Splaytrees und Optimale Suchbäume

Theorem

Gegeben sei ein Suchbaum T für n Schlüssel. Eine bestimmte Folge von m Suchoperationen in T benötige Zeit t.
Wenn wir die n Schlüssel in einen Splay-Baum einfügen und dann dieselbe Folge von Suchoperatioen ausführen, dauert dies $O(n^2 + t)$.

Bedeutung:

Asymptotisch verhalten sich Splay-Bäume ebensogut wie optimale Suchbäume.

Der Splay-Baum benötigt aber Zeit, um die Zugriffshäufigkeiten zu „lernen“.

Splaytrees und Optimale Suchbäume

Theorem

Gegeben sei ein Suchbaum T für n Schlüssel. Eine bestimmte Folge von m Suchoperationen in T benötige Zeit t.
Wenn wir die n Schlüssel in einen Splay-Baum einfügen und dann dieselbe Folge von Suchoperatioen ausführen, dauert dies $O(n^2 + t)$.

Bedeutung:

Asymptotisch verhalten sich Splay-Bäume ebensogut wie optimale Suchbäume.

Der Splay-Baum benötigt aber Zeit, um die Zugriffshäufigkeiten zu „lernen“.

Beweis.

Sei $d(k)$ die Tiefe des Knotens mit Schlüssel k in T und d die Gesamttiefe von T. Wir definieren $g(k) = 3^{d-d(k)}$ als Gewichtsfunktion.

Die amortisierten Kosten einer Suche nach k sind:

$$O\left(\log\left(\frac{\bar{g}(w)}{\bar{g}(k)}\right)\right)$$

Es gilt $\bar{g}(w) \leq \sum_{i=1}^{n} 3^{d-d(k_i)} \leq \sum_{j=0}^{d} 2^j 3^{d-j} \leq 3^{d+1}$ und $\bar{g}(k) \geq g(k) = 3^{d-d(k)}$.

Die Kosten sind daher höchstens $O(\log(3^{d+1}/3^{d-d(k)}) = O(d(k))$.

Die Suche in T benötigt aber $\Omega(d(k))$ Zeit.

Das Aufbauen des Splaytrees geht in $O(n^2)$. □

Beweis.

Sei $d(k)$ die Tiefe des Knotens mit Schlüssel k in T und d die Gesamttiefe von T. Wir definieren $g(k) = 3^{d-d(k)}$ als Gewichtsfunktion.

Die amortisierten Kosten einer Suche nach k sind:

$$O\left(\log\left(\frac{\bar{g}(w)}{\bar{g}(k)}\right)\right)$$

Es gilt $\bar{g}(w) \leq \sum_{i=1}^{n} 3^{d-d(k_i)} \leq \sum_{j=0}^{d} 2^j 3^{d-j} \leq 3^{d+1}$ und $\bar{g}(k) \geq g(k) = 3^{d-d(k)}$.

Die Kosten sind daher höchstens $O(\log(3^{d+1}/3^{d-d(k)}) = O(d(k))$.

Die Suche in T benötigt aber $\Omega(d(k))$ Zeit.

Das Aufbauen des Splaytrees geht in $O(n^2)$. □

Beweis.

Sei $d(k)$ die Tiefe des Knotens mit Schlüssel k in T und d die Gesamttiefe von T. Wir definieren $g(k) = 3^{d-d(k)}$ als Gewichtsfunktion.

Die amortisierten Kosten einer Suche nach k sind:

$$O\left(\log\left(\frac{\bar{g}(w)}{\bar{g}(k)}\right)\right)$$

Es gilt $\bar{g}(w) \leq \sum_{i=1}^{n} 3^{d-d(k_i)} \leq \sum_{j=0}^{d} 2^j 3^{d-j} \leq 3^{d+1}$ und $\bar{g}(k) \geq g(k) = 3^{d-d(k)}$.

Die Kosten sind daher höchstens $O(\log(3^{d+1}/3^{d-d(k)}) = O(d(k))$.

Die Suche in T benötigt aber $\Omega(d(k))$ Zeit.

