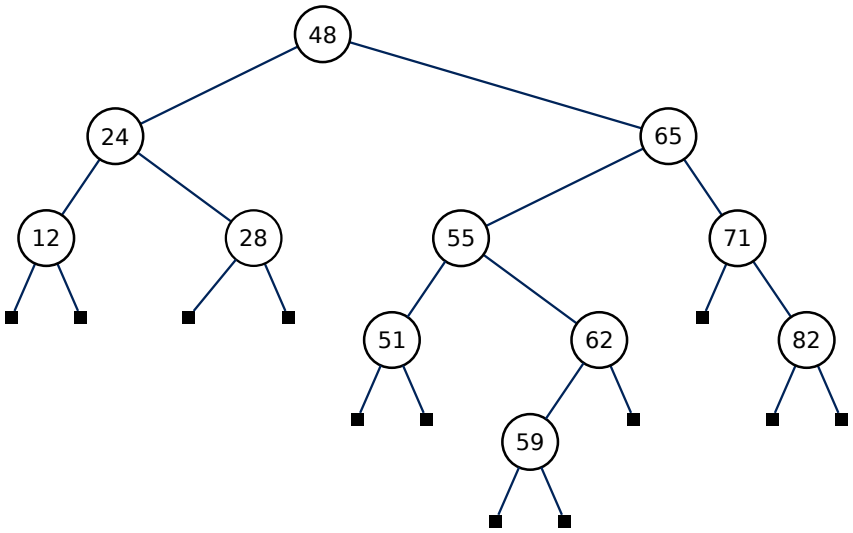
Beispiel: `treeRemove(42)` Fall 2a [Slide 9]



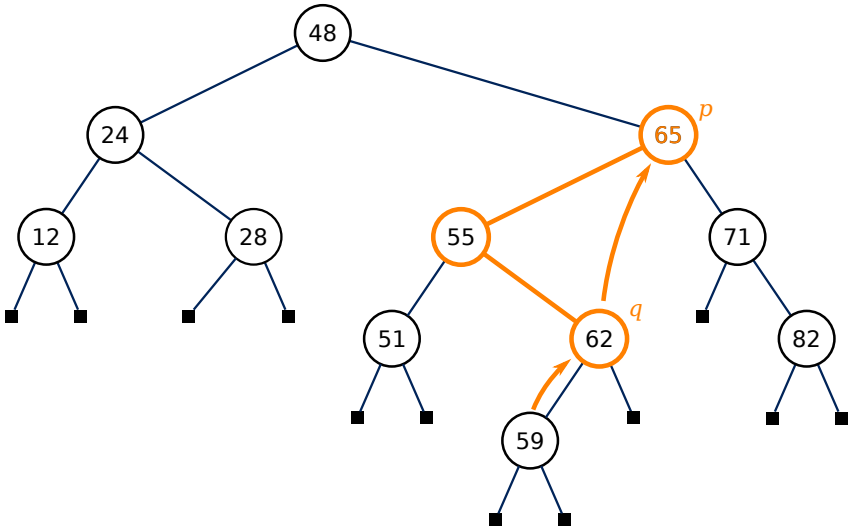
Erklärung

Liegt das zu entfernende Element in einem Knoten p mit genau einem internen Kind, dann entfernen wir den Knoten p und ersetzen ihn durch dieses interne Kind. Hat er zwei externe Kinder, dann ersetzen wir ihn einfach durch ein leeres Blatt.

Beispiel: `treeRemove(42)` Fall 2a [Slide 10]



Beispiel: `treeRemove(65)` Fall 2b [Slide 11]

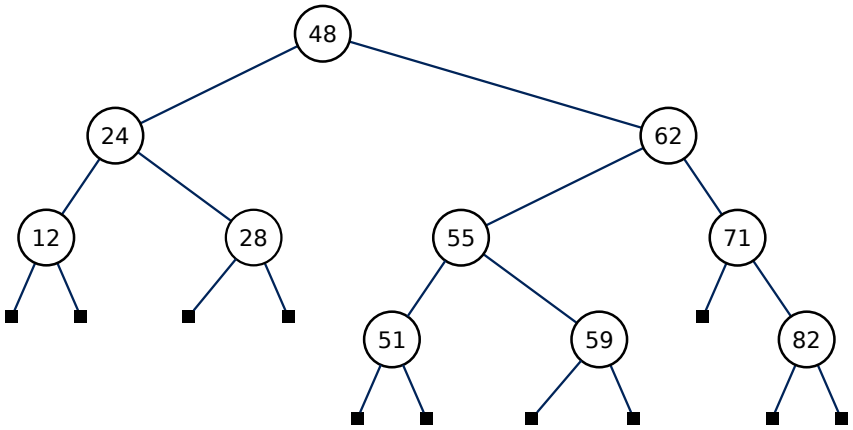


Erklärung

Andernfalls liegt unser zu entfernendes Element in einem Knoten p mit zwei internen Kindern. Einen solchen Knoten können wir nicht einfach entfernen. Stattdessen ersetzen wir das entfernte Element durch seinen Inorder-Vorgänger, was die Reihenfolge der Traversierung nicht beeinflusst.

Diesen Inorder-Vorgängerknoten q finden wir als den am weitesten rechts liegenden internen Nachkommen des linken Kindes von p. Daher muss mindestens das rechte Kind von q ein leeres Blatt sein.

Beispiel: `treeRemove(65)` Fall 2b [Slide 12]



Erklärung

Dieses können wir also nun entfernen, wie bereits im Fall 2a gezeigt.

Java: TreeMap() [Slide 13]

```
public class TreeMap<K,V> extends AbstractSortedMap<K,V> {
  // Lots of code omitted

  /** Utility used when inserting a new entry at a leaf of the tree */
  private void expandExternal(Position<Entry<K,V>> p, Entry<K,V> entry) {
    tree.set(p, entry);      // store new entry at p
    tree.addLeft(p, null);   // add new sentinel leaves as children
    tree.addRight(p, null);
  }

  /**
   * Returns the position in p's subtree having the given key (or else
   * the terminal leaf).
   * @param key a target key
   * @param p a position of the tree serving as root of a subtree
   * @return Position holding key, or last node reached during search
   */
  private Position<Entry<K,V>> treeSearch(Position<Entry<K,V>> p, K key) {
    if (isExternal(p))
      return p;              // key not found; return the final leaf
    int comp = compare(key, p.getElement());
    if (comp == 0)
      return p;                      // key found; return its position
    else if (comp < 0)
      return treeSearch(left(p), key); // search left subtree
    else
      return treeSearch(right(p), key); // search right subtree
  }

  /**
   * Returns the position with the maximum key in the subtree rooted
   * at p.
   * @param p a Position of the tree serving as root of a subtree
   * @return Position with maximum key in subtree
   */
  protected Position<Entry<K,V>> treeMax(Position<Entry<K,V>> p) {
    Position<Entry<K,V>> walk = p;
    while (isInternal(walk))
      walk = right(walk);
    return parent(walk);     // we want the parent of the leaf
  }

  /**
   * Returns the value associated with the specified key, or null if
   * no such entry exists.
   * @param key the key whose associated value is to be returned
   * @return the associated value, or null if no such entry exists
   */
  @Override
  public V get(K key) throws IllegalArgumentException {
```

```
    checkKey(key);           // may throw IllegalArgumentException
    Position<Entry<K,V>> p = treeSearch(root(), key);
    rebalanceAccess(p);      // hook for balanced tree subclasses
    if (isExternal(p)) return null; // unsuccessful search
    return p.getElement().getValue(); // match found
  }

  /**
   * Associates the given value with the given key. If an entry with
   * the key was already in the map, this replaced the previous value
   * with the new one and returns the old value. Otherwise, a new
   * entry is added and null is returned.
   * @param key   key with which the specified value is to be associated
   * @param value value to be associated with the specified key
   * @return the previous value associated with the key (or null, if
   * no such entry)
   */
  @Override
  public V put(K key, V value) throws IllegalArgumentException {
    checkKey(key);           // may throw IllegalArgumentException
    Entry<K,V> newEntry = new MapEntry<>(key, value);
    Position<Entry<K,V>> p = treeSearch(root(), key);
    if (isExternal(p)) {     // key is new
      expandExternal(p, newEntry);
      rebalanceInsert(p);    // hook for balanced tree subclasses
      return null;
    } else {                 // replacing existing key
      V old = p.getElement().getValue();
      set(p, newEntry);
      rebalanceAccess(p);    // hook for balanced tree subclasses
      return old;
    }
  }

  /**
   * Removes the entry with the specified key, if present, and returns
   * its associated value. Otherwise does nothing and returns null.
   * @param key the key whose entry is to be removed from the map
   * @return the previous value associated with the removed key, or
   * null if no such entry exists
   */
  @Override
  public V remove(K key) throws IllegalArgumentException {
    checkKey(key);           // may throw IllegalArgumentException
    Position<Entry<K,V>> p = treeSearch(root(), key);
    if (isExternal(p)) {     // key not found
      rebalanceAccess(p);    // hook for balanced tree subclasses
      return null;
    }
    else {
      V old = p.getElement().getValue();
      if (isInternal(left(p)) && isInternal(right(p))) {
```

```
      // both children are internal
      Position<Entry<K,V>> replacement = treeMax(left(p));
      set(p, replacement.getElement());
      p = replacement;
    } // now p has at most one child that is an internal node
    Position<Entry<K,V>> leaf = (isExternal(left(p)) ? left(p) : right(p));
    Position<Entry<K,V>> sib = sibling(leaf);
    tree.remove(leaf);
    tree.remove(p);        // sib is promoted in p's place
    rebalanceDelete(sib); // hook for balanced tree subclasses
    return old;
  }
}

/**
 * Returns the entry with greatest key less than or equal to given
 * key (or null if no such key exists).
 * @return entry with greatest key less than or equal to given (or
 * null if no such entry)
 * @throws IllegalArgumentException if the key is not compatible
 * with the map
 */
@Override
public Entry<K,V> floorEntry(K key) throws IllegalArgumentException {
  checkKey(key);           // may throw IllegalArgumentException
  Position<Entry<K,V>> p = treeSearch(root(), key);
  if (isInternal(p)) return p.getElement(); // exact match
  while (!isRoot(p)) {
    if (p == right(parent(p)))
      return parent(p).getElement(); // parent has next lesser key
    else
      p = parent(p);
  }
  return null;             // no such floor exists
}

/**
 * Returns the entry with greatest key strictly less than given key
 * (or null if no such key exists).
 * @return entry with greatest key strictly less than given (or null
 * if no such entry)
 * @throws IllegalArgumentException if the key is not compatible
 * with the map
 */
@Override
public Entry<K,V> lowerEntry(K key) throws IllegalArgumentException {
  checkKey(key);           // may throw IllegalArgumentException
  Position<Entry<K,V>> p = treeSearch(root(), key);
  if (isInternal(p) && isInternal(left(p)))
    return treeMax(left(p)).getElement(); // this is the predecessor to p
  // otherwise, we had failed search, or match with no left child
  while (!isRoot(p)) {
```

```
      if (p == right(parent(p)))
        return parent(p).getElement(); // parent has next lesser key
      else
        p = parent(p);
    }
    return null;              // no such lesser key exists
  }

  /**
   * Returns an iterable containing all entries with keys in the range
   * from <code>fromKey</code> inclusive to <code>toKey</code>
   * exclusive.
   * @return iterable with keys in desired range
   * @throws IllegalArgumentException if <code>fromKey</code> or <code>toKey</code> is not compa
   */
  @Override
  public Iterable<Entry<K,V>>
    subMap(K fromKey, K toKey) throws IllegalArgumentException {
    checkKey(fromKey);        // may throw IllegalArgumentException
    checkKey(toKey);          // may throw IllegalArgumentException
    ArrayList<Entry<K,V>> buffer = new ArrayList<>(size());
    if (compare(fromKey, toKey) < 0) // ensure that fromKey < toKey
      subMapRecurse(fromKey, toKey, root(), buffer);
    return buffer;
  }

  // utility to fill subMap buffer recursively (while maintaining order)
  private void subMapRecurse(K fromKey, K toKey, Position<Entry<K,V>> p,
                             ArrayList<Entry<K,V>> buffer) {
    if (isInternal(p))
      if (compare(p.getElement(), fromKey) < 0)
        // p's key is less than fromKey,
        // so any relevant entries are to the right
        subMapRecurse(fromKey, toKey, right(p), buffer);
      else {
        // first consider left subtree:
        subMapRecurse(fromKey, toKey, left(p), buffer);
        if (compare(p.getElement(), toKey) < 0) { // p is within range
          buffer.add(p.getElement()); // so add it to buffer, and consider
          // right subtree as well:
          subMapRecurse(fromKey, toKey, right(p), buffer);
        }
      }
  }
}
```

Laufzeiten der Methoden [Slide 14]

Methode	Laufzeit
`size`, `isEmpty`	$O(1)$
`get`, `put`, `remove`	$O(h)$
`firstEntry`, `lastEntry`	$O(h)$
`ceilingEntry`, `floorEntry`, `lowerEntry`, `higherEntry`	$O(h)$
`subMap`	$O(s+h)$
`entrySet`, `keySet`, `values`	$O(n)$

3 Rotation für selbstausgleichende Suchbäume

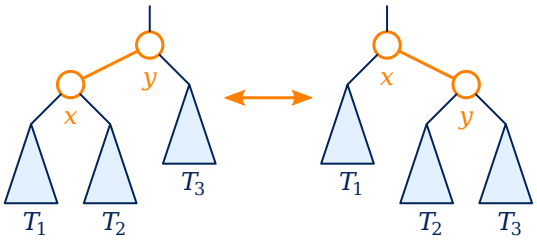
Video 2 beginnt hier.