Das Aufbauen des Splaytrees geht in $O(n^2)$. □

Java

```
private void splay(Searchtreenode⟨K, D⟩ t) {
  while(t.parent ≠ null) {
    if(t.parent.parent == null) {
      if(t == t.parent.left) t.parent.rotateright();   // Zig
      else t.parent.rotateleft(); }  // Zag
    else if(t == t.parent.left && t.parent == t.parent.parent.left) {
      t.parent.parent.rotateright();   // Zig-zig
      t.parent.rotateright(); }
    else if(t == t.parent.left && t.parent == t.parent.parent.right) {
      t.parent.rotateright();   // Zig-zag
      t.parent.rotateleft(); }
    else if(t == t.parent.right && t.parent == t.parent.parent.right) {
      t.parent.parent.rotateleft();   // Zag-zag
      t.parent.rotateleft(); }
    else if(t == t.parent.right && t.parent == t.parent.parent.left) {
      t.parent.rotateleft();   // Zag-zig
      t.parent.rotateright(); }
  }
  root = t;
}
```

Java

```
public boolean containsKey(K k) {
  if(root == null) return false;
  Searchtreenode⟨K, D⟩ n = root, last = root;
  int c;
  while(n ≠ null) {
    last = n;
    c = k.compareTo(n.key);
    if(c < 0) n = n.left;
    else if(c > 0) n = n.right;
    else { splay(n); return true; }
  }
  splay(last); return false;
}
```

Java

```
public void insert(K k, D d) {
  super.insert(k, d);
  containsKey(k);
}
```

Java

```
public D find(K k) {
  containsKey(k);
  if(root ≠ null && root.key.equals(k)) return root.data;
  return null;
}
```

Java

```
public void delete(K k) {
  if(!containsKey(k)) return;
  if(root.left ≠ null) {
    Searchtreenode⟨K, D⟩ max = root.left;
    while(max.right ≠ null) max = max.right;
    splay(max);
  }
  super.delete(k);
}
```

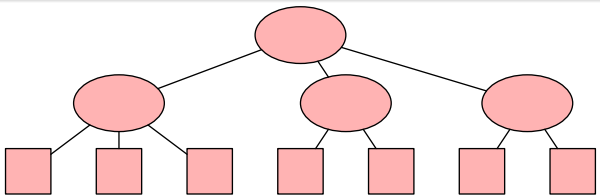
(a, b)-Bäume

Es sei $a \geq 2$ und $b \geq 2a - 1$.

Definition

Ein (a, b)-Baum ist ein Baum mit folgenden Eigenschaften:

1. Jeder Knoten hat höchstens b Kinder.
2. Jeder innere Knoten außer der Wurzel hat mindestens a Kinder.
3. Alle Blätter haben die gleiche Tiefe.



Ein $(2, 3)$-Baum.

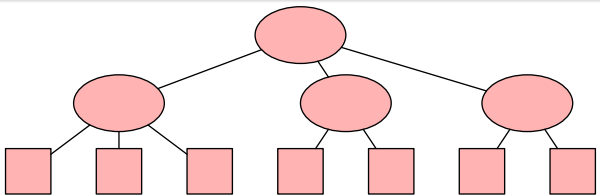
(a, b)-Bäume

Es sei $a \geq 2$ und $b \geq 2a - 1$.

Definition

Ein (a, b)-Baum ist ein Baum mit folgenden Eigenschaften:

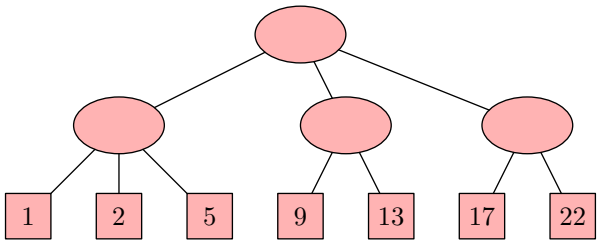
1. Jeder Knoten hat höchstens b Kinder.
2. Jeder innere Knoten außer der Wurzel hat mindestens a Kinder.
3. Alle Blätter haben die gleiche Tiefe.



Ein $(2, 3)$-Baum.