Erklärung

Wir haben gesehen, dass unsere Hilfsmethoden `treeInsert()` und `treeRemove()` einen Ast des Suchbaums verlängern bzw. verkürzen. Folglich kann der Suchbaum beliebige Formen annehmen, je nach Sequenz der eingefügten Schlüssel. Um den Baum auf logarithmischer Höhe zu halten, können wir ihn jedoch umstrukturieren, ohne dass die Reihenfolge der Traversierung seiner Elemente verändert wird.

(Kleine) *Rotation* [Slide 15]

Basisoperation zur Reduktion der Höhe eines Suchbaums.



- Reihenfolge der Traversierung unverändert.
- Konstante Anzahl geänderter Eltern-Kind-Relationen ⇒ Implementierung in $O(1)$ möglich.

Diese Rotation eines Knotens über einen anderen liegt der nachfolgend beschriebenen einfachen und doppelten Rotation zugrunde. Daher bezeichnen wir sie hier als ***kleine Rotation***.

Erklärung

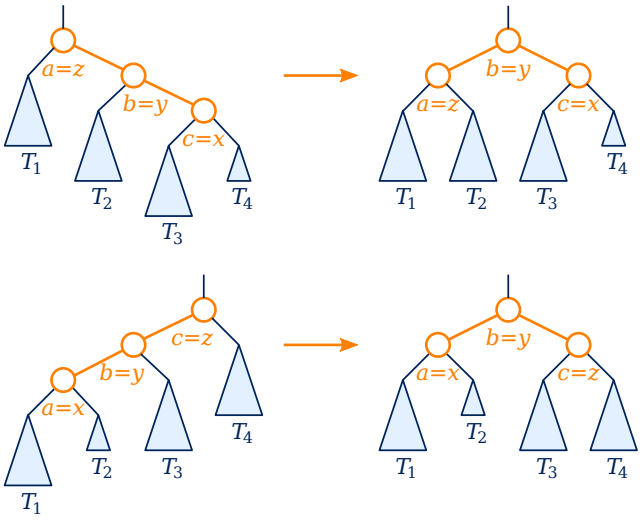
Die Basisoperation für solche Umstrukturierungen nennt sich ***Rotation*** und läuft ab, wie hier veranschaulicht. Jedes der blauen Dreiecke T_1, T_2 und T_3 repräsentiert einen Suchbaum. Dies sind drei Unterbäume der hier gezeigten Unterbäume, links mit Wurzel y und rechts mit Wurzel x.

Diese beiden Unterbäume sind äquivalent: Sie enthalten dieselben Knoten, die in derselben Reihenfolge traversiert werden. Obwohl im linken Unterbaum x ein Kind von y ist und im rechten Unterbaum umgekehrt, befinden sich in beiden Fällen alle Knoten in T_1 links von x, alle in T_2 rechts von x und links von y, und alle Knoten in T_3 rechts von y.

Was uns an dieser Rotation interessiert, ist die Tatsache, dass sie T_1 um ein Niveau anhebt und T_3 um ein Niveau absenkt. Dies können wir uns zunutze machen, um nach Einfüge- und Entfernungsoperationen eine logarithmische Höhe sicherzustellen.

Wichtig ist noch zu bemerken, dass eine solche Rotation bei einer geeigneten Datenstruktur lediglich eine konstante Laufzeit beansprucht, da nur eine konstante Anzahl Knoten und Kanten berührt wird.

Trinode Restructuring: *Einfache Rotation* [Slide 16]



Erklärung

Wir werden uns nun eine generalisierte Umstrukturierungsoperation definieren, mit der wir gezielt eine Reduktion der Höhe eines Suchbaums bewirken können. Sie betrifft bis zu drei Knoten, statt wie bisher zwei, und wird deshalb als *Trinode Restructuring* bezeichnet.

In den beiden hier gezeigten Fällen scheinen wir den höchstgelegenen Knoten z um den mittleren Knoten y als Scharnier nach unten zu rotieren. Diese sogenannte ***einfache Rotation*** wendet genau die Basisoperation, die wir soeben eingeführt haben, auf die Knoten y und z an. Der Knoten x bleibt davon unberührt.

Wir sehen, dass in beiden Beispielen die Höhe des gezeigten Unterbaums um eins reduziert wird.

Die Höhen der blau gezeichneten Unterbäume haben System. Die drei gleich groß gezeichneten Dreiecke repräsentieren Unterbäume identischer Höhe, und das kleiner gezeichnete Dreieck bezeichnet einen Unterbaum, dessen Höhe entweder um eins geringer ist als die der großen Dreiecke, oder auch identisch zu diesen. Dies ist eines von verschiedenen, typischen Szenarien, in denen diese einfache Rotation Anwendung findet.

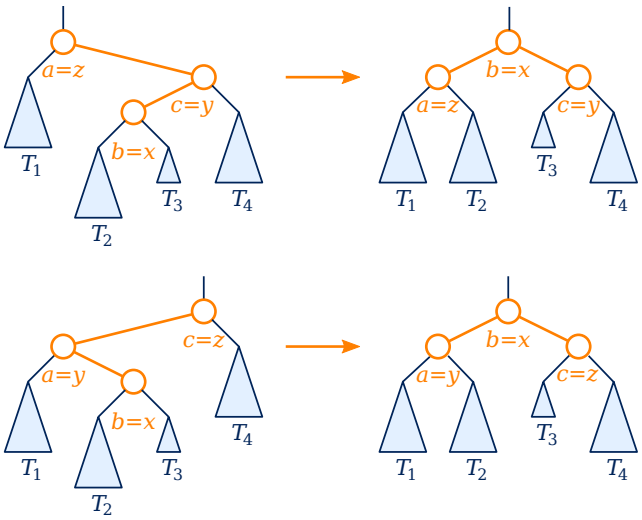
Beide Varianten der einfachen Rotation können wir übrigens auf dieselbe Weise einfach beschreiben:

Die drei rot hervorgehobenen Knoten bezeichnen wir von links nach rechts als a, b, c und von unten nach oben als x, y, z. Dann machen wir den mittleren Knoten b zur Wurzel dieses Unterbaums, und a und c zu seinem linken bzw. rechten Kind. Als deren Kinder hängen wir von links nach rechts die vier Unterbäume T_1 bis T_4.

Dies sind zwei von insgesamt vier möglichen Konfigurationen unserer drei Knoten als Vorgänger und Nachkommen voneinander.

Trinode Restructuring: *Doppelte Rotation* [Slide 17]

Tritt auf, wenn x bei Inorder-Traversierung zwischen y und z liegt.



Erklärung

Die beiden anderen ergeben sich, wenn der unterste Knoten x in der Inorder-Traversierung *zwischen* den beiden anderen Knoten y und z liegt. In diesem Fall ähnelt die gezeigte Umstrukturierung einer Rotation des oberen Knotens um den mittleren nach unten, und des unteren Knotens um den mittleren nach oben. Daher wird diese Operation ***doppelte Rotation*** genannt.

Tatsächlich lässt sich eine doppelte Rotation als zwei einfache Rotationen implementieren: Zuerst rotieren wir x über y. Das Ergebnis ist dann eine der beiden Ausgangssituationen der vorigen Seite. Als zweites rotieren wir x über z.

Wir machen uns die Sache hier aber noch einfacher. Wenn wir die beiden Varianten der doppelten Rotation wieder mittels a, b, c und x, y, z beschreiben, dann stellen wir fest, dass sich exakt dieselbe Beschreibung ergibt wie bei der einfachen Rotation!

Trinode Restructuring [Slide 18]

Diese Operation implementiert sowohl die einfache als auch die doppelte Rotation auf einheitliche und einfache Weise.

`restructure(`x`)`

Eingabe: Position x mit Elternknoten y und Großelternknoten z in einem Binärbaum T.

Ausgabe: T nach einem *trinode restructuring*.

1. Sei (a, b, c) eine Inorder-Liste der Positionen x, y und z, und sei (T_1, T_2, T_3, T_4) eine Inorder-Liste der vier Unterbäume, deren Wurzeln x, y und z sind.
2. Ersetze den Unterbaum mit Wurzel z durch den Unterbaum mit Wurzel b.
3. Mache a zum linken Kind von b. Erreiche ggf. durch Umhängen, dass T_1 und T_2 jeweils der linke und rechte Unterbaum von a sind.
4. Mache c zum rechten Kind von b. Erreiche ggf. durch Umhängen, dass T_3 und T_4 jeweils der linke und rechte Unterbaum von c sind.

Erklärung

Wir werden also algorithmisch nicht mehr zwischen einfacher und doppelter Rotation oder deren beiden Varianten unterscheiden, sondern alle vier Transformationen einheitlich durch einen einzigen Algorithmus `restructure(x)` beschreiben. x ist hier wie bisher der unterste der drei beteiligten Knoten. Damit ergeben sich y als der Elternknoten von x und z als der Elternknoten von y. Die Bezeichner a, b, c und T_1, T_2, T_3, T_4 ergeben sich aus den Eltern-Kind-Relationen zwischen x, y, z.

Die Umstrukturierung erfolgt nun genau so, wie wir sie vorhin beschrieben haben: Wir machen b zur Wurzel a und c zu seinen Kindern, und T_1 bis T_4 machen wir zu den Unterbäumen von a und c.

Damit haben wir in `restructure(x)` eine allgemeine Funktion, die wir nutzen können, um die Höhe des am Großelternknoten von x gewurzelten Unterbaums um ein Niveau zu reduzieren.

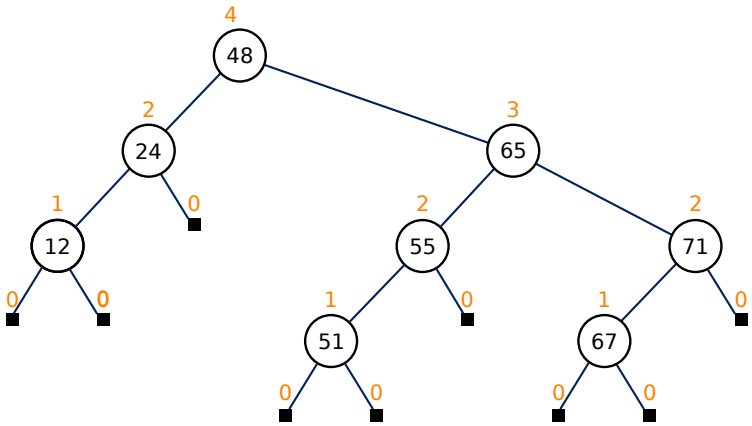
4 AVL-Bäume

Video 3 beginnt hier.

AVL-Baum [Slide 19]

Ein binärer Suchbaum ist ein ***AVL-Baum***, falls er die Eigenschaft des ***Höhenausgleichs*** *(height balance)* besitzt:

Für jeden internen Knoten unterscheiden sich die Höhen seiner Kinder maximal um 1.



Anmerkung

Es folgt unmittelbar, dass jeder Unterbaum eines AVL-Baums seinerseits ein AVL-Baum ist.

Definition

Der erste selbstbalancierende Suchbaum, den wir betrachten, ist der ***AVL-Baum***. Er ist nach den Anfangsbuchstaben der Familiennamen seiner Erfinder benannt, Adelson-Velskij und Landis.

Ein AVL-Baum ist ein binärer Suchbaum mit der Eigenschaft des ***Höhenausgleichs***. Höhenausgleich liegt vor, wenn sich für jeden internen Knoten die Höhen seiner Kinder um maximal 1 unterscheiden.

Der hier gezeigte Baum ist ein AVL-Baum. Die Höhe jedes Unterbaums ist über seiner Wurzel in rot angegeben. So sehen wir leicht, dass die Eigenschaft des Höhenausgleichs überall gegeben ist.

Höhe eines AVL-Baums [Slide 20]

Proposition: Für einen AVL-Baum, $h(n) \in O(\log n)$.

Beweisskizze:

Die minimale Anzahl $n(h)$ interner Knoten wächst exponentiell in h:

- $n(1) = 1$ und $n(2) = 2$.
- Für minimales $n(h)$ unterscheiden sich die Höhen sämtlicher Unterbaum-Geschwister jeweils um 1.
- Daher, für $h \geq 3$:

  $$n(h) = 1 + n(h-1) + n(h-2) > 2n(h-2)$$

Also $n(h) \in \Omega(c^h)$ für eine gewisse Konstante $c > 1$, und damit $h(n) \in O(\log n)$.

Erklärung

Garantiert der Höhenausgleich, dass die Höhe des gesamten Baums logarithmisch in der Anzahl seiner Knoten ist?