(a, b)-Bäume als assoziatives Array

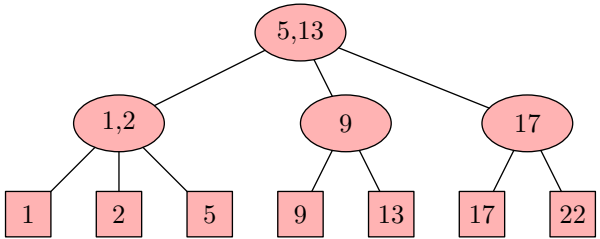
Wir speichern Schlüssel und Datenelemente nur in den Blättern:



Die Schlüssel sind von links nach rechts geordnet.

(a, b)-Bäume als assoziatives Array

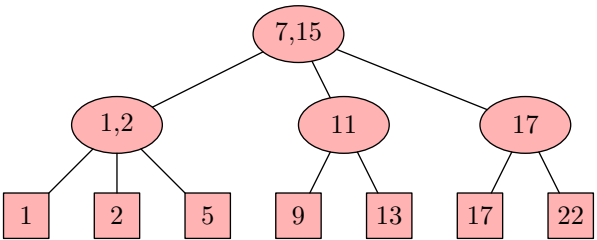
Als Suchhilfe enthält ein innerer Knoten mit m Kindern genau $m - 1$ Schlüssel.



Jetzt kann effizient nach einem Element gesucht werden.

(a, b)-Bäume als assoziatives Array

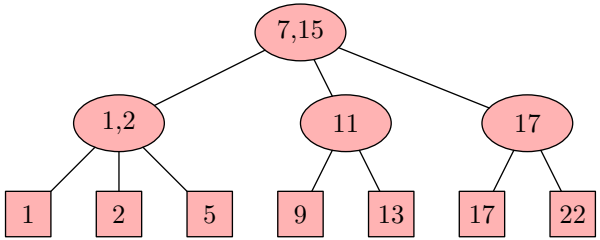
Wir bestehen nicht darauf, daß die Hilfsschlüssel in inneren Knoten in den Blättern vorkommen:



$\rightarrow$ etwas flexibler

(a, b)-Bäume – Einfügen

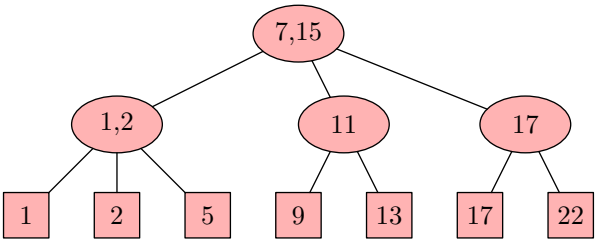
Die Blätter enthalten die Schlüssel in aufsteigender Reihenfolge.



- Neues Blatt an richtiger Stelle einfügen
- Problem, falls Elternknoten mehr als b Kinder hat
- $\rightarrow$ in zwei Knoten mit $\lfloor (b+1)/2) \rfloor$ und $\lceil (b+1)/2 \rceil$ Kindern teilen
- Jetzt kann Vater überfüllt sein etc.

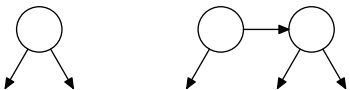
(a, b)-Bäume – Löschen

Die Blätter enthalten die Schlüssel in aufsteigender Reihenfolge.



- Blatt mit Schlüssel entfernen
- Problem, falls Elternknoten weniger als a Kinder hat
- $\rightarrow$ mit einem Geschwisterknoten vereinigen
- Dieser muß vielleicht wieder geteilt werden
- Der nächste Elternknoten kann nun wieder unterbelegt sein

(a, b)-Bäume als assoziatives Array

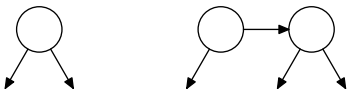


- $(2, 3)$-Bäume sind als assoziatives Array geeignet
- Löschen, Suchen, Einfügen in $O(\log n)$
- Welche anderen (a, b) sind interessant?

Bei großen Datenmengen, die nicht mehr im Hauptspeicher Platz haben, sind die bisher betrachteten Datenstrukturen nicht gut geeignet.

Verwende $(m, 2m)$-Bäume mit großem m

(a, b)-Bäume als assoziatives Array

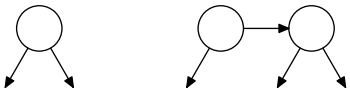


- $(2, 3)$-Bäume sind als assoziatives Array geeignet
- Löschen, Suchen, Einfügen in $O(\log n)$
- Welche anderen (a, b) sind interessant?

Bei großen Datenmengen, die nicht mehr im Hauptspeicher Platz haben, sind die bisher betrachteten Datenstrukturen nicht gut geeignet.

Verwende $(m, 2m)$-Bäume mit großem m

(a, b)-Bäume als assoziatives Array



- $(2, 3)$-Bäume sind als assoziatives Array geeignet
- Löschen, Suchen, Einfügen in $O(\log n)$
- Welche anderen (a, b) sind interessant?

Bei großen Datenmengen, die nicht mehr im Hauptspeicher Platz haben, sind die bisher betrachteten Datenstrukturen nicht gut geeignet.

Verwende $(m, 2m)$-Bäume mit großem m

(2,3)-Bäume: Beispiel

(2,4)-Bäume: Beispiel

(3,6)-Bäume: Beispiel

B-Bäume und Datenbanken

Ein B-Baum ist ein $(m, 2m)$-Baum.

Wir wählen m so groß, daß ein Knoten soviel Platz wie eine Seite im Hintergrundspeicher benötigt (z.B. 4096 Byte).

Blätter eines Elternknotens gemeinsam speichern.

Zugriffszeit:

Nur $O(\log_m(n))$ Zugriffe auf den Hintergrundspeicher.

Wurzel kann immer im RAM gehalten werden.

Falls $m \approx 500$, dann enthält ein B-Baum der Höhe 3 bereits mindestens $500 \cdot 500 \cdot 500 = 125\,000\,000$ Schlüssel und Datenelemente.

Jede Suche greift auf nur zwei Seiten zu!

B-Bäume und Datenbanken

Ein B-Baum ist ein $(m, 2m)$-Baum.

Wir wählen m so groß, daß ein Knoten soviel Platz wie eine Seite im Hintergrundspeicher benötigt (z.B. 4096 Byte).

Blätter eines Elternknotens gemeinsam speichern.

Zugriffszeit:

Nur $O(\log_m(n))$ Zugriffe auf den Hintergrundspeicher.

Wurzel kann immer im RAM gehalten werden.

Falls $m \approx 500$, dann enthält ein B-Baum der Höhe 3 bereits mindestens $500 \cdot 500 \cdot 500 = 125\,000\,000$ Schlüssel und Datenelemente.

Jede Suche greift auf nur zwei Seiten zu!

B-Bäume und Datenbanken

Ein B-Baum ist ein $(m, 2m)$-Baum.

Wir wählen m so groß, daß ein Knoten soviel Platz wie eine Seite im Hintergrundspeicher benötigt (z.B. 4096 Byte).

Blätter eines Elternknotens gemeinsam speichern.

Zugriffszeit:

Nur $O(\log_m(n))$ Zugriffe auf den Hintergrundspeicher.

Wurzel kann immer im RAM gehalten werden.

Falls $m \approx 500$, dann enthält ein B-Baum der Höhe 3 bereits mindestens $500 \cdot 500 \cdot 500 = 125\,000\,000$ Schlüssel und Datenelemente.

Jede Suche greift auf nur zwei Seiten zu!

B-Bäume und Datenbanken

Ein B-Baum ist ein $(m, 2m)$-Baum.

Wir wählen m so groß, daß ein Knoten soviel Platz wie eine Seite im Hintergrundspeicher benötigt (z.B. 4096 Byte).

Blätter eines Elternknotens gemeinsam speichern.

Zugriffszeit:

Nur $O(\log_m(n))$ Zugriffe auf den Hintergrundspeicher.

Wurzel kann immer im RAM gehalten werden.

Falls $m \approx 500$, dann enthält ein B-Baum der Höhe 3 bereits mindestens $500 \cdot 500 \cdot 500 = 125\,000\,000$ Schlüssel und Datenelemente.

Jede Suche greift auf nur zwei Seiten zu!