Zum Beweis argumentieren wir, dass die minimale Anzahl n der internen Knoten exponentiell mit der Höhe h des AVL-Baums wächst. Hierzu drücken wir die minimale Anzahl $n(h)$ interner Knoten als Funktion der Höhe h rekursiv aus. Die minimale Anzahl $n(h)$ interner Knoten eines Baums ist gleich der minimalen Anzahl der internen Knoten seiner beiden Unterbäume, plus den gemeinsamen Elternknoten dieser beiden Unterbäume.

In einem AVL-Baum mit minimaler Anzahl Knoten bei gegebener Höhe unterscheiden sich die Höhen sämtlicher Unterbaum-Geschwister um 1. Also ist $n(h) = 1 + n(h-1) + n(h-2)$. Der Rekursionsanfang ist durch AVL-Bäume der Höhe 1 und 2 gegeben, die jeweils mindestens einen bzw. zwei interne Knoten besitzen.

Da $n(h-1) > n(h-2)$ sein muss, ist $n(h) > 2n(h-2)$. $n(h)$ muss sich also für jede additive Vergrößerung von h um 2 mindestens verdoppeln. Eine Funktion, deren Wert multiplikativ wächst, wenn sich ihr Argument additiv vergrößert, wächst exponentiell. Daher wächst $n(h)$ mindestens exponentiell in h. Umgekehrt bedeutet dies, dass h höchstens logarithmisch in n wachsen kann.

So weisen wir nach, dass der Höhenausgleich tatsächlich logarithmische Baumhöhen garantiert.

Einfügen [Slide 21]

1. p =`treeInsert()`

   Siehe `treeInsert()`.

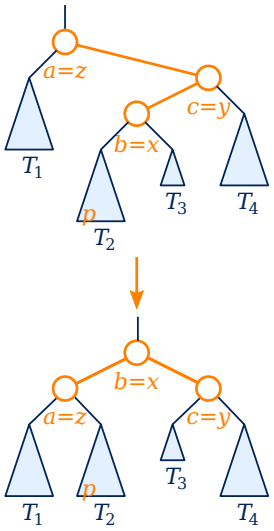
2. Sei z der erste *unausgeglichene* Knoten entlang des Pfades von p bis zur Wurzel.

   Sei y (ein Vorfahr von p) das höhere Kind von z.

   Sei x (ebenfalls ein Vorfahr von p, möglicherweise $x = p$) das höhere Kind von y.

   `restructure(`x`)`

   Siehe `restructure()`.



Der Baum ist nun *global* ausgeglichen.

Erklärung

Nun müssen wir sicherstellen, dass nach dem Einfügen oder Entfernen eines Knotens der Höhenausgleich wieder hergestellt wird. Beginnen wir mit dem Einfügen.

Zuerst verwenden wir `treeInsert()`, um das neue Element in den Baum einzufügen. Hier kann es passieren, dass (erstens) der neue Knoten p die Höhe eines Unterbaums vergrößert und (zweitens) dadurch der Höhenausgleich verletzt wird. Dies ist der Fall, der uns interessiert.

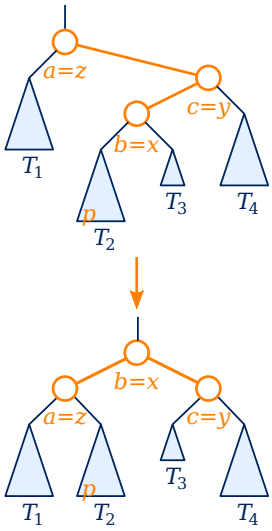
Rechts sehen wir einen solchen Fall illustriert. Der neu eingefügte Knoten p vergrößert die Höhe des Unterbaums T_2. Dadurch erhöht sich die Höhendifferenz der Kinder des Knotens z von 1 auf 2.

Sei z allgemein der erste solche Knoten auf dem Pfad von p zur Wurzel, dessen Höhenausgleich durch die Einfügung von p verletzt wird. Sei y das höhere Kind von z (das also auf demselben Pfad liegen muss), und sei x das höhere Kind von y (das ebenfalls auf diesem Pfad liegen muss).

Nun rufen wir `restructure(x)` auf. Dies stellt den Höhenausgleich der beteiligten Unterbäume x, y, z wieder her. Dabei wird die Höhe des betrachteten Unterbaums nach der Einfügung um eins reduziert. Mit anderen Worten, nach dem Umstrukturieren ist die Höhe dieses Unterbaums dieselbe wie seine Höhe vor der Einfügung. Damit ist der Baum nun *global* ausgeglichen.

Korrektheit der Einfügung [Slide 22]

- Beide *höheren Kinder* sind *eindeutig* (ansonsten existiert z nicht).
- x muss existieren, da das Einfügen von p unmöglich den Höhenausgleich von p's Elternknoten verletzen kann.
- Da y nach Einfügen von p *ausgeglichen bleibt* (sonst wäre es z), waren seine Unterbäume ursprünglich gleich hoch, und der Unterbaum von x ist nach `treeInsert()` um 1 höher als vorher.
- Die Höhe von b nach `restructure()` ist gleich der ursprünglichen von z. Die Gesamthöhe des Baums bleibt also unverändert.



Frage: Häh? Bedeutet dies, dass eine Einfügung niemals eine Erhöhung des Baums nach sich ziehen kann?

Antwort: Dieser Beweis betrifft nur den Fall, wo der Baum nach der Einfügung nicht mehr höhenausgeglichen ist. Natürlich gibt es Fälle, wo er höhenausgeglichen bleibt (und damit z nicht existiert); dabei kann er sich durchaus erhöhen.

Erklärung

Garantiert diese Prozedur tatsächlich den Höhenausgleich nach einer Einfügung?

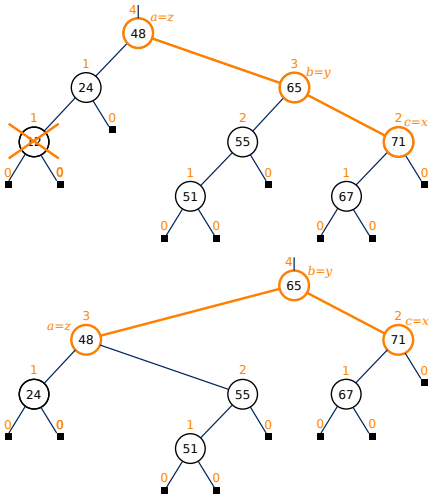
Zunächst stellen wir fest, dass x, y, z wie definiert existieren müssen. Gemäß der Definition von z sind y und x eindeutig.

x ist möglicherweise mit p identisch. x muss existieren, denn ansonsten wäre p mit y identisch, aber dies widerspricht der Tatsache, dass ein neu eingefügter Knoten nicht den Höhenausgleich seines Elternknotens verletzen kann.

y bleibt durch die Einfügung von p ausgeglichen, ansonsten wäre er nicht y, sondern allenfalls z. Sein Unterbaum, in den p eingefügt wird, muss nach der Einfügung höher sein als sein anderer Unterbaum. Ansonsten würde z nicht existieren. Daraus folgt, dass vor der Einfügung beide Unterbäume von y dieselbe Höhe besaßen.

Da `restructure(x)` durch Anheben der Unterbäume von x die Höhe des gesamten betrachteten Unterbaums um eins reduziert, bleibt die Gesamthöhe des Baumes durch diese Prozedur von Einfügen und Umstrukturieren unverändert.

Entfernen [Slide 23]



1. `treeRemove()`

   Siehe `treeRemove()`.

   Sei p das Kind des entfernten Knotens, das ihn ersetzt hat.

2. Sei z der (einzige) *unausgeglichene* Knoten entlang des Pfades von p bis zur Wurzel.

   Sei y das höhere Kind von z.

   y kann kein Vorfahr von p sein.

   Sei x

   - das höhere Kind von y, falls es existiert; andernfalls
   - das Kind auf derselben Seite von y wie y von z.

     Übung: Warum?

   `restructure(`x`)`

   Siehe `restructure()`.

   Der Baum ist nun *lokal* unterhalb von b ausgeglichen.

3. Sei $p = b$, und iteriere Schritt 2.

Erklärung

Betrachten wir nun die Entfernung eines Knotens. Hierbei kann es passieren, dass (erstens) die Höhe des Unterbaums, aus dem dieser Knoten entfernt wurde, um eins reduziert wird, und (zweitens) dadurch der Höhenausgleich eines Vorfahrs dieses Knotens verletzt wird. In diesem Fall muss der Höhenausgleich des AVL-Baums wieder hergestellt werden.

Sei z dieser unausgeglichene Knoten auf dem Pfad von p zur Wurzel. Sei y das höhere Kind von z, also im anderen Unterbaum und kein Vorfahr von p.

Sei nun x das höhere Kind von y, falls diese Kinder verschieden hoch sind. Sind beide Kinder von y gleich hoch, dann sei x das Kind von y, das auf derselben Seite liegt wie y von z. Warum? Das können Sie sich selbst überlegen!

Eine solche Situation ist im Beispiel rechts illustriert. Vor der Entfernung des Knotens betrugen die Höhen der Kinder des Knotens mit der 48 2 und 3. Durch die Entfernung des Knotens mit der 12 reduziert sich die Höhe dieses Unterbaums von 2 auf 1 und verletzt dadurch den Höhenausgleich des Knotens mit der 48. Daher ist dieser Knoten unser z, und sein anderes, rechtes Kind ist y. Die Höhen der Kinder von y sind identisch. Daher ist x wiederum das *rechte* Kind von y.

Nun rufen wir `restructure(x)` auf. Dadurch wird x um ein Niveau angehoben und z – und damit auch der durch die Entfernung erniedrigte Unterbaum – um ein Niveau abgesenkt. Damit ist der betrachtete Unterbaum ab seiner neuen Wurzel b wieder ausgeglichen.

An dieser Stelle gibt es zwei Möglichkeiten. Hat sich die Höhe des betrachteten Unterbaums durch diese Operation nicht verändert, sind also die Höhen von z *vor* dem Umstrukturieren und von b *nach* dem Umstrukturieren identisch, dann ist der Baum nun auch *global* ausgeglichen. Dies ist in unserem Beispiel der Fall.

Es ist jedoch möglich, dass sich die Höhe dieses Unterbaums durch das Umstrukturieren

> um eins vermindert. In unserem Beispiel wäre dies ohne den Knoten mit der 51 der Fall. In dieser Situation bezeichnen wir die Wurzel b des höhenreduzierten Unterbaums als p und iterieren.
>
> Auf diese Weise werden wir maximal $O(\log n)$ Mal umstrukturieren müssen, bis wir im Extremfall bei der Wurzel ankommen. Dort kann eine Reduktion der Höhe des Baums keine Verletzung des Höhenausgleichs mehr bedeuten.

Korrektheit der Entfernung [Slide 24]

- Das *trinode restructuring* reduziert möglicherweise die Höhe des Unterbaums mit Wurzel b, was möglicherweise den Höhenausgleich eines Vorfahrs von b verletzt.
- Daher muss Schritt 2 bis zu $O(\log n)$ Mal iteriert werden.

Übung: Beweisen Sie die Existenz von z, y, x und vervollständigen Sie den Korrektheitsbeweis.

Laufzeiten der AVL-Methoden [Slide 25]

Wie beim allgemeinen binären Suchbaum, mit $h \in O(\log n)$.

Anmerkung

Umstrukturierende AVL-Methoden bestehen allgemein aus zwei Phasen:

- Baumabwärts: Suche
- Baumaufwärts: Rotationen

Java: AVL-Baum [Slide 26]

Anmerkung

- Die Basis-Klasse `TreeMap` ruft an den notwendigen Stellen `rebalance()`-Methoden auf, die von der abgeleiteten Klasse `AVLTreeMap` implementiert werden.
- Jeder Knoten speichert die Höhe seines Unterbaums.

5 (2,4)-Bäume

Mehrweg-Suchbäume *(multiway search trees)* [Slide 27]

Video 4 beginnt hier.

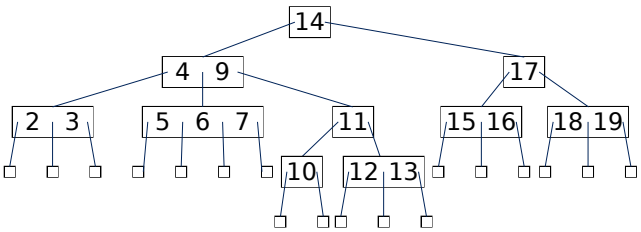
Knoten w eines geordneten Baums ist ein ***d-Knoten***, wenn er d Kinder hat.

Ein ***Mehrweg-Suchbaum*** ist ein geordneter Baum T mit den folgenden Eigenschaften:

1. Jeder interne Knoten ist ein d-Knoten mit $d \geq 2$.
2. Jeder interne d-Knoten w mit Kindern $c_1, ..., c_d$ speichert $d-1$ Elemente (k_i, v_i), wobei $k_1 \leq \cdots \leq k_{d-1}$.
3. Wir definieren $k_0 = -\infty$ und $k_d = +\infty$. Für jedes Element (k, v), das in einem Unterbaum von w mit Wurzel c_i gespeichert ist, gilt $k_{i-1} \leq k \leq k_i$.

Anmerkung

Ein binärer Suchbaum ist ein Mehrweg-Suchbaum.



Definition

Bisher haben wir uns nur mit binären Suchbäumen befasst, und wir werden in Kürze auch dorthin zurückkehren. Zwischendurch ist es jedoch hilfreich, einen Blick auf nicht-binäre Suchbäume zu werfen, sogenannte ***Mehrweg***-Suchbäume.

Zunächst definieren wir einen d-Knoten als einen Knoten mit d Kindern, die eine *Sequenz* bilden, und nicht lediglich eine Menge. Sie erinnern sich: Ein Baum, dessen Kindknoten diese Eigenschaft besitzen, ist ein *geordneter* Baum.

Ein *Mehrweg-Suchbaum* ist nun eine einfache Generalisierung eines binären Suchbaums:

1. Jeder interne Knoten hat *mindestens* zwei Kinder.
2. Jeder interne d-Knoten speichert $d-1$ Schlüssel-Wert-Paare, wobei die Schlüssel eine aufsteigende Sequenz gemäß eines gegebenen Operators $\leq$ bilden.
3. Die Kinder eines Knotens sind gewissermaßen abwechselnd zwischen den Schlüsseln des Knotens eingehängt, so dass die Werte aller Schlüssel, die sich in einem gegebenen Unterbaum befinden, zwischen den Schlüsseln liegen, zwischen denen dieser Unterbaums an seinem Elternknoten hängt.

Diese Eigenschaften können wir gut an dieser Illustration nachvollziehen. Jeder interne Knoten hat mindestens 2 Kinder, und jeder interne d-Knoten speichert genau $d-1$ Elemente. (Die Grafik zeigt hier wieder nur die Schlüssel).

Alle Schlüssel im linken Unterbaum der Wurzel mit dem Schlüssel 14 sind kleiner als 14, und alle Schlüssel in ihrem rechten Unterbaum sind größer als 14. Alle Schlüssel im mittleren Unterbaum des Knotens mit den Schlüsseln 4 und 9 liegen zwischen 4 und 9. Und so weiter.

Wir können uns leicht eine Generalisierung der Inorder-Traversierung von Binärbäumen vorstellen, die die Schlüssel eines Mehrweg-Suchbaums in aufsteigender Reihenfolge besucht.

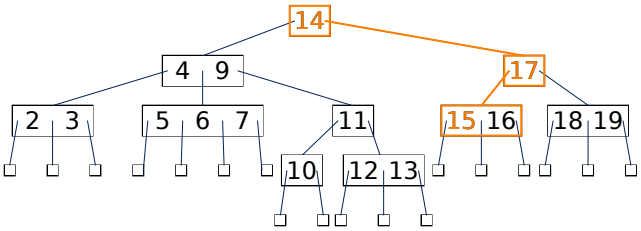
Zahl externer Knoten [Slide 28]

Proposition: Ein Mehrweg-Suchbaum mit n Elementen hat genau $n+1$ externe Knoten.

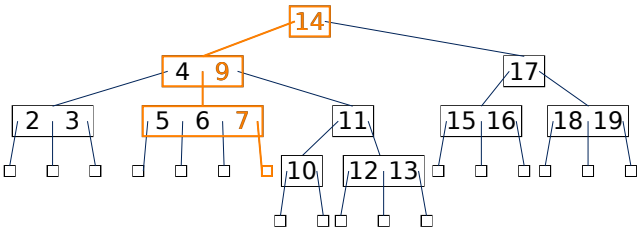
Beweis: (Übung)

Beispiel: Suche [Slide 29]

Suche nach der 15:



Suche nach der 8:



Erklärung

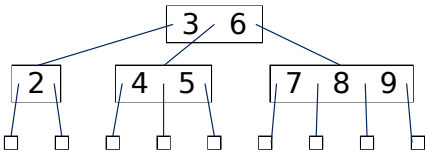
Die Suche nach Schlüsseln in einem Mehrweg-Suchbaum erfolgt analog zum binären Suchbaum. In jedem Knoten vergleichen wir den gesuchten Schlüssel mit den in diesem Knoten gepeicherten Schlüsseln. Finden wir ihn darunter, wie hier oben die 15, dann ist die Suche beendet. Andernfalls identifizieren wir den Unterbaum, in dem sich der gesuchte Schlüssel befinden muss, falls er existiert, und suchen dort rekursiv weiter. Landen wir in einem leeren Blatt, dann wissen wir, dass der gesuchte Schlüssel nicht existiert, wie hier im unteren Beispiel der Schlüssel 8.

B-Bäume [Slide 30]

Ein ***B-Baum*** der Ordnung m ist ein Mehrweg-Suchbaum mit den folgenden Eigenschaften:

Größeneigenschaft: Jeder interne Knoten hat mindestens $m/2$ und höchstens m Kinder.

Tiefeneigenschaft: Alle externen Knoten haben dieselbe Tiefe.



Definition

Genau wie allgemeine binäre Suchbäume haben allgemeine Mehrweg-Suchbäume das Problem, dass sie im Verhältnis zur Anzahl ihrer Knoten sehr tief und damit ineffizient werden können. Daher führen wir hier eine ***Tiefeneigenschaft*** ein, die besagt, dass alle externen Knoten dieselbe Tiefe besitzen.

Beachten Sie, dass diese Eigenschaft stärker ist, als die Eigenschaft des Höhenausgleichs, die wir bei AVL-Bäumen eingefordert haben. Das liegt natürlich daran, dass ein binärer

Suchbaum nur dann diese starke Tiefeneigenschaft besitzen kann, wenn er bei einer Höhe von h exakt $2^h - 1$ Schlüssel enthält. Um eine beliebige Anzahl Schlüssel zuzulassen, mussten wir daher ein schwächeres Höhenausgleichskriterium formulieren.

Bei einem Mehrweg-Suchbaum ist dies nicht nötig, da wir die Anzahl Schlüssel *innerhalb* der Knoten variieren können, um die stärkere Tiefeneigenschaft durchzusetzen.

Ein sogenannter ***B-Baum*** der Ordnung m ist ein Mehrweg-Suchbaum mit dieser Tiefeneigenschaft und darüber hinaus der ***Größeneigenschaft***, dass jeder interne Knoten mindestens $m/2$ und höchstens m Kinder hat.

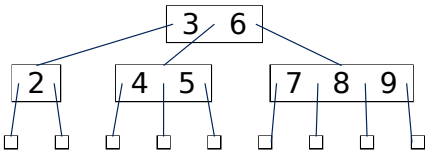
Der hier gezeigte Mehrweg-Suchbaum ist ein B-Baum der Ordnung 4.

B-Bäume sind gut für Dateisysteme geeignet, da es effizienter ist, wenige größere d-Knoten einzulesen als viele 2-Knoten. In der Tat werden sie in vielen Dateisystemen und Datenbanken verwendet.

(2,4)-Bäume [Slide 31]

Wir führen (2,4)-Bäume (und allgemeine B-Bäume) hauptsächlich ein, um Hintergrund und Intuition für Rot-Schwarz-Bäume zu vermitteln.

Ein ***(2,4)-Baum*** (auch 2–4-Baum oder 2–3–4-Baum) ist ein B-Baum der Ordnung 4.



Erklärung

Wir werden uns nun auf einen speziellen Vertreter der B-Baum-Familie konzentrieren, nämlich den B-Baum der Ordnung 4, auch als (2,4)-Baum bekannt. Unser bereits gezeigtes Beispiel ist also ein (2,4)-Baum.

Den (2,4)-Baum führen wir hier hauptsächlich ein, um Hintergrund und Intuition für die sogenannten Rot-Schwarz-Bäume zu vermitteln, denen wir uns in Kürze zuwenden werden.

Höhe eines (2,4)-Baums [Slide 32]

Proposition: Die Höhe eines (2,4)-Baums ist $\Theta(\log n)$.

Beweis: Sei h die Höhe eines (2,4)-Baums mit n Elementen. Wir zeigen nun, dass $\frac{1}{2}\log(n+1) \leq h \leq \log(n+1)$.

- Wegen der Größeneigenschaft haben wir maximal 4 Knoten in Tiefe 1, 4^2 Knoten in Tiefe 2, usw. Die Zahl *externer* Knoten ist also maximal 4^h.
- Ebenso haben wir wegen der Tiefeneigenschaft mindestens 2 Knoten in Tiefe 1, 2^2 Knoten in Tiefe 2, usw. Die Zahl *externer* Knoten ist also mindestens 2^h.
- Die Zahl externer Knoten ist gleich $n+1$. Daher haben wir

$$\begin{array}{rcccl} 2^h & \leq & n+1 & \leq & 4^h \\ h & \leq & \log_2(n+1) & \leq & 2h \end{array}$$

Erklärung

Wie immer bei Suchbäumen ist essenziell, dass die Höhe h des Baums maximal logarithmisch in der Anzahl der gespeicherten Elemente ist.

Um dies für den (2,4)-Baum zu zeigen, stellen wir zunächst fest, dass ein (2,4)-Baum wegen der Größeneigenschaft maximal 4 Knoten in Tiefe 1 enthalten kann, maximal 4^2 Knoten in Tiefe 2, usw. Damit ist die Zahl der *externen* Knoten maximal 4^h.

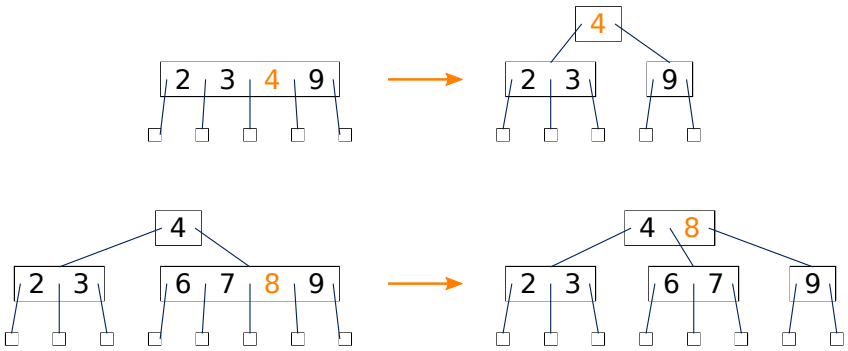
Wegen der Tiefeneigenschaft müssen sich mindestens 2 Knoten in Tiefe 1 befinden, mindestens 2^2 Knoten in Tiefe 2, usw. Folglich ist die Zahl der externen Knoten *mindestens* 2^h.

Wie man leicht zeigen kann, besitzt ein Mehrweg-Suchbaum mit n Elementen exakt $n+1$ externe Knoten. Diese Zahl liegt also zwischen den soeben hergeleiteten Schranken. Logarithmieren wir nun die gesamte Ungleichungskette, dann haben wir unsere Proposition bewiesen: Die Höhe eines (2,4)-Baums mit n Elementen ist $\Theta(\log n)$.

Einfügen [Slide 33]

Video 5 beginnt hier.

Bei ***Überlauf*** wird eine ***Spaltung*** durchgeführt:



Die Spaltung entweder eliminiert den Überlauf oder propagiert ihn in den Elternknoten, in welchem Fall iteriert wird. Ggf. wird ein neuer Wurzelknoten generiert.

Erklärung

Ein neues Element wird bei einem (2,4)-Baum im Gegensatz zu einem Binärbaum nicht in ein leeres Blatt eingefügt, sondern in einen Knoten, dessen Kinder leere Blätter sind. Enthält dieser Knoten danach vier Elemente, dann handelt es sich um einen ***Überlauf***. Dieser Überlauf wird dadurch behoben, dass der Knoten in zwei möglichst gleich große Knoten aufgespalten wird. Da der gemeinsame Elternknoten dadurch ein Kind mehr bekommt, muss es auch ein Element mehr bekommen. Dieses entnehmen wir einem der beiden neuen Knoten. Damit es *zwischen* den beiden neuen Knoten zu liegen kommt, nehmen wir entweder das letzte Element des linken Knotens oder das erste Element des rechten Knotens.

Im unteren Beispiel spalten wir auf diese Weise den 5-Knoten in einen 3-Knoten und einen 2-Knoten und verschieben dabei den Schlüssel 8 in den Elternknoten. Falls dieser Elternknoten nicht existiert, dann bekommt der Baum eine neue Wurzel, wie im oberen Beispiel. Falls der Elternknoten durch die Spaltung seinerseits überläuft, dann wird iteriert.

Quiz [Slide 34]

Oben ist ja genug Platz! Wir könnten die 8 genauso gut gleich oben einsortieren, anstatt sie erst nach unten durchzureichen und dann wieder hochzupropagieren.

A: Ja klar! Schlau.

B: Nicht schlau; das geht nicht oder bringt nichts.

D: weiß nicht

Entfernen [Slide 35]

1. Erreiche, dass das zu entfernende Element e des Knotens z sich in einem Knoten w befindet, der nur externe Kinder hat:
   (a) Finde den Vorgänger von e als am weitesten rechts liegendes Element f in Knoten w.
   (b) Vertausche die Elemente e und f.
2. Entferne das Element e und den zugehörigen externen Kind-Knoten.

Erklärung

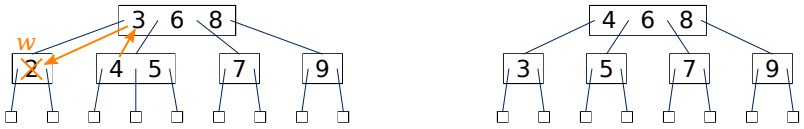
Entfernt werden können, wie beim binären Suchbaum, nur Elemente aus Knoten, die ein leeres Kind haben. Dies wird hier wieder dadurch erreicht, dass wir das zu entfernende Element ggf. durch seinen Vorgänger im Suchbaum ersetzen. Dessen freigewordenen Platz können wir dann mitsamt eines seiner benachbarten leeren Kinder entfernen.

Dabei schrumpft der betroffene Knoten.

Entfernen: Unterlauf [Slide 36]

3. Falls w nun die Größeneigenschaft verletzt (***Unterlauf***):

   - Hat ein *unmittelbarer Geschwisterknoten* 3 oder 4 Kinder, gleiche durch ***Transfer*** von Elementen aus.



   - Andernfalls ***fusioniere*** w mit einem Geschwisterknoten zu w'.

     Falls dies einen Unterlauf des Elternknotens von w' bewirkt, iteriere Schritt 3.

Erklärung

Falls dieser Knoten in einem B-Baum der Ordnung m nun weniger als $m/2$ Kinder hat, dann ist ein ***Unterlauf*** eingetreten. Bei einem (2,4)-Baum bedeutet dies, dass dieser Knoten nun überhaupt kein Element mehr enthält.

Diesen Unterlauf beheben wir auf eine von zwei verschiedenen Weisen:

Falls einer der beiden unmittelbaren, also direkt benachbarten, Geschwisterknoten über mehr als das Minimum der Elemente verfügt, füllen wir den unterbesetzten Knoten per ***Transfer*** von Elementen. Hierzu holen wir uns das Element des gemeinsamen Elternknotens, das zwischen den beiden betroffenen Unterbäumen liegt, in den unterbesetzten Knoten herunter. Die Lücke im Elternknoten füllen wir auf, indem wir das benachbarte Element des unmittelbaren Geschwisterknotens hinaufschieben. Dadurch bleiben die Mehrweg-Suchbaum-Eigenschaften erhalten.

Ist eine solche Transferlösung nicht möglich, weil sie einen Unterlauf im Nachbarknoten bewirken würde, dann wird der unterbesetzte Knoten mit einem seiner Nachbarknoten ***fusioniert***. Da sich dadurch die Anzahl der Kinder des Elternknotens reduziert, muss der Elternknoten seinerseits ein Element abgeben. Dies ist das Element zwischen den beiden betroffenen Kindknoten, und wir bringen es im neu fusionierten Knoten an seinem Platz in der Reihenfolge unter.

Dies kann natürlich einen Unterlauf im Elternknoten auslösen. In diesem Fall wird dieser Schritt iteriert. Im Extremfall tritt ein fusionierter Knoten an die Stelle der bisherigen Wurzel.

6 Rot-Schwarz-Bäume

Rot-Schwarz-Bäume [Slide 37]

Video 6 beginnt hier.

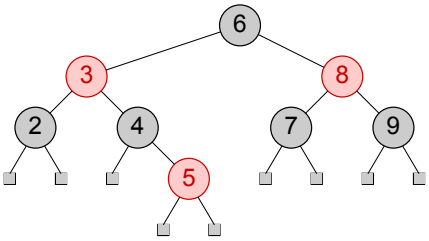
Ein ***Rot-Schwarz-Baum*** ist ein binärer Suchbaum mit den folgenden Eigenschaften:

Wurzel-Eigenschaft: Die Wurzel ist schwarz.

Externe Eigenschaft: Alle externen Knoten sind schwarz.

Rote Eigenschaft: Alle Kinder eines roten Knotens sind schwarz.

Tiefeneigenschaft: Alle externen Knoten haben dieselbe ***schwarze Tiefe***, definiert als die Anzahl der *echten* Vorfahren, die schwarz sind.



Die ***schwarze Höhe*** eines Rot-Schwarz-Baums ist die schwarze Tiefe seiner Blätter.

Definition

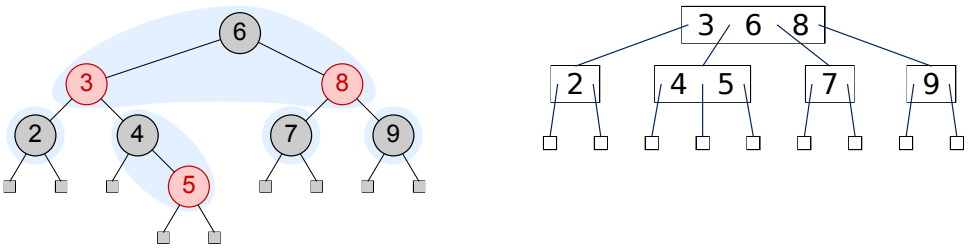
Rot-Schwarz-Bäume gehören zu den beliebtesten Datenstrukturen für sortierte Zuordnungstabellen. In einem Rot-Schwarz-Baum, wie unten illustriert, ist jeder Knoten entweder rot oder schwarz, und zwar nach der folgenden Systematik:

- Die Wurzel ist schwarz.
- Alle externen Knoten sind schwarz.
- Alle Kinder eines roten Knotens sind schwarz. Dies ist die sogenannte ***rote Eigenschaft***.

Ein Rot-Schwarz-Baum ist ein binärer Suchbaum, der über diese Eigenschaften hinaus auch die ***Tiefeneigenschaft*** besitzt: Alle Pfade von einem externen Knoten zur Wurzel enthalten dieselbe Anzahl schwarzer Knoten. Diese Anzahl, exklusive des externen Knotens selbst, wird als ***schwarze Tiefe*** bezeichnet. In unserer Illustration ist die schwarze Tiefe der externen Knoten gleich 2.

Rot-Schwarz → (2,4) [Slide 38]

Fusioniere jeden roten Knoten mit seinem Elternknoten.



Anmerkung

Die Tiefeneigenschaft des Rot-Schwarz-Baums entspricht der Tiefeneigenschaft des (2,4)-Baums, da jeder schwarze Knoten genau einem Knoten im (2,4)-Baum entspricht.

Daher die Färbung in zwei Farben: Nur schwarze Knoten zählen für die Tiefeneigenschaft. Rote Knoten erlauben kontrolliertes Ungleichgewicht (die Anzahl der Knoten entlang jedes Pfades ist begrenzt durch die Anzahl der schwarzen Knoten entlang dieses Pfades).

Erklärung

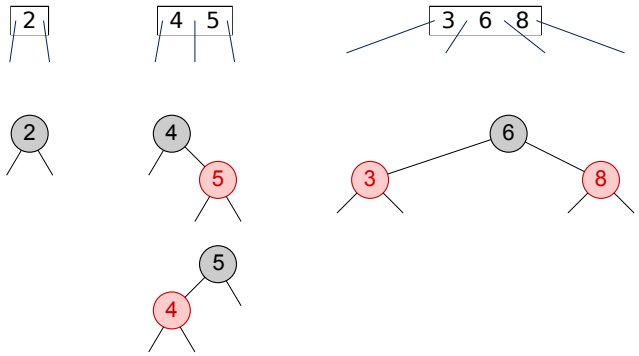
Rot-Schwarz-Bäme sind in dem Sinne zu (2,4)-Bäumen äquivalent, dass sie sich ineinander umwandeln lassen. Jeder Knoten eines (2,4)-Baums entspricht einem schwarzen Knoten eines Rot-Schwarz-Baums, samt seiner roten Kinder.

Damit sind auch die Tiefeneigenschaften beider Baumtypen äquivalent, denn nur schwarze Knoten zählen für die Tiefeneigenschaft. Rote Knoten erlauben ein kontrolliertes Ungleichgewicht des Baums. Dieses Ungleichgewicht ist nämlich dadurch beschränkt, dass entlang jedes Pfades zur Wurzel die Gesamtanzahl der Knoten durch die Anzahl der schwarzen Knoten begrenzt ist.

Aus der Korrespondenz von schwarzen Knoten zu (2,4)-Baum-Knoten folgt unmittelbar eine einfache Methode zur Umwandlung eines Rot-Schwarz-Baums in einen (2,4)-Baum: Fusioniere jeden roten Knoten mit seinem Elternknoten. Hier sehen wir unseren beispielhaften Rot-Schwarz-Baum und seinen äquivalenten (2,4)-Baum.

(2,4) → Rot-Schwarz [Slide 39]

Färbe jeden Knoten schwarz und transformiere ihn wie folgt:



Damit hat jeder rote Knoten einen schwarzen Elternknoten.

Erklärung

Diese Transformation lässt sich umkehren. Um einen (2,4)-Baum in einen Rot-Schwarz-Baum zu transformieren, machen wir jeden 2-Knoten zu einem schwarzen Knoten, jeden 3-Knoten zu einem schwarzen Knoten mit einem roten Kind, und jeden 4-Knoten zu einem schwarzen Knoten mit zwei roten Kindern.

Die Transformation eines 3-Knotens ist jedoch nicht eindeutig, da wir frei wählen können, ob der rote Knoten ein linkes oder ein rechtes Kind des schwarzen Knotens wird.

Höhe eines Rot-Schwarz-Baums [Slide 40]

Proposition: Die Höhe eines Rot-Schwarz-Baums ist $O(\log n)$.

Beweis: Sei h die Höhe eines Rot-Schwarz-Baums T mit n Elementen. Wir zeigen nun, dass $h \leq 2\log(n+1)$.

- Sei d die schwarze Höhe von T.
- Sei T' der (2,4)-Baum, der T entspricht, und sei $h' = d$ die Höhe von T'.
- Wegen der Höhe eines (2,4)-Baums ist $d = h' \leq \log(n+1)$.
- Wegen der *roten Eigenschaft* ist $h \leq 2d$, also $h \leq 2\log(n+1)$.

Erklärung

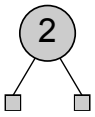
Um zu zeigen, dass die Höhe eines Rot-Schwarz-Baums logarithmisch in der Zahl n seiner internen Knoten ist, argumentieren wir einfach mit dem entsprechenden Beweis für den äquivalenten (2,4)-Baum. Aus der roten Eigenschaft folgt bereits die Proposition.

Einfügen [Slide 41]

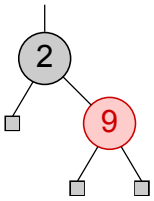
1. $x =$`treeInsert()`

   Siehe `treeInsert()`.

   Ist x der erste Knoten in T (die Wurzel), färbe ihn schwarz, und terminiere.



2. Andernfalls färbe ihn rot.



   Damit wird möglicherweise die *rote Eigenschaft* verletzt, falls der Elternknoten y von x ebenfalls rot ist (***Doppelrot***). Hier unterscheiden wir zwei Fälle, je nach Farbe des Geschwisterknotens s von y.

Erklärung

Die Einfüge- und Entfernungsoperationen sind bei Rot-Schwarz-Bäumen etwas komplizierter als bei den anderen Suchbäumen, die wir bisher betrachtet haben, da sie die rote Eigenschaft und die Tiefeneigenschaft erhalten müssen.

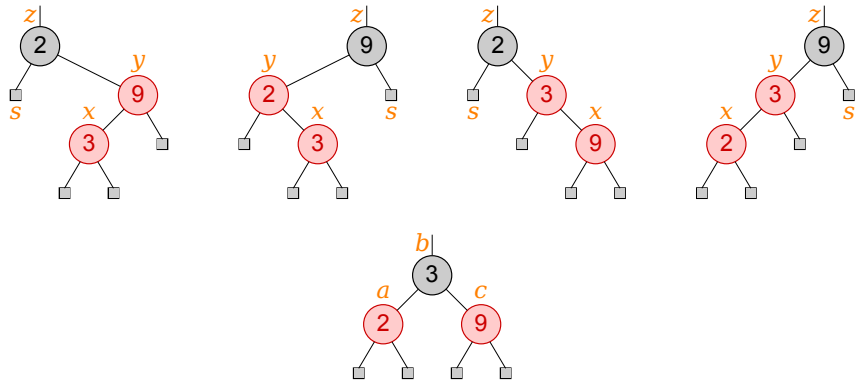
Beim Einfügen ist die Ausgangssituation einfach: Nach dem `treeInsert()` färben wir den neu eingefügten Knoten x schwarz, falls er die Wurzel ist. Andernfalls färben wir ihn rot.

Ist der Elternknoten y von x schwarz, dann kann der neue Knoten die Eigenschaften von Rot-Schwarz-Bäumen nicht verletzt haben, und die Einfügung ist beendet.

Ist der Elternknoten y von x allerdings ebenfalls rot, dann ist die rote Eigenschaft verletzt. Diese Situation bezeichnen wir als ***doppelrot***.

Einfügen Fall 1: s ist schwarz [Slide 42]

1. `restructure(`x`)` (die schwarze Höhe von T bleibt unverändert)
2. Färbe b schwarz und a und c rot.



Übung: Warum funktioniert dies nicht, wenn s rot ist?

Anmerkung

Die hier gezeigten Blätter können auch *interne* schwarze Knoten derselben schwarzen Höhe sein; siehe Einfügen Fall 2: s ist rot.

Erklärung

Um die rote Eigenschaft wieder herzustellen, unterscheiden wir zwei verschiedene Fälle, je nach der Farbe des Geschwisterknotens s des Elternknotens y des neu eingefügten Knotens x.

In diesem Beispiel haben wir soeben das Element 3 eingefügt. Dieser Knoten ist also unser x und rot. Sein Elternknoten heißt y und ist ebenfalls rot; ansonsten wäre die Einfügeoperation bereits beendet. Der Elternknoten z von y muss schwarz sein; ansonsten hätten wir bereits vor dem Einfügen der 3 ein Doppelrot. Und das andere Kind von z ist unser s, der Geschwisterknoten von y.

Im ersten Fall ist der Geschwisterknoten s des Elternknotens y schwarz, wie in diesen Beispielen. Alle vier Strukturen, die hier auftreten können, werden auf dieselbe Weise behandelt: Mittels `restructure(x)` wird der Knoten mit dem mittleren Schlüssel zur Wurzel dieses Unterbaums gemacht und schwarz gefärbt, und die beiden anderen beteiligten Knoten werden zu seinen roten Kindern. Damit ist das Doppelrot beseitigt. Gleichzeitig ist erreicht, dass alle Blätter die gleiche schwarze Tiefe haben, nämlich dieselbe schwarze Tiefe wie vor dem Einfügen des Knotens x.

Falls das Doppelrot tatsächlich durch Einfügen des Knotens x entstand, dann sind diese Blätter tatsächlich Blätter, denn die Kinder eines mittels `treeInsert()` eingefügten Knotens sind immer Blätter. Daher muss auch das andere Kind von y ein Blatt sein, denn sonst wäre die Tiefeneigenschaft nicht gegeben. Aus demselben Grund muss s ein Blatt sein, da es im betrachteten Fall 1 schwarz ist.

Dieser Fall 1 kann jedoch auch unter anderen Umständen eintreten, wie wir in Kürze sehen werden. Dann funkioniert diese Prozedur ganz genauso, aber die hier illustrierten Blätter sind nicht unbedingt Blätter, sondern komplette Unterbäume derselben schwarzen Höhe.

Ist s rot, dann würde diese Prozedur erneut zu einem Doppelrot führen. Daher behandeln wir diesen Fall auf andere Weise.

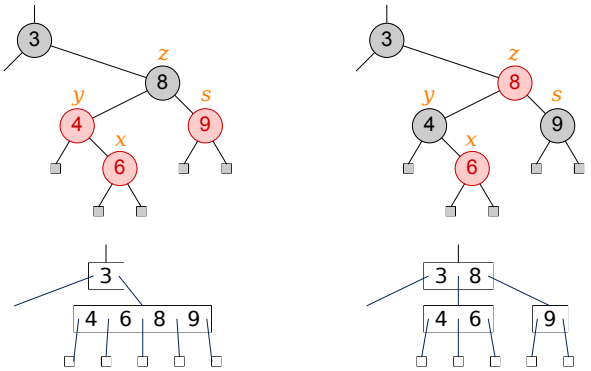
Einfügen Fall 2: s ist rot [Slide 43]

Dies entspricht dem Auftreten eines illegalen 5-Knotens im entsprechenden (2,4)-Baum. Es erfolgt das Äquivalent einer Spaltung durch ***Umfärbung*** von y, s und ihrem Elternknoten z:

- y und s werden schwarz.
- Ist z die Wurzel, bleibt es schwarz; andernfalls wird es rot.

Führt dies zu einem neuen Doppelrot, iteriere.

Die schwarze Höhe des Baums bleibt unverändert, es sei denn, z ist die Wurzel, in welchem Fall sie sich um 1 erhöht.



Übung: Warum funktioniert dies nicht, wenn s bereits vorher schwarz ist?

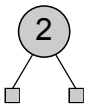
Erklärung

Im 2. Fall ist der Geschwisterknoten s des Elternknotens y des neu eingefügten Knotens x rot. Diese Konstellation entspricht einem illegalen 5-Knoten im entsprechenden (2,4)-Baum. Um diesen zu spalten, genügt es zunächst, einige Knoten des Rot-Schwarz-Baums umzufärben.

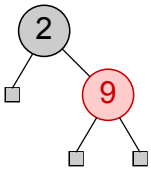
Die Geschwisterknoten y und s werden schwarz und damit zu separaten Knoten im äquivalenten (2,4)-Baum. x bleibt rot und befindet sich damit im (2,4)-Baum weiterhin in demselben Knoten wie y. z reichen wir zum gemeinsamen Elternknoten von y und s im (2,4)-Baum hoch, was einer Umfärbung auf rot im Rot-Schwarz-Baum entspricht. Ist kein solcher Elternknoten vorhanden, weil dieser 5-Knoten bereits die Wurzel war, dann wird z zur neuen Wurzel und bleibt schwarz. In diesem Fall erhöht sich die schwarze Höhe des Baums um 1.

Ist der Elternknoten von z rot, dann löst die Umfärbung von z auf rot wiederum ein Doppelrot aus, also einen Überlauf des Elternknotens im äquivalenten (2,4)-Baum. In diesem Fall bezeichnen wir z als unser neues x und iterieren.

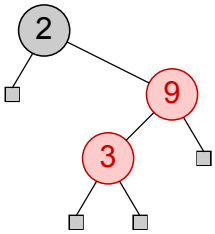
Beispielsequenz von Einfügungen [Slide 44]



Beispielsequenz von Einfügungen [Slide 45]

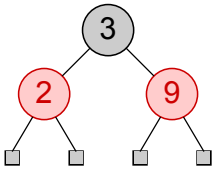


Beispielsequenz von Einfügungen [Slide 46]

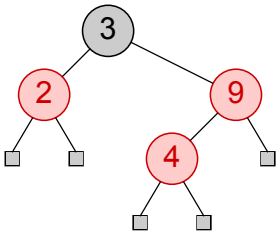


Einfügen Fall 1: s ist schwarz

Beispielsequenz von Einfügungen [Slide 47]

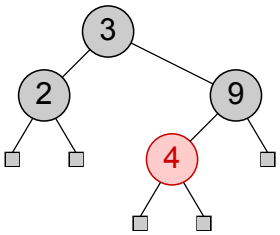


Beispielsequenz von Einfügungen [Slide 48]

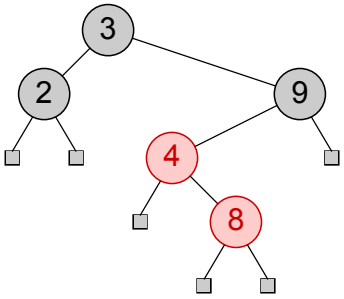


Einfügen Fall 2: s ist rot

Beispielsequenz von Einfügungen [Slide 49]

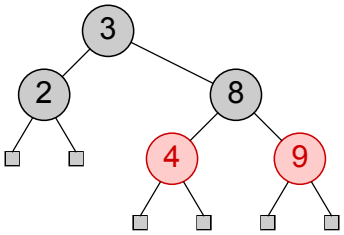


Beispielsequenz von Einfügungen [Slide 50]

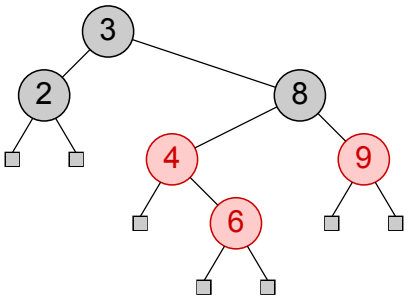


Einfügen Fall 1: s ist schwarz

Beispielsequenz von Einfügungen [Slide 51]

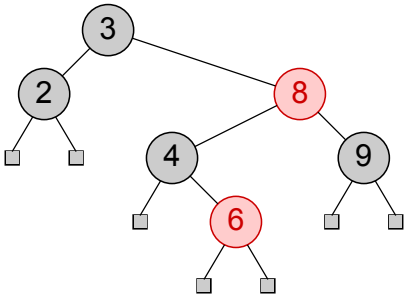


Beispielsequenz von Einfügungen [Slide 52]

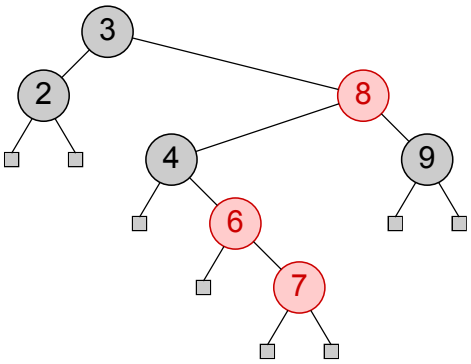


Einfügen Fall 2: s ist rot

Beispielsequenz von Einfügungen [Slide 53]

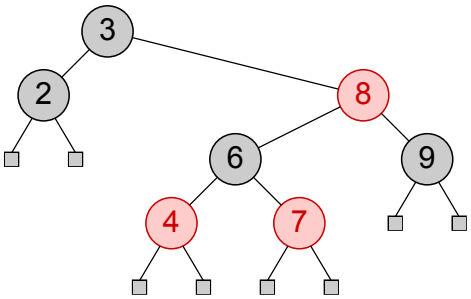


Beispielsequenz von Einfügungen [Slide 54]

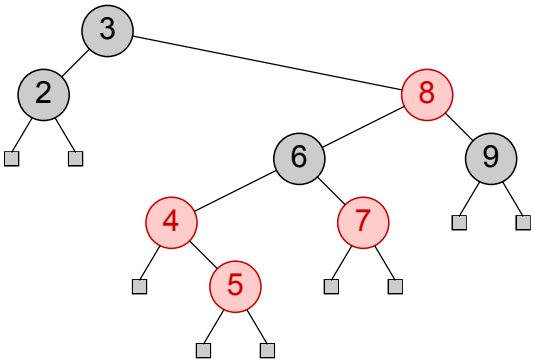


Einfügen Fall 1: s ist schwarz

Beispielsequenz von Einfügungen [Slide 55]

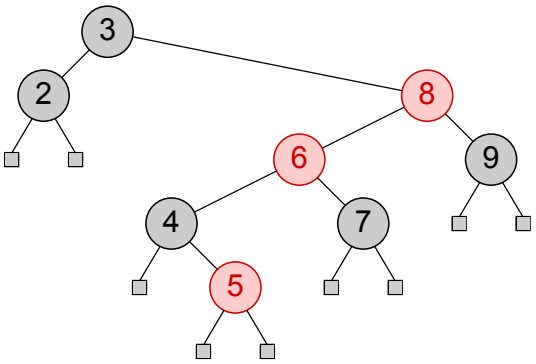


Beispielsequenz von Einfügungen [Slide 56]



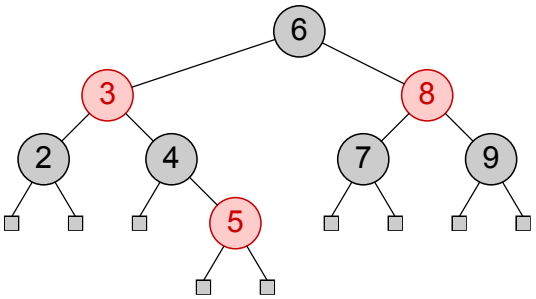
Einfügen Fall 2: s ist rot

Beispielsequenz von Einfügungen [Slide 57]



Neues Doppelrot; Einfügen Fall 1: s ist schwarz

Beispielsequenz von Einfügungen [Slide 58]



Animation

Betrachten wir nun eine Sequenz verschiedener Einfügeoperationen in einen zunächst leeren Rot-Schwarz-Baum. Der erste Knoten, mit dem Schlüssel 2, wird zur Wurzel des Baums und damit schwarz. Die beiden leeren Kindknoten sind, wie immer, ebenfalls schwarz.

Der zweite Knoten wird zum rechten Kind der Wurzel, da sein Schlüssel 9 größer ist als die 2 der Wurzel. Als neu eingefügter Knoten wird er rot. Er bleibt rot, da er keine rote Eigenschaft verletzt.

Als nächstes fügen wir den Schlüssel 3 ein. Unsere `treeInsert()`-Prozedur fügt ihn dort ein, wo `treeSearch()` ihn fände, nämlich im linken Kind der 9. Dieser neu interne, rote Knoten verletzt nun zusammen mit seinem Elternknoten die rote Eigenschaft. Um diese wieder herzustellen, schauen wir uns die Farbe des Geschwisterknotens des Elternknotens des neuen Knotens mit der 3 an. Dieser Knoten ist das externe Kind der 2 und schwarz. Damit handelt es sich um unseren Fall 1. Diesen behandeln wir, indem wir `restructure()` auf dem neuen Knoten 3 aufrufen und dann die neue Wurzel dieses Unterbaums schwarz und seine beiden Kinder rot färben. In diesem Fall führt `restructure()` eine doppelte Rotation durch. Unsere Animation zeigt diese als zwei einfache Rotationen. Die erste rotiert die 3 über die 9, und die zweite die 3 über die 2. Abschließend färben wir die 3 schwarz und seine beiden Kinder 2 und 9 rot.

Nun fügen wir den Schlüssel 4 ein. Wie zuvor führt dies zu einem Doppelrot. Nun aber ist der Geschwisterknoten des Elternknotens 9, nämlich die 2, rot. Damit handelt es sich um unserem Fall 2. Hier beseitigen wir das Doppelrot, indem wir jene beiden Knoten 2 und 9 schwarz färben. Deren gemeinsamen Elternknoten 3 würden wir im Allgemeinen nun rot färben, um die schwarze Tiefe dieses Unterbaums nicht zu erhöhen, aber da es sich um die Wurzel handelt, lassen wir ihn schwarz. Dadurch erhöht sich die schwarze Tiefe des gesamten Baums um 1, zum ersten Mal, seitdem wir den ersten Knoten eingefügt haben.

Anschließend fügen wir den Schlüssel 8 ein. Das resultierende Doppelrot beheben wir wieder gemäß Fall 1, da der Geschwisterknoten des Elternknotens des neu internen Knotens schwarz ist, nämlich das rechte Kind der 9. Das `restructure()` auf dem Knoten mit der 8 führt wiederum eine doppelte Rotation aus.

Die nächste Einfügung ist wieder unser Fall 2, da die 9 rot ist. Wir färben also die 4 und die 9 schwarz. Deren gemeinsamen Elternknoten 8 färben wir nun rot, da es sich nicht um die Wurzel handelt und wir so verhindern, dass die schwarze Tiefe des rechten Unterbaums der 3 erhöht und damit die Tiefeneigenschaft des Rot-Schwarz-Baums verletzt wird. Diese Rotfärbung führt hier zu keinem neuen Doppelrot, da der Elternknoten der umgefärbten 8 schwarz ist.

Nun fügen wir den Schlüssel 7 ein. Der Geschwisterknoten seines Elternknotens, das linke Kind der 4, ist schwarz, und damit haben wir Fall 1, den wir wieder mittels doppelter Rotation und anschlißender Umfärbung behandeln.

Als Letztes fügen wir noch den Schlüssel 5 ein. Hier wird es noch einmal interessant. Die Auflösung des Doppelrots nach Fall 2 mittels `restructure()` auf dem Knoten mit der 5 und anschließender Umfärbung führt nämlich zu einem neuen Doppelrot zwischen der 6 und der 8. Hier greift wieder Fall 1, denn der Knoten mit der 2 ist schwarz. `restructure()` auf dem Knoten mit der 6 rotiert die 6 über die 8 und anschließend über die 3, und macht sie damit zur neuen Wurzel unseres Rot-Schwarz-Baums.

Entfernen [Slide 59]

Video 7 beginnt hier.

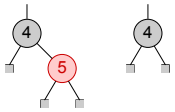
1. `treeRemove()`

   Siehe `treeRemove()`.

2. - War der entfernte interne Knoten rot, terminiere.

     Die schwarzen Tiefen und roten Eigenschaften bleiben unverändert.

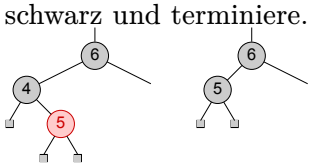
     (Im entsprechenden (2,4)-Baum: Schrumpfen eines 4- oder 3-Knotens)

   - War der entfernte interne Knoten schwarz, hatte er eine schwarze Höhe von 1.

     Ansonsten hätten seine Nachkommen unterschiedliche schwarze Höhen; schließlich hatte er ein externes, schwarzes Kind.

     – Hatte er genau ein rotes Kind mit zwei externen Kindern, färbe es schwarz und terminiere.

     (Im entsprechenden (2,4)-Baum: Schrumpfen eines 3-Knotens)

     – Hatte er zwei externe Kinder, ist die Tiefeneigenschaft verletzt; die schwarze Tiefe des angehobenen Blattes p ist um 1 reduziert. Markiere es ***doppelschwarz***.

     Hier unterscheiden wir 3 Fälle, je nach den Eigenschaften des Geschwisterknotens y von p.

     (Im entsprechenden (2,4)-Baum: Unterlauf nach Entfernen eines Elements aus einem 2-Knoten)

Erklärung

Das Entfernen eines Elements beginnt mit `treeRemove()`. Sie erinnern sich, dass diese Funktion das zu entfernende Element möglicherweise durch seinen Inorder-Vorgänger ersetzt und anschließend dessen Knoten entfernt, der mindestens ein externes Kind haben muss. `treeRemove()` liefert eine Referenz p auf denjenigen Knoten zurück, der an die Stelle des entfernten Knotens getreten ist.

War der entfernte interne Knoten rot, dann ist die Arbeit bereits getan, da dies keine der Eigenschaften eines Rot-Schwarz-Baums verletzen kann. Dies entspricht dem Schrumpfen eines 4- oder 3-Knotens im entsprechenden (2,4)-Baum.

War der entfernte interne Knoten schwarz, dann hatte er eine schwarze Höhe von 1. Da er ja mindestens ein externes, schwarzes Kind hatte, wäre andernfalls die Tiefeneigenschaft verletzt.

Es kann sein, dass sein anderes Kind ein roter interner Knoten war. Dieser ist durch die Entfernung seines Elternknotens an dessen Stelle gerückt. Damit nimmt er nun auch seine schwarze Farbe an, und die Tiefeneigenschaft bleibt erhalten.

Hatte der entfernte Knoten jedoch zwei externe Kinder, dann wird die Tiefe des angehobenen Blatts p um 1 reduziert, und verletzt damit die Tiefeneigenschaft. In diesem Fall färben wir p ***doppelschwarz***. Dies entspricht einem Unterlauf im entsprechenden (2,4)-Baum.

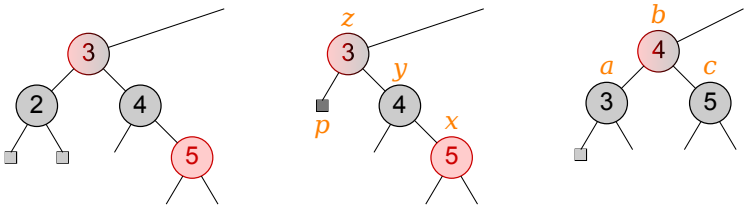
Um die Tiefeneigenschaft wiederherzustellen, müssen wir nun drei Fälle unterscheiden, je nach den Eigenschaften des Geschwisterknotens y des angehobenen Knotens p.

Entfernen Fall 1: y schwarz mit rotem Kind x [Slide 60]

1. `restructure(`x`)`
2. Färbe a und c schwarz, und gebe b die frühere Farbe von z.

   Der Wurzelknoten dieses Unterbaums ist entweder schwarz oder rot.

   Die schwarze Tiefe von p wurde um 1 erhöht; färbe ihn wieder schwarz.



(2,4)-Baum: Transfer; y hat ein rotes Kind, repräsentiert also einen 3- oder 4-Knoten.

Anmerkung

Der doppelschwarze Knoten p muss nicht unbedingt ein Blatt sein; siehe die beiden folgenden Fälle.

Übung:

- Zeichnen Sie korrespondiere (2,4)-Unterbäume für die Fälle, dass z rot bzw. schwarz ist.
- Warum funktioniert dies nicht ohne rotes Kind?

Erklärung

Im 1. Fall ist der Geschwisterknoten y des angehobenen Knotens p schwarz und hat mindestens ein rotes Kind. Das angehobene, doppelschwarze Blatt ist in diesem Beispiel dunkelgrau kenntlich gemacht.

Dieses Blatt repräsentiert den leeren Knoten im entsprechenden (2,4)-Baum. Sein Geschwisterknoten ist im Rot-Schwarz-Baum der Unterbaum mit Wurzel y und rotem Kind x. Diese Knoten entsprechen also einem 3-Knoten oder möglicherweise 4-Knoten im (2,4)-Baum. Wir reparieren also diesen Unterlauf mittels eines Transfers: Das Elternelement holen wir herunter in den leeren Knoten, und y wandert dafür in den Elternknoten. Dies erreichen wir durch ein `restructure(x)` und eine entsprechende Umfärbung: Die linken und rechten Knoten a und c färben wir schwarz, und b bekommt die frühere Farbe von z, schwarz oder rot, je nachdem ob der entsprechende Knoten im (2,4)-Baum ein 2-Knoten ist oder nicht.

Die schwarze Tiefe von p hat sich durch dieses Umstrukturieren wieder um 1 erhöht. Wir färben ihn also von doppelschwarz wieder auf schwarz.

Die schwarzen Tiefen der anderen drei Unterbäume ändern sich durch diese Umstrukturierung nicht. Auch ein Doppelrot kann nicht entstehen. Folglich sind nun die Eigenschaften eines Rot-Schwarz-Baums wieder hergestellt, und die Entfernungsoperation ist beendet.

Entsteht die hier gezeigte Situation unmittelbar durch Entfernen eines Knotens, dann müssen die hier gezeigten Blätter tatsächlich Blätter sein, denn gemäß unserer Prämisse hatte der entfernte Knoten zwei externe Kinder. Dieser Fall 1 kann jedoch auch auf andere Weise zustande kommen, so dass p nicht unbedingt ein Blatt ist, wie wir in Kürze sehen werden. Dies hat auf das Funktionieren dieser Prozedur jedoch keine Auswirkungen.

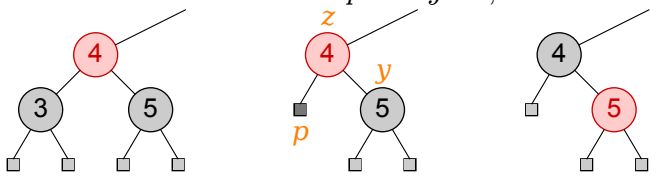
Entfernen Fall 2: y schwarz mit zwei schwarzen Kindern [Slide 61]

1. Färbe p schwarz und y rot.

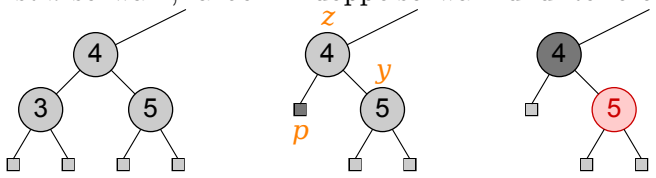
   Dies kann keine Rot-Verletzung zwischen y und seinen Kindern hervorrufen.

2. Restaurierung der Tiefeneigenschaft:

   - Ist der Elternknoten z von p und y rot, färbe ihn schwarz und terminiere.



   - Ist z schwarz, färbe ihn doppelschwarz und iteriere.



(2,4)-Baum: Fusion; y repräsentiert einen 2-Knoten.

Anmerkung

Die hier gezeigten Blätter können auch *interne* schwarze Knoten derselben schwarzen Höhe sein.

Übung:

- Zeichnen Sie korrespondiere (2,4)-Unterbäume für die Fälle, dass z rot bzw. schwarz ist.
- Warum funktioniert dies nicht mit rotem Kind?

Erklärung

Im 2. Fall ist der Geschwisterknoten y des angehobenen Knoten p schwarz und hat *kein* rotes Kind. Im entsprechenden (2,4)-Baum ist y also ein 2-Knoten, den wir nun mit dem unterbesetzten Knoten p fusionieren werden.

Dies erreichen wir dadurch, dass wir ein Element aus dem gemeinsamen Elternknoten herunterholen. Enthält dieser Elternknoten im (2,4)-Baum mindestens zwei Elemente, dann können wir dies ganz einfach tun. Im Rot-Schwarz-Baum entspricht dies einer Umfärbung des roten Elternknotens z auf schwarz. Damit haben wir die schwarze Tiefe des doppelschwarzen Knotens p um 1 erhöht und färben ihn wieder schwarz. Die Tiefen der beiden Unterbäume von y bleiben unberührt. Damit ist die Einfügung beendet.

Ist der Elternknoten z jedoch bereits schwarz, dann verursacht das Herunterholen seines Elements im (2,4)-Baum einen Unterlauf. Im Rot-Schwarz-Baum wird z daher doppelschwarz. Wir benennen ihn nun mit p und iterieren.

Wenn dieser Fall nicht unmittelbar durch das Entfernen eines Knotens eintritt, sondern durch Iteration nach einer solchen Entfernung, dann kann es sich statt der hier gezeigten Blätter auch um Unterbäume derselben schwarzen Höhe handeln.

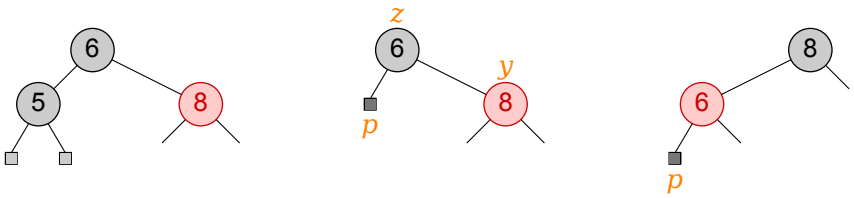
Entfernen Fall 3: y rot [Slide 62]

1. Sei z der (schwarze) Elternknoten von p und y.

   Rotiere y über z, und vertausche deren Farben.

   (2,4)-Baum: z und y repräsentieren einen 3-Knoten; die Rotation orientiert sie um.

2. Der Geschwisterknoten des doppelschwarzen p ist schwarz, also weiter mit Fall 1 (immer terminierend) oder Fall 2 (terminiert hier, da der Elternknoten von p nun rot ist).



Anmerkung

Die hier gezeigten Blätter können auch *interne* schwarze Knoten derselben schwarzen Höhe sein.

Übung: Warum funktioniert dies nicht mit schwarzem y?

Erklärung

Der einzige Fall, den wir nun noch abdecken müssen, ist ein roter Geschwisterknoten y. Im entsprechenden (2,4)-Baum bilden y und sein Elternknoten z einen 3-Knoten. Wie wir gesehen haben, gibt es für jeden 3-Knoten zwei verschiedene Varianten des entsprechenden Rot-Schwarz-Baums, je nachdem, welchen der beiden Schlüssel des 3-Knotens wir zum Elternknoten des anderen machen.

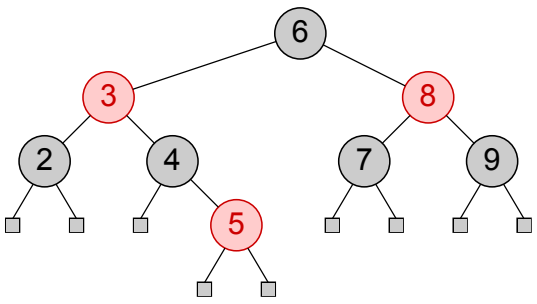
Hier drehen wir diese Eltern-Kind-Relation um, indem wir y über z rotieren und die Farben dieser beiden Knoten vertauschen.

Was hat dies nun an unserer verletzten Tiefeneigenschaft geändert? Gar nichts! Alle schwarzen Tiefen sind unverändert geblieben. Wir haben jedoch erreicht, dass dieser Fall 3 nicht mehr zutrifft, denn der Geschwisterknoten von p muss nun schwarz sein. Wie wir diese Situation auflösen, haben wir bereits besprochen.

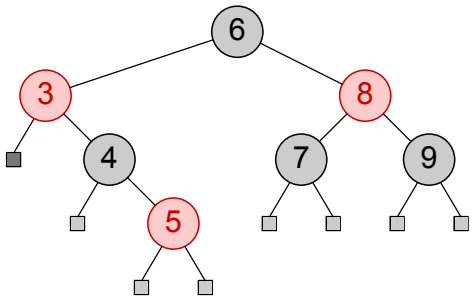
Hat der neue Geschwisterknoten von p ein rotes Kind, handelt es sich um Fall 1, der immer terminiert.

Hat dieser Knoten kein rotes Kind, tritt damit Fall 2 ein. Dieser führt zu weiteren Iterationen, falls der gemeinsame Elternknoten schwarz ist. Hier ist er jedoch rot. Also terminiert nach Fall 3 auch Fall 2.

Beispielsequenz von Entfernungen [Slide 63]

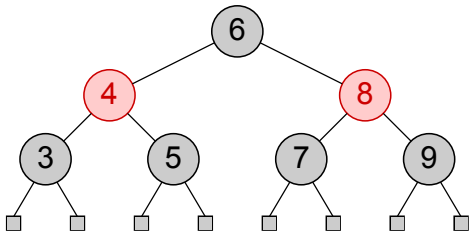


Beispielsequenz von Entfernungen [Slide 64]

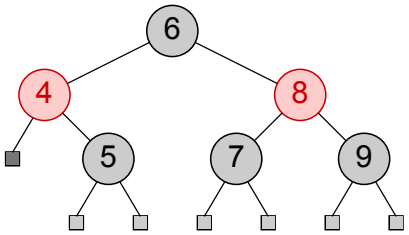


2 entfernt; p doppelschwarz; Entfernen Fall 1: y schwarz mit rotem Kind x

Beispielsequenz von Entfernungen [Slide 65]

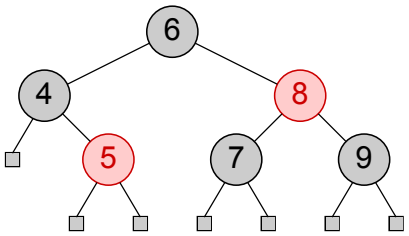


Beispielsequenz von Entfernungen [Slide 66]

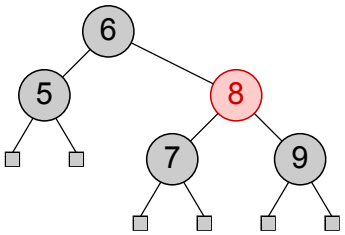


3 entfernt; p doppelschwarz; Entfernen Fall 2: y schwarz mit zwei schwarzen Kindern; z rot

Beispielsequenz von Entfernungen [Slide 67]

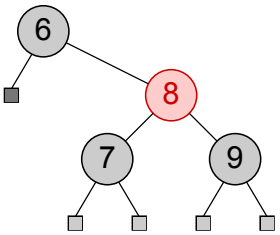


Beispielsequenz von Entfernungen [Slide 68]



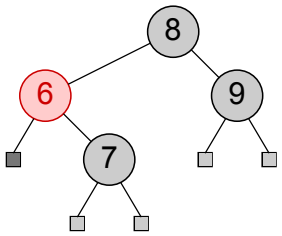
4 entfernt (genau ein rotes Kind mit zwei externen Kindern)

Beispielsequenz von Entfernungen [Slide 69]



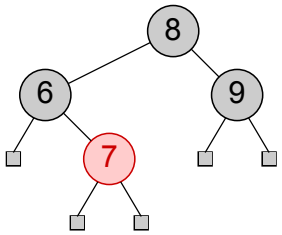
5 entfernt; p doppelschwarz; Entfernen Fall 3: y rot

Beispielsequenz von Entfernungen [Slide 70]



Weiter mit Entfernen Fall 2: y schwarz mit zwei schwarzen Kindern; z rot

Beispielsequenz von Entfernungen [Slide 71]



Animation

Illustrieren wir die verschiedenen Fälle, indem wir einige Schlüssel aus dem Baum unserer vorigen Animation entfernen.

Als erstes entfernen wir den Schlüssel 2. Dieser liegt in einem Knoten mit mindestens einem externen Kind. `treeRemove()` ersetzt diesen also durch eines seiner Kinder. Da der entfernte interne Knoten schwarz war und zwei externe Kinder hatte, reduziert sich hierdurch die schwarze Tiefe dieses Unterbaums, die Tiefeneigenschaft des Rot-Schwarz-Baums ist verletzt, und das angehobene Blatt wird doppelschwarz markiert.

Wie beheben wir dieses Doppelschwarz? Wir müssen uns den Geschwisterknoten des doppelschwarzen Knotens anschauen. Dies ist hier der Knoten mit der 4. Er ist schwarz und hat ein rotes Kind. Damit handelt es sich um Fall 1. Wir lösen ihn, indem wir `restructure()` auf diesem roten Kind aufrufen, um den doppelschwarzen Knoten eine Zeile tiefer zu verfrachten. Wir stellen sicher, dass sich damit auch seine *schwarze* Tiefe um eins erhöht, indem wir seinen Elternknoten schwarz färben, falls er es nicht bereits ist.

Die `restructure()`-Operation führt in diesem Fall eine einfache Rotation durch. Diese reduziert die Tiefe der 5 um eins. Um die schwarzen Tiefen in dessen Unterbaum nicht zu reduzieren, müssen wir diesen also ebenfalls schwarz umfärben. Die neue Wurzel dieses Unterbaums, also die 4, bekommt die Farbe rot der alten Wurzel vor der Restrukturierung, damit die schwarze Höhe dieses Unterbaums unverändert bleibt. Damit sind die Eigenschaften des Rot-Schwarz-Baums global wieder hergestellt.

Als nächstes entfernen wir den Schlüssel 3. Dies führt aus den gleichen Gründen wie zuvor zu einem Doppelschwarz. Nun ist der Geschwisterknoten 5 des doppelschwarzen Knotens schwarz mit zwei schwarzen Kindern. Dies ist also Fall 2. Hier stellen wir die Tiefeneigenschaft dieses Unterbaums wieder her, indem wir diesen Geschwisterknoten rot umfärben, und den doppelschwarzen Knoten schwarz. Damit haben wir allerdings die schwarze Höhe dieses Unterbaums um eins reduziert. Falls der gemeinsame Elternknoten

rot ist, können wir ihn schwarz färben und damit die Tiefeneigenschaft global wiederherstellen. Dies ist hier in der Tat der Fall. Wäre die 4 bereits schwarz, müssten wir sie doppelschwarz färben und nach oben iterieren.

Nun entfernen wir den Schlüssel 4. Er ist schwarz und hatte genau ein rotes Kind mit zwei externen Kindern. Damit können wir ihn einfach durch sein rotes Kind ersetzen und dieses schwarz färben, um die schwarzen Tiefen seiner Kinder trotz seiner Anhebung zu erhalten.

Die anschließende Entfernung des Schlüssels 5 führt wieder zu einem Doppelschwarz. Diesmal ist der Geschwisterknoten des doppelschwarzen Knotens rot. Es handelt sich also um Fall 3, und der gemeinsame Elternknoten muss schwarz sein. Wir rotieren nun den roten Geschwisterknoten über den schwarzen Elternknoten, und vertauschen deren Farben. Damit bleiben alle schwarzen Tiefen unverändert, und unser doppelschwarzer Knoten bleibt doppelschwarz. Wir haben allerdings erreicht, dass der neue Geschwisterknoten des doppelschwarzen Knotens schwarz sein muss, denn es handelt sich um ein ehemaliges Kind des vormals roten Geschwisterknotens. Wir haben nun also entweder Fall 1 oder Fall 2. In diesem Beispiel handelt es sich um Fall 2, denn dieser neue Geschwisterknoten hat zwei schwarze Kinder. Da der gemeinsame Elternknoten rot ist, können wir einfach die Farben des Eltern- und des Geschwisterknotens vertauschen. Damit erhöht sich die schwarze Tiefe unseres doppelschwarzen Knotens um eins, und wir können ihn wieder schwarz färben. Die schwarzen Tiefen des rechten Unterbaums der 6 bleiben dabei unverändert. Folglich haben wir nun sämtliche Eigenschaften des Rot-Schwarz-Baums global wieder hergestellt.

Laufzeiten der Methoden [Slide 72]

Asymptotisch identisch zum AVL-Baum.

Aber:

Proposition: Ein Einfügen erfordert $O(\log n)$ Umfärbungen und höchstens ein *trinode restructuring*.

Proposition: Ein Entfernen erfordert $O(\log n)$ Umfärbungen und höchstens zwei umstrukturierende Operationen.

Beweise: (Übung)

Erklärung

Die asymptotischen Laufzeiten der Methoden des abstrakten Datentyps Sortierte Zuordnungstabelle unterscheiden sich nicht zwischen dem AVL-Baum und dem Rot-Schwarz-Baum. Nichtsdestotrotz sind die Einfüge- und Entfernungsoperationen des Rot-Schwarz-Baums in der Praxis dem AVL-Baum überlegen, denn sie erfordern im Gegensatz zum Letzteren nur eine *konstante* Anzahl umstrukturierender Operationen. Lediglich Umfärbungen können logarithmisch oft auftreten. Diese verursachen jedoch erheblich weniger Speicherzugriffe als Umstrukturierungen.

Daher sind Rot-Schwarz-Bäume insbesondere in Echtzeit-Umgebungen und Betriebssystemen sehr beliebt. Auch der Linux-Kernel macht von ihnen intensiv Gebrauch.

Ein kleiner Preis ist jedoch dafür zu bezahlen: Dieser Laufzeitvorteil der Änderungsoperationen wird durch ein Balance-Kriterium erkauft, das weniger streng ist als das Balance-Kriterium des AVL-Baums. AVL-Bäume haben daher im Schnitt eine leicht geringere Höhe als Rot-Schwarz-Bäume, was sich in einem kleinen Geschwindigkeitsvorteil bei den Abfrageoperationen niederschlägt.

Die Wahl der Datenstruktur hängt also wieder einmal davon ab, welche Operationen vorrangig effizient unterstützt werden sollen.

Quiz [Slide 73]

Jeder AVL-Baum ist ein Rot-Schwarz-Baum. Dies kann man deutlich machen, indem man jeden seiner Knoten auf geeignete Weise rot bzw. schwarz färbt.

A: ja

B: nein

D: weiß nicht

ziuQ [Slide 74]

Jeder Rot-Schwarz-Baum ist ein AVL-Baum.

A: ja

B: nein

D: weiß nicht

7 Zusammenfassung

Zusammenfassung [Slide 75]

- ADT: Sortierte Zuordnungstabelle
- Binäre Suchbäume und Rotationen
- AVL-Bäume
- B-Bäume und (2,4)-Bäume als Motivator für Rot-Schwarz-Bäume
- Rot-Schwarz-Bäume

Bibliographie [Slide 76]

Goodrich, Michael, Roberto Tamassia und Michael Goldwasser (Aug. 2014). *Data Structures and Algorithms in Java*. Wiley